

Bloque 1 – Representación de Conocimiento y Búsqueda

Tema 2: Resolución de problemas mediante búsqueda heurística

Bloque 1, Tema 2 – Búsqueda heurística

1. Búsqueda heurística
2. Búsqueda voraz
3. Búsqueda A*
4. Diseño de funciones heurísticas
 - 4.1 Heurísticas para el problema del 8-puzzle
 - 4.2 Heurísticas para el problema del viajante de comercio
5. Evaluación de funciones heurísticas.

Bibliografía

- S. Russell, P. Norvig. **Artificial Intelligence. A modern approach**. Prentice Hall, 4th edición, 2022 (Capítulo 3) <http://aima.cs.berkeley.edu/>
- S. Russell, P. Norvig. **Artificial Intelligence. A modern approach**. Prentice Hall, 3rd edición, 2010 (Capítulo 3)
- S. Russell, P. Norvig. **Inteligencia artificial . Una aproximación moderna**. Prentice Hall, 2^a edición, 2004 (Capítulos 3 y 4) <http://aima.cs.berkeley.edu/2nd-ed/>

1. Búsqueda heurística

Anchura, Profundidad, Coste Uniforme, Profundización Iterativa (ID) expanden nodos siguiendo siempre un criterio sistemático de expansión

Anchura $\rightarrow f(n)=\text{nivel}(n)$ ($f(n)=g(n)$ si los costes de las acciones son todos iguales)

Profundidad $\rightarrow f(n)= -\text{nivel}(n)$

Coste Uniforme $\rightarrow f(n)=g(n)$

Profundización Iterativa \rightarrow combinación de anchura y profundidad

Estas estrategias no aprovechan la información del problema en la búsqueda

1. Búsqueda heurística

Búsqueda heurística o informada: utiliza conocimiento específico del problema para guiar la búsqueda.

Puede encontrar soluciones más eficientemente que una búsqueda no informada. Es especialmente útil en problemas complejos de explosión combinatoria (p. ej.: problema de viajante).

Función heurística: *Simplificación o conjetura que reduce o limita la búsqueda de soluciones en dominios complejos o poco conocidos.*

1. Búsqueda heurística

Búsqueda heurística o informada: utiliza conocimiento específico del problema para guiar la búsqueda.

Puede encontrar soluciones más eficientemente que una búsqueda no informada. Es especialmente útil en problemas complejos de explosión combinatoria (p. ej.: problema de viajante).

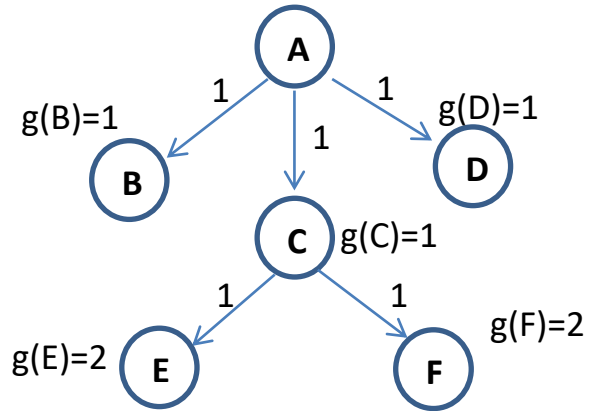
Función heurística: *Simplificación o conjetura que reduce o limita la búsqueda de soluciones en dominios complejos o poco conocidos.*

Aproximación general búsqueda **primero-el-mejor**:

- Al igual que en la búsqueda no informada, se utiliza una función de evaluación $f(n)$ para determinar el orden de expansión de los nodos
- En la búsqueda primer-el-mejor, **$f(n)$ es una estimación de coste**, de modo que el nodo con el mínimo valor de $f(n)$ se extrae primero de la cola de prioridades.
- Dos casos especiales de búsqueda primero el mejor: *búsqueda voraz* y *búsqueda A^** .

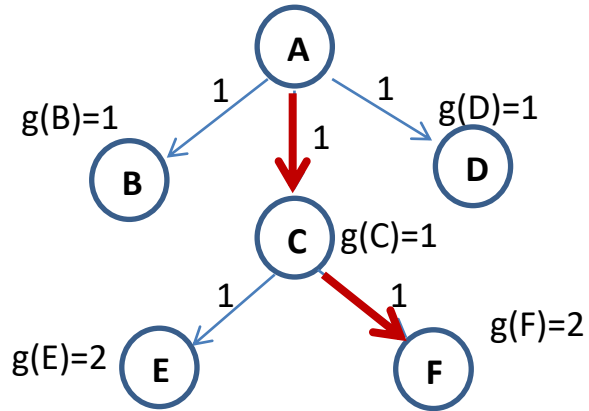
1. Búsqueda heurística

1. Búsqueda heurística



$g(n)$: función coste (el coste de un camino de A a n)

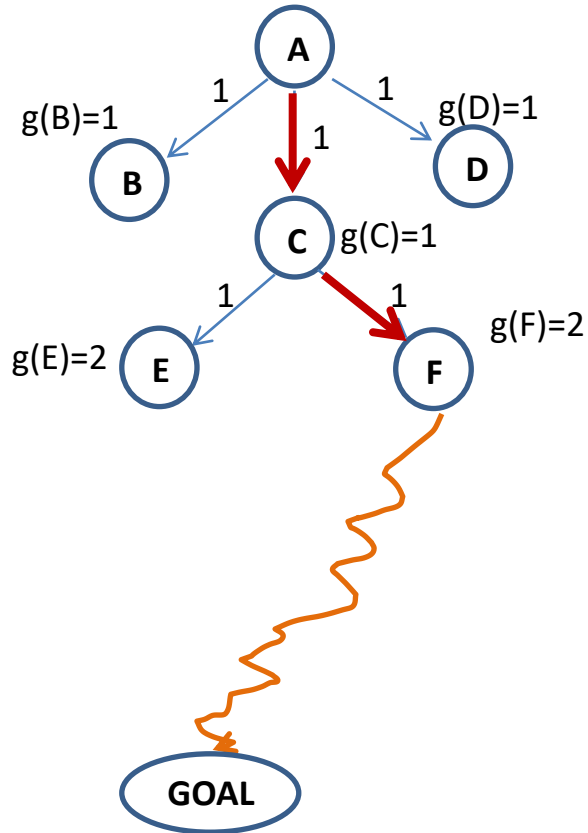
1. Búsqueda heurística



$g(n)$: función coste (el coste de un camino de A a n)

$g(F) = 2$ porque todos los operadores tienen coste 1

1. Búsqueda heurística

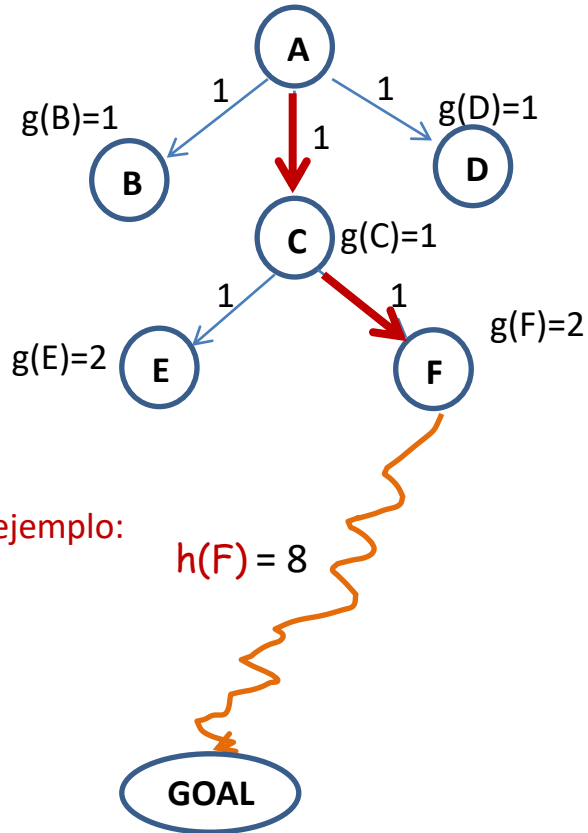


$g(n)$: función coste (el coste de un camino de A a n)

$g(F) = 2$ porque todos los operadores tienen coste 1

$h(n)$: función heurística → función que **estima** el coste de un nodo **n** al nodo objetivo (GOAL)

1. Búsqueda heurística

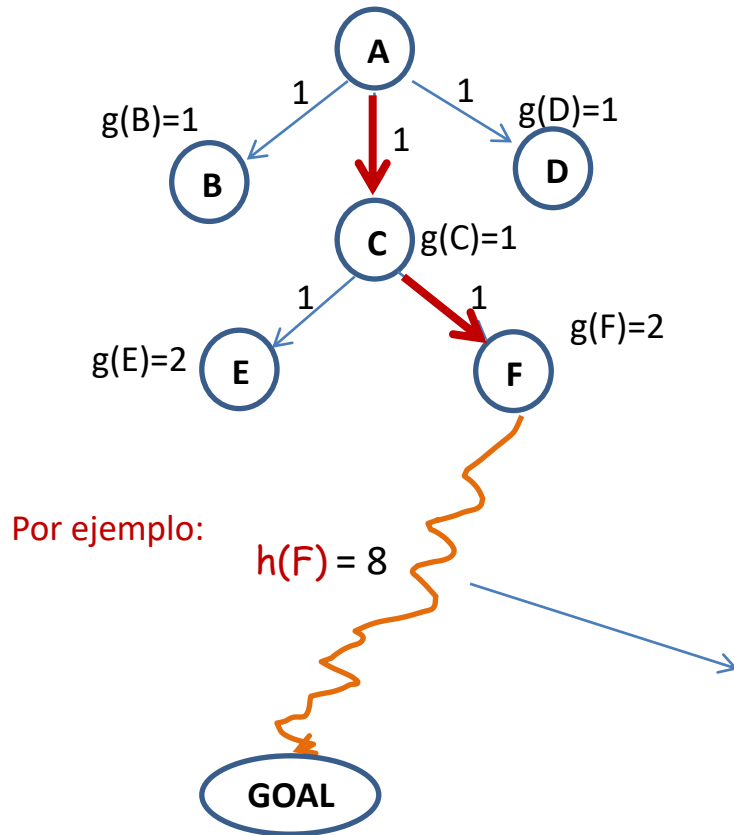


$g(n)$: función coste (el coste de un camino de A a n)

$g(F) = 2$ porque todos los operadores tienen coste 1

$h(n)$: función heurística → función que **estima** el coste de un nodo n al nodo objetivo (GOAL)

1. Búsqueda heurística



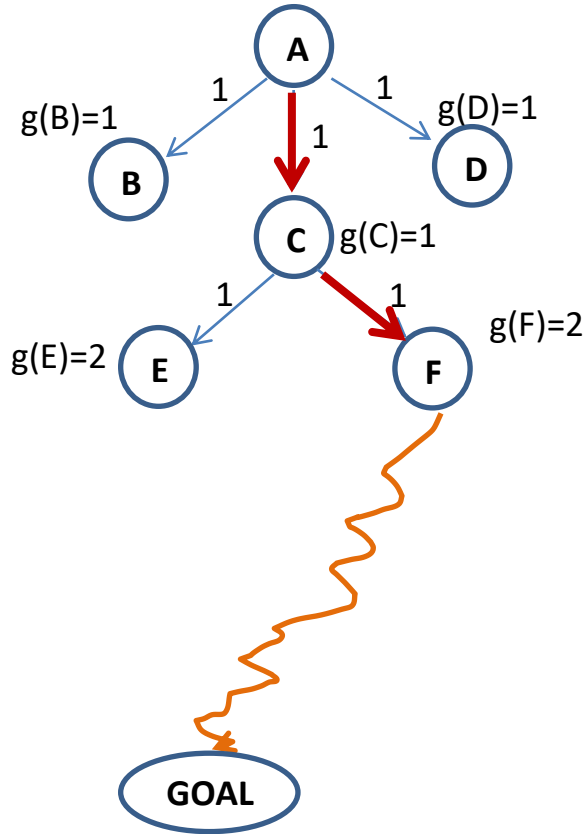
$g(n)$: función coste (el coste de un camino de A a n)

$g(F) = 2$ porque todos los operadores tienen coste 1

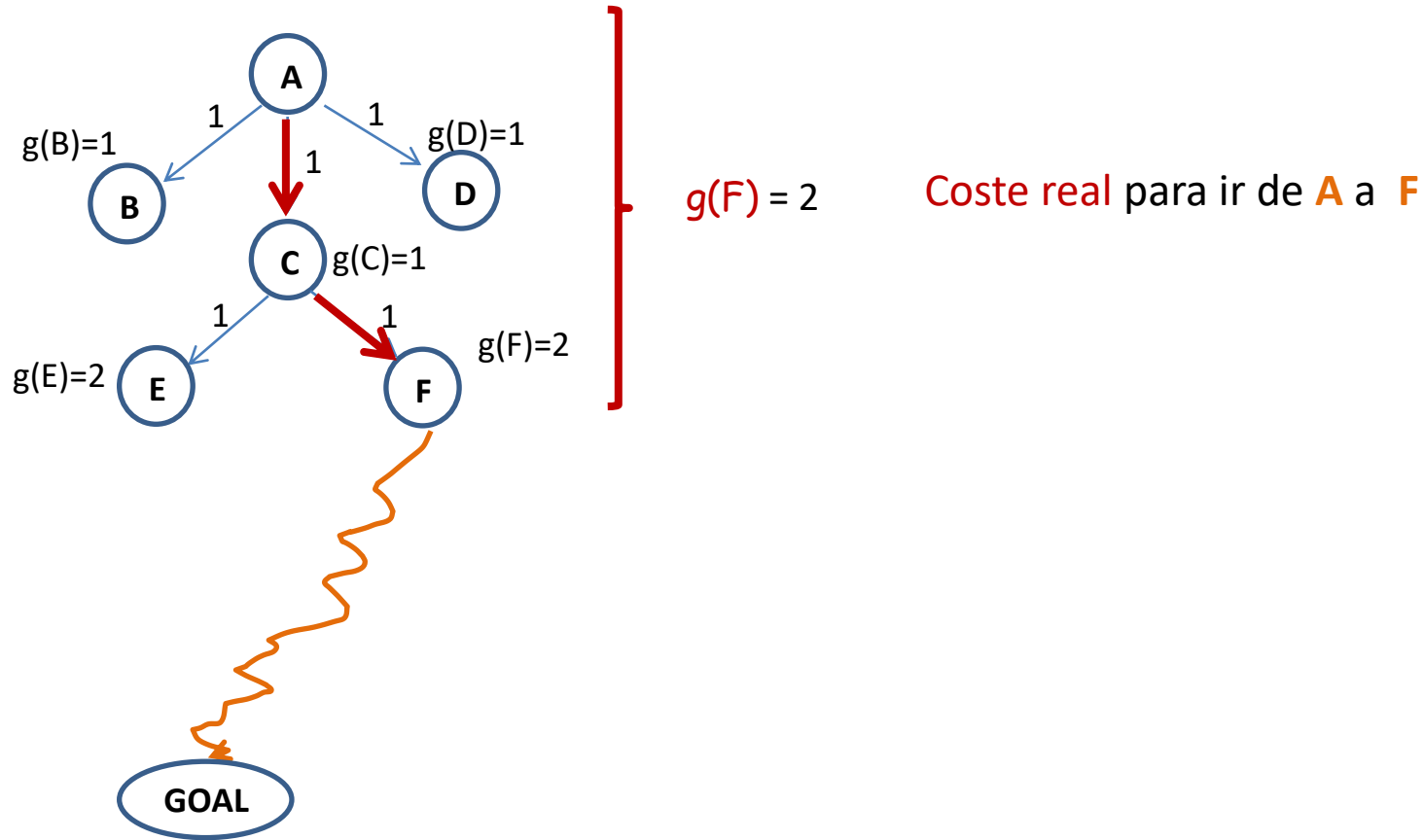
$h(n)$: función heurística → función que **estima** el coste de un nodo **n** al nodo objetivo (GOAL)

Significa que hay una estimación de que el coste del camino óptimo desde **F** hasta **GOAL** es 8 (8 movimientos u operadores porque todos los operadores tienen el mismo coste = 1)

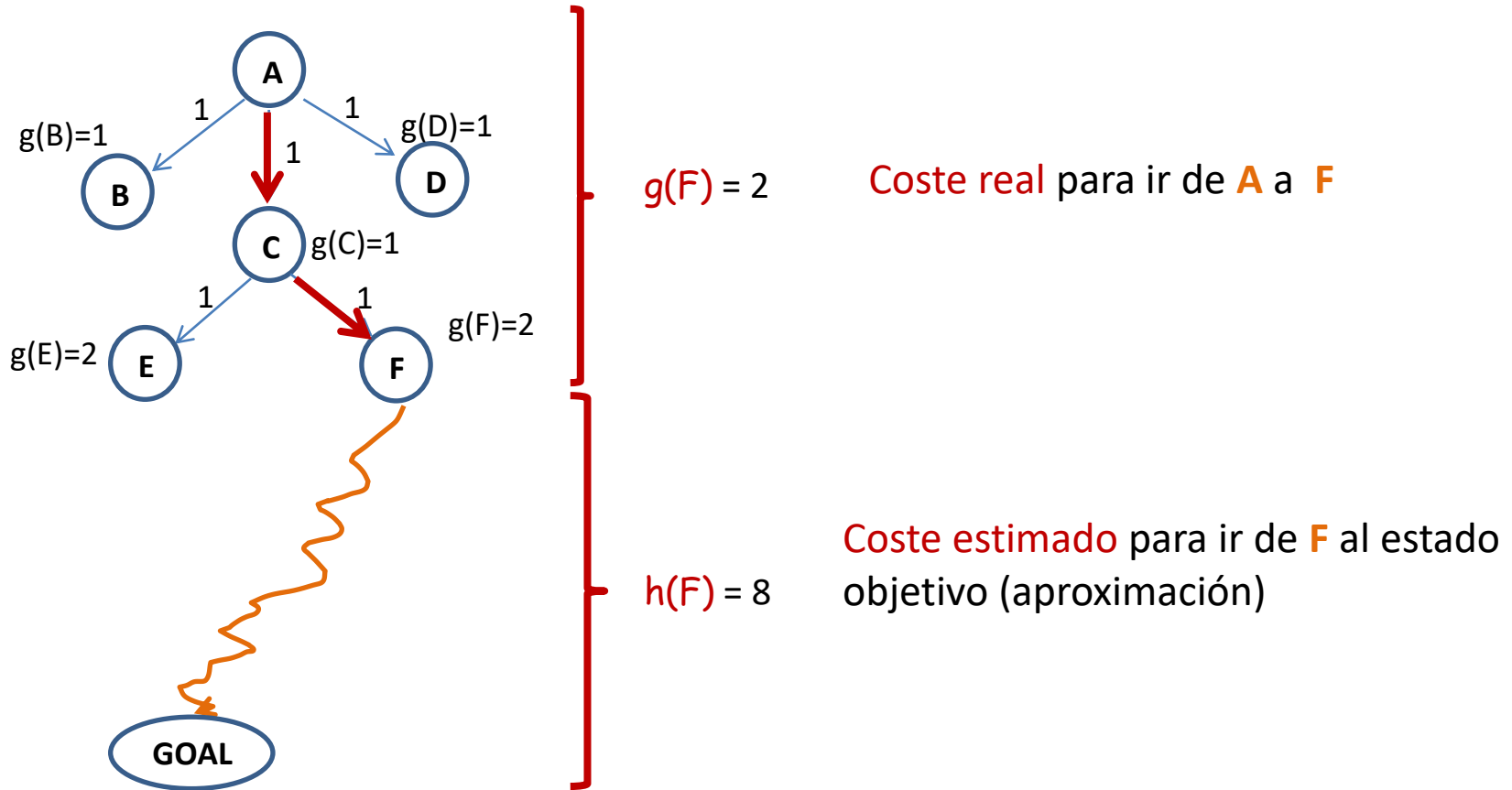
1. Búsqueda heurística



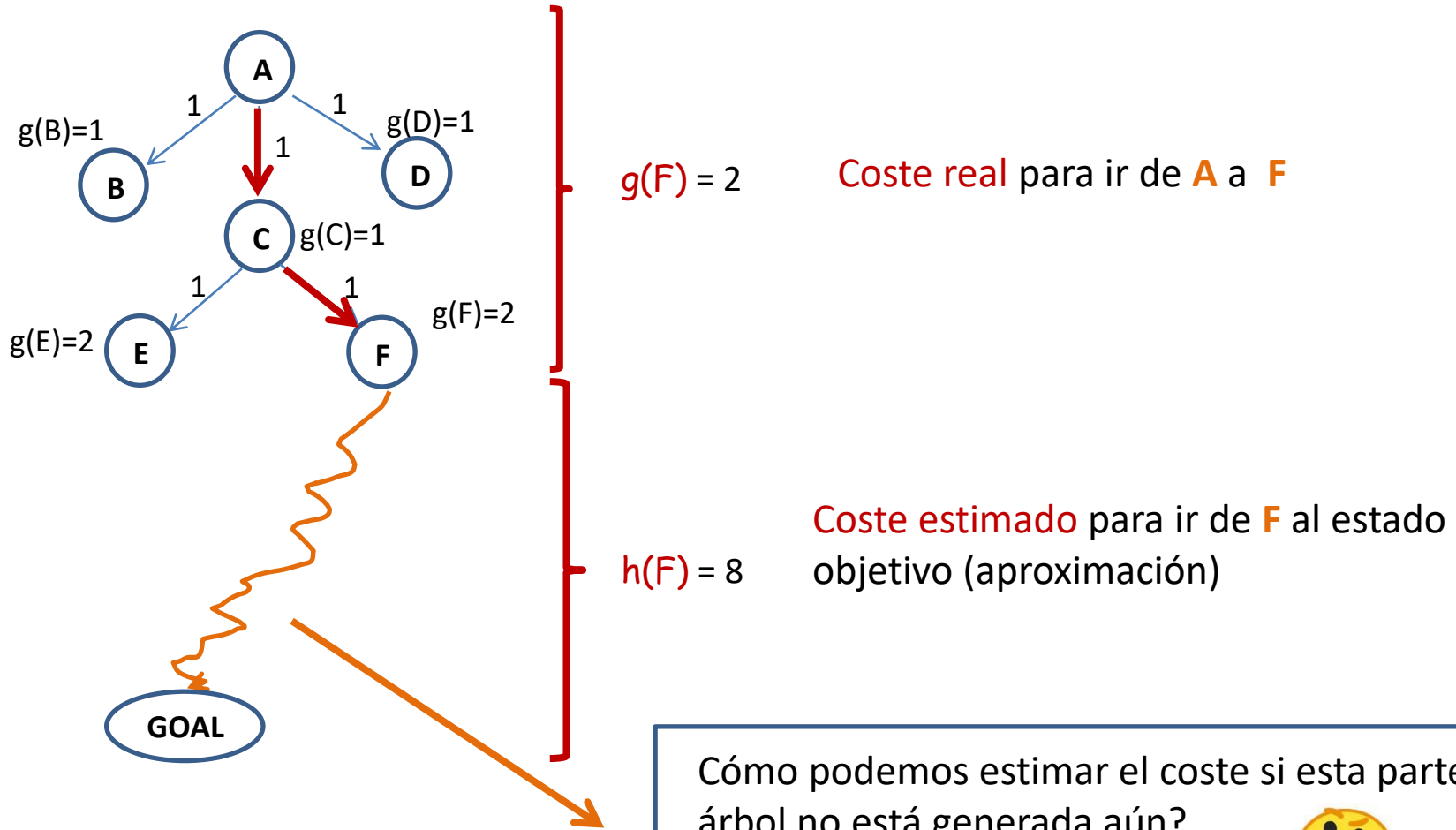
1. Búsqueda heurística



1. Búsqueda heurística



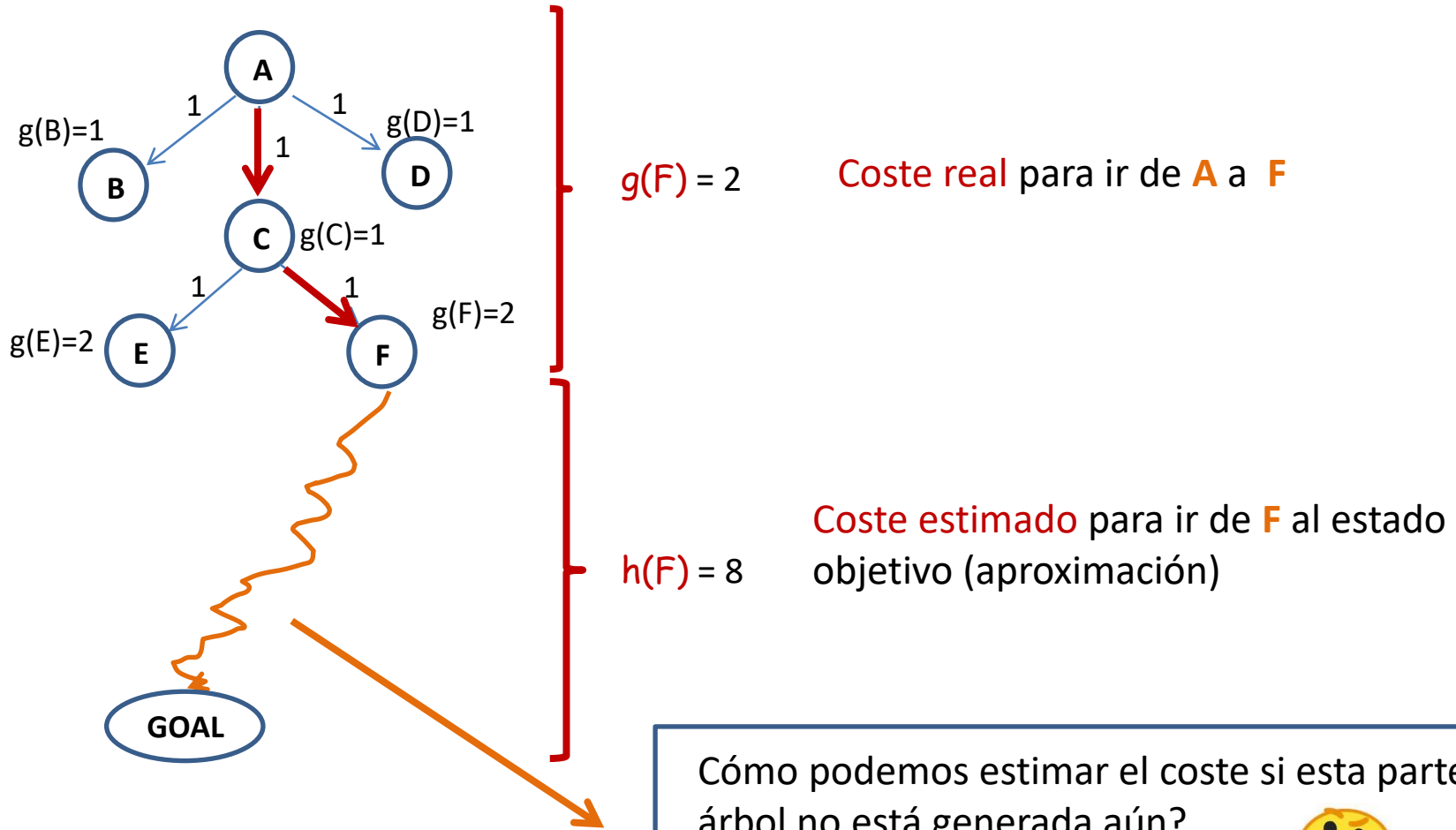
1. Búsqueda heurística



Cómo podemos estimar el coste si esta parte del árbol no está generada aún?

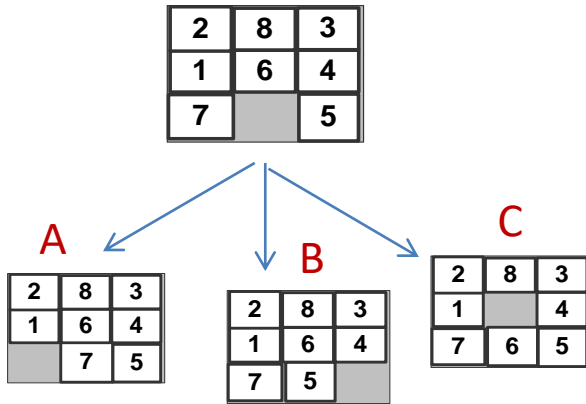


1. Búsqueda heurística



Explotando información específica del problema

1. Búsqueda heurística



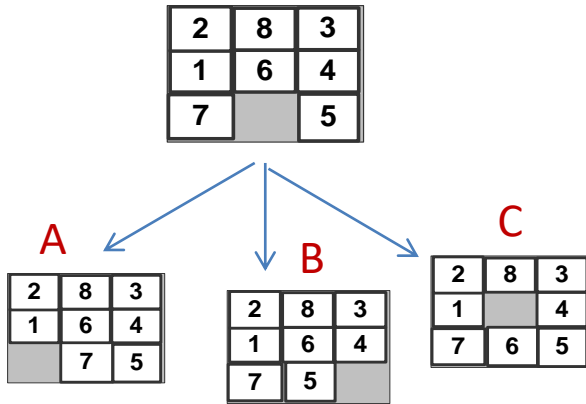
GOAL

1	2	3
8		4
7	6	5

1. Búsqueda heurística

Analizamos el nodo **C**

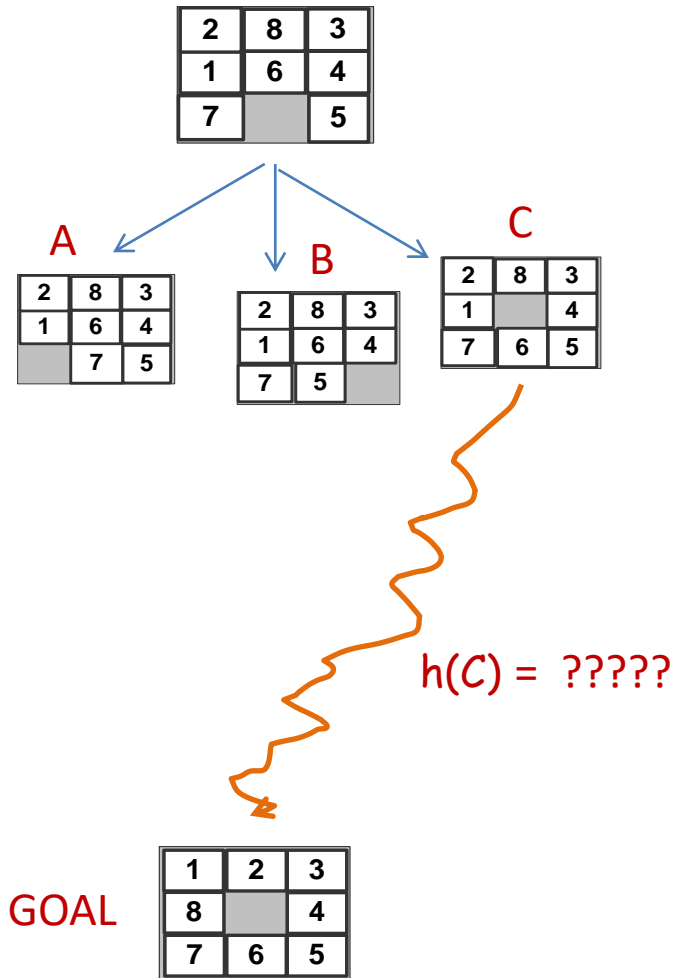
$g(C) = 1$ → Un movimiento desde el estado inicial con coste = 1



GOAL

1	2	3
8		4
7	6	5

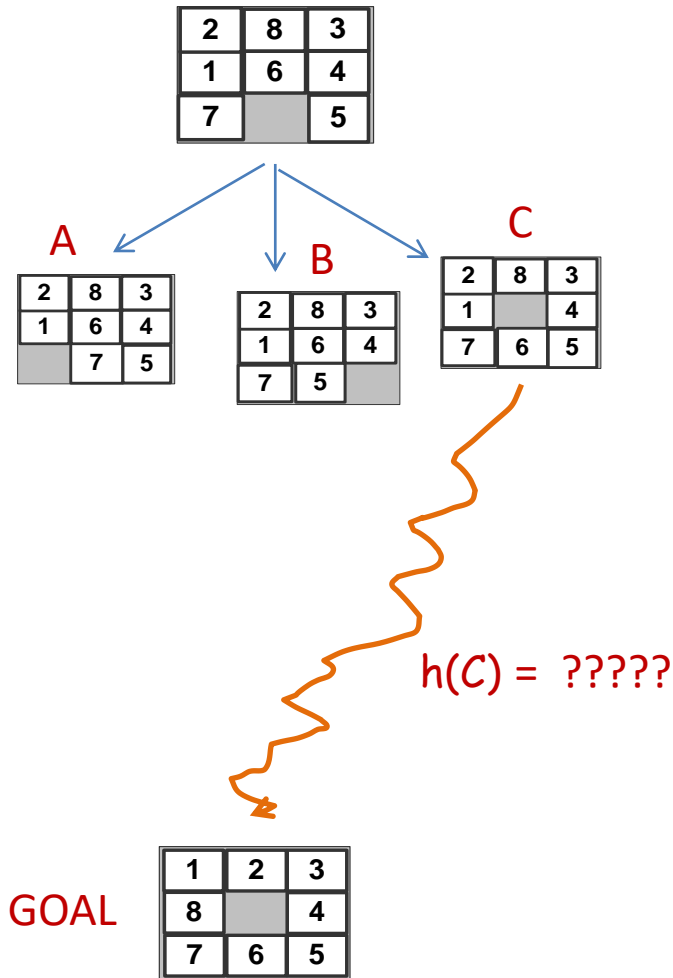
1. Búsqueda heurística



Analizamos el nodo C

$g(C) = 1$ → Un movimiento desde el estado inicial con coste = 1

1. Búsqueda heurística

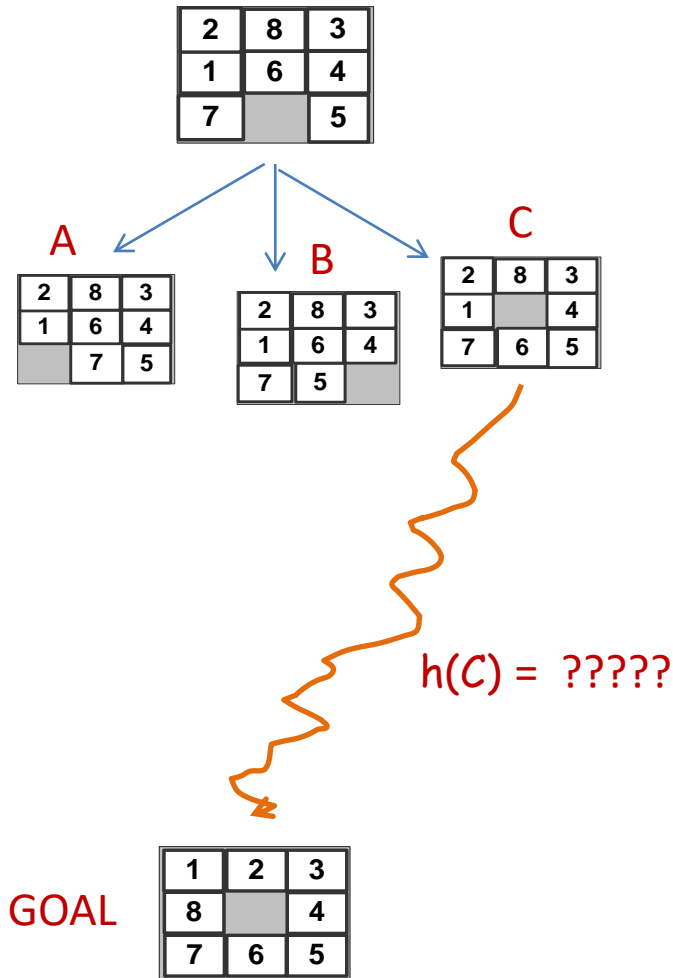


Analizamos el nodo **C**

$g(C) = 1$ → Un movimiento desde el estado inicial con coste = 1

Viendo el nodo C, sabemos que la solución desde **C** hasta el objetivo son 4 pasos
(8 abajo, 2 derecha, 1 arriba, 8 izquierda)

1. Búsqueda heurística



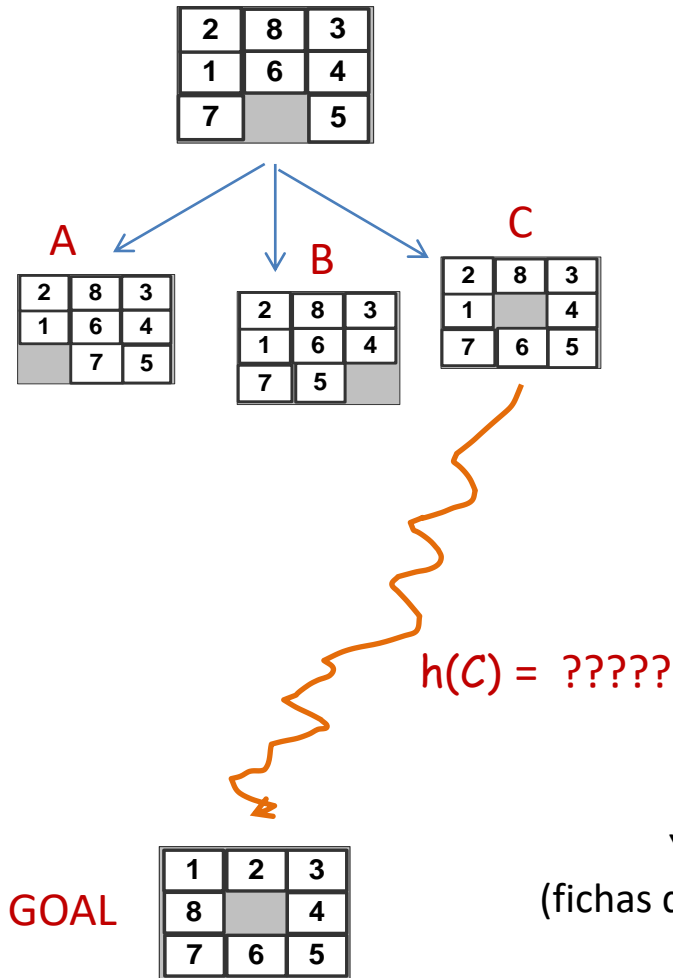
Analizamos el nodo **C**

$g(C) = 1$ → Un movimiento desde el estado inicial con coste = 1

Viendo el nodo C, sabemos que la solución desde **C** hasta el objetivo son 4 pasos
(8 abajo, 2 derecha, 1 arriba, 8 izquierda)

Para llegar a la solución necesitamos una estimación ya que $h(C)$ es un coste aproximado

1. Búsqueda heurística



Analizamos el nodo **C**

$g(C) = 1$ → Un movimiento desde el estado inicial con coste = 1

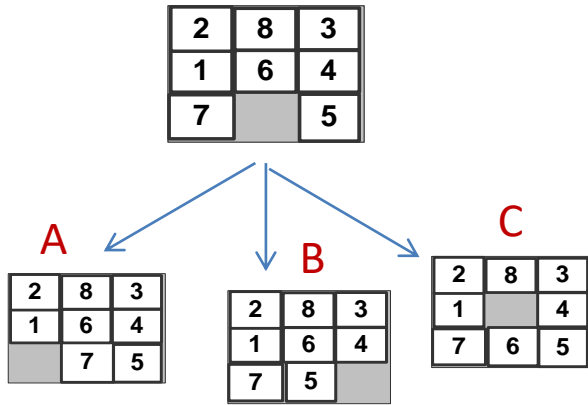
Viendo el nodo C, sabemos que la solución desde **C** hasta el objetivo son 4 pasos
(8 abajo, 2 derecha, 1 arriba, 8 izquierda)

Para llegar a la solución necesitamos una estimación ya que $h(C)$ es un coste aproximado

Y si cuento el número de **fichas descolocadas**
(fichas que no están en su posición objetivo en el nodo C) ??



