Tema 2.
Resolución de problemas mediante búsqueda

Búsqueda informada

Lecturas :
- CAPÍTULO 4 de Russell & Norvig
- CAPÍTULOS 9, 10, 11 de Nilsson

Herramientas:
http://qiao.github.io/PathFinding.js/visual/

Heurística

- Del griego *ευρισκω* = encontrar, descubrir.
  - “EUREKA!” De Arquímedes
  - Reglas que generalmente (pero no siempre) ayudan a dirigir la búsqueda hacia la solución.
    - 1945, Georg: “How to Solve It” Métodos para solucionar problemas.
      1. Entender el problema.
      2. Construir un plan.
      3. Ejecutar el plan
      4. Mejorar el plan.
    - 1958, Simposio Teddington sobre “Mecanización de procesos mentales”, UK
      - John McCarthy: "Programas con sentido común"
      - Oliver Selfridge: "Pandemonium"
      - Marvin Minsky: ”Algunos métodos de programación heurística e inteligencia artificial"
    - 1963, Newell
    - Mediados de los 60’s-80’s: Proyecto de Programación Heurística de Stanford*(HPP),* dirigido por E. Feigenbaum para desarrollar sistemas expertos basados en reglas (DENDRAL, MYCIN)

Búsqueda informada

- Las estrategias de búsqueda no informada (ciega) son generalmente muy ineficientes.
- El uso de **conocimiento específico** sobre el problema (más allá de la definición del problema) **para guiar la búsqueda** puede mejorar enormemente la eficiencia.
  - Versión informada de algoritmos de búsqueda general: **búsqueda primero-el-mejor**
    - Búsqueda avariciosa primero-el-mejor
    - Búsqueda A*
    - Búsqueda heurística con memoria acotada
      - IDA* (A* con profundidad iterativa)
      - RBFS (búsqueda primero-el-mejor recursiva)
      - MA* (A* con memoria acotada)
      - SMA* (MA* simplificada)
  - Funciones heurísticas
  - Búsqueda local y optimización.
  - Búsqueda online y exploración.

Búsqueda primero-el-mejor

Realizar la elección del nodo de *lista-abierta* que expandiremos de acuerdo a una **función de evaluación** [f(*n*)] que da el coste del camino menos costoso que va desde el nodo *n* al objetivo.

- Implementación:
  - Usar búsqueda-en-grafo con una **cola de prioridad** para la lista de candidatos a expandir (*lista-abierta*).

    Los nodos nuevos que se generan se insertan en la cola en orden ascendente de sus valores de f $\Rightarrow$ nodos con valores de f más pequeños se expanden primero.

- Rendimiento:

  Por definición es óptima, completa y tiene la menor complejidad posible, pero no es una búsqueda.

Problema: f es generalmente desconocida. Podemos usar solamente estimaciones de la distancia existente entre un nodo dado y el objetivo.

Búsqueda primero-el-mejor avariciosa, I

- Función de evaluación = función heurística

  f(n) = h(n)

  - h(n) = coste estimado del camino menos costoso desde el nodo n al estado objetivo

- Pseudocódigo:
  - Búsqueda primero-el-mejor general

```
function BÚSQUEDA-PRIMERO-EL-MEJOR (problema, Eval-Fn) devuelve una secuencia solución

  entradas:     problema, un problema
                Eval-Fn, una función de evaluación

  Fn-Encolamiento ← una función que ordena nodos por Eval-Fn
  return BÚSQUEDA-GENERAL (problema, Fn-Encolamiento)
```

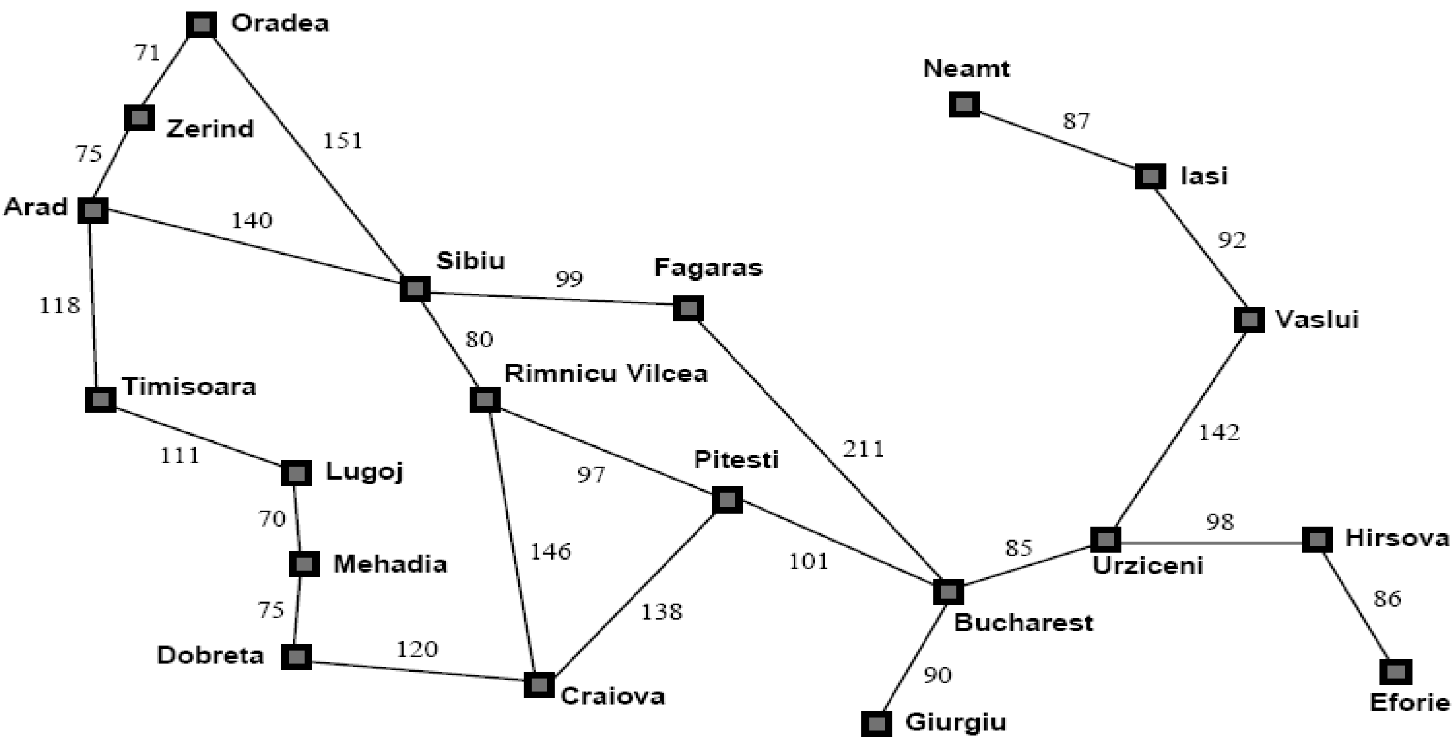
  - Búsqueda primero-el-mejor avariciosa

```
function BÚSQUEDA-PRIMERO-EL-MEJOR-AVARICIOSA (problema) devuelve una secuencia solución, o fallo

   return BÚSQUEDA-PRIMERO-EL-MEJOR (problema, h)
```

Problema: mapa de carreteras

Encontrar el mejor itinerario entre dos ciudades en Rumanía



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

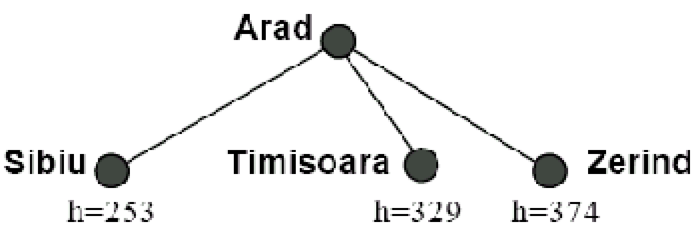
Función heurística (admisible, monótona)

h(n) = distancia en línea recta desde la ciudad n a **Bucarest**.

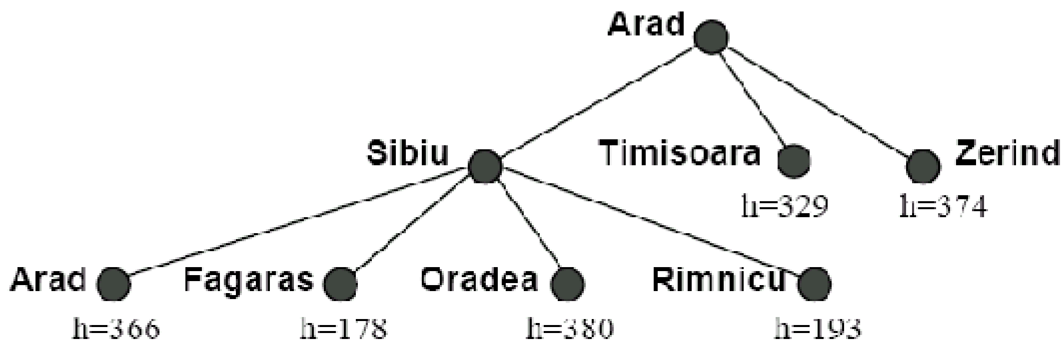
Búsqueda avariciosa para el problema del mapa de carreteras

Estado inicial:
Ciudad de salida

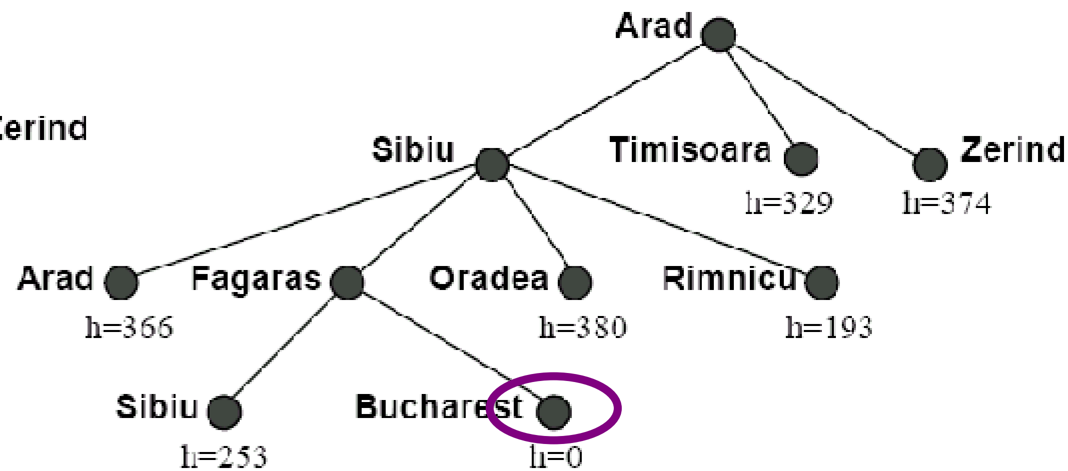
1. Expande Arad



2. Expande Sibiu (valor de h más pequeño, 253)



3. Expande Fagaras (valor de h más pequeño, 178)

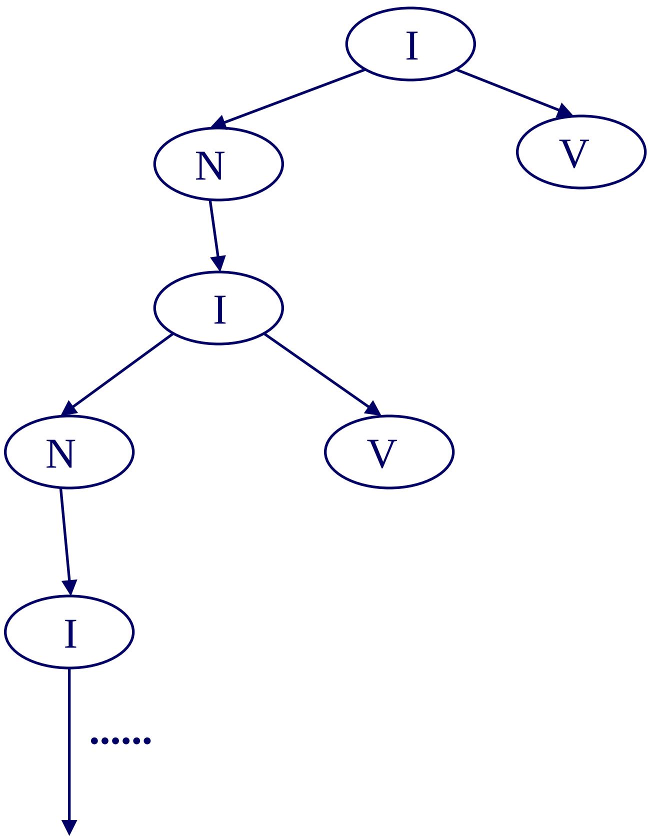


4. Objetivo alcanzado: Bucarest

Búsqueda avariciosa: estados repetidos

Si no tenemos en cuenta posibles bucles, la búsqueda primero-el-mejor avariciosa puede no llegar a encontrar una solución

Ejemplo: Encontrar el itinerario entre “**Iasi**” y “**Fagaras**”:



I: Iasi
N: Neamt
V: Vaslui

Búsqueda avariciosa: Eficiencia

- No es completa
- No es óptima
- Complejidad del peor caso:
  - Complejidad temporal : $O(b^m)$
  - Complejidad espacial: Todos los nodos se almacenan en memoria

$$O(b^m)$$

    - m=profundidad máxima del árbol

Búsqueda A*

- Función de evaluación

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

  - $g(n)$ = coste real del camino entre el estado inicial y el nodo n
  - $h(n)$ = coste estimado del camino menos costoso desde n al estado objetivo
  - $f(n)$ = coste estimado de la solución menos costosa (camino desde el estado inicial al estado objetivo) que pasa por n:

function BÚSQUEDA-PRIMERO-EL-MEJOR (*problema*, *Eval-Fn*) **devuelve** una secuencia solución

