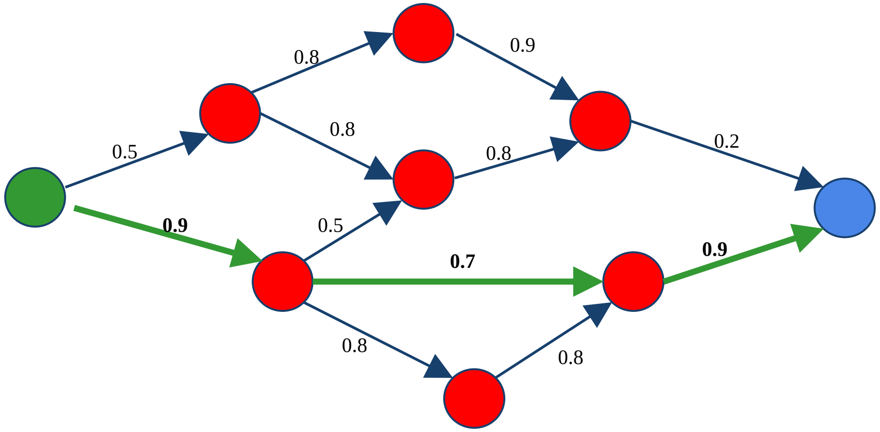
Grado en ingeniería informática
Inteligencia Artificial 2020/2021

2.2. Búsqueda informada



Universidad Autónoma de Madrid

Lecturas :

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

Herramientas:

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

Métodos informados o heurísticos

- Métodos no informados
  - Muy ineficientes en la mayoría de los casos
  - Ante la explosión combinatoria, la fuerza bruta es impracticable
- Métodos informados o heurísticos
  - La búsqueda utiliza conocimiento del dominio para orientar la búsqueda
    - La heurística proporciona información sobre la proximidad de cada estado a un estado objetivo,
    - Utilizando dicha información se puede orientar la búsqueda: Eligiendo como siguiente nodo a expandir el que es más “prometedor”
    - En caso de que la heurística sea fiable reduce la complejidad de la explosión combinatoria de la exploración
      - No genera nodos no prometedores y así mejora rendimiento
      - Puede encontrar una solución aceptablemente buena en tiempo razonable
  - Limitaciones
    - No evita la explosión combinatoria: la complejidad sigue siendo exponencial.
    - Si la heurística no es fiable, empeora la eficiencia.
    - En algunos casos no garantizan encontrar una solución (pueden no ser completos) y, en caso de encontrarla, no garantizan que esta sea óptima.

Heurística

- Del griego $\varepsilon\upsilon\rho\iota\sigma\kappa\omega$ = 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 heurística

- Método de búsqueda heurística:

  Algoritmo que usa una heurística para guiar la búsqueda en el espacio de estados
- Heurística: técnica que mejora la eficiencia de la búsqueda
  - Utiliza conocimiento específico del dominio del problema, más allá de la definición del problema en sí.
  - Proporciona una guía para el algoritmo de búsqueda

    Ordena los nodos generados que están pendientes de expandir (en *lista-abiertos*) de forma que sean expandidos primero aquellos que son más prometedores de acuerdo con información local.

    Esta información local es aquella que se puede obtener directamente a partir del estado actual sin realizar búsqueda (sin generar sucesores).
  - Es posible que la búsqueda heurística no sea ni completa ni óptima
    - Es posible que no se encuentre ninguna solución, habiéndola
    - No garantiza que, en caso de encontrar una solución esta sea la de menor coste.
- Importante seleccionar una función heurística adecuada.
  - Cómo escoger una buena?

Función heurística: *h(n)*

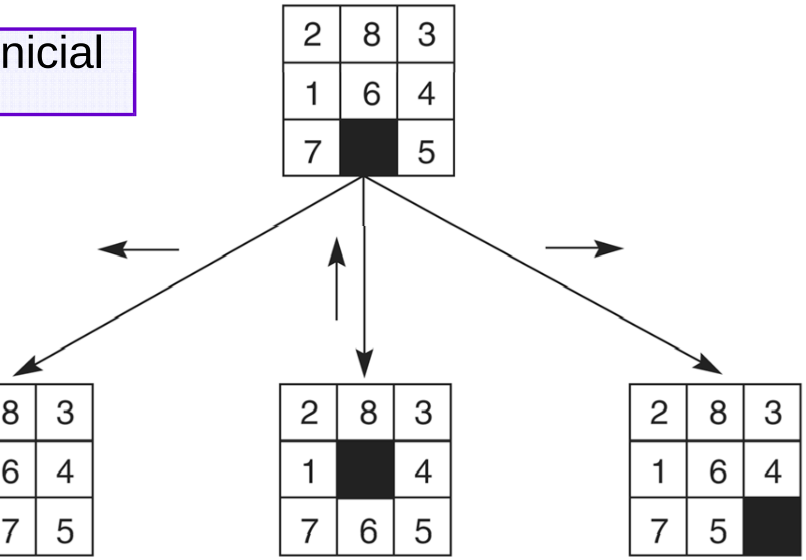
- *h(n)*: El valor de la función heurística *h* evaluada en el nodo *n* es un número que proporciona una estimación de cuán prometedor es el estado correspondiente a ese nodo para alcanzar un estado objetivo.
- Esta información se utiliza en el algoritmo de búsqueda para determinar el orden de expansión de los nodos de *lista-abiertos.*
- Dos posibles interpretaciones
  1. Estimación de la "calidad" de un estado
     - Los estados cuyo valor para la heurística sea mayor son preferidos
  2. Estimación del coste para alcanzar el estado objetivo de menor coste desde de un estado dado.
     - Los estados cuyo valor para la heurística sea menor son preferidos

  Ambos puntos de vista son complementarios: Un cambio de signo permite pasar de una perspectiva a la otra

- Convenio: 2ª interpretación
  - **Valores de la heurística no negativos (menor es mejor)**
  - **A los estados que cumplen el *test-objetivo* se les asigna un valor 0 para la heurística**
  - **Atención:** puede haber estados que no cumplen el *test-objetivo* y tienen un valor 0 para la heurística.

Ejemplo: heurísticas para el 8-puzzle

Estado inicial



Estado objetivo

1	2	3
8		4
7	6	5

Ejemplo: heurísticas para el 8-puzzle

- ***h_a*** = suma las distancias de las fichas a sus posiciones en el tablero objetivo
  - Como no hay movimientos en diagonal, se suman las distancias horizontales y verticales
  - Llamada distancia de Manhattan, distancia taxi o distancia en la ciudad
    - Ejemplo: 1+1+0+0+0+1+1+2 = 6

2	8	3
1	6	4
	7	5

1	2	3
8		4
7	6	5

- ***h_b*** = nº de fichas descolocadas (respecto al tablero objetivo)
  - Es la heurística más sencilla y parece bastante intuitiva
    - Ejemplo: 5
  - Pero no usa la información relativa al esfuerzo (nº de movimientos) necesario para llevar una ficha a su sitio

Ejemplo: heurísticas para el 8-puzzle

- Ninguna de estas dos heurísticas le da importancia a la inversión de fichas
  - Si 2 fichas están dispuestas de forma contigua, han de intercambiar sus posiciones para estar ordenas como en un estado objetivo, por ello, se necesitan más de 2 movimientos para alcanzar dicha ordenación

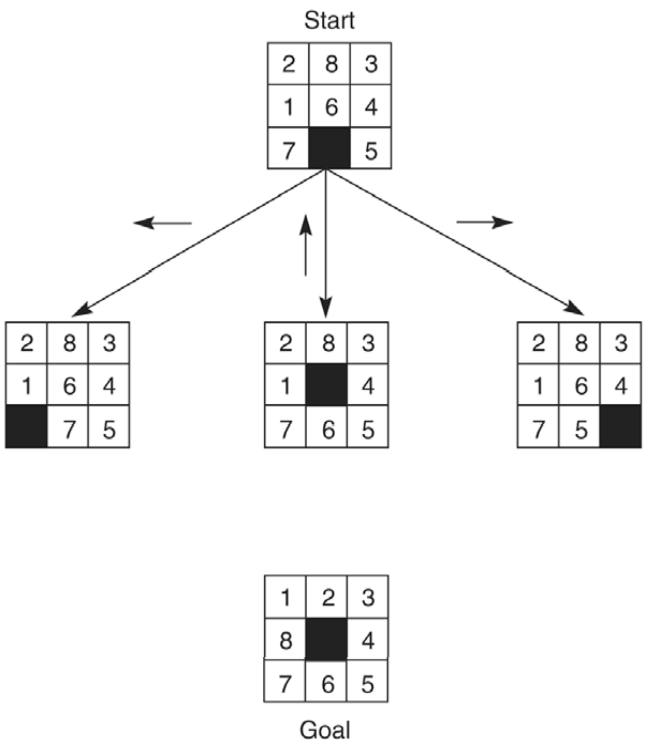
- h_c = el doble del nº de pares de fichas cuyas posiciones es necesario intercambiar para alcanzar la meta
  - Esta heurística también es pobre, puesto que se concentra en una cierta característica
    - En particular, tienen valor 0 muchos tableros que no son objetivo
    - Puede servir en combinación con otras medidas de distancia a la meta para diseñar una función heurística final

- Ejemplo: 2*1= 2

1	2	3
8		4
7	5	6

Ejemplo: heurísticas para el 8-puzzle

- $\boldsymbol{h_d}(n) = h_a(n) + h_c(n)$
  - Es la heurística más fina, pero requiere un mayor esfuerzo de cálculo
  - Aún podría mejorarse...



	$\boldsymbol{h_a}$	$\boldsymbol{h_b}$	$\boldsymbol{h_c}$	$\boldsymbol{h_d}$
2 8 3 1 6 4 ■ 7 5	6	5	0	6
2 8 3 1 ■ 4 7 6 5	4	3	0	4
2 8 3 1 6 4 7 5 ■	6	5	0	6

Búsqueda informada

- 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-abiertos* que expandiremos de acuerdo a una **función de evaluación** [f(*n*)] que proporciona el coste del camino óptimo (de menor coste) 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-abiertos*).

    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.

Búsquedas primero el mejor

- Primero el aparentemente mejor:
  - El nodo más prometedor de acuerdo con la información disponible localmente (sin realizar búsqueda; es decir, sin generar sucesores)
  - Si supiéramos cuál es el mejor nodo para expandir en cada paso, esto **no** sería una búsqueda sino una marcha directa al objetivo.

- Selección del siguiente nodo a expandir de acuerdo con una función de evaluación *f(n)* que tenga en cuenta
  - *g(n)* = coste del camino desde nodo inicial hasta *n*
  - *h(n)* = estimación del coste necesario para llegar a la solución de menor coste a partir del nodo *n*
- El nodo seleccionado para la expansión será aquel de entre los que se encuentran en *lista-abiertos* que minimiza la función de evaluación
  - *lista-abiertos* es una cola de prioridad: lista ordenada de menor a mayor valor para la función de evaluación.
  - De esta forma, basta con elegir el primer nodo de la lista para la expansión.

- Asumiremos que el **coste de los operadores es no negativo** (≥ 0).

Búsquedas primero el mejor

La función de evaluación *f* puede ser de dos tipos:

1. ***f (n) = h(n)*** **[Búsqueda voraz]**

   Se ignora el coste de camino hasta ese momento y considera exclusivamente lo que "falta": el coste mínimo estimado para llegar a una solución a partir del nodo *n.*

2. ***f (n) = g(n) + h(n)*** **[Búsqueda A*]**

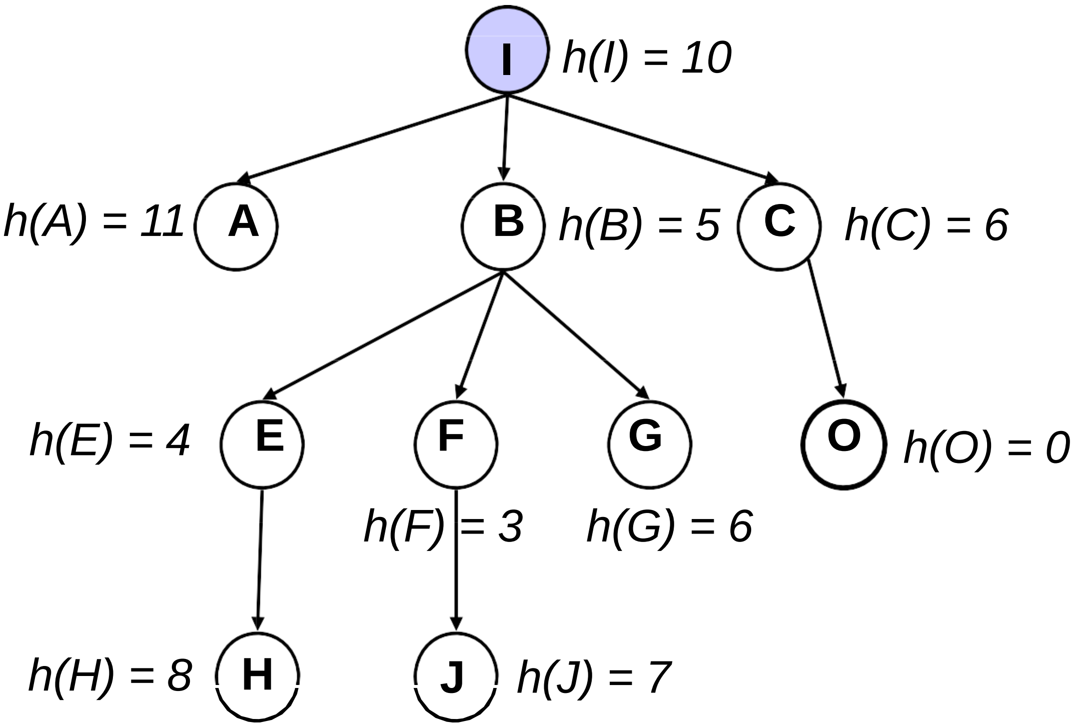
   - La función ***f*** está formada por dos componentes:
     - *g(n)* = coste del camino desde nodo inicial hasta *n*
       - No es una estimación, sino un coste **real** calculado exactamente
     - *h(n)* = coste mínimo estimado para llegar a nodo objetivo desde *n*
   - *h(n)* = estimación del coste total del camino óptimo desde el nodo inicial a un nodo objetivo y que pase el nodo *n.*

1. Búsqueda voraz *(greedy)*

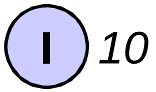
- ***f (n) = h(n)***
  - Representa el coste mínimo estimado para llegar desde *n* a un nodo objetivo (por el camino de menor coste)
    - *h(n) = 0* para los nodos objetivo
- Es un algoritmo voraz
- Propiedades
  - No es óptima ni completa
    - No tiene en cuenta el coste real desde el nodo inicial al actual, solo una estimación del coste del camino óptimo desde el nodo actual hasta el objetivo.
  - Caso peor: si la heurística es deficiente, puede tener mayor coste computacional que la búsqueda no informada *(no realista)*
    - Tiempo: $O(b^m)$, siendo *m* la profundidad máxima del espacio de búsqueda
    - Espacio: $O(b^m)$ (mantiene todos los nodos que genera si GraphSearch)
  - Si la heurística es buena, los costes computacionales podrían reducirse
    - Esto depende del problema y de la calidad de la heurística
    - Si la heurística es buena, expande pocos nodos: bajo coste computacional.

Ejemplo: búsqueda voraz

Espacio de estados



Árbol de búsqueda

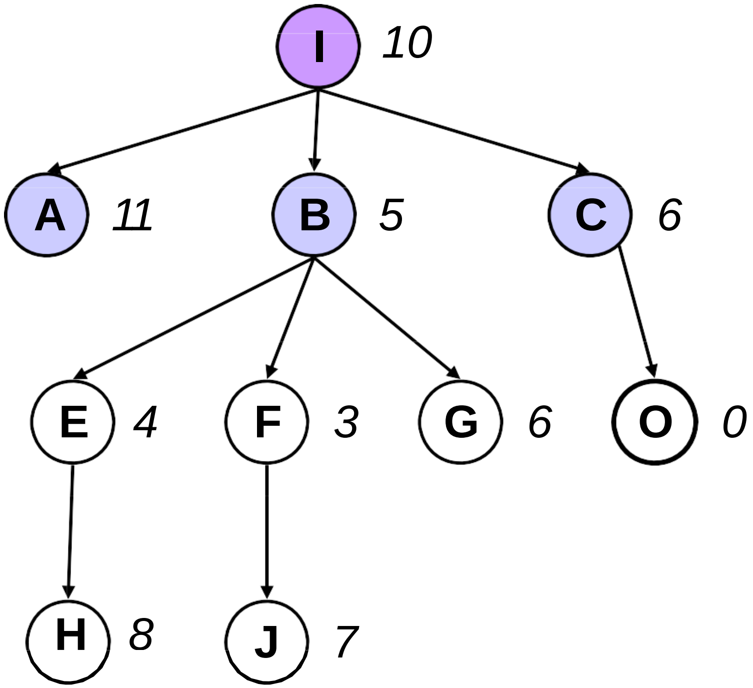


lista-abiertos (cola de prioridad): (I_{10})

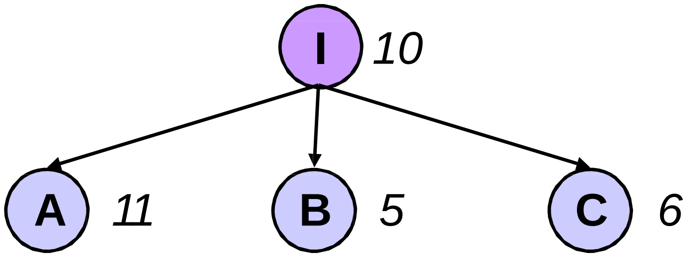
lista-cerrados: ( )

Ejemplo: búsqueda voraz

Espacio de estados



Árbol de búsqueda

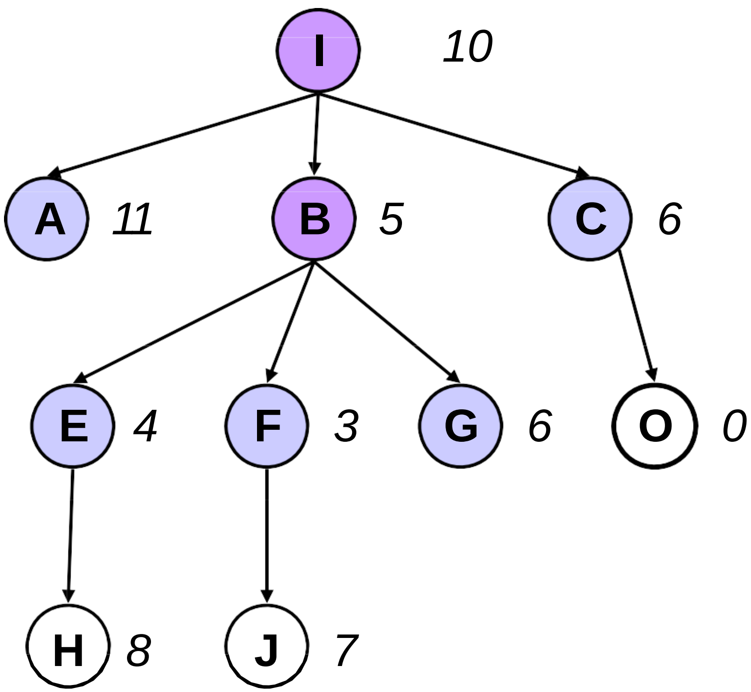


lista-abiertos (cola de prioridad): (B_5 C_6 A_{11})

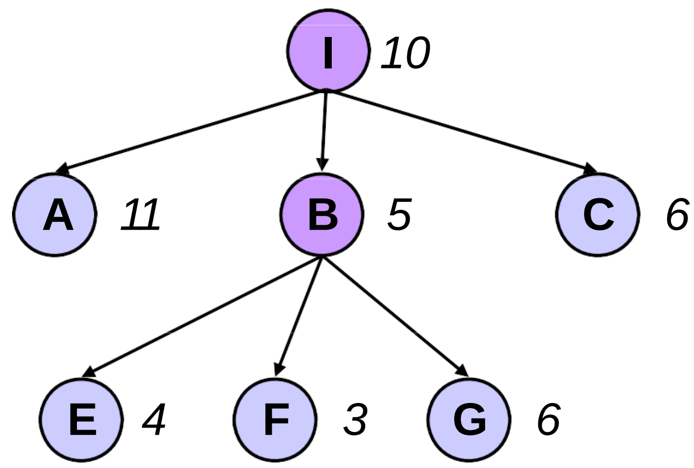
lista-cerrados: (I)

Ejemplo: búsqueda voraz

Espacio de estados



Árbol de búsqueda

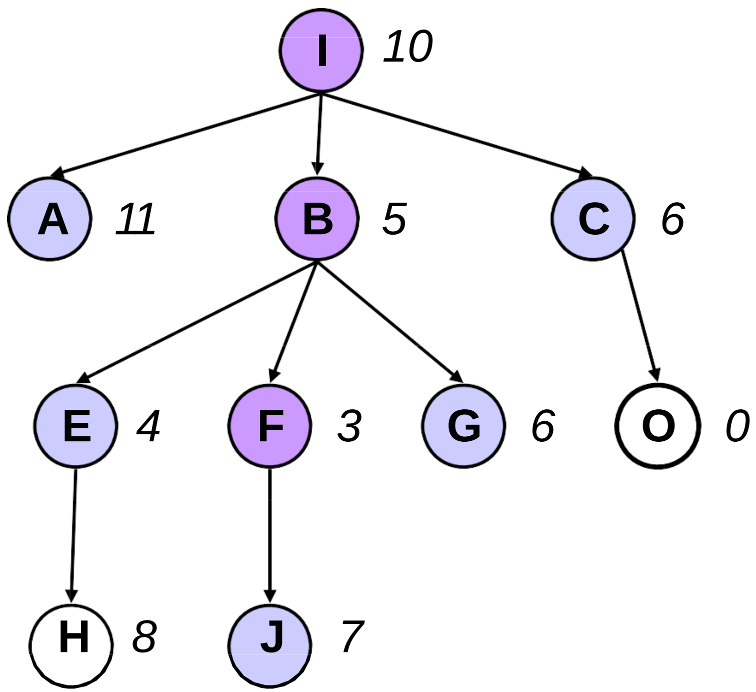


lista-abiertos (cola de prioridad): (F_3 E_4 C_6 G_6 A_{11})