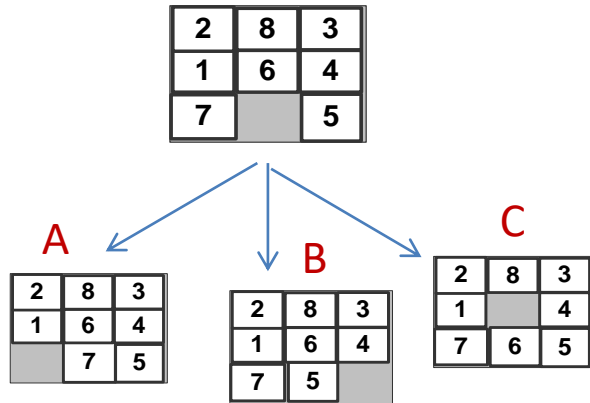
1. Búsqueda heurística



GOAL

1	2	3
8		4
7	6	5

1. Búsqueda heurística



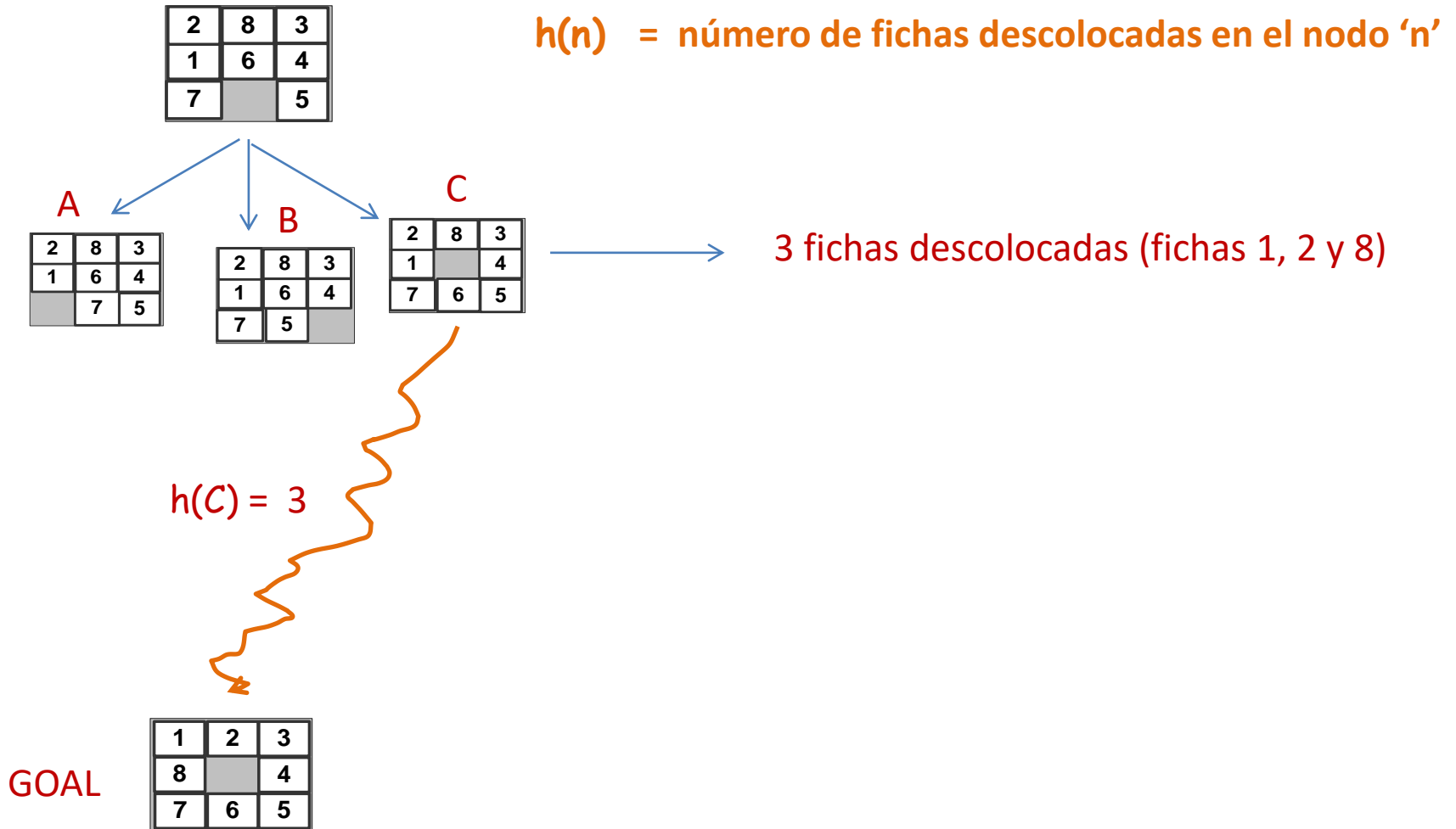
$h(n)$ = número de fichas descolocadas en el nodo 'n'

3 fichas descolocadas (fichas 1, 2 y 8)

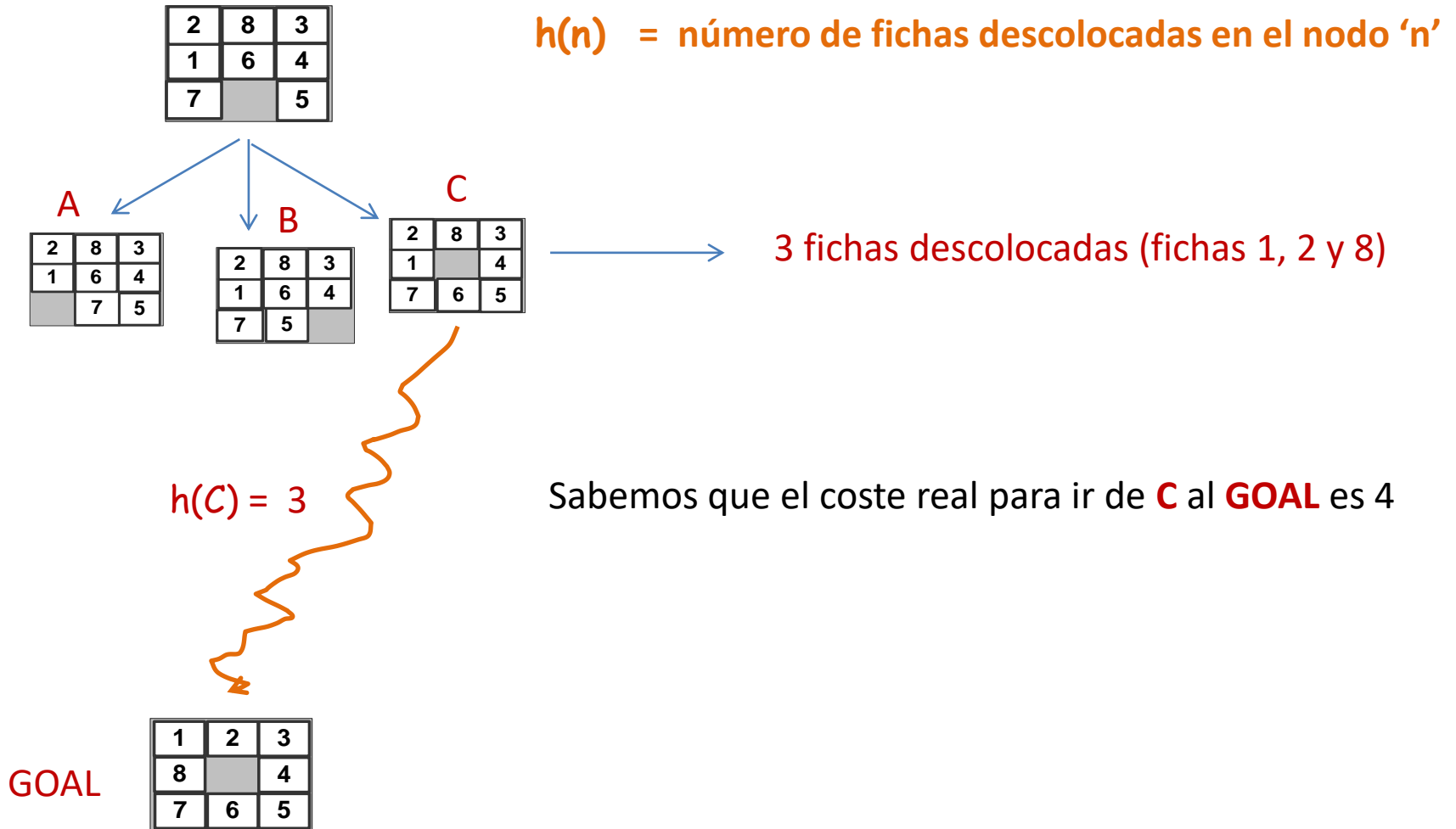
GOAL

1	2	3
8		4
7	6	5

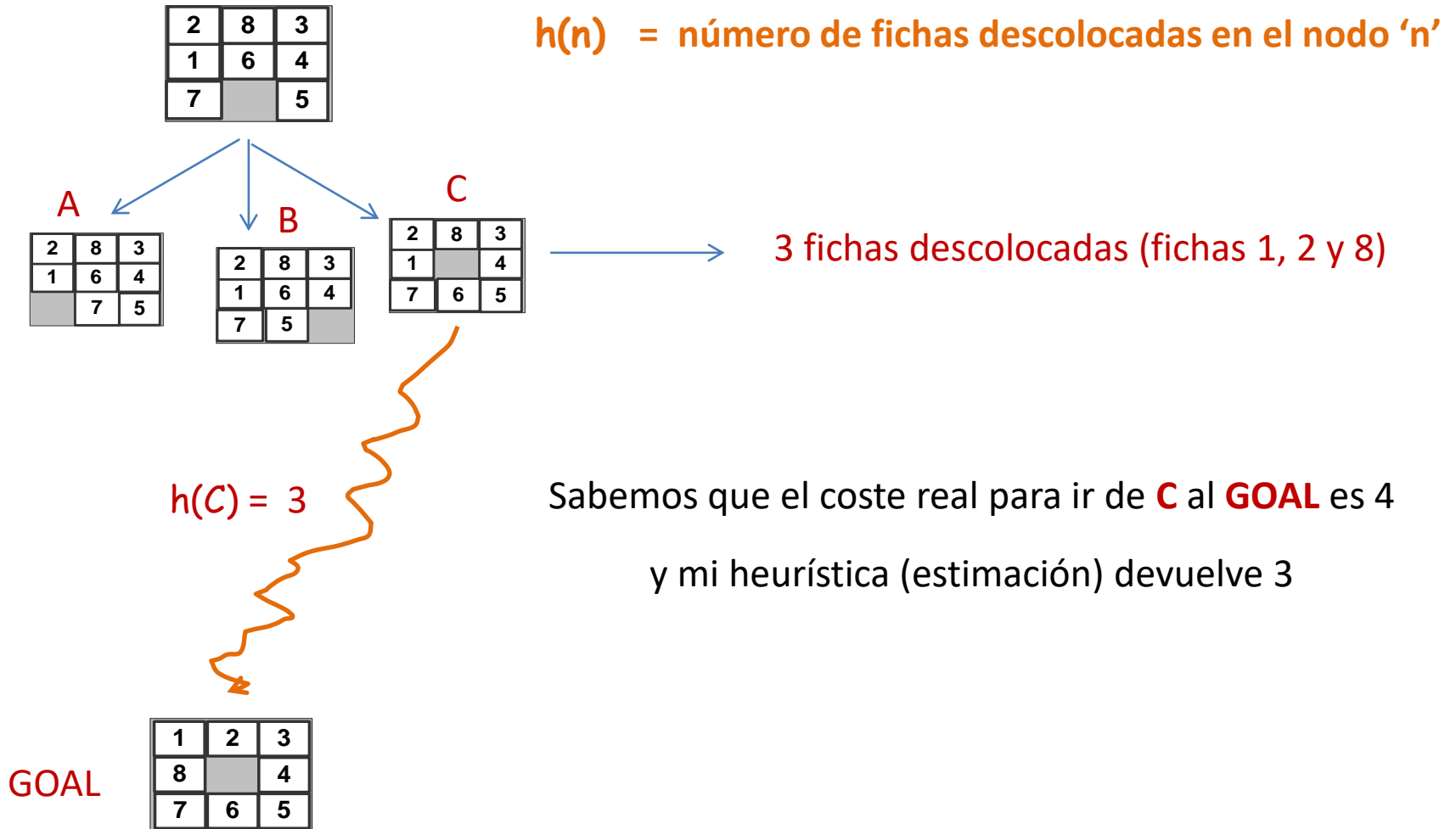
1. Búsqueda heurística



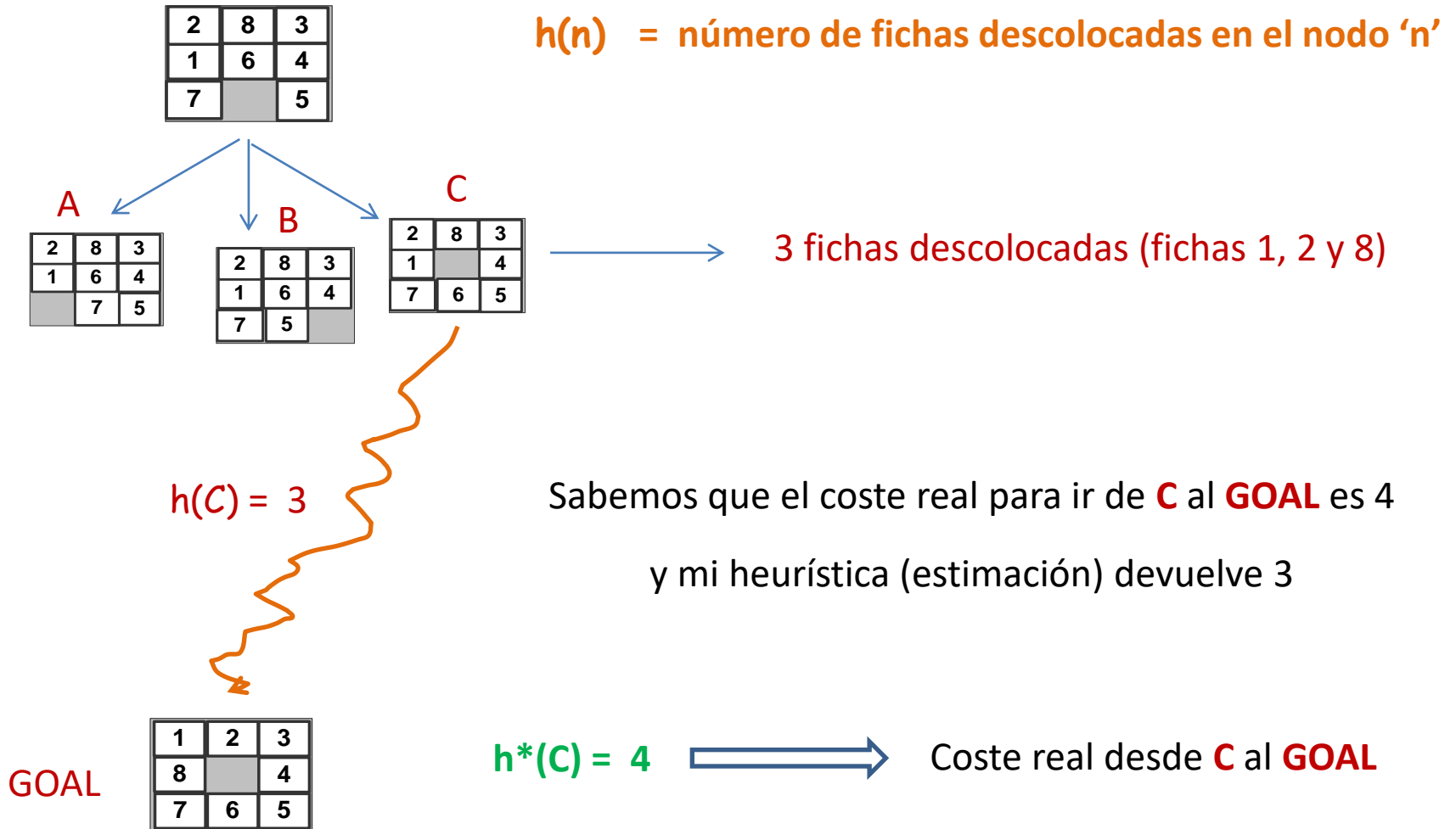
1. Búsqueda heurística



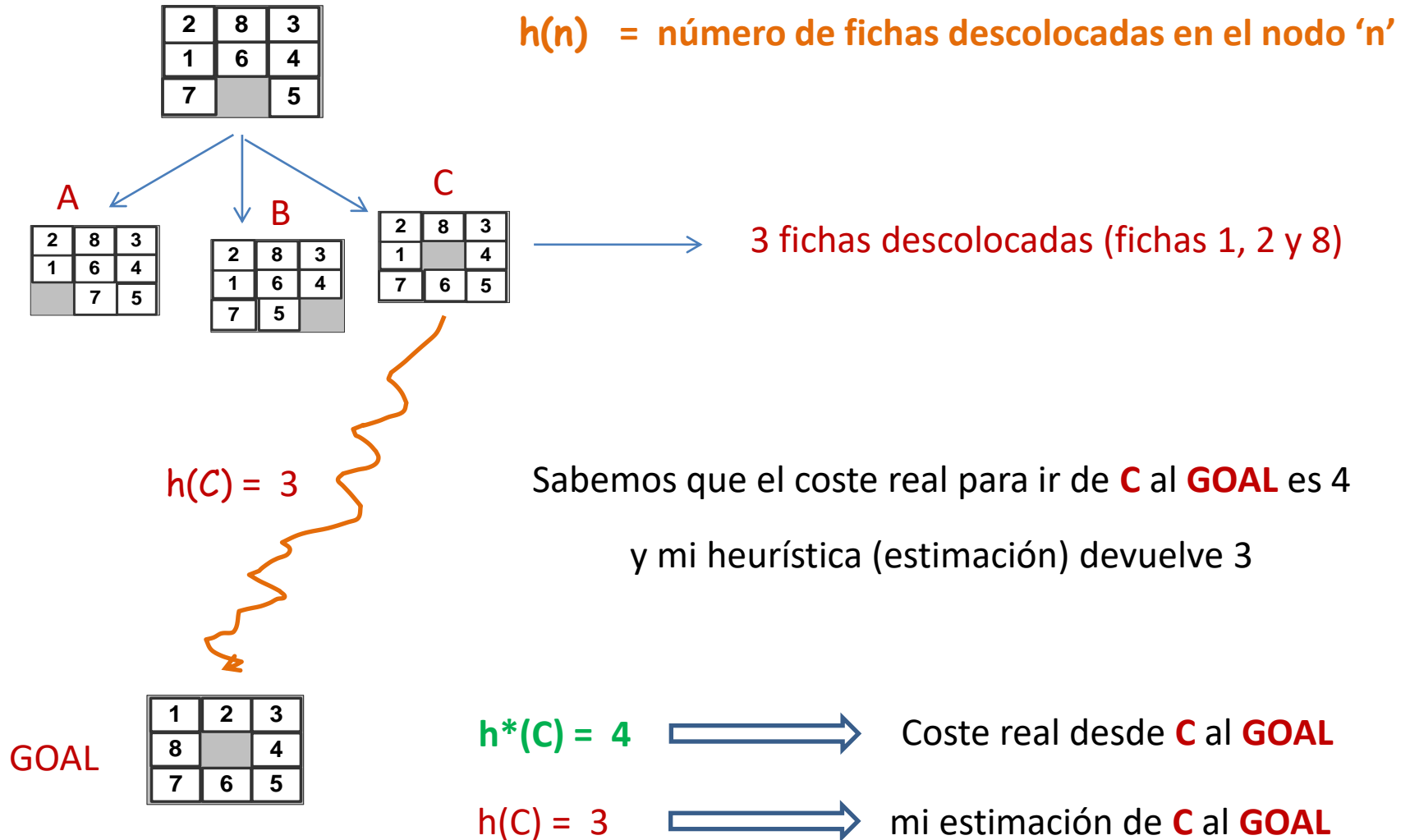
1. Búsqueda heurística



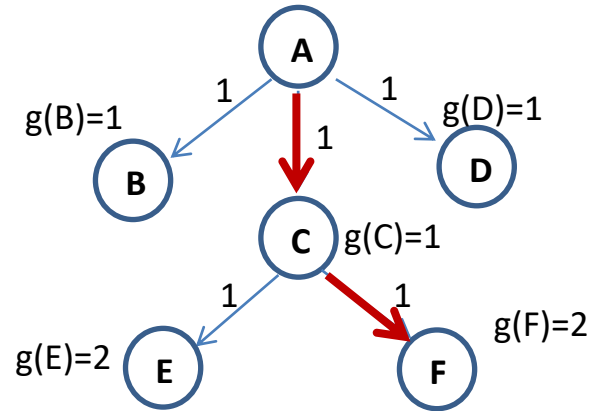
1. Búsqueda heurística



1. Búsqueda heurística

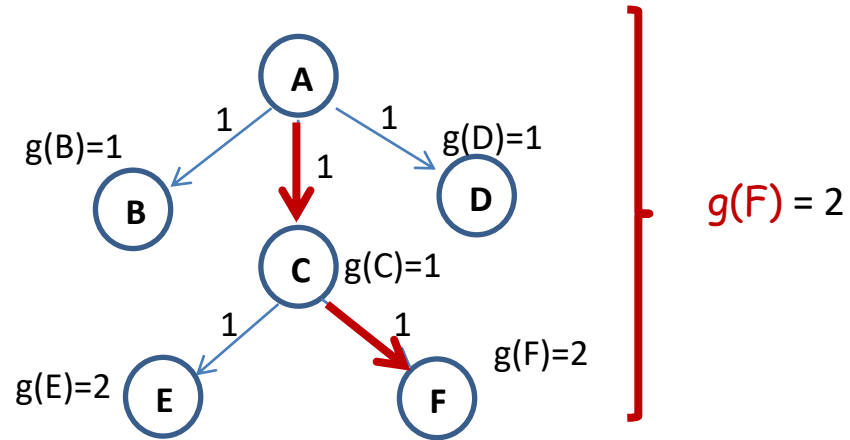


1. Búsqueda heurística



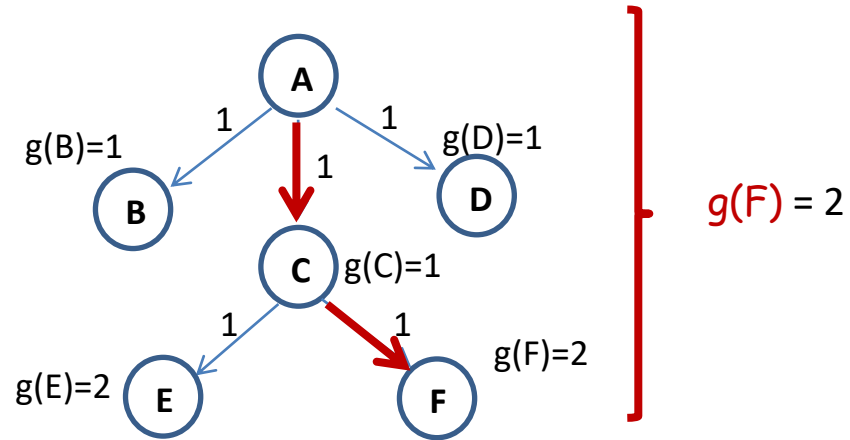
GOAL

1. Búsqueda heurística



GOAL

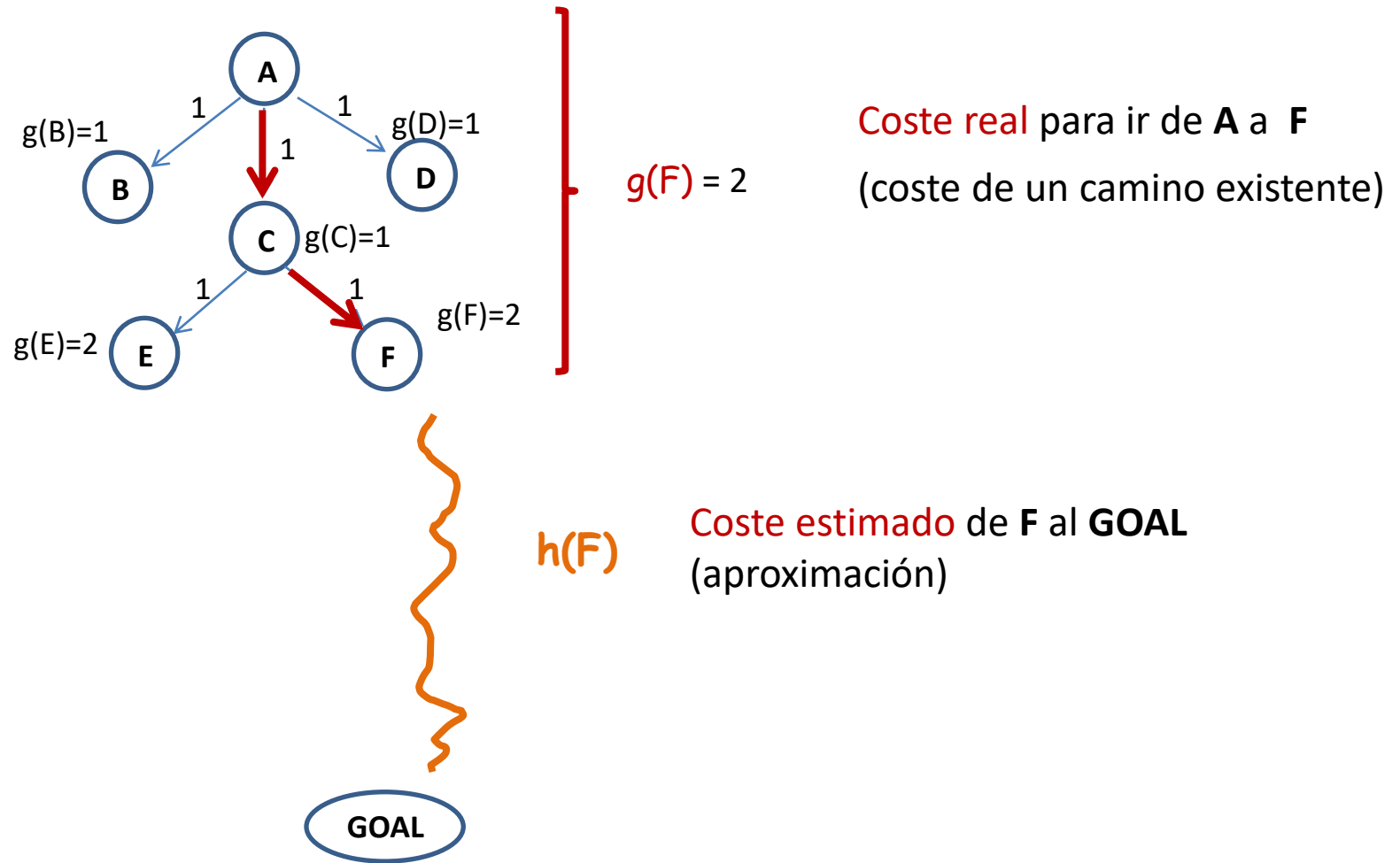
1. Búsqueda heurística



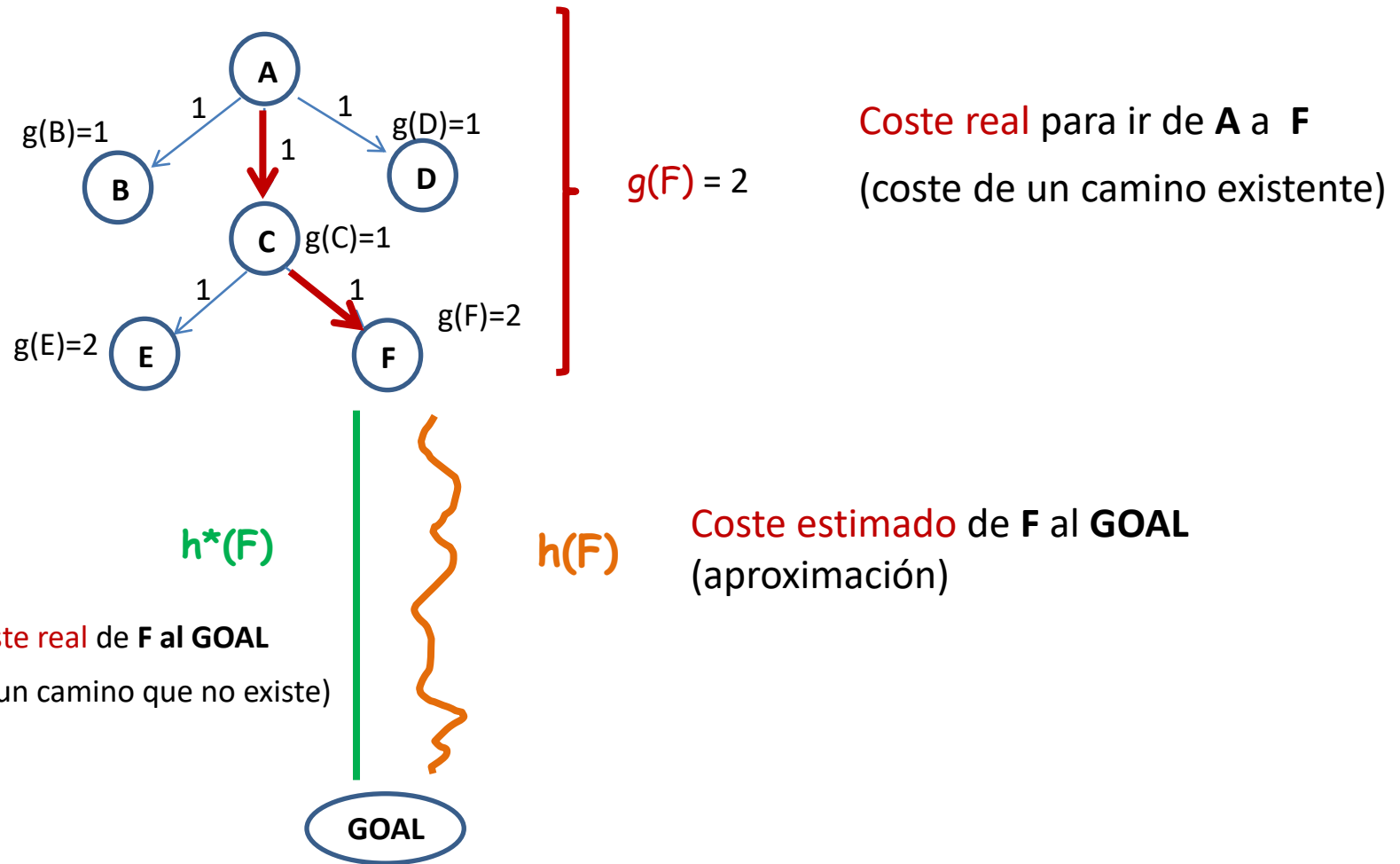
Coste real para ir de **A** a **F**
(coste de un camino existente)

GOAL

1. Búsqueda heurística



1. Búsqueda heurística



1. Búsqueda heurística

1. Búsqueda heurística

¿Por qué utilizar heurísticas? En ocasiones no es viable utilizar una búsqueda sistemática que garantice optimalidad. La utilización de algunas heurísticas permiten obtener una *buena* solución aunque no sea la óptima.

Guía inteligente del proceso de búsqueda que permite podar grandes partes del árbol de búsqueda.

1. Búsqueda heurística

¿Por qué utilizar heurísticas? En ocasiones no es viable utilizar una búsqueda sistemática que garantice optimalidad. La utilización de algunas heurísticas permiten obtener una *buena* solución aunque no sea la óptima.

Guía inteligente del proceso de búsqueda que permite podar grandes partes del árbol de búsqueda.

¿Por qué utilizar heurísticas es apropiado?

1. Normalmente, en problemas complejos, no necesitamos soluciones óptimas, una *buena* solución es suficiente.
2. La solución de la heurística para el caso peor puede no ser muy buena, pero en el mundo real el caso peor es poco frecuente.
3. Comprender el por qué (por qué no) funciona una heurística ayuda a profundizar en la comprensión del problema

2. Búsqueda voraz

La mayoría de los algoritmos primero-el-mejor incluyen una **función heurística $h(n)$** como componente de la función $f(n)$.

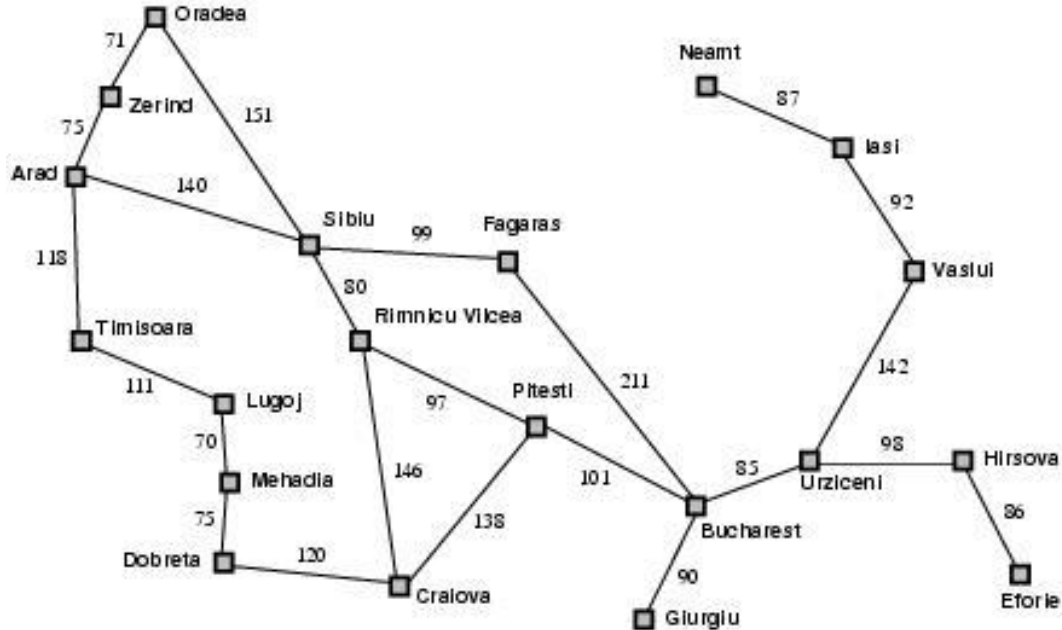
$h(n)$ = *coste estimado* del camino óptimo desde el estado representado en el nodo **n** al estado de objetivo.

Si **n** es el estado de objetivo entonces $h(n)=0$.

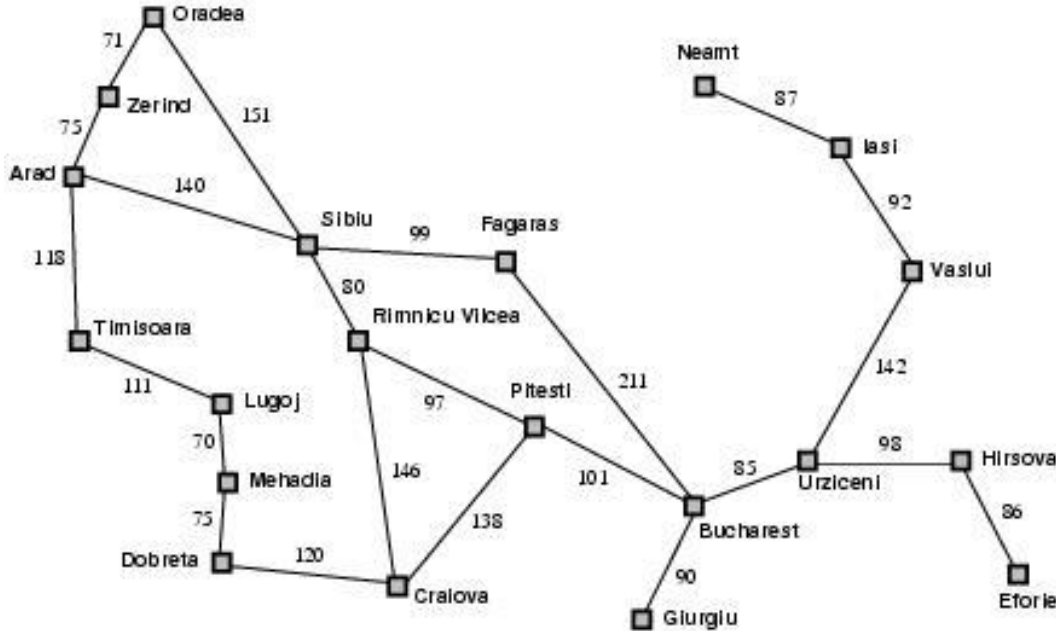
La búsqueda *voraz* expande el nodo que **parece estar** más cerca del objetivo ya que, probamente, dicho nodo conduce más rápidamente a una solución. Evalúa nodos utilizando simplemente: **$f(n)=h(n)$** .

2. Búsqueda voraz: el ejemplo de Rumanía

2. Búsqueda voraz: el ejemplo de Rumanía



2. Búsqueda voraz: el ejemplo de Rumanía



h_{SLD} : heurística 'distancia en línea recta' de una ciudad a Bucarest

h_{SLD} NO se puede calcular a partir de la descripción del problema

Ejemplos:

$$h(\text{Arad})=366$$
$$h(\text{Fagaras})=176$$
$$h(\text{Bucarest})=0$$

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

2. Búsqueda voraz: el ejemplo de Rumanía

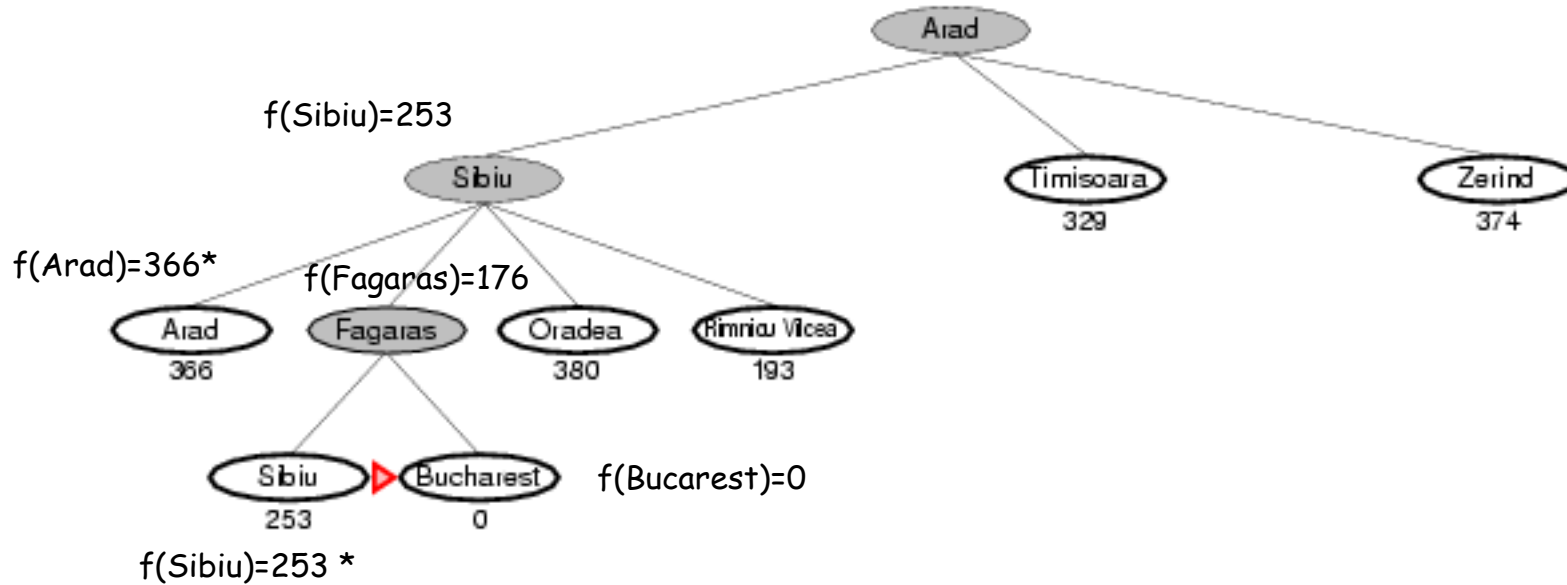
$$f(n)=h(n)$$

- Expande el nodo más cercano al objetivo
- Búsqueda primero-el-mejor voraz

2. Búsqueda voraz: el ejemplo de Rumanía

$f(n)=h(n)$

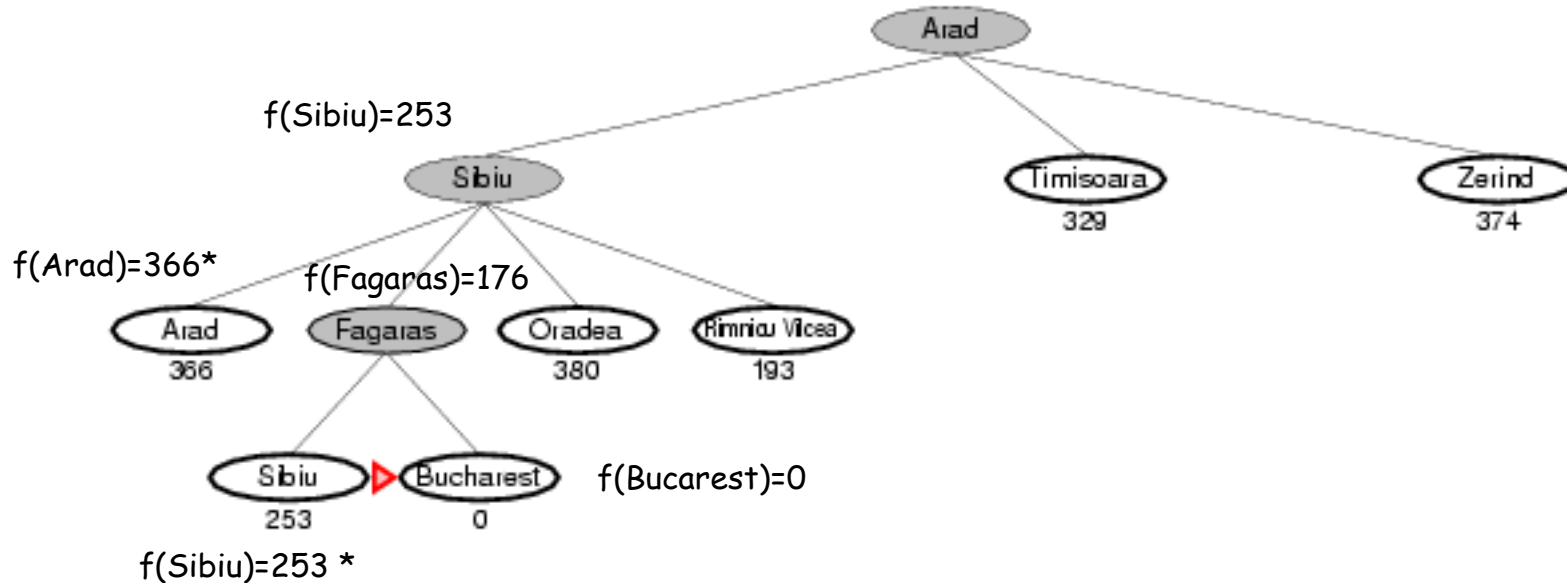
- Expande el nodo más cercano al objetivo
- Búsqueda primero-el-mejor voraz



2. Búsqueda voraz: el ejemplo de Rumanía

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

- Expande el nodo más cercano al objetivo
- Búsqueda primero-el-mejor voraz



Objetivo alcanzado: solución no óptima

(ver solución alternativa: Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucarest)

* En la versión GRAPH-SEARCH se comprobarían estados repetidos en la lista CLOSED y el nodo repetido 'Arad' no se volvería a introducir en la lista OPEN.

