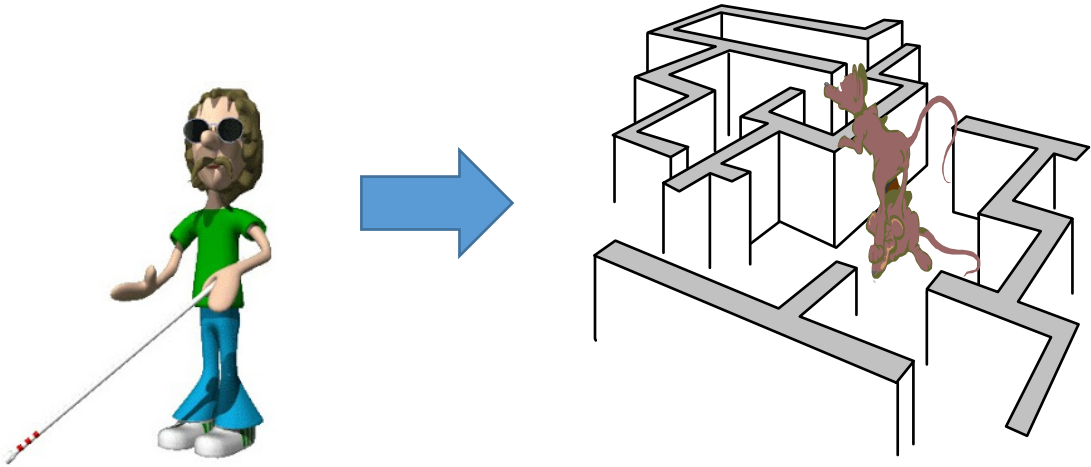
Tema 2.3 Búsqueda heurística

Tema 2. Resolución de problemas con búsqueda

- Representación de problemas y espacio de estados
- Búsqueda de soluciones
  - Búsqueda ciega
  - **Búsqueda heurística**
  - Búsqueda local
  - *Planificación*
- Aplicaciones



Material desarrollado por distintos autores del departamento de ISIA

BÚSQUEDA CIEGA	Completa	Óptima	Eficiencia Tiempo (caso peor)	Eficiencia Espacio (caso peor)
Primero en Anchura	Sí	Si coste ≈ profundidad	$O(r^p)$	$O(r^p)$
Coste uniforme	Si no hay caminos ∞ de coste finito	Si coste operadores >=0	$O(r^p)$	$O(r^p)$
Primero en profundidad	No	No	$O(r^m)$	O(r*m)
Profundidad limitada	Si L >= p	No	$O(r^L)$	O(r*L)
Profundización iterativa	Sí	Si coste ≈ profundidad	$O(r^p)$	O(r*p)
Bidireccional	Sí (anchura)	Si coste ≈ profundidad	$O(r^{p/2})$	$O(r^{p/2})$

r = factor de ramificación
m = máxima profundidad del árbol

p = profundidad mínima solución
L = límite de profundidad

Motivación

- El espacio de búsqueda (o **árbol** de búsqueda) solo contiene los nodos generados según la estrategia de búsqueda
- Los métodos **no informados** tienen **un coste temporal** que es una función exponencial del tamaño de la entrada: número de operadores (r),.. profundidad de la solución(p)..

  → Muy ineficientes en la mayoría de los casos

  El tiempo para encontrar la mejor solución a un problema no es asumible en problemas reales

Ante la explosión combinatoria, la fuerza bruta es *impracticable* → Búsqueda heurística

Coste exponencial r^p

- factor de ramificación efectivo: 10
- tiempo: 1000 nodos/segundo
- memoria: 100 bytes/nodo

p	nodos	tiempo	memoria
0	1	1 ms	100 Bytes
2	111	100 ms	11 KB
4	11.111	11 s	1 MB
6	10^6	18 min	111 MB
8	10^8	31 horas	11 GB
10	10^{10}	128 días	1 TB
12	10^{12}	35 años	111 TB
14	10^{14}	3500 años	11.111 TB

Métodos informados o heurísticos

- Algoritmo que orienta la búsqueda aplicando conocimiento del dominio sobre la proximidad de cada estado a un estado objetivo, guiando así la búsqueda hacia el camino más “prometedor”
- Evitan la explosión combinatoria **podando** el árbol de búsqueda
  - No expanden nodos no prometedores y así mejoran rendimiento
  - Pueden encontrar una solución aceptablemente buena en tiempo razonable

Heurísticas

- La heurística mejora la **eficiencia** de la búsqueda aunque puede sacrificar la **completitud** y la **optimalidad**
  - Hay posibilidad de fallar (no encontrar solución, habiéndola) o de encontrar soluciones no óptimas
- Las heurísticas permiten utilizar conocimiento específico del dominio del problema más allá de la definición del problema en sí mismo
- Las heurísticas proporcionan recomendaciones sobre la elección del sucesor más prometedor de un estado orientando acerca del orden de los sucesores de un estado
  - ➢Tiene efectos locales en cada paso del algoritmo
  - ➢**Ya no es un orden fijo**

Método de búsqueda heurística

- La clave está en definir una función heurística adecuada
- El uso de heurísticas tiene dos efectos en la búsqueda:
  - Ahorra esfuerzo de búsqueda
  - Coste de calcular la heurística en cada nodo
  - ahorro > coste

- En IA se emplea la **búsqueda heurística:**
  - Cuando para un problema no existe una solución exacta/óptima debido a **ambigüedades** inherentes al problema o a los datos disponibles
    - Ej: diagnóstico médico
  - Cuando un problema tiene una solución exacta pero el **coste computacional** de encontrarla es prohibitivo
    - Ej: ajedrez, problema del agente viajero…

Función heurística (I)

- Una función heurística **asocia a cada estado del espacio de estados un valor numérico** que evalúa lo prometedor que es ese estado para alcanzar un estado objetivo
- Se selecciona el estado con mejor valor heurístico
- Dos posibles interpretaciones:
  - Estimación de la "calidad" de un estado → Los estados de mayor valor heurístico son los preferidos
  - **Estimación de la distancia de un estado a un estado objetivo → Los estados de menor valor heurístico son los preferidos**
- Habitualmente asumiremos la 2ª interpretación → el mejor es el menor
  - Los **objetivos tienen un valor heurístico 0**

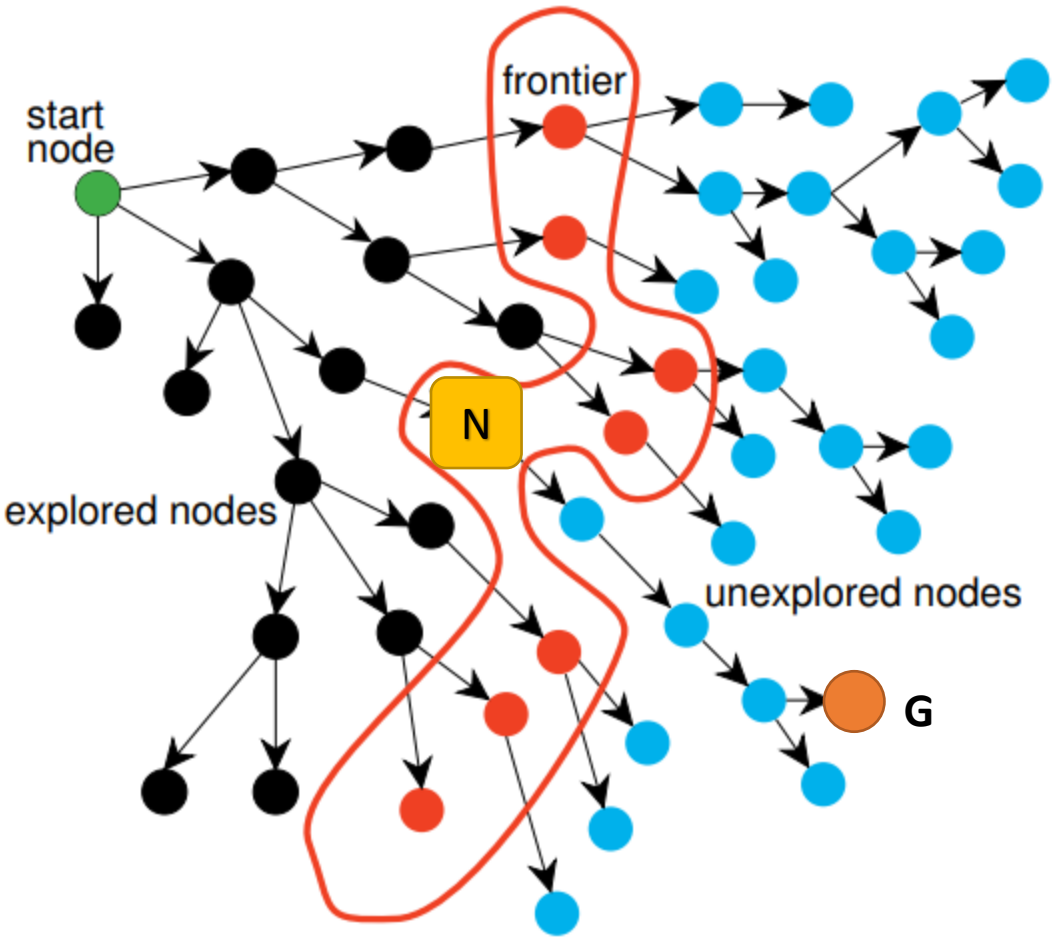
Función heurística

- h' es **admisible** si h'(n) ≤ h(n) para todo n, donde
  - **h(n) representa el coste real** para ir desde **n** a un nodo objetivo por el camino de **menor coste**
  - h' **no sobrestima nunca** el coste real para alcanzar el objetivo
  - Las **heurísticas admisibles son optimistas** porque estiman que el coste de solucionar el problema es menor que el que realmente es.
- Una función heurística h_1 es **más informada** que h_2 si ambas son admisibles y además **para cualquier nodo** n h(n) ***>= $h_1(n)>=h_2(n)$***
- Se dice que una función heurística es **consistente** sí solo sí para todos los n_i y n_j (siendo n_j descendiente de n_i) se cumple:

$$h'(n_i) - h'(n_j) <= coste(n_i, n_j)$$

$coste(n_i, n_j)$: coste para ir de n_i a n_j

Árbol de búsqueda

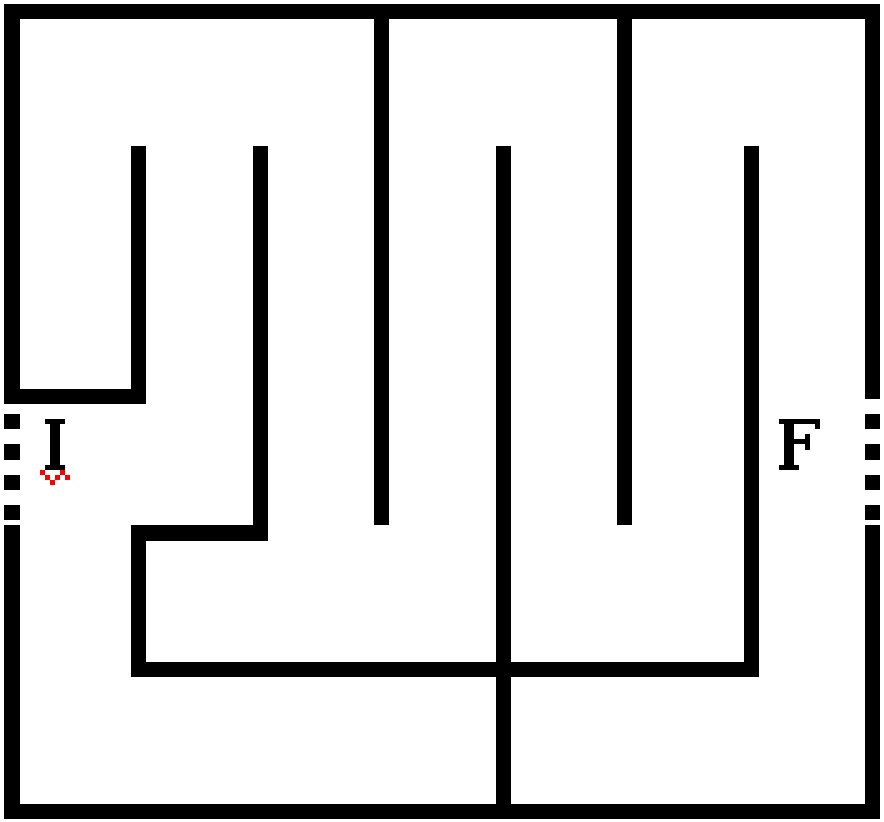


h'(N) ≤ **h(N)**

El coste real para ir de N a G no lo se todavía lo estimo (sin pasarme)

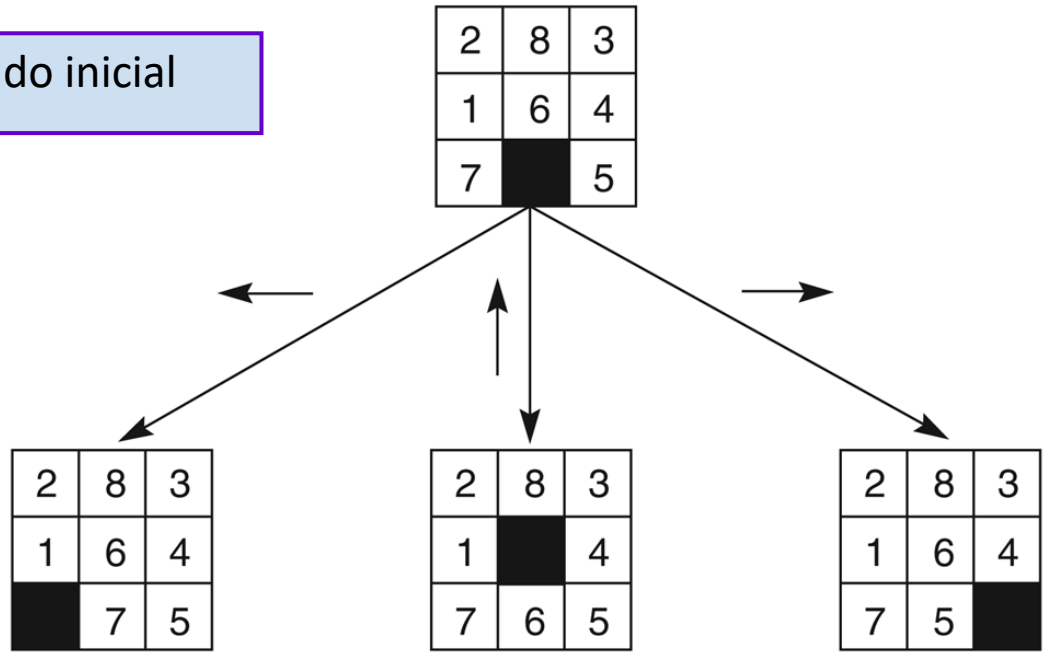
Ejemplo: búsqueda heurística en mapas

- ¿Qué heurística usar?
  - Admisible: 0 en el estado objetivo y menor que el coste real

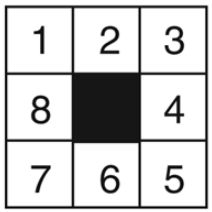


Función heurística. Ejemplo 8-puzzle

Estado inicial



Estado objetivo



Función heurística. Ejemplo 8-puzzle

1	2	3
8		4
7	6	5

Objetivo

h'_a= suma de las distancia de las fichas a sus posiciones en el tablero objetivo

h'_a = 1+1+0+0+0+1+1+2 = 6

h'_b = 5

h'_b = nº de fichas descolocadas (respecto al tablero objetivo)

Problema misioneros - ESTADO

- Formalmente, un estado será representado por una terna (M_i, C_i, B, M_f, C_f) en la que:
  - $B \in [i, f]$ indica la **posición de la barca**, por lo que toma el valor i si está en el extremo inicial, o f si está en el final (d)
  - $Mi, Ci, Mf, Cf \in$ **[0,..., 3]** indican el **número de misioneros y caníbales** que quedan en el extremo inicial y final del río, respectivamente
- De esta manera, el **estado inicial** se representa como **(3, 3, i, 0, 0)** y el **estado final** como **(0, 0, f, 3, 3)**

Heurísticas para los misioneros

- Pensar simplificaciones del problema

La barca debe estar en la orilla

Asumir que hay **infinitas barcas** tanto en un lado como en otro.
Este problema simplificado tiene una solución muy sencilla, que es asumir que siempre viajan un caníbal y un misionero juntos, con la acción Mover1M1C (no hay que volver).

Por tanto, la heurística resultante de este problema simplificado es:

$$h_1(n) = \frac{C_i + M_i}{2}$$

Si hay más caníbales se comen a los misioneros

Problema **relajado en el que los caníbales nunca se comen a los misioneros**.
La solución es que en cada viaje de ida y vuelta (2*) podemos transportar a una persona (dado que la otra tendrá que volver para llevar la barca).
Por tanto, la heurística resultante es:

$$h_2(n) = 2(C_i + M_i) - orilla(n)$$

donde orilla(n) = 1 si en el estado n la barca está en la orilla i (B = i), y orilla(n) = 0 si la barca está en la orilla final (B = f).

Misioneros y caníbales (heurísticas)

- Las dos heurísticas son admisibles (resultado de relajar el problema original)
- Para decidir qué heurística elegir, h1 o h2, estudiamos **cuál es la más informada**, puesto que será la que, siendo admisible, tendrá un valor más cercano a h^*.
- Se observa fácilmente que la más informada es h_2, puesto que **$h_1(n) <= h_2(n)$**, sea cual sea el valor de C_i y de M_i para el estado n. Por tanto, elegimos $h_2(n)$.

$$h_1(n) = \frac{C_i + M_i}{2}$$

$$h_2(n) = 2(C_i + M_i) - orilla(n)$$

- https://www.youtube.com/watch?v=UFcEmJjKP0Y

Finding Good Heuristics

- How can we find good heuristics for a given problem?
- Can this process be automated?

Búsqueda primero el mejor

- Primero el nodo más prometedor
- Selección del siguiente nodo a expandir en base a una función de evaluación f '(n)
  - Estimación del coste h’(n) necesario para llegar a una solución a partir de ese nodo n
  - Heurística en lugar de criterios fijos (1º profundidad o anchura) → El nodo seleccionado será aquel que minimice la función de evaluación
- Para gestionar los abiertos se usa **una cola de prioridad, ordenada por el valor de la función de evaluación heurística f '(n)**

Búsqueda primero el mejor. Función de evaluación

La función de evaluación (f ‘) puede ser de dos tipos:

1. Función que considera exclusivamente lo que “falta” → el coste mínimo estimado para llegar a una solución a partir del nodo n

   f '(n) = h'(n) => **Búsqueda voraz**

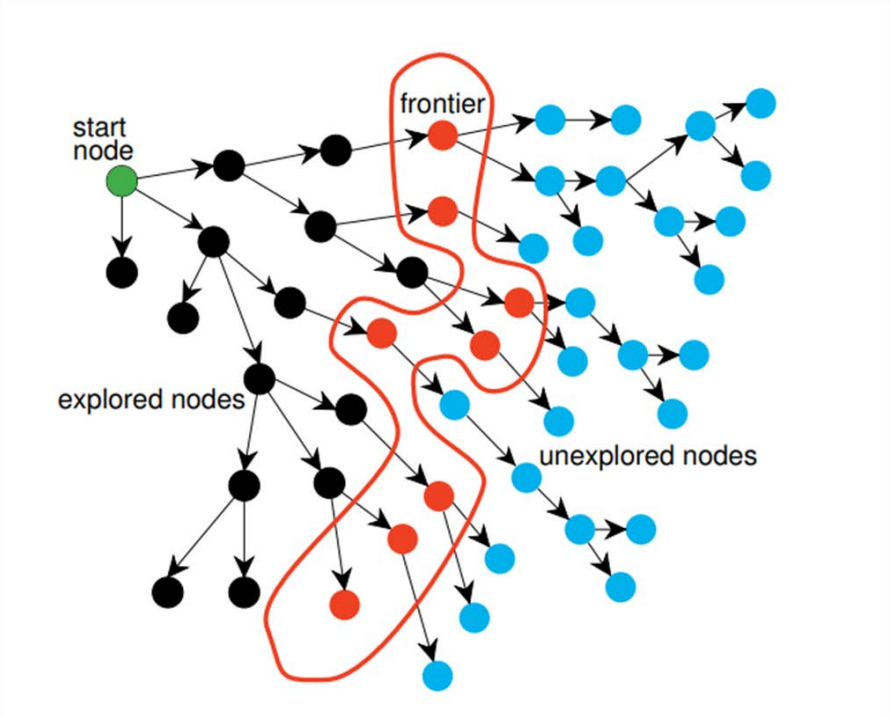
2. Función que considere el coste total estimado del camino desde el nodo inicial a un nodo objetivo pasando por el nodo n. La función **f '** está formada por dos componentes:
   - g(n) = coste del camino desde el nodo inicial hasta n
     - ➢ No es una estimación, sino un coste real
   - h'(n) = coste mínimo estimado para llegar a un nodo objetivo desde n

   f '(n) = g(n) + h'(n) => **Búsqueda A***

Búsqueda voraz

- **f '(n) = h'(n)** → Coste mínimo estimado para llegar desde n a un nodo objetivo (por el camino más barato)
  - ➢h'(n) = 0 para los nodos objetivo