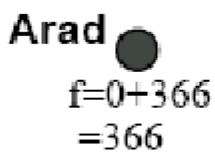
entradas: *problema*, un problema
Eval-Fn, una función de evaluación

Fn-Encolamiento ← una función que ordena nodos por *Eval-Fn*
return BÚSQUEDA-GENERAL (*problema*, *Fn-Encolamiento*)

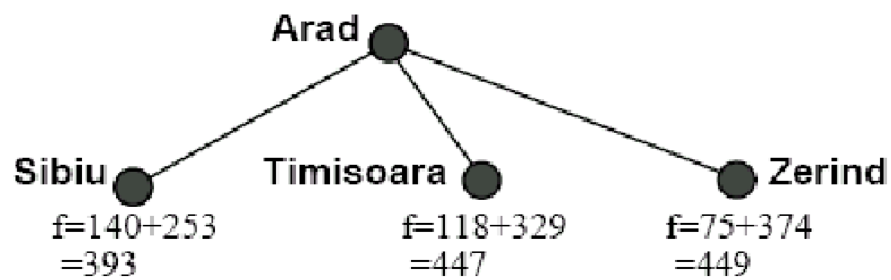
function BÚSQUEDA-A* (*problema*) **devuelve** una secuencia solución, o fallo

return BÚSQUEDA-PRIMERO-EL-MEJOR (*problema*, *g+h*)

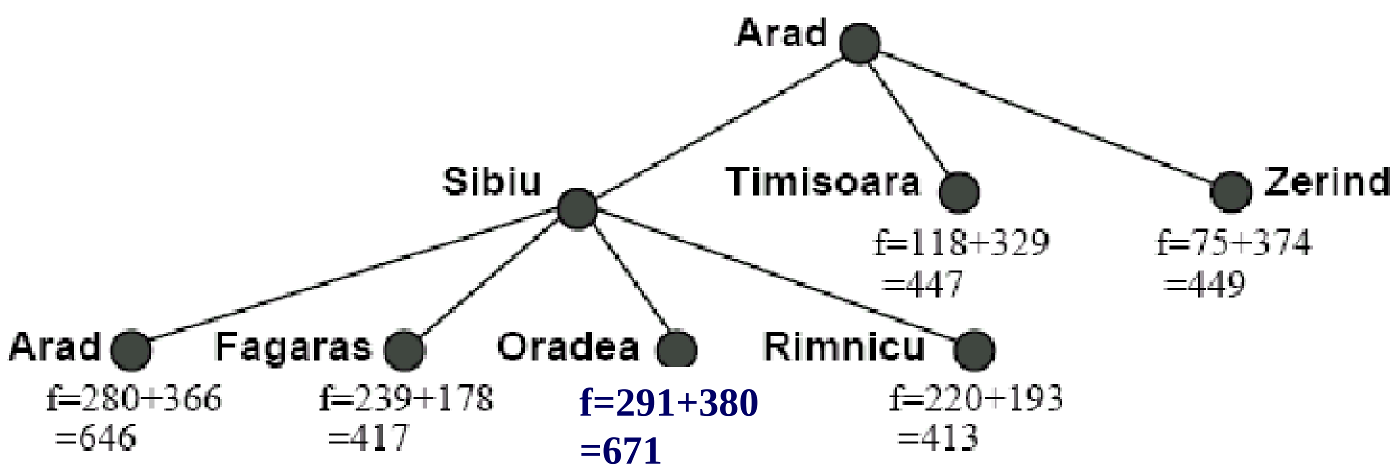
Búsqueda A*: Mapa de carreteras, I



1. Expande Arad

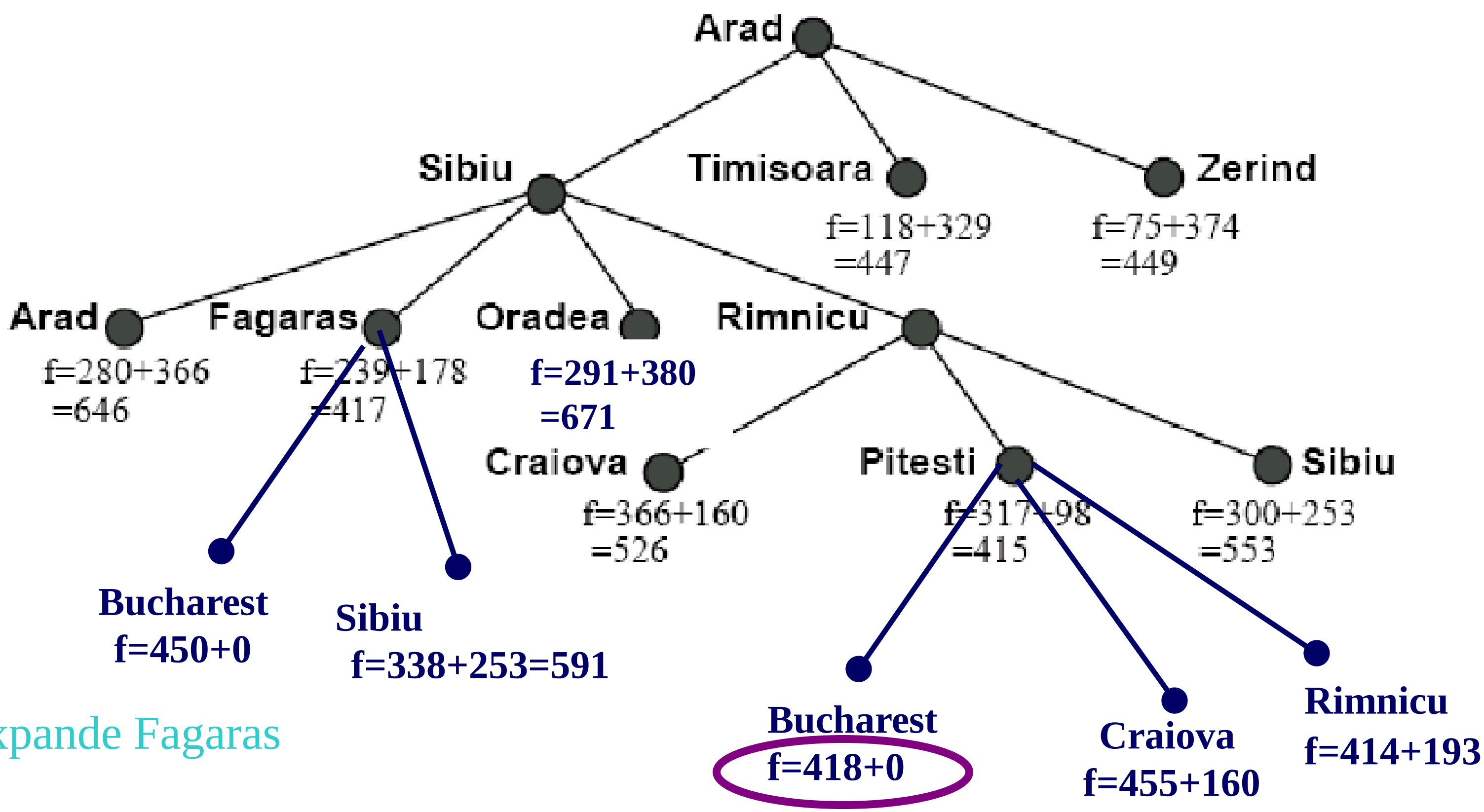


2. Expande Sibiu



Búsqueda A*: Mapa de carreteras, II

3. Expande Rimnicu (f más pequeño, 413)



5. Expande Fagaras

4. Expande Pitesti

6. Objetivo encontrado: Bucarest

Heurística admisible

La **función heurística** h(n) es **admisible** si **nunca sobreestima** el **coste** de **alcanzar el objetivo** (es decir, es una estimación optimista).

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

- h*(n)= coste real del camino óptimo (es decir, con menor coste) desde *n* al objetivo.
- Ejemplo: En el problema del mapa de carreteras, la distancia en línea recta es una función heurística admisible.

⌘ TEOREMA: Si se usa **búsqueda-en-árbol**, y **h es admisible** $\Rightarrow$ **A* es completa y óptima**

Demostración:

- C* : coste de la solución óptima
- Considérese G_2 un nodo objetivo subóptimo (i.e. $g(G_2) > C^*$, $h(G_2) = 0$) que está en la frontera del árbol de búsqueda:

  $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^* \quad \Rightarrow \quad f(G_2) > C^*$ **(1)**

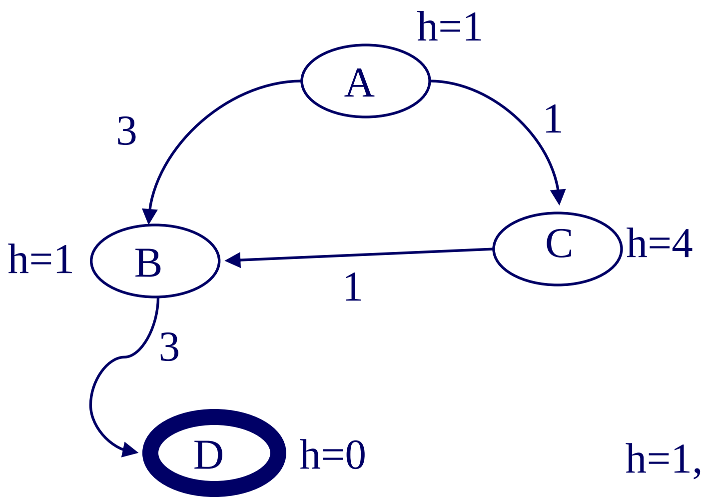
- Considérese el nodo n del conjunto frontera del árbol de búsqueda que está en un camino solución óptimo.
  - Dado que n está en el camino solución óptimo, $g(n) = g^*(n)$
  - Dado que h is admisible: $h(n) \leq h^*(n)$

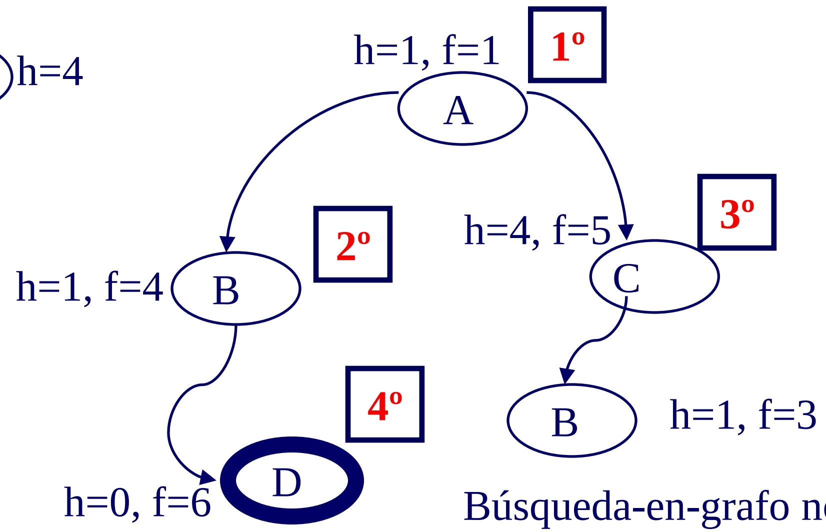
  $f(n) = g(n) + h(n) \leq g^*(n) + h^*(n) = C^* \Rightarrow \quad f(n) \leq C^*$ **(2)**

(1)+(2) $\Rightarrow$ $f(n) \leq C^* < f(G_2)$ y se explora n antes que G_2

A* + heurística admisible

⌘ Si se usa **búsqueda-en-grafo**, **A* puede no ser óptima** incluso si **h es admisible:** Se pueden generar soluciones subóptimas si el camino óptimo a un estado repetido no es el que primero se genera.





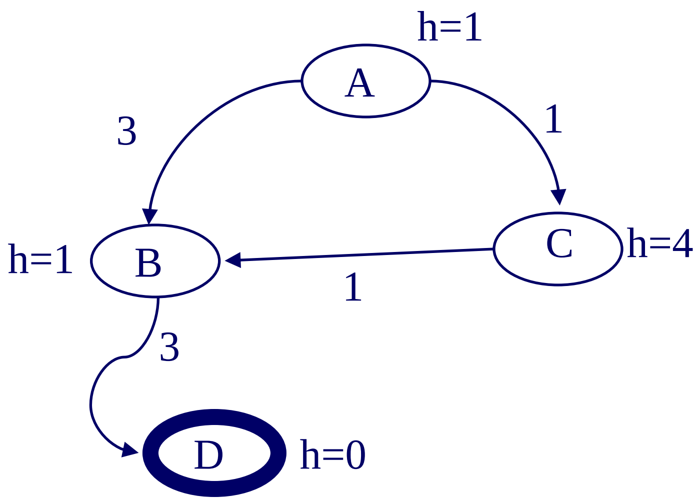
La solución encontrada es subóptima: coste = 6

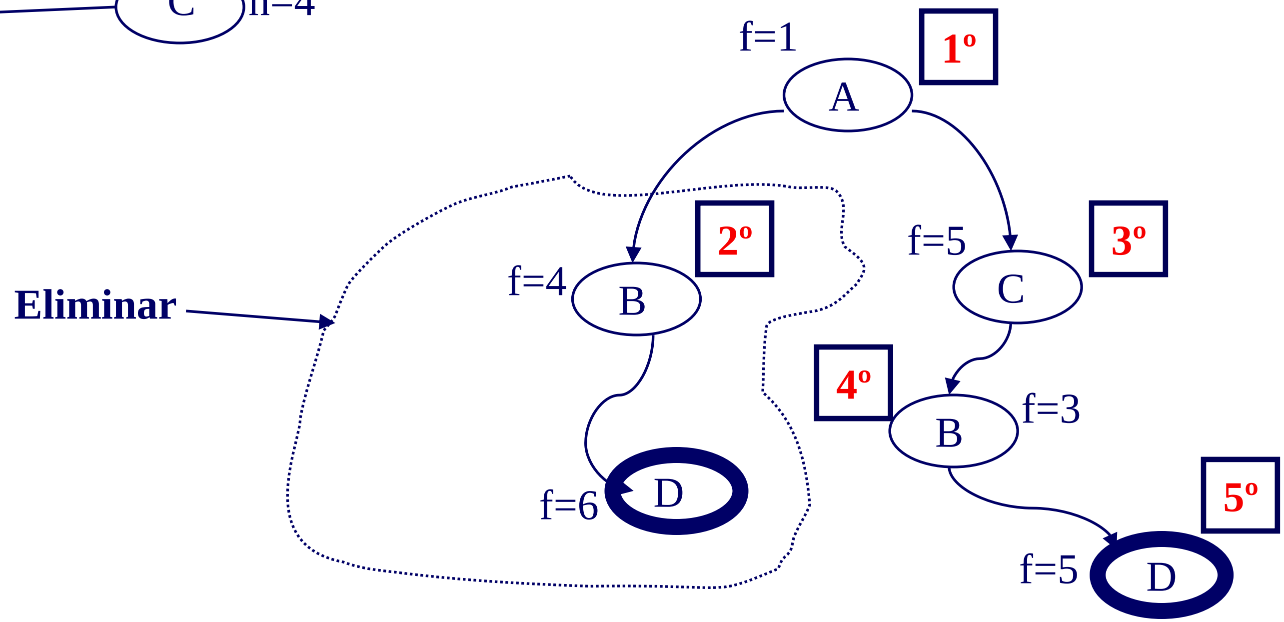
Búsqueda-en-grafo no expande B (B está en lista-cerrada)

A* + heurística admisible

<u>Solución</u>: Descartar el **camino** con coste más alto.