2. Búsqueda voraz: evaluación

- Completa:
 - NO, se puede quedar estancado en un ciclo; p. ej. ir de Iasi a Fagaras: Iasi → Neamt → Iasi → Neamt (callejón sin salida, bucle infinito)
 - se asemeja a primero en profundidad (prefiere seguir un único camino al objetivo)
 - La versión GRAPH-SEARCH es completa
- Óptima:
 - No, en cada paso escoge el nodo más cercano al objetivo (**voraz**)

2. Búsqueda voraz: evaluación

- Completa:

- NO, se puede quedar estancado en un ciclo; p. ej. ir de Iasi a Fagaras: Iasi → Neamt → Iasi → Neamt (callejón sin salida, bucle infinito)
- se asemeja a primero en profundidad (prefiere seguir un único camino al objetivo)
- La versión GRAPH-SEARCH es completa

- Óptima:

- No, en cada paso escoge el nodo más cercano al objetivo (**voraz**)

- Complejidad temporal:

- $O(b^m)$ donde **m** es la profundidad máxima del espacio de búsqueda (como el peor caso en profundidad)
- La utilización de buenas heurísticas puede mejorar la búsqueda notablemente
- La reducción del espacio de búsqueda dependerá del problema particular y la calidad de la heurística

2. Búsqueda voraz: evaluación

- Completa:

- NO, se puede quedar estancado en un ciclo; p. ej. ir de Iasi a Fagaras: Iasi \rightarrow Neamt \rightarrow Iasi \rightarrow Neamt (callejón sin salida, bucle infinito)
- se asemeja a primero en profundidad (prefiere seguir un único camino al objetivo)
- La versión GRAPH-SEARCH es completa

- Óptima:

- No, en cada paso escoge el nodo más cercano al objetivo (**voraz**)

- Complejidad temporal:

- $O(b^m)$ donde m es la profundidad máxima del espacio de búsqueda (como el peor caso en profundidad)
- La utilización de buenas heurísticas puede mejorar la búsqueda notablemente
- La reducción del espacio de búsqueda dependerá del problema particular y la calidad de la heurística

- Complejidad espacial:

- $O(b^m)$ donde m es la profundidad máxima del espacio de búsqueda

3. Búsqueda A*

A* es el algoritmo más conocido de búsqueda primero el mejor

Evalúa los nodos combinando $g(n)$, el coste de alcanzar el nodo n , y $h(n)$, el valor heurístico: $f(n)=g(n)+h(n)$

$f(n)$ es el **coste total estimado** de la solución óptima a través del nodo n

Los algoritmos que utilizan una función de evaluación de la forma $f(n)=g(n)+h(n)$ se denominan **algoritmos de tipo A**.

3. Búsqueda A*

La búsqueda A* utiliza una función heurística admisible

Una heurística es admisible si **nunca sobreestima** el coste para lograr el objetivo.

Formalmente:

- Una heurística $h(n)$ es **admisible** si $\forall n, h(n) \leq h^*(n)$, donde $h^*(n)$ es el **coste real** de alcanzar el objetivo desde el estado n .
- Al utilizar un heurístico admisible, la búsqueda A* devuelve la solución óptima
- $h(n) \geq 0$ así que $h(G)=0$ para cualquier objetivo G

P. ej. $h_{SLD}(n)$ nunca sobreestima la distancia en carretera real entre dos ciudades

3. Búsqueda A*: el ejemplo de Rumanía

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


- $h(n)=h_{SLD}(n)$
- expande nodo con el menor coste estimado total al objetivo
- Búsqueda A*

3. Búsqueda A*: el ejemplo de Rumanía

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

- $h(n)=h_{SLD}(n)$
- expande nodo con el menor coste estimado total al objetivo
- Búsqueda A*

Iteración 1:

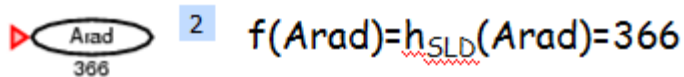
►  2 $f(\text{Arad})=h_{SLD}(\text{Arad})=366$

3. Búsqueda A*: el ejemplo de Rumanía

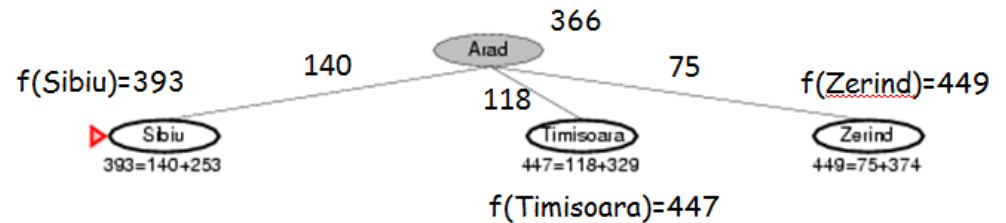
$$f(n)=g(n)+h(n)$$

- $h(n)=h_{SLD}(n)$
- expande nodo con el menor coste estimado total al objetivo
- Búsqueda A*

Iteración 1:



Iteración 2:

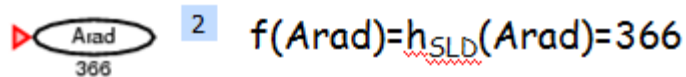


3. Búsqueda A*: el ejemplo de Rumanía

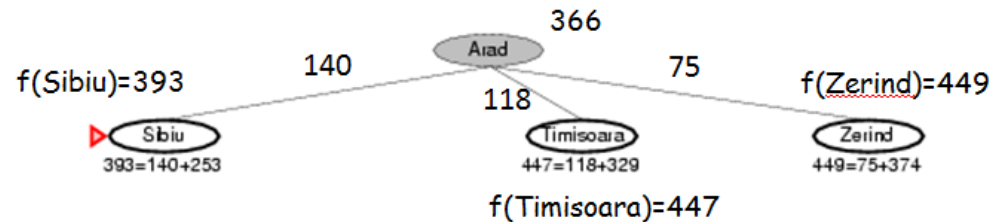
$$f(n)=g(n)+h(n)$$

- $h(n)=h_{SLD}(n)$
- expande nodo con el menor coste estimado total al objetivo
- Búsqueda A*

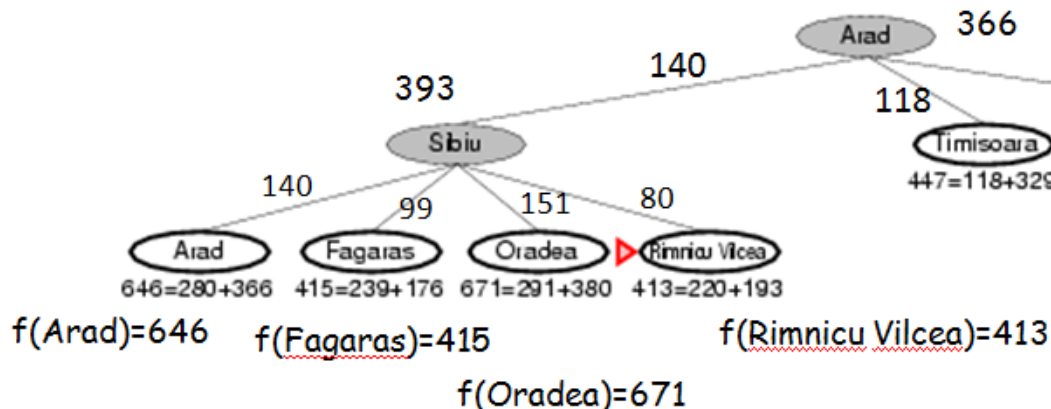
Iteración 1:



Iteración 2:



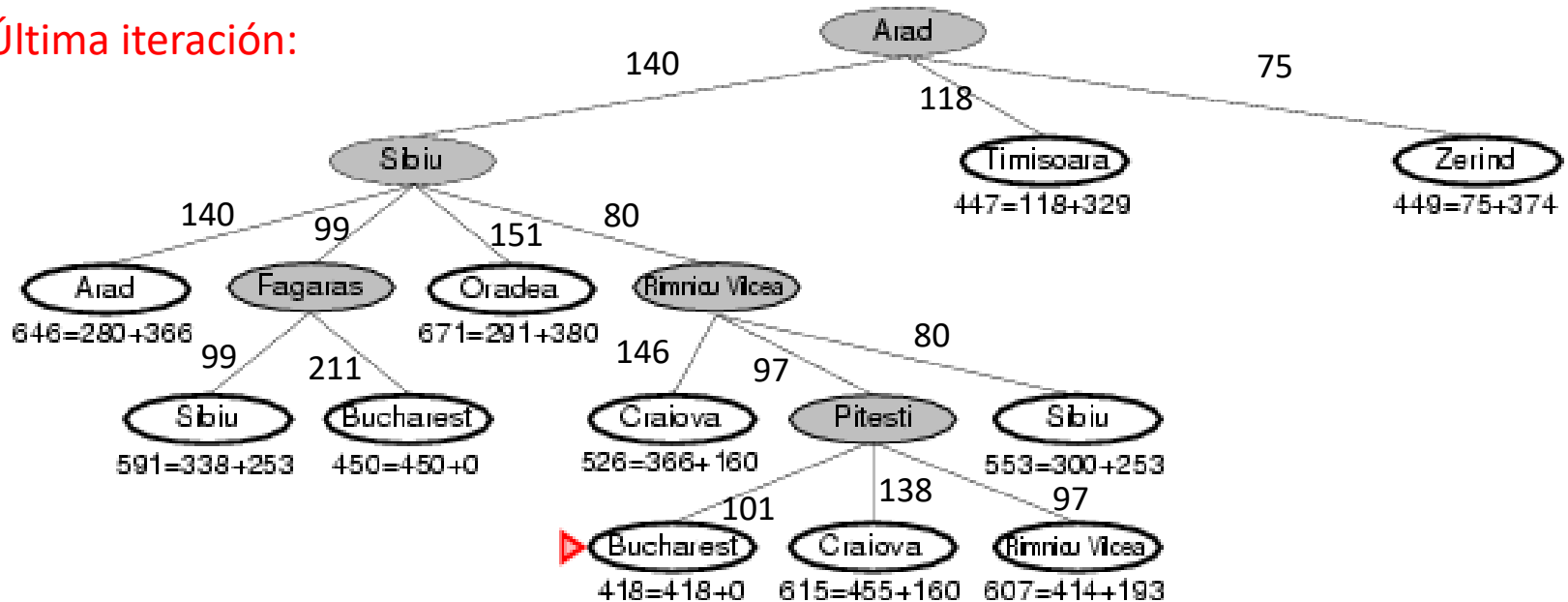
Iteración 3:



En la versión GRAPH-SEARCH, 'Arad' es un estado repetido; el nodo 'Arad' de la lista CLOSED tiene menor coste (mejor valor de $f(n)$)

3. Búsqueda A*: el ejemplo de Rumanía

Última iteración:

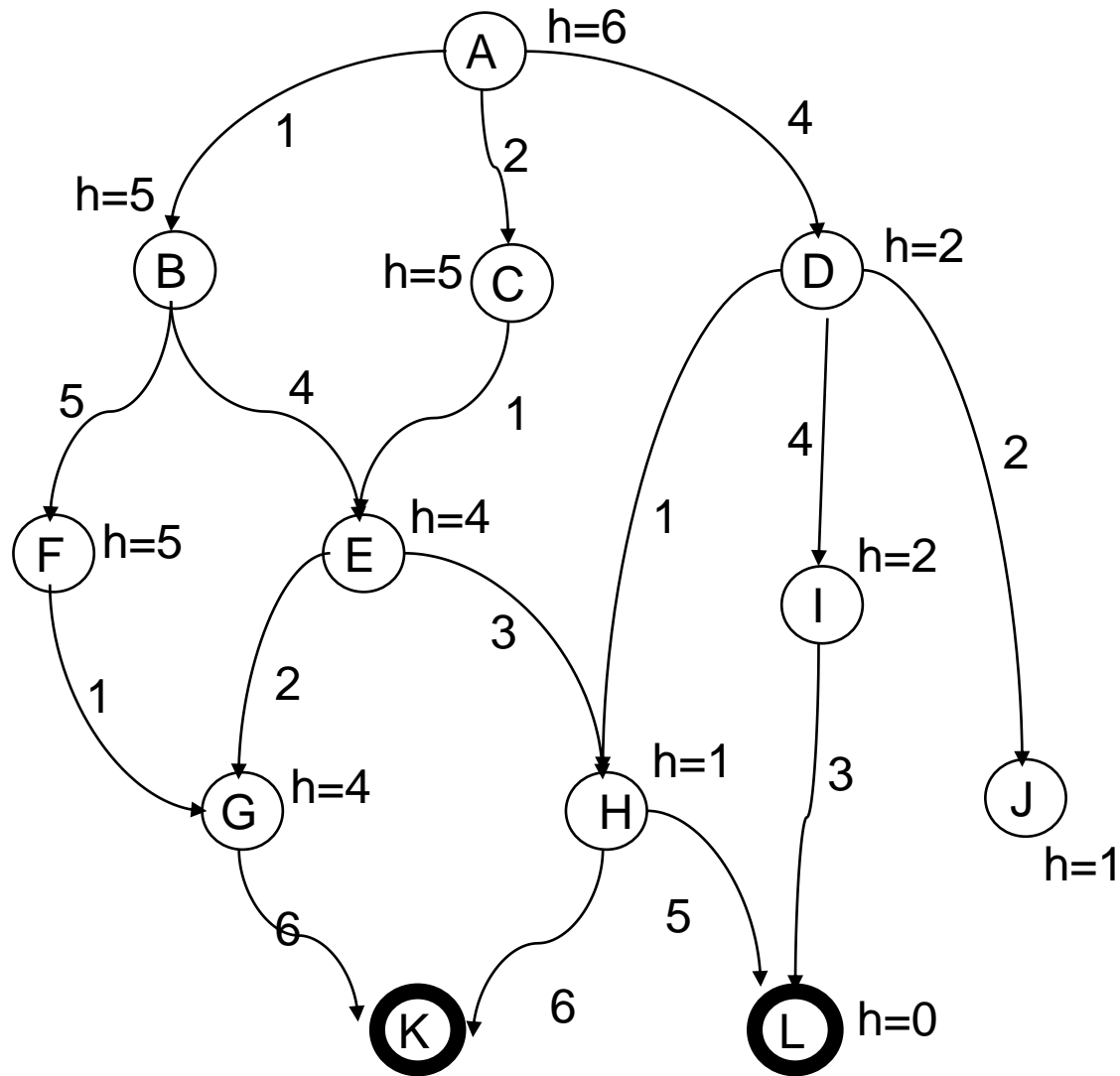


lista OPEN= {Bucarest(418), Timisoara(447), Zerind(449), Craiova(526), Oradea (671)}

lista CLOSED = {Arad, Sibiu, Rimnicu Vilcea, Fagaras, Pitesti}

El nodo Bucarest ya está en OPEN con coste=450. El nuevo nodo 'Bucarest' tiene un coste estimado menor (coste=418) que el nodo que está en OPEN. Reemplazamos el nodo de Bucarest en OPEN con el nuevo nodo encontrado.

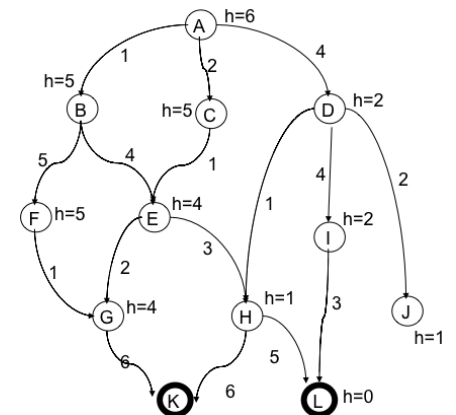
3. Búsqueda A*: otro ejemplo



3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos

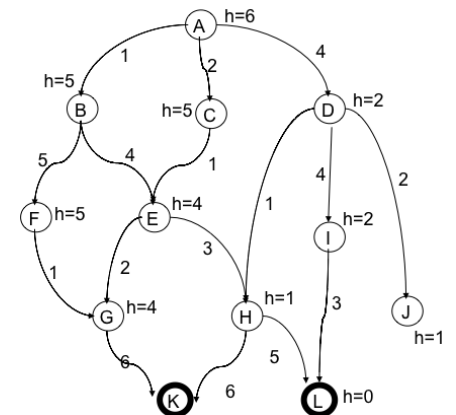
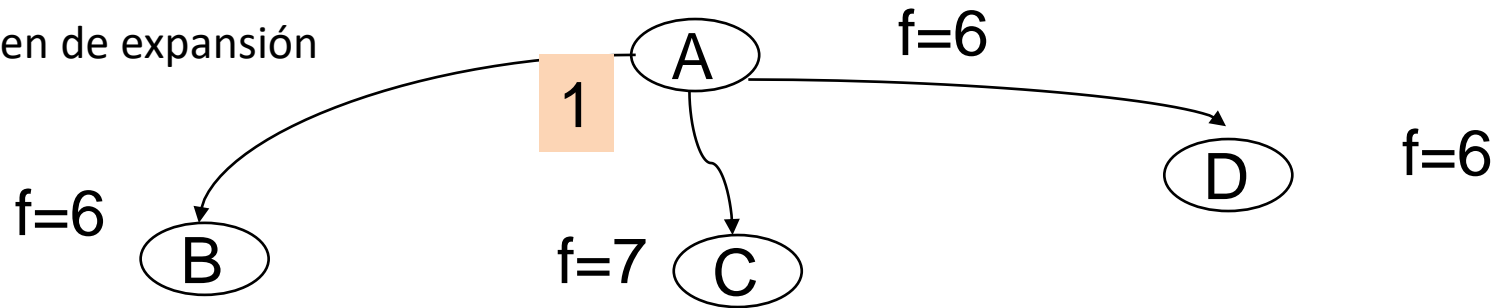
Orden de expansión

A

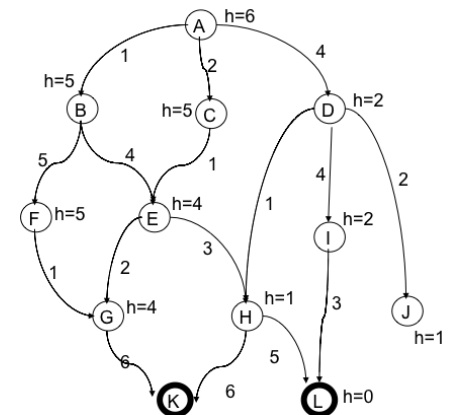
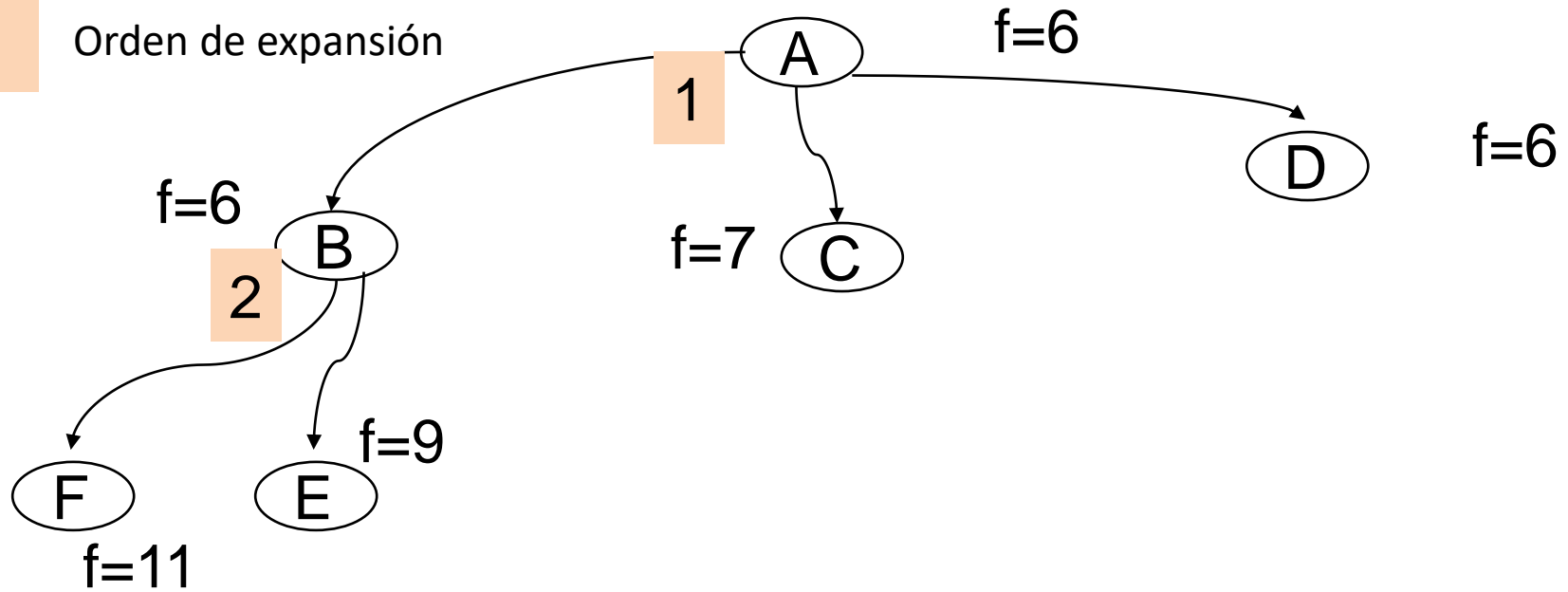


3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos

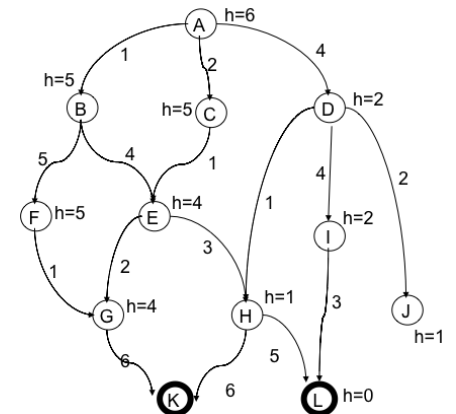
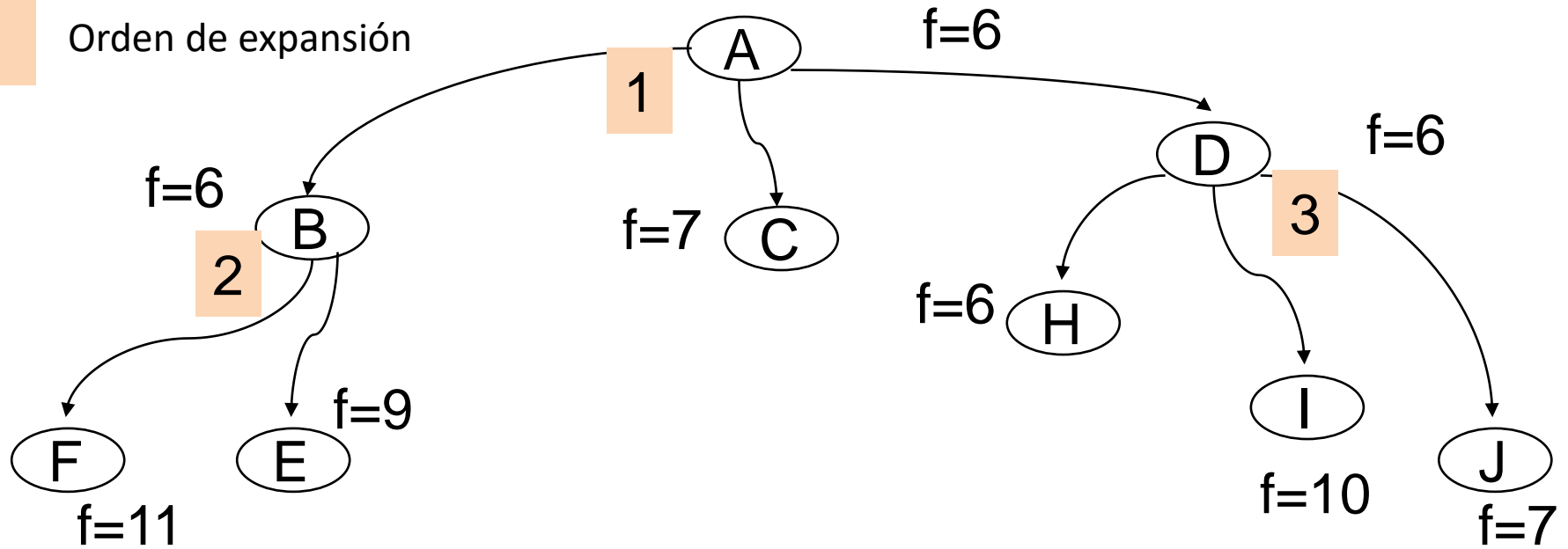
Orden de expansión



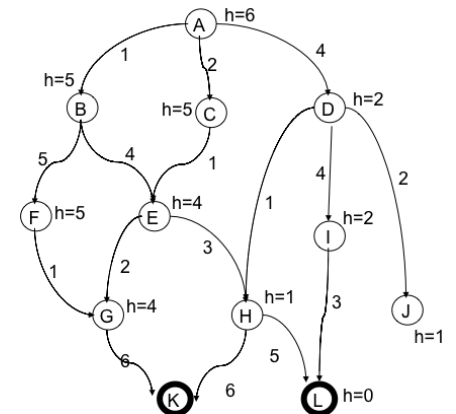
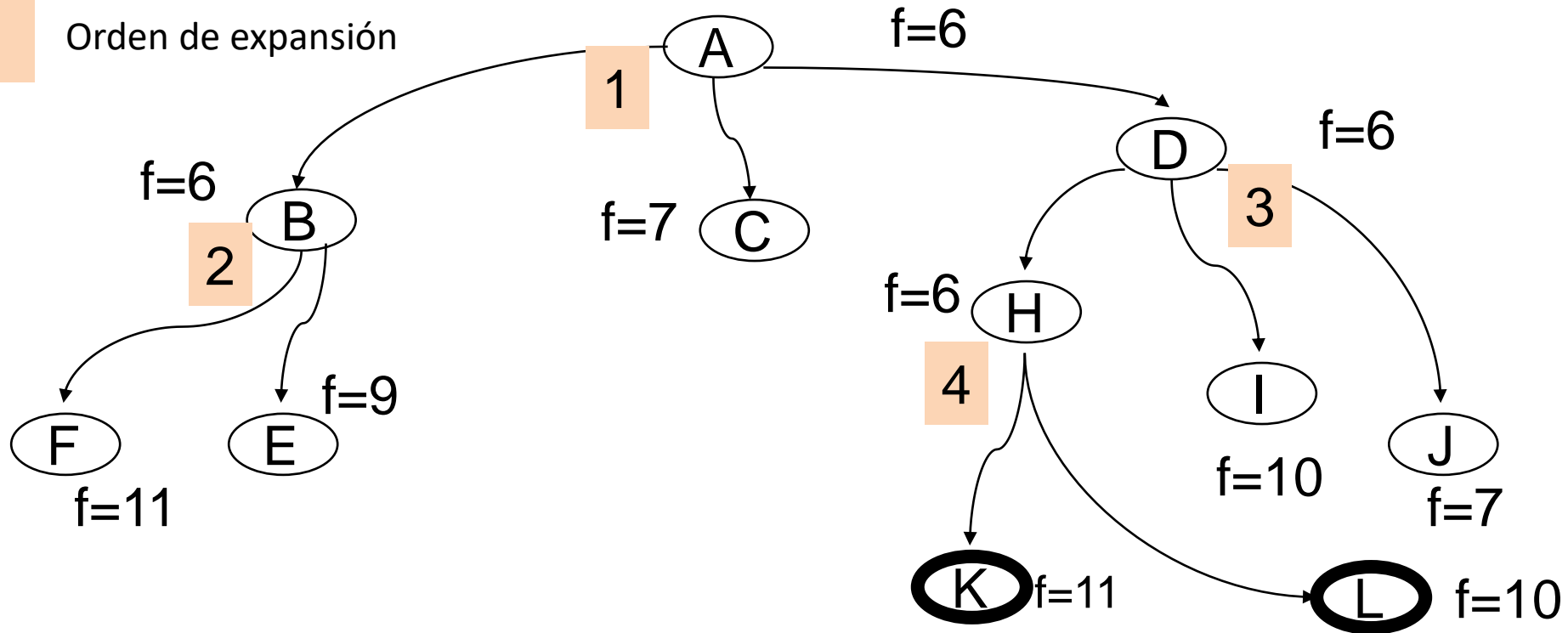
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



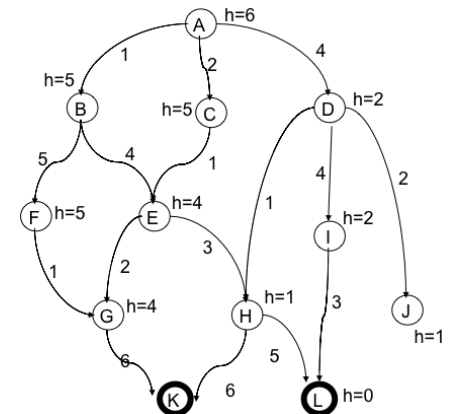
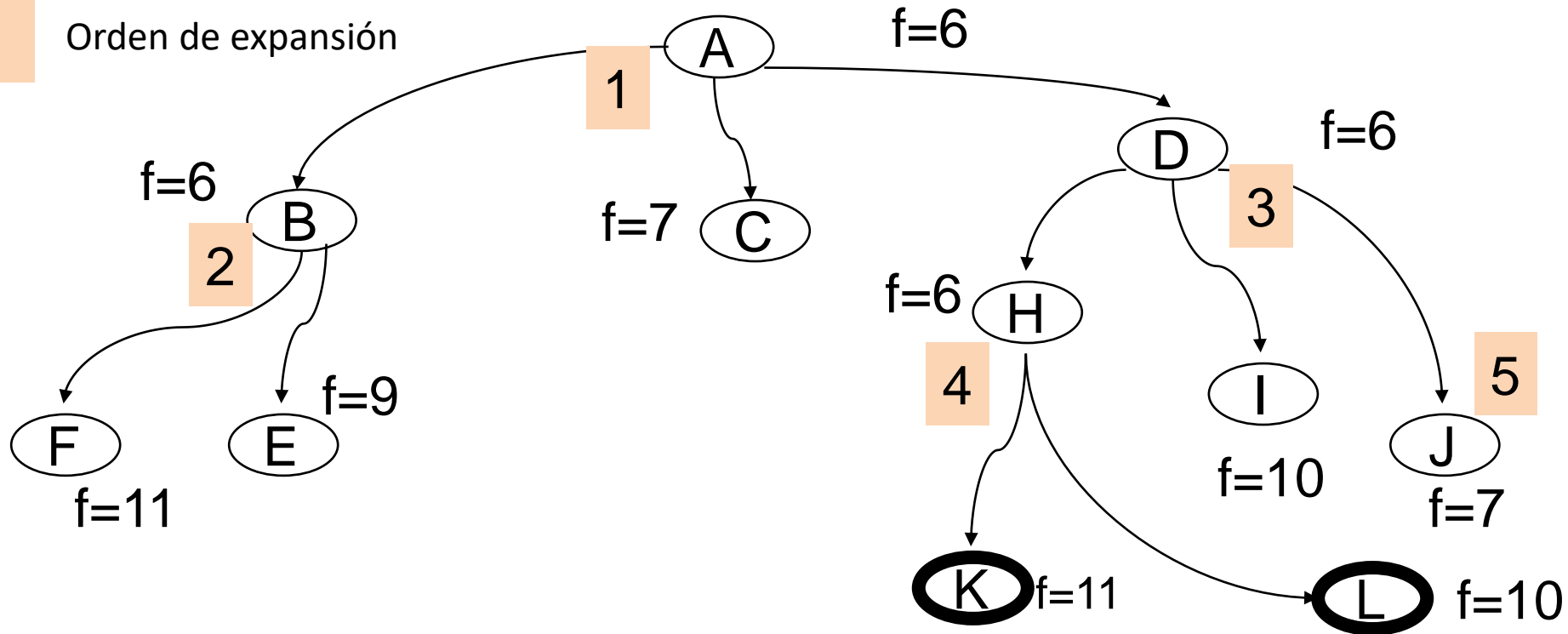
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



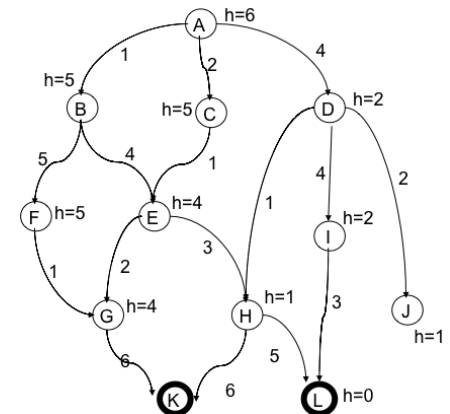
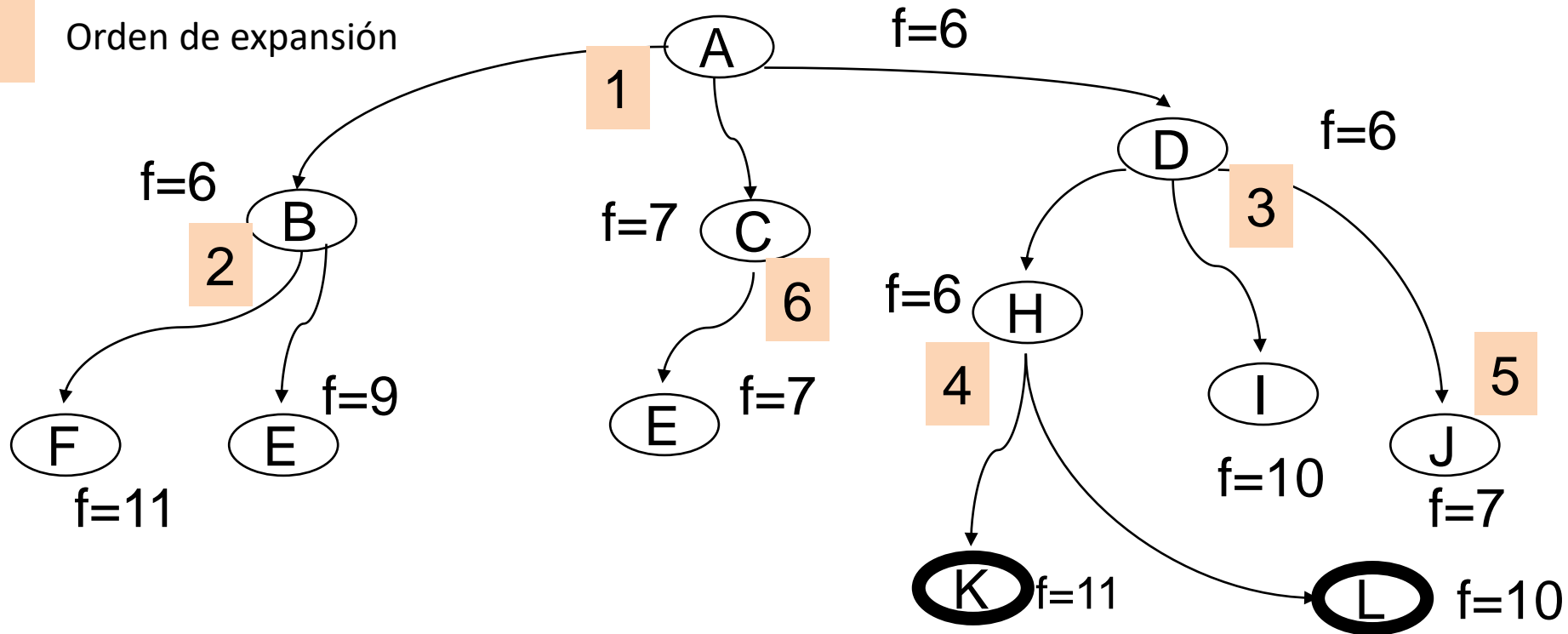
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



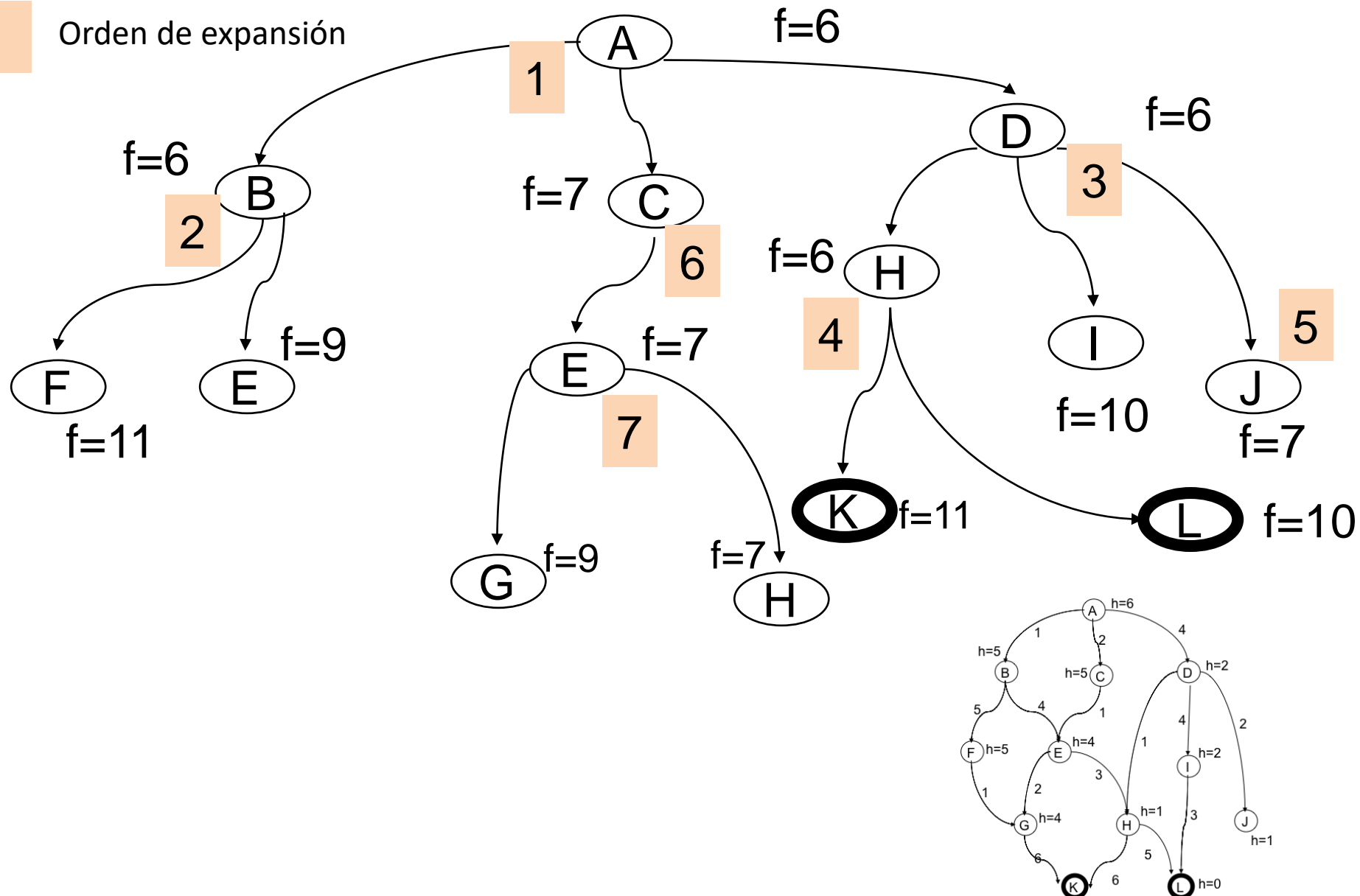
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



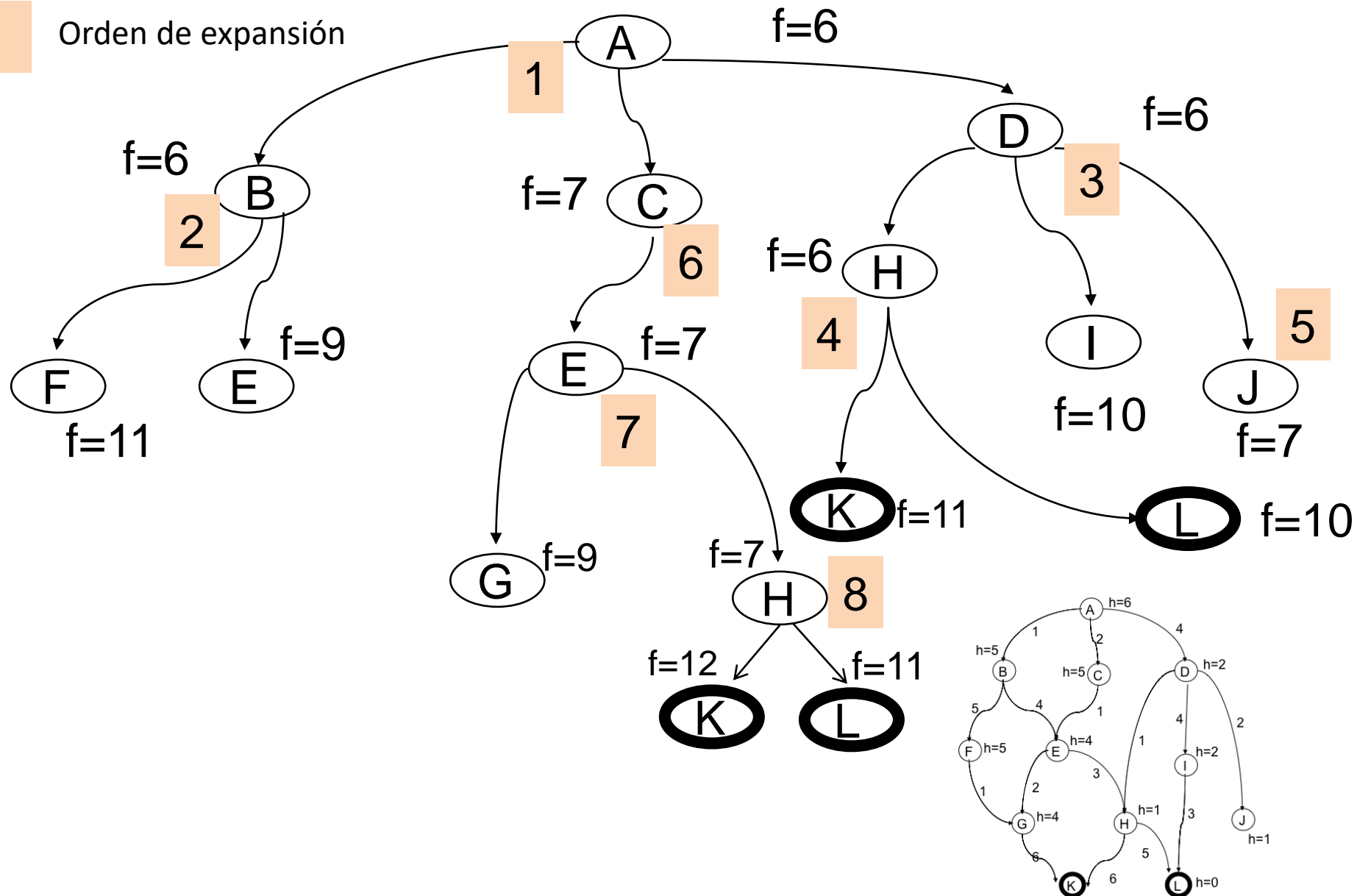
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