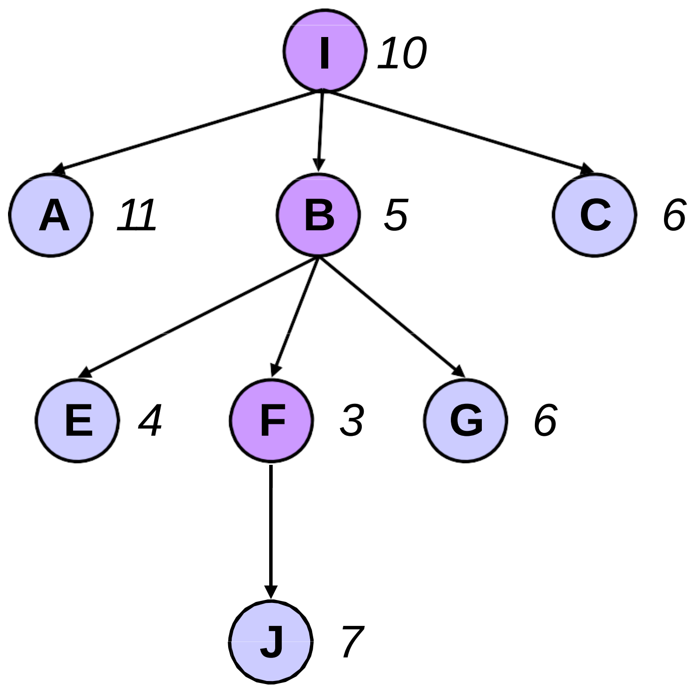
lista-cerrados: (B I)

Ejemplo: búsqueda voraz

Espacio de estados



Árbol de búsqueda

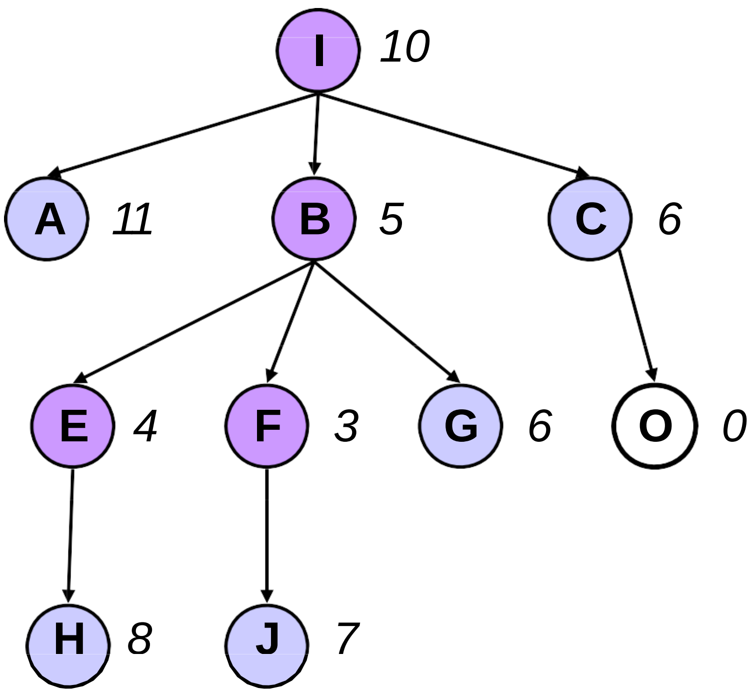


lista-abiertos (cola de prioridad): (E_4 C_6 G_6 J_7 A_{11})

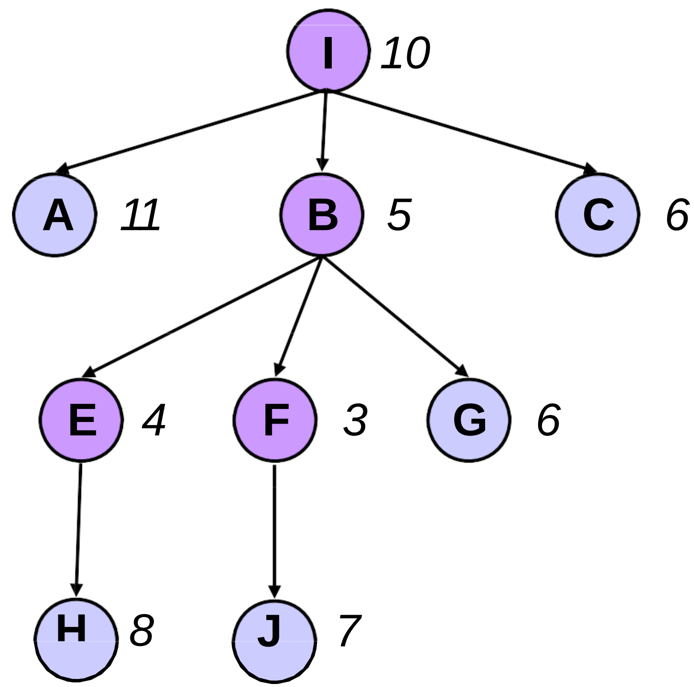
lista-cerrados: (F B I)

Ejemplo: búsqueda voraz

Espacio de estados



Árbol de búsqueda

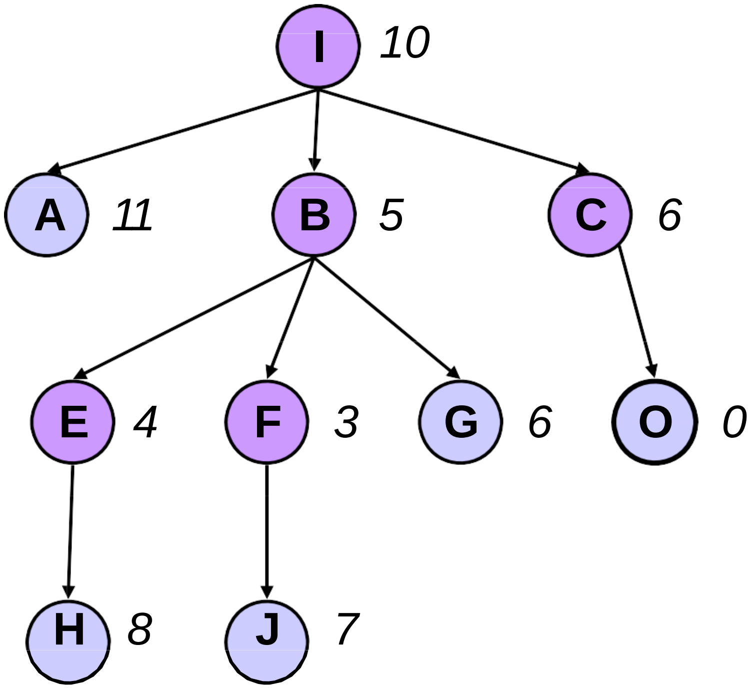


lista-abiertos (cola de prioridad): (C_6 G_6 J_7 H_8 A_{11})

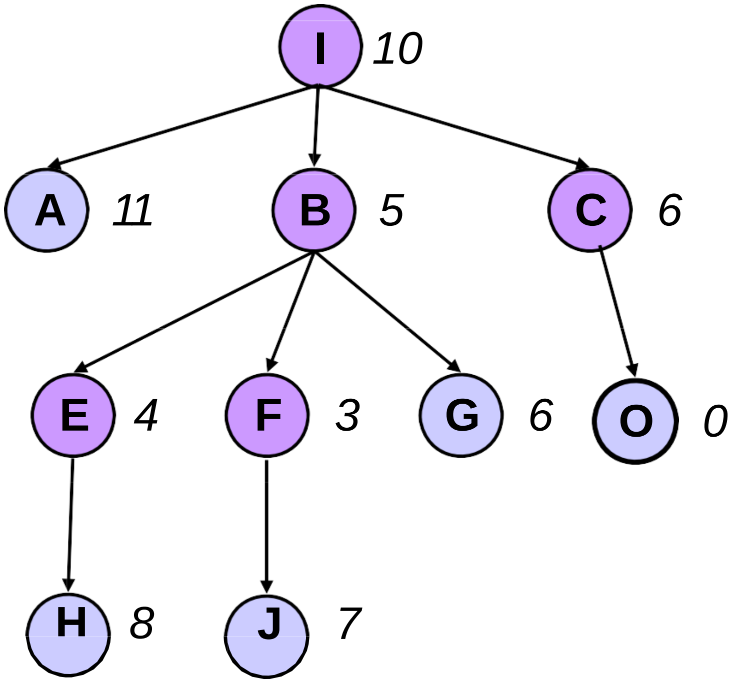
lista-cerrados: (E F B I)

Ejemplo: búsqueda voraz

Espacio de estados



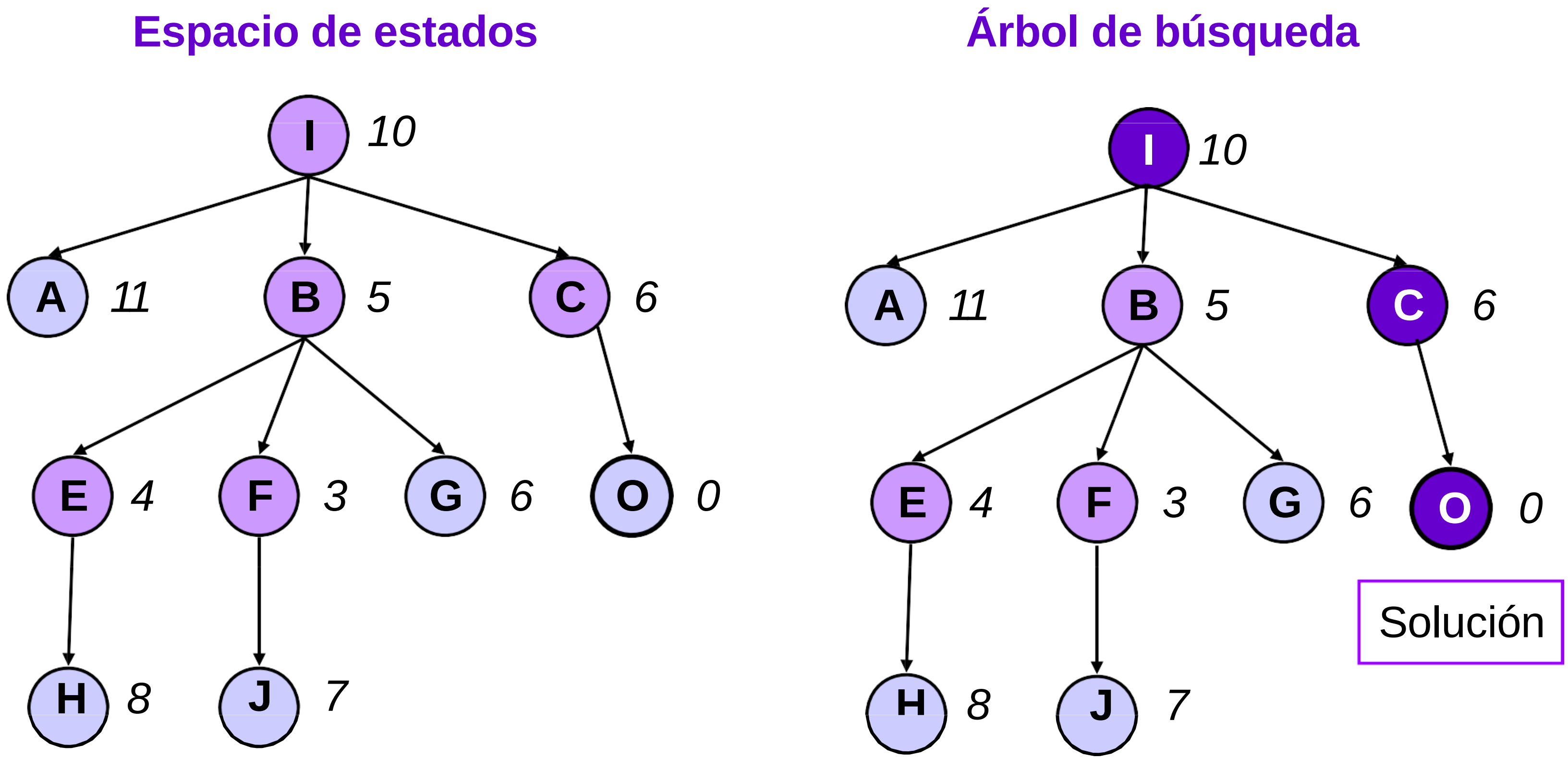
Árbol de búsqueda



lista-abiertos (cola de prioridad): (O_0 G_6 J_7 H_8 A_{11})

lista-cerrados: (C E F B I)

Ejemplo: búsqueda voraz

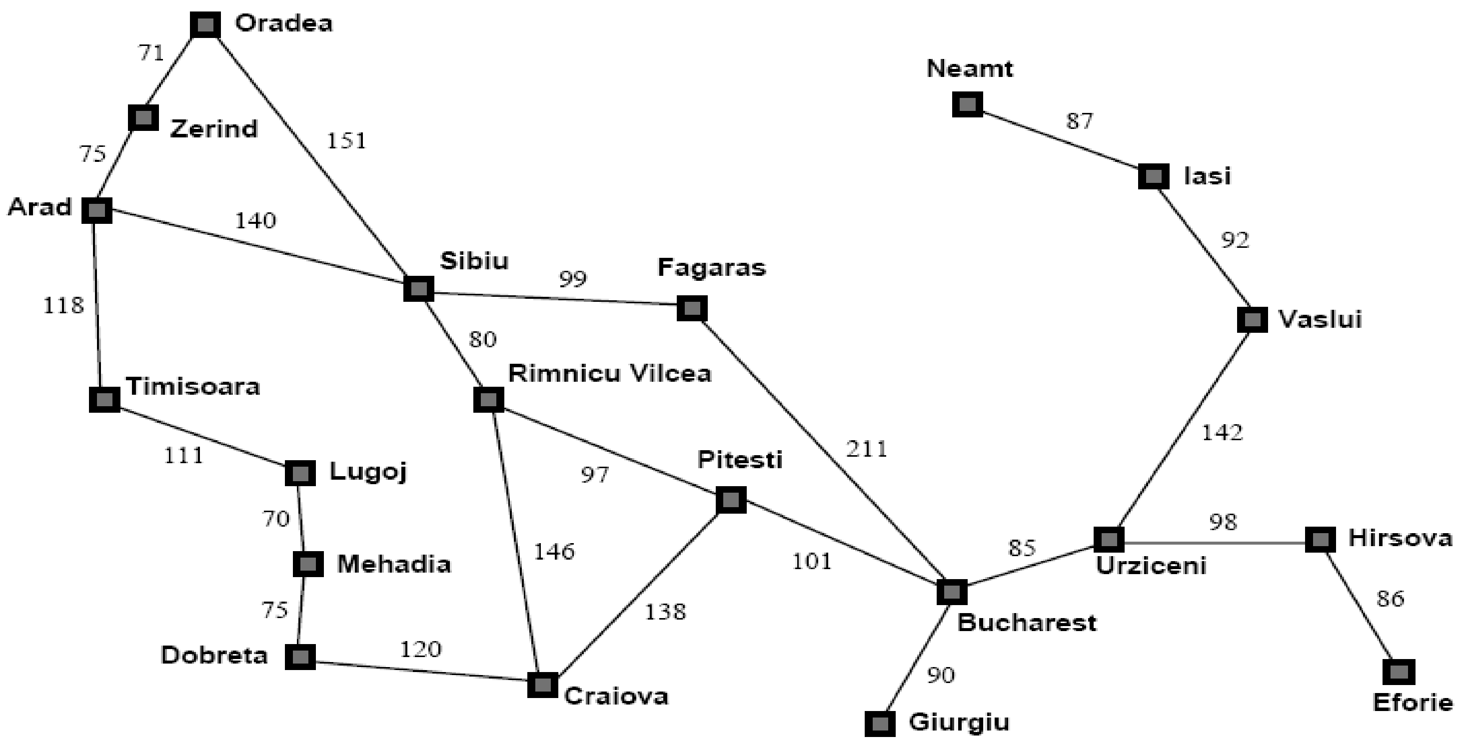


lista-abiertos (cola de prioridad): (G_6 J_7 H_8 A_{11})

lista-cerrados: (C E F B I)

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

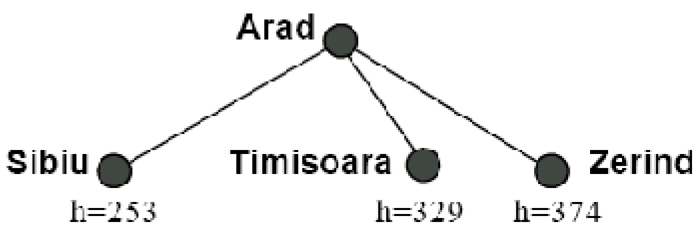
Función heurística (admisible, monótona)

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

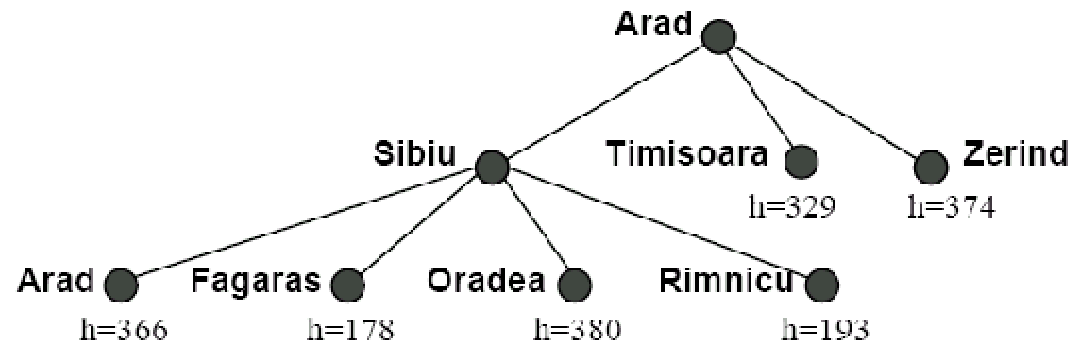
Búsqueda voraz para el problema del mapa de carreteras

Estado inicial:
Ciudad de salida

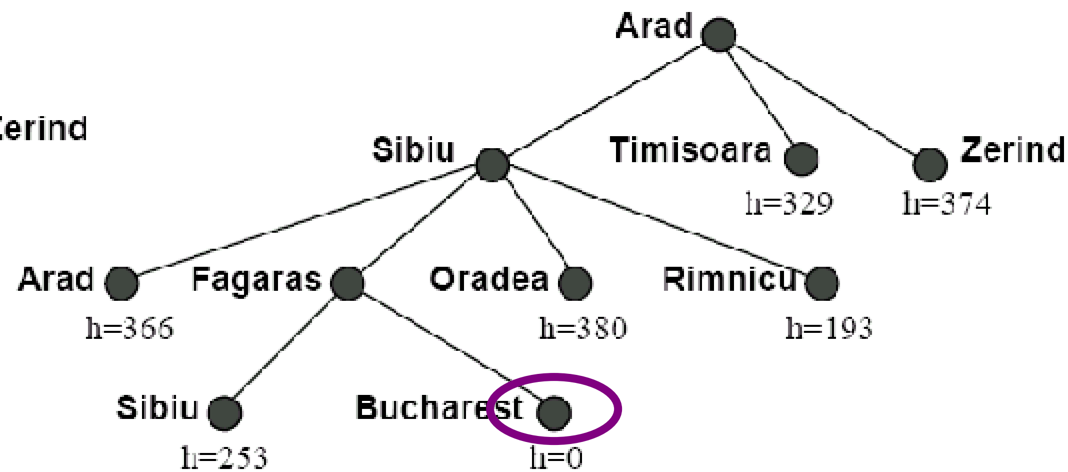
1. Expande Arad



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



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

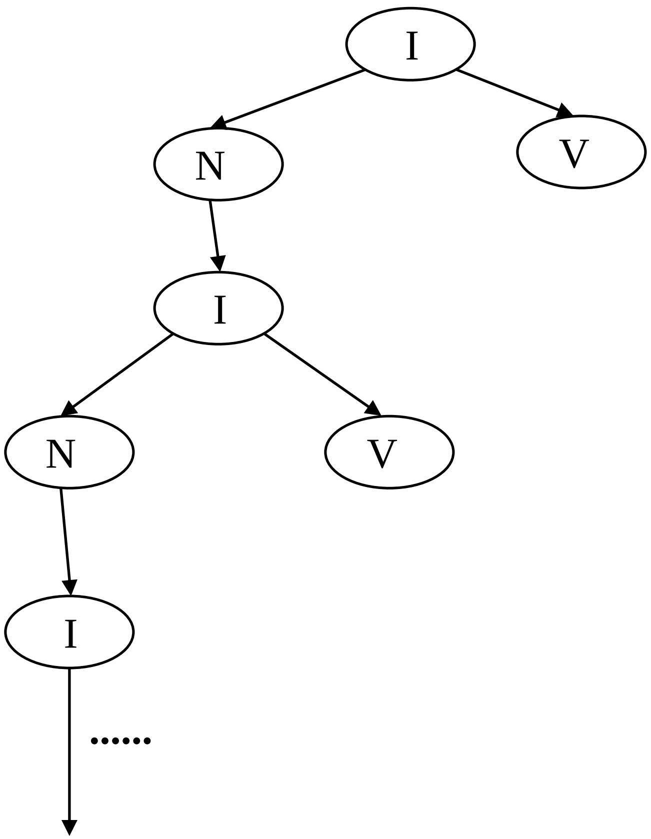


4. Objetivo alcanzado: Bucarest

Búsqueda voraz: 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

2. Búsqueda algoritmo A*

- ***f (n) = g(n) + h(n)***
  - *f (n)* = estimación del coste mínimo total (desde el inicial hasta un objetivo) de cualquier solución que pase por el nodo *n*
    - *g(n)* = coste real del camino hasta *n*
  - *h(n)* = estimación del coste mínimo desde *n* a un nodo objetivo
    - Si *h* = 0 => búsqueda de coste uniforme *("no informada": 1º menor coste)*
    - Si *g* = 0 => búsqueda voraz

- Combina el tipo primero en anchura con el tipo primero en profundidad
  - La componente *g* de *f* le da el toque realista,
  - Impidiendo que se guíe exclusivamente por una *h* demasiado optimista
    - *h* tiende a primero en profundidad
    - *g* tiende a primero en anchura: fuerza la vuelta atrás cuando domina a *h*
    - Se cambia de camino cada vez que haya otros más prometedores

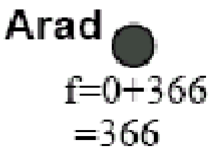
Condiciones sobre h

- *h* es una heurística admisible si $h(n) \leq g^*(n)$ para todo *n*
  - $g^*(n)$ es el coste real para ir desde el nodo *n* a un nodo objetivo por el camino de menor coste (coste óptimo).
  - h no sobreestima para ningún nodo el coste óptimo para alcanzar el objetivo.
  - Es, por lo tanto, una heurística optimista.