- En cada paso trata de ponerse lo más cerca posible del objetivo



Búsqueda voraz. Propiedades

- Depende del problema y de la calidad de la heurística

- **¿Es completa?**
- **No es óptima** → ignora el coste del camino completo
- **Caso peor:** la heurística es muy mala
  - Tiempo: $O(r^m)$, siendo m la profundidad máxima del espacio de búsqueda
  - Espacio: $O(r^m)$, mantiene todos los nodos que genera
- Si la heurística es buena la complejidad se reduce

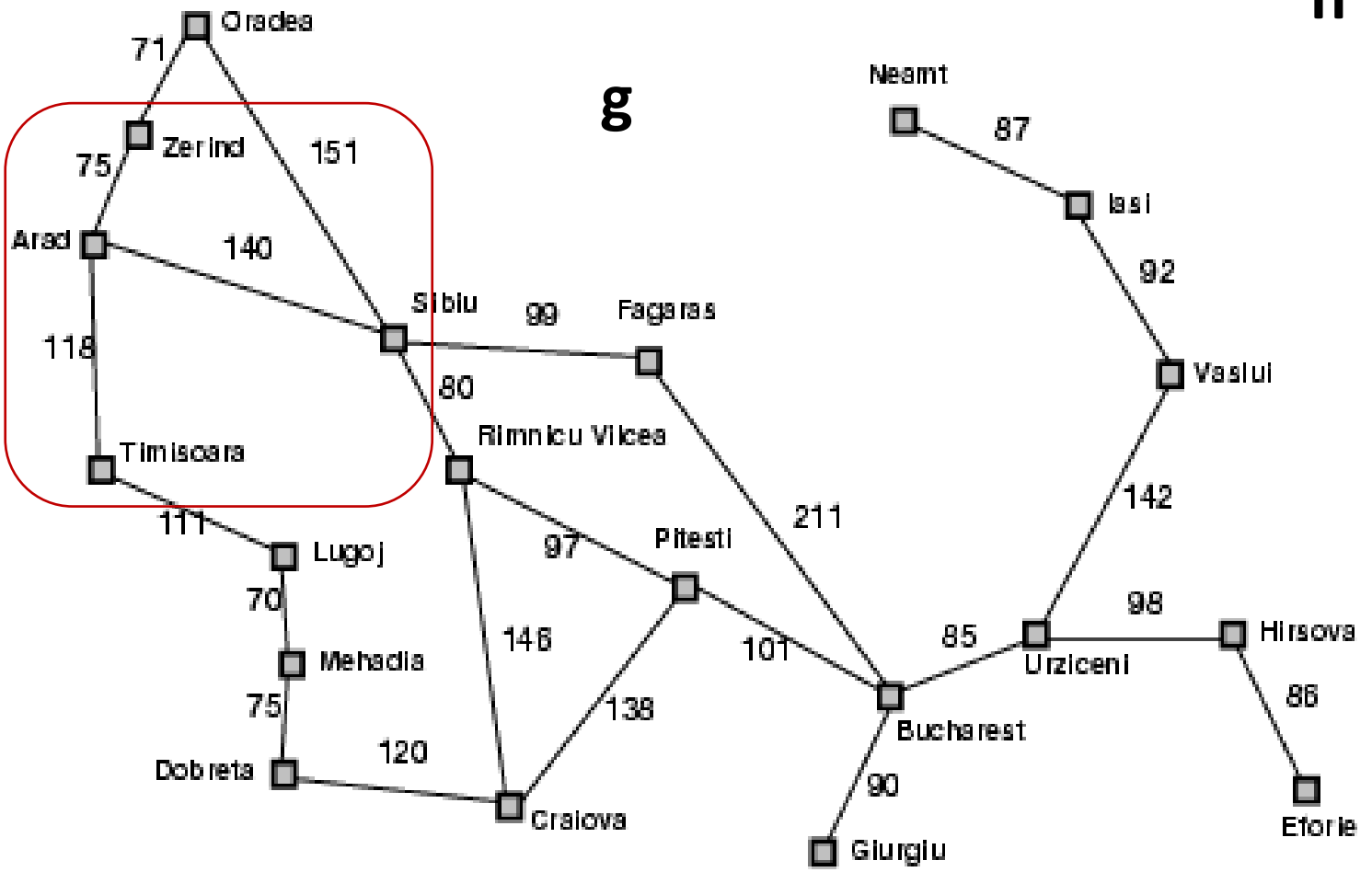
Búsqueda voraz. Propiedades

- Depende del problema y de la calidad de la heurística

- **Sólo es completa con h' adecuada**
  - **No es completa en general si hay ciclos puede entrar en rama infinita y no terminar (ver ejemplo)**
- **No es óptima** → ignora el coste del camino completo
- **Caso peor:** la heurística es muy mala
  - Tiempo: $O(r^m)$, siendo m la profundidad máxima del espacio de búsqueda
  - Espacio: $O(r^m)$, mantiene todos los nodos que genera
- Si la heurística es buena la complejidad se reduce

Greedy Best-First Search

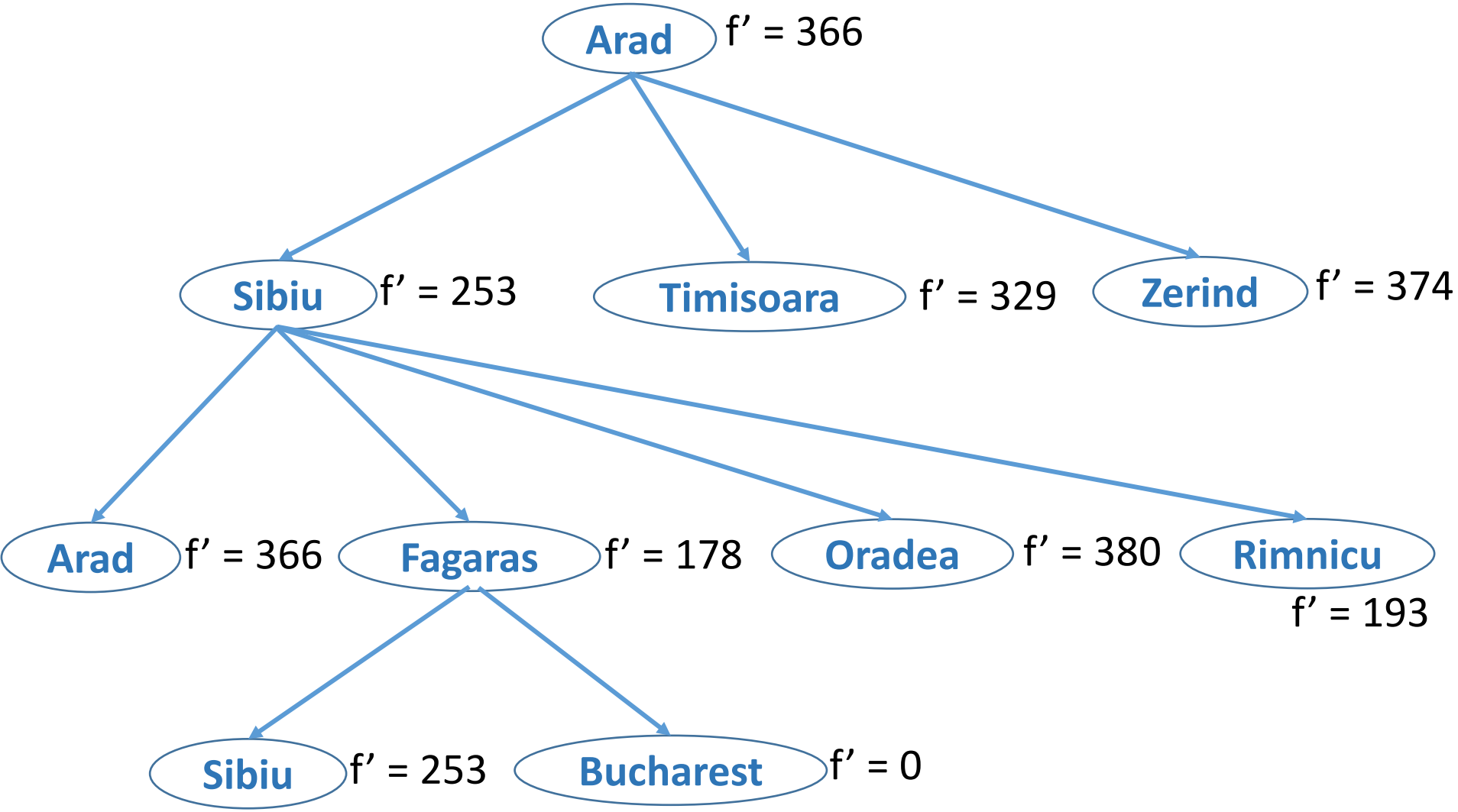
Ejemplo Mapa de Rumanía.



h'

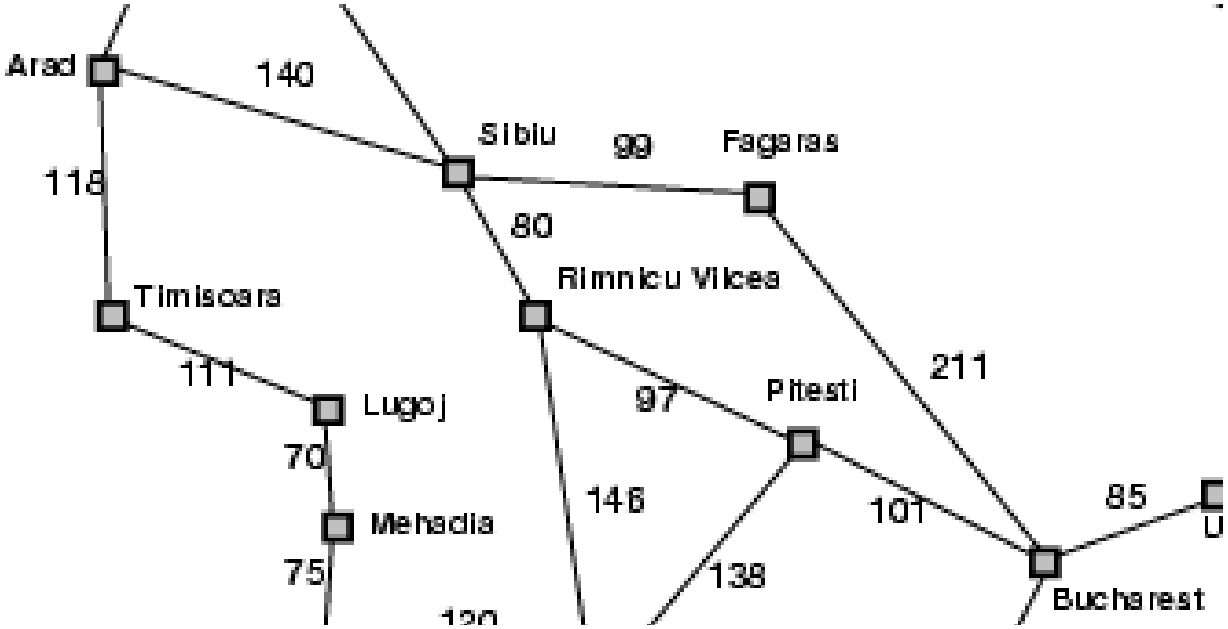
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Ejemplo Mapa de Rumanía. Búsqueda voraz.