- Aumenta la complejidad del algoritmo: necesita eliminar de *lista-abierta* el nodo con coste más alto y sus descendientes.





La solución encontrada es óptima: coste = 5

Heurística monótona (o "consistente")

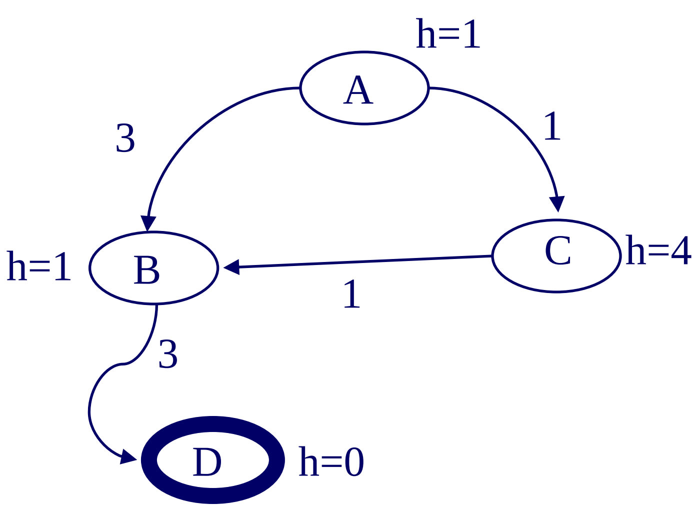
- Una **función heurística** h(n) es **monótona** si se satisface la siguiente **desigualdad triangular:**

$$h(n) \leq \text{coste}(n \to n') + h(n'),$$
$$\forall n, n' [n' \text{ sucesor de n}]$$

  - Ejemplo: Para el problema del mapa de carreteras, la distancia en línea recta es una función heurística monótona.
- Si **h es monótona** $\Rightarrow$ **h es admisible**

[Ejercicio: Demostrar esto]

- Hay heurísticas admisibles que no son monótonas



A* + heurística monótona

- TEOREMA:

  Si **h es monótona** $\Rightarrow$ los valores de **f(n)** a lo largo del camino buscado por A* son **no-decrecientes**

  Demostración:

  Supongamos que n′ es un sucesor de n

  $$f(n') = g(n') + h(n') = g(n) + coste(n \rightarrow n') + h(n')$$

  $$\geq g(n) + h(n) = f(n) \qquad \Rightarrow \qquad \mathbf{f(n') \geq f(n)}$$

- TEOREMA:

  Si **h es monótona** $\Rightarrow$ **A*** usando **búsqueda-en-grafo es completa y óptima.**

  Demostración:

  Dado que f(n) es no-decreciente el primer nodo objetivo expandido debe ser el correspondiente a la solución óptima.

A* + heurística monótona

- TEOREMA: Si h es monótona y A* ha expandido un nodo n, se cumple $g(n)=g^*(n)$

  Demostración: Consideremos el problema de búsqueda relacionado con el mismo estado inicial y con nodo n como nodo objetivo.

  Definamos la nueva heurística para este nuevo problema:

  $$h'(m) = h(m) - h(n), \qquad \forall\, m \,/\, f(m) \leq f(n).$$

  Dado que la diferencia entre h y h' es una constante:

  - Las búsquedas (A* con h) y (A* con h') empezando desde el mismo estado inicial, exploran la misma secuencia de nodos antes de expandir n.
  - h' es una heurística monótona para el nuevo problema.

  Dado que A* con una heurística monótona es completa y óptima, la búsqueda (A* con h') encuentra el camino óptimo entre el estado inicial y el nodo n. Este camino será también el óptimo para la búsqueda (A* con h) $\Rightarrow$ $g(n)=g^*(n)$.

- TEOREMA: **A*** es óptimamente eficiente.

  Para una heurística dada, ningún otro algoritmo expandirá menos nodos que los que expande A* (excepto posibles empates)

  Demostración: si hay nodos que no se expanden entre el origen y la curva de nivel óptima, no está garantizado que el algoritmo encuentre la solución óptima.

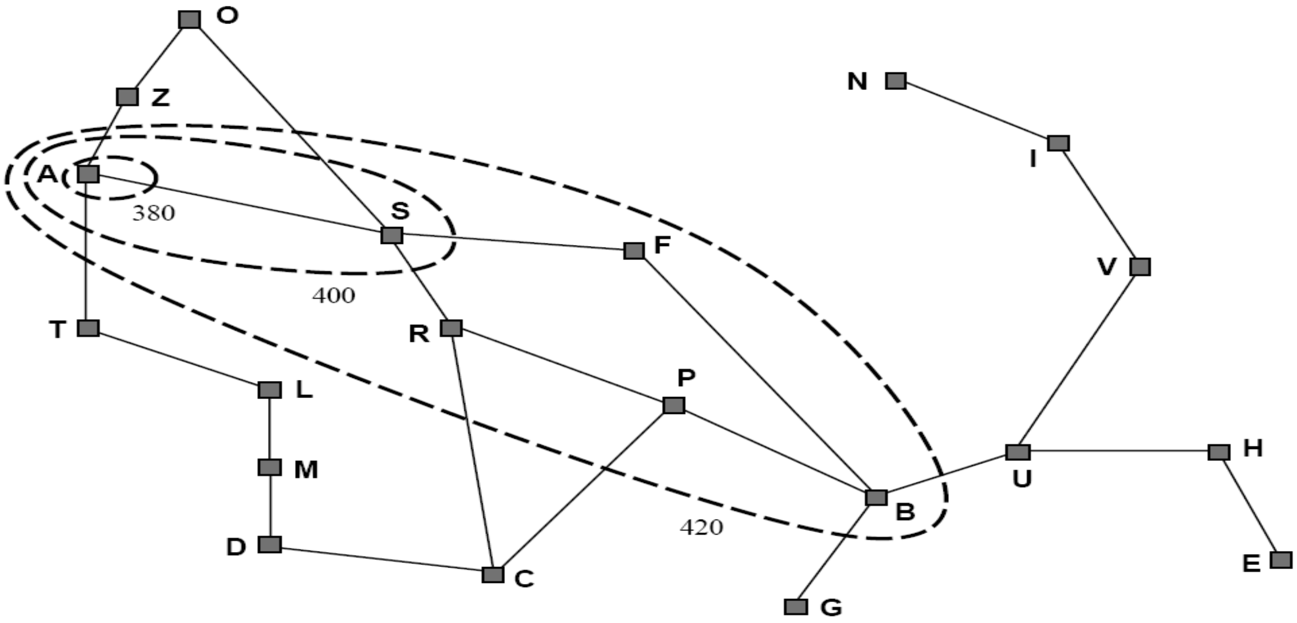
  [Dechter, Pearl, 1985]

A* + heurística monótona

Si h es una heurística monótona, la exploración se realiza en curvas de nivel con valores crecientes de f(n).

• En una búsqueda de coste uniforme [h(n) =0], las curvas de nivel son "concéntricas" alrededor del estado de partida.

• En heurísticas mejores estas curvas forman bandas que se extienden hacia el estado objetivo.



Complejidad de A*

- Complejidad temporal
  - **Exponencial para h arbitrario**

$$O(b^{\tilde{d}}), \qquad \tilde{d} = \frac{C^*}{\varepsilon}$$

C* = Coste óptimo; ε = mínimo coste por acción

  - **Subexponencial** si $|h(n) - h^*(n)| \le O(\log h^*(n))$

- Complejidad espacial:
  - Igual a la complejidad temporal (se mantienen todos los nodos en memoria).
  - Normalmente es el factor limitante.

Debido al gran requerimiento de memoria, A* no es un algoritmo práctico para problemas grandes.

Búsqueda IDA*

- Búsqueda A* con profundidad iterativa

  Realizar una búsqueda primero-en-profundidad con una profundidad límite de f, e ir aumentando este valor.

  - n_0= nodo inicial
  - Sean

$$C_0 = f(n_0);$$

$$C_k = \min_n \{ f(n) \mid f(n) > C_{k-1} \}; \quad k = 1,2,\ldots$$

  - Iteración k: expandir todos los nodos que cumplan $\{n \mid f(n) \leq C_k\}$

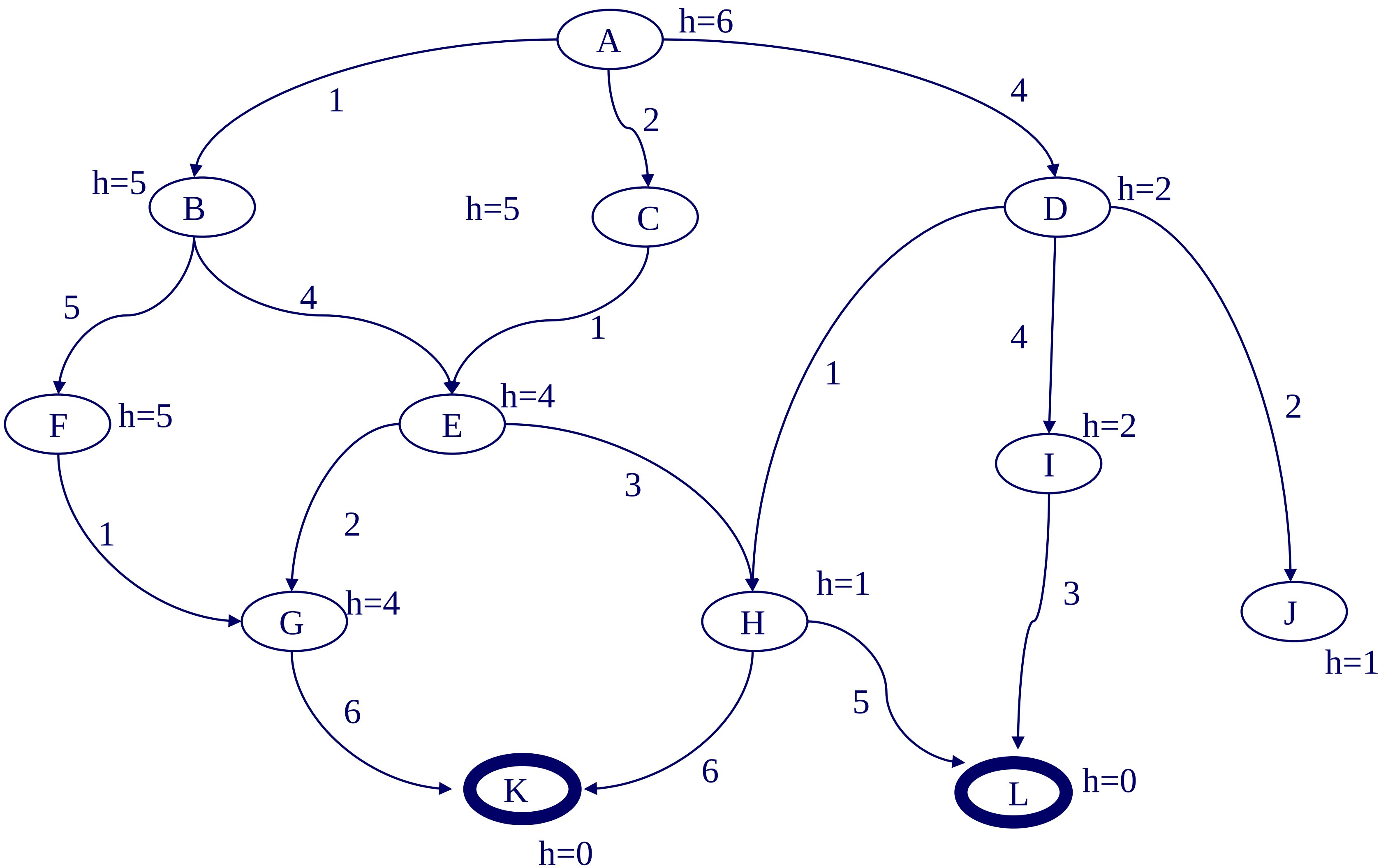
- Propiedades
  - Si h es monótona $\Rightarrow$ IDA* es completa y óptima.
  - Complejidad espacial: O(b·d); d = C*/ε.