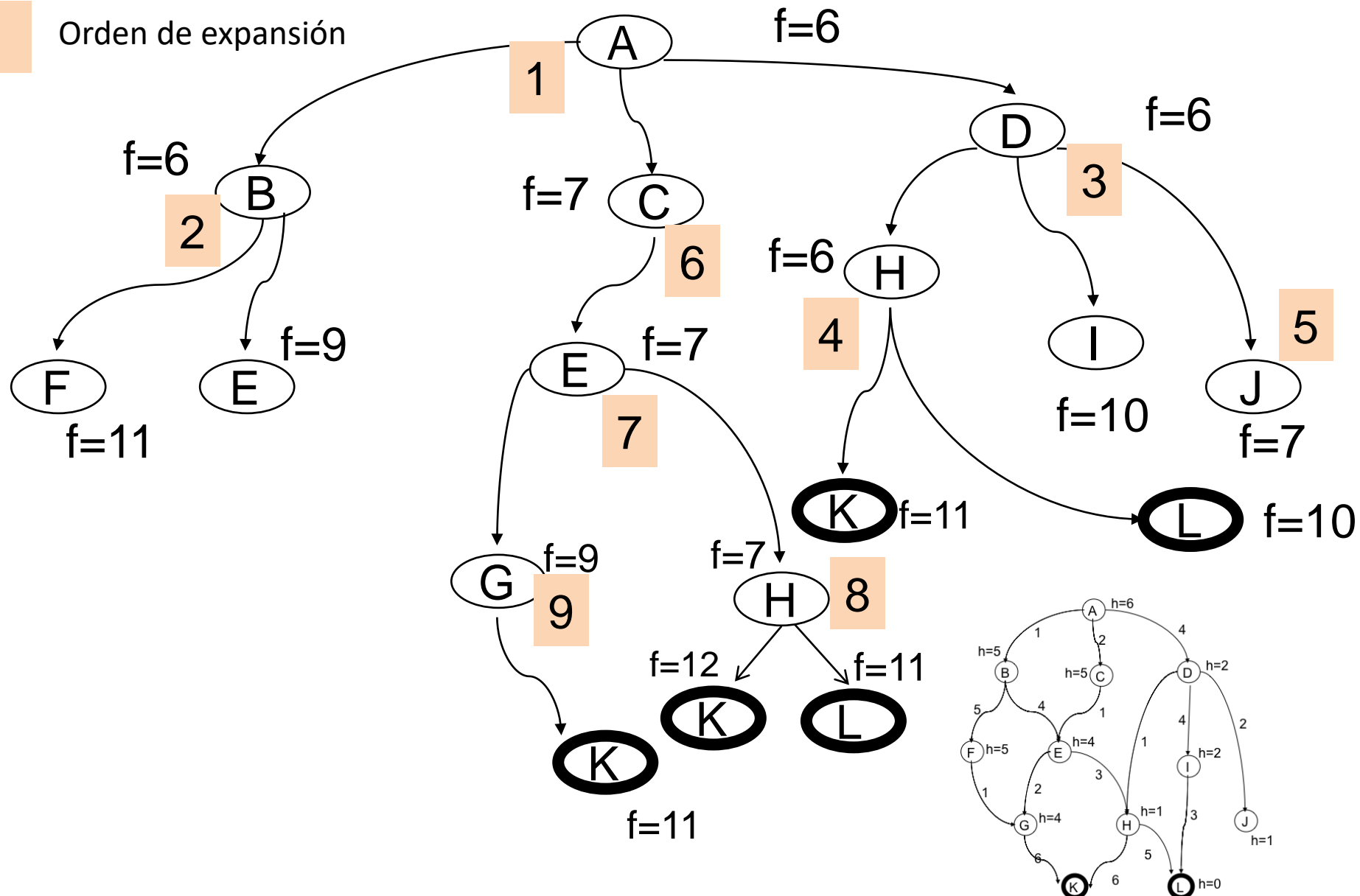
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



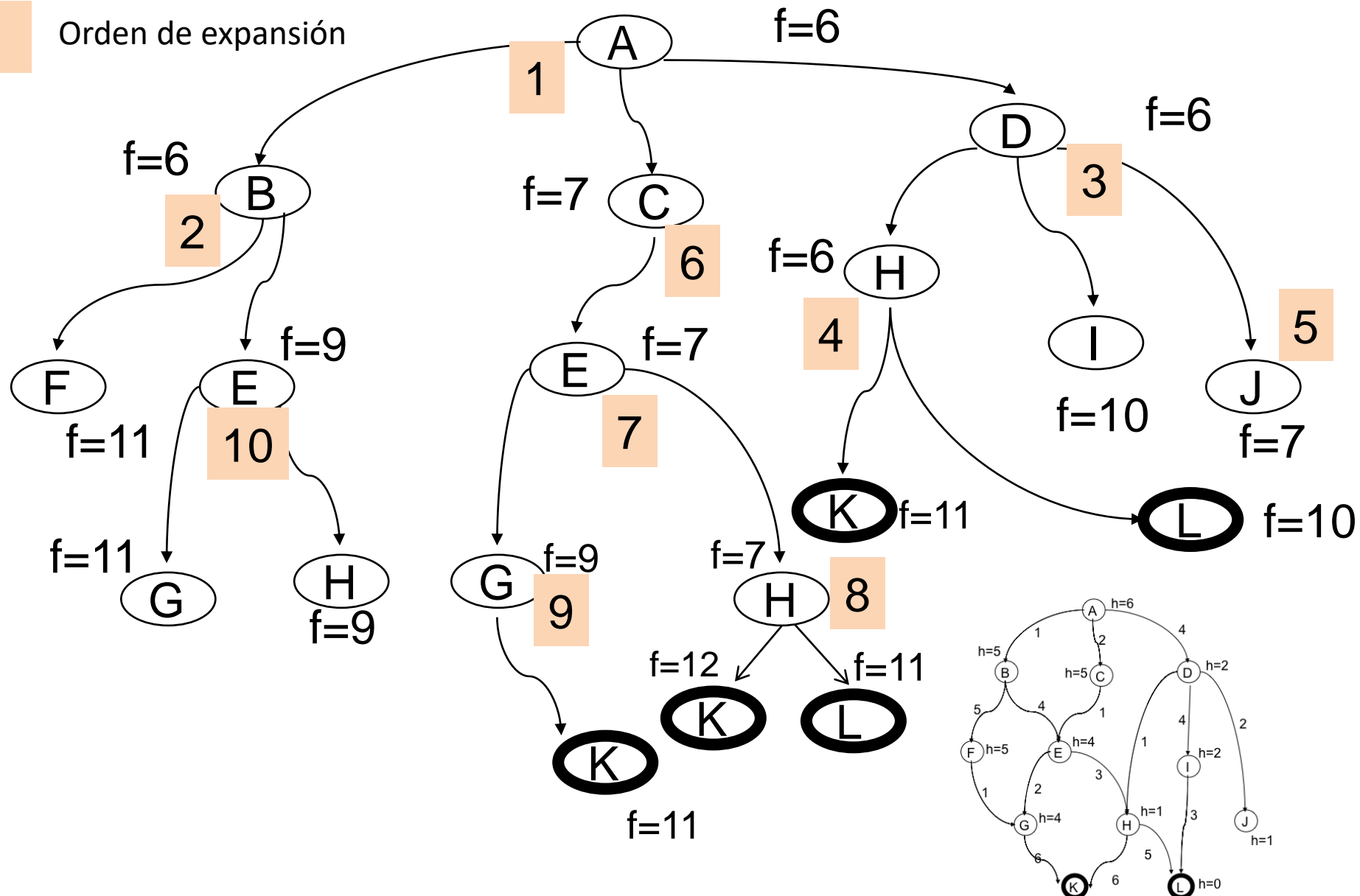
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



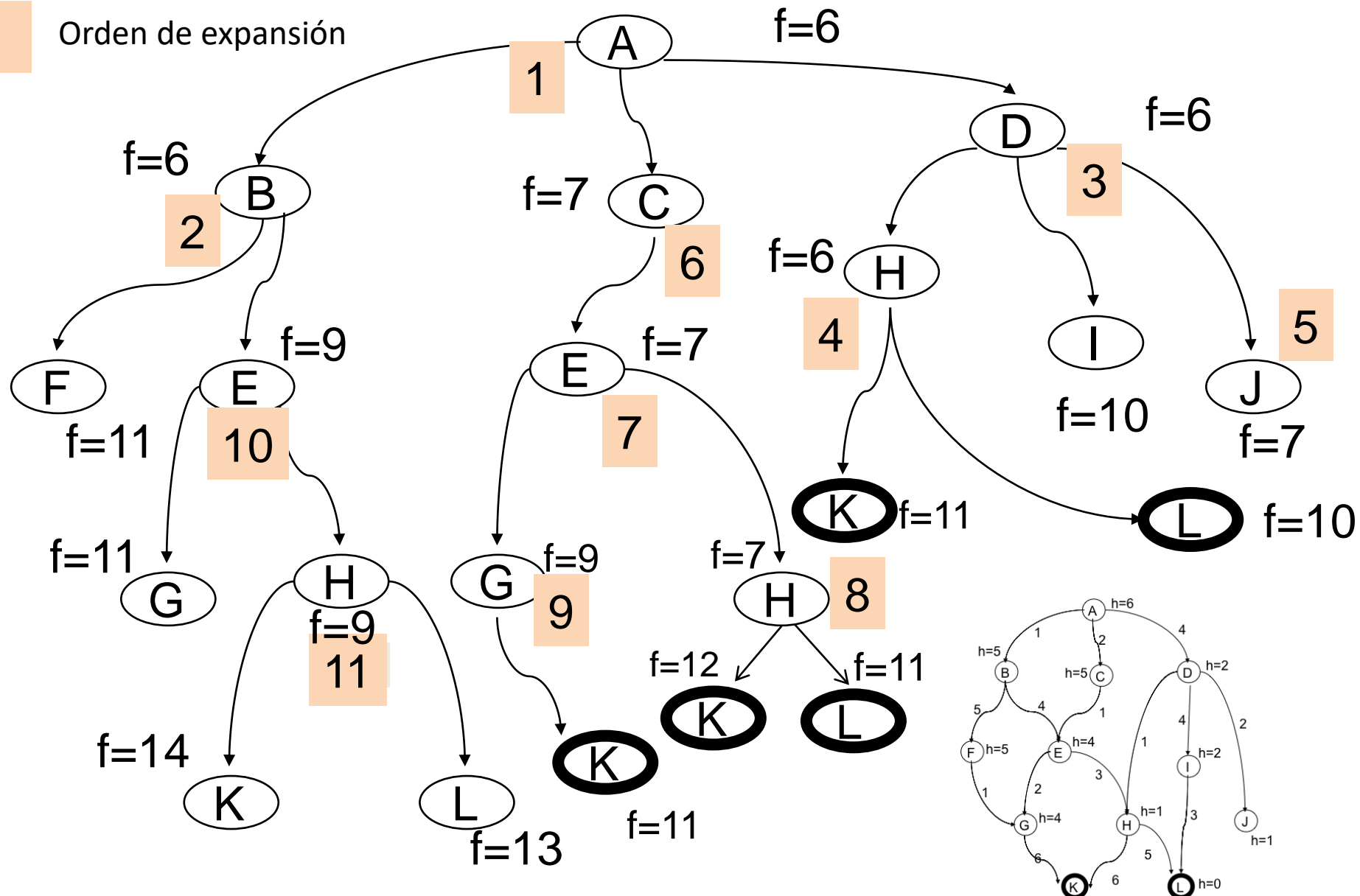
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



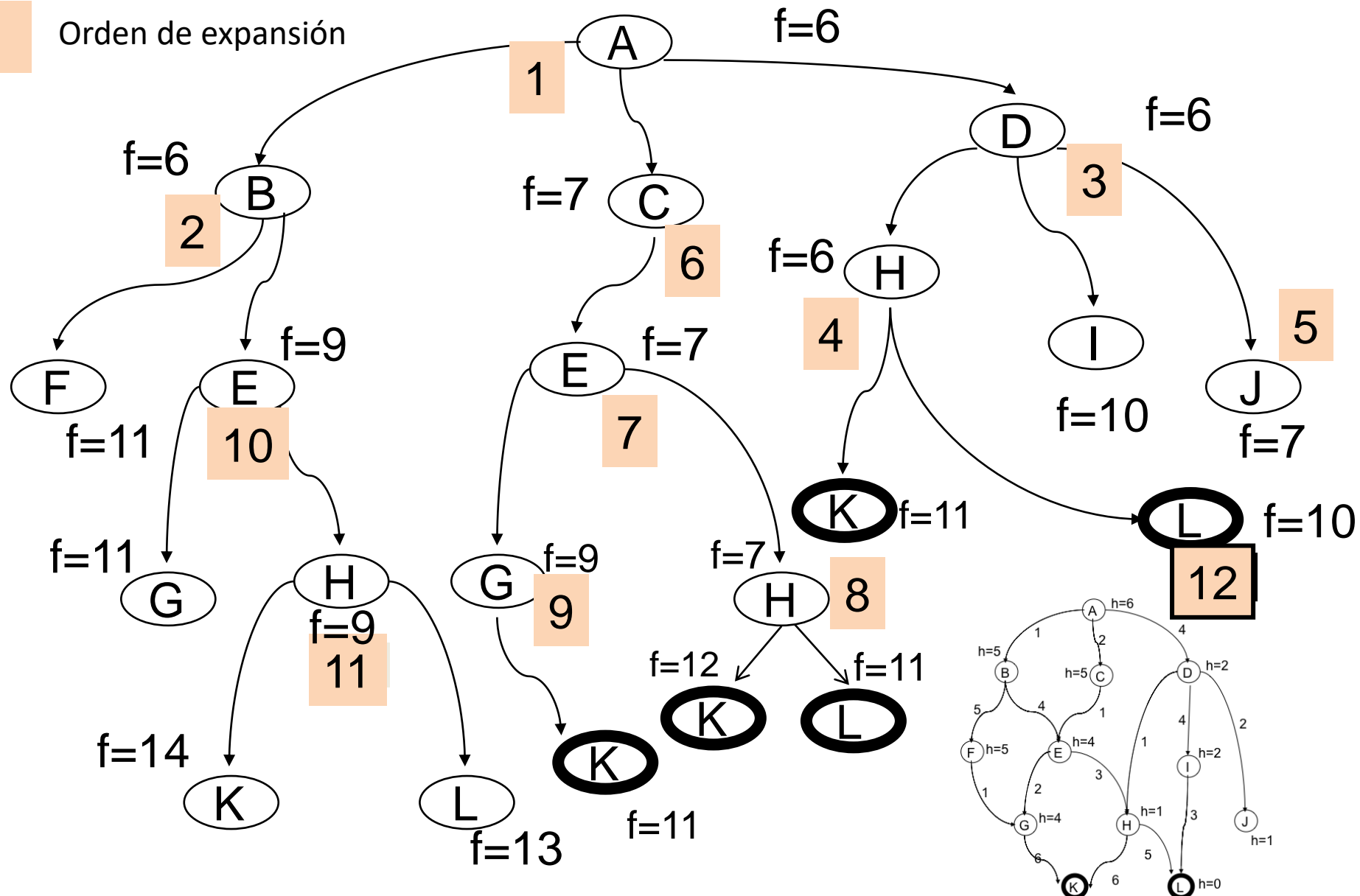
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos

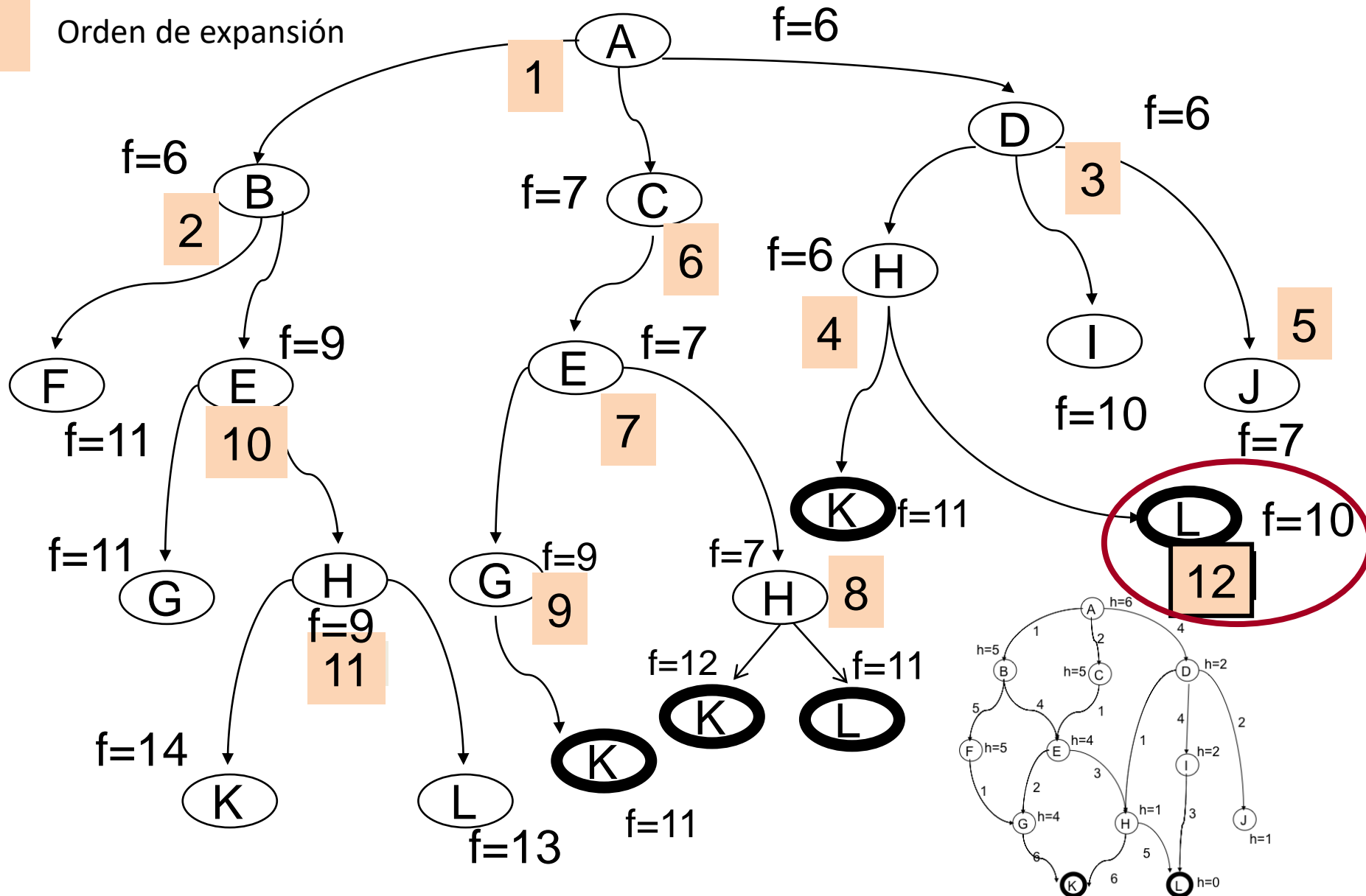


3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos



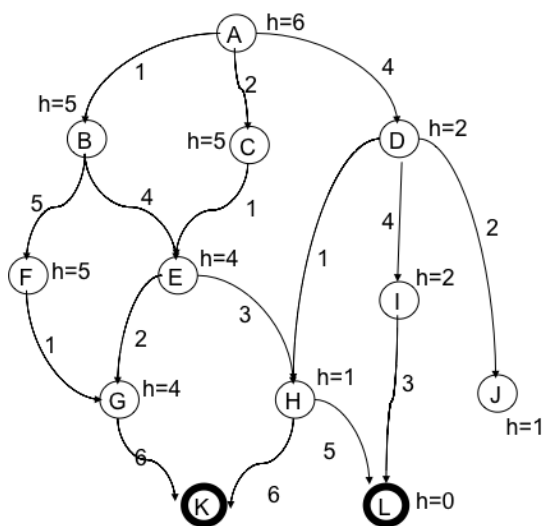
3. Búsqueda A*: versión TREE-SEARCH sin control de nodos repetidos

Orden de expansión



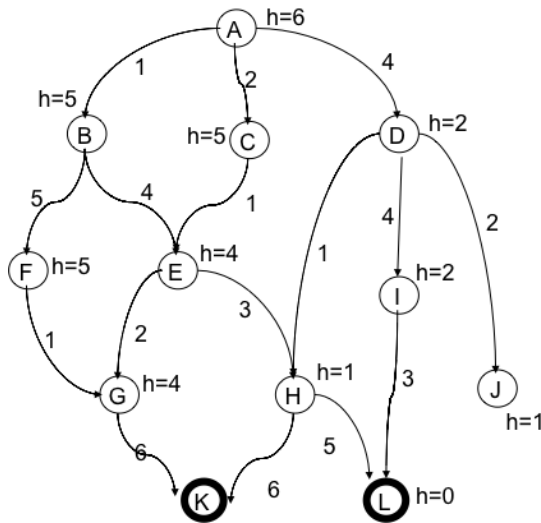
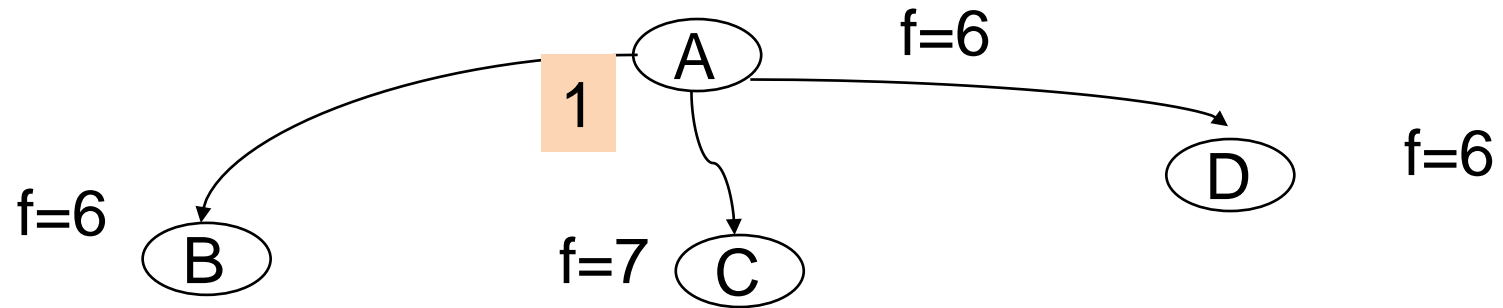
3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos

A



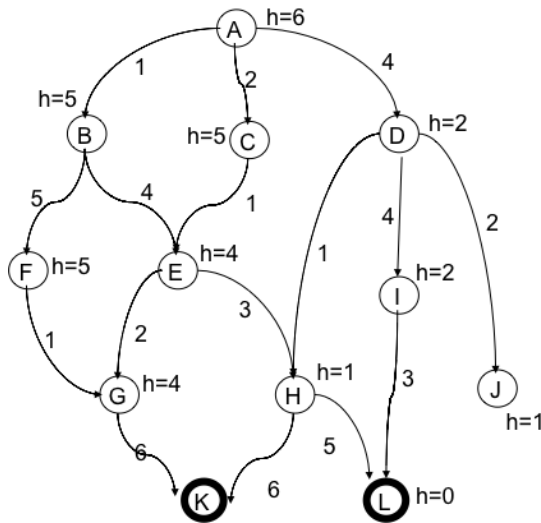
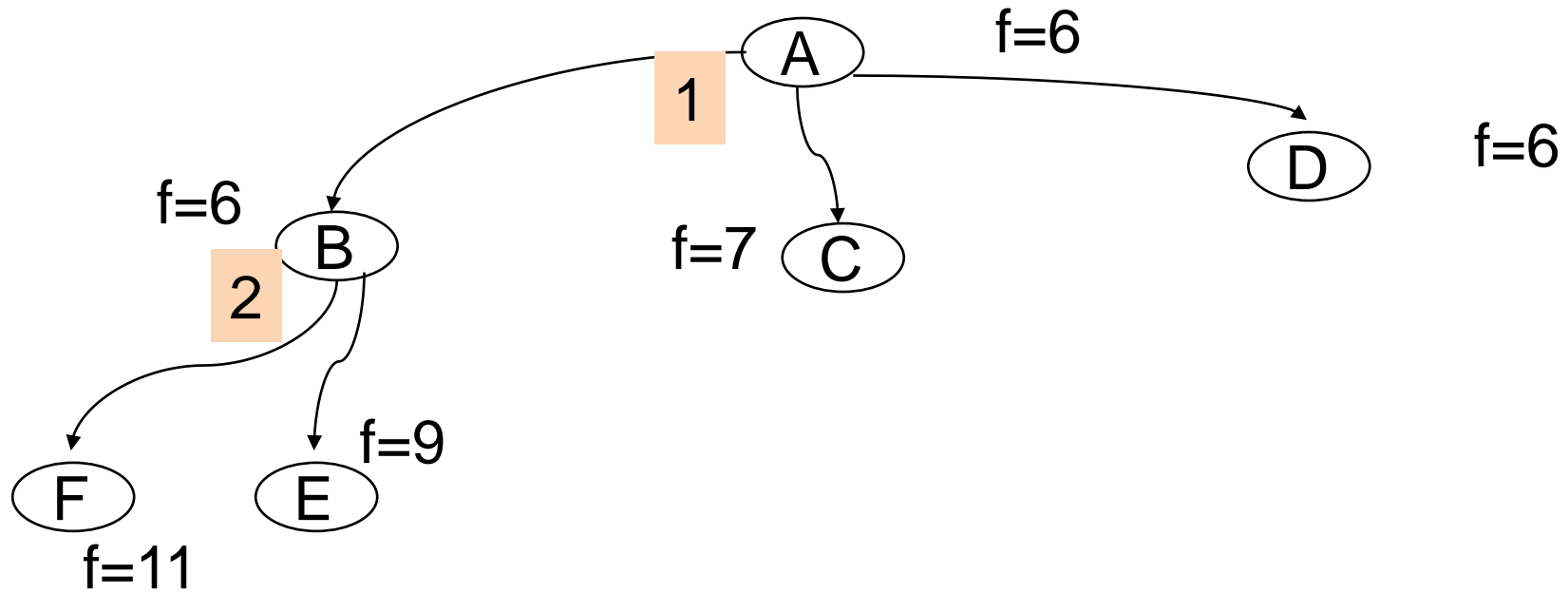
Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



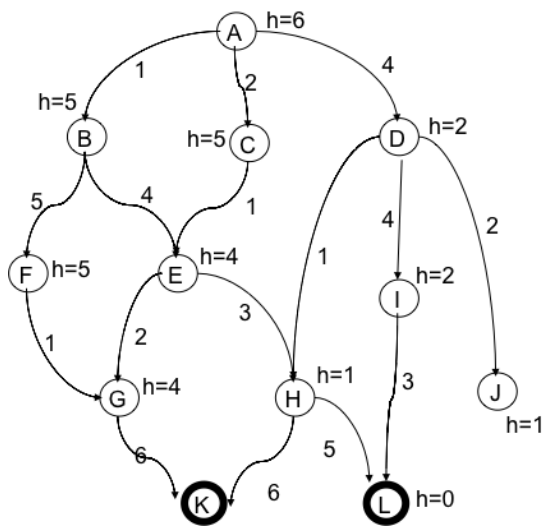
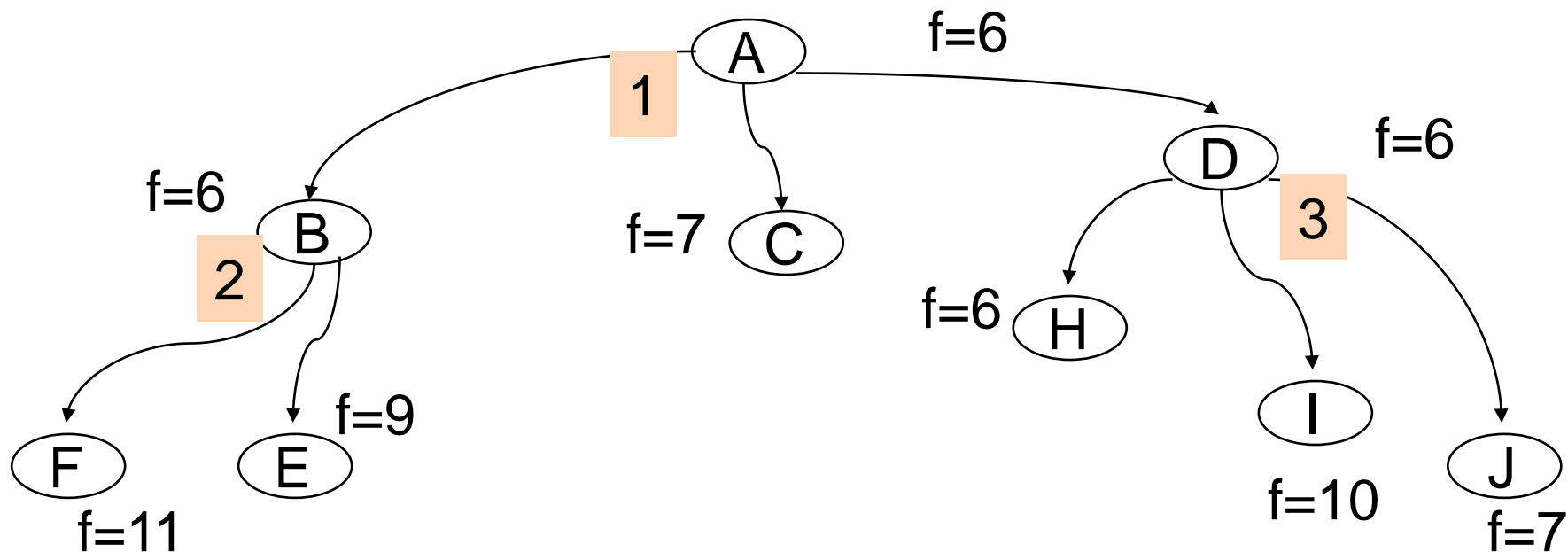
Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



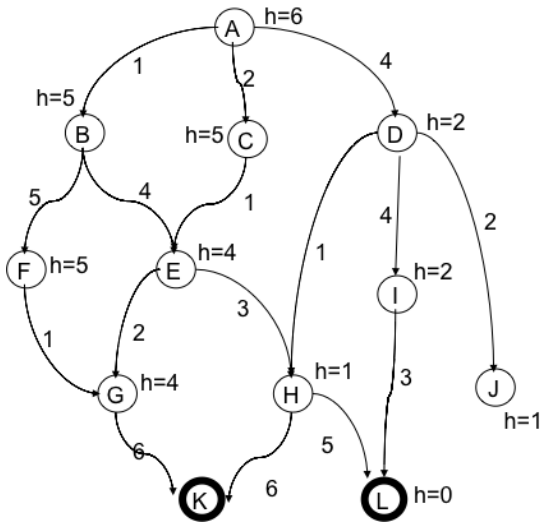
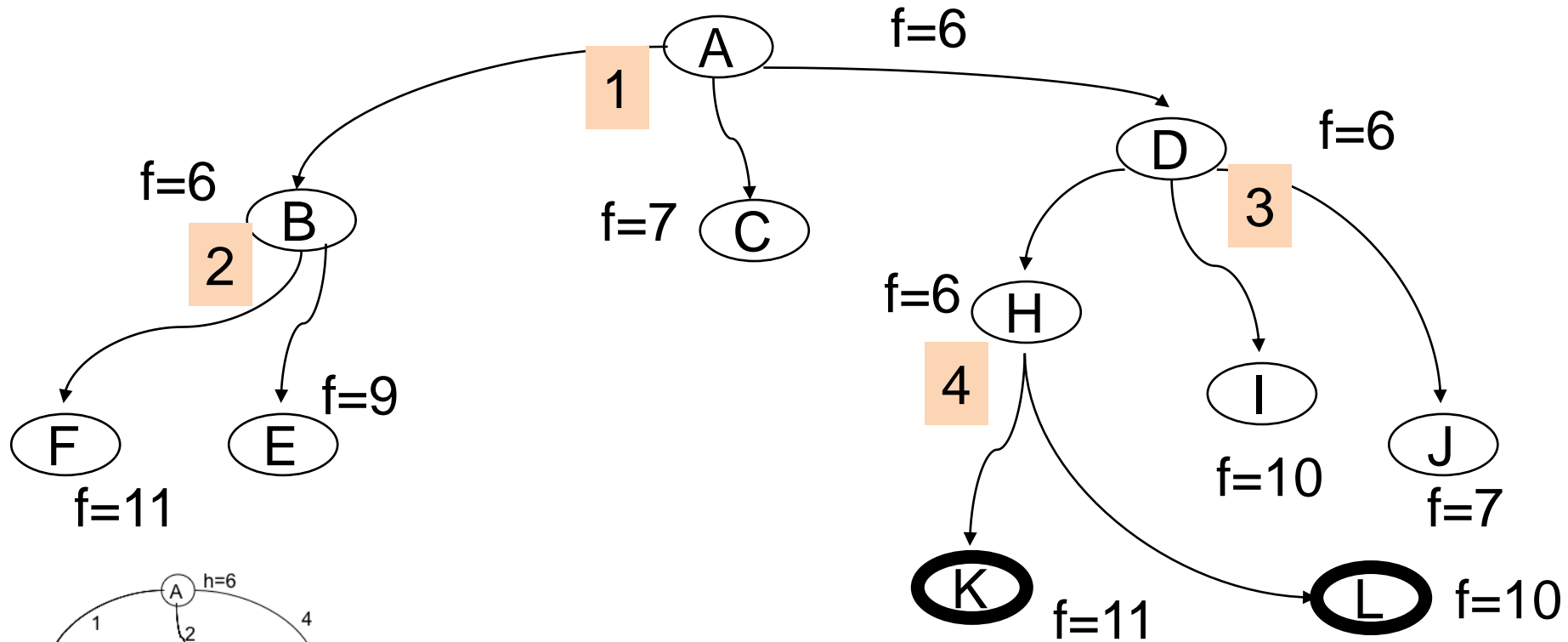
 Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



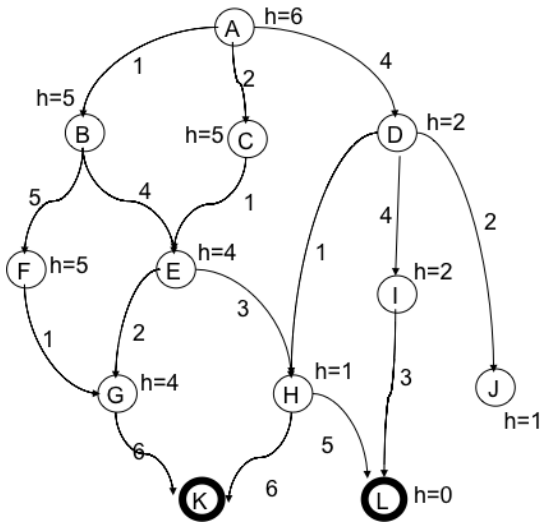
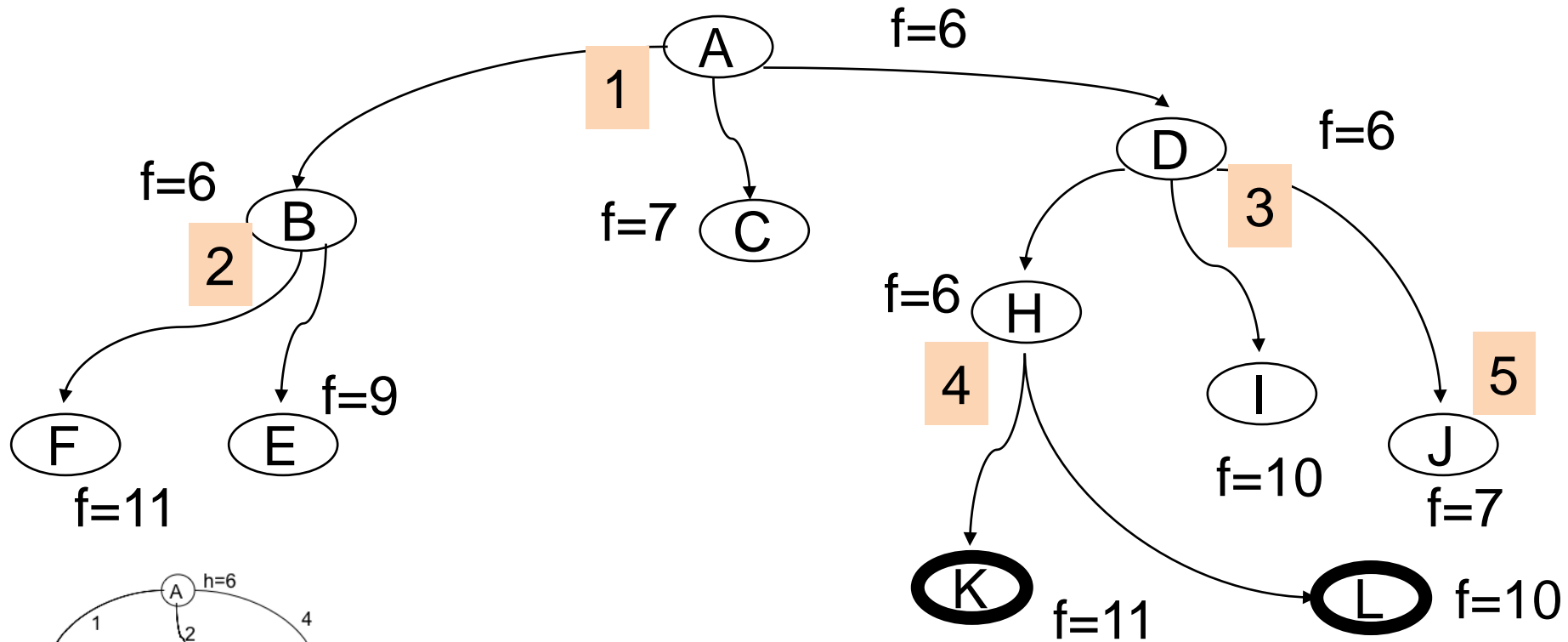
Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



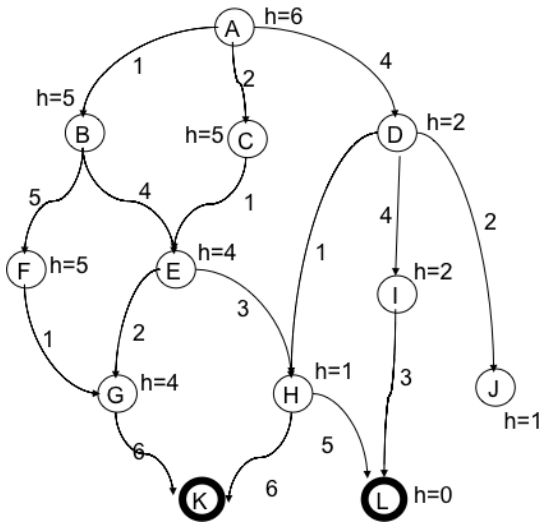
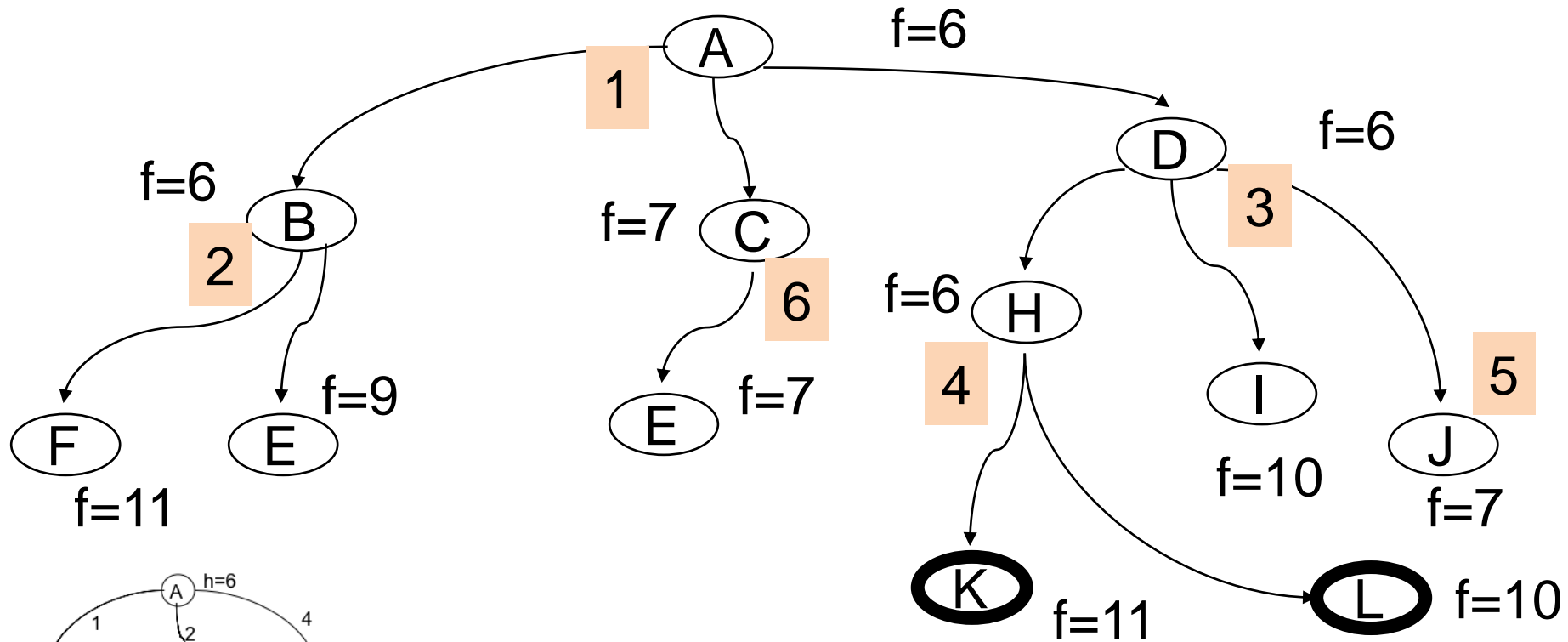
Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