Ejemplo Mapa de Rumanía. Búsqueda voraz.

- Solución encontrada: Arad – Sibiu – Fagaras – Bucharest
  - Coste: 140 + 99 + 211 = **450**
- Solución óptima: Arad – Sibiu – Rimnicu – Pitesti – Bucharest
  - Coste: 140 + 80 + 97 + 101 = 418



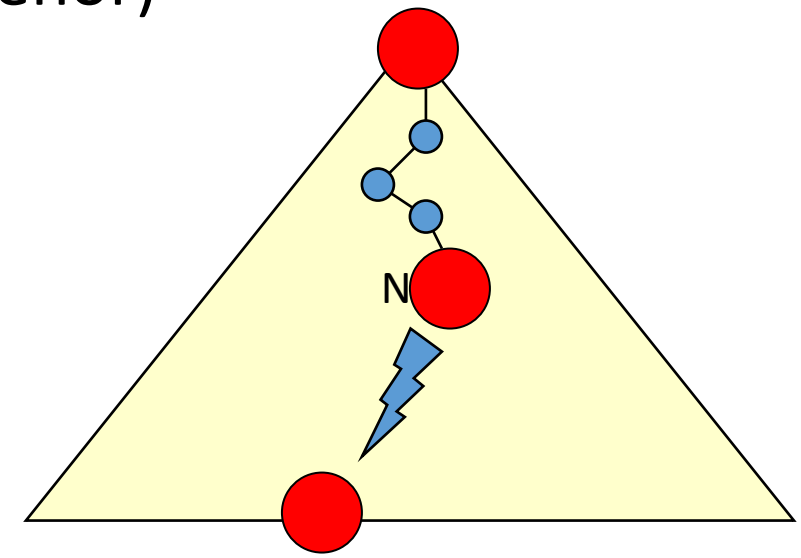
Búsqueda A* (I)

- **f '(n) = g(n) + h'(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

- La componente g de f ' le da el toque **realista**
- La componente h' le da el toque **optimista**

Función de evaluación

- Se anota en un nodo
- Es el valor que se utiliza para ordenar los nodos pendientes de explorar (primero el de coste previsto menor)



Coste del mejor camino hasta el momento: g(N)

+

Estimación (heurística) de lo que falta: h'(N)

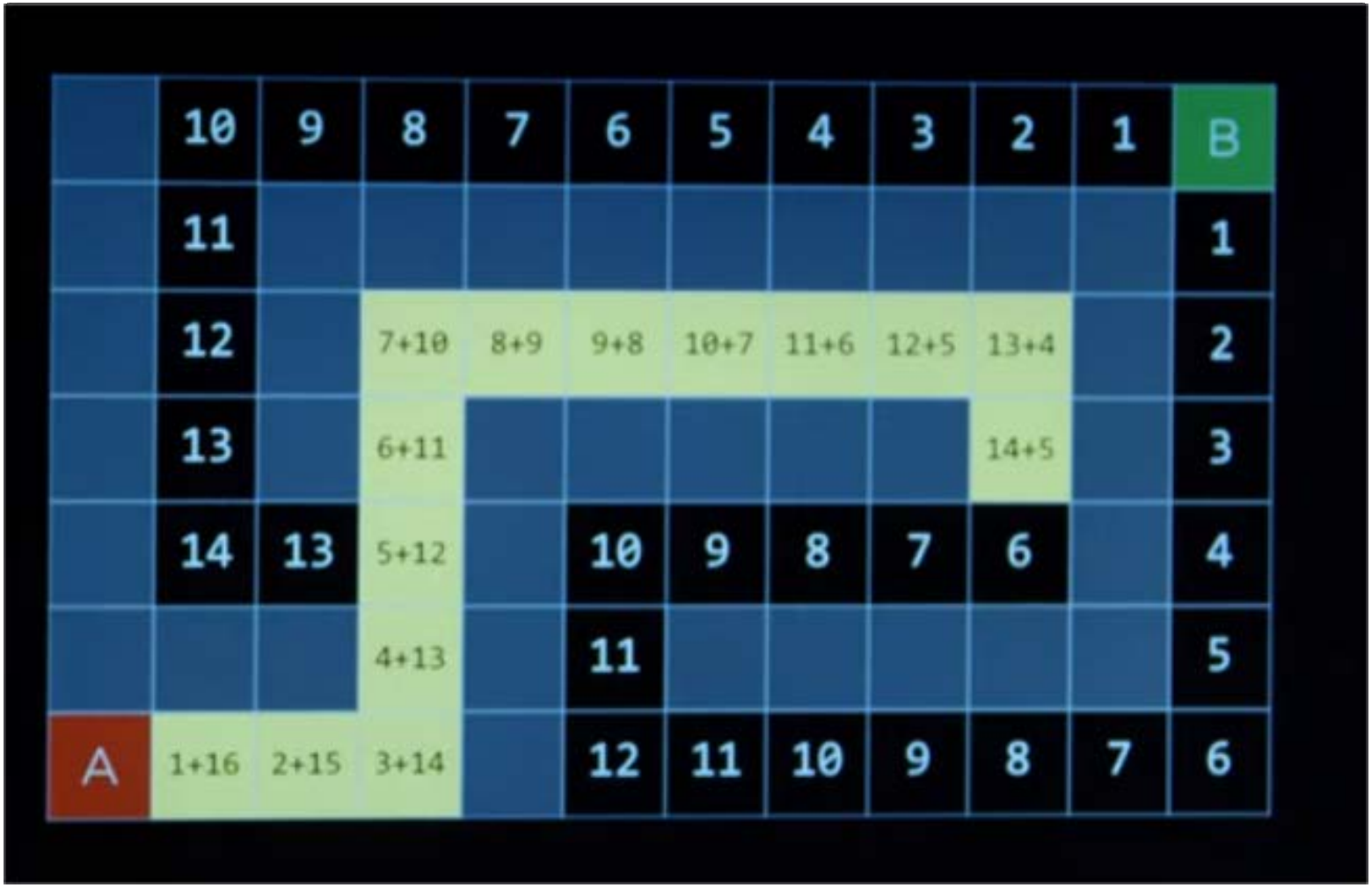
$$f'(N) = g(N) + h'(N)$$

Representa una estimación del coste total del mejor camino desde el estado inicial al objetivo que pasa por ese nodo

Búsqueda A* (I)

- Combina primero en anchura con primero en profundidad
  - 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

- **Si h' es admisible**, la búsqueda con f '(n) = g(n) + h'(n) se denomina A* o búsqueda óptima

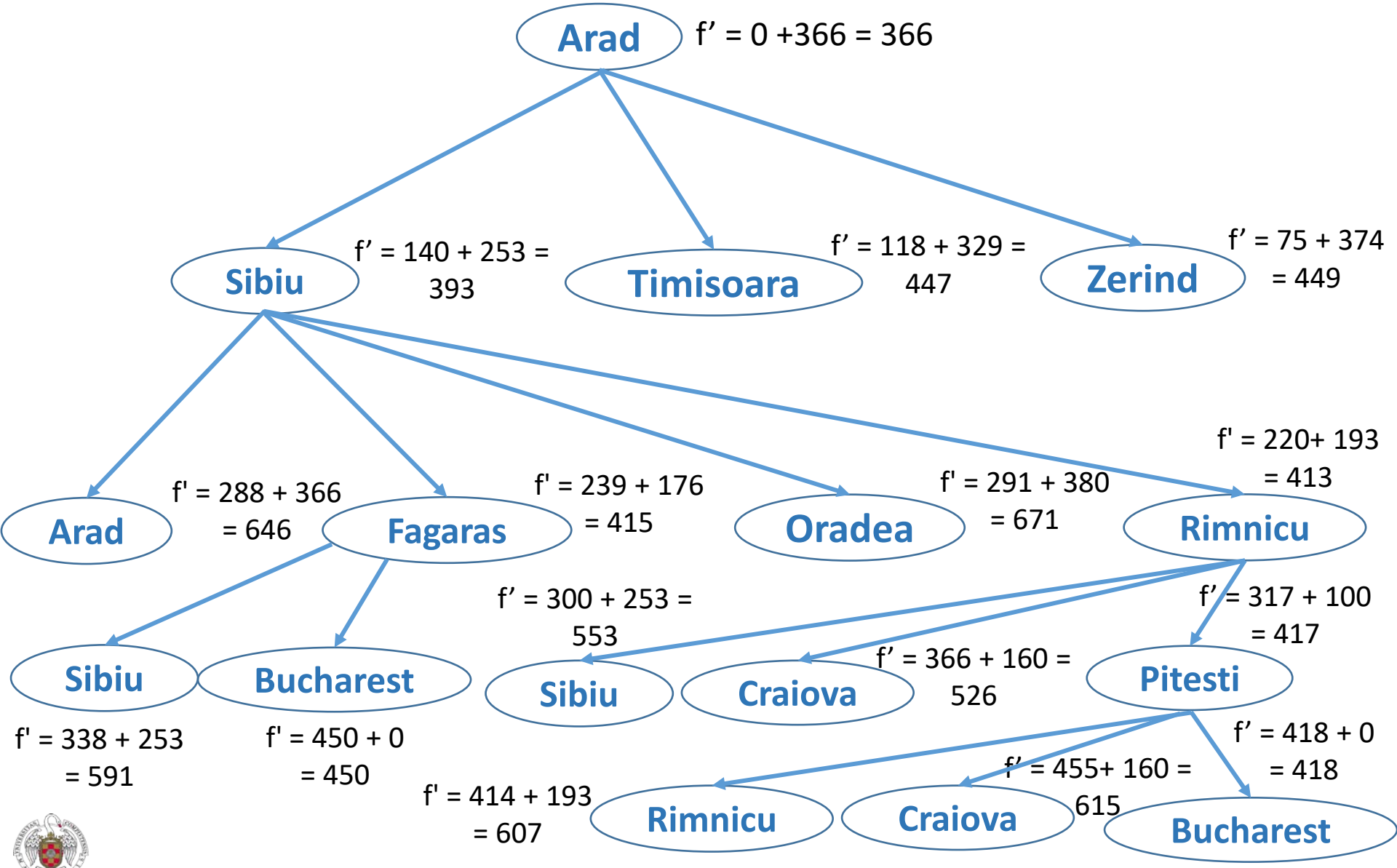

10 9 8 7 6 5 4 3 2 1 B
11 1
12 7+10 8+9 9+8 10+7 11+6 12+5 13+4 2
13 6+11 14+5 3
14 13 5+12 10 9 8 7 6 4
4+13 11 5
A 1+16 2+15 3+14 12 11 10 9 8 7 6