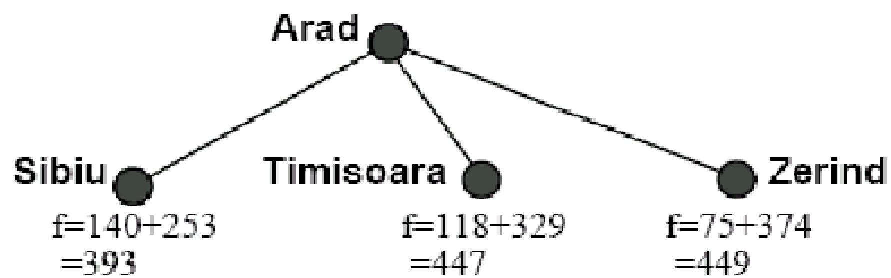
- **La búsqueda A* sin eliminación de estados repetidos y con una heurística admisible** es **óptima** (es decir, garantiza encontrar la solución der menor coste)

Búsqueda A*: Mapa de carreteras, I

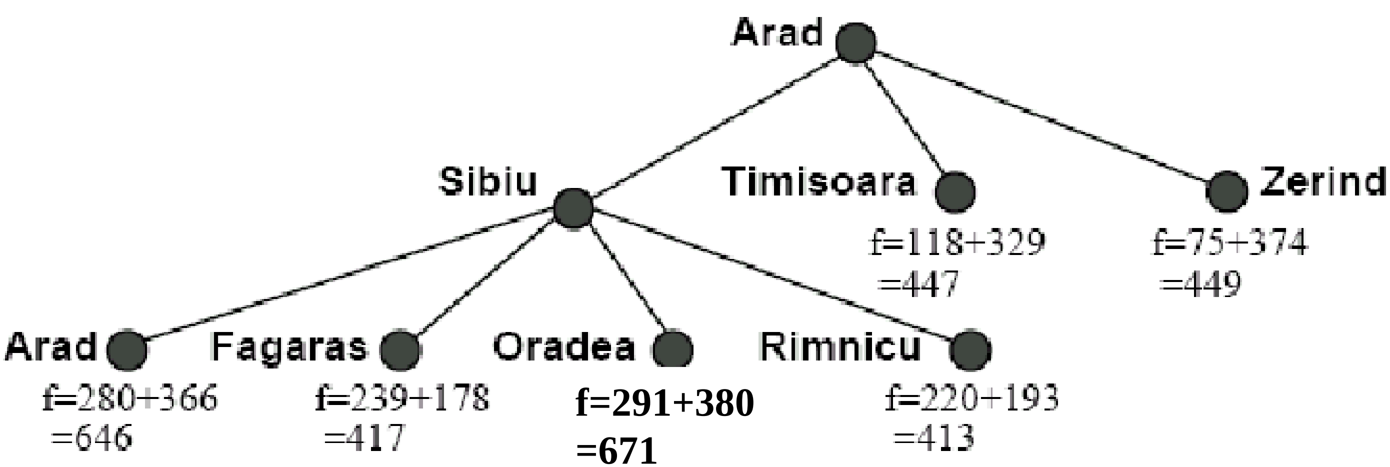
•



1. Expande Arad

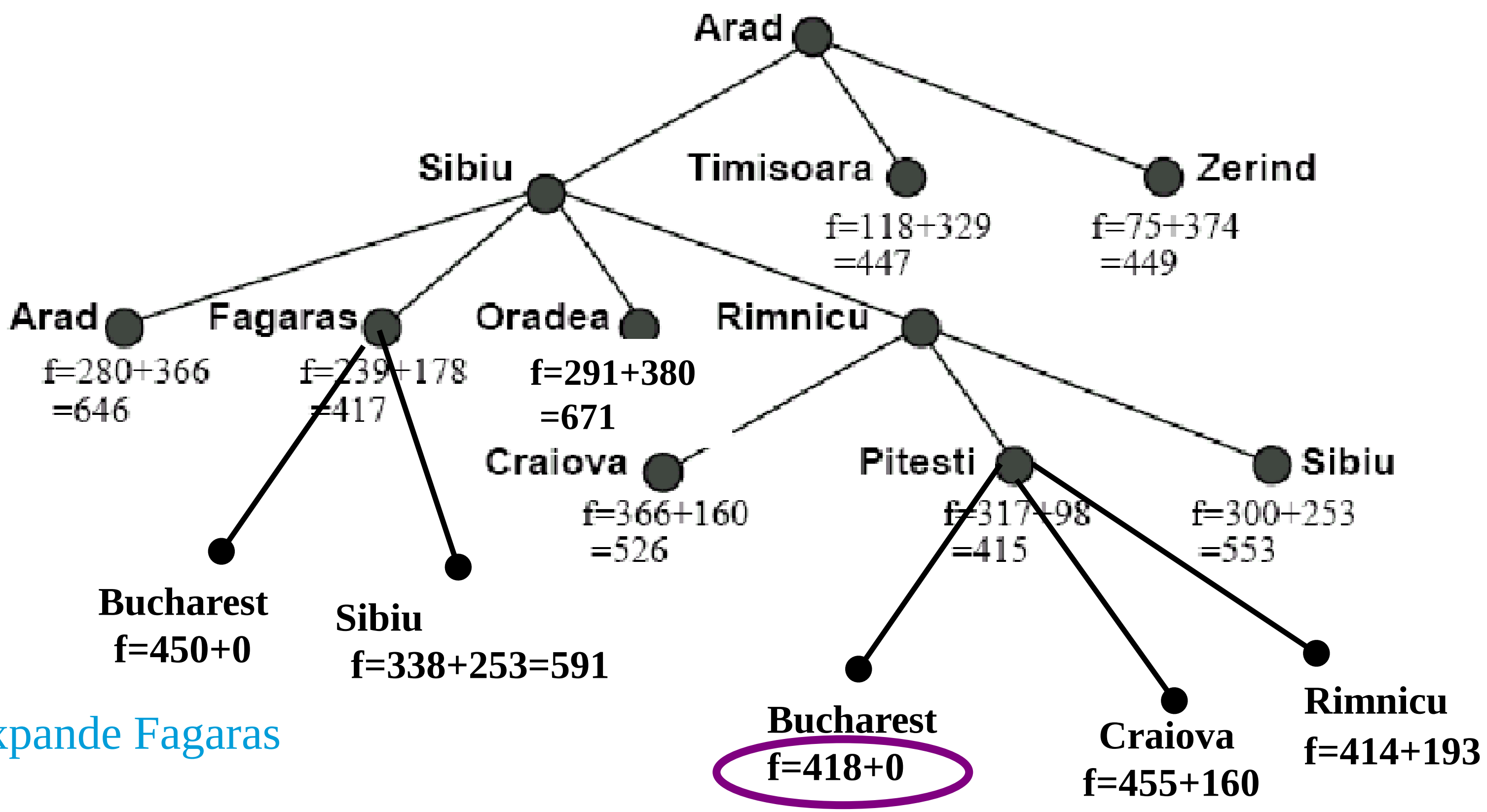


2. Expande Sibiu



Búsqueda A*: Mapa de carreteras, II

3. Expande Rimnicu (f más pequeño, 413)



5. Expande Fagaras

4. Expande Pitesti

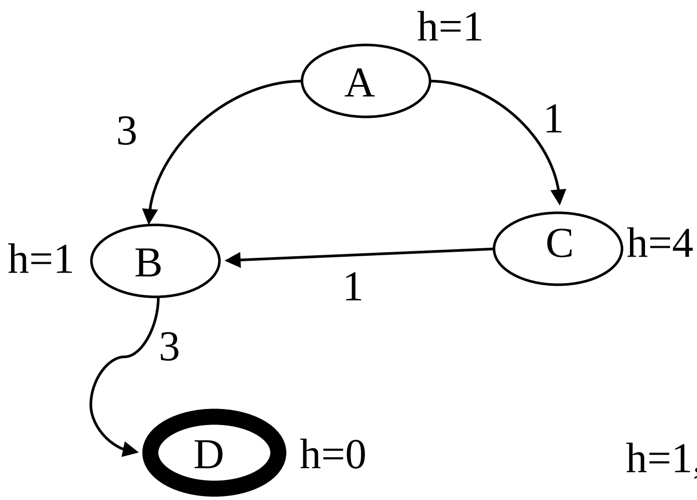
6. Objetivo encontrado: Bucarest

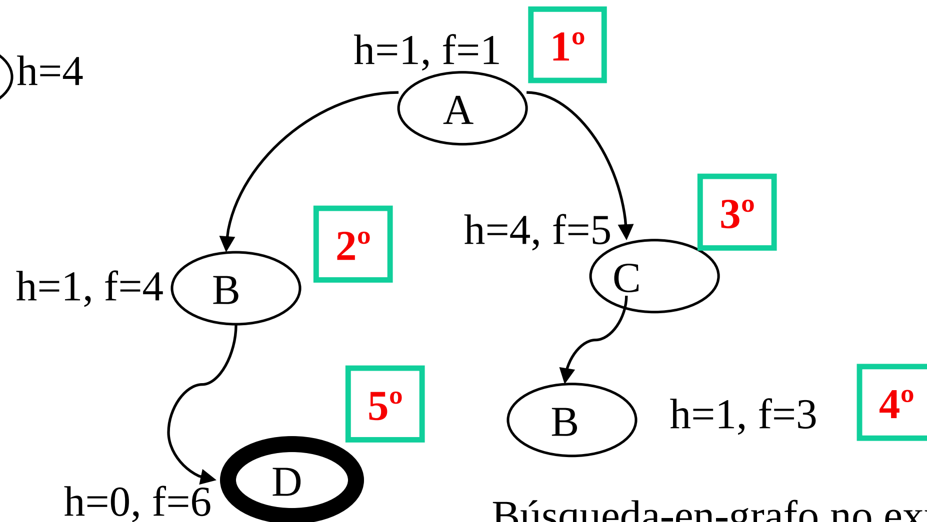
Implementación de A*

- A* puede ser implementado mediante
  - *búsqueda-en-árbol* (sin eliminación de estados repetidos)
  - *búsqueda-en-grafo* (con eliminación de estados repetidos)
- En la implementación con *búsqueda-en-grafo* necesitamos:
  - Información almacenada para el nodo ***n***:
    - Descripción del estado correspondiente a dicho nodo.
    - Referencia al nodo padre: para reconstruir el camino al encontrar la solución
    - Valores de $f(n) = g(n) + h(n)$
  - Dos estructuras para almacenar los nodos:
    - *lista-abiertos*: (copa del árbol de búsqueda): cola prioridad en la que los nodos generados, pero aún no expandidos están ordenados de menor a mayor valor de la función de evaluación (f)
    - *lista-cerrados*: Nodos ya expandidos.
      - No se eliminan al ser generados, sino al intentar expandirlos.
  - **PROBLEMA con la eliminación de estados repetidos**:
    Aunque la heurística sea admisible, puede que la solución óptima se obtenga explorando a partir del nodo que se elimina por haber sido ya expandido.

A* + heurística admisible

- Si se usa **búsqueda-en-grafo (con eliminación de estados repetidos)**, **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.





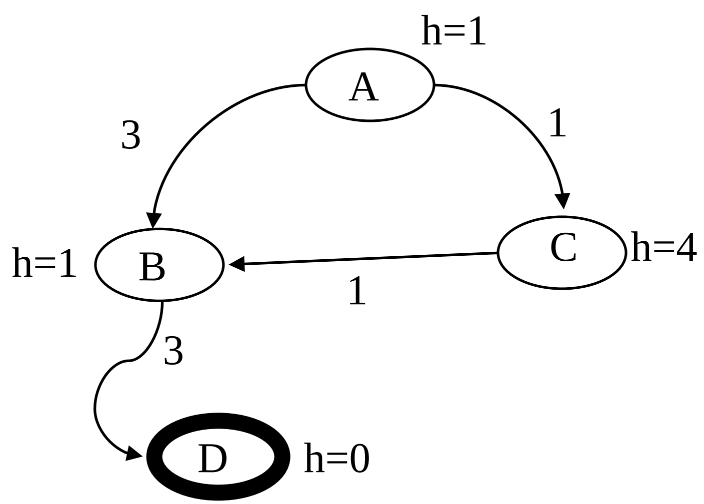
La solución encontrada es subóptima: coste = 6

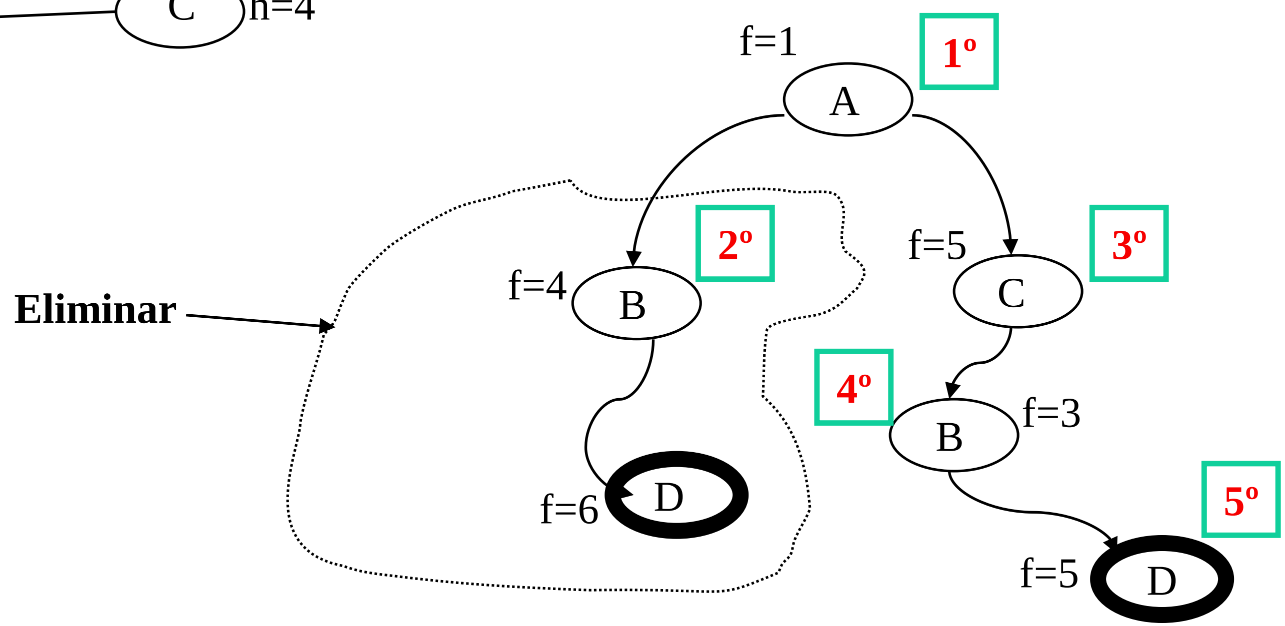
Búsqueda-en-grafo no expande B (B está en lista-cerrada)

A* + heurística admisible

Solución: Descartar el **camino** con coste más alto.

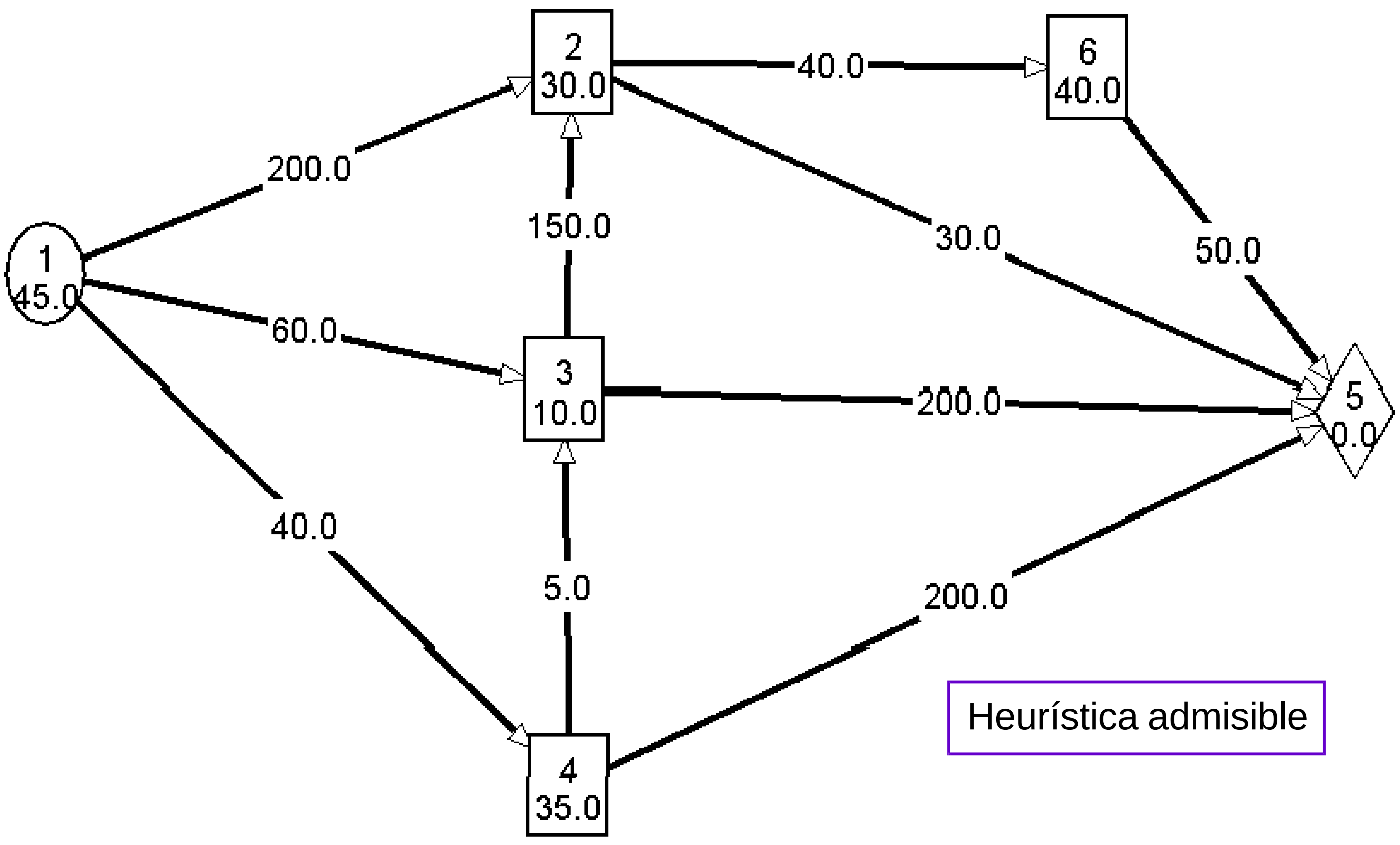
- Aumenta la complejidad del algoritmo: necesita eliminar de *lista-abierta* el nodo con coste más alto y sus descendientes.





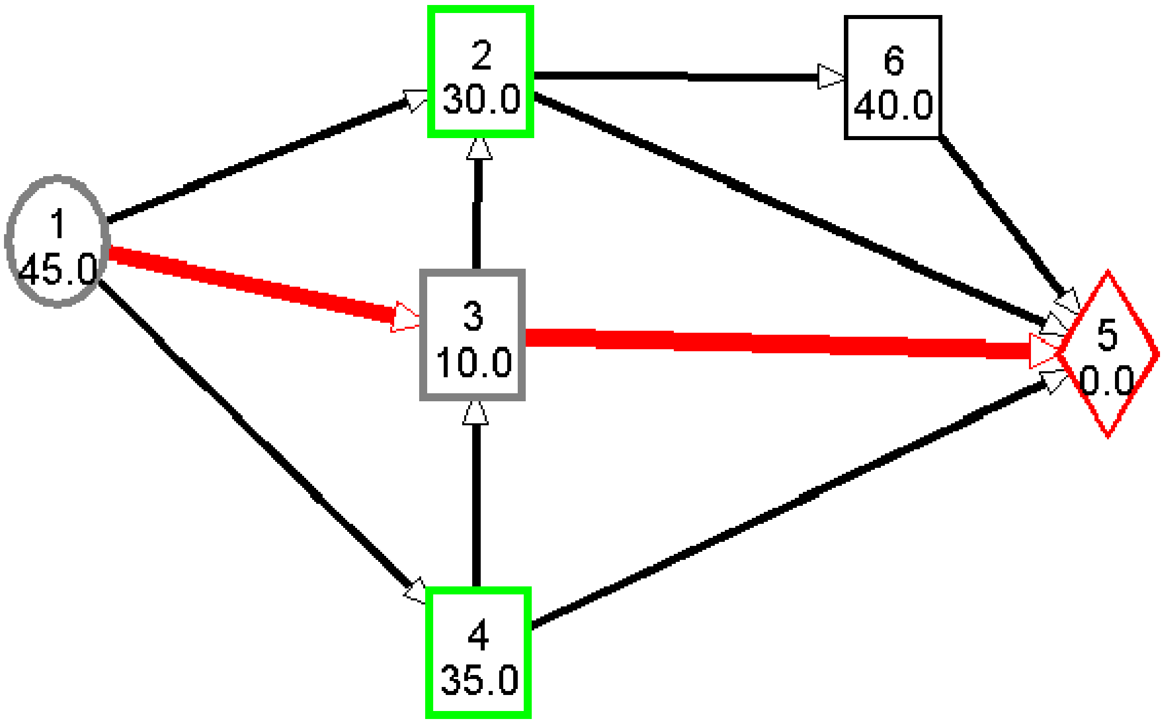
La solución encontrada es óptima: coste = 5