    C* = coste óptimo; ε = mínimo coste por acción
  - Complejidad temporal: En el peor caso, un sólo nodo es expandido en cada iteración. Asumiendo que el último nodo que se expande es el nodo solución, el número de iteraciones es 1+2+...+N ~O(N^2)
    - Variación IDA*:

$$C_k = f(n_0) + k\Delta C; \quad k \geq 0$$

    - Número de iteraciones ~ O(C*/ΔC)

Ejemplo (A*, IDA*)



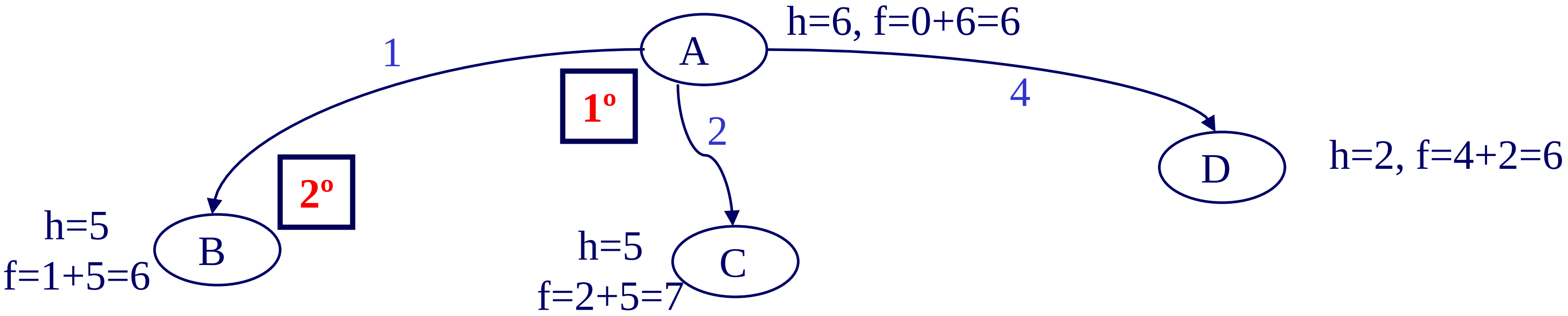
Estados finales: K, L

A* + búsqueda en árbol

A h=6, f=0+6=6

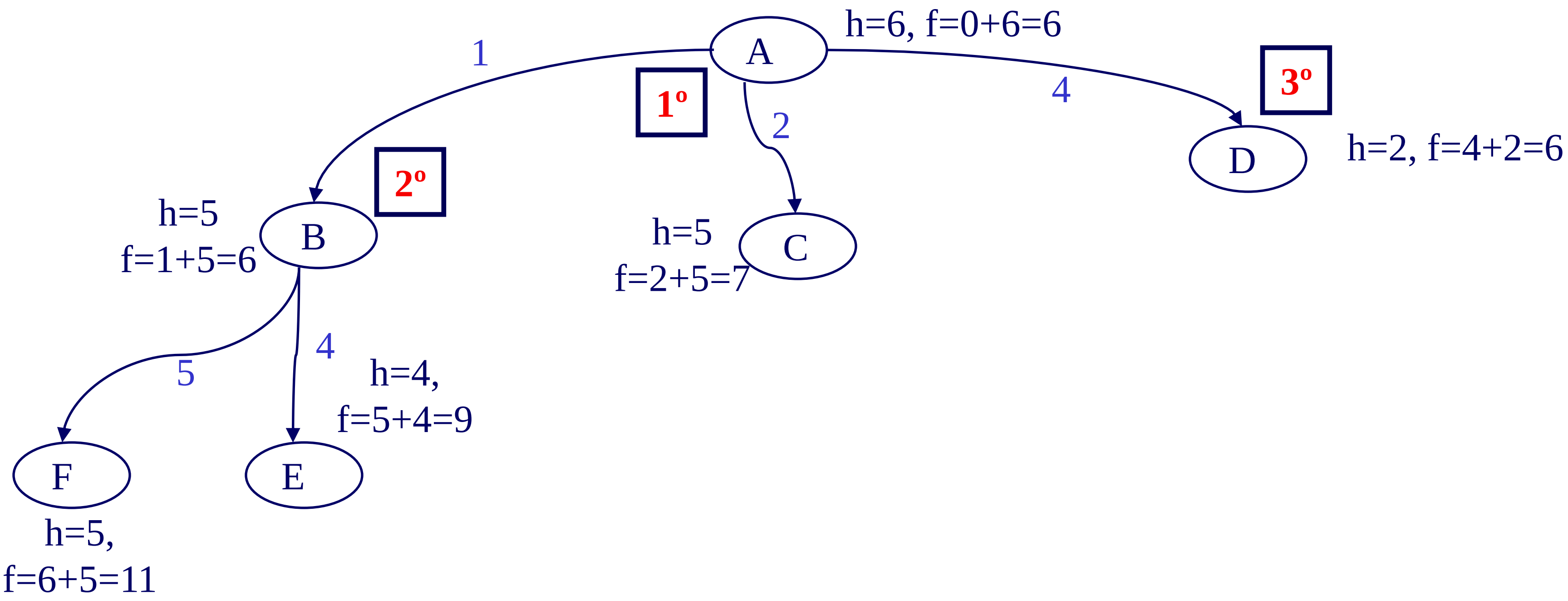
1º

A* + búsqueda en árbol



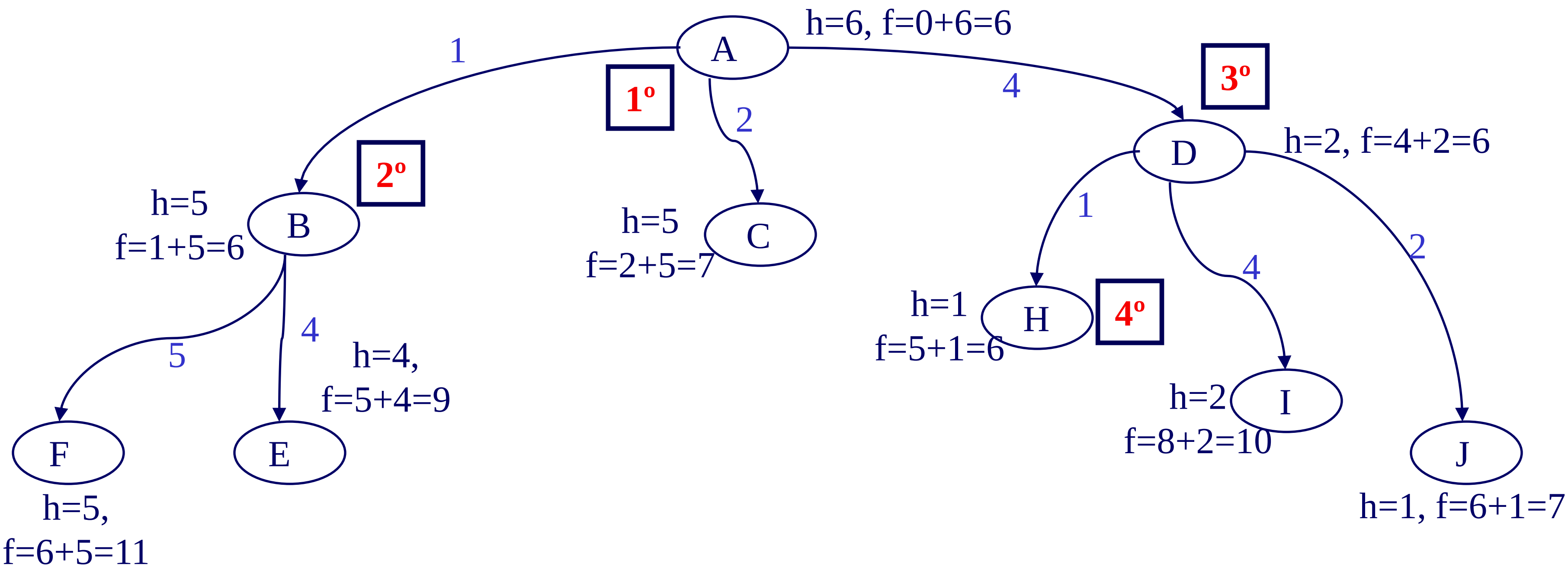
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



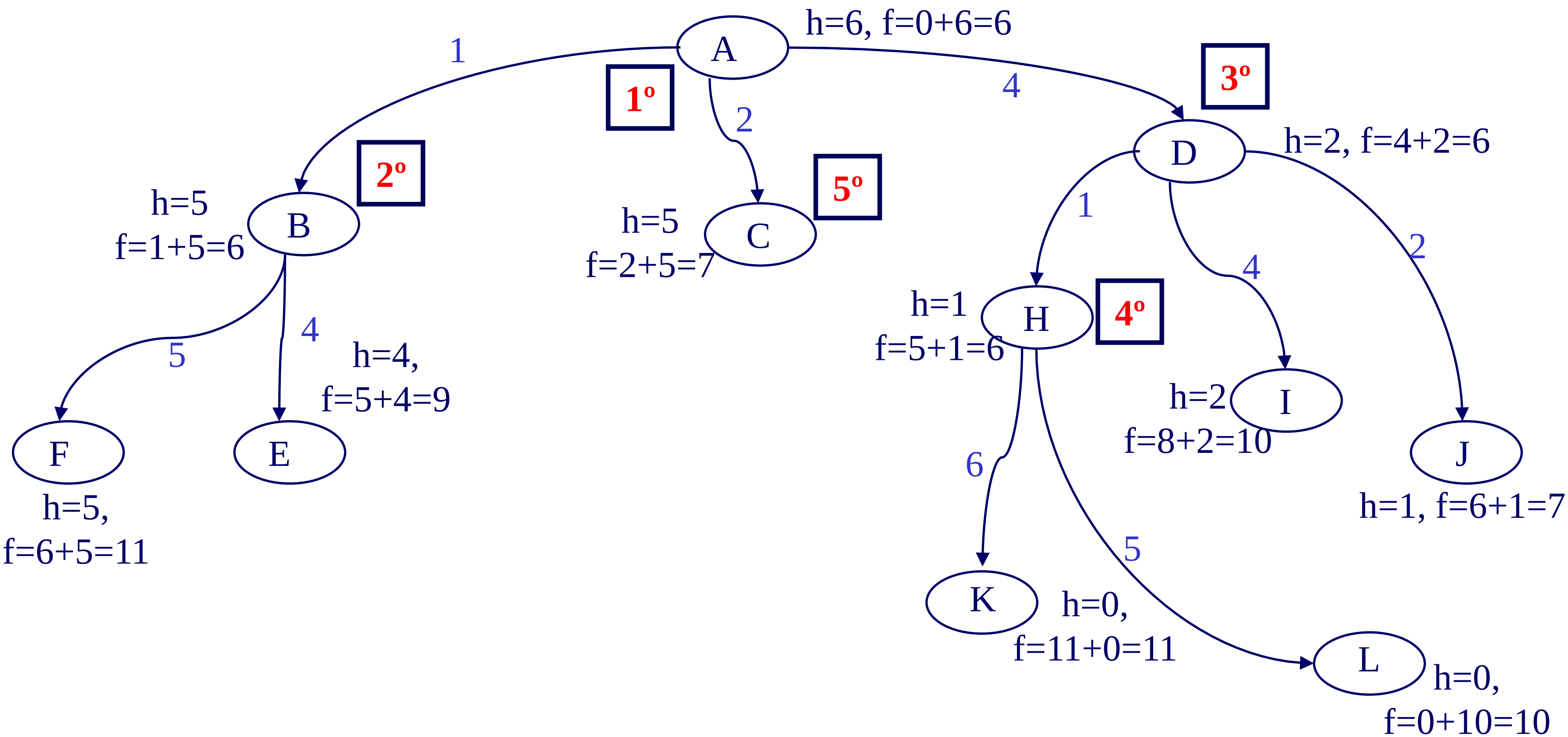
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



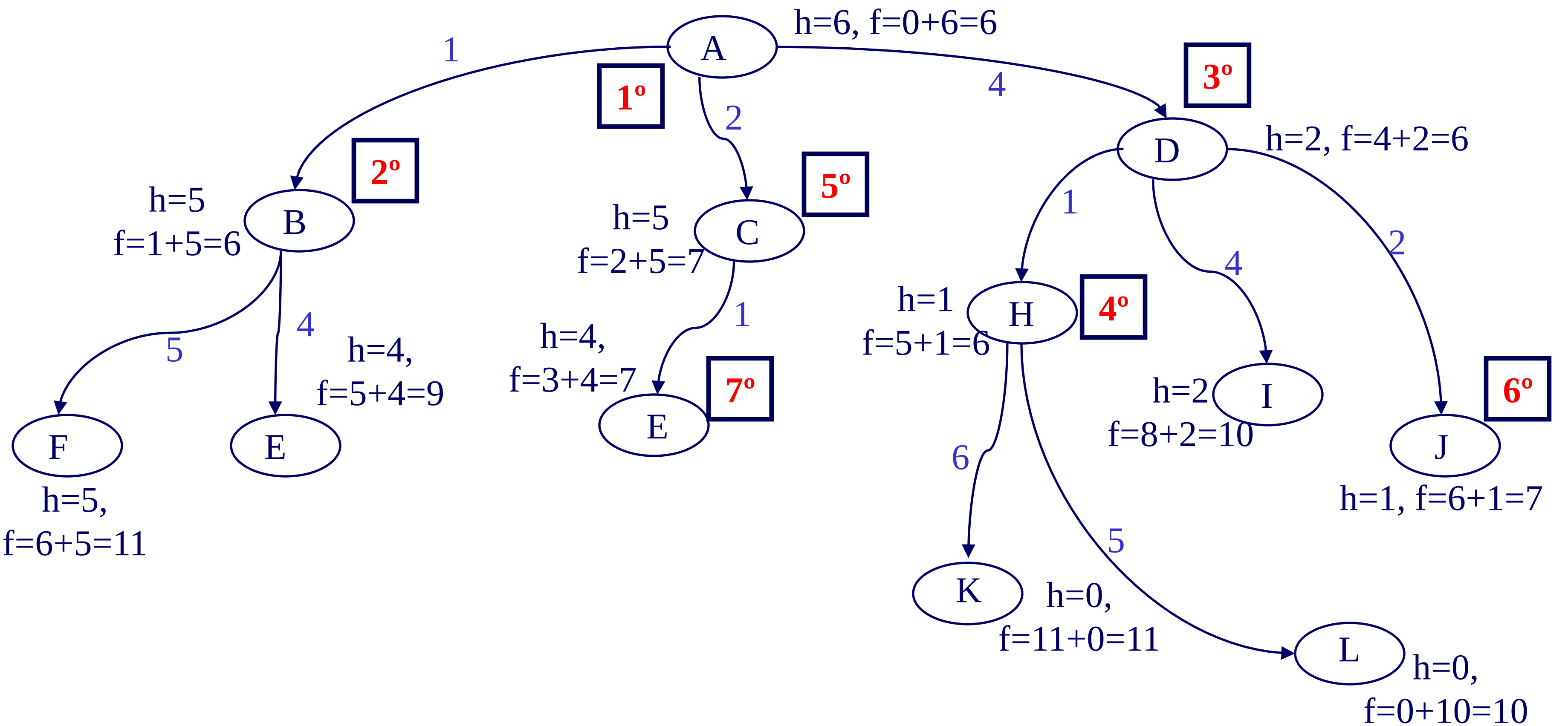
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