Intuiciones sobre cómo funciona A*

- Mantiene **el mejor camino** a todos los nodos procesados
- Para cada nodo
  - Exploro sus hijos
    - El orden de exploración viene dado por el ***coste total previsto*** (heurística + coste acumulado)
    - Primero exploro el hijo de coste total previsto menor
  - Si el hijo es la primera vez que aparece
    - Calculo el ***coste total previsto*** y guardo el camino (añado el nodo a la lista de abiertos)
  - Si el hijo ya había aparecido antes y encuentro un camino **mejor** que el que tenía guardado
    - Guardo el mejor y deshecho el otro

Control de repetidos si es Graph Search

Ejemplo Mapa de Rumanía. A* TreeSearch



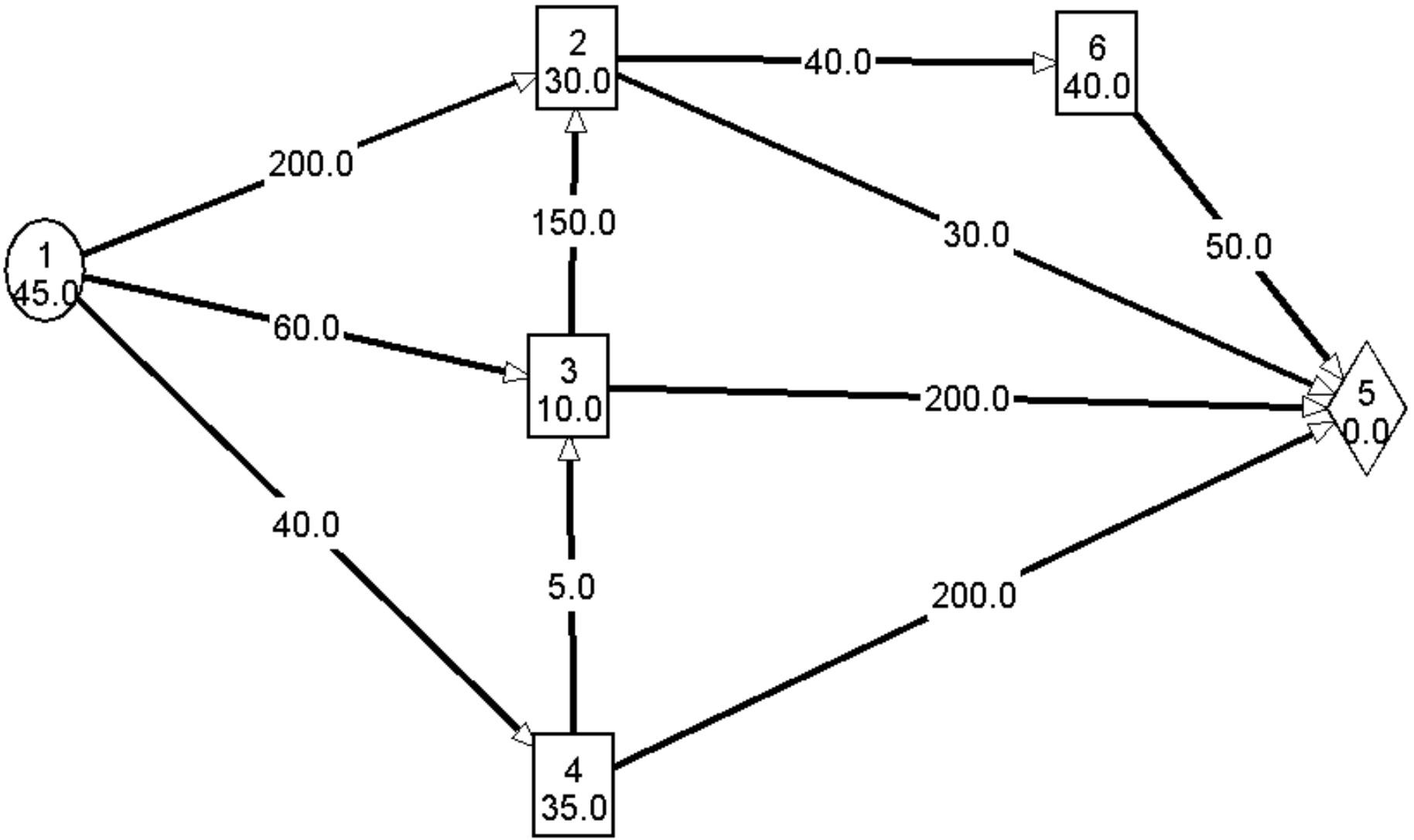
Ejemplo Mapa de Rumanía. A* TreeSearch

- Solución encontrada:
  Arad – Sibiu – Rimnicu – Pitesti – Bucharest
  - ➢Coste: 140 + 80 + 97 + 101 = 418

- Solución óptima:
  Arad – Sibiu – Rimnicu – Pitesti – Bucharest
  - ➢Coste: 140 + 80 + 97 + 101 = 418

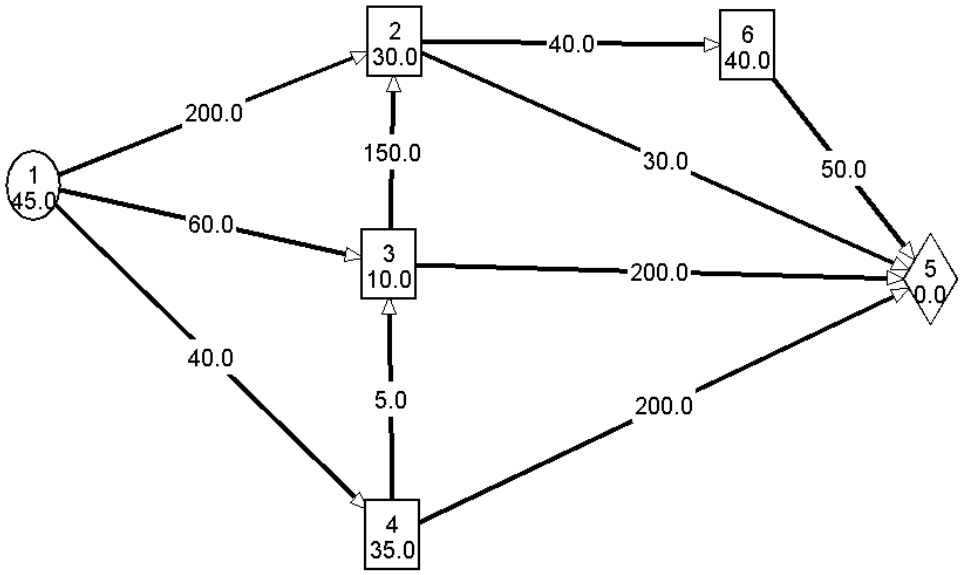
- ¿Siempre se consigue la óptima? Sí porque h' es admisible.
  - La h' cumple la propiedad de **consistencia → afecta al control del repetidos.**

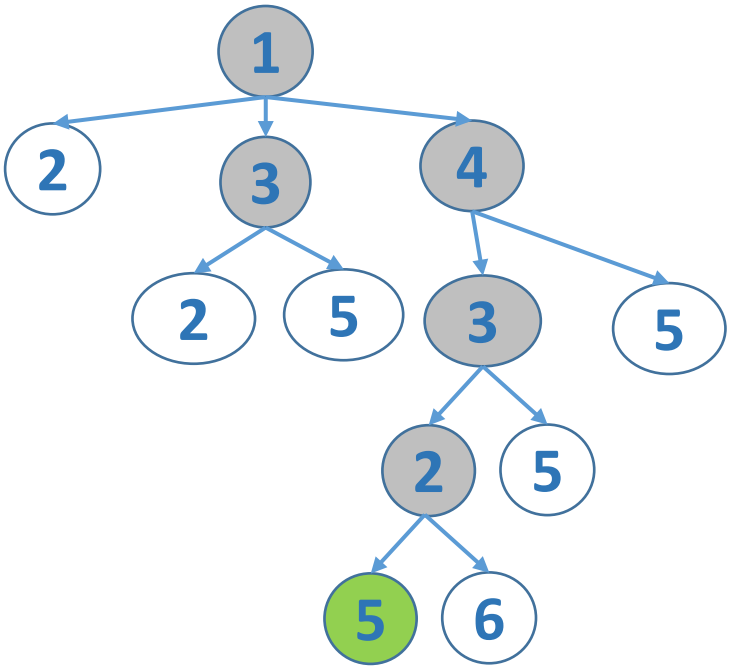
Búsqueda A*. Ejemplo



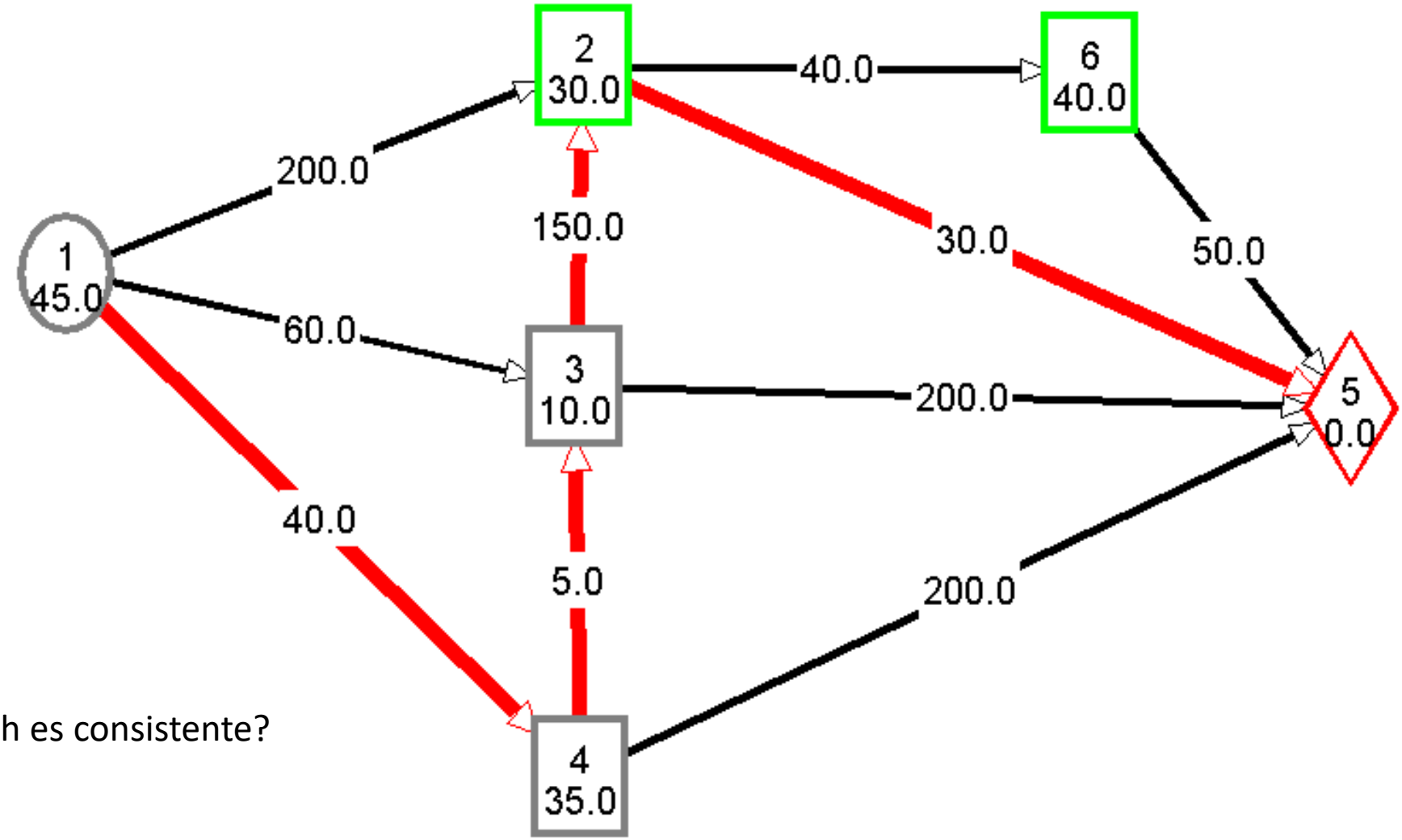
Búsqueda A*. Ejemplo. Solución con TreeSearch