Ejemplo 2



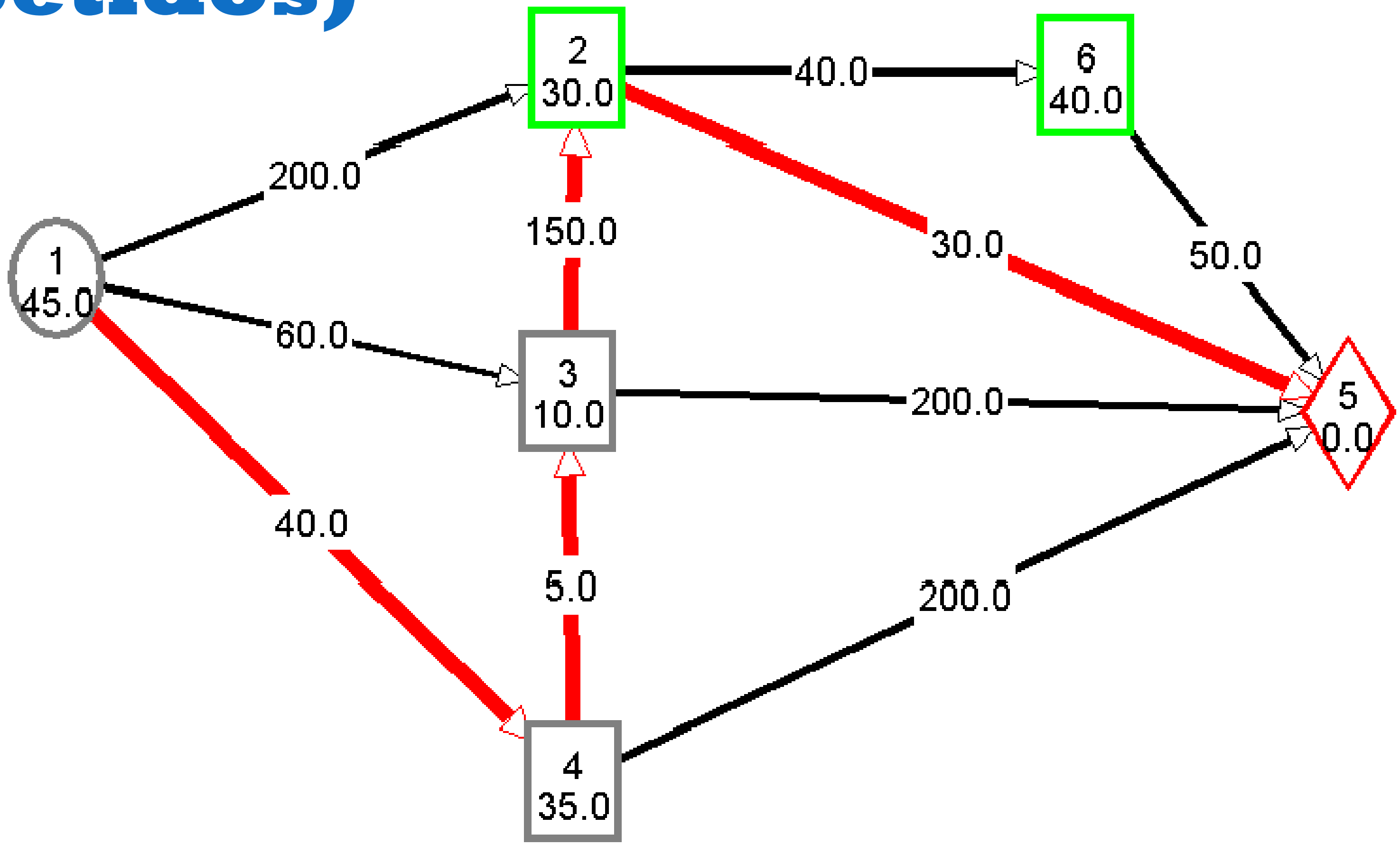
Heurística admisible

Solución con búsqueda voraz



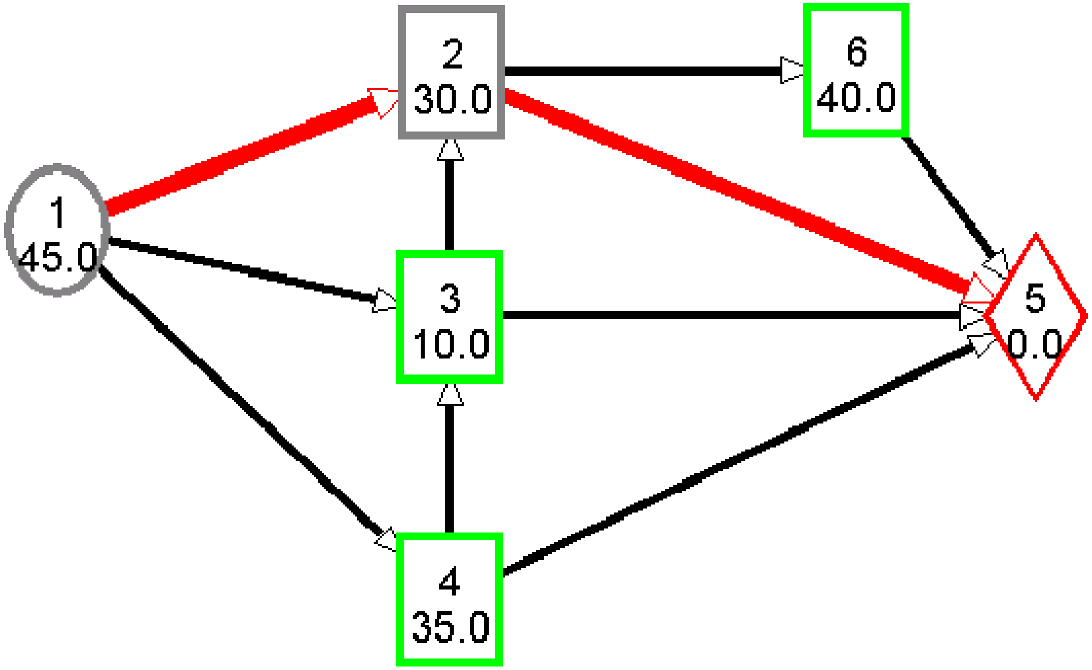
Búsqueda voraz: 1-3-5 Coste (no tenido en cuenta): 60+200 = 260

Solución: A* + búsqueda-en-árbol (sin eliminación de estados repetidos)



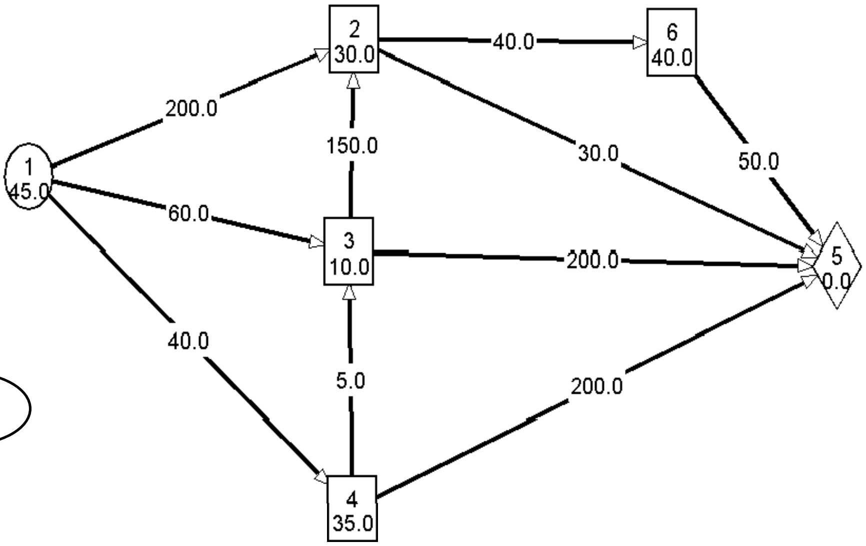
Solución con A* + búsqueda en árbol : 1-4-3-2-5
Coste: 40+5+150+30 = 225 (óptimo)

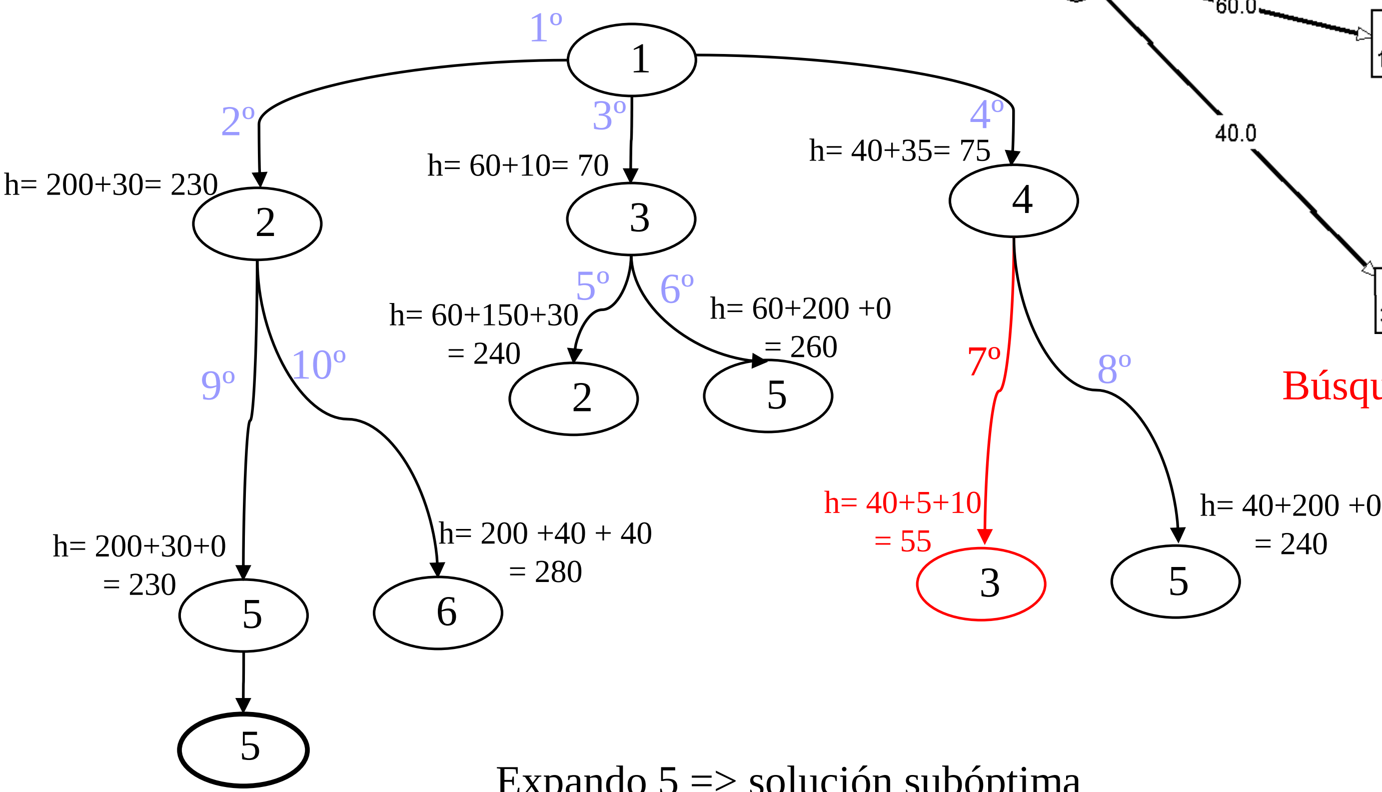
Solución: A* + búsqueda-en-grafo (con eliminación de estados repetidos)



Solución con A* + búsqueda en grafo: 1-4-3-2-5
Coste: 40+5+150+30 = 225 (subóptimo)

En detalle

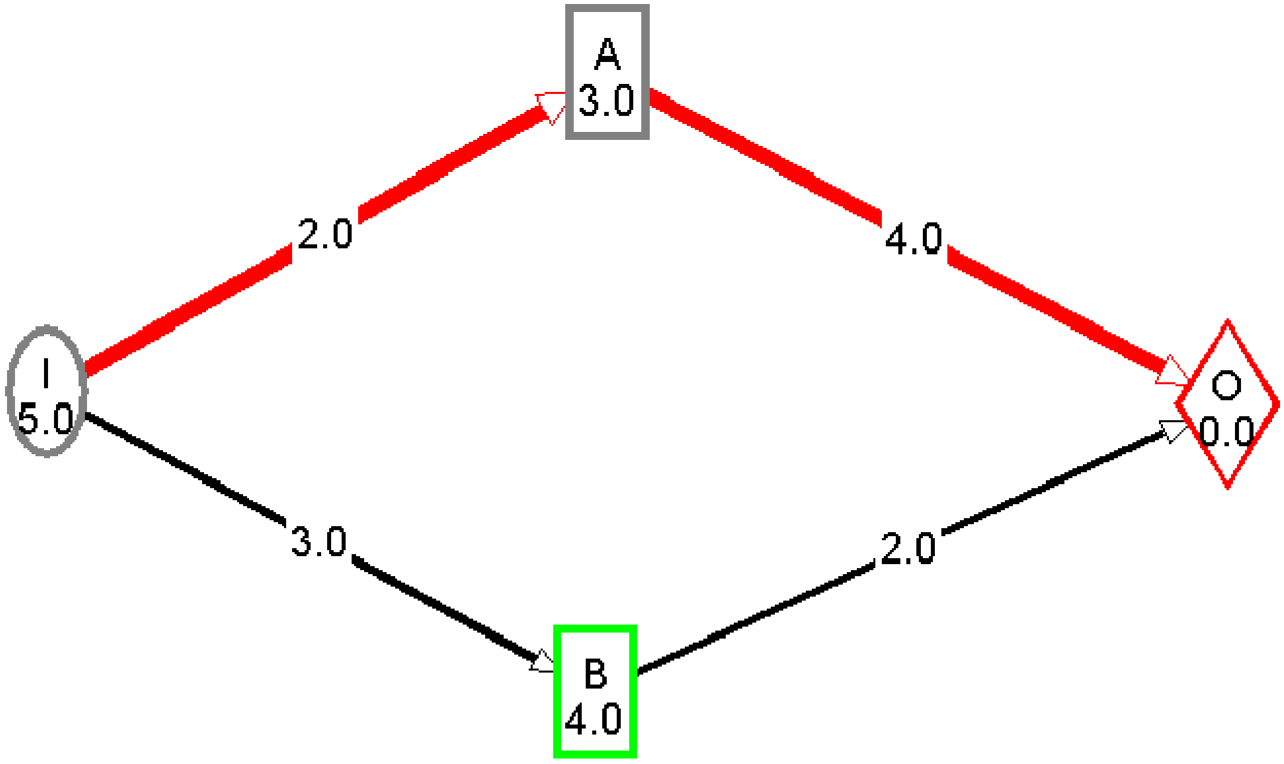




Búsqueda-en-grafo no expande 3 de nuevo
(está en lista-cerrados)

Expando 5 => solución subóptima
coste 230. 1-2-5

A* + eliminación de estados repetidos + heurística no admisible no garantiza encontrar la solución de menor coste



Heurística no admisible:
`no garantiza encontrar la solución de menor coste

Propiedades de A*

TEOREMA:

Si se usa **búsqueda-en-árbol (sin eliminación de estados repetidos**, y **h es admisible** $h(n) \leq h^*(n), \forall n$ $\Rightarrow$ **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

Heurística monótona (o “consistente”)

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

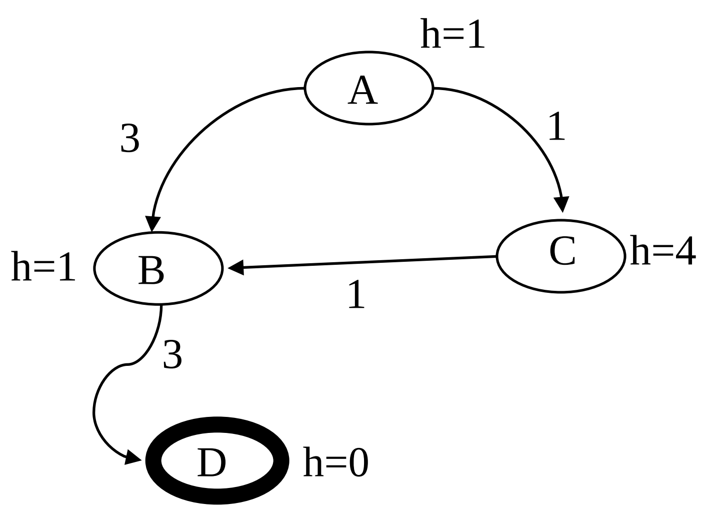
$$h(n) \leq \text{ coste}(\text{n} \rightarrow \text{n}') + h(n'),$$
$$\forall n, n' [n' \text{ sucesor de n}]$$

  - 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

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

- TEOREMA:

  Si **h es monótona** $\Rightarrow$ **A*** usando **búsqueda-en-grafo (con eliminación de estados repetidos) 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 $g(n)=g^*(n)$
  - Siendo $g^*(n)$ el coste del camino óptimo entre el nodo inicial y 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.

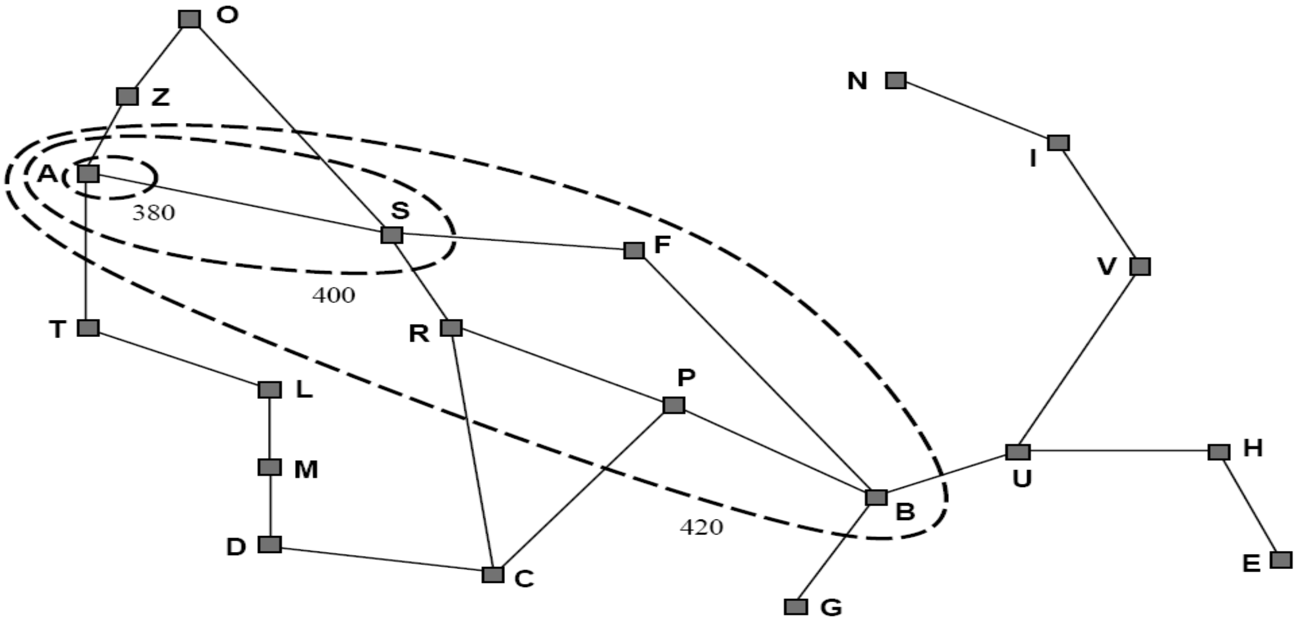
[Dechter, Pearl, 1985]

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.

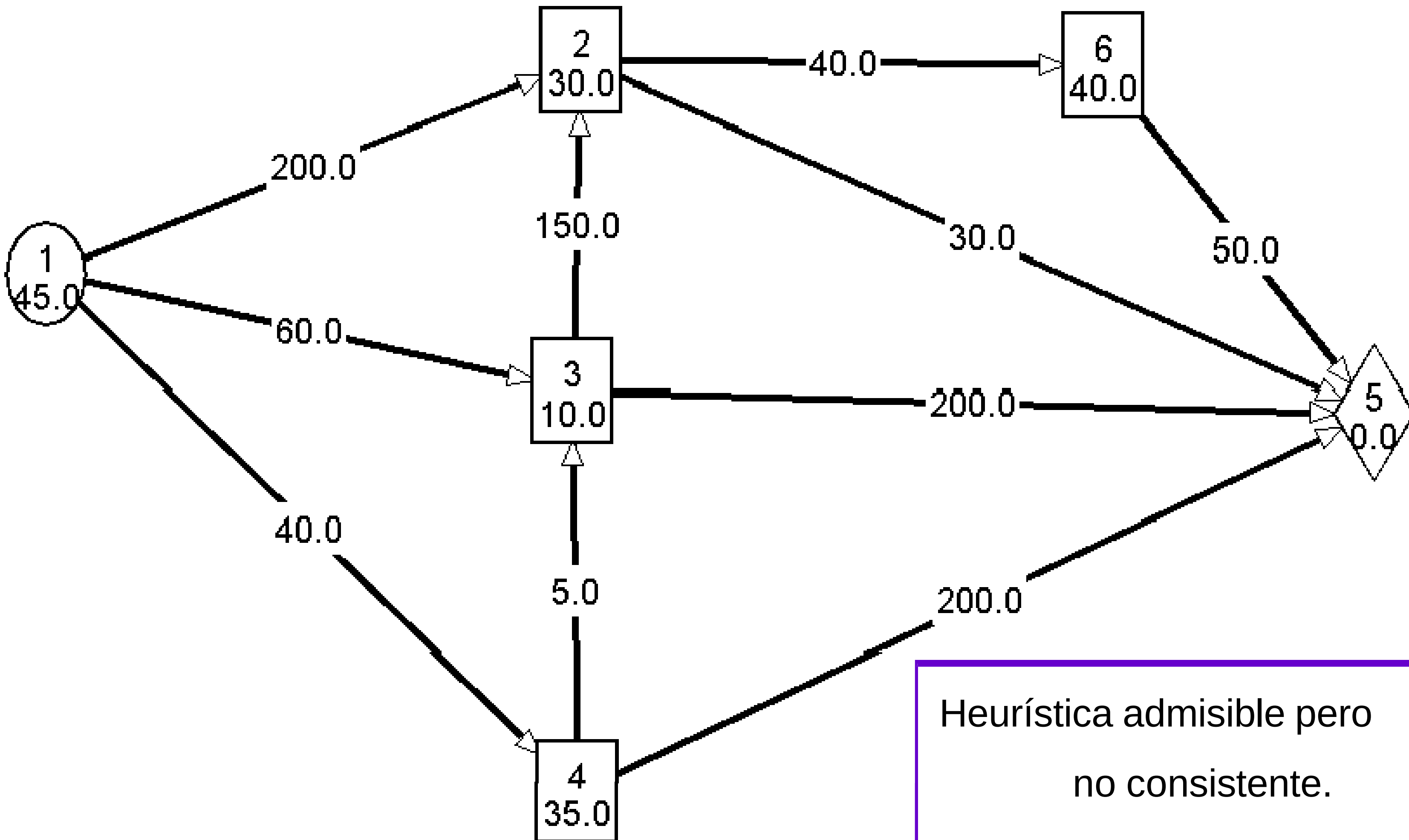
• En heurísticas mejores estas curvas forman bandas que se extienden hacia el estado objetivo.