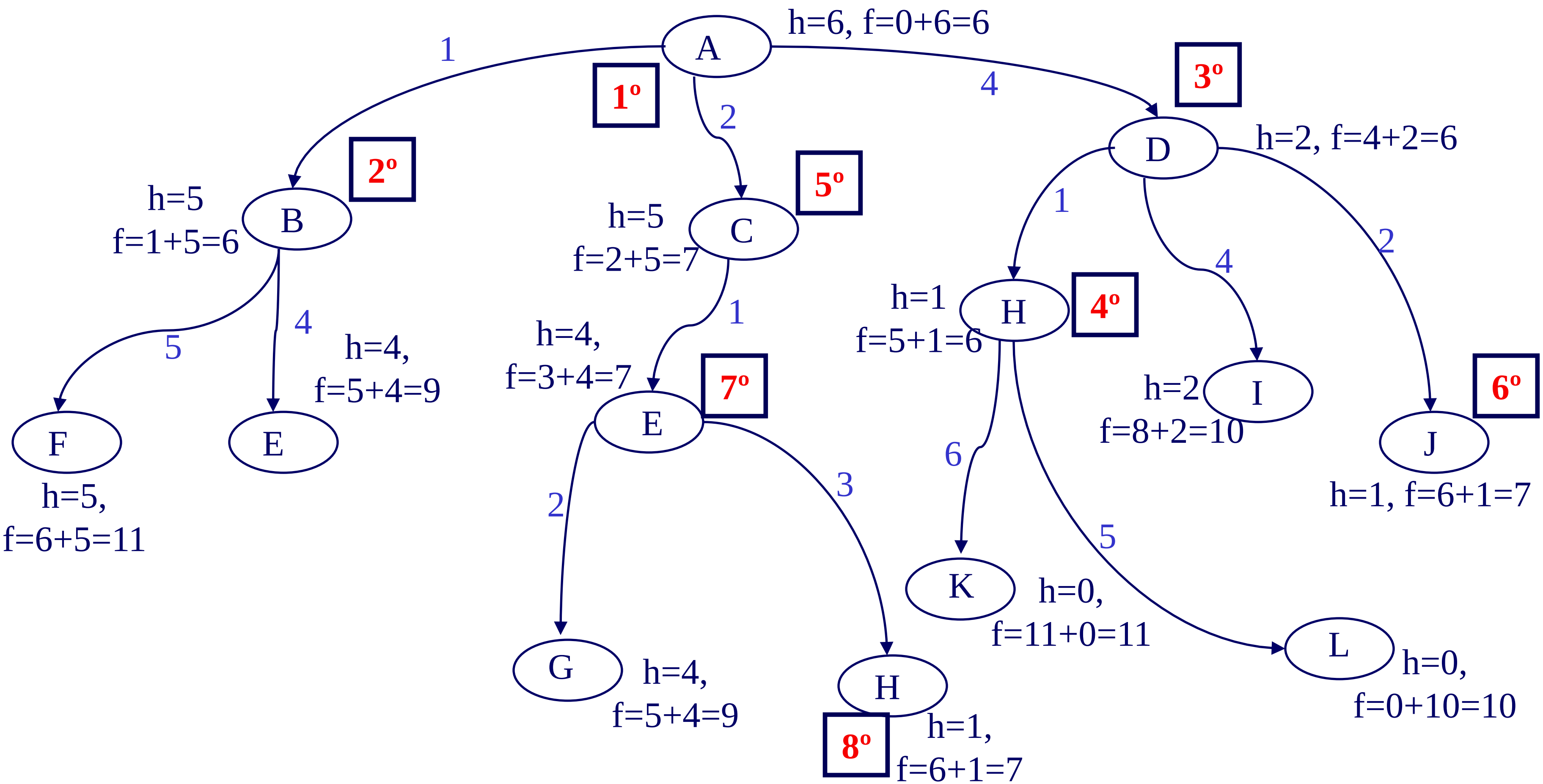
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



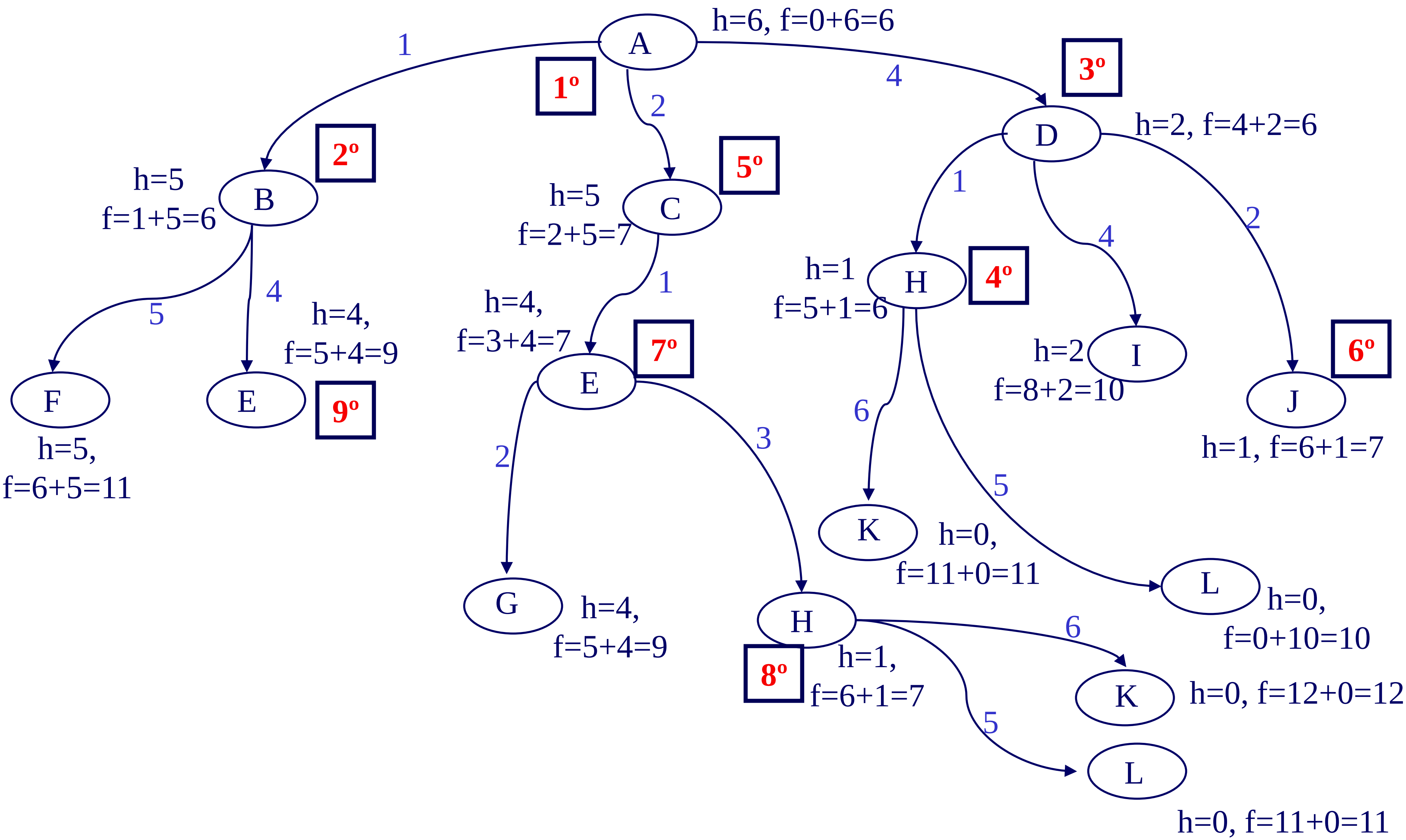
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



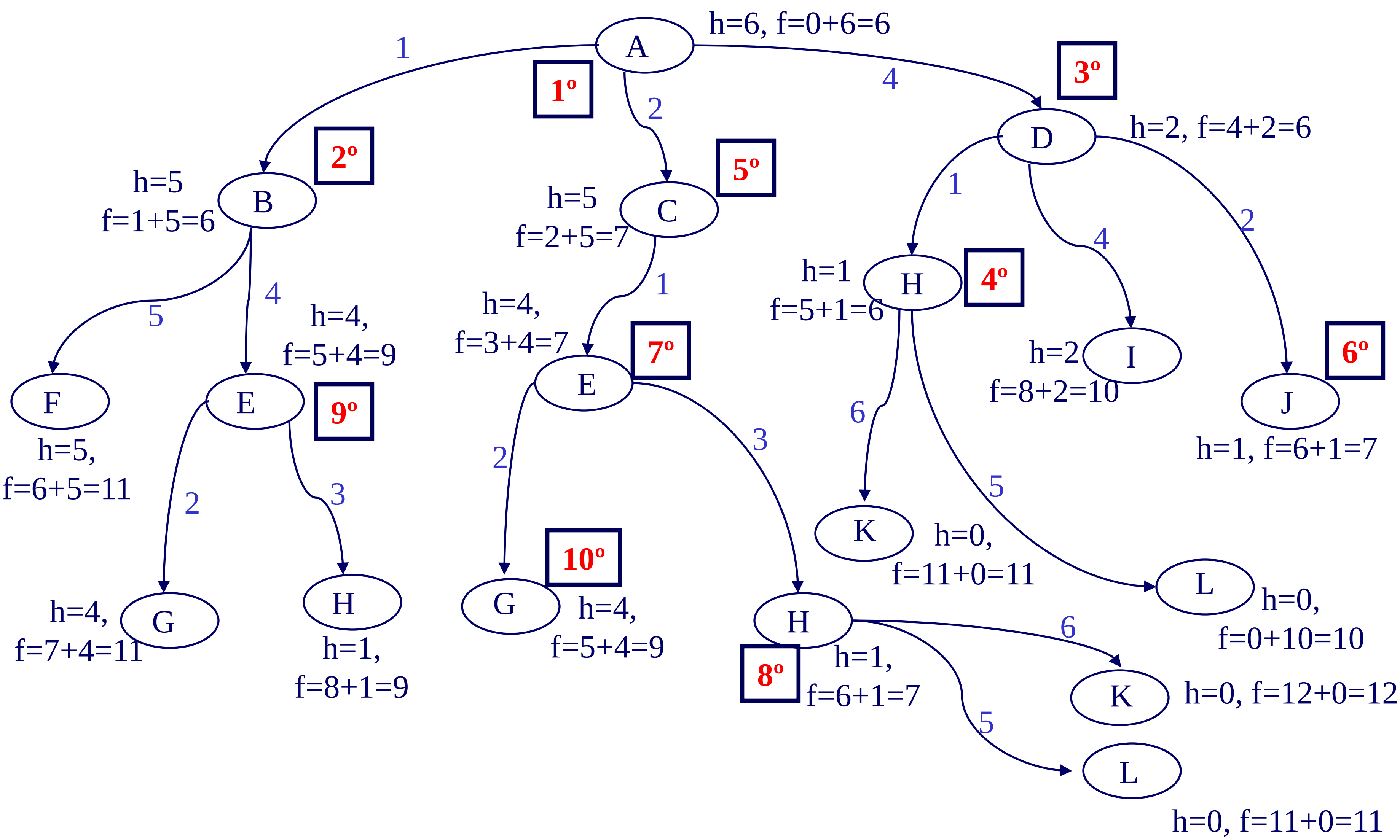
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