camino	1	1,2	1,3	1,4	1,3,2	1,3,5	1,4,3	1,4,5	1,4,3,2	1,4,3,5	1,4,3,2,5	1,4,3,2,6
padre		1	1	1	3	3	4	4	3	3	2	2
estado	1	2	3	4	2	5	3	5	2	5	5	6
g	0	200	60	40	210	260	45	240	195	245	225	235
h'	45	30	10	35	30	0	10	0	30	0	0	40
f'	45	230	70	75	240	260	55	240	225	245	225	275





Búsqueda A*. Ejemplo. Solución con TreeSearch

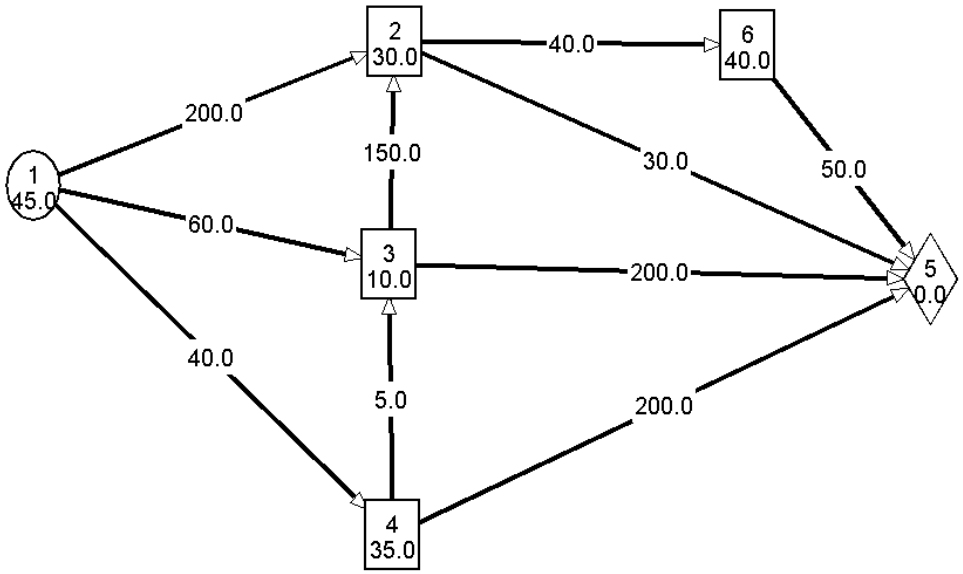


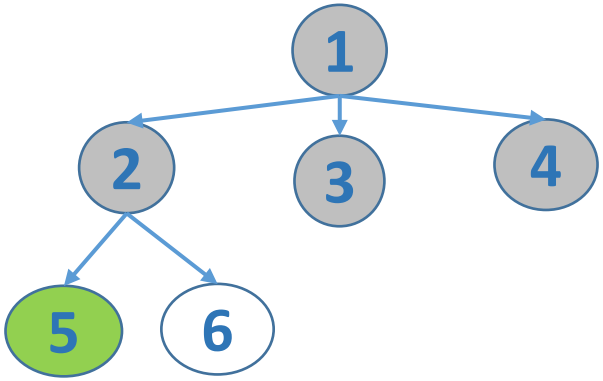
¿la h es consistente?

Solución : 1-4-3-2-5 Coste: 40+5+150+30 = 225

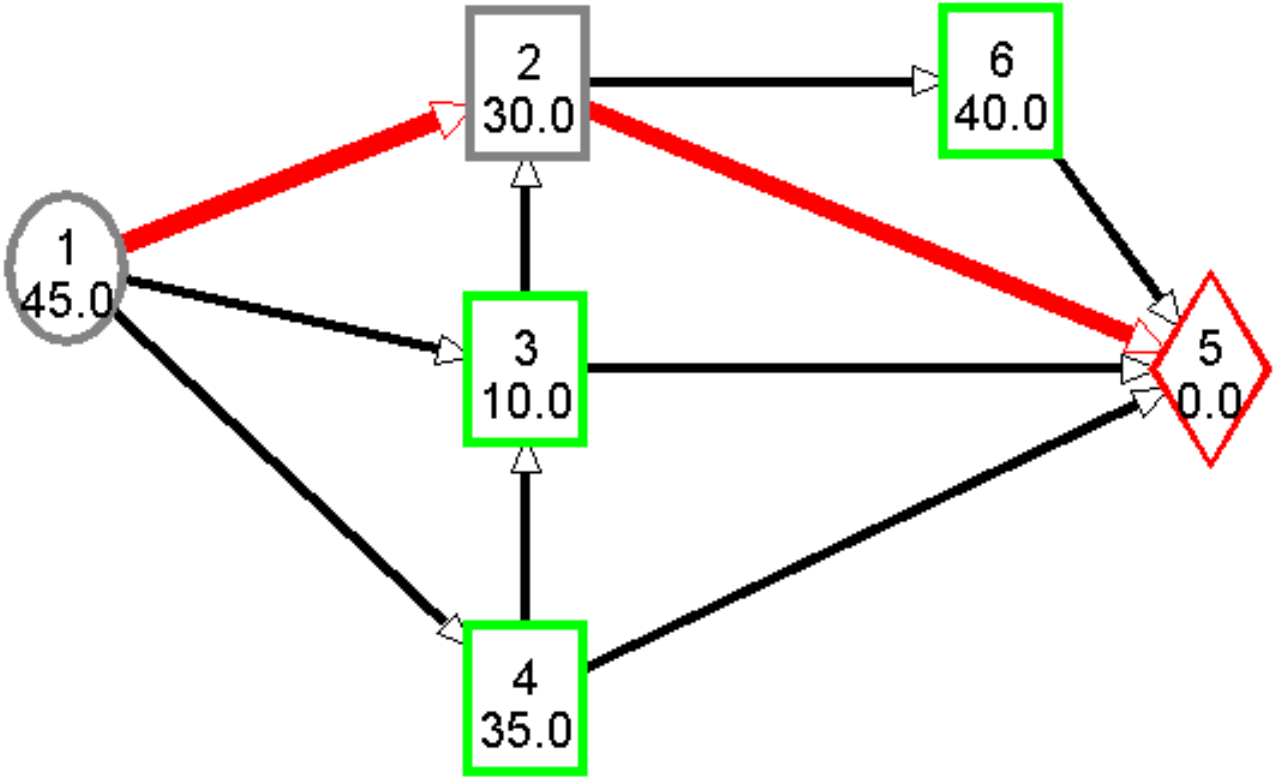
Búsqueda A*. Ejemplo. Solución con GraphSearch

camino	1	1,2	1,3	1,4	**1,2,5**	1,2,6		
padre		1	1	1	**2**	2		
estado	1	2	3	4	**5**	6		
g	0	200	60	40	**230**	240		
h'	45	30	10	35	**0**	40		
f'	45	230	70	75	**230**	280		