Resumen: Optimalidad de A* GraphSearch

- **Consistencia de *h* (o monotonía de *f*)**
  - *h* es consistente si, para cada nodo *n* y cada sucesor *n'* de *n,* el coste estimado de alcanzar el objetivo desde *n* no es mayor que el coste real de alcanzar *n'* más el coste estimado de alcanzar el objetivo desde *n'*
    - $h(n) \leq c(n, n') + h(n')$ *(desigualdad triangular)*
    - Toda heurística consistente también es admisible *(pero no al revés)*
    - Si *h* es consistente entonces los valores de *f* a lo largo de cualquier camino no disminuyen *(f monótona no decreciente)*
  - Si *f* es monótona no decreciente, la secuencia de nodos expandidos por A* estará en orden no decreciente de *f (n): f (n) <= f (n')*
    - El primer nodo objetivo seleccionado para la expansión debe ser una solución óptima, ya que todos los posteriores serán al menos tan costosos como él
  - Si h es consistente, cada vez que expanda un nodo
    - habrá encontrado un **camino óptimo** a dicho nodo desde el inicial
    - Incrementa la eficiencia al no necesitar revisitar nodos: 1ª expansión, la mejor

Ejemplo 2



Heurística admisible pero no consistente.

$h(4) > c(4,3) + h(3)$

Optimalidad de A* según sus implementaciones

- A* + búsqueda-en-árbol (sin eliminación de estados repetidos)
  - Es óptima si la heurística es admisible.
- A* + búsqueda-en-grafo (con eliminación de estados repetidos)
  - Es óptima si la heurística es consistente.

Para garantizar la optimalidad de A*

- Si h no es consistente, pero sí admisible, podemos hacerla consistente
  - puede modificarse h dinámicamente durante la búsqueda para
  - que cumpla la condición de consistencia $h(n) \leq c(n, n') + h(n')$
    - En cada paso, comprobamos los valores de h
      - para los sucesores del nodo n que acaba de ser expandido
    - Si para alguno de estos valores de h' se cumple que
    - $h(n') < h(n) - c(n, n')$ entonces hacemos $h(n') = h(n) - c(n, n')$
      - *En el ejemplo: nuevo valor de* $h(3) = h(4) - c(4, 3) = 35 - 5 = 30$
- Comportamiento de A* con h consistente
  - Si f^* es el coste de la solución óptima, entonces
    - A* expande todos los nodos con $f(n) < f^*$
    - A* podría expandir algunos nodos directamente sobre "la curva de nivel objetivo" (donde $f(n) = f^*$) antes de seleccionar un nodo objetivo
    - A* no expandirá ningún nodo con $f(n) > f^*$ (ahí está la poda)

Completitud y eficiencia de A*con heurísticas consistentes

- Completitud
  - Si existe solución, tendrá que llegar a un nodo objetivo, salvo que haya una sucesión infinita de nodos *n* en los que se cumpla $f(n) \leq f^*$
  - Esto puede ocurrir si
    - A. Hay nodos con factor de ramificación infinito, ó
    - B. Si hay caminos de coste finito con un número infinito de nodos
  - *A* es completo*
    - si el factor de ramificación ***b*** es finito y
    - existe una constante $\varepsilon > 0$ tal que
      - el coste de cualquier operador es siempre $\geq \varepsilon$
- Es óptimamente eficiente
  - Ningún otro algoritmo óptimo garantiza expandir menos nodos que A*
    - Salvo quizás los desempates entre nodos con igual valor de *f*
    - Esto es debido a que cualquier algoritmo que no expanda todos los nodos con $f(n) < f^*$ corre el riesgo de omitir la solución óptima

Complejidad de A*

- Con las restricciones establecidas, la búsqueda A* es completa, óptima y óptimamente eficiente,
  - pero A* no es la respuesta a todas las necesidades de búsqueda
- En el caso peor sigue siendo exponencial:

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

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

- El crecimiento exponencial no ocurre si
  - El error en la heurística no crece más rápido que el logaritmo del coste real

    $| h(n) - h^*(n) | \leq O(\log h^*(n))$
- En la práctica, para casi todas las heurísticas, el error es al menos
  - proporcional al coste del camino $| h(n) - h^*(n) | \approx O(h^*(n))$ y no a su logaritmo
- El crecimiento exponencial desborda la capacidad de cualquier ordenador

  ➔ exponencial en tiempo y en <u>espacio</u>
  - Necesita mantener todos los nodos generados en memoria
  - Nada adecuada para problemas grandes

Variantes para solucionar el crecimiento exponencial

- En la práctica, no es conveniente insistir en la optimalidad
  - Se usan variantes de A*: encuentran rápidamente soluciones subóptimas
  - Se utiliza A* con heurísticas ligeramente no admisibles para obtener soluciones ligeramente subóptimas
    - Acotando el exceso de *h* sobre *h* podemos acotar el exceso en coste de la solución alcanzada con respecto al coste de la solución óptima
  - En cualquier caso, el empleo de buenas heurísticas proporciona enormes ahorros comparados con el empleo de una búsqueda no informada
- Algunas variantes de A*:
  - RTA* (Real Time A*) : acota el tiempo
    - Tareas de tiempo real: obligan a tomar una decisión cada cierto tiempo
  - IDA* (Iterative Deepening A*): acota el coste
    - Límite con *f*: expande sólo estados con coste inferior a ese límite
  - SMA* (Simplified Memory-bounded A*): acota el espacio
    - Si al generar un sucesor falta memoria,
      - se libera el espacio de los nodos de *abiertos* menos prometedores

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

$$C_0 = f(n_0);$$

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

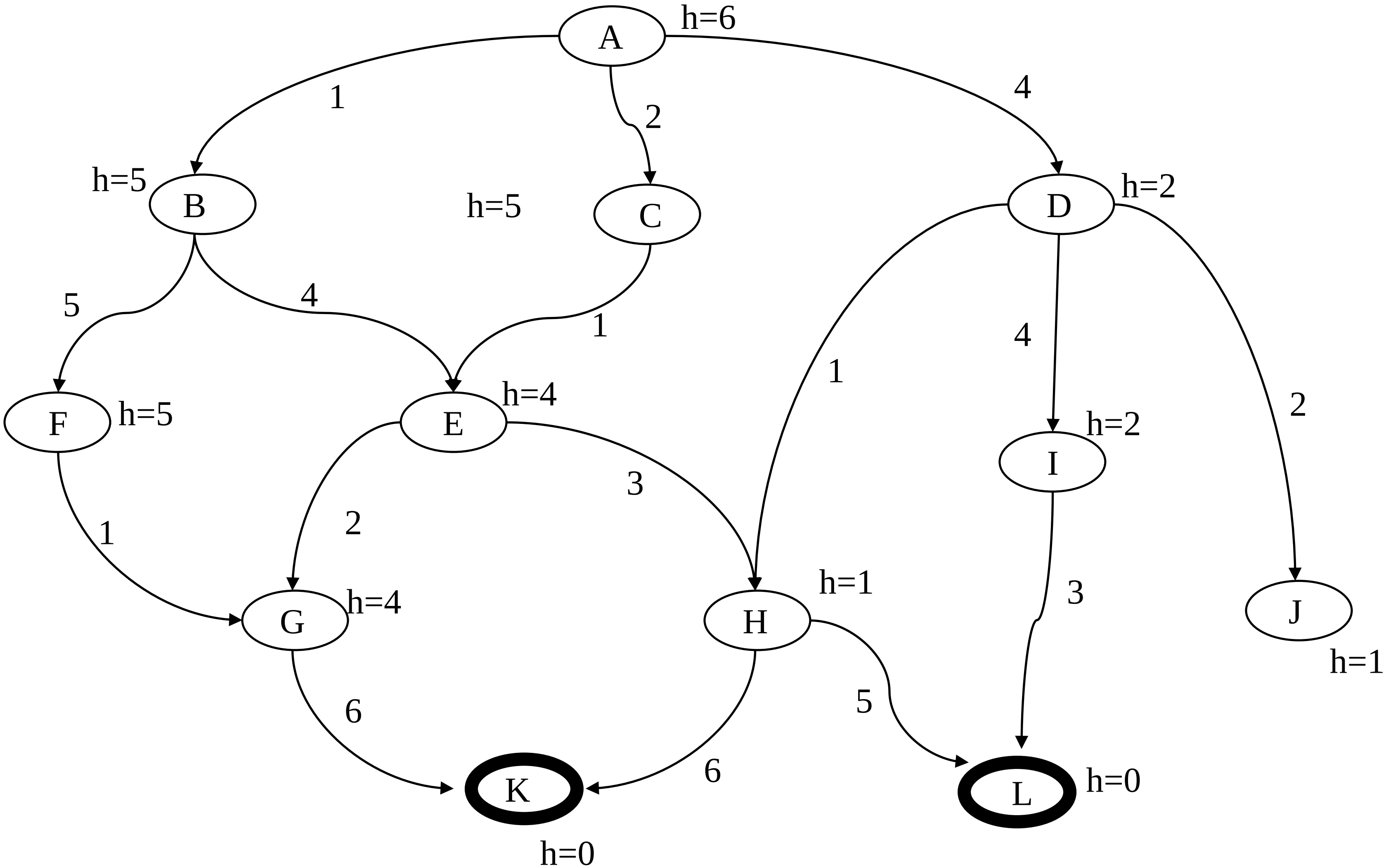
  - Iteración k: expandir todos los nodos que cumplan $\{n \mid f(n) \leq C_k\}$

- 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

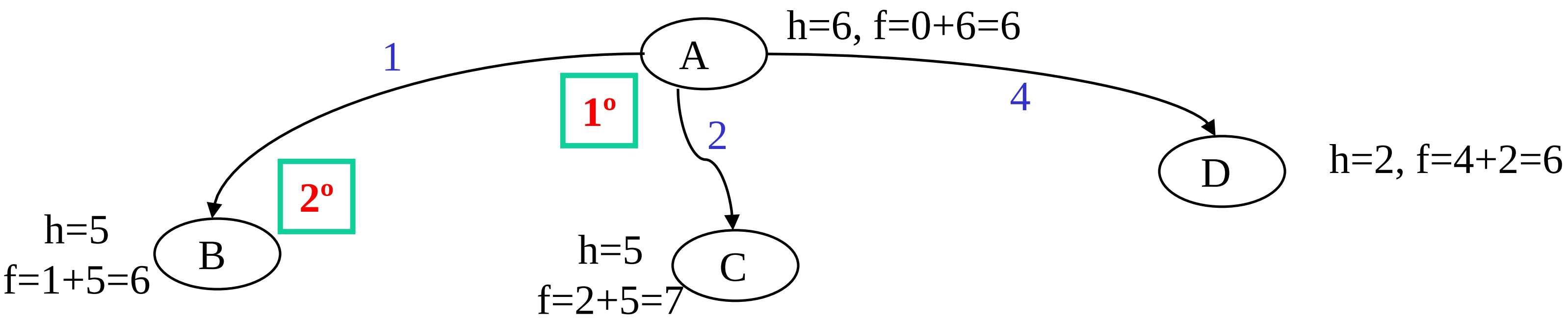
A* + búsqueda en árbol

A

h=6, f=0+6=6

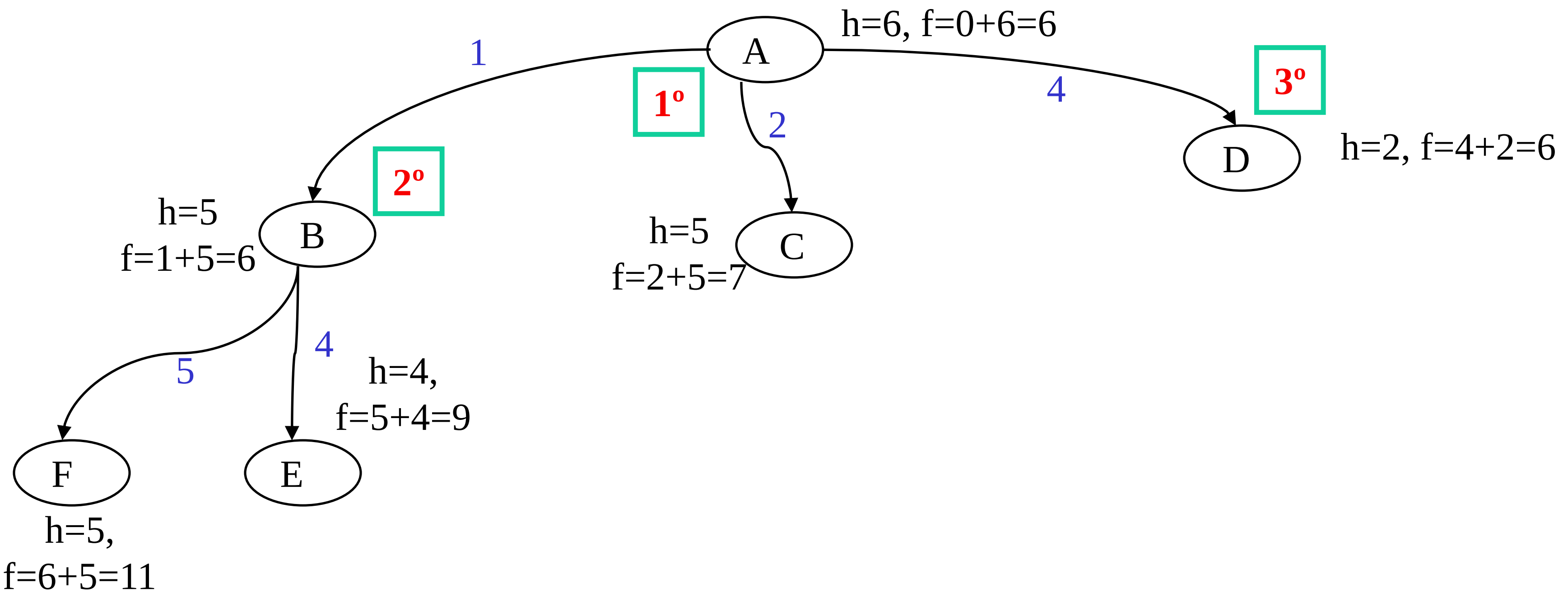
1º

A* + búsqueda en árbol



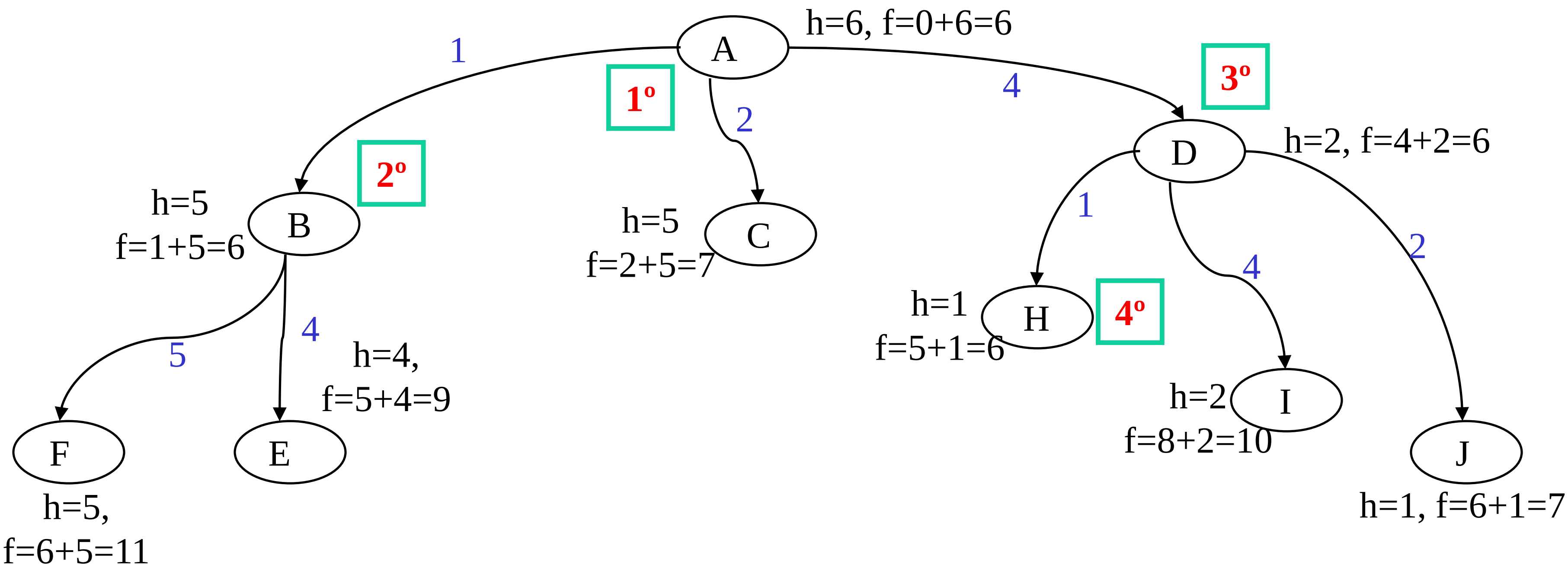
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



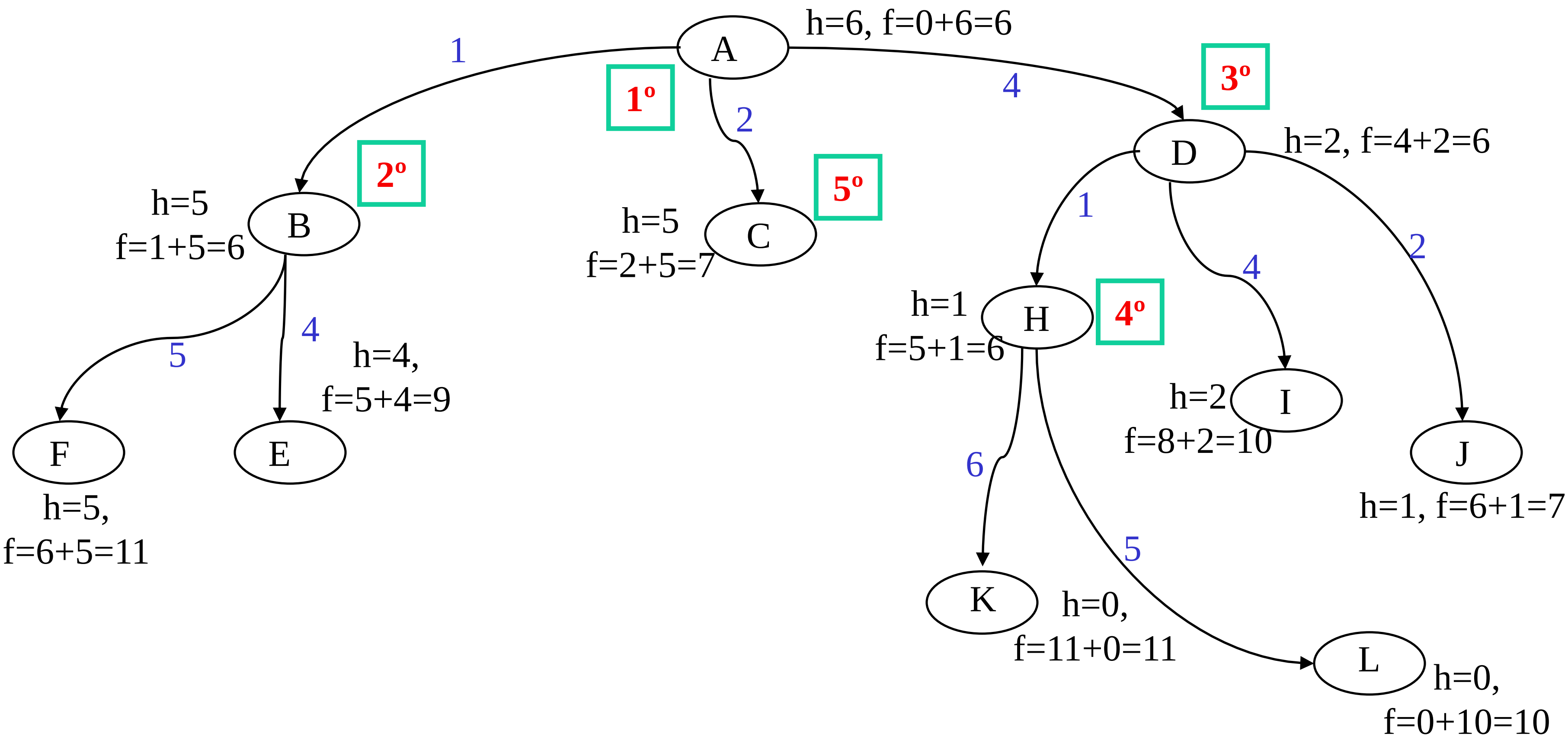
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



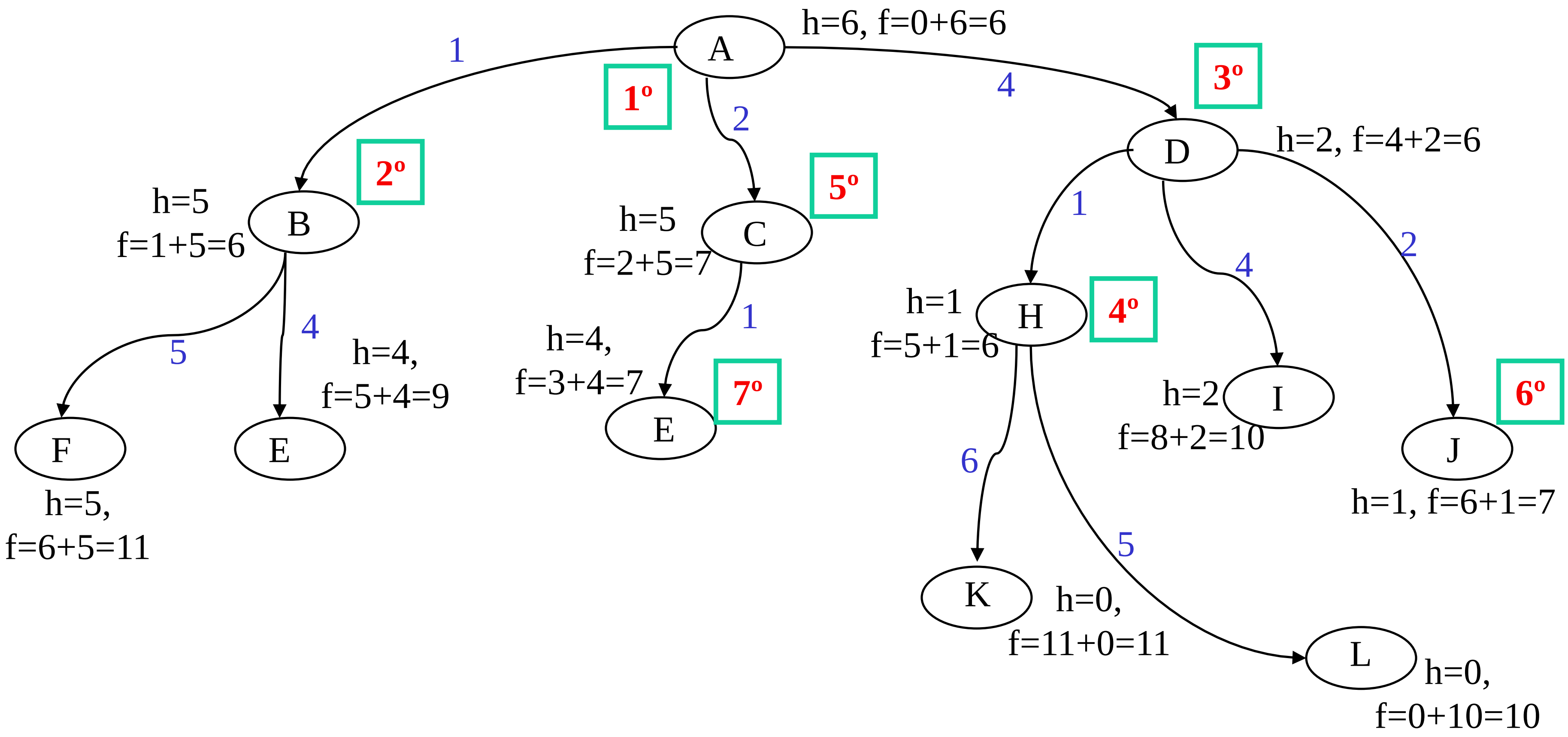
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