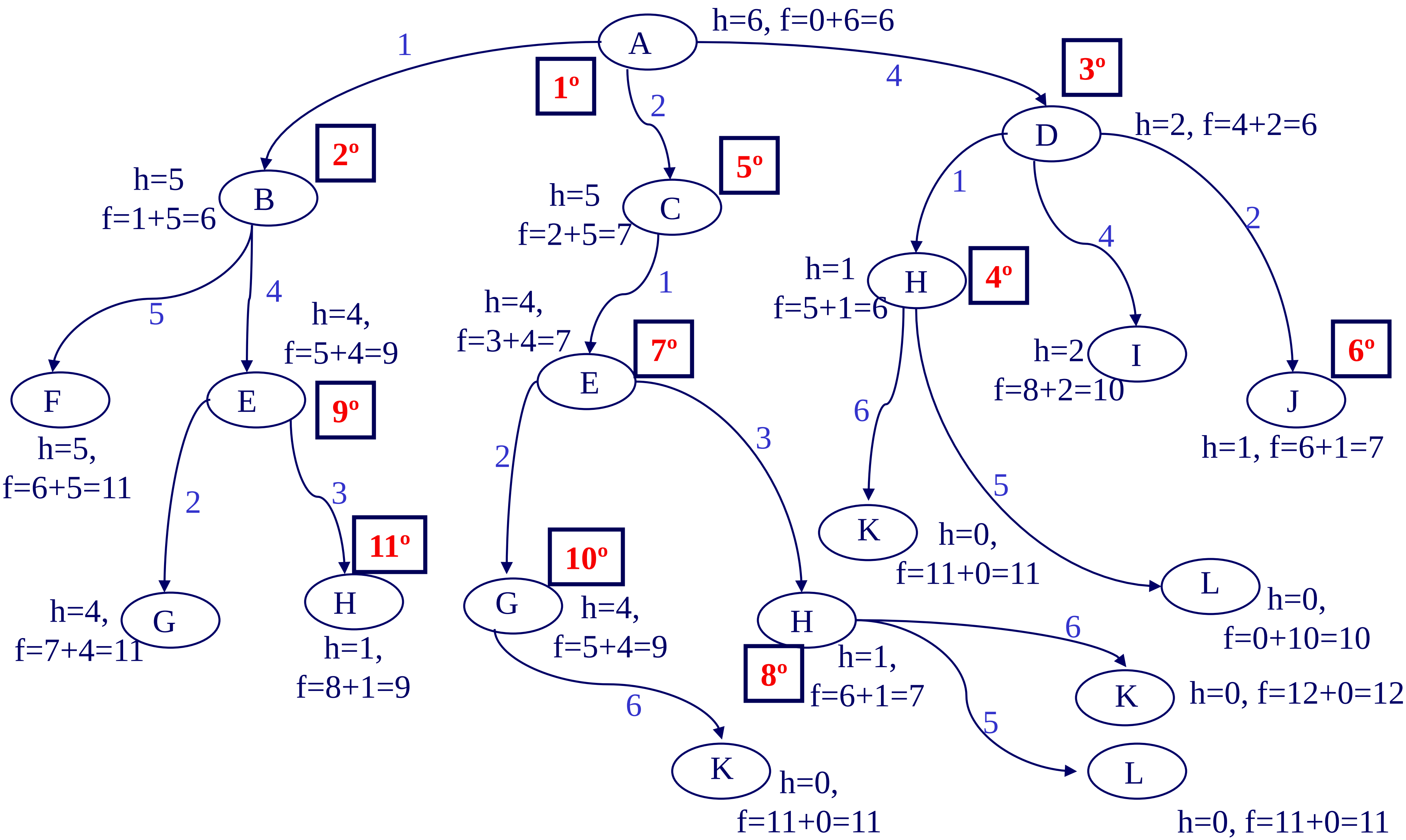
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



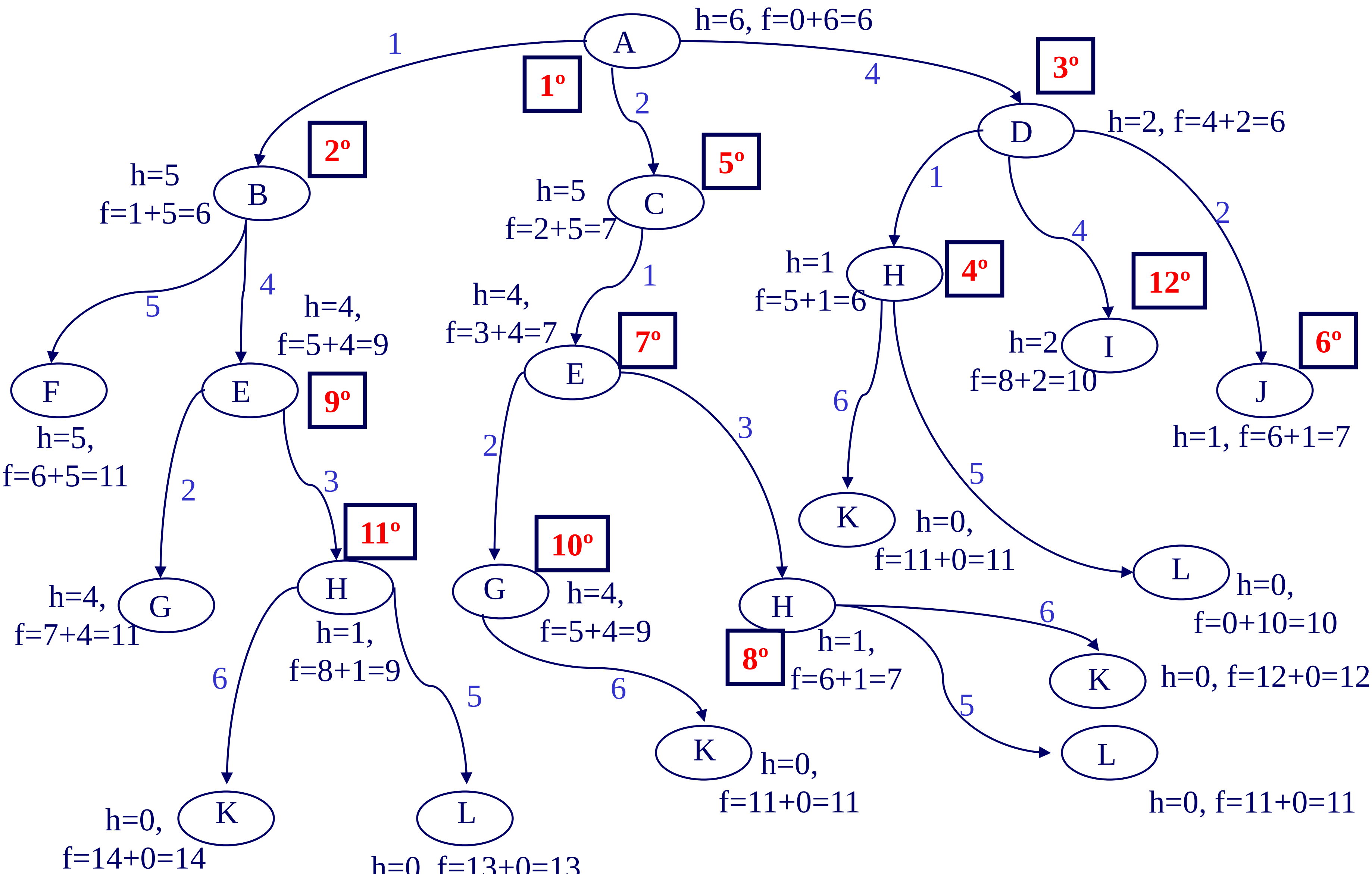
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



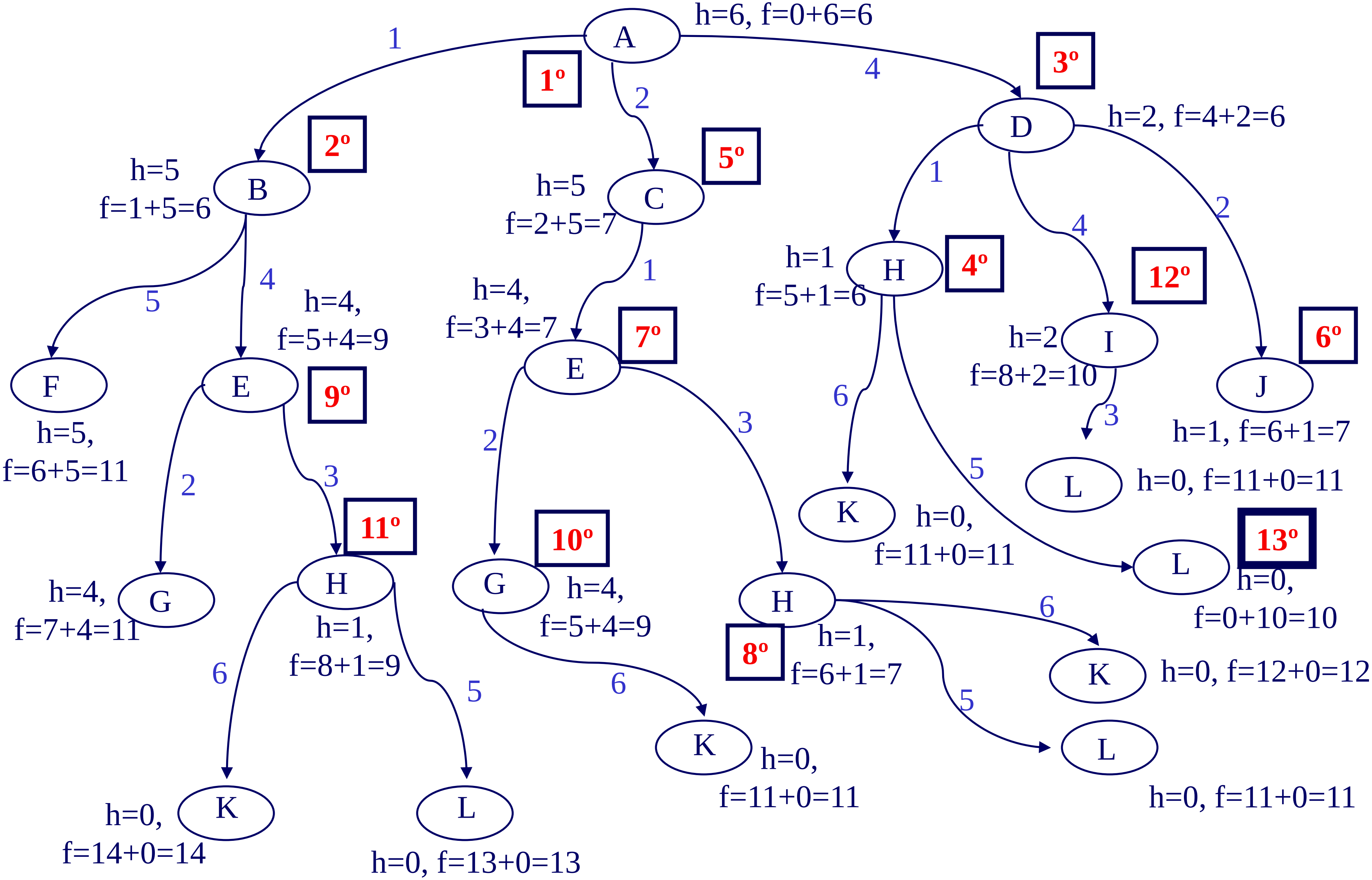
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



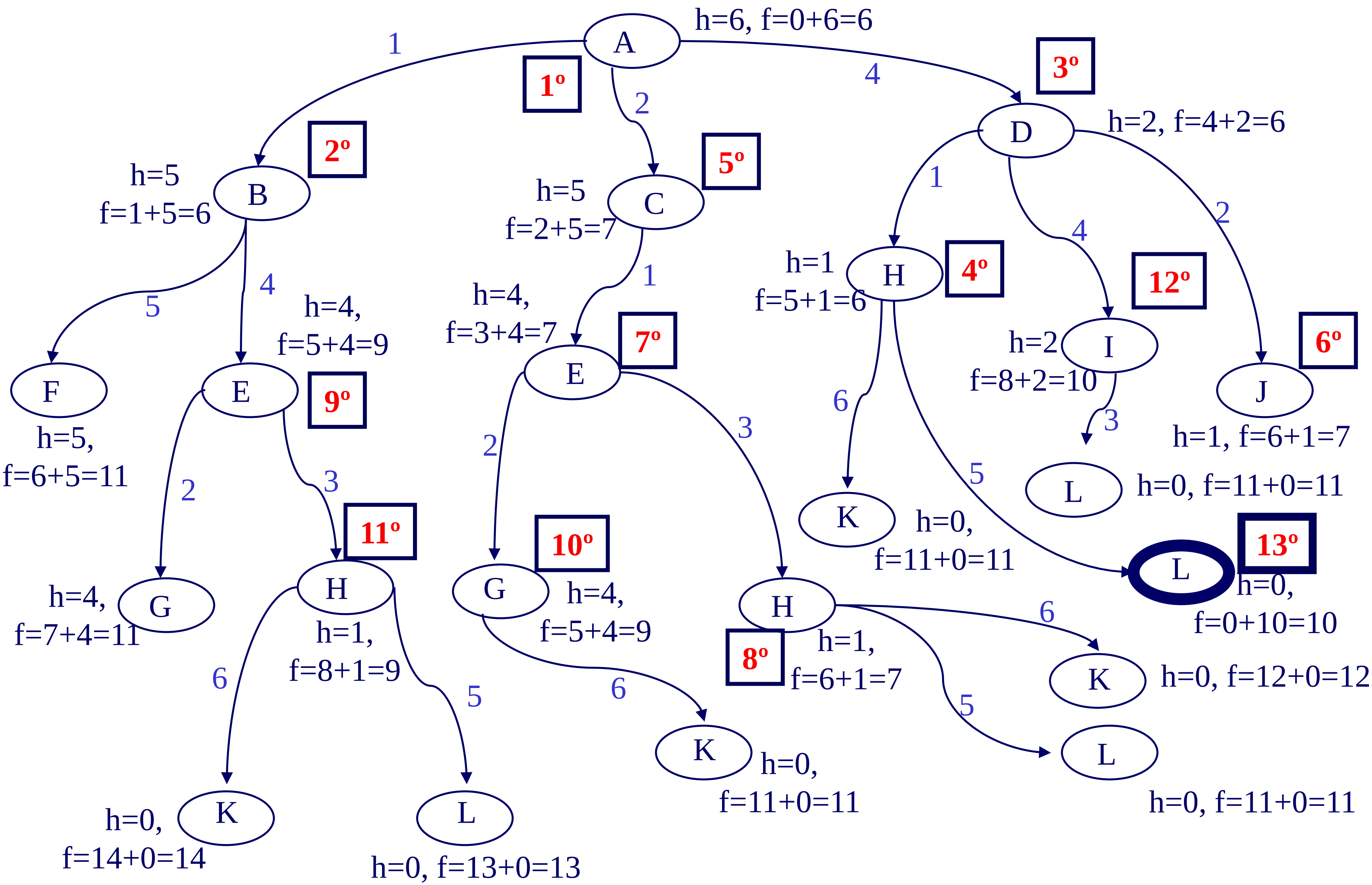
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