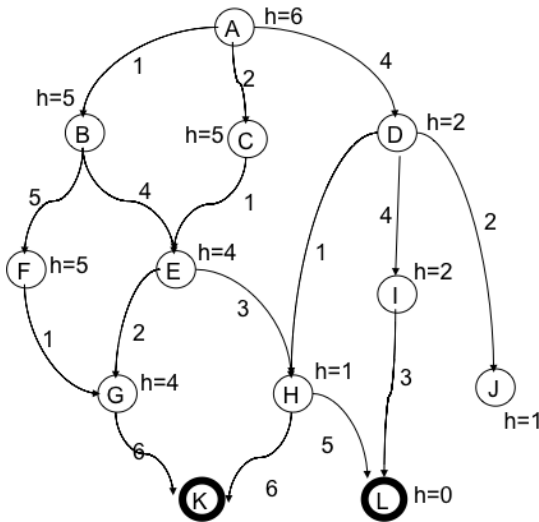
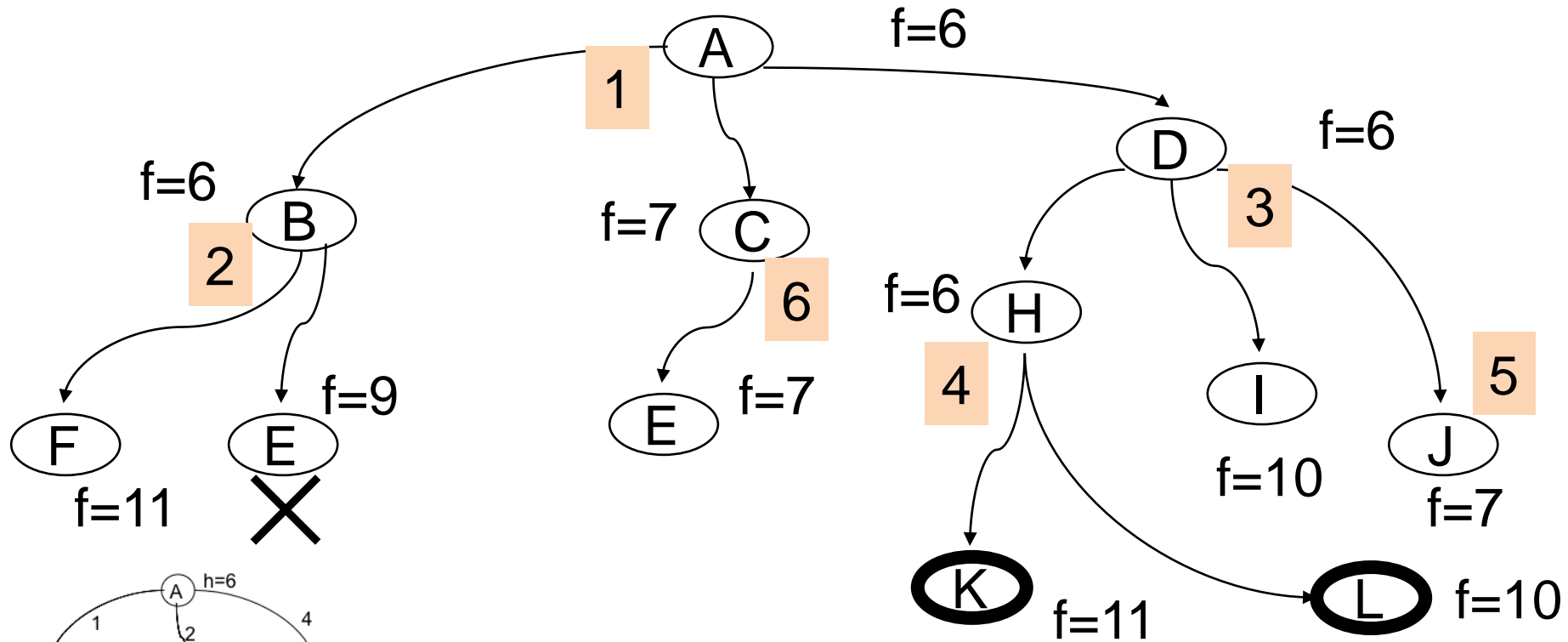
 Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



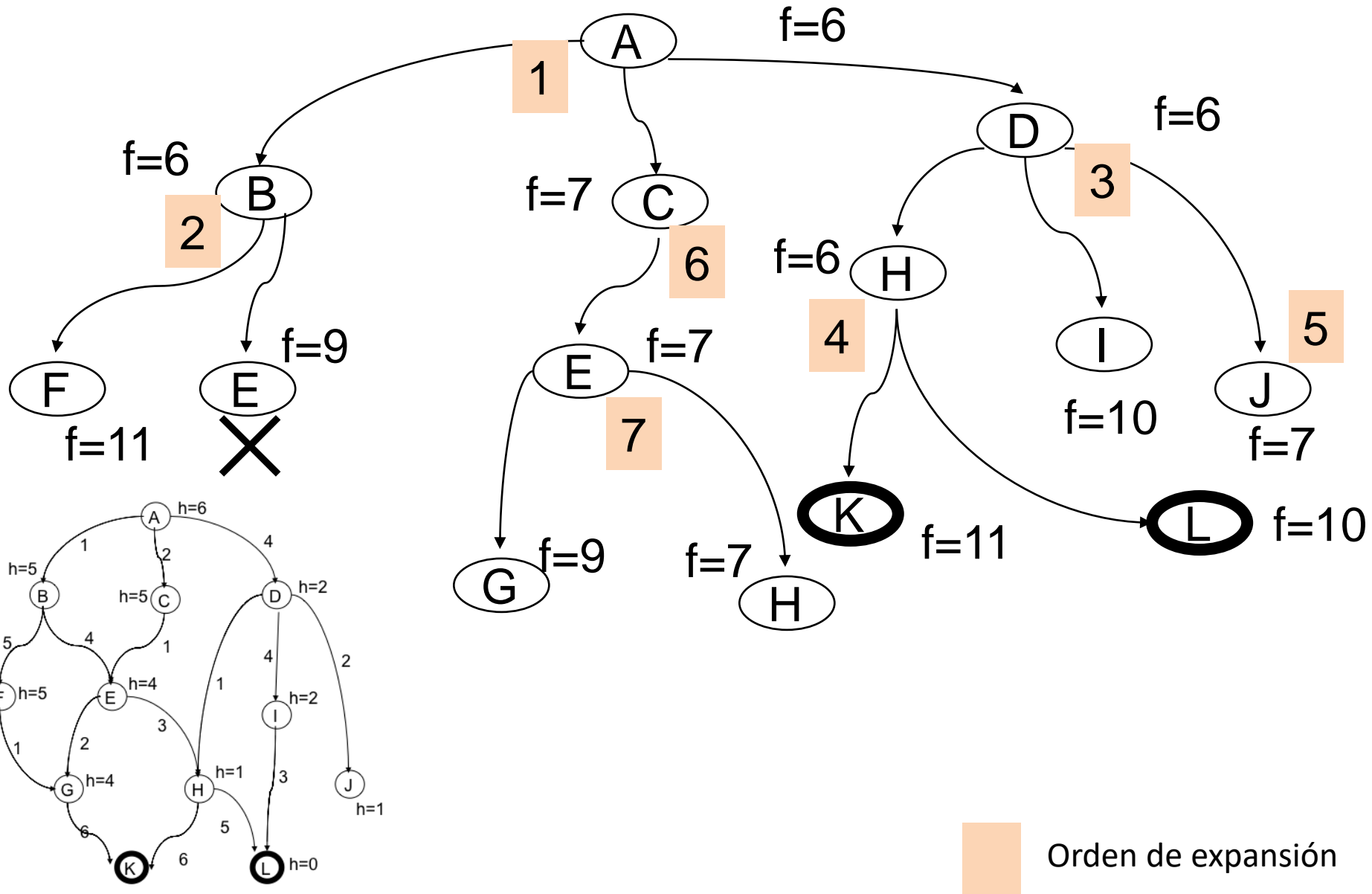
 Orden de expansión

3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos

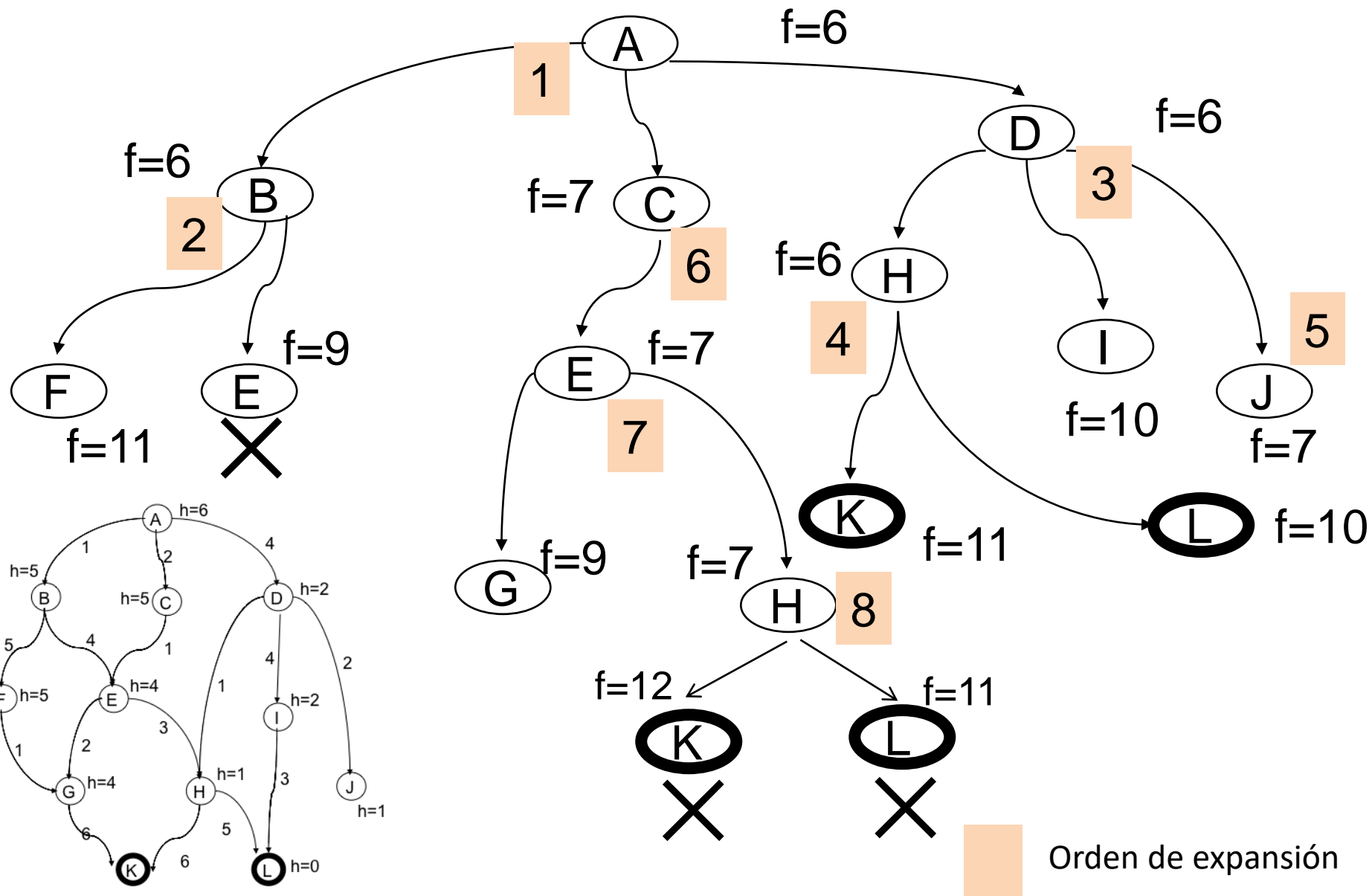


 Orden de expansión

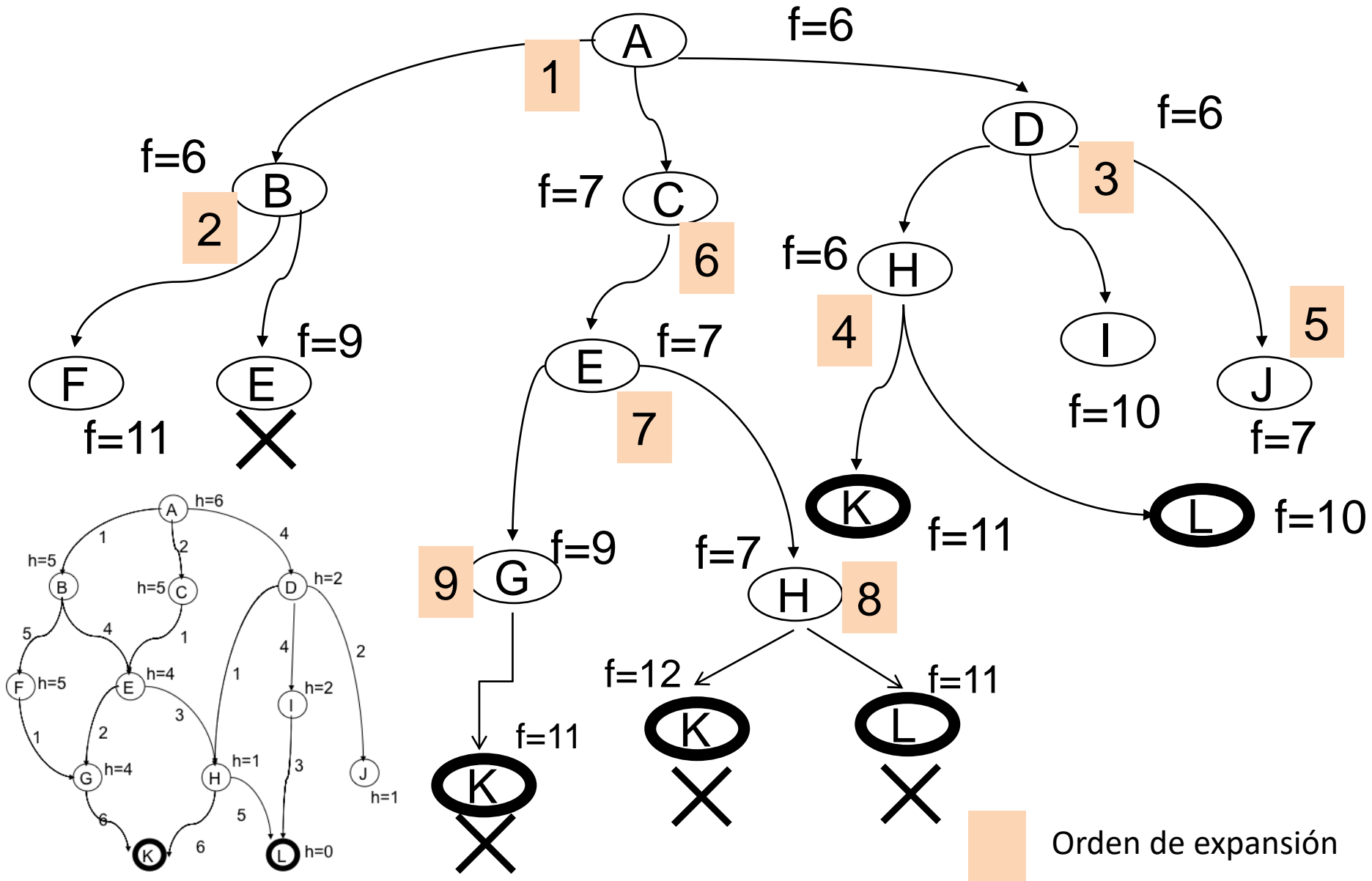
3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



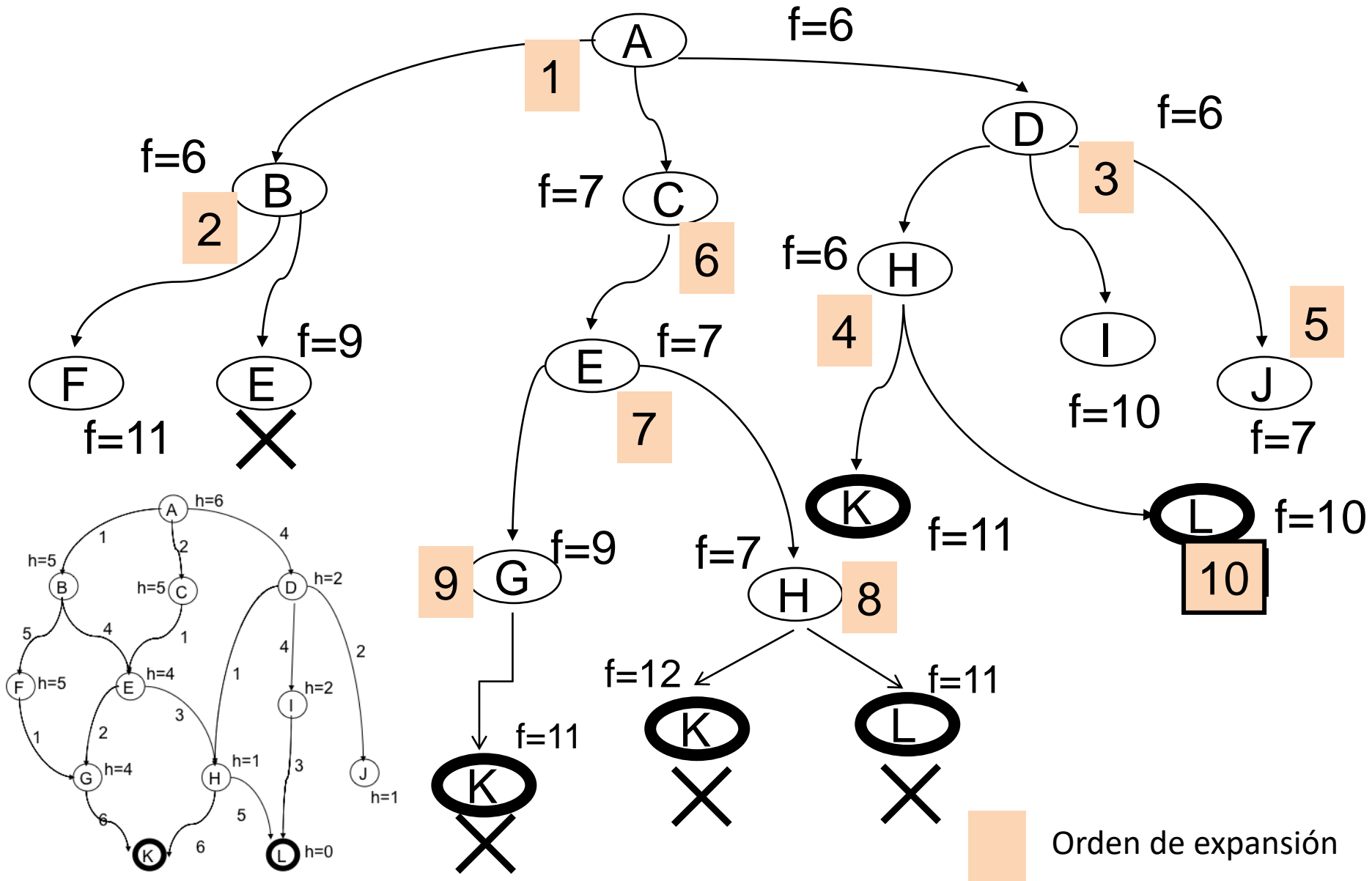
3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



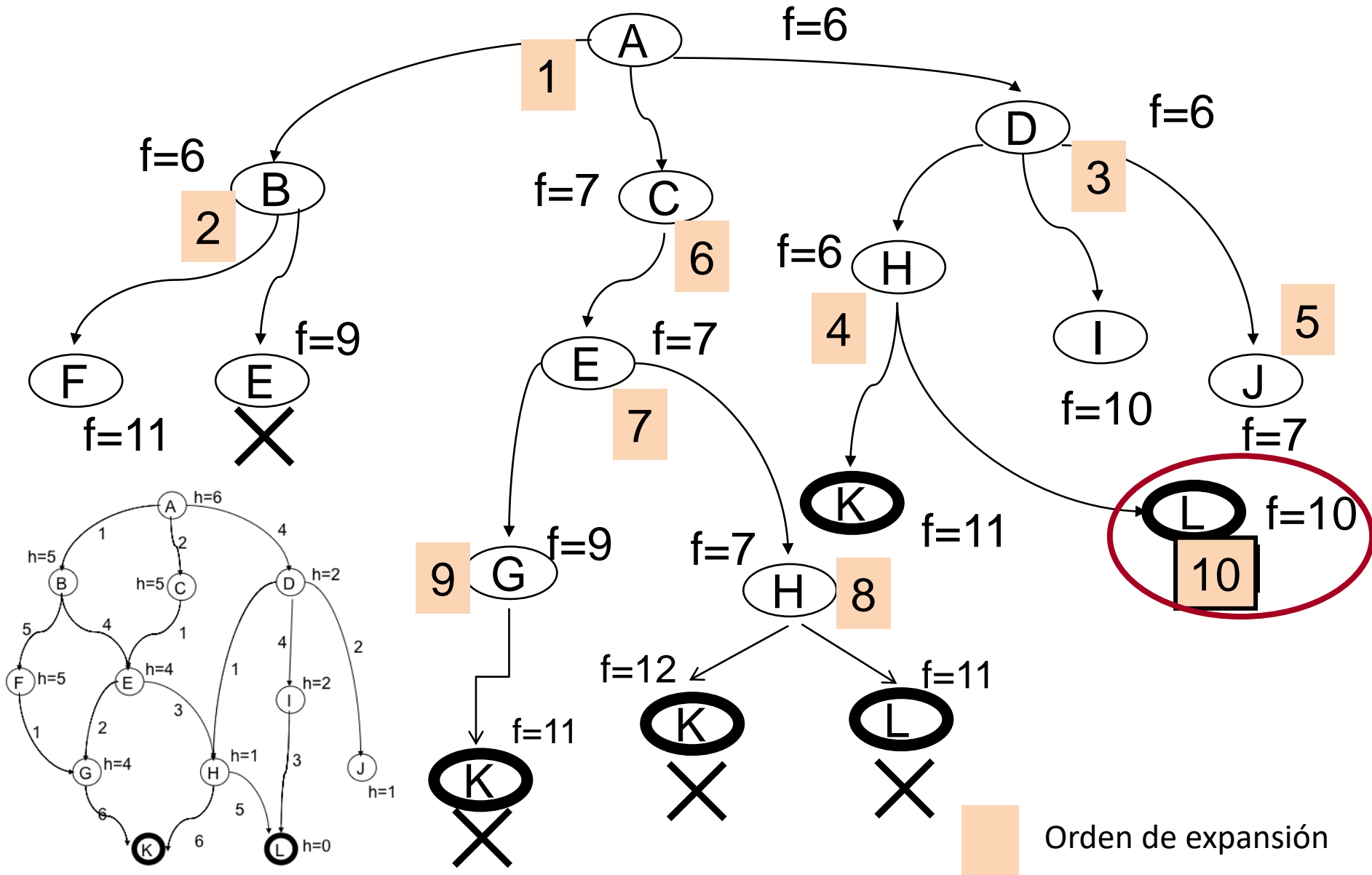
3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos

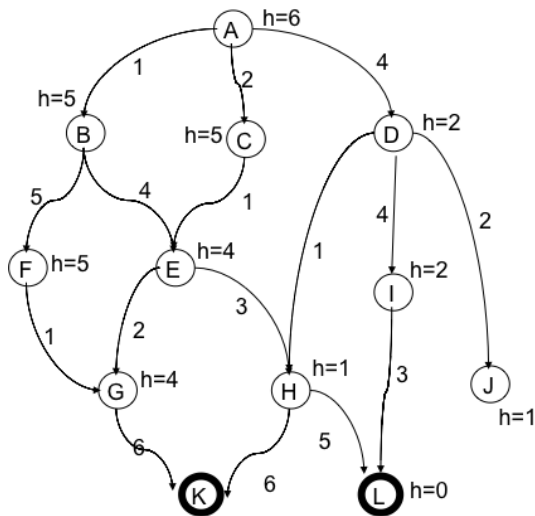


3. Búsqueda A*: versión TREE-SEARCH con control de nodos repetidos



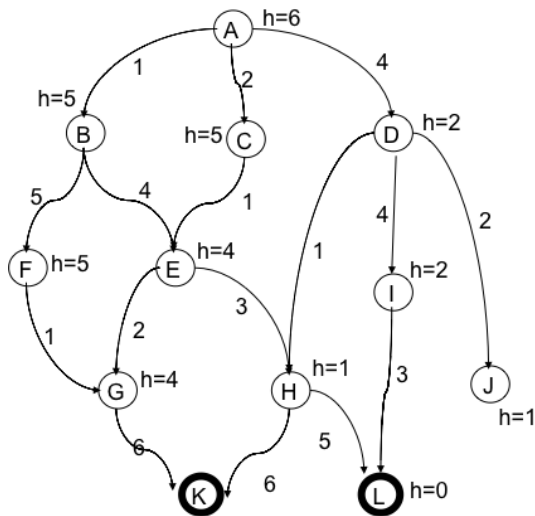
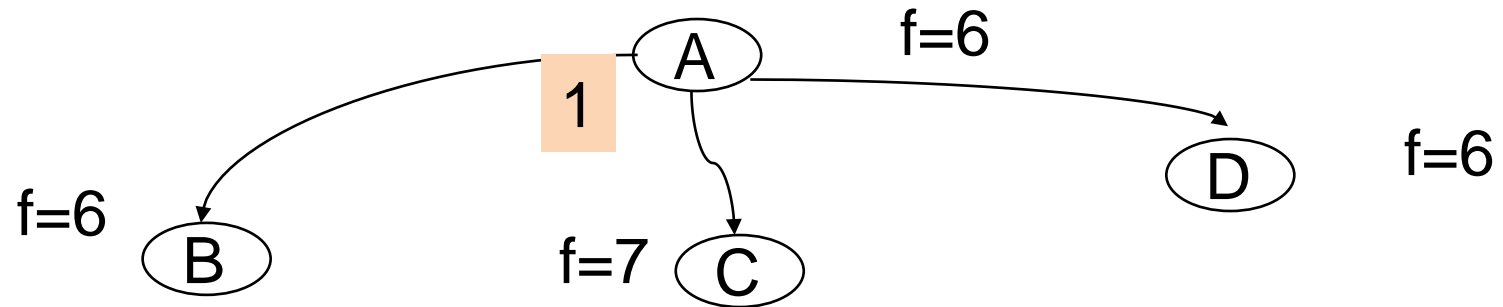
3. Búsqueda A*: versión GRAPH-SEARCH

A



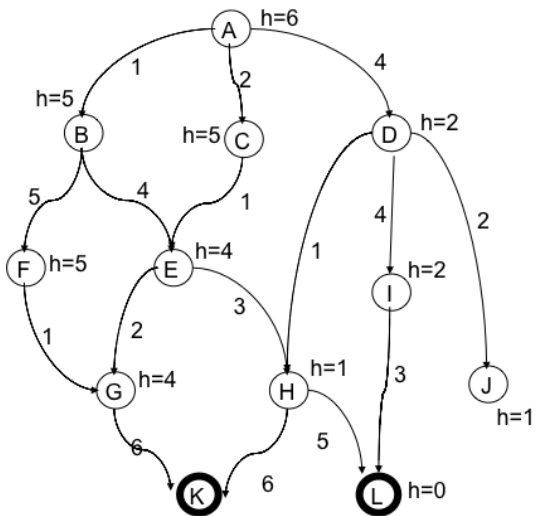
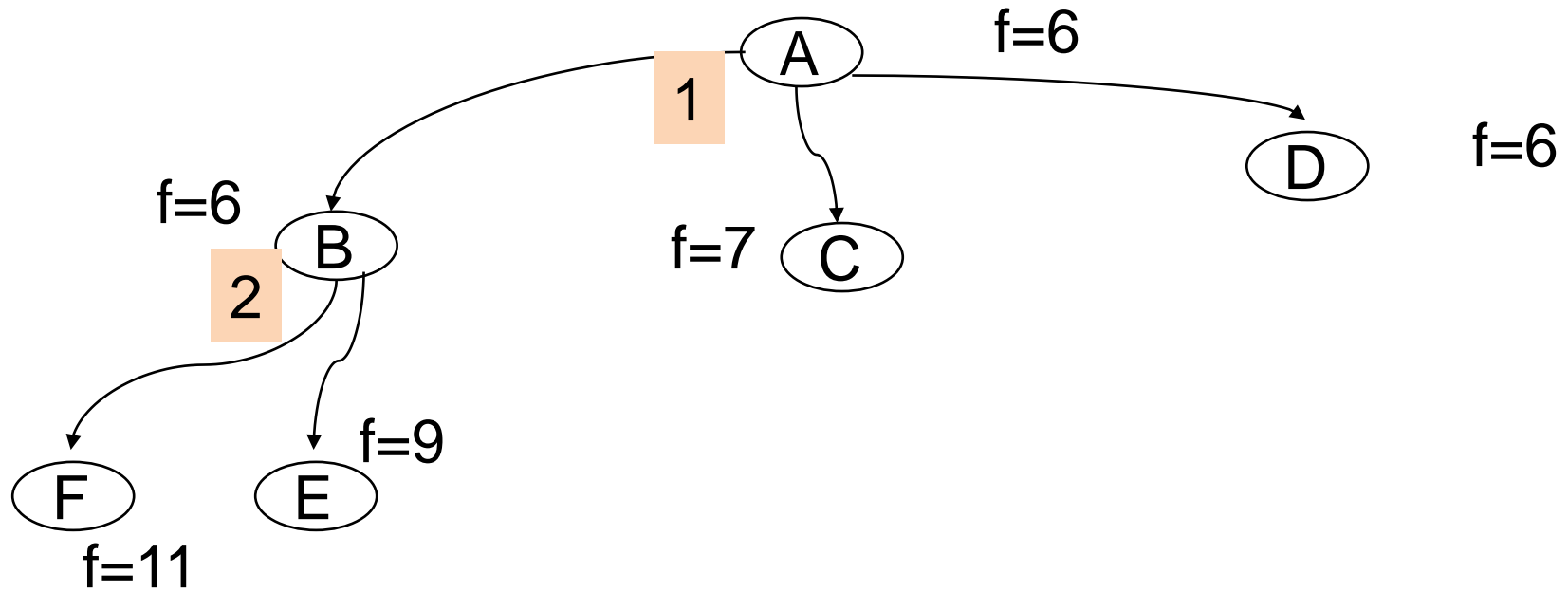
Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH



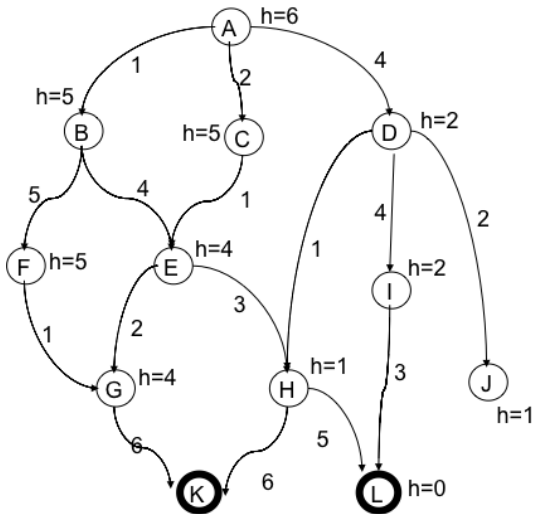
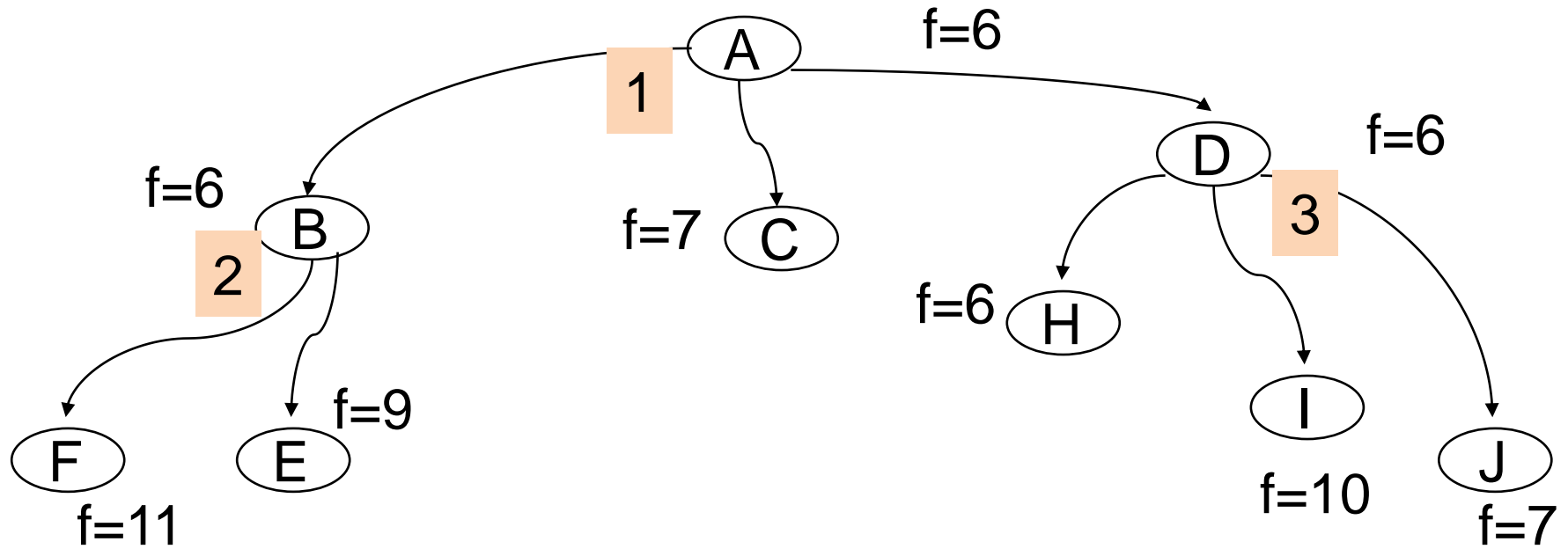
Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH



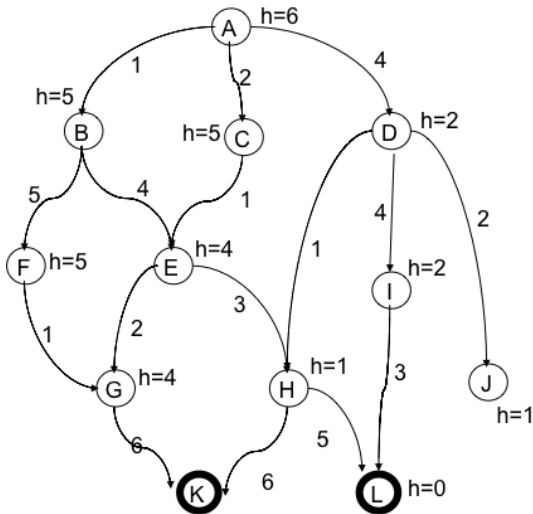
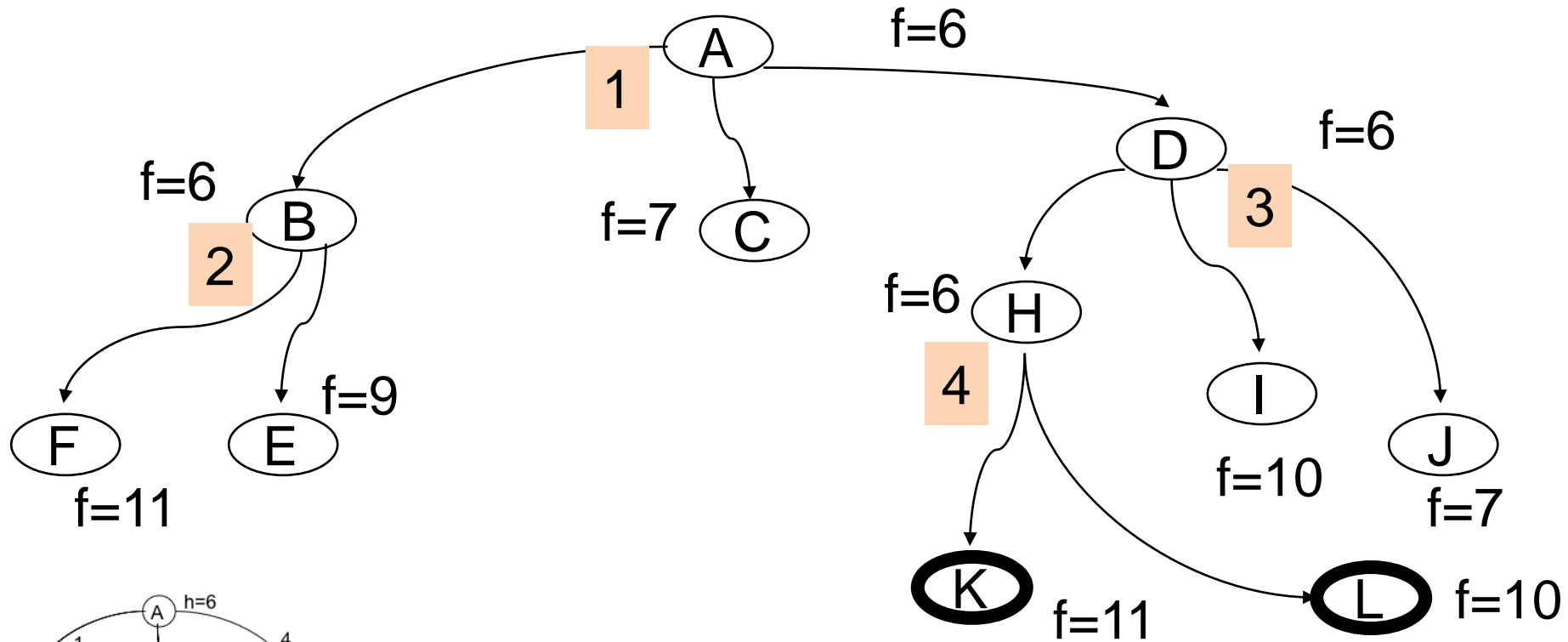
Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH



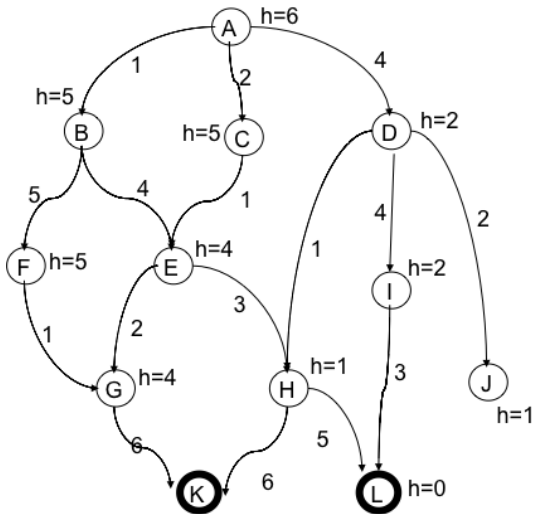
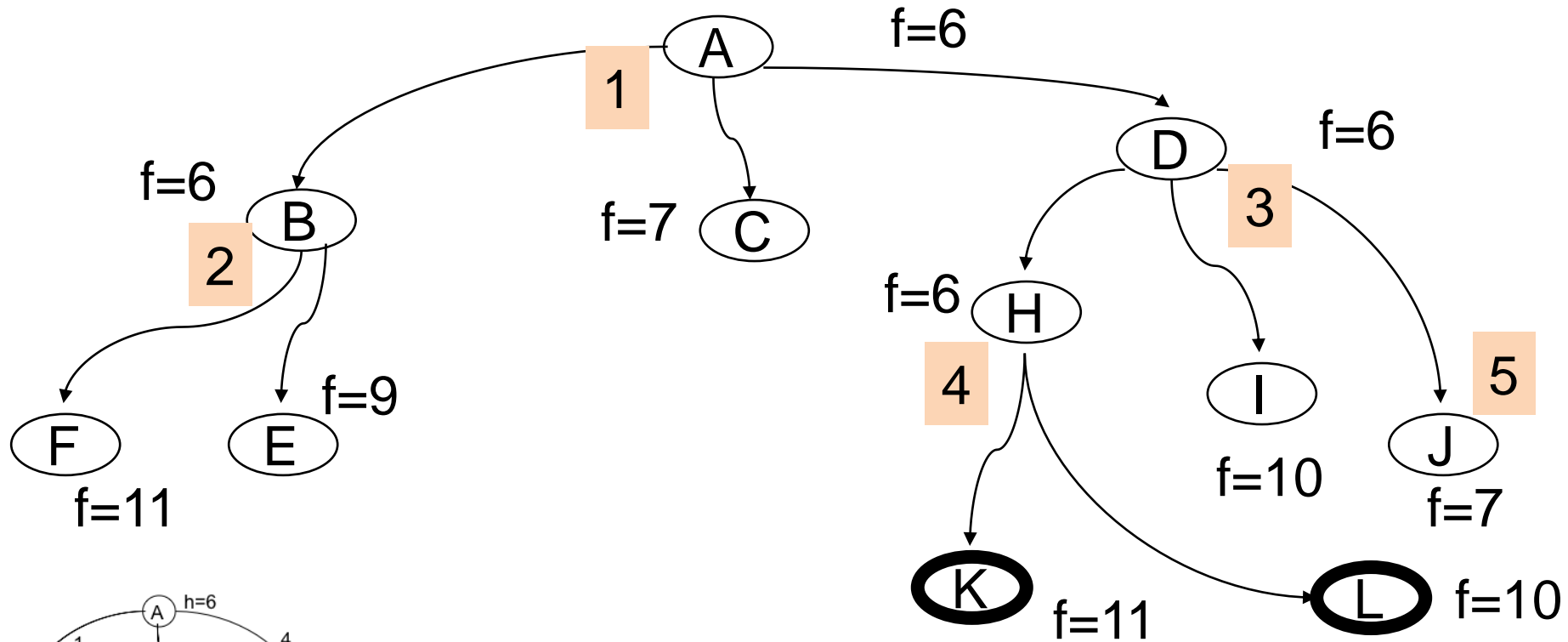
 Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH



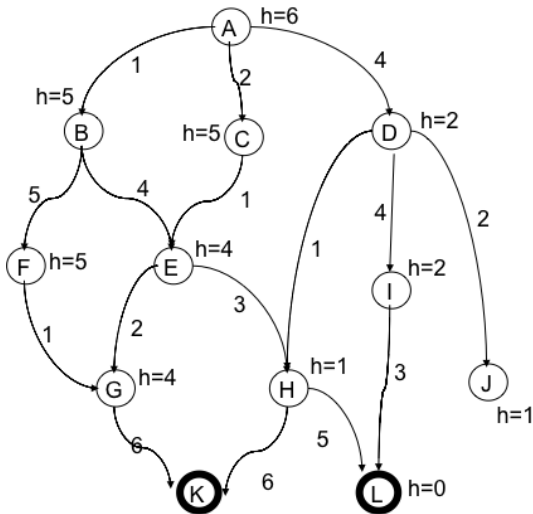
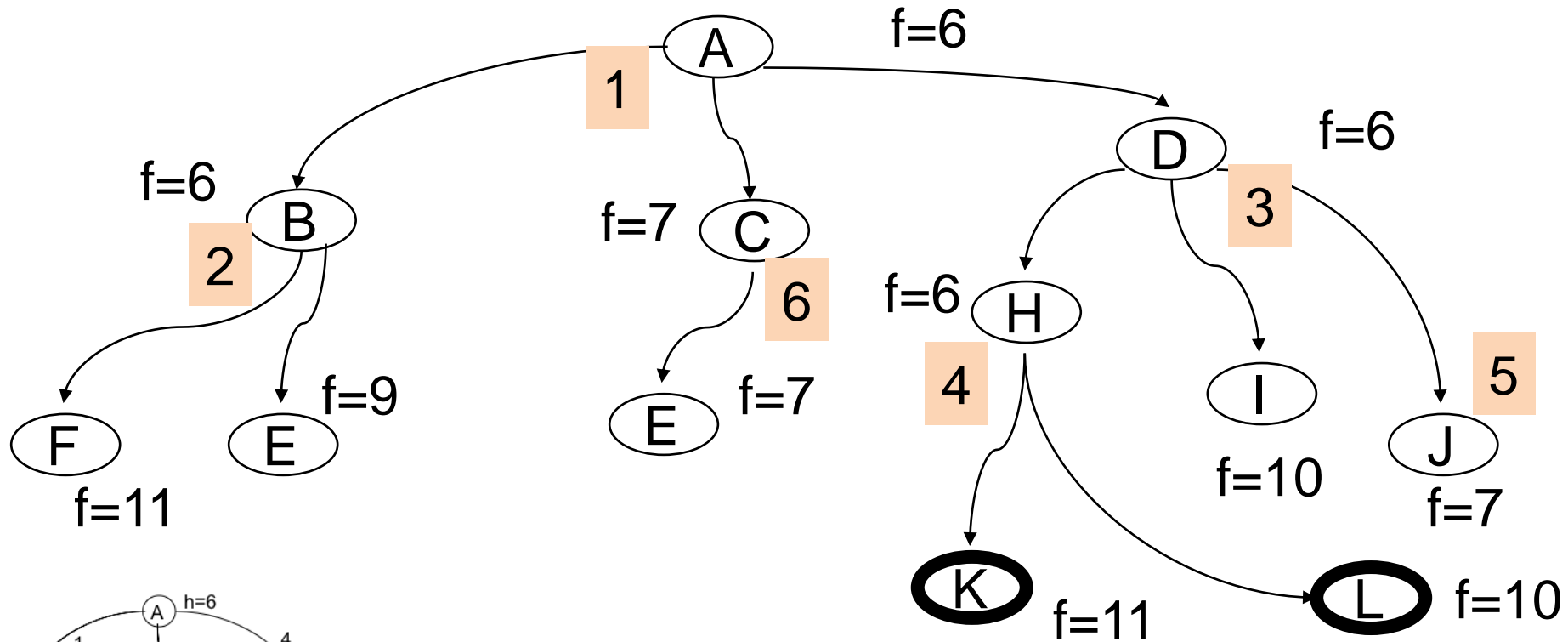
Orden de expansión


3. Búsqueda A*: versión GRAPH-SEARCH



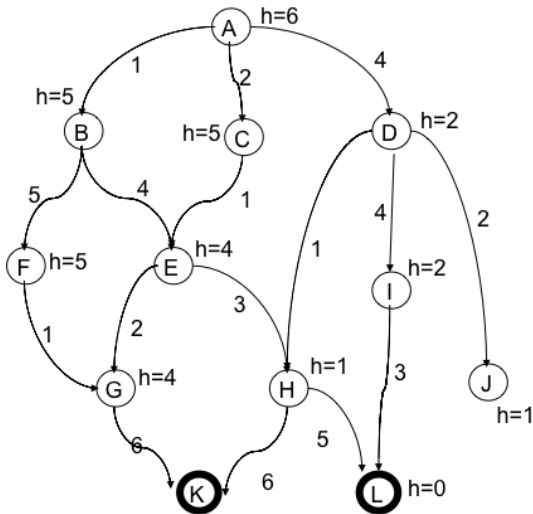
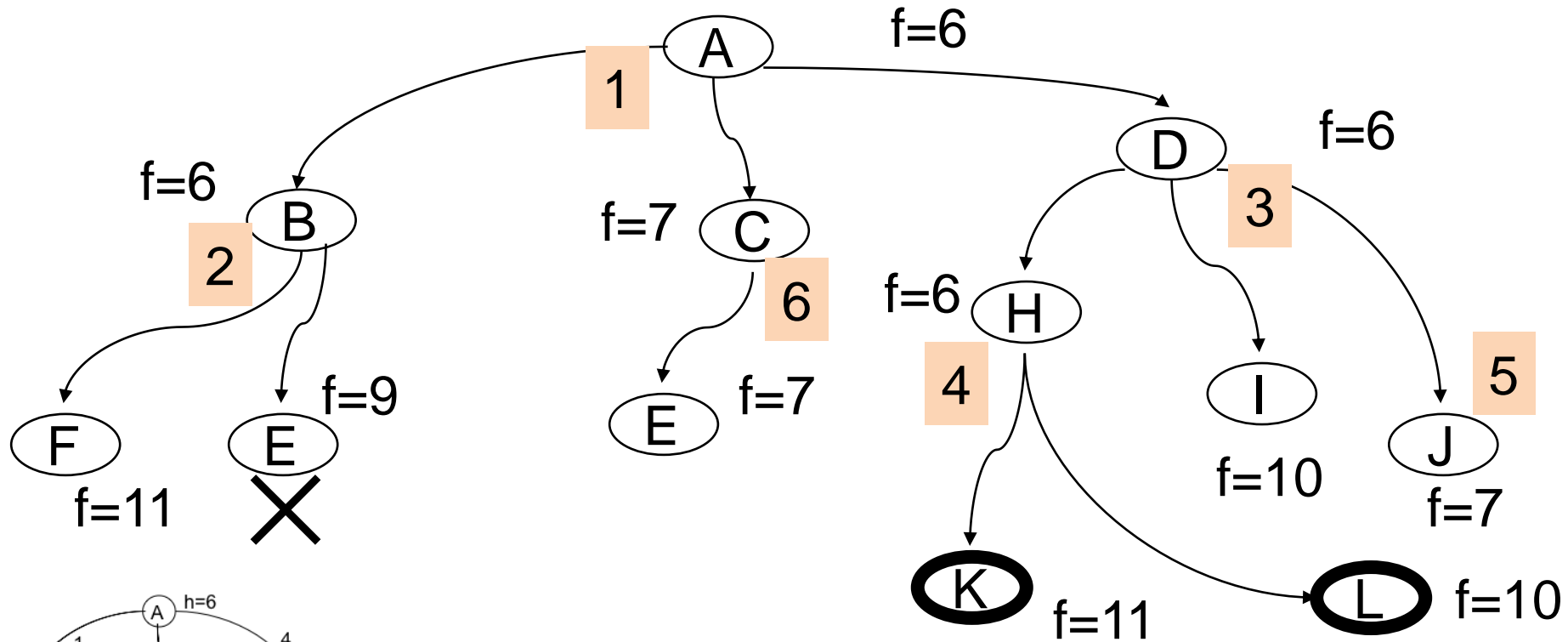
 Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH



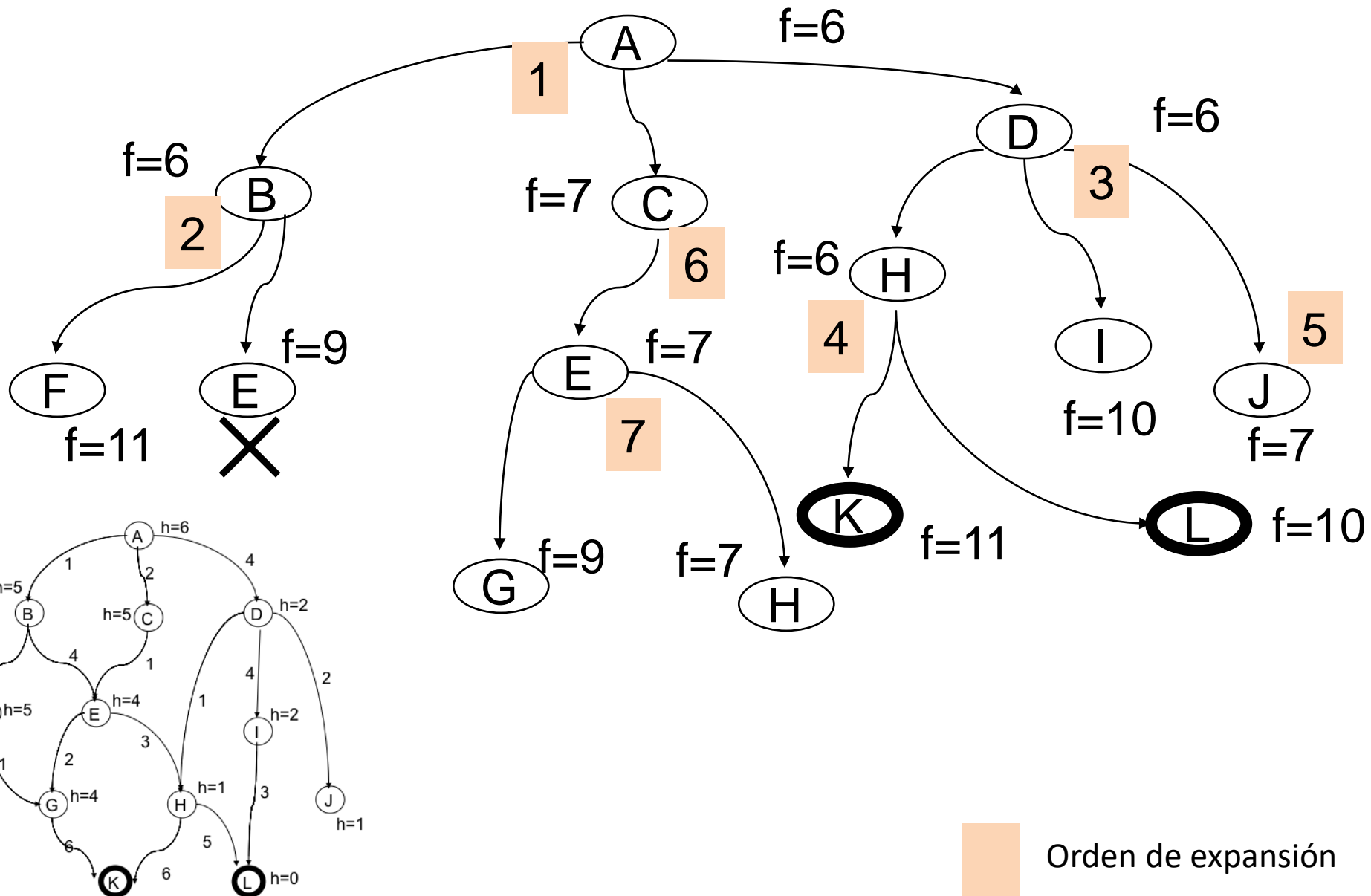
 Orden de expansión

3. Búsqueda A*: versión GRAPH-SEARCH

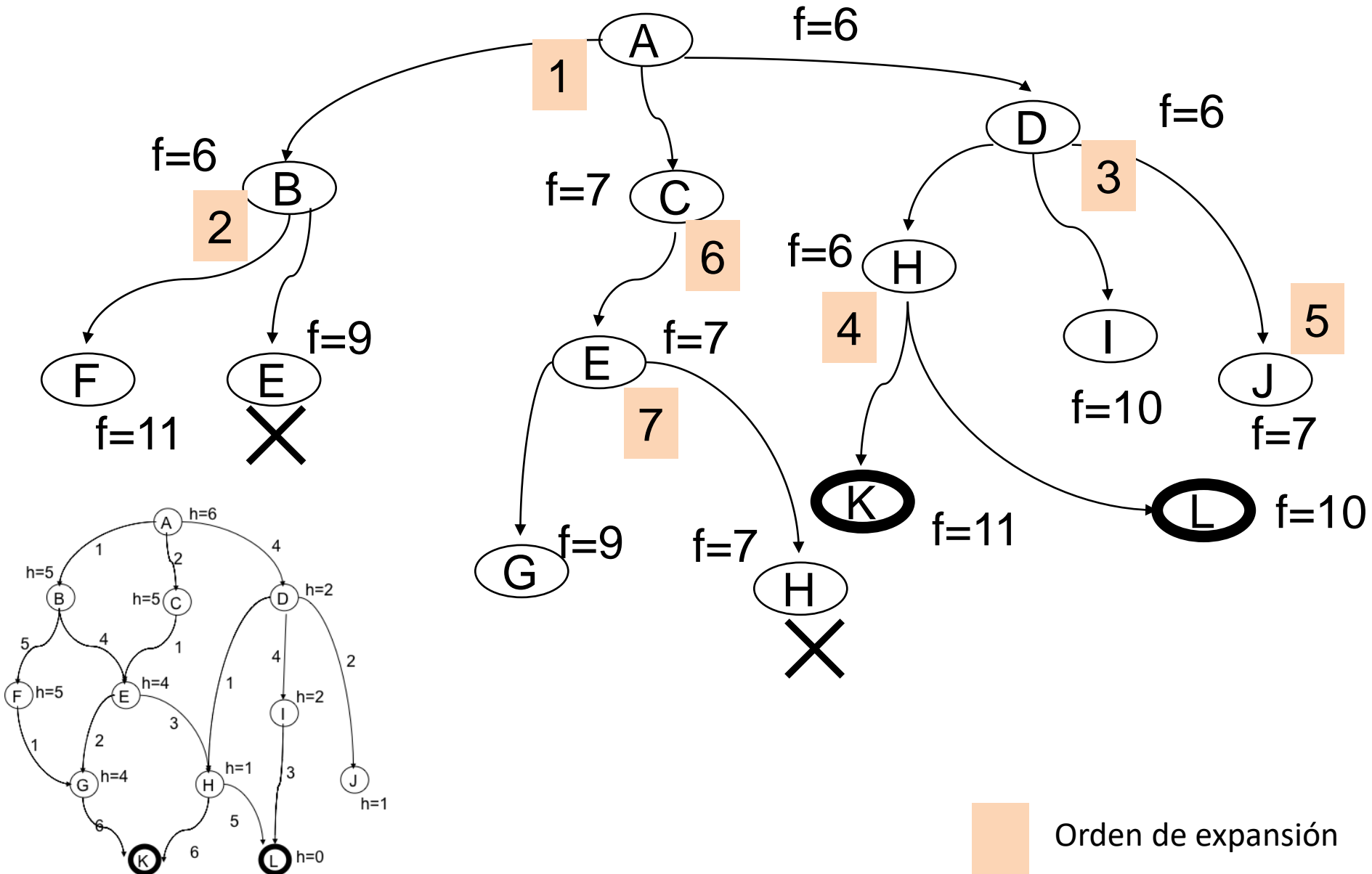


 Orden de expansión

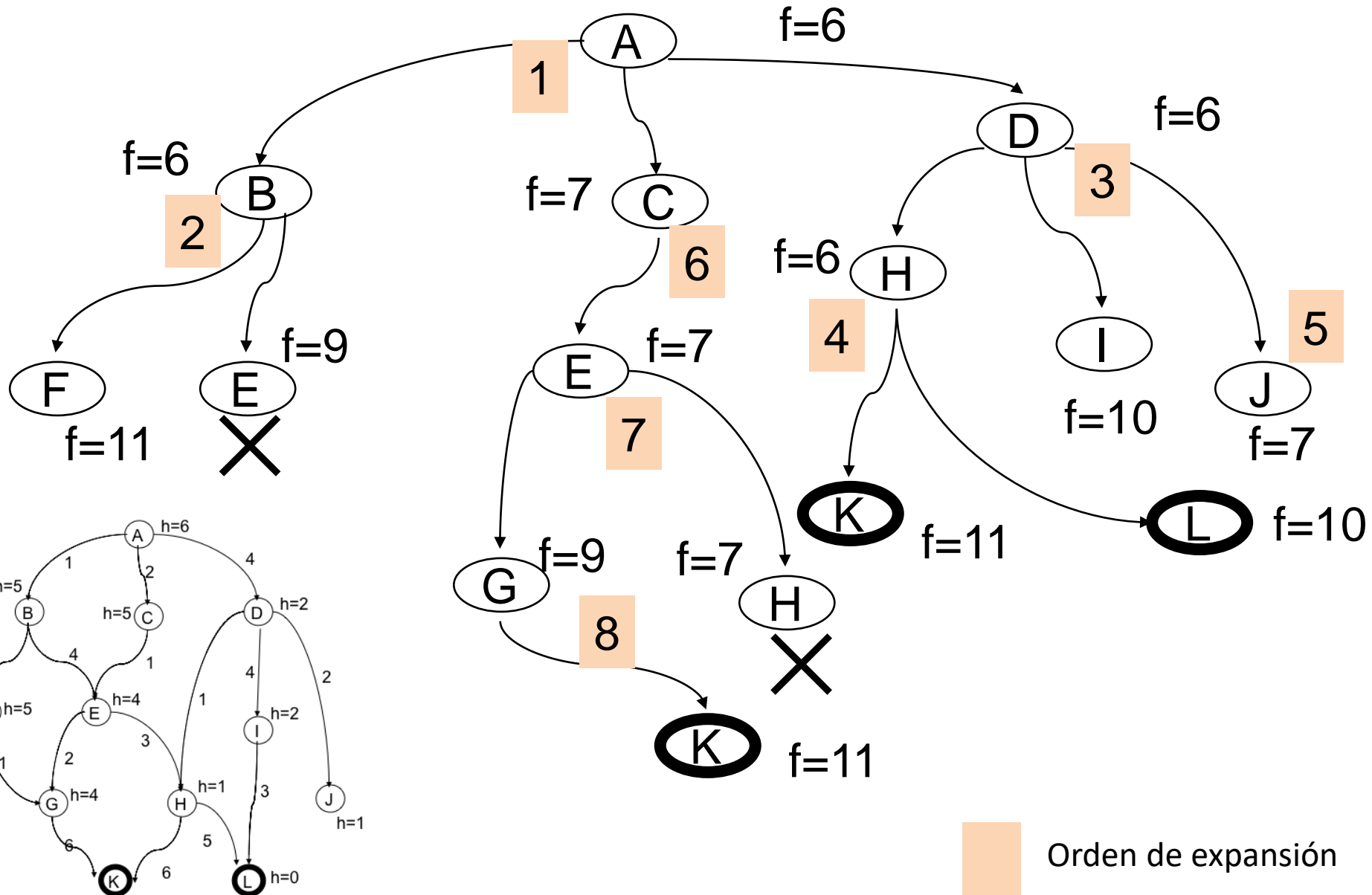
3. Búsqueda A*: versión GRAPH-SEARCH



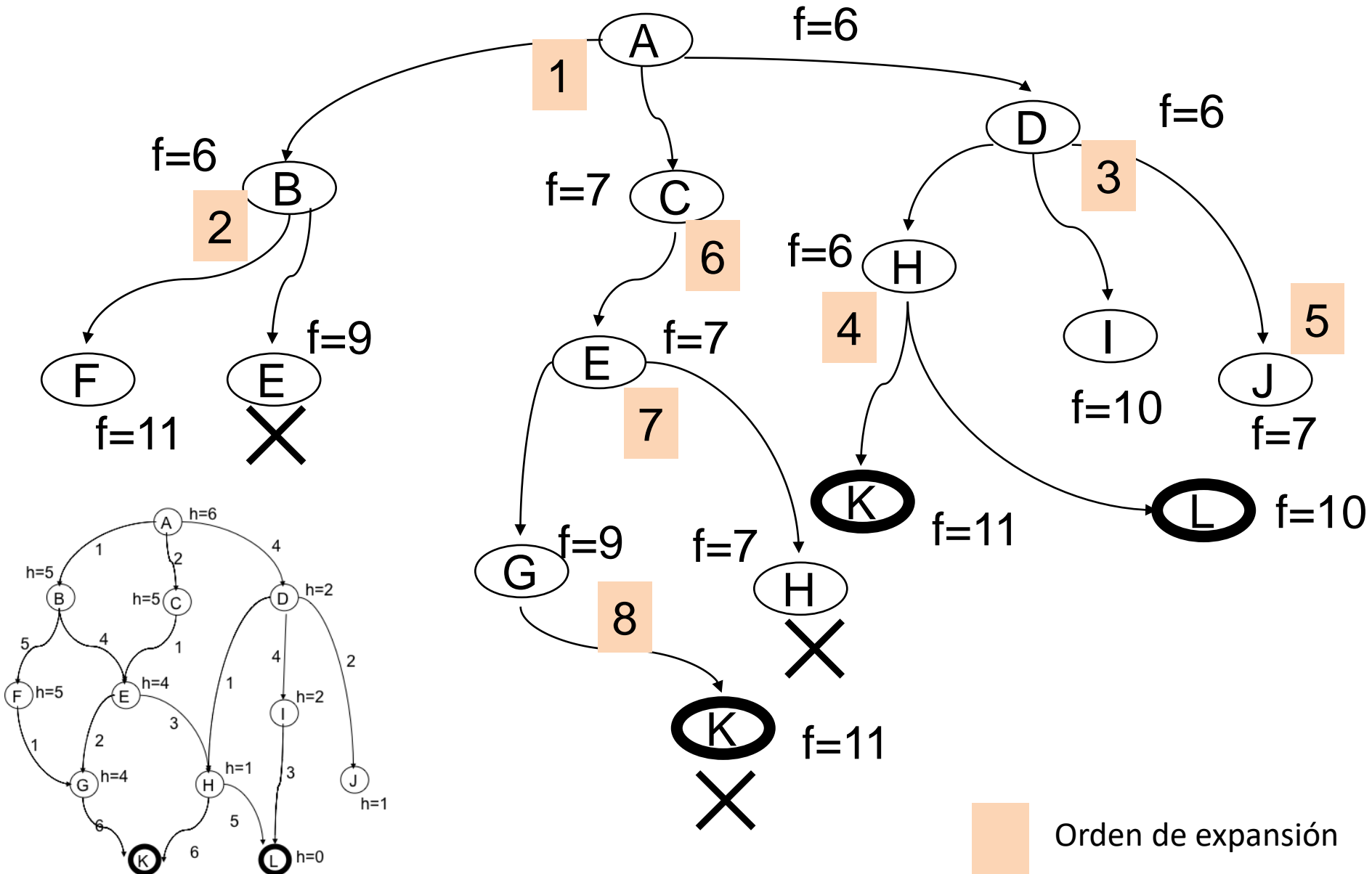
3. Búsqueda A*: versión GRAPH-SEARCH



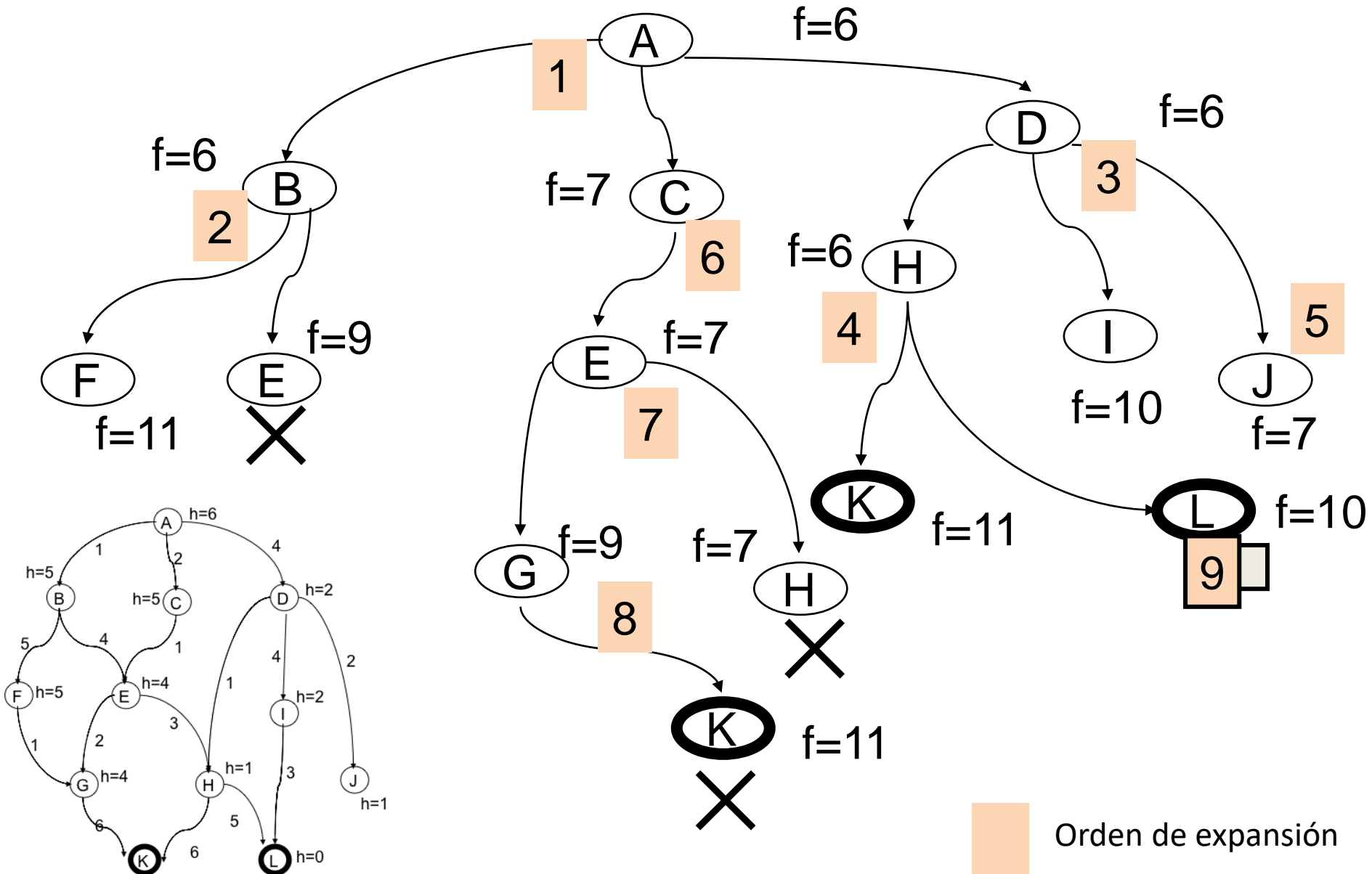
3. Búsqueda A*: versión GRAPH-SEARCH



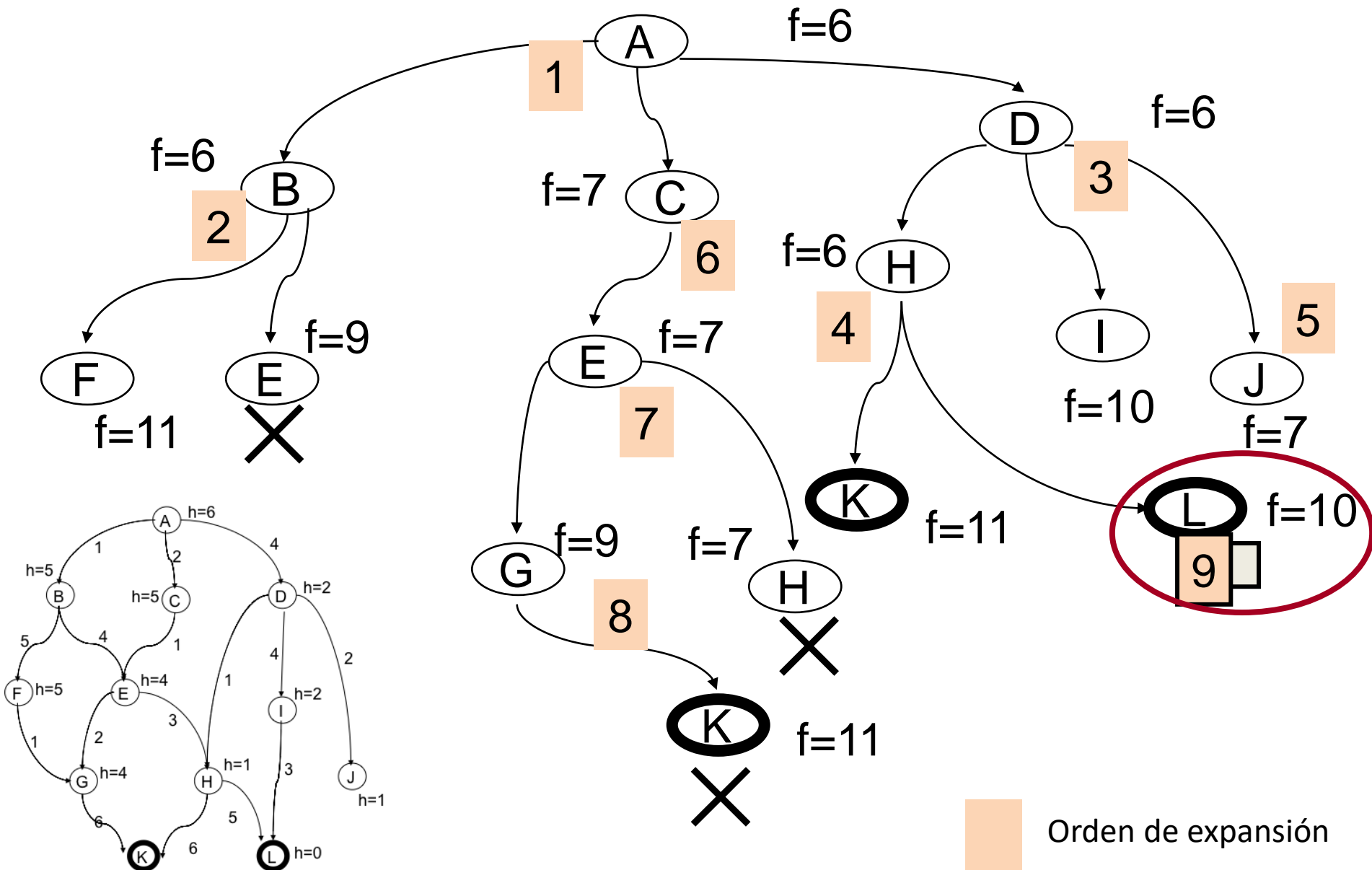
3. Búsqueda A*: versión GRAPH-SEARCH



3. Búsqueda A*: versión GRAPH-SEARCH



3. Búsqueda A*: versión GRAPH-SEARCH



3. Búsqueda A*: comparación y análisis

Dadas dos **funciones heurísticas admisibles** $h_1(n)$ y $h_2(n)$

Si $h_2(n) \geq h_1(n) \forall n$ entonces h_2 **domina** a h_1 (h_2 **es más informada** que h_1)

y

$f_2(n) = g(n) + h_2(n)$ nunca expandirá más nodos que $f_1(n) = g(n) + h_1(n)$

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real

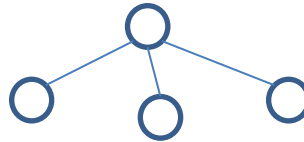
h^*

$h(n)$

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real



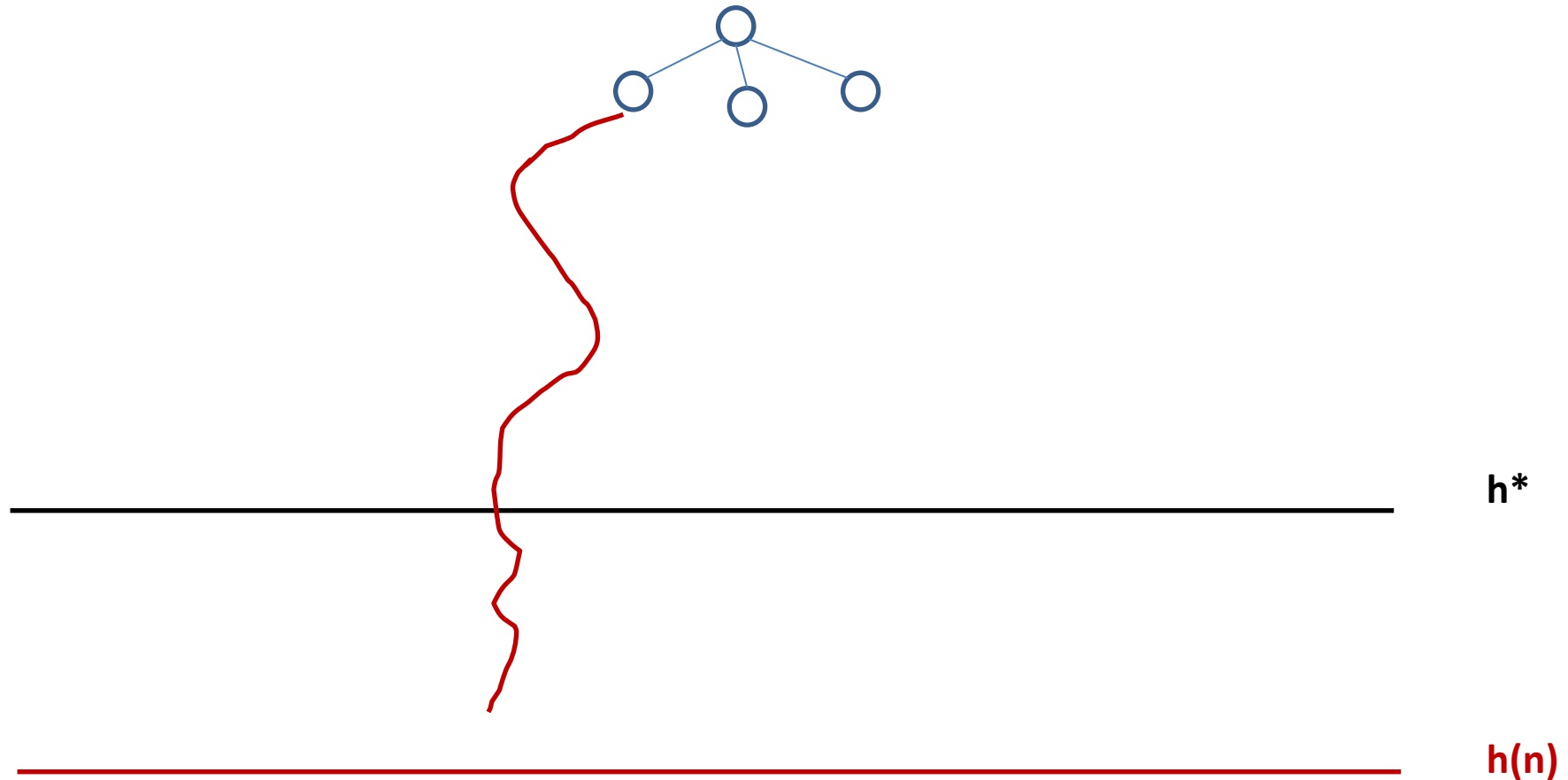
h^*

$h(n)$

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

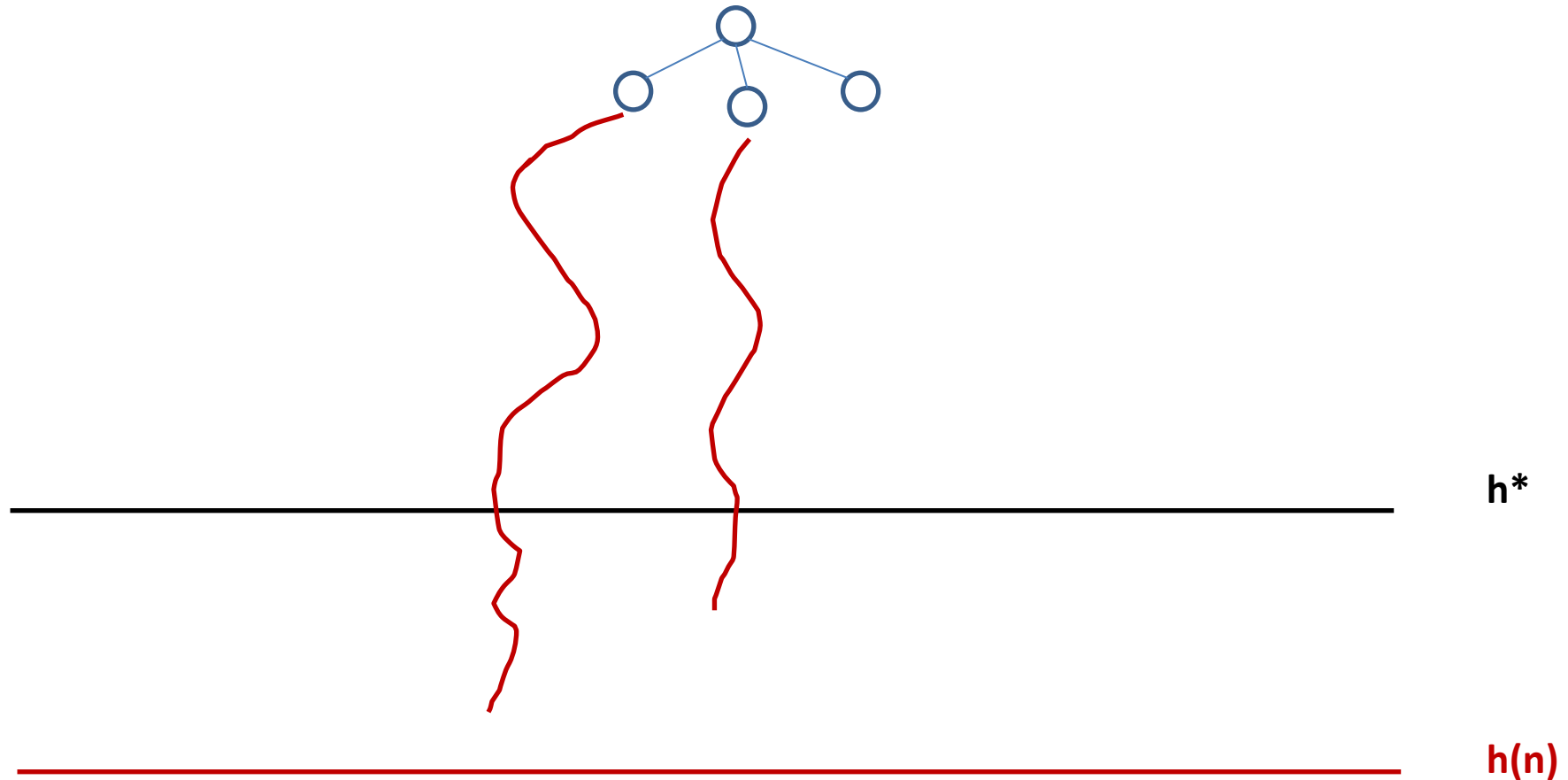
$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real



3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

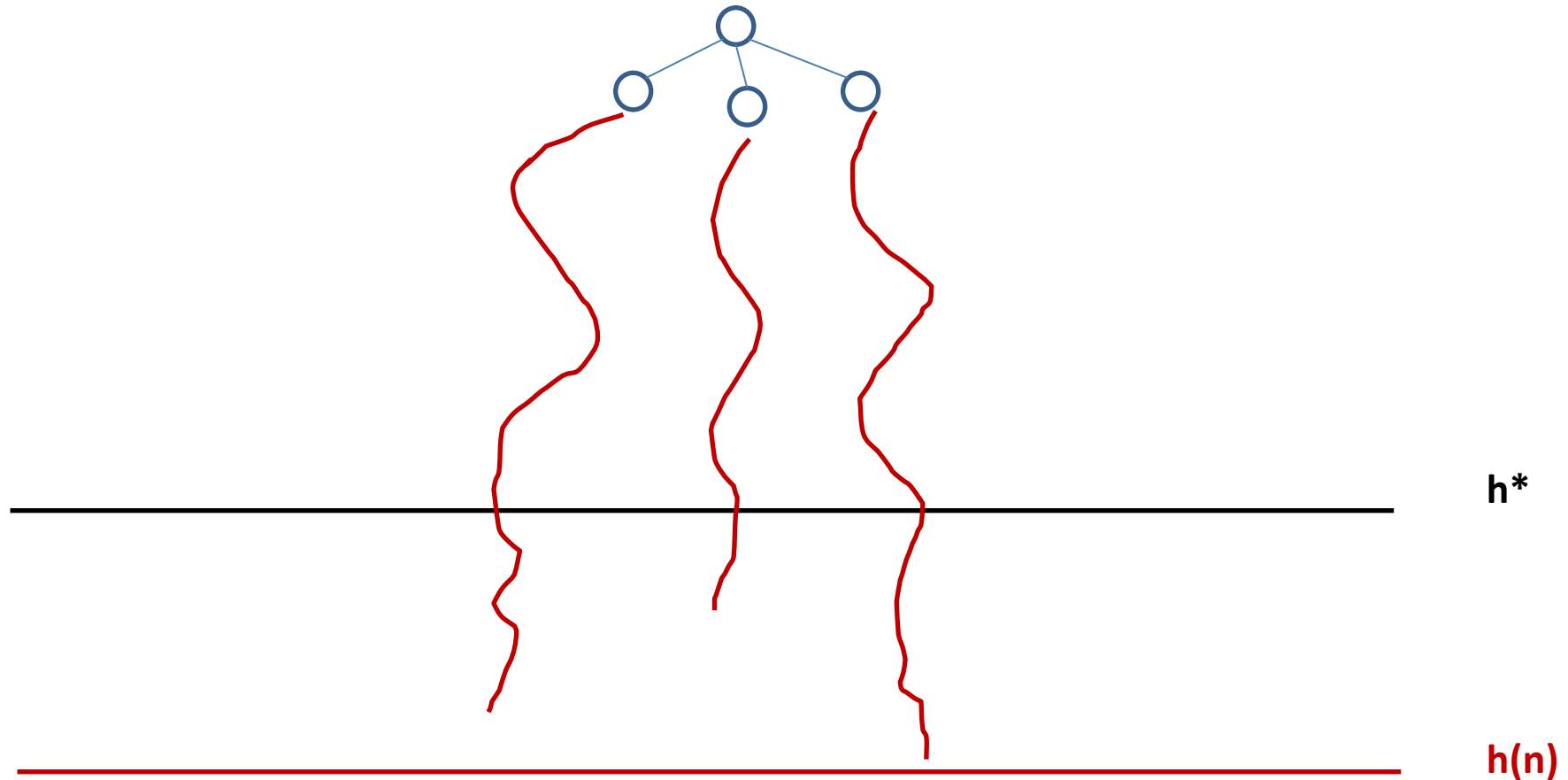
$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real



3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

$\exists n \ h(n) > h^*(n) \rightarrow h(n)$ sobreestima el coste real



3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

El componente $h(n)$ en $f(n)=g(n)+h(n)$ tiene más peso que $g(n)$

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

El componente $h(n)$ en $f(n)=g(n)+h(n)$ tiene más peso que $g(n)$

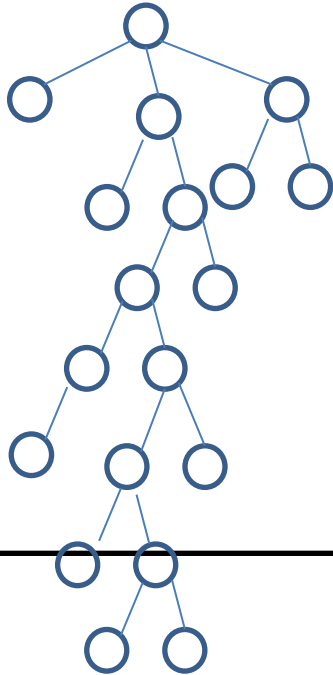
Cuanto más peso tiene $h(n)$ con respecto a $g(n)$ en una función de evaluación, más similar es la búsqueda a un algoritmo **voraz** ($f(n)=h(n)$)

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

El componente $h(n)$ en $f(n)=g(n)+h(n)$ tiene más peso que $g(n)$

Cuanto más peso tiene $h(n)$ con respecto a $g(n)$ en una función de evaluación, más similar es la búsqueda a un algoritmo **voraz** ($f(n)=h(n)$)



h*

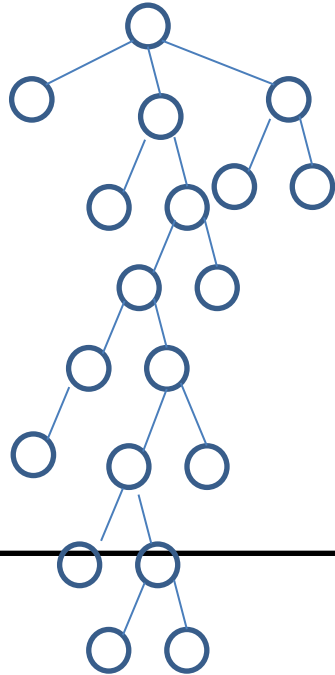
h(n)

3. Búsqueda A*: comparación y análisis

¿Y qué podemos decir de las funciones **heurísticas no admisibles**?