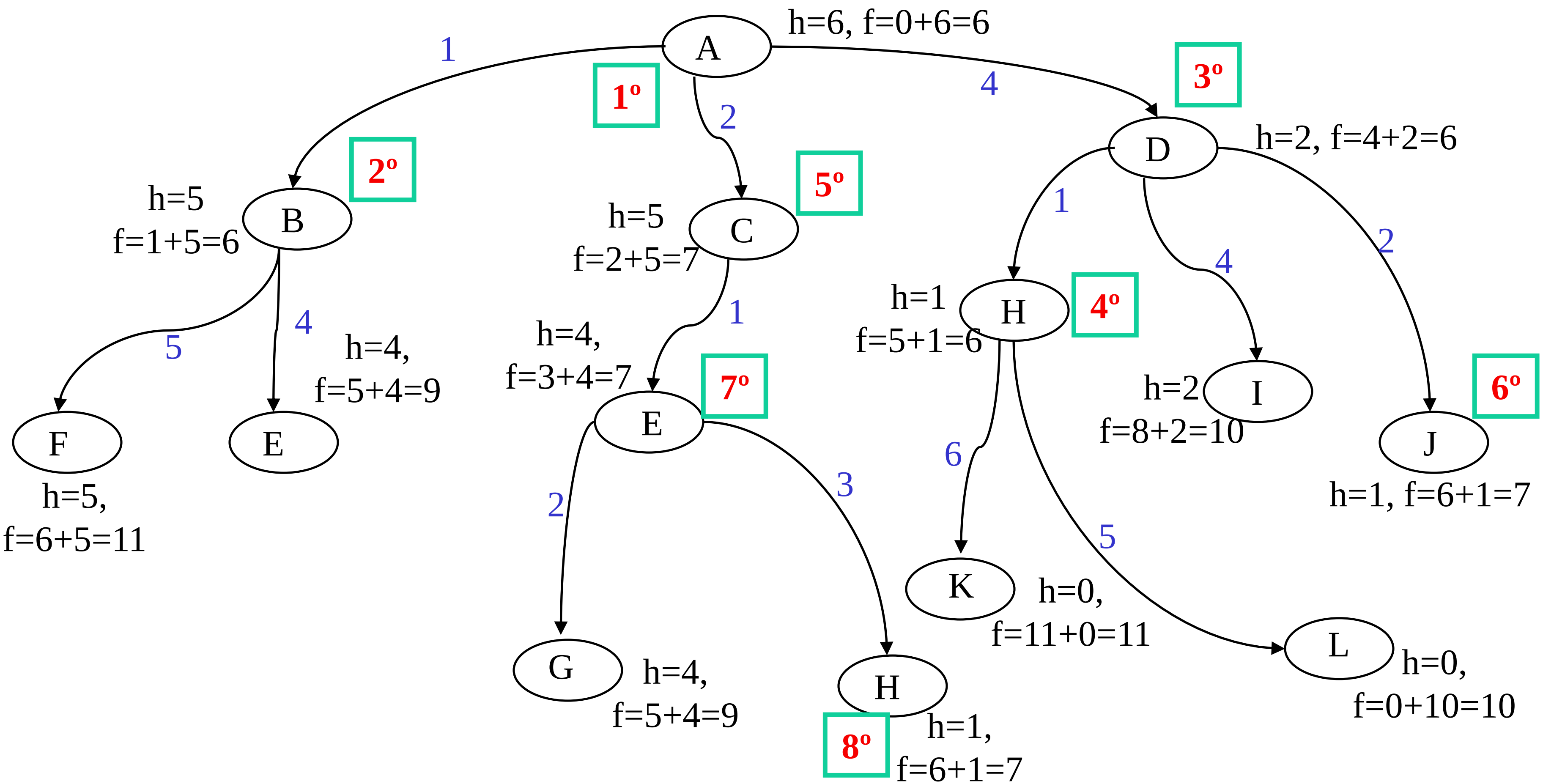
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



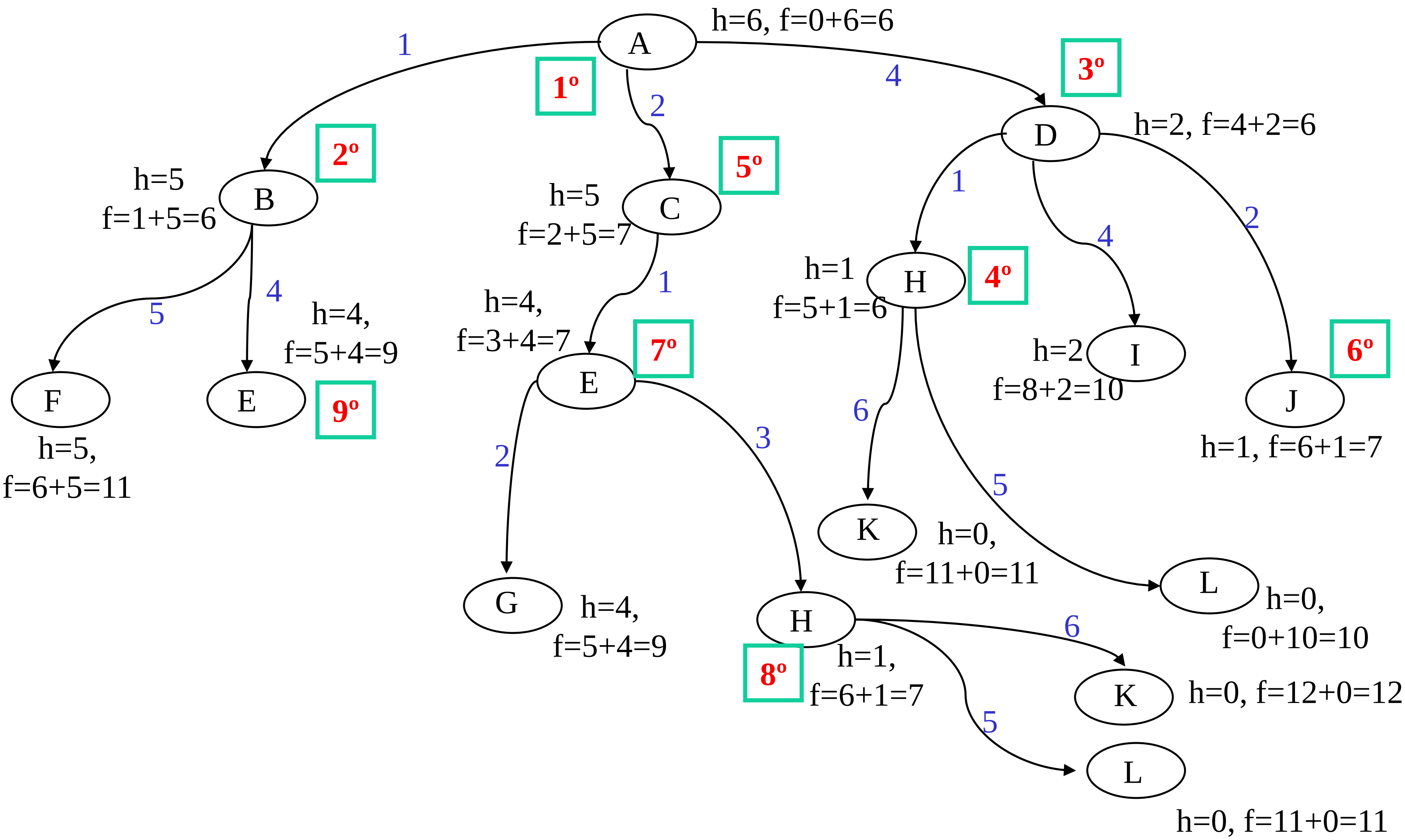
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



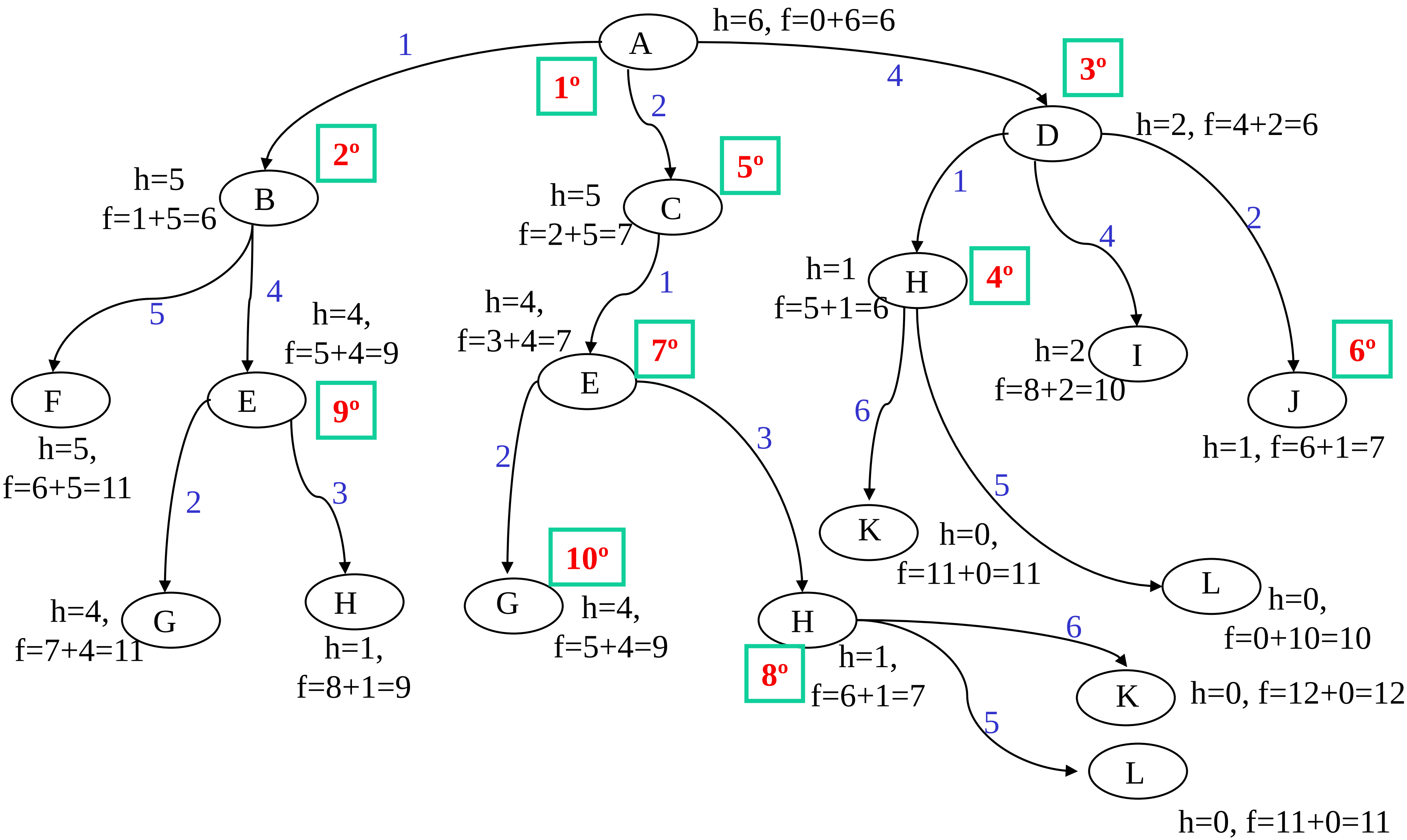
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



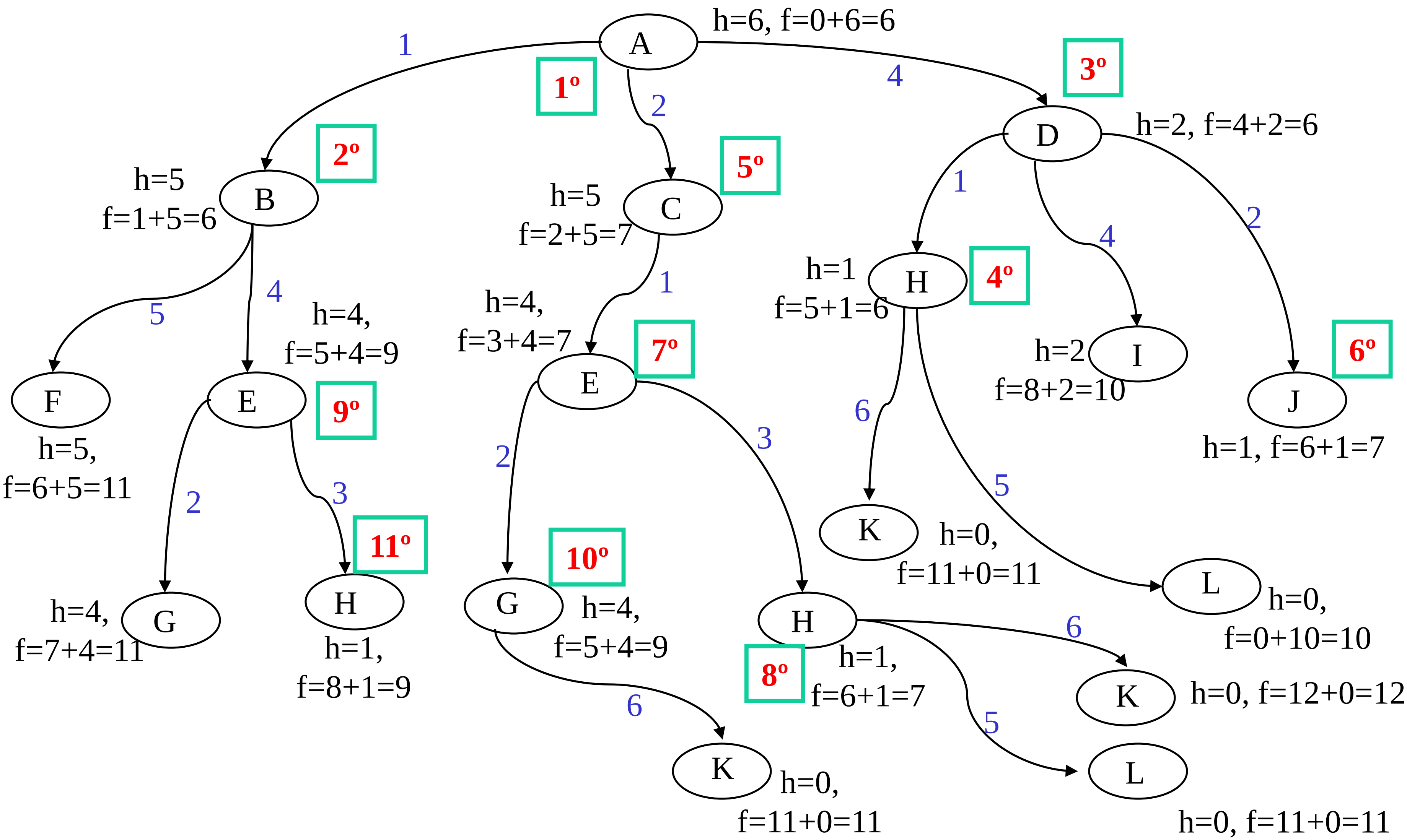
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



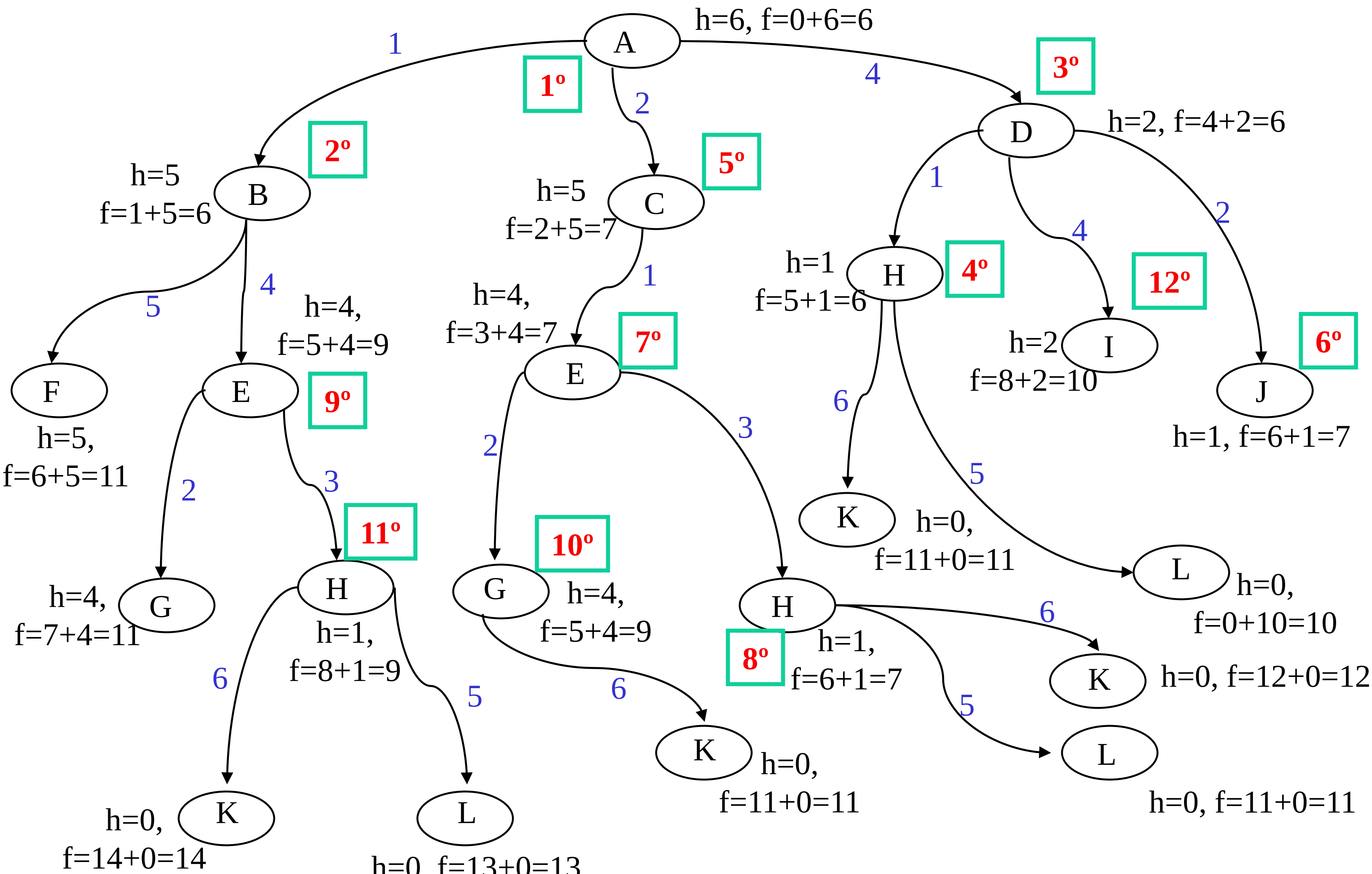
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



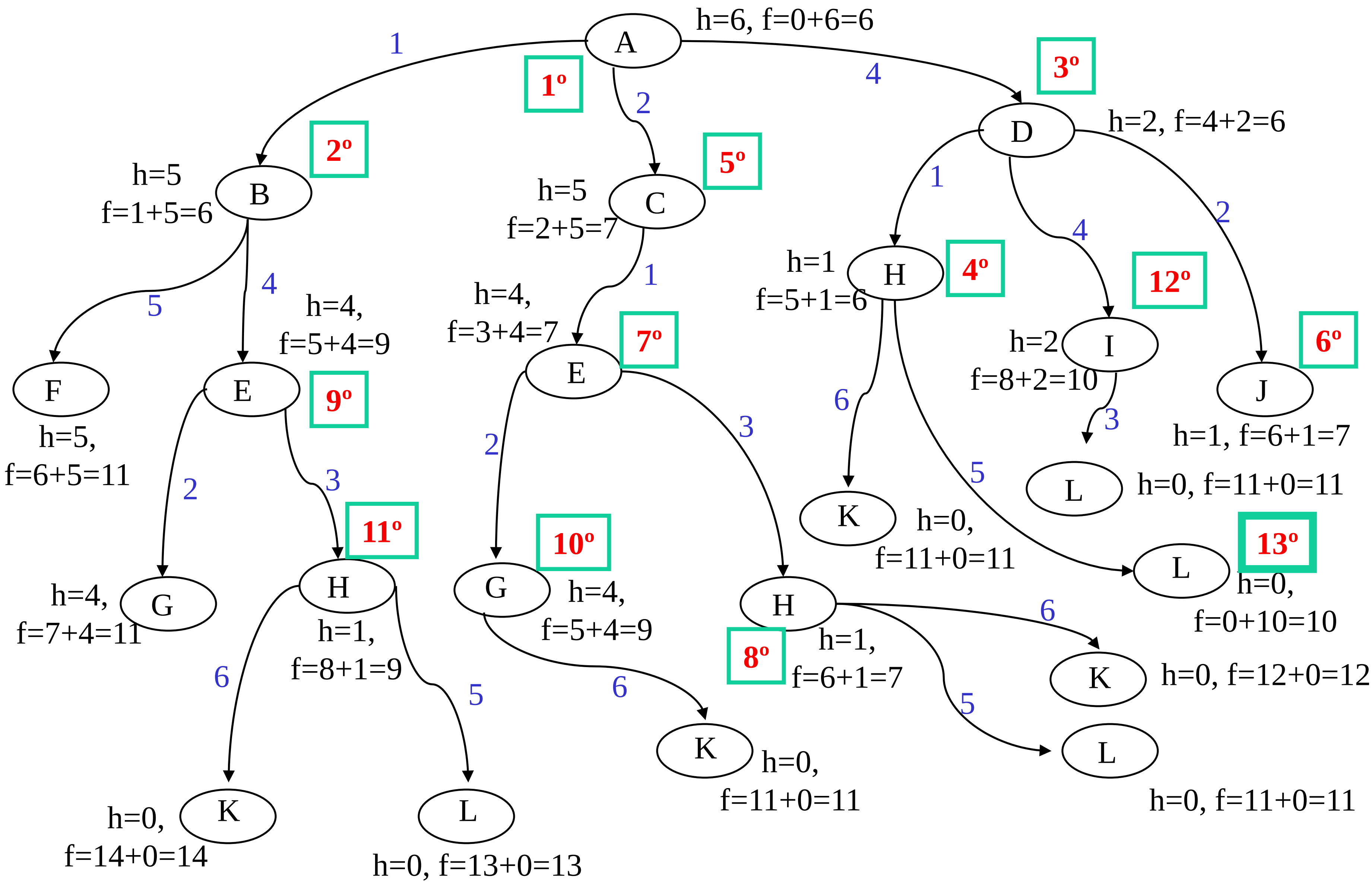
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