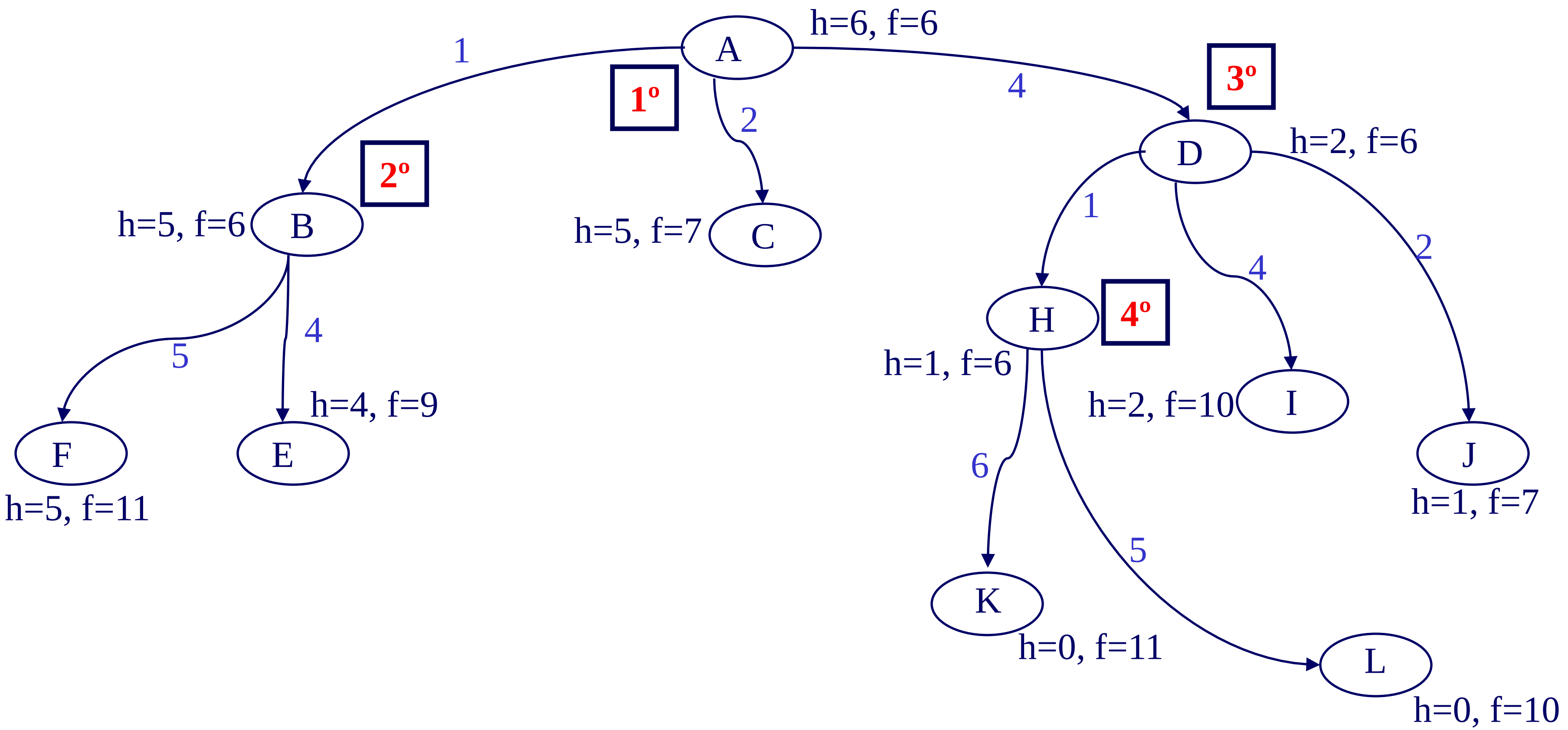
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



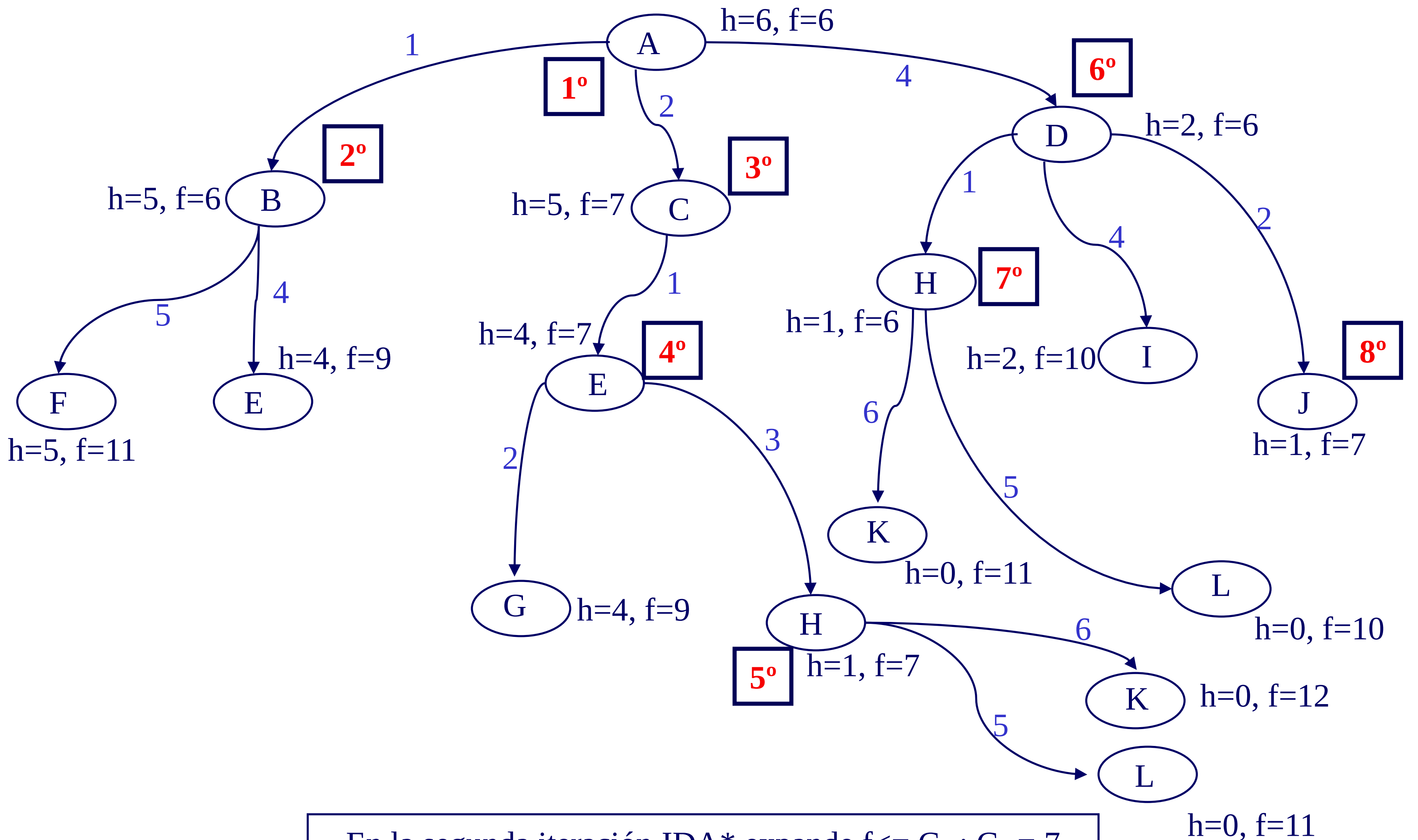
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

IDA* + búsqueda en árbol, I



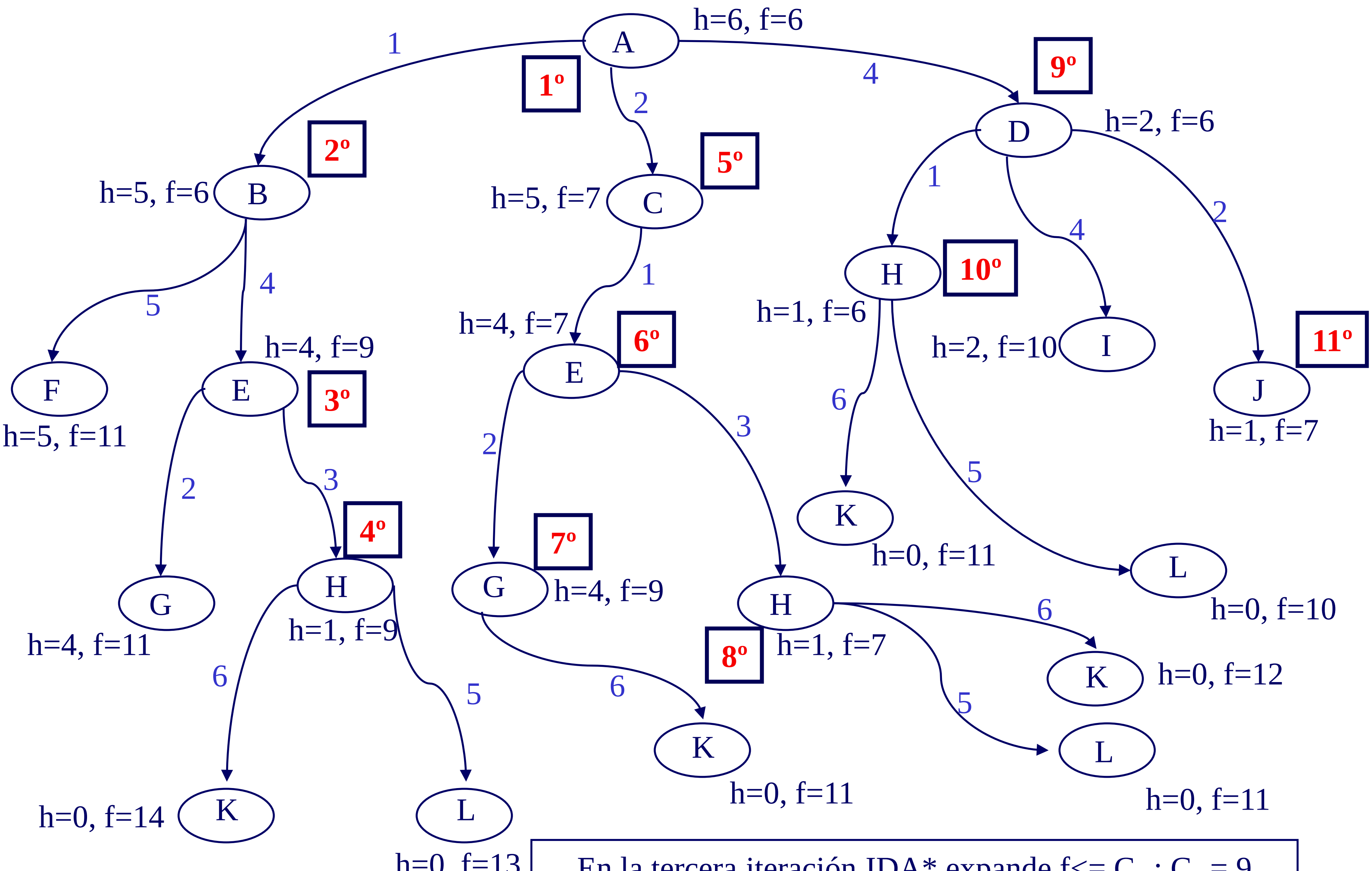
En la primera iteración IDA* expande f<= C_0 ; C_0 = 6
C_1 = 7

IDA* + búsqueda en árbol, II



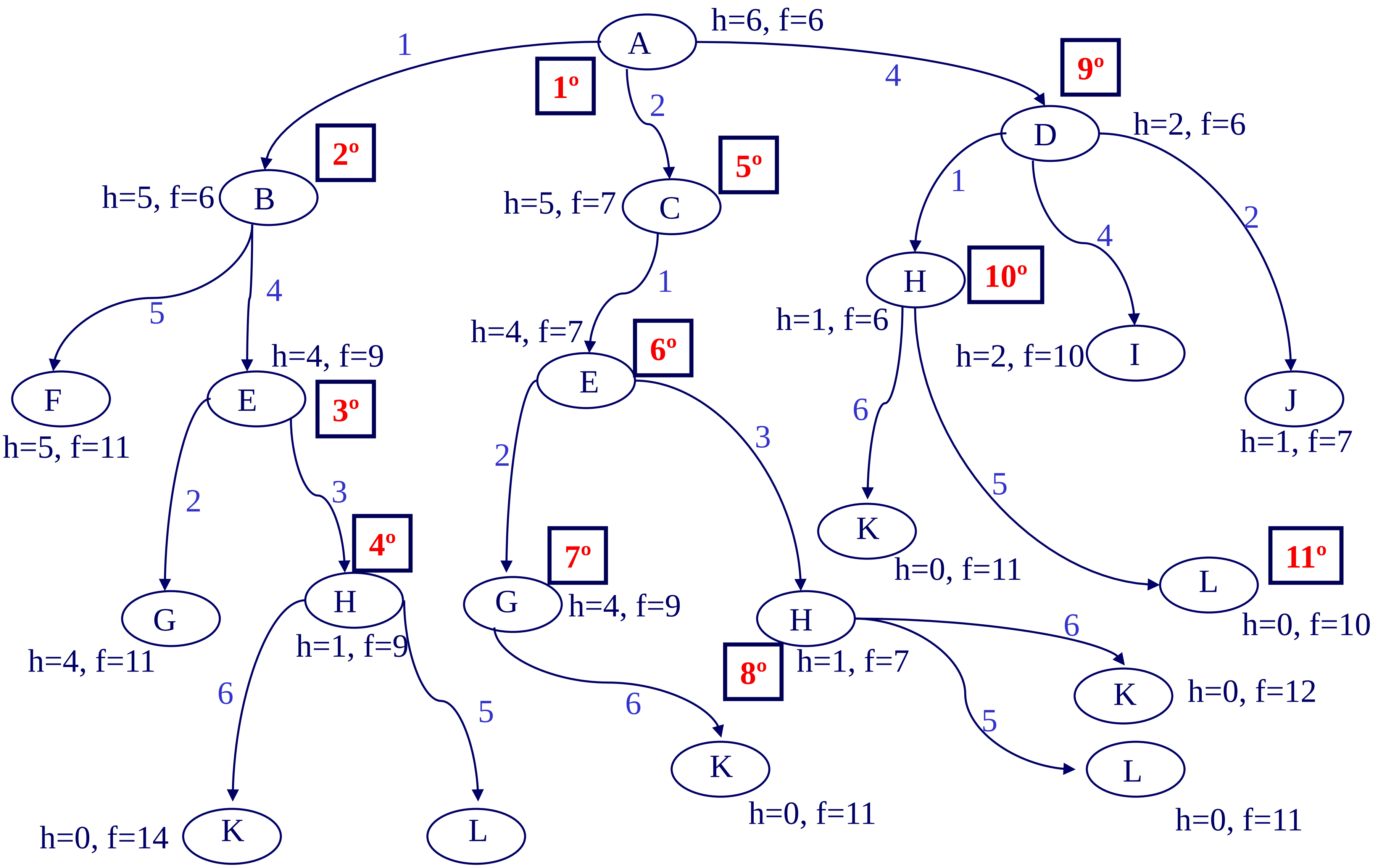
En la segunda iteración IDA* expande f<= C_1 ; C_1 = 7
C_2 = 9