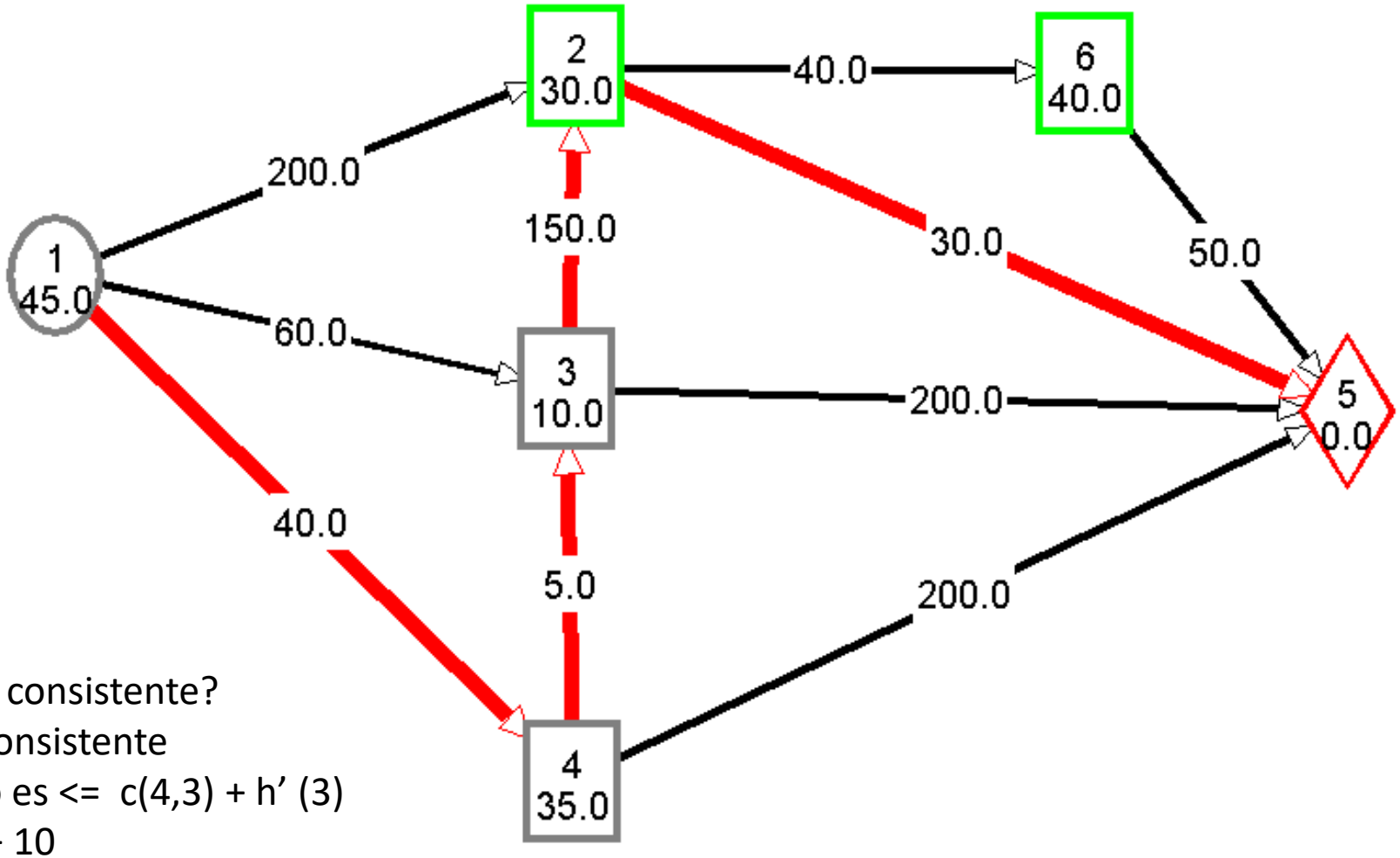


Búsqueda A*. Ejemplo. Solución con GraphSearch



Solución : 1-2-5 Coste: 200+30 = 230, no óptimo

¿Por qué no encuentra el óptimo?



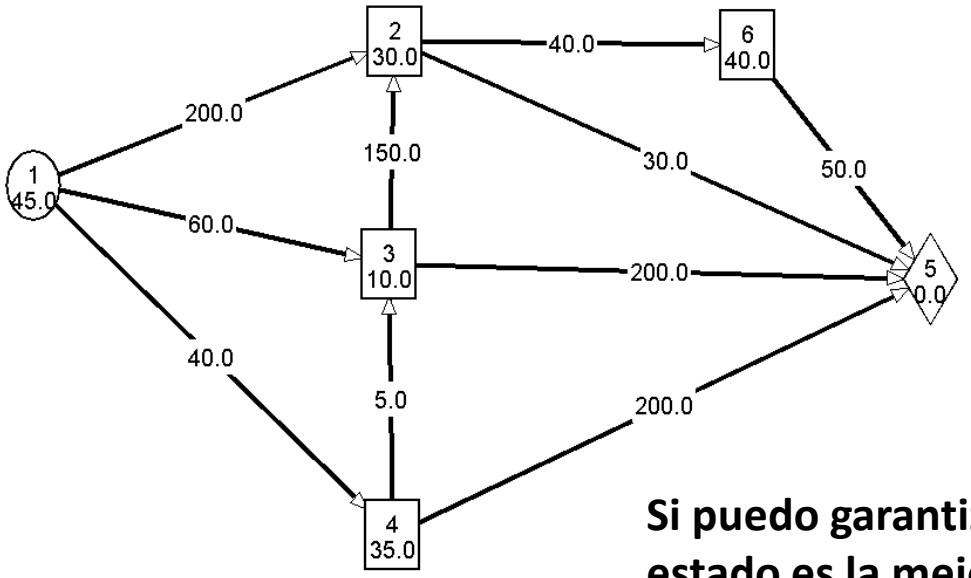
¿la h es consistente?
No es consistente
h'(4) no es <= c(4,3) + h' (3)
35 > 5 + 10
Como precisamente el camino optimo pasa por allí por eso no lo encuentra.

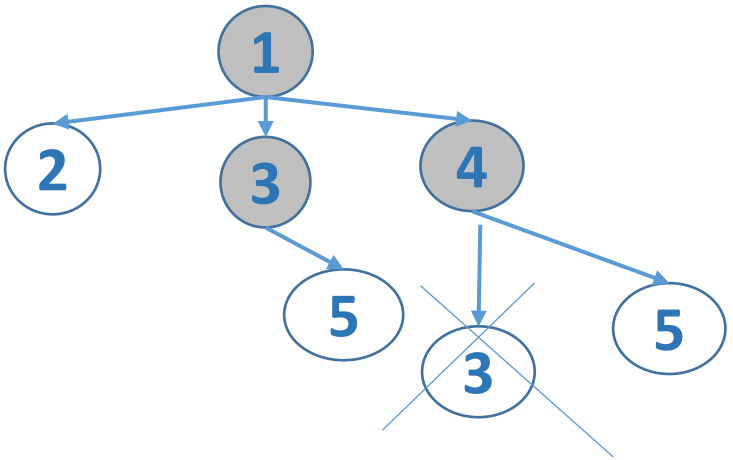
Solución óptima : 1-4-3-2-5 Coste: 40+5+150+30 = 225

En este paso....

camino	1	1,2	1,3	1,4	1,3,5		1,4,5		
padre		1	1	1	3		4		
estado	1	2	3	4	5		5		
g	0	200	60	40	260		240		
h'	45	30	10	35	0		0		
f'	45	230	70	75	260		240		

El nodo (4->3) **NO** entra en abiertos porque el estado 3 ya está expandido en el nodo (1->3)





Si puedo garantizar que la primera vez que aparece un estado es la mejor → no hace falta control de repetidos en cerrados

Búsqueda A*. Optimalidad (I)

- 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) <= c(n, n') + h'(n') (desigualdad triangular)
  - h' ha de ser localmente consistente con el coste de los arcos
  - **Toda heurística consistente también es admisible (al revés no)**
  - 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')
- 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
  - ➢ 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

Búsqueda A*. Optimalidad (II)

- A* con TreeSearch → Garantiza una solución óptima si la heurística es **admisible**

  Ineficiente si el espacio de estados tiene muchos ciclos

- A* con GraphSearch → Garantiza una solución óptima si la heurística es **consistente**
  - Si la heurística es consistente no es posible que nos encontremos el mismo nodo por un camino alternativo con un coste menor → esto hace que el tratamiento de nodos **cerrados duplicados** sea innecesario

- Para **h' no consistente** si queremos mantener la optimalidad de A* → tratamiento de nodos cerrados duplicados (MUY COSTOSO).

Búsqueda A*. Propiedades

Si la **heurística es consistente**:

- Si existe solución, tendrá que llegar a un nodo objetivo, salvo que haya una sucesión infinita de nodos *n* en los que se cumpla $f'(n) <= f(n)$. Esto puede ocurrir si
  - Hay nodos con factor de ramificación infinito, ó
  - Si hay caminos de coste finito con un número infinito de nodos
- ***A* es completo*** si el factor de ramificación ***r*** es finito y existe una constante $\varepsilon > 0$ tal que el coste de cualquier operador es siempre $>= \varepsilon$
- **A * 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 $f'(n) < f(n)$ corre el riesgo de omitir la solución óptima

- Modificaciones a A* si la h’ no es admisible
  - Hay que entender claramente qué efecto produce que no sea admisible → pierde optimalidad
  - Encontrar una solución subóptima no es un problema grave
  - Si queremos mantener la optimalidad
    - Expandidos → tabla de dispersión (diccionario)
    - Si no está → lo expando pero si está tengo que actualizar el nodo y quedarme con el mejor
    - control de repetidos en abiertos y cerrados

Búsqueda A*. Complejidad

- 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 la complejidad (tiempo y espacio) sigue siendo exponencial $O(r^p)$
- El crecimiento exponencial no ocurre si **el error en la heurística** no crece más rápido que el logaritmo del coste real (h(n))

$$|h'(n) - h(n)| \leq O(\log h(n))$$

- En la práctica, para casi todas las heurísticas, el error es del orden del coste del camino O(h(n)) y no de su logaritmo
- El crecimiento exponencial desborda la capacidad de cualquier ordenador → exponencial en tiempo y en espacio
  - Necesita mantener todos los nodos generados en memoria
  - No adecuada para problemas grandes

Variantes optimizadas de A*

Búsqueda A*. Variantes (I)

- Se suelen usar variantes de A* que encuentran rápidamente **soluciones subóptimas**
- Algunas variantes de A*:
  - RTA* (Real Time A*): acota el tiempo de búsqueda
  - IDA* (Iterative Deepening A*): acota el coste
  - SMA* (Simplified Memory-bounded A*): acota el espacio

- Si no se puede calcular h' admisible 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

Búsqueda A*. Variantes (I)

- **RTA*: Real Time A***
  - Se aplica en tareas de tiempo real donde no se puede esperar a encontrar la solución óptima
  - Obliga a tomar una decisión cada periodo de tiempo
- **IDA*: A* con profundización iterativa (cota de coste)**
  - El objetivo es reducir las necesidades de memoria
  - Búsqueda por profundización iterativa controlada por la función de evaluación f' de A*
  - Límite de coste no de profundidad: se expande sólo estados con coste dentro de la cota establecida
  - El resultado de cada iteración se usa para establecer la cota de la siguiente iteración
- **SMA*: A* acotado por memoria**
  - Mientras hay memoria, funcionamiento normal
  - Si al generar un sucesor falta memoria, libera el espacio de los estados menos prometedores