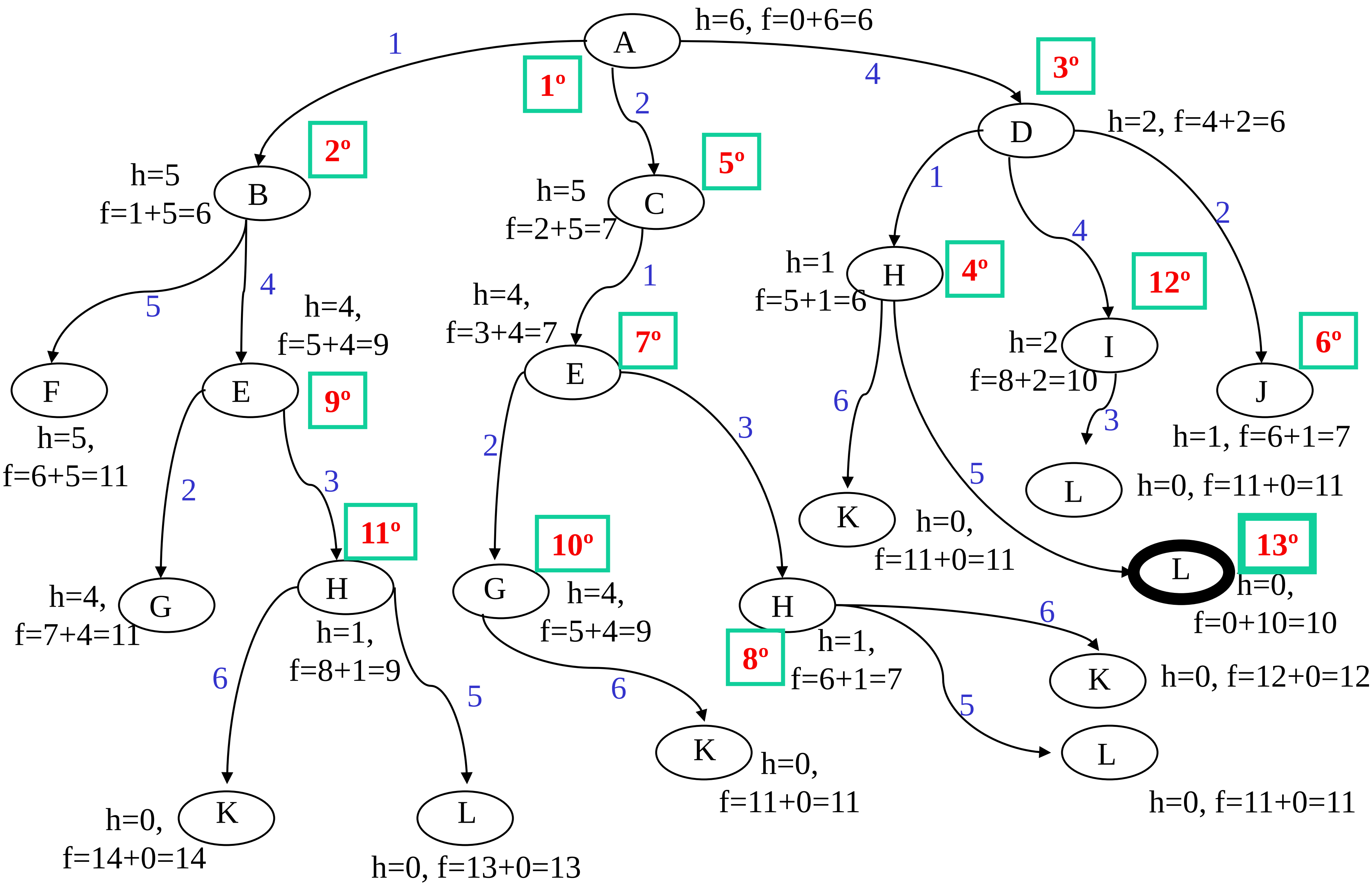
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



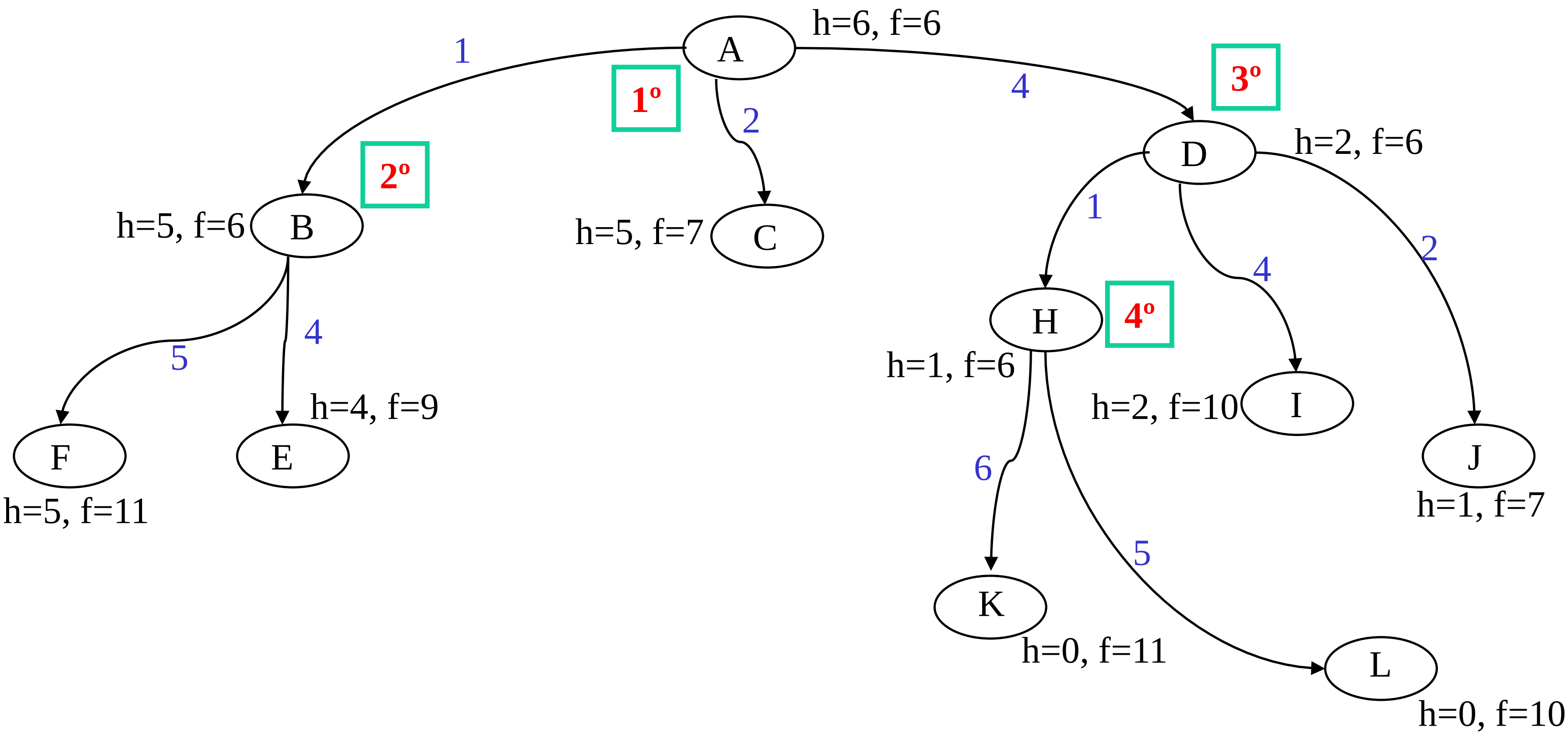
Nota: Los empates se resuelven expandiendo primero los nodos más antiguos (1) + orden alfabético (2)

A* + búsqueda en árbol



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

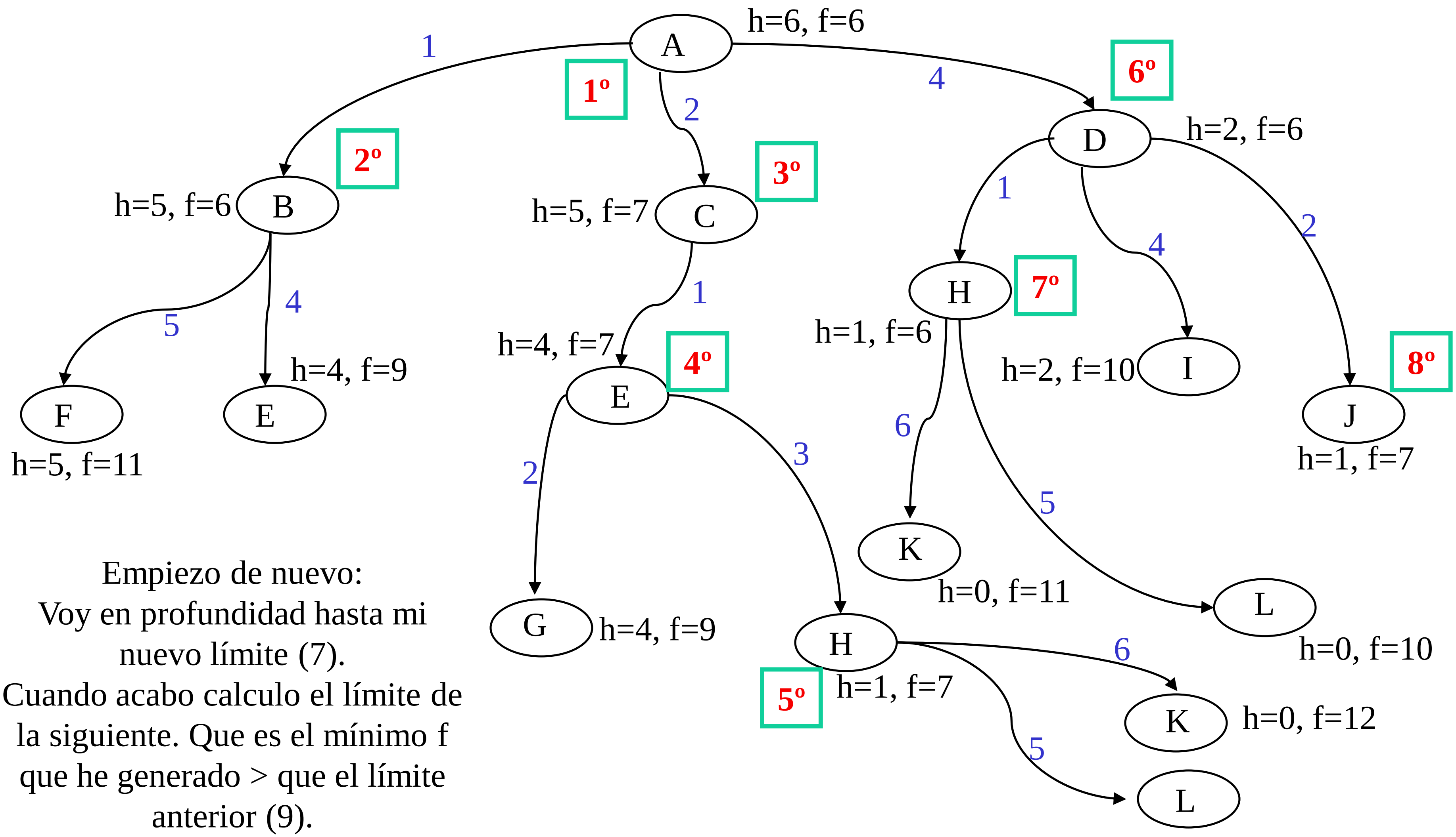
IDA* + búsqueda en árbol, I



Voy en profundidad hasta mi límite, i.e no expando nada > h=6.
Cuando acabo calculo el límite de la siguiente. Que es el mínimo f que he generado > que el límite anterior, i.e 7.

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

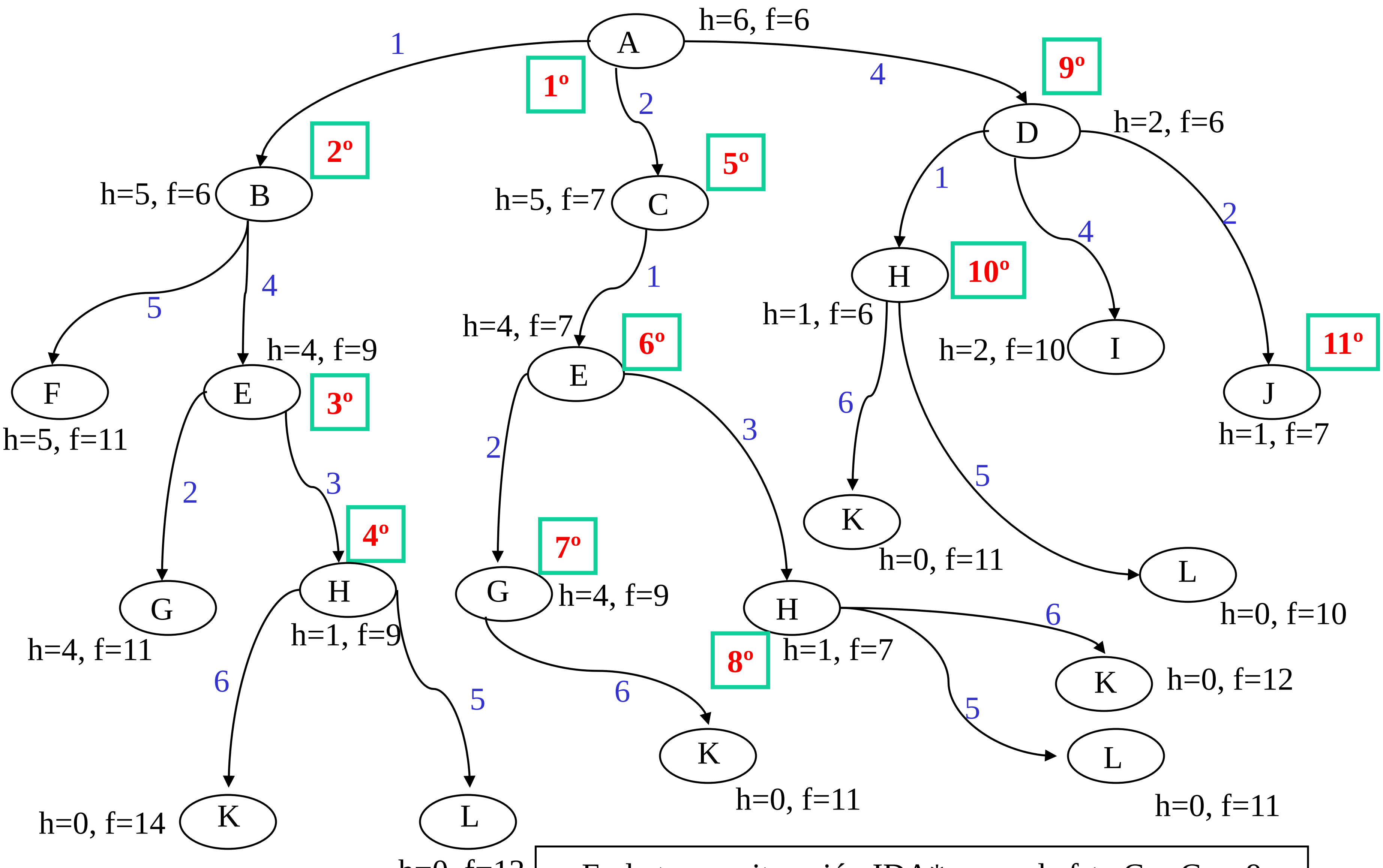
IDA* + búsqueda en árbol, II



Empiezo de nuevo:
Voy en profundidad hasta mi nuevo límite (7).
Cuando acabo calculo el límite de la siguiente. Que es el mínimo f que he generado > que el límite anterior (9).

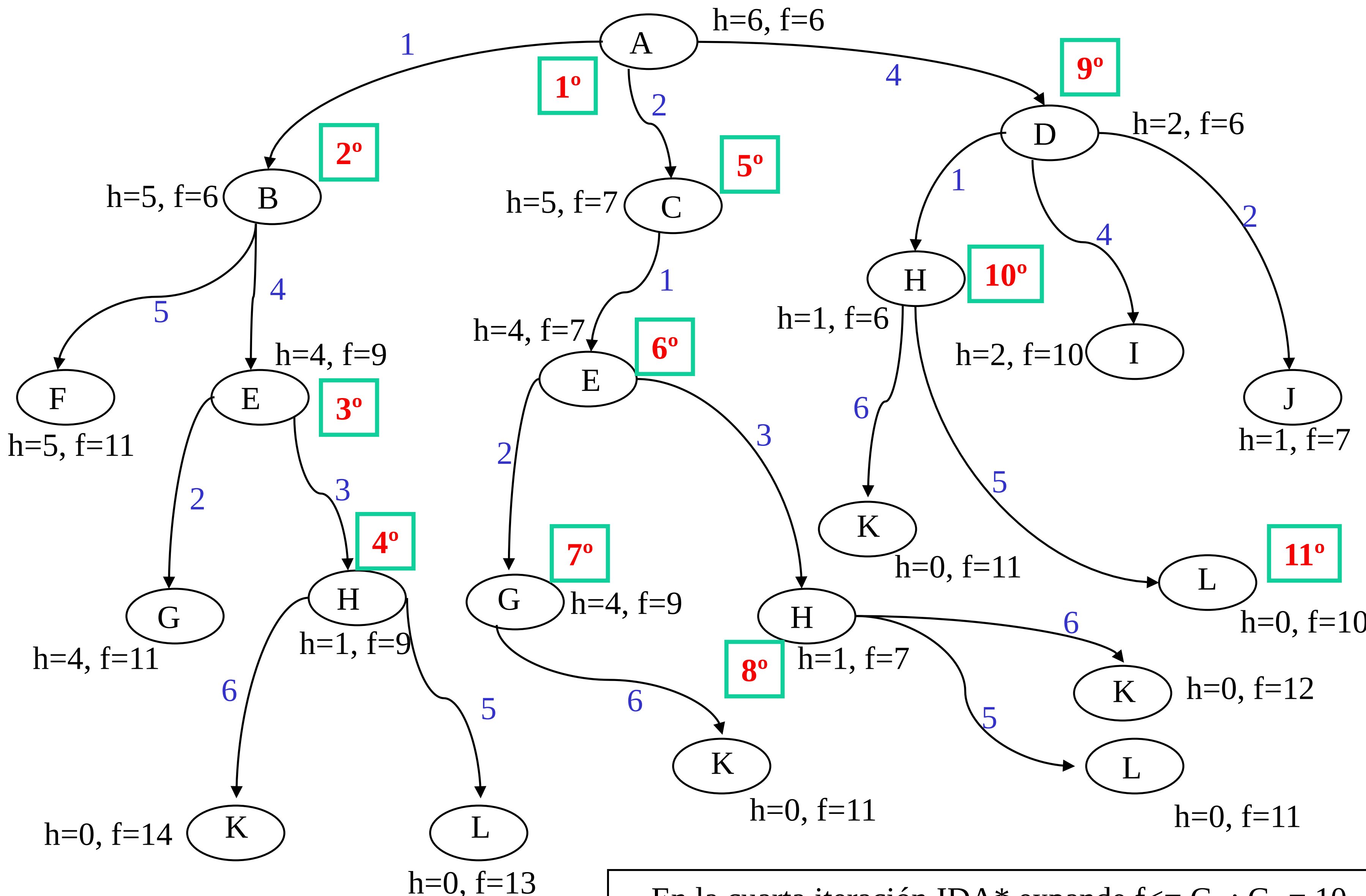
En la segunda iteración IDA* expande f<= C_1 ; C_1 = 7
C_2 = 9

IDA* + búsqueda en árbol, III



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

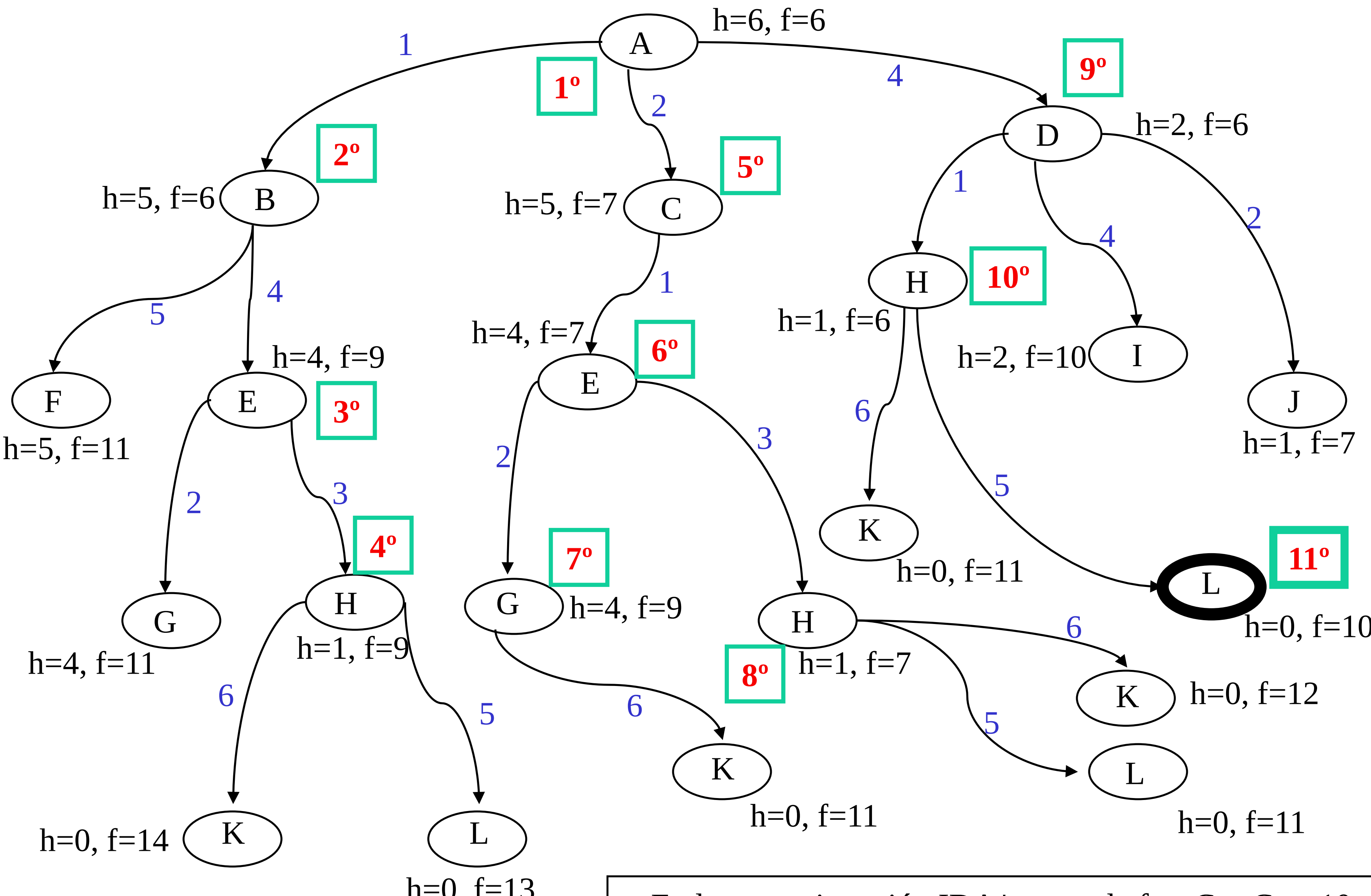
IDA* + búsqueda en árbol, IV



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

3. Comparación de la calidad de heurísticas

A)- Evaluación ad-hoc: Qué valorar?