IDA* + búsqueda en árbol, III



En la tercera iteración IDA* expande f<= C_2 ; C_2 = 9
C_3 = 10

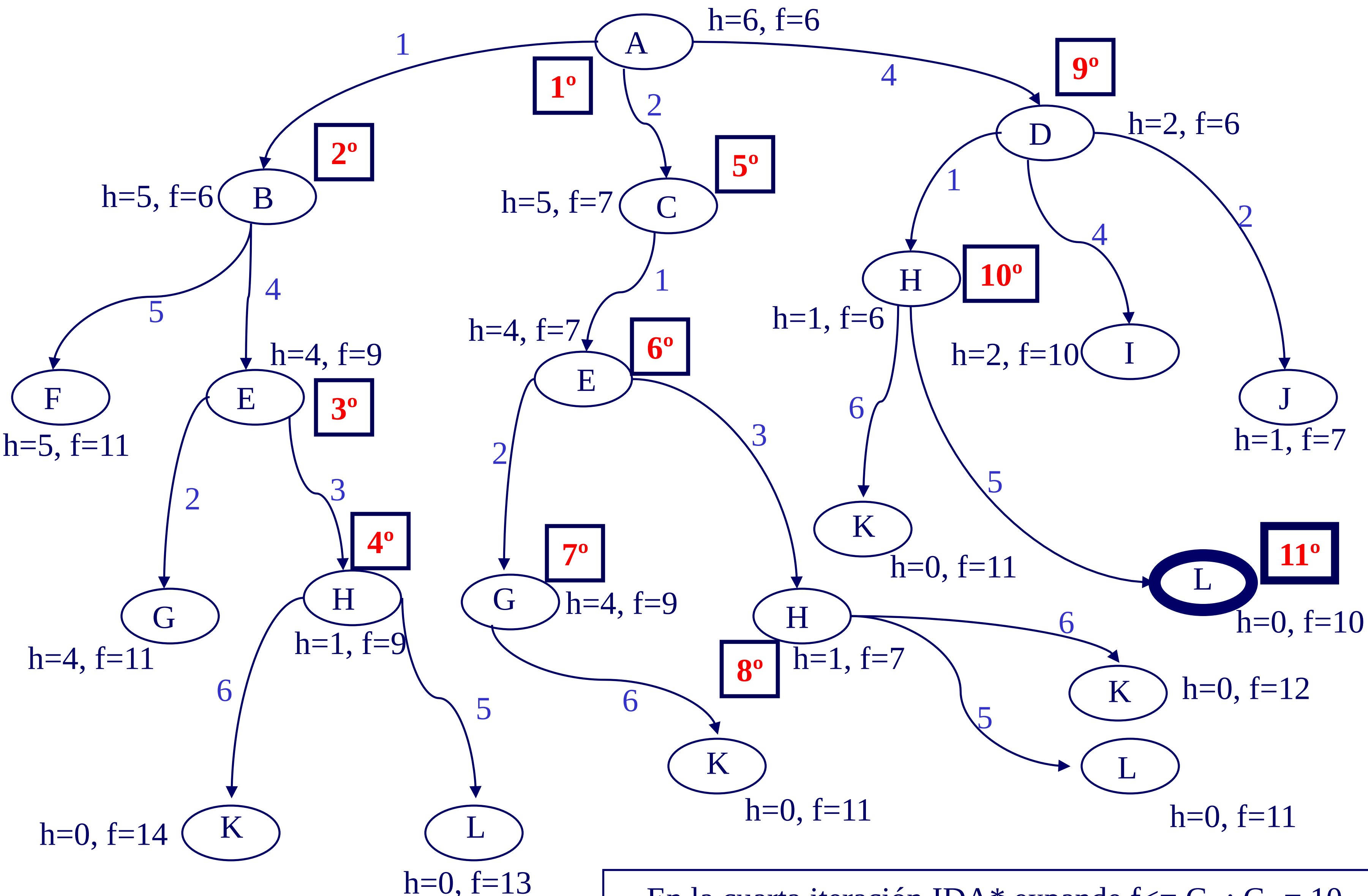
IDA* + búsqueda en árbol, IV



En la cuarta iteración IDA* expande f<= C_3 ; C_3 = 10

Objetivo encontrado

IDA* + búsqueda en árbol, IV



En la cuarta iteración IDA* expande f<= C_3 ; C_3 = 10

Objetivo encontrado

Evaluación de la heurística

- Factor de ramificación efectivo (b*):
  - N = Número de nodos expandidos por A*.
  - d = Profundidad de la solución
  - b* = factor de ramificación de un árbol uniforme de profundidad d, donde N = # nodos que se necesita expandir para llegar a la solución óptima.

$$N = b* + (b*)^2 + \ldots + (b*)^d = b* \frac{(b*)^d - 1}{b* - 1}$$

  - Ejemplo: d=5, N=52 $\Rightarrow$ b*=1.92.
  - Promediar b* para diferentes ejemplos del mismo problema
  - De manera ideal: b* lo más cercano posible a 1.
- Comparación de las heurísticas admisibles h_1, h_2 :

  h_2 domina a h_1 , si $\forall n$: $h_2(n) \geq h_1(n)$
  - Si usamos búsqueda A* y h_2 domina a h_1
    - h_2 nunca expande más nodos que h_1
    - Generalmente, $b_2^* \leq b_1^*$
  - Usar h_2 si h_2 domina a h_1 y los costes de computar las heurísticas son comparables.

Búsqueda de heurísticas

- Si hay disponibles varias heurísticas admisibles (h_1, h_2, ... h_K), la heurística $h_{max}(n) = max\{h_1(n), h_2(n), ... h_K(n)\}$ es admisible y domina a h_1, h_2, ... h_K.
- **Método de relajación**:
  - Definir un **problema relajado** eliminando algunas restricciones del problema original.
  - Usar como heurística para el problema original la **función de coste óptimo** del **problema relajado.**
  - La **heurística** obtenida es **admisible y monótona** (por ser func. de coste óptimo)
  - Ejemplo del 8-puzzle:
    - Problema original: La ficha en la posición A puede moverse a B si A es adyacente a B y B está vacía.
    - Problemas relajados:

      La ficha en la posición A puede moverse a B, aunque B esté ocupado:

      1. Sin restricciones (h_1: # fichas cuya colocación es incorrecta)

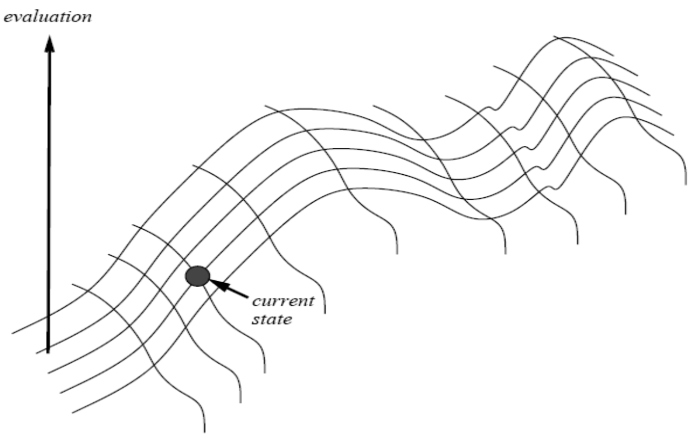
      2. Si A adyacente a B (h_2: distancia de Manhattan o distancia de bloques)

h_2 domina a h_1

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Búsqueda local

- **Problemas de optimización** donde lo importante es sólo el **objetivo** (el camino hasta él es irrelevante).
  - **Maximizar una función objetivo** (o minimizar una función de coste) que es computable con un sólo estado.
  - **Paisaje del espacio de estados**
    - **Máximos globales vs. locales**.
    - Mesetas, crestas.
    - Discontinuidades.



- **<u>Búsqueda local</u>**: Mover desde el estado actual a uno de los vecinos.
  - Un sólo estado actual (requerimientos pequeños de memoria)
  - Puede encontrar una solución razonable en espacios de estados grandes o infinitos (es decir, problemas continuos), donde una búsqueda exhaustiva es imposible.
  - **<u>Completa</u>**: Si garantiza convergencia a un máximo.
  - **<u>Óptima</u>**: Si el máximo encontrado es un máximo global.

Algoritmo de escalada

```
function ESCALADA (problema) devuelve un estado solución
  entrada:        problema, un problema
  variables locales:
                  actual, siguiente: nodos

  actual ← GENERAR-NODO ( ESTADO-INICIAL[problema] )
  loop do
                  siguiente ← nodo sucesor de actual con mayor valor
                  if VALOR[siguiente] < VALOR[actual] then return actual
                  actual ← siguiente
  end
```

- Búsqueda local avariciosa: Moverse al vecino que tenga el valor más alto de la función objetivo.
  - <u>Completa</u>: Converge a un máximo local.
  - <u>No óptima</u>: Puede no converger a un máximo global.
    - Problemas: máximos locales, crestas, mesetas.
- Variaciones
  - <u>Escalada estocástica</u>: Moverse a un vecino elegido aleatoriamente, que tenga un valor de función objetivo mayor que el estado actual.
  - <u>Escalada con reinicio aleatorio</u>: Escalada desde estados iniciales generados aleatoriamente.
    - Conforme el número de reinicios aumenta, el algoritmo se hace completo (generará eventualmente el estado objetivo como uno de los estados iniciales)

Optimización continua

- **Ascenso por gradiente** (búsqueda local avariciosa en un espacio continuo): Moverse en la dirección del gradiente de la función objetivo

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \, \nabla f(\mathbf{x})$$

  - La dirección del gradiente de una función es la dirección de mayor variación local.
  - α es la tasa de ascenso (un valor pequeño)
  - Si no disponemos de la forma analítica del gradiente, podemos usar estimaciones numéricas

$$\frac{\partial}{\partial x_i} f(\mathbf{x}) \approx \frac{f(\mathbf{x}+\mathbf{h}_i) - f(\mathbf{x}-\mathbf{h}_i)}{2h_i} \qquad i = 1,2,\ldots,D$$

$$\mathbf{h}_i = \begin{pmatrix} 0 \\ \vdots \\ h_i \\ \vdots \\ 0 \end{pmatrix} \leftarrow \text{componente número i} \qquad h_i \to 0$$

- **Newton-Raphson**

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{H}^{-1}(\mathbf{x}) \cdot \nabla f(\mathbf{x}); \quad H_{ij} = \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} \quad \text{(Matriz Hessiana)}$$

- **Cuasi-Newton**
- **Gradiente conjugado**
- **<u>Optimización con restricciones</u>**: lineales, cuadráticas, programación no lineal.

Temple simulado

- Maximiza una “función de energía” de acuerdo al siguiente esquema

```
function TEMPLE-SIMULADO (problema, programa) devuelve un estado solución
  entradas:     problema, un problema
                programa: una correspondencia entre iteración y “temperatura”
  variables locales:
                actual, siguiente: nodos
                T, “temperatura” que controla la probabilidad de ir pendiente abajo

actual ← GENERAR-NODO ( ESTADO-INICIAL[problema] )
for t ← 1 to ∞ do
                T ← programa[T]
                if T=0 then return actual
                siguiente ← un sucesor de actual elegido aleatoriamente
                ΔE ← VALOR[siguiente] – VALOR[actual]
                if ΔE > 0 then actual ← siguiente
                else actual ← siguiente sólo con probabilidad exp(ΔE/T)
```

- Seleccionar aleatoriamente un vecino del estado actual
  - Si $\Delta E > 0$ aceptar.
  - Si $\Delta E \leq 0$ aceptar sólo con probabilidad $p = \exp(\Delta E/T)$
- $T = 0 \Rightarrow$ Escalada.
- $T = \infty \Rightarrow$ Búsqueda estocástica.

- Programa geométrico de temple
  - Mantener T constante durante un número de iteraciones (una “época”)
  - Entre época y época, reducir T multiplicándolo por $\beta<1$.

Búsqueda local paralela

- ⌘ Búsqueda local en haz: Búsquedas estocásticas paralelas en k estados
  1. Seleccionar aleatoriamente k estados iniciales.
  2. Generar todos los sucesores de los k estados.
  3. Seleccionar de estos sucesores los mejores k estados.
  4. Repetir paso 2 con el conjunto elegido, hasta que se satisfaga el criterio de convergencia.
- ⌘ Algoritmos genéticos: Usar evolución artificial para maximizar la función de fitness
  - Codificación del problema:
    - Un individuo corresponde a una posible solución del problema.
    - Cada individuo se representa por un **cromosoma**: Cadena de longitud fija que codifica completamente al individuo (por ejemplo, usando una cadena de {0,1}).
  - Ejemplo de algoritmo genético:
    1. Inicializar aleatoriamente una población de M individuos
    2. **Seleccionar** los padres **de acuerdo al fitness**.
    3. Generar una nueva población de M hijos mediante **cruces** entre los padres elegidos.
    4. Introducir variaciones en individuos usando **mutación**.
    5. Repetir desde el paso 2 hasta que se satisfaga el criterio de convergencia.