El componente $h(n)$ en $f(n)=g(n)+h(n)$ tiene más peso que $g(n)$

Cuanto más peso tiene $h(n)$ con respecto a $g(n)$ en una función de evaluación, más similar es la búsqueda a un algoritmo **voraz** ($f(n)=h(n)$)



No hay garantía de solución óptima!

h^*

$h(n)$

3. Búsqueda A*: comparación y análisis

¿Podemos decir entonces que una **heurística no admisible** genera menos nodos que una heurística admisible?

3. Búsqueda A*: comparación y análisis

¿Podemos decir entonces que una **heurística no admisible** genera menos nodos que una heurística admisible?

Puede que sí

Puede que no

3. Búsqueda A*: comparación y análisis

¿Podemos decir entonces que una **heurística no admisible** genera menos nodos que una heurística admisible?

Puede que sí

Puede que no

No Podemos asegurarlo.

No podemos garantizar que una heurística no admisible generará menos nodos porque **la heurística (solución) no está acotada**

3. Búsqueda A*: comparación y análisis

¿Podemos decir entonces que una **heurística no admisible** genera menos nodos que una heurística admisible?

Puede que sí

Puede que no

No Podemos asegurarlo.

No podemos garantizar que una heurística no admisible generará menos nodos porque **la heurística (solución) no está acotada**

Depende de la profundidad de la solución, de lo buena que sea la estimación de $h(n)$, etc..

3. Búsqueda A*: comparación y análisis

3. Búsqueda A*: comparación y análisis

- Comparación con otras estrategias de búsqueda:
 - Búsqueda en anchura (óptima si todos los operadores tienen el mismo coste). Equivalente a $f(n)=\text{nivel}(n)+0$, donde $h(n)=0 < h^*(n)$
 - Coste uniforme (óptima). Equivalente a $f(n)=g(n)+0$ donde $h(n)=0 < h^*(n)$
 - Profundidad (no óptima): $f(n)=-\text{nivel}(n)$. No comparable a A*
- Conocimiento heurístico:
 - $h(n)=0$, ausencia de conocimiento
 - $h(n)=h^*(n)$, conocimiento máximo

3. Búsqueda A*: comparación y análisis

3. Búsqueda A*: comparación y análisis

$h(n) = 0$: coste computacional de $h(n)$ nulo. (Búsqueda lenta. Admisible.)

$h(n) = h^*(n)$: coste computacional grande de $h(n)$. (Búsqueda rápida. Admisible).

$h(n) > h^*(n)$: coste computacional de $h(n)$ muy alto. (Búsqueda muy rápida. No admisible)

En general, h^* no es conocido pero es posible establecer si h es una cota inferior de h^* o no.

3. Búsqueda A*: comparación y análisis

$h(n) = 0$: coste computacional de $h(n)$ nulo. (Búsqueda lenta. Admisible.)
 $h(n) = h^*(n)$: coste computacional grande de $h(n)$. (Búsqueda rápida. Admisible).
 $h(n) \gg h^*(n)$: coste computacional de $h(n)$ muy alto. (Búsqueda muy rápida. No admisible)

En general, h^* no es conocido pero es posible establecer si h es una cota inferior de h^* o no.

Para problemas muy complejos se recomienda reducir el espacio de búsqueda. En estos casos, merece la pena utilizar $h(n) \gg h^*(n)$ con el objetivo de encontrar una solución en un coste razonable incluso aunque ésta no sea la solución óptima.

3. Búsqueda A*: comparación y análisis

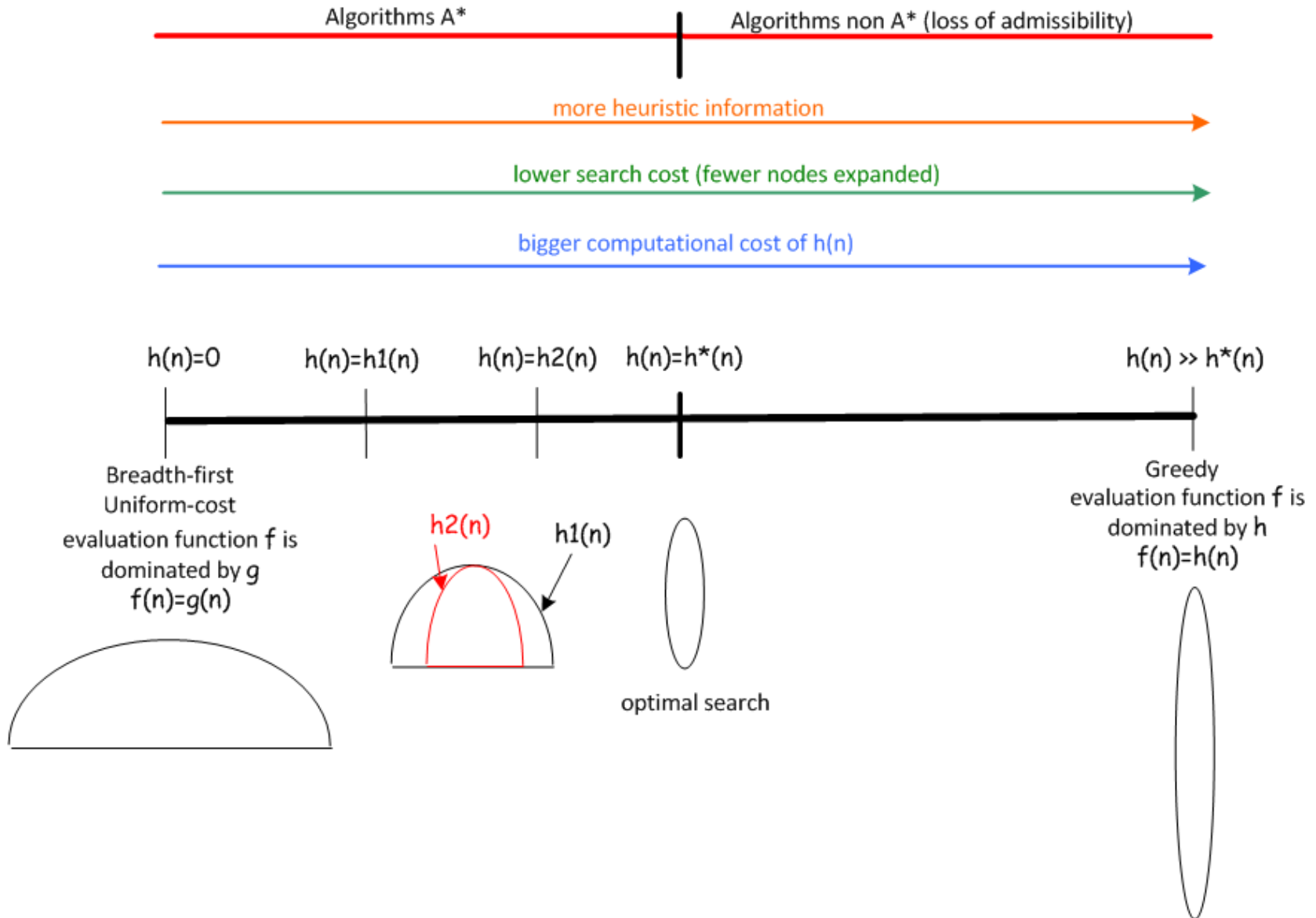
$h(n) = 0$: coste computacional de $h(n)$ nulo. (Búsqueda lenta. Admisible.)
 $h(n) = h^*(n)$: coste computacional grande de $h(n)$. (Búsqueda rápida. Admisible).
 $h(n) > h^*(n)$: coste computacional de $h(n)$ muy alto. (Búsqueda muy rápida. No admisible)

En general, h^* no es conocido pero es posible establecer si h es una cota inferior de h^* o no.

Para problemas muy complejos se recomienda reducir el espacio de búsqueda. En estos casos, merece la pena utilizar $h(n) > h^*(n)$ con el objetivo de encontrar una solución en un coste razonable incluso aunque ésta no sea la solución óptima.

Para cada problema, el objetivo es encontrar un equilibrio entre el **coste de búsqueda** y el **coste del camino solución**.

3. Búsqueda A*: comparación y análisis



3. Búsqueda A*: optimalidad en TREE-SEARCH

3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobreestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobreestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si $h(n)$ es admisible entonces A* es óptimo:

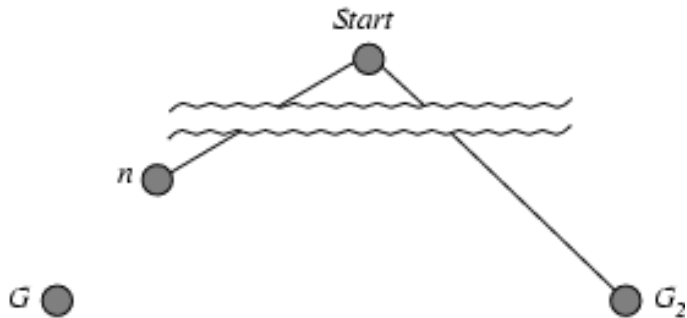
3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobreestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si $h(n)$ es admisible entonces A* es óptimo:



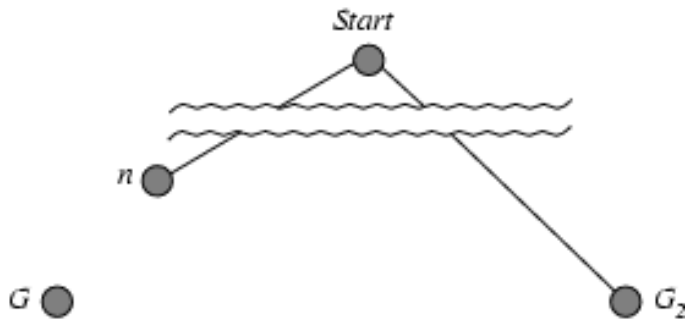
3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobrestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si $h(n)$ es admisible entonces A* es óptimo:



- Sea *Start* el estado inicial de un problema tal que la solución óptima desde *Start* conduce a un nodo solución G : $f(G) = g(G)$
- Sea G_2 otro nodo solución ($f(G_2) = g(G_2)$) tal que el coste de G_2 es mayor que el de G : $g(G_2) > g(G)$ **(2)**
- Sea n un nodo de la lista OPEN en el camino óptimo a G . Y supongamos que n y G_2 están en la lista OPEN

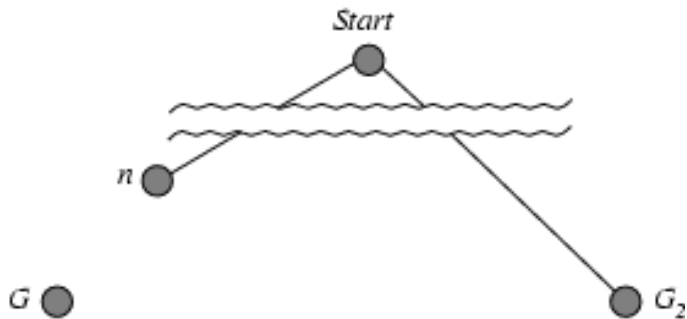
3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobrestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ (1)

Probar que si $h(n)$ es admisible entonces A* es óptimo:



- Sea *Start* el estado inicial de un problema tal que la solución óptima desde *Start* conduce a un nodo solución G : $f(G) = g(G)$
- Sea G_2 otro nodo solución ($f(G_2) = g(G_2)$) tal que el coste de G_2 es mayor que el de G : $g(G_2) > g(G)$ (2)
- Sea n un nodo de la lista OPEN en el camino óptimo a G . Y supongamos que n y G_2 están en la lista OPEN

Si no se escoge n para expansión entonces $f(n) >= f(G_2)$ (3)

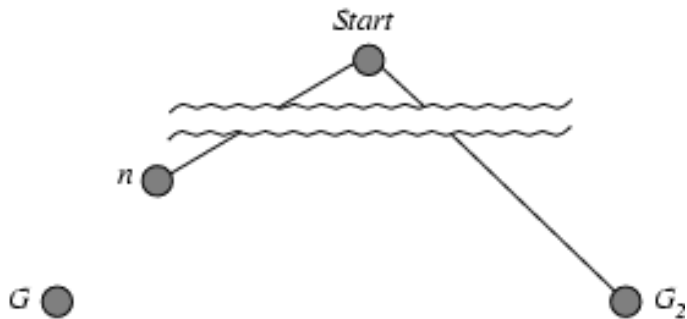
3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobrestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si $h(n)$ es admisible entonces A* es óptimo:



- Sea *Start* el estado inicial de un problema tal que la solución óptima desde *Start* conduce a un nodo solución G : $f(G) = g(G)$
- Sea G_2 otro nodo solución ($f(G_2) = g(G_2)$) tal que el coste de G_2 es mayor que el de G : $g(G_2) > g(G)$ **(2)**
- Sea n un nodo de la lista OPEN en el camino óptimo a G . Y supongamos que n y G_2 están en la lista OPEN

Si no se escoge n para expansión entonces $f(n) > f(G_2)$ **(3)**

Si combinamos **(3)** y **(1)**:

$$f(G_2) < f(n) < g(G) \Rightarrow g(G_2) \leq g(G)$$

Esto contradice **(2)** así que se escoge n

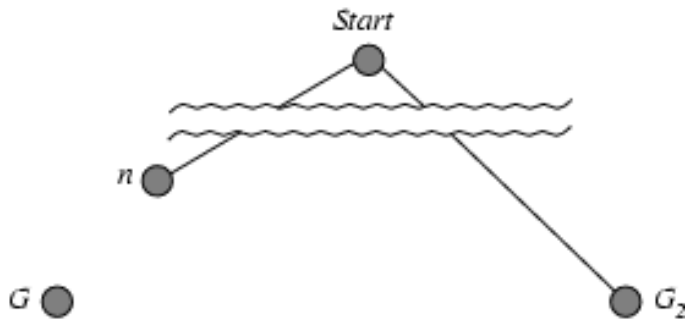
3. Búsqueda A*: optimalidad en TREE-SEARCH

La optimalidad de un algoritmo A* TREE-SEARCH se garantiza si $h(n)$ es una **heurística admisible**

Admisibilidad:

- $h(n)$ es admisible si nunca sobrestima el coste de alcanzar el objetivo: $\forall n, h(n) \leq h^*(n)$
- Sea G un nodo objetivo/solución y n un nodo en el camino óptimo a G . $f(n)$ nunca sobrestima el coste del camino a través de n .
 - $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)$ **(1)**

Probar que si $h(n)$ es admisible entonces A* es óptimo:



- Sea *Start* el estado inicial de un problema tal que la solución óptima desde *Start* conduce a un nodo solución G : $f(G) = g(G)$
- Sea $G2$ otro nodo solución ($f(G2) = g(G2)$) tal que el coste de $G2$ es mayor que el de G : $g(G2) > g(G)$ **(2)**
- Sea n un nodo de la lista OPEN en el camino óptimo a G . Y supongamos que n y $G2$ están en la lista OPEN

Si no se escoge n para expansión entonces $f(n) > f(G2)$ **(3)**

Si combinamos **(3)** y **(1)**:

$$f(G2) < f(n) < g(G) \Rightarrow g(G2) \leq g(G)$$

Esto contradice **(2)** así que se escoge n

Por tanto, el algoritmo devolverá la solución óptima

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

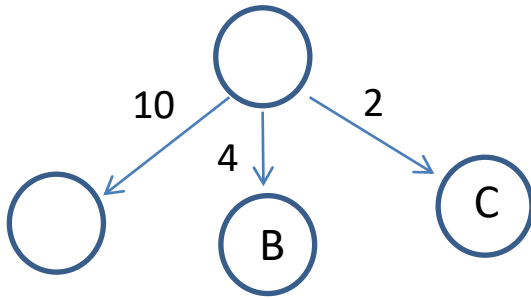
En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

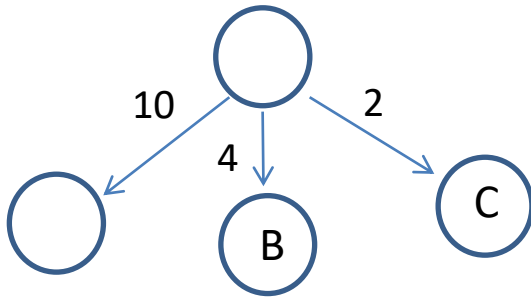
Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



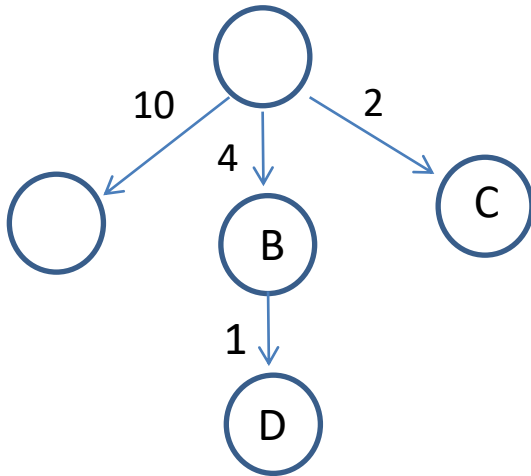
$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



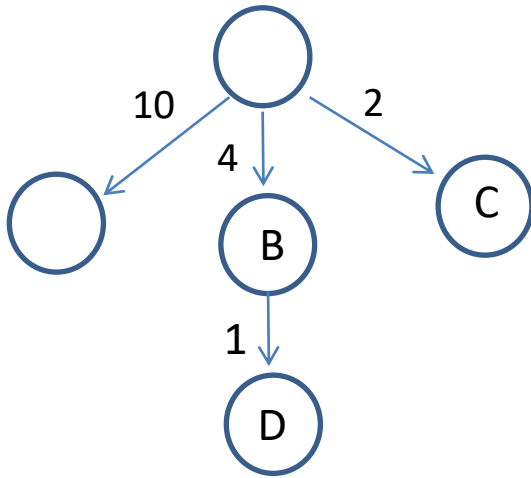
$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

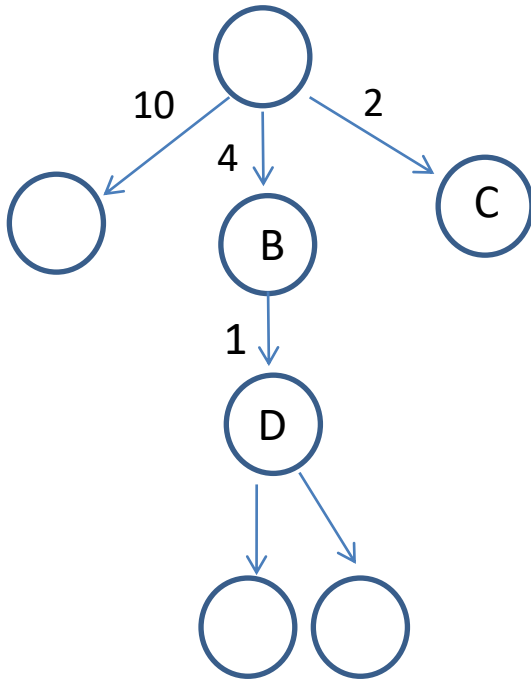
$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

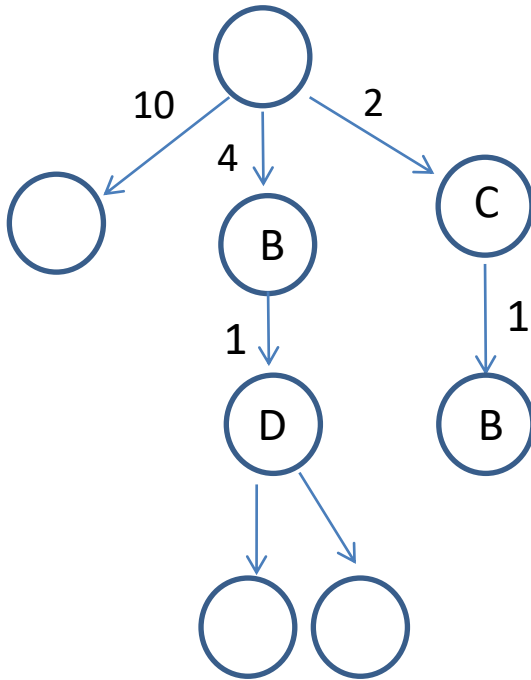
$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

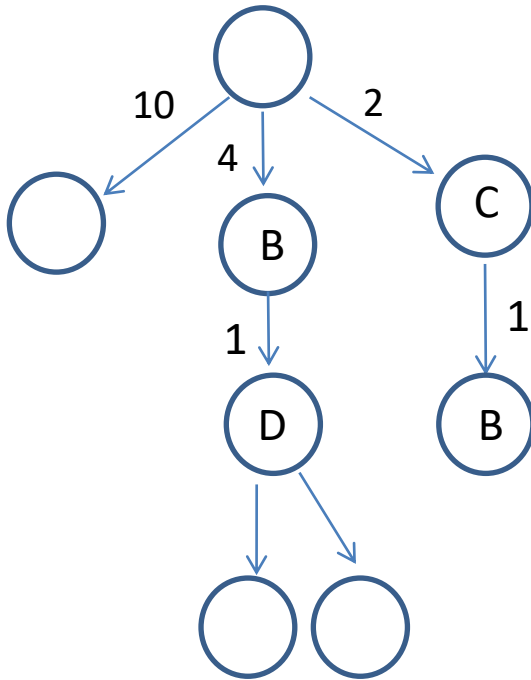
$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

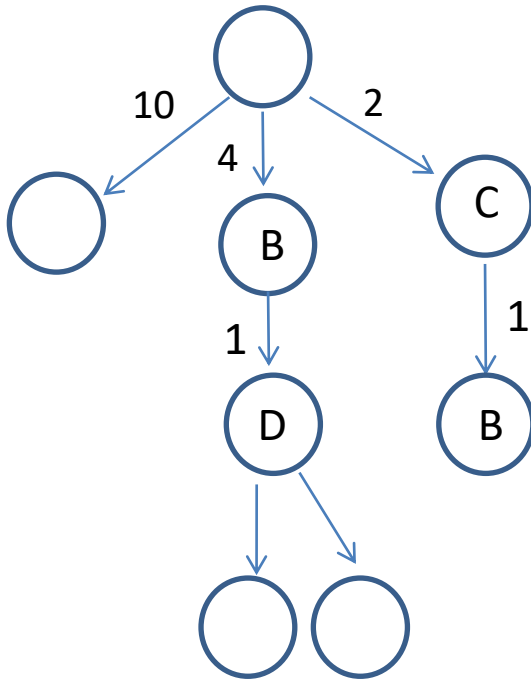
$$f(D) = 5 + h(D) = 5 + 3 = 8$$

$$f(B) = 3 + h(B) = 3 + 3 = 6$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

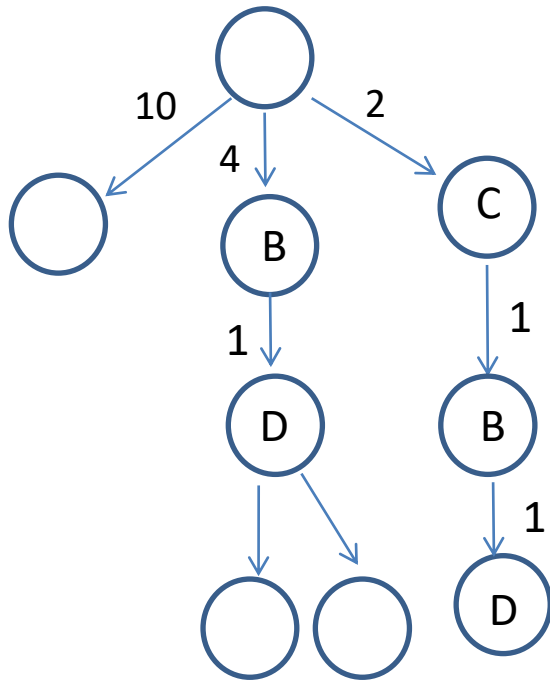
$$f(B) = 3 + h(B) = 3 + 3 = 6$$

Re-expandimos el nodo B

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

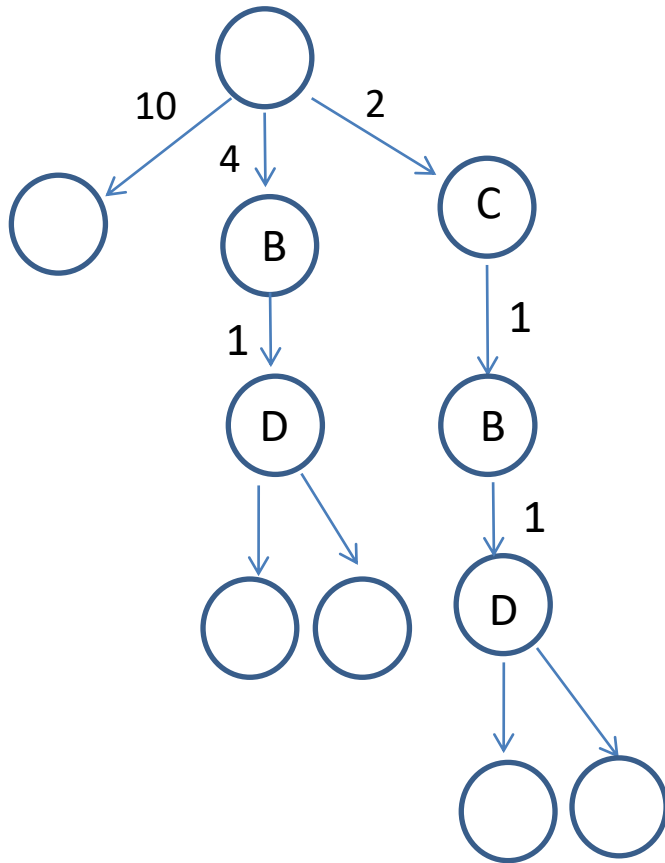
$$f(B) = 3 + h(B) = 3 + 3 = 6$$

Re-expandimos el nodo B

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

En un algoritmo A* GRAPH-SEARCH con re-expansión de nodos repetidos en CLOSED se garantiza que se encuentra la solución óptima si $h(n)$ es una **heurística admisible**.

Supongamos que aplico $f(n)=g(n)+h(n)$ donde $h(n)$ es admisible



$$f(B) = 4 + h(B) = 4 + 3 = 7$$

$$f(C) = 2 + h(C) = 2 + 6 = 8$$

$$f(D) = 5 + h(D) = 5 + 3 = 8$$

$$f(B) = 3 + h(B) = 3 + 3 = 6$$

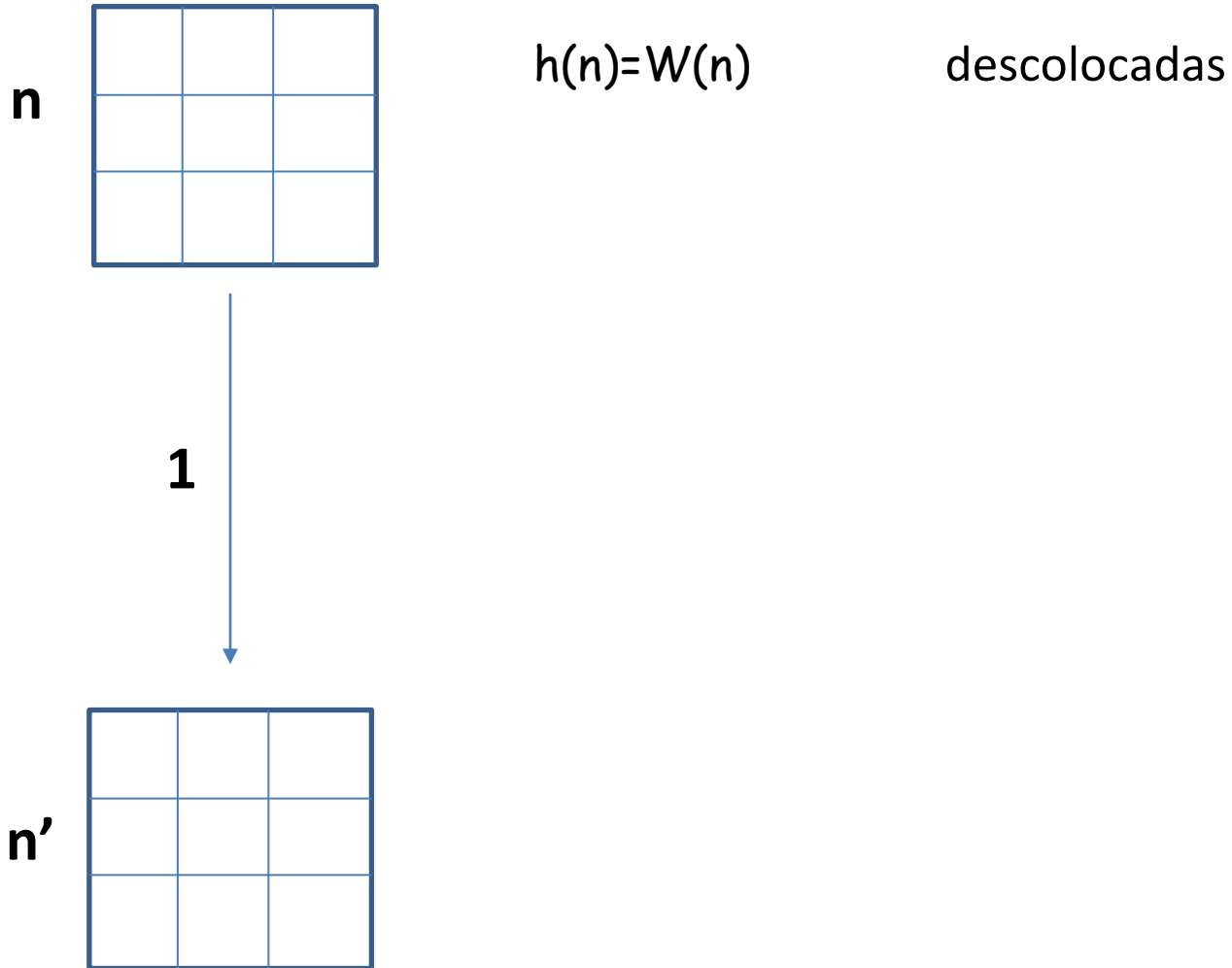
Re-expandimos el nodo B

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

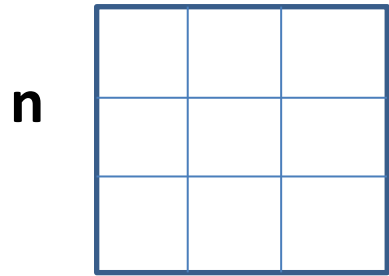
En un algoritmo A* GRAPH-SEARCH sin re-expansión de nodos repetidos se garantiza que se encuentra la solución óptima si se puede asegurar que el primer camino que se encuentra a un nodo es el mejor camino hasta dicho nodo (**heurística consistente**).

Consistencia: $h(n)$ es consistente si, para cada nodo n y cada sucesor n' de n generado con una acción a se cumple $h(n) \leq h(n') + \text{Action-cost}(n, a, n')$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



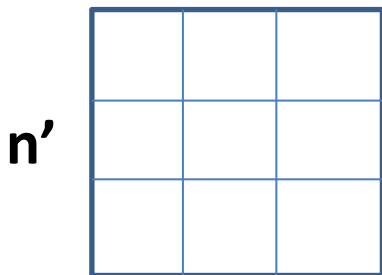
$$h(n) = W(n)$$

descolocadas

1

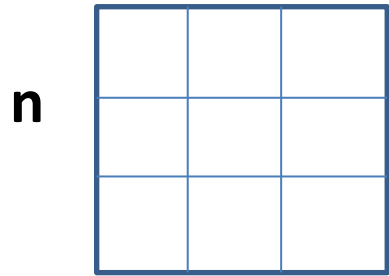


A downward arrow indicating a transition from state n to state n'.



$$h(n') = W(n) - 1$$

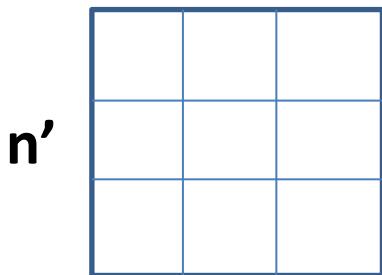
3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$$h(n) = W(n)$$

descolocadas

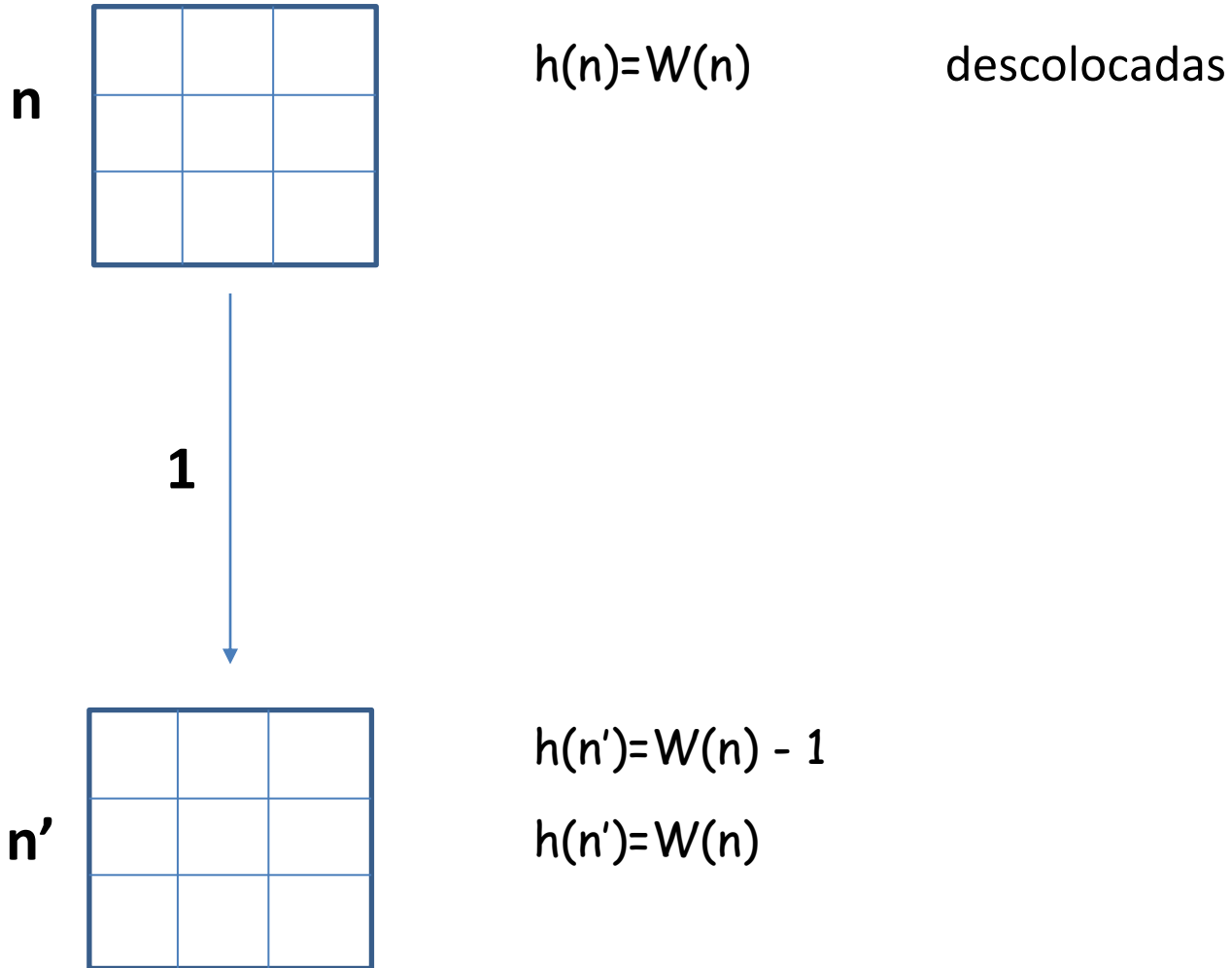
1



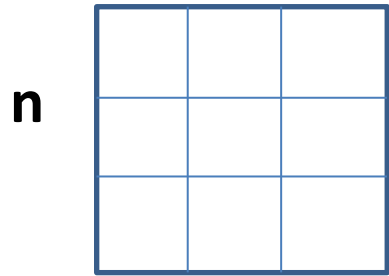
$$h(n') = W(n) - 1$$

$$h(n) \leq h(n') + c(n, a, n')$$
$$W(n) \leq W(n) - 1 + 1$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