- Rango amplio (número de valores posibles)
- Valores diferentes para cada sucesor inmediato a un nodo
- Que sea posible aplicarla a todos los estados permitidos
- Sea 0 para estados objetivo. Distinta de 0 para estados no objetivo
- Basada en características dinámicas del problema,
  - Asigna pesos diferentes según importancia,
  - Usa términos separados para cada característica importante.
- (…si se puede comprobar: admisible y consistente)

B)- Comparar la calidad de dos heurísticas

- Por demostración de dominancia (método teórico)
  - Es mejor la más dominante (más precisión)
- Por factor de ramificación efectivo **b*** (método experimental)
  - Es mejor la que menor b***** tenga

Factor de ramificación efectivo b*

- Si ***N*** es el número de nodos expandidos por un algoritmo de búsqueda (p.e.: A*)
  - para un problema particular y la profundidad de la solución es ***d***,
    - entonces ***b**** es el factor de ramificación que un árbol uniforme ficticio de profundidad ***d*** debería tener para contener ***N*** nodos
- Se cumple: $N = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$

  $52 = 1 + 1.91 + (1.91)^2 + (1.91)^3 + (1.91)^4 + (1.91)^5 \rightarrow b^* = 1.91$
- Conocemos ***d*** y ***N***. Entonces **N = O(b*)**, un cálculo aproximado $b^* = \sqrt[d]{N} \rightarrow 2{,}2$
- ***b**** es prácticamente constante a partir de ciertas profundidades
- Las medidas experimentales de ***b**** sobre un conjunto de casos generados aleatoriamente
  - proporcionan una buena guía para la utilidad total de la heurística
- Una buena heurística tendrá un valor de ***b**** cercano a **1**
- Ejemplo: 8-puzzle con *d = 12*
  - *Búsqueda ciega (profundización iterativa): N = 3.644.035, b* = 2.78*
  - *A*(h'$_b$): N = 227, b* = 1.42* — *A*(h'$_a$): N = 73, b* = 1.24*

    descolocadas — *Manhattan*

Evaluación de la heurística

- 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.

Generación de heurísticas: Tres Métodos

- Tres métodos para definir o generar heurísticas de un problema:

  1.- Relajación: Relajar las restricciones o reglas del problema. Las soluciones exactas del problema relajado se usan como heurísticas para el original.

  2.- Abstracción: Definir problemas derivados del original. Sus soluciones exactas se almacenan y se usan como heurísticas del problema original.

  3.- Aprendizaje: Resolver muchos problemas del tipo dado y extraer conclusiones sobre su comportamiento

  Ej. AlphaStar aprende a evaluar pantallas de StarCraft 2 mediante una red neuronal profunda.

  https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

Búsqueda de heurísticas

- Si hay disponibles varias heurísticas admisibles ($h_1, h_2, \dots h_K$),

  la heurística $h_{max}(n) = \max\{h_1(n), h_2(n), \dots h_K(n)\}$ es admisible y domina a $h_1, h_2, \dots h_K$.
- **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

Funciones heurísticas: Pasos para diseñar h

1.- Analizar el problema: escoger qué características son relevantes para la h

- Características
  - Dinámicas : qué cambia entre estados
  - Estáticas: se mantienen igual entre estados
  - Si las relajamos, qué características simplifican el problema? → Relajación
- Estudiar las restricciones del problema (ej.: las reglas del juego) → Relajación
  - Si las relajamos, cuáles simplifican el problema ?
- Generar árboles parciales de estados: nodos al azar + varios niveles
  - Qué dificultades aparecen para llegar a la solución?
    - Podemos eliminar detalles y hacer un problema más simple → Abstracción
    - que podamos resolver y <u>calcular</u> el coste para usarlo como heurística?
  - Hay estados difíciles que necesiten alguna relajación especial?
    - Hay interacciones entre los elementos que perjudiquen el avance

2.- Crear la h con varios términos: varias h combinadas: $3 * h_a(n) + 5 * h_c(n)$

- Qué características y restricciones deben participar en h (son los términos)
  - Escoger aquellas características, restricciones y interacciones
  que afecten en mayor grado a la resolución del problema
  - El mayor o menor grado decide el peso de cada término: el valor que multiplica

Resumen Criterios de calidad: h cumple la mayoría?

1. h es aplicable sobre todos los estado válidos?
2. h guía hacia el objetivo?
   - El rango de valores que genera h es amplio?
   - Los estados vecinos / hijos tienen valores de h diferentes?
3. h(estados objetivo) = 0 ?
4. h es consistente?   (o admisible al menos)
5. Si puedes encontrar otra función que “domina” a h : úsala
6. Si puedes hacer experimentos estudia que b* sea cercano a 1

Tema 2: Resolución de problemas y espacio de búsqueda

3. **Métodos informados o heurísticos**
   - Introducción
   - Búsqueda primero el mejor
   - Algoritmos de mejora iterativa
     - Introducción
     - Escalada simple
     - Escalada por máxima pendiente
     - Enfriamiento simulado

Algoritmos de mejora iterativa: Búsqueda Local

- En muchos problemas el camino al objetivo es irrelevante
  - 8-reinas lo que importa es la configuración final
  - Dominios:
    - diseño de circuitos integrados, disposición del suelo, planificación del trabajo,
    - programación automática, optimización de redes, gestión de carteras...
- Una clase diferente de algoritmos que no se preocupan de los caminos
    - *ignoran el coste del camino, en particular*
- Algoritmos de búsqueda local funcionan
  - con un solo estado actual y, generalmente,
  - se mueven sólo a los vecinos del estado
      - No como A* o voraz que dan saltos en el espacio de búsqueda, guiados por *f*
- Aunque no son sistemáticos, tienen dos ventajas clave:
  - Usan muy poca memoria
    - Los caminos seguidos por la búsqueda no suelen retenerse
  - Encuentran soluciones aceptables en espacios de estados grandes

Métodos informados de optimización local

- Algoritmos de búsqueda local resuelven problemas de optimización puros
  - El objetivo es encontrar el mejor estado según una cierta función objetivo
- Métodos informados de optimización local: algunos problemas de optimización
  - La solución en sí tiene un coste asociado que se quiere optimizar
    - EJ: en el problema de la mochila optimizo la solución
    - Pero el coste del camino es indiferente
  - Planteamiento habitual como búsqueda en el espacio de soluciones
  - Extrapolable a búsqueda en espacio de estados *(usando heurística)*
- Un tipo son los Algoritmos de escalada
  - Consumen pocos recursos
  - Pero pueden quedarse bloqueados o atascados en un óptimo local
  - Complejidad constante en espacio: *abiertos* nunca posee más de un nodo
    - Irrevocables: se mantiene en expectativa un único camino *(sin vuelta atrás)*
  - Ni óptimos ni completos
  - Podan sensiblemente el espacio de búsqueda pero sin ofrecer garantías

Escalada simple *(hill climbing)*

- El nombre viene de usar valores mayores para nodos mejores
  - Es como escalar una montaña
  - Nosotros usamos la otra versión: mejor es cuando es menor valor
- En cada paso, el nodo actual se intenta sustituir por el primer vecino mejor
  - *Primer sucesor con una medida heurística más baja que el nodo actual*
- Intenta moverse en dirección de un valor mejor
  - *cuesta abajo, busca un valor decreciente*
- Termina cuando
  - encuentra una solución o
  - alcanza un nodo en el que ningún vecino tiene valor *más bajo*
- No mantiene un árbol, sino solo el nodo actual
  - el estado y su valor según la función objetivo a optimizar (solo usa h)
- No mira adelante más allá del vecino inmediato del estado actual
- Es como un "genera y prueba"
  - matizado por una función heurística (le da conocimiento del dominio)
  - Se usa si sólo se tiene una buena h y ningún otro conocimiento útil

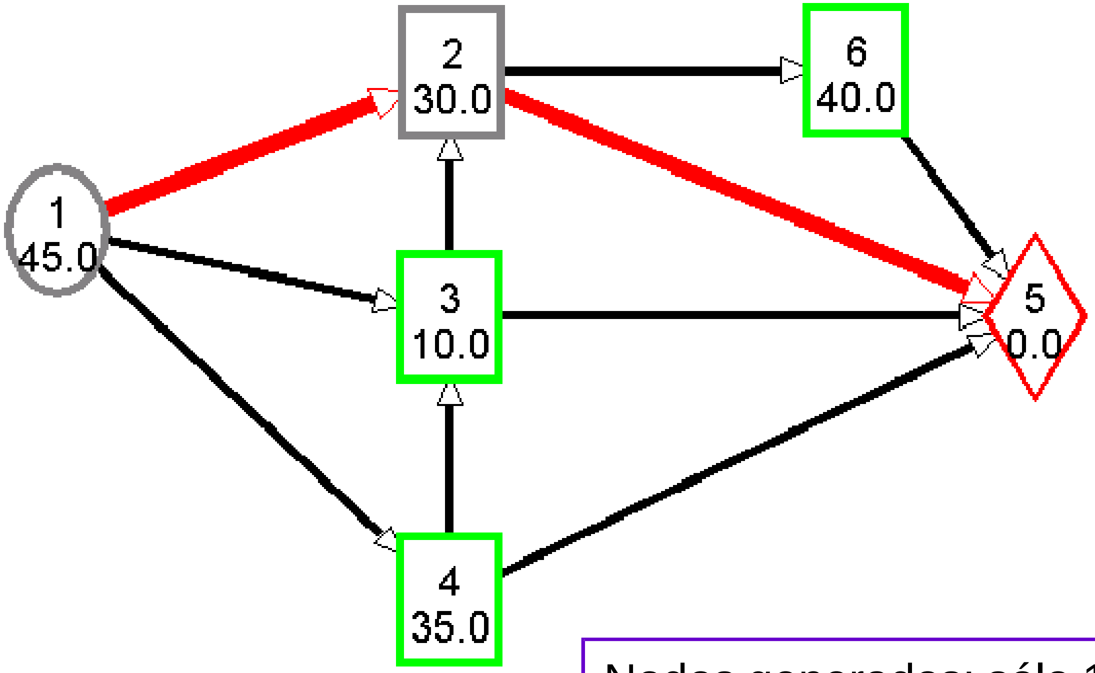
Escalada simple *(hill climbing)*

```
evaluar el estado INICIAL
si es un estado objetivo entonces devolverlo y parar
si no ACTUAL := INICIAL
mientras haya operadores aplicables a ACTUAL y
          no encontrada solución hacer
   seleccionar un operador no aplicado todavía a ACTUAL
   aplicar operador y generar NUEVO_ESTADO
   evaluar NUEVO_ESTADO
   si es un estado objetivo entonces devolverlo y parar
   si no
        si NUEVO_ESTADO tiene mejor h’ que ACTUAL entonces
               ACTUAL := NUEVO_ESTADO
```

- Como 1º en profundidad guiado por *h*, pero sólo desciende si mejora
- Muy dependiente del orden de generación de hijos

Solución con escalada simple

Orden de generación de hijos: orden creciente



Nodos generados: sólo 1, 2 y 5

Solución: 1-2-5 Coste (no tenido en cuenta): 200+30 = 230

Escalada por máxima pendiente: Ascenso por gradiente

- Variante: estudia todos los vecinos del nodo actual
- En cada paso, el nodo actual se sustituye por el mejor vecino
  - Vecino con el mejor valor de h, es el
  - *Sucesor con medida heurística más baja*
    - El que supone un descenso más abrupto de *h*,
    - con lo que desciende por el camino de máxima pendiente
- Se mueve en dirección del valor decreciente, es decir, *cuesta abajo*
- Termina cuando encuentra
  - una solución o
  - alcanza un nodo en el que ningún vecino tiene valor *más bajo*
- No mira adelante más allá de los vecinos inmediatos del estado actual
- A veces se le llama búsqueda local avariciosa/voraz
  - porque toma el mejor estado vecino sin pensar hacia dónde irá después
- Progresa muy rápido hacia una solución,
  - pero suele atascarse por varios motivos en mínimos locales

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

- **Newton-Raphson**

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

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

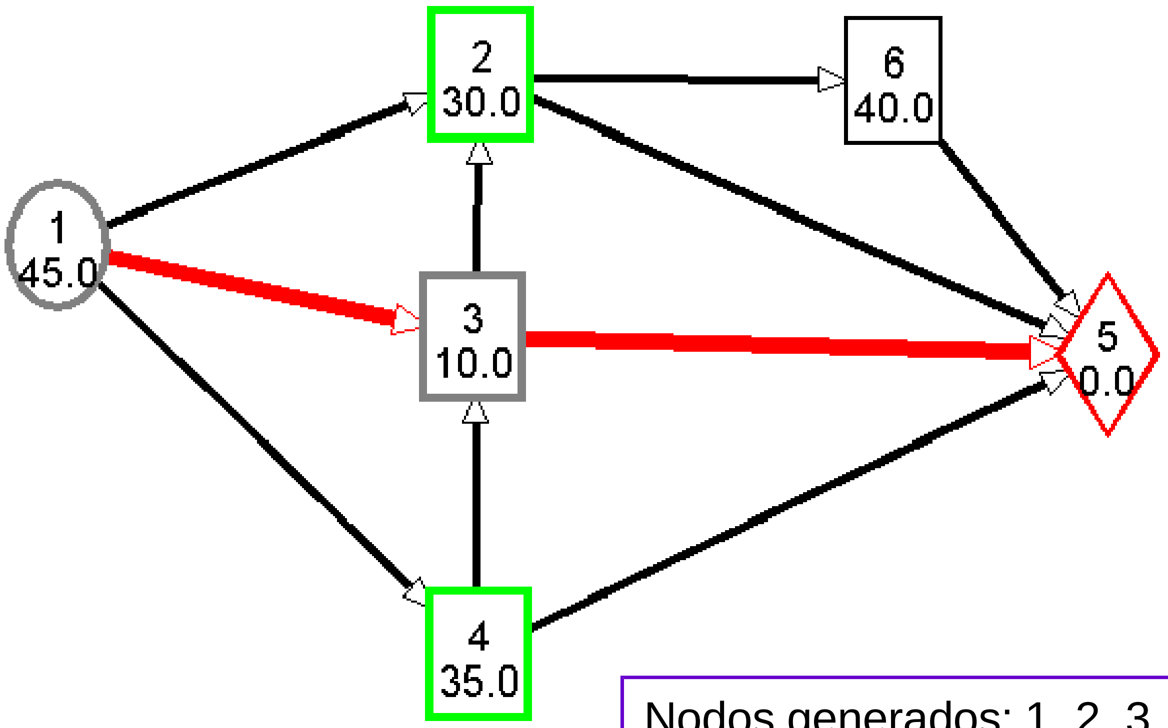
- **Cuasi-Newton**
- **Gradiente conjugado**
- **<u>Optimización con restricciones</u>**: lineales, cuadráticas, programación no lineal.

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

Escalada por máxima pendiente

```
evaluar el estado INICIAL
si es un estado objetivo entonces devolverlo y parar
si no ACTUAL := INICIAL
mientras no parar y no encontrada solución hacer
    SIG := nodo peor que cualquier sucesor de ACTUAL
    para cada operador aplicable a ACTUAL
        aplicar operador y generar NUEVO_ESTADO
        evaluar NUEVO_ESTADO
        si es un objetivo entonces devolverlo y parar
         si no
            si NUEVO_ESTADO es mejor que SIG entonces
               SIG := NUEVO_ESTADO
     si SIG es mejor que ACTUAL entonces ACTUAL := SIG
     si no parar
```

Solución con escalada por máxima pendiente



Nodos generados: 1, 2, 3, 4 y 5

Solución: 1-3-5 Coste (no tenido en cuenta): 60+200 = 260

Problemas de los algoritmos de escalada

- Pueden no encontrar una solución:
  - estado que no es objetivo y que no tiene vecinos mejores.
- Esto sucederá si el algoritmo ha alcanzado:
  1. Un *mínimo local* (u óptimo local)
     - Un estado mejor que sus vecinos pero peor que otros estados más alejados
  2. Una meseta
     - Todos los estados vecinos tienen el mismo valor heurístico
     - Es imposible determinar el mejor movimiento: sería búsqueda ciega
  3. Una cresta: Mezcla de los anteriores
     - región en la que h no guía hacia ningún estado objetivo.
     - Puede terminar en un mínimo local o tener un efecto como la meseta

Variantes de los algoritmos de escalada: mejoras

Las variantes mejoran procurando resolver los bloqueos

1. Profundidad + escalada: los nodos de igual profundidad son ordenados poniendo al comienzo los más prometedores y se permite *backtracking*
   - Recupera completitud y exhaustividad
2. Reiniciar toda o parte de la búsqueda
3. Dar un paso más: generar sucesores de sucesores y ver qué pasa
   - Si aparece un óptimo local, volver a un nodo anterior y probar dirección distinta
   - Si aparece una meseta, hacer un salto grande para salir de la meseta
     - ¿Cómo escaparse?
       - Reinicio aleatorio: comenzar búsqueda desde distintos puntos elegidos aleatorios,
         - guardando la mejor solución encontrada hasta el momento
       - No aplicable a problemas de estado inicial prefijado

Temple simulado

- El éxito depende mucho de la forma del paisaje del espacio de estados
    - Problemas NP-duros: suelen tener un nº exponencial de óptimos locales
        - IA: para resolverlos en tiempo aceptable y de forma aproximada

A- Un algoritmo de escalada que nunca hace movimientos en sentido inverso hacia estados "peores" es necesariamente incompleto

- A menudo, conviene empeorar un poco para mejorar después

B- Un algoritmo puramente aleatorio es completo pero muy ineficiente

C- Algoritmo de enfriamiento o temple simulado

- Combina la escalada con elección aleatoria de caminos
- Produce tanto eficiencia como completitud
- En metalurgia, se sigue este proceso para templar metales y cristales
    - calentándolos a una temperatura alta y luego enfriándolos gradualmente,
    - para que el material se solidifique en un estado cristalino de energía baja

Temple simulado

- Es como un problema de minimización: la función a optimizar es la energía **E**
  - En el algoritmo, la **E** es una función a definir: evaluar(estado)
  - Al comenzar, la temperatura **T** es alta

    se permiten movimientos contrarios al criterio de optimización: empeorar
  - Al final del proceso, cuando **T** es baja,

    se comporta como un algoritmo de escalada simple
  - La temperatura **T** va en función del número de ciclos ya ejecutado
  - La planificación del enfriamiento (variación de **T**) se determina empíricamente y está fijada previamente (otra función a definir)
- Si el enfriamiento (disminución T) va lo bastante lento
  - se alcanza un óptimo global con probabilidad cercana a 1
- Se usa mucho en diseño
  - VLSI, en planificación de fábricas, en tareas de optimización a gran escala
  - Parece ser la estrategia de búsqueda informada más utilizada

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 $\beta<1$.

Enfriamiento simulado

```
evaluar(INICIAL)
si INICIAL es solución entonces devolverlo y parar
    si no
          ACTUAL := INICIAL
          MEJOR_HASTA_AHORA := ACTUAL
          T := TEMPERATURA_INICIAL       empieza alta: permite “malos”movtos
mientras haya operadores aplicables a ACTUAL y no encontrado solución
    seleccionar aleatoriamente operador no aplicado a ACTUAL
                                   {escoge movimiento aleatoriamente (no el mejor)}
    aplicar operador y obtener NUEVOESTADO
    calcular ΔE := evaluar(NUEVOESTADO) - evaluar(ACTUAL)
    si NUEVOESTADO es solución entonces devolverlo y parar
          si no . . .
```

Enfriamiento simulado (continuación)

[. . .sino] **si** **NUEVOESTADO mejor que ACTUAL** {si mejora situación}

ACTUAL := NUEVOESTADO {se acepta el movimiento}

si **NUEVOESTADO mejor que MEJOR_HASTA_AHORA** **entonces**
MEJOR_HASTA_AHORA := NUEVOESTADO

si no {si no mejora la situación, se acepta con prob. < 1}

calcular $P' := e^{-\Delta E/T}$
{probabilidad de pasar a un estado peor: se disminuye
exponencialmente con la "maldad" del movimiento, y
cuando la temperatura T baja}

obtener N {nº aleatorio en el intervalo [0,1]}

decide aleatoriamente si quedarse con el mvto

si **N < P'** se acepta el movimiento}
entonces **ACTUAL := NUEVOESTADO**

actualizar **T de acuerdo con la** planificación **del enfriamiento**

se decide a priori la planificación: cómo de deprisa queremos disminuir T

devolver **MEJOR_HASTA_AHORA como solución**

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.