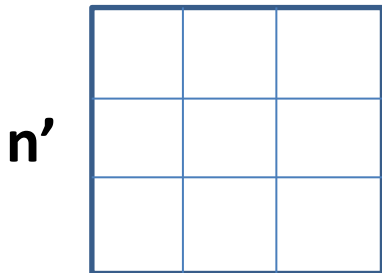
3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$$h(n) = W(n)$$

descolocadas

1

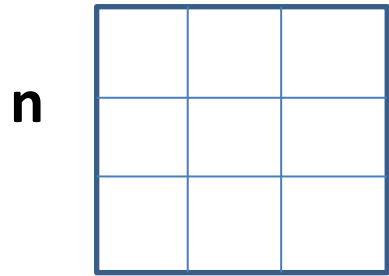


$$h(n') = W(n) - 1$$

$$h(n') = W(n)$$

$$h(n) \leq h(n') + c(n, a, n')$$
$$W(n) \leq W(n) + 1$$

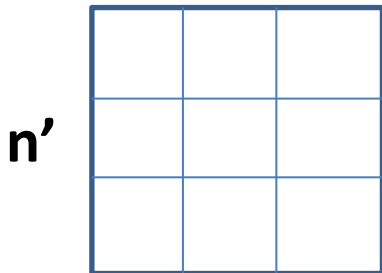
3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$$h(n) = W(n)$$

descolocadas

1

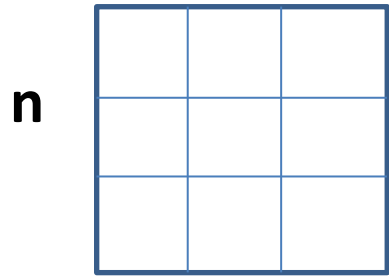


$$h(n') = W(n) - 1$$

$$h(n') = W(n)$$

$$h(n') = W(n) + 1$$

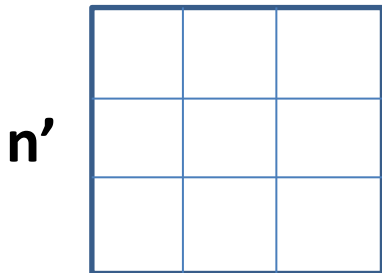
3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$$h(n) = W(n)$$

descolocadas

1



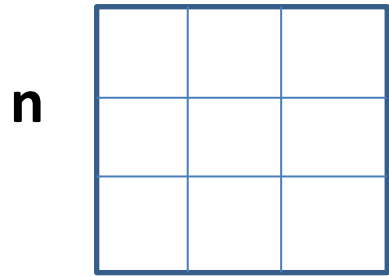
$$h(n') = W(n) - 1$$

$$h(n') = W(n)$$

$$h(n') = W(n) + 1$$

$$h(n) \leq h(n') + c(n, a, n')$$
$$W(n) \leq W(n) + 1 + 1$$

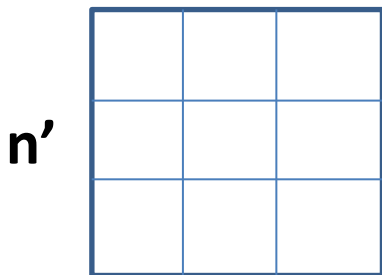
3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$$h(n) = W(n)$$

descolocadas

1



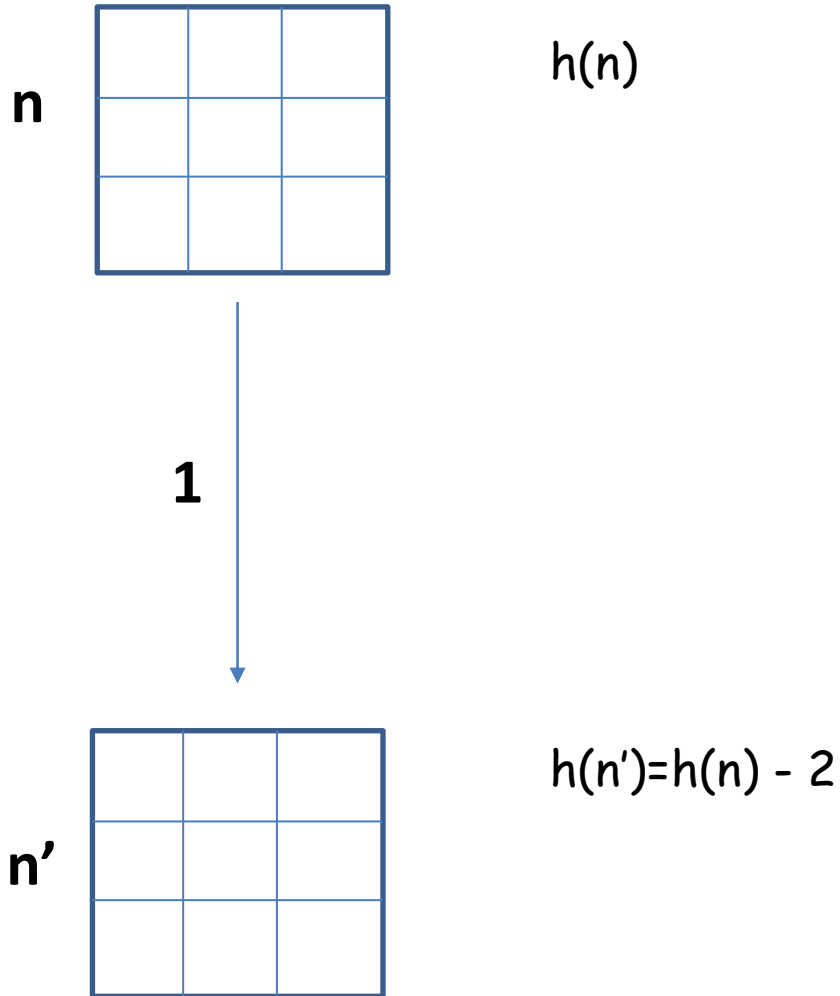
$$h(n') = W(n) - 1$$

$$h(n') = W(n)$$

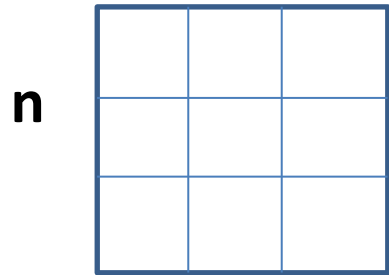
$$h(n') = W(n) + 1$$

La heurística $W(n)$ es consistente (con los costes)

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)

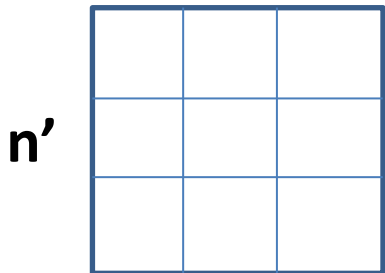


3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$h(n)$

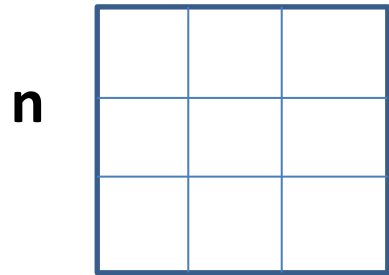
1



$h(n') = h(n) - 2$

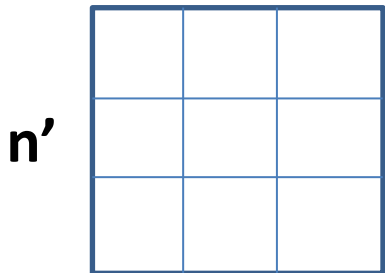
$$h(n) \leq h(n') + c(n, a, n')$$
$$h(n) \leq h(n) - 2 + 1$$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



$h(n)$

1



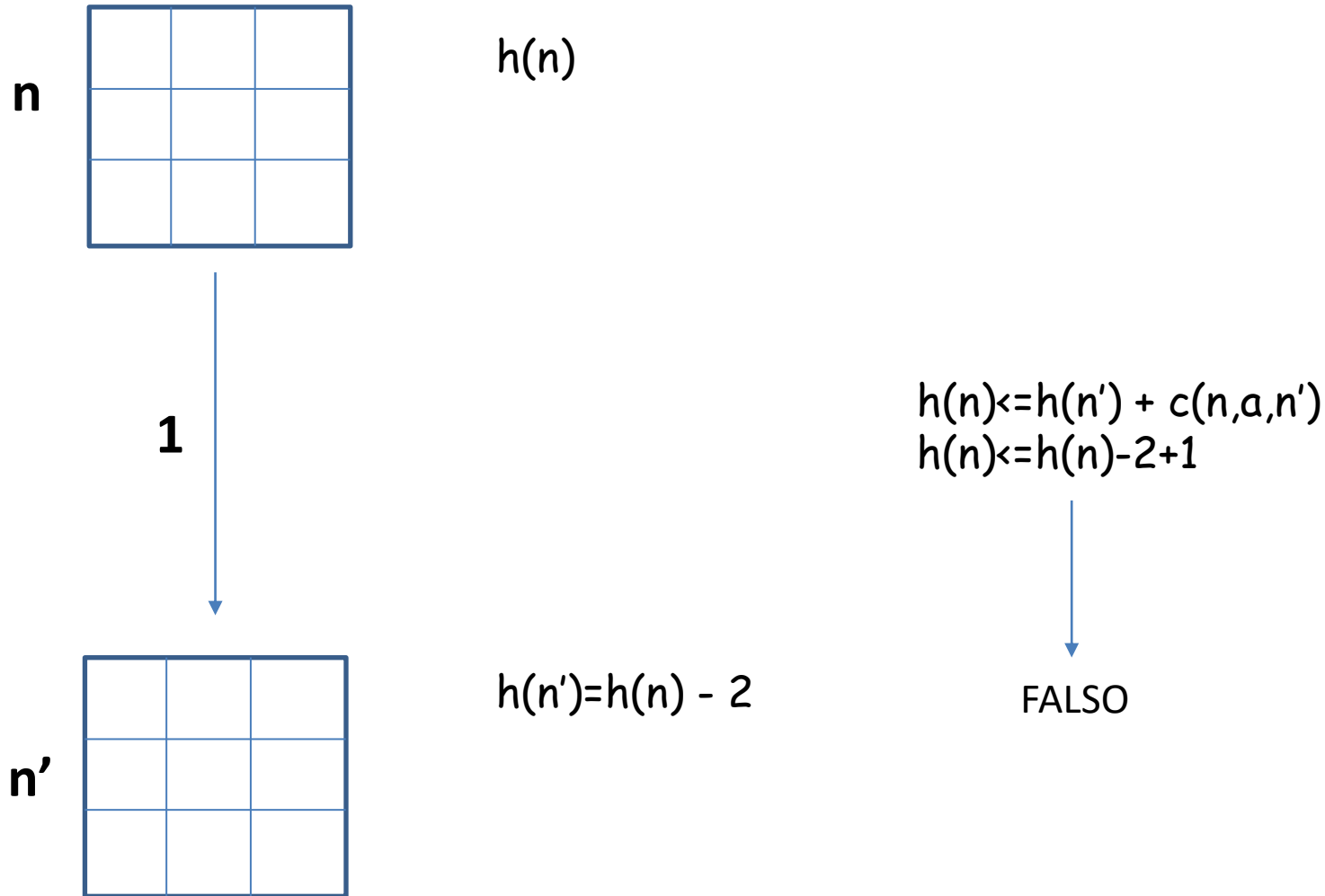
$h(n') = h(n) - 2$

$$h(n) \leq h(n') + c(n, a, n')$$
$$h(n) \leq h(n) - 2 + 1$$



FALSO

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)



La heurística NO ES consistente (con los costes)

3. Búsqueda A*: optimalidad en GRAPH-SEARCH (consistencia)

if $h(n)$ es consistente (y por tanto es admissible)

then no hay necesidad de re-expansión

porque se garantiza que cuando un nodo **n** se expande, hemos encontrado la solución óptima desde el estado inicial al nodo **n**

La situación del nodo B del grafo de la transparencia 31 no habría ocurrido porque habríamos encontrado primero el camino $g(B)=3$.

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

Consistencia (también llamada *monotonicidad*) es una condición ligeramente más fuerte que la admisibilidad.

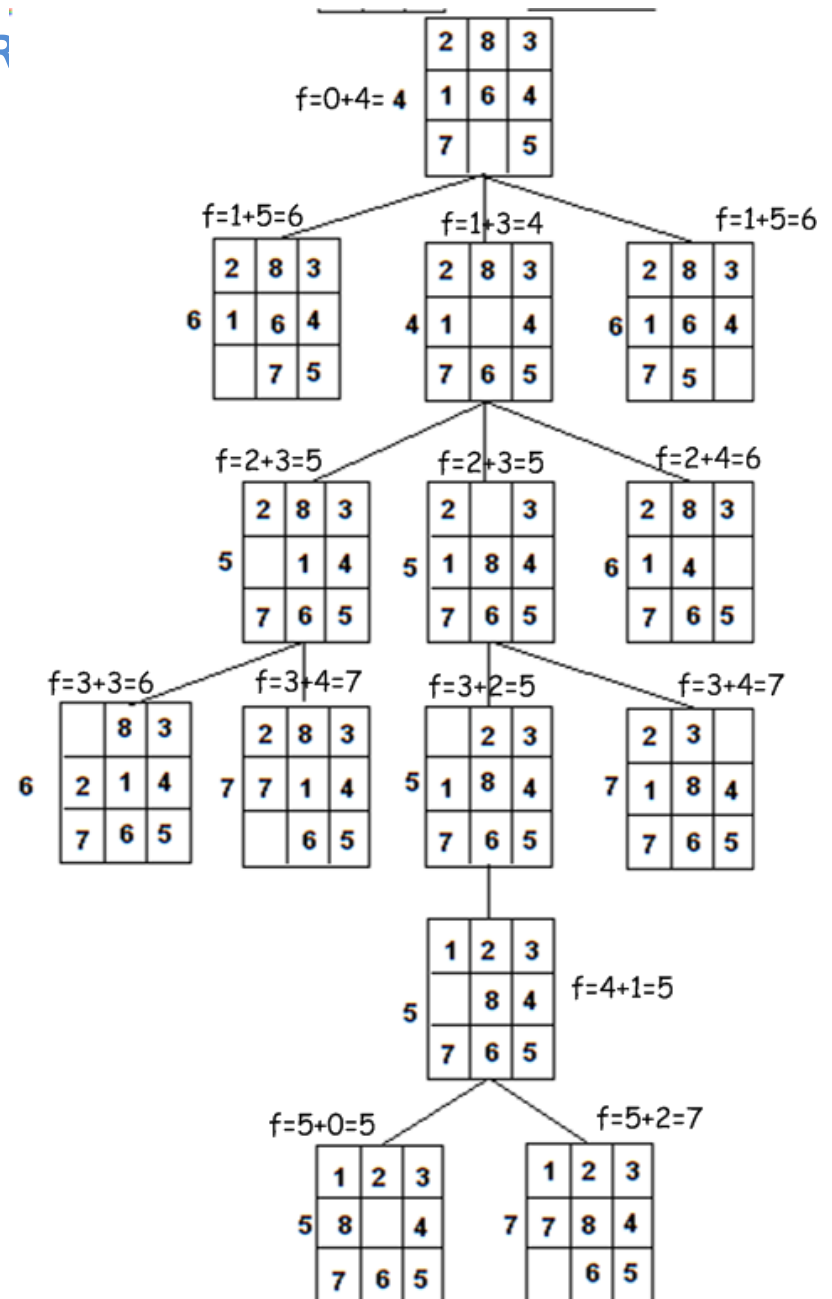
Por tanto, un algoritmo A* GRAPH-SEARCH sin re-expansión de nodos repetidos que utiliza una heurística consistente $h(n)$ también devuelve la **solución óptima** porque:

- se garantiza que cuando se expande un nodo n , se ha encontrado la solución óptima a dicho estado; por tanto, si aparece un nodo repetido de n (que estaría en CLOSED), no haría falta re-expandir dicho nodo repetido porque su coste nunca será mejor que el coste del nodo n ya expandido en CLOSED.
- una derivación de la condición de consistencia es que la secuencia de nodos expandidos por A* GRAPH-SEARCH es una función no decreciente de $f(n)$

3. Búsqueda A*: optimalidad en GRAPH-SEARCH

- una derivación de la condición de consistencia es que la secuencia de nodos expandidos por la version GRAPH-SEARCH de un algoritmo A* es una función **no decreciente de $f(n)$**

Búsqueda A* ($f(n)=g(n)+h(n)$) usando $h(n)=$ **piezas descolocadas**



3. Búsqueda A*: evaluación

3. Búsqueda A*: evaluación

- Completo:
 - Sí, si existe al menos una solución, A* la encuentra (a menos que existan infinitos nodos n tal que $f(n) < f(G)$)
- Óptima:
 - Sí, si se cumple la condición de consistencia (para la versión GRAPH-SEARCH)
 - Siempre existe al menos un nodo n en OPEN que pertenece al camino óptimo de la solución
 - Si C^* es el coste de la solución óptima:
 - A* expande todos los nodos con $f(n) < C^*$
 - A* podría expandir algunos nodos con $f(n)=C^*$.
 - A* no expande nodos con $f(n) > C^*$

3. Búsqueda A*: evaluación

- Completo:
 - Sí, si existe al menos una solución, A* la encuentra (a menos que existan infinitos nodos n tal que $f(n) < f(G)$)
- Óptima:
 - Sí, si se cumple la condición de consistencia (para la versión GRAPH-SEARCH)
 - Siempre existe al menos un nodo n en OPEN que pertenece al camino óptimo de la solución
 - Si C^* es el coste de la solución óptima:
 - A* expande todos los nodos con $f(n) < C^*$
 - A* podría expandir algunos nodos con $f(n)=C^*$.
 - A* no expande nodos con $f(n) > C^*$
- Complejidad temporal:
 - $O(b^{C^*/\text{min_coste_acción}})$; exponencial con la longitud de camino
 - El número de nodos expandidos es exponencial con la longitud de la solución

3. Búsqueda A*: evaluación

- **Completo:**
 - Sí, si existe al menos una solución, A* la encuentra (a menos que existan infinitos nodos n tal que $f(n) < f(G)$)
- **Óptima:**
 - Sí, si se cumple la condición de consistencia (para la versión GRAPH-SEARCH)
 - Siempre existe al menos un nodo n en OPEN que pertenece al camino óptimo de la solución
 - Si C^* es el coste de la solución óptima:
 - A* expande todos los nodos con $f(n) < C^*$
 - A* podría expandir algunos nodos con $f(n) = C^*$.
 - A* no expande nodos con $f(n) > C^*$
- **Complejidad temporal:**
 - $O(b^{C^*/\text{min_coste_acción}})$; exponencial con la longitud de camino
 - El número de nodos expandidos es exponencial con la longitud de la solución
- **Complejidad espacial:**
 - Mantiene todos los nodos en memoria (como todas las versiones GRAPH-SEARCH)
 - A* normalmente se queda sin espacio mucho antes de que se agote el tiempo
 - Por tanto, el mayor problema es el espacio, no el tiempo

4. Diseño de funciones heurísticas

4. Diseño de funciones heurísticas

Las heurísticas son funciones dependientes del problema.

¿Cómo diseñar una función heurística (admisible) para un problema?

Técnica común:

Relajación de las restricciones del problema

Considerar el problema con menos restricciones, obteniendo así otro problema que se resuelve con una complejidad menor que la del problema inicial.

El coste de la solución del problema relajado se utiliza como una estimación (admisible) del coste del problema original.

4.1. Heurísticas para el problema de 8-puzzle

8-puzzle: una ficha situada en una casilla **A** se puede mover a una casilla **B** si

Restricción 1: **B** es adyacente a **A**

Restricción 2: **B** es el espacio vacío

4.1. Heurísticas para el problema de 8-puzzle

8-puzzle: una ficha situada en una casilla **A** se puede mover a una casilla **B** si

Restricción 1: **B** es adyacente a **A**

Restricción 2: **B** es el espacio vacío

h1: fichas descolocadas

- elimina ambas restricciones
- relajación de $h1(n)$: una ficha se puede mover a cualquier casilla
- $h1(n)$ devuelve una estimación muy optimista (solución óptima para el problema relajado)

4.1. Heurísticas para el problema de 8-puzzle

8-puzzle: una ficha situada en una casilla **A** se puede mover a una casilla **B** si

Restricción 1: **B** es adyacente a **A**

Restricción 2: **B** es el espacio vacío

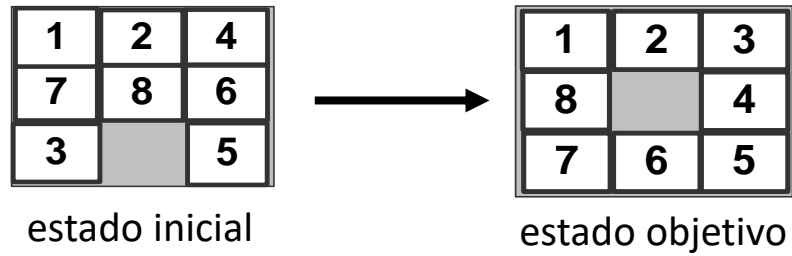
h1: fichas descolocadas

- elimina ambas restricciones
- relajación de $h1(n)$: una ficha se puede mover a cualquier casilla
- $h1(n)$ devuelve una estimación muy optimista (solución óptima para el problema relajado)

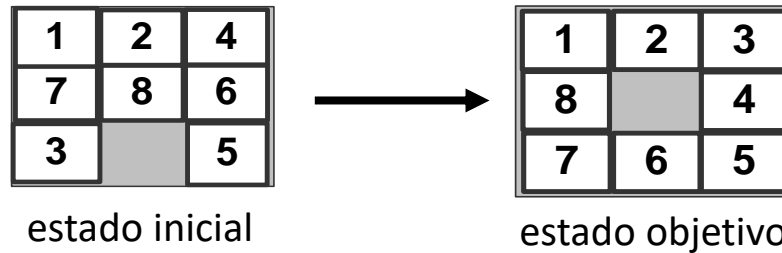
h2: distancias de Manhattan

- suma de las distancias de cada ficha a su posición objetivo)
- elimina restricción 2
- Relajación de $h2(n)$: una ficha se puede mover a cualquier casilla adyacente
- $h2(n)$ devuelve una estimación optimista (solución óptima para el problema relajado)

4.1. Heurísticas para el problema de 8-puzzle

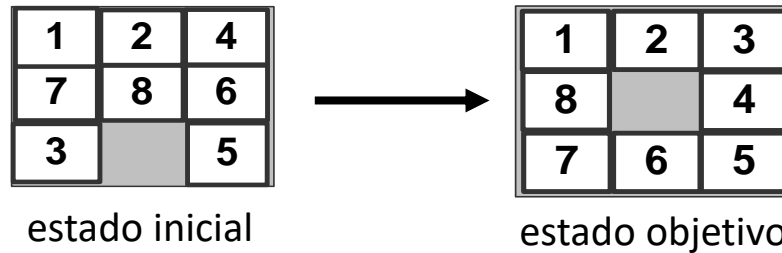


4.1. Heurísticas para el problema de 8-puzzle



- El problema de 8-puzzle:
 - El coste medio de una solución es aproximadamente 22 pasos (factor de ramificación ≈ 3)
 - Búsqueda exhaustiva a profundidad 22: $3^{22} \approx 3.1 \times 10^{10}$ estados
 - Una buena función heurística puede reducir el proceso de búsqueda

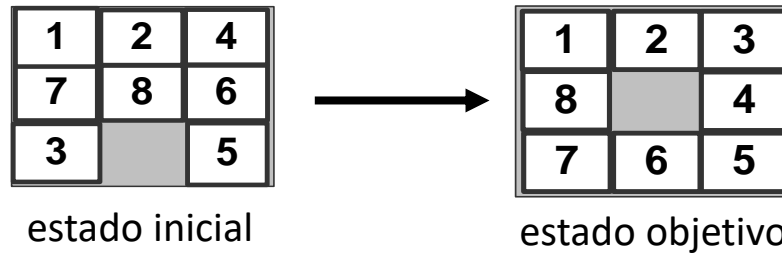
4.1. Heurísticas para el problema de 8-puzzle



- El problema de 8-puzzle:
 - El coste medio de una solución es aproximadamente 22 pasos (factor de ramificación ≈ 3)
 - Búsqueda exhaustiva a profundidad 22: $3^{22} \approx 3.1 \times 10^{10}$ estados
 - Una buena función heurística puede reducir el proceso de búsqueda

Heurística 1 (fichas descolocadas): $h_1(n)=5$ para el ejemplo

4.1. Heurísticas para el problema de 8-puzzle

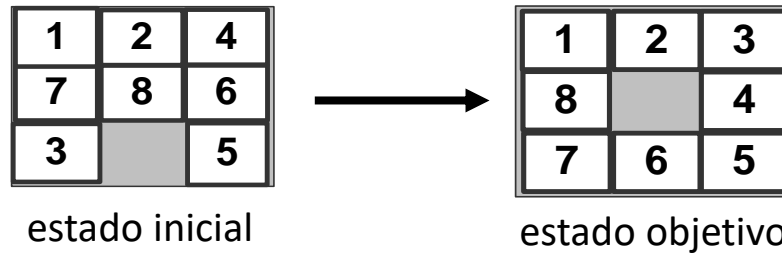


- El problema de 8-puzzle:
 - El coste medio de una solución es aproximadamente 22 pasos (factor de ramificación ≈ 3)
 - Búsqueda exhaustiva a profundidad 22: $3^{22} \approx 3.1 \times 10^{10}$ estados
 - Una buena función heurística puede reducir el proceso de búsqueda

Heurística 1 (fichas descolocadas): $h1(n)=5$ para el ejemplo

Heurística 2 (distancias de Manhattan): $h2(n)=1+2+4+1+1=9$ para el ejemplo

4.1. Heurísticas para el problema de 8-puzzle



- El problema de 8-puzzle:
 - El coste medio de una solución es aproximadamente 22 pasos (factor de ramificación ≈ 3)
 - Búsqueda exhaustiva a profundidad 22: $3^{22} \approx 3.1 \times 10^{10}$ estados
 - Una buena función heurística puede reducir el proceso de búsqueda

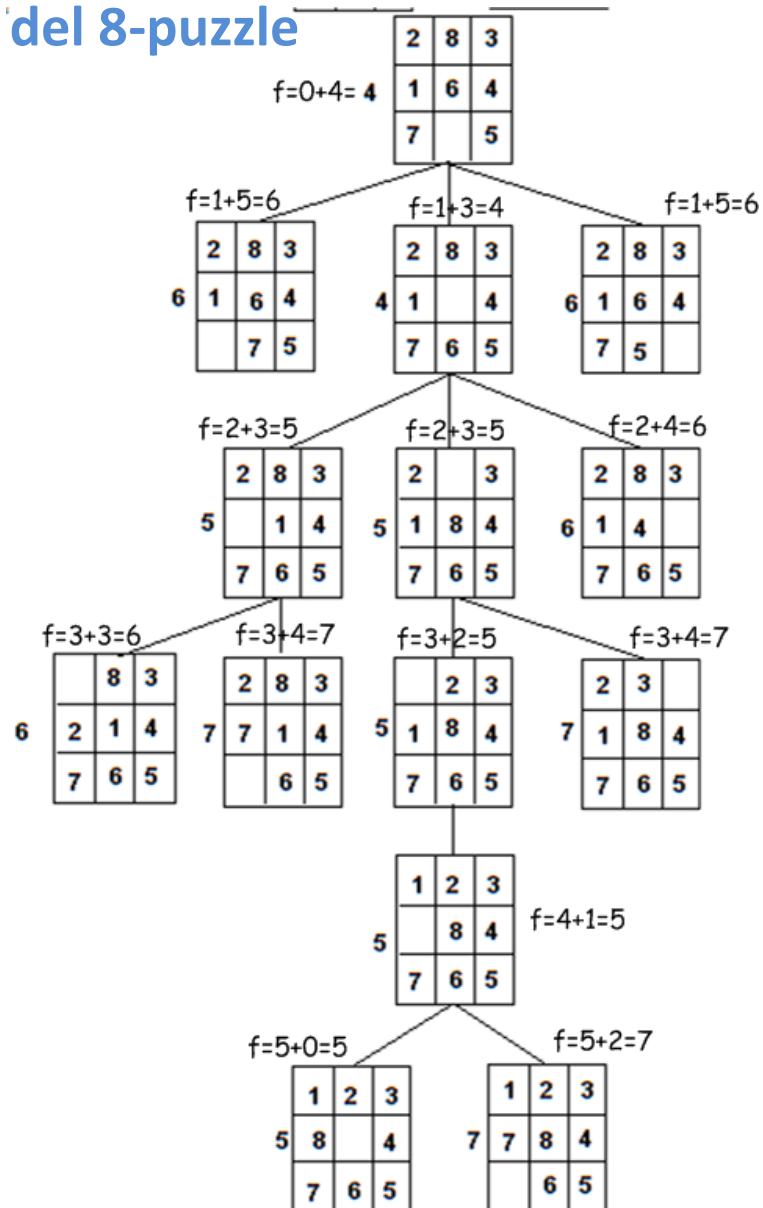
Heurística 1 (fichas descolocadas): $h1(n)=5$ para el ejemplo

Heurística 2 (distancias de Manhattan): $h2(n)=1+2+4+1+1=9$ para el ejemplo

Distancias de Manhattan domina a fichas descolocadas ($h2(n) \geq h1(n), \forall n$)

4.1. Heurísticas para el problema del 8-puzzle

Búsqueda A* ($f(n)=g(n)+h(n)$)
con la heurística **fichas descolocadas**



4.2. Heurísticas para el problema del viajante de comercio

6 ciudades: A,B,C,D,E,F

Estados: secuencias de ciudades que comienzan en la ciudad A y representan rutas parciales

Inicio: A

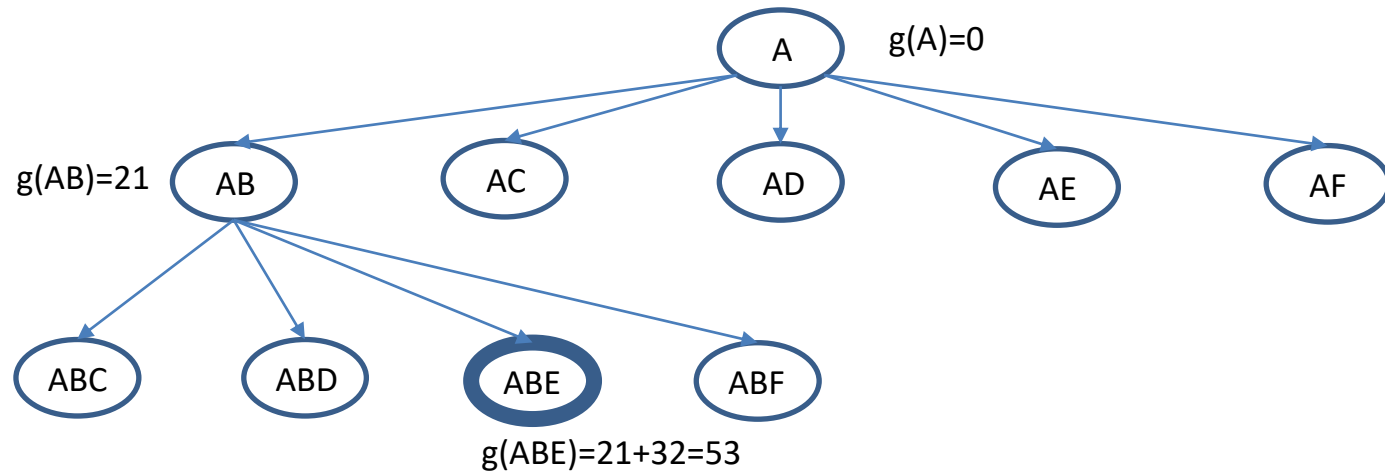
Final: secuencias que empiezan y terminan en A y pasan por todas las ciudades

Reglas u operadores: añadir al final de cada estado una ciudad que no está en la secuencia

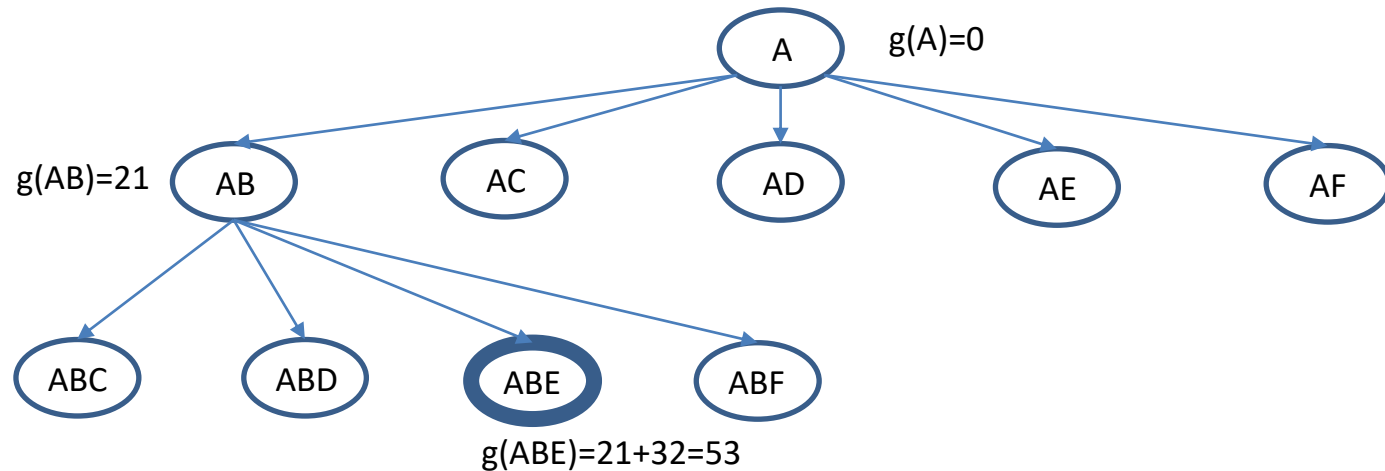
Coste de los operadores: distancia entre la última ciudad de la secuencia y la nueva ciudad añadida (ver tabla)

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio



4.2. Heurísticas para el problema del viajante de comercio



Faltan por visitar: C, D, F, volver a A

$$h^*(ABE) = \min(\text{ECDFA}, \text{ECFDA}, \text{EDCFA}, \text{EDFCA}, \text{EFCDA}, \text{EFDCA})$$
$$\min(154, 92, 297, 251, 57, 73)$$

4.2. Heurísticas para el problema del viajante de comercio

$h(ABE) \rightarrow$
C, D, F, volver a A

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(ABE) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(ABE) = 4 * 5 = 20$]

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(ABE) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(ABE)=4*5=20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(ABE)=5+5+7+7=24$]

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo

$$[h_2(\text{ABE}) = 4 \cdot 5 = 20]$$

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)

$$[h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24]$$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 \cdot 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 * 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar
[$h_4(\text{ABE}) = 17 \{\text{abandonar E}\} +$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 * 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

[$h_4(\text{ABE}) = 17 \text{ {abandonar E}} +$
 $5 \text{ {abandonar C}} +$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 \cdot 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

$$[h_4(\text{ABE}) = 17 \{\text{abandonar E}\} + 5 \{\text{abandonar C}\} + 5 \{\text{abandonar D}\} + \dots]$$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 * 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

$$\begin{aligned} h_4(\text{ABE}) = & 17 \{\text{abandonar E}\} + \\ & 5 \{\text{abandonar C}\} + \\ & 5 \{\text{abandonar D}\} + \\ & 9 \{\text{abandonar F}\} = 36 \end{aligned}$$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo
[$h_2(\text{ABE}) = 4 * 5 = 20$]

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)
[$h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24$]

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

$$\begin{aligned} h_4(\text{ABE}) = & 17 \{\text{abandonar E}\} + \\ & 5 \{\text{abandonar C}\} + \\ & 5 \{\text{abandonar D}\} + \\ & 9 \{\text{abandonar F}\} = 36 \end{aligned}$$

$h_7(n)$ = número de arcos que faltan multiplicado por el coste medio de los arcos

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

4.2. Heurísticas para el problema del viajante de comercio

$h(\text{ABE}) \rightarrow$
C, D, F, volver a A

$h_2(n)$ = número de arcos que faltan multiplicado por el coste del arco mínimo

$$[h_2(\text{ABE}) = 4 * 5 = 20]$$

$h_3(n)$ = suma de los p arcos más cortos si faltan p arcos (arcos no dirigidos)

$$[h_3(\text{ABE}) = 5 + 5 + 7 + 7 = 24]$$

$h_4(n)$ = suma de los arcos más cortos que parten de las ciudades que faltan por abandonar

$$\begin{aligned} [h_4(\text{ABE}) = & 17 \text{ \{abandonar E\} + } \\ & 5 \text{ \{abandonar C\} + } \\ & 5 \text{ \{abandonar D\} + } \\ & 9 \text{ \{abandonar F\} } = 36] \end{aligned}$$

$h_7(n)$ = número de arcos que faltan multiplicado por el coste medio de los arcos

$$[h_7(\text{ABE}) = 4 * 40.33 = 161.33]$$

	A	B	C	D	E	F
A		21	12	15	113	92
B			7	25	32	9
C				5	18	20
D					180	39
E						17

5. Evaluación de funciones heurísticas

5. Evaluación de funciones heurísticas

Dificultad de aplicar un análisis matemático

Se aplican métodos estadísticos/experimentales.

Objetivo: buscar balance entre coste de la búsqueda y coste de la solución

Coste computacional (temporal coste):

Coste de búsqueda: número de nodos generados o operadores aplicables +

Coste de calcular $h(n)$: coste para seleccionar el nodo (operador aplicable)

5. Evaluación de funciones heurísticas

Dificultad de aplicar un análisis matemático

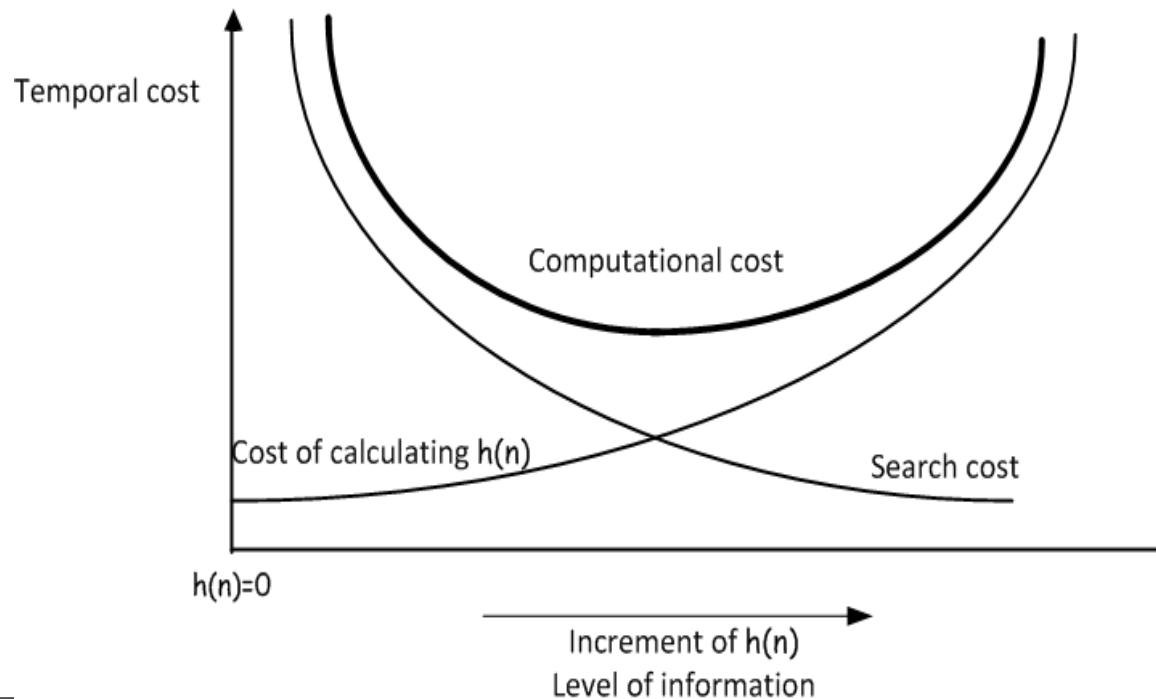
Se aplican métodos estadísticos/experimentales.

Objetivo: buscar balance entre coste de la búsqueda y coste de la solución

Coste computacional (temporal coste):

Coste de búsqueda: número de nodos generados o operadores aplicables +

Coste de calcular $h(n)$: coste para seleccionar el nodo (operador aplicable)



5. Evaluación de funciones heurísticas

Factor efectivo de ramaje (b^*)

N = Número total de nodos generados por un método de búsqueda para un problema particular

d = Profundidad de la solución

b^* = Factor de ramificación de un árbol uniforme de profundidad d con $N+1$ nodos.

Si el número total de nodos generados por un algoritmo A^* para un problema particular es N , y la solución se encuentra en el nivel de profundidad d , entonces b^* es el factor de ramificación que un árbol uniforme de profundidad d tendría para contener $N+1$ nodos:

$$N+1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

5. Evaluación de funciones heurísticas

Factor efectivo de ramaje (b^*)

N = Número total de nodos generados por un método de búsqueda para un problema particular

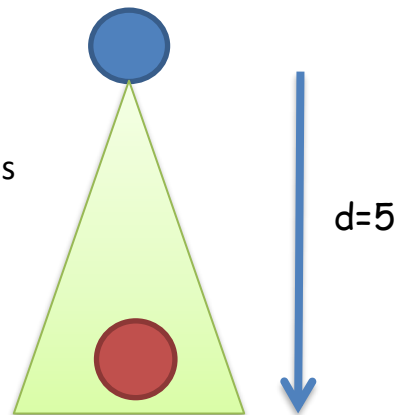
d = Profundidad de la solución

b^* = Factor de ramificación de un árbol uniforme de profundidad d con $N+1$ nodos.

Si el número total de nodos generados por un algoritmo A^* para un problema particular es N , y la solución se encuentra en el nivel de profundidad d , entonces b^* es el factor de ramificación que un árbol uniforme de profundidad d tendría para contener $N+1$ nodos:

$$N+1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

Solución en $d=5$
utilizando $N=52$ nodos
 $\rightarrow b^*=1.92$



5. Evaluación de funciones heurísticas

Factor efectivo de ramaje (b^*)

N = Número total de nodos generados por un método de búsqueda para un problema particular

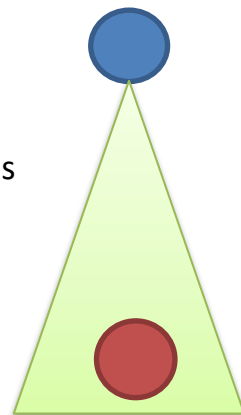
d = Profundidad de la solución

b^* = Factor de ramificación de un árbol uniforme de profundidad d con $N+1$ nodos.

Si el número total de nodos generados por un algoritmo A^* para un problema particular es N , y la solución se encuentra en el nivel de profundidad d , entonces b^* es el factor de ramificación que un árbol uniforme de profundidad d tendría para contener $N+1$ nodos:

$$N+1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

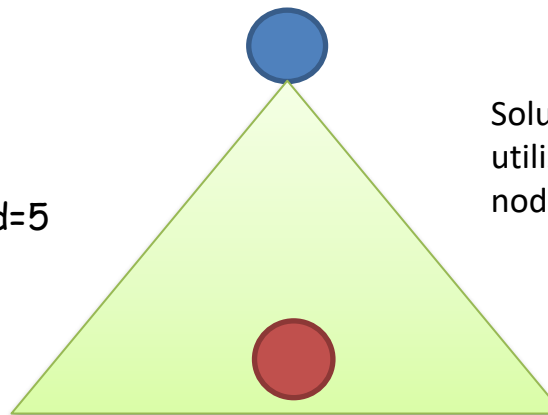
Solución en $d=5$
utilizando $N=52$ nodos
 $\rightarrow b^*=1.92$



$d=5$



Solución en $d=5$
utilizando $N=560$ nodos
 $\rightarrow b^*=3.3$



5. Evaluación de funciones heurísticas

- b^* define si la búsqueda hacia el objetivo está bien enfocada o no: un valor de b^* cercano a 1 corresponde a una búsqueda que está altamente enfocada hacia la meta, con muy poca ramificación en otras direcciones.
- b^* es razonablemente independiente de la longitud del camino (d)
- b^* puede variar de un problema a otro pero suele ser bastante constante para problemas complejos
- Una heurística bien diseñada tendría un valor de b^* cercano a 1, lo que permitiría resolver bastantes instancias complejas del problema.
- Mediciones experimentales de b^* en un pequeño conjunto de problemas pueden proporcionar una buena orientación sobre la utilidad general de la heurística