Algoritmo IDA*

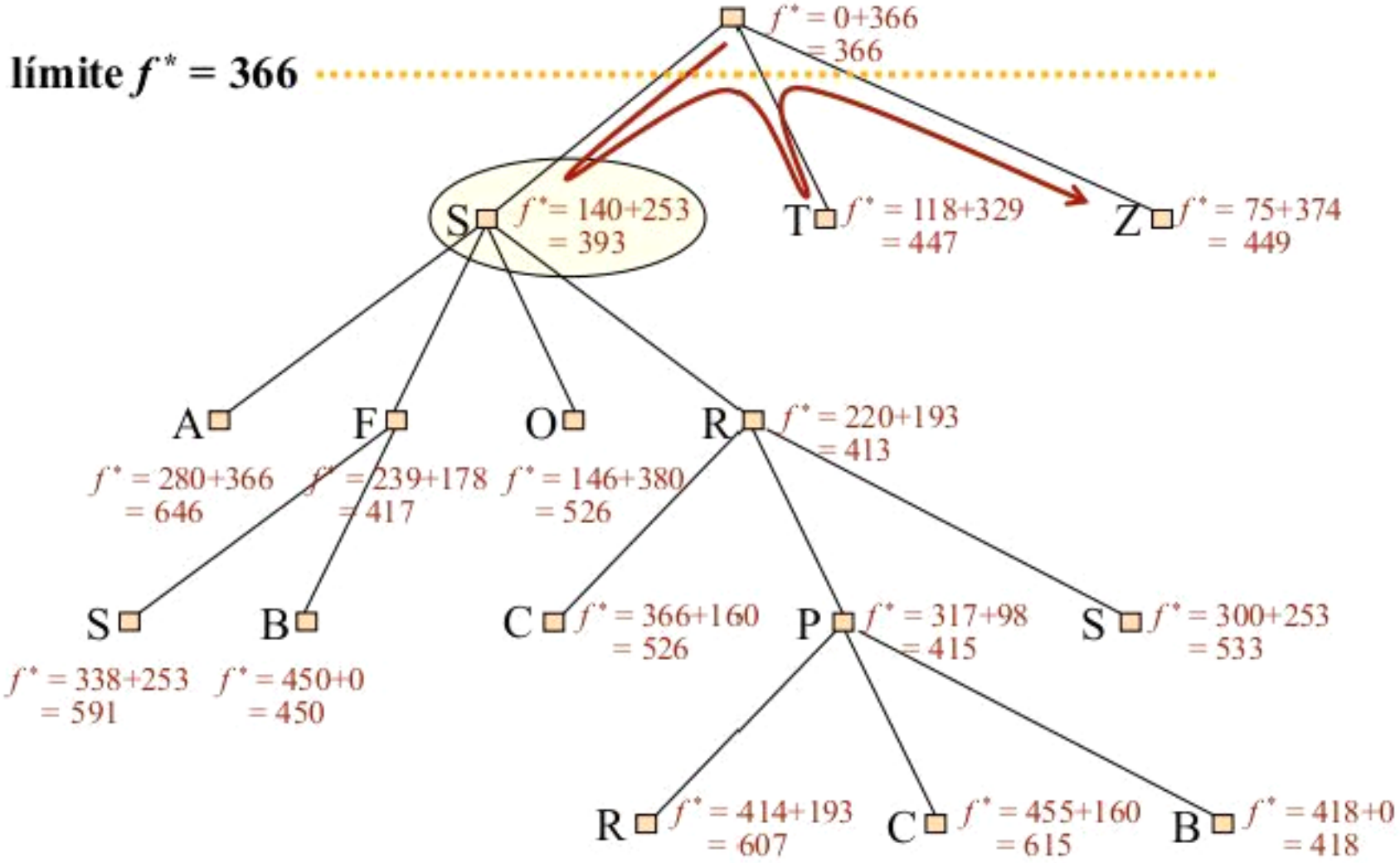
Algoritmo:

- un subprograma *bp-limite-f* que realiza búsqueda en profundidad hasta un límite f^* dado
- devuelve el siguiente f^* más bajo
- un subprograma IDA* que actualiza el límite f^* y detecta éxito/fallo

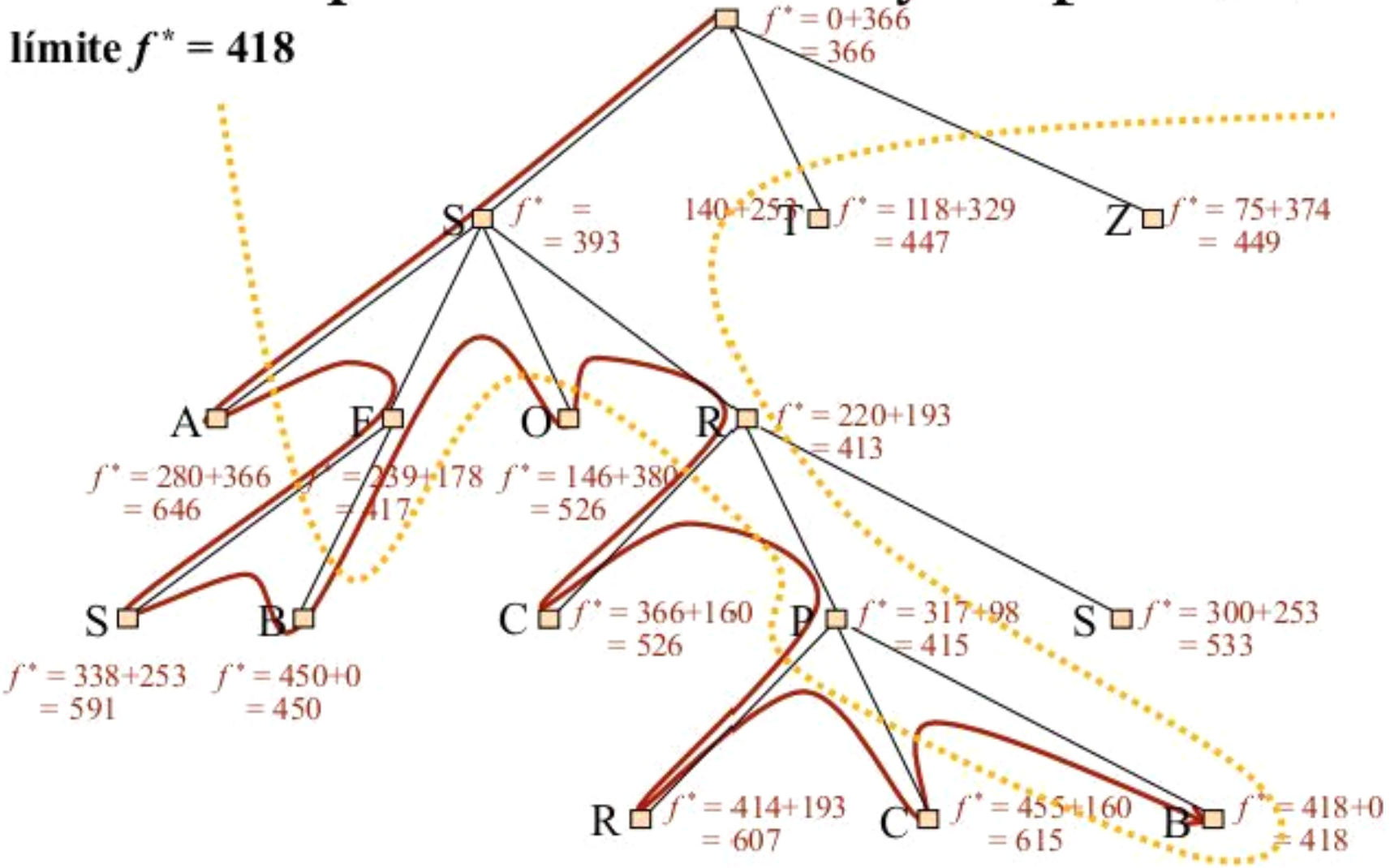
```
{IDA*}
limite-f ← f*(s0)
Repetir
    limite-f ← bp-limite-f(limite-f)
    Si éxito ent. devolver(solución)
    Si limite-f =∞ ent. devolver(fallo)
    Fin {repetir}
```

```
{bp-limite-f}
abierta ← s0
f-siguiente ← ∞
Repetir
    Si vacia?(abierta) entonces
        devolver(f-siguiente)        {fallo}
    nodo ← primero(abierta)
    Si meta?(nodo) entonces
        devolver(nodo)               {éxito}
    sucesores ← expandir(nodo)
    Para cada n∈ sucesores hacer
      Si f*(n )≤ límite-f entonces
        n.padre ← nodo
        ordInsertar(n,abierta,cabeza)
      Sino
        f-siguiente ← min(f-siguiente, f*(n ))
    Fin {repetir}
```

Búsqueda IDA*: Ejemplo (1)

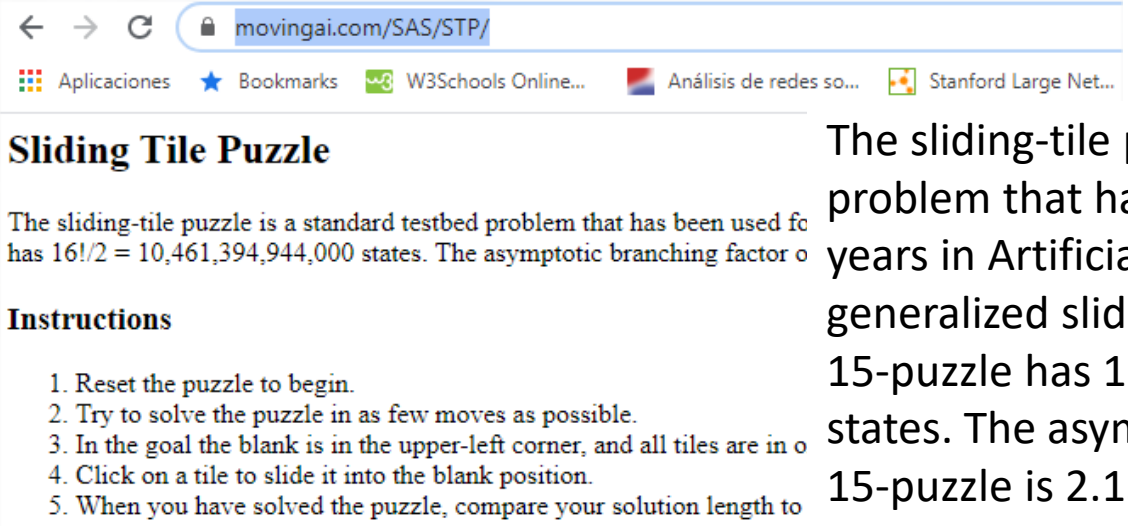


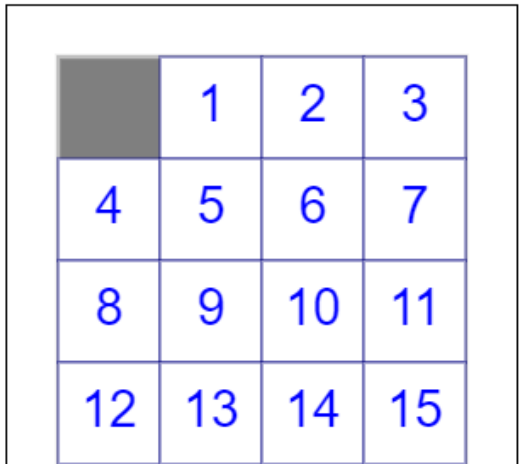
Búsqueda IDA*: Ejemplo (6)



Búsqueda Heurística moderna

https://www.movingai.com/AAAI-HS20/





The sliding-tile puzzle is a standard testbed problem that has been used for more than 50 years in Artificial Intelligence and search. The generalized sliding-tile puzzle is NP-hard. The 15-puzzle has 16!/2 = 10,461,394,944,000 states. The asymptotic branching factor of the 15-puzzle is 2.1304.

https://www.movingai.com/SAS/STP/

Single-Agent Search Demo Page

https://www.movingai.com/SAS/

This page lists demos that can be used for teaching and exploring algorithms in single-agent search.

- Domains
  - The Sliding Tile Puzzle
- Optimal Graph Search

Suboptimal Map Pathfinding Algorithms

This demo illustrates several different suboptimal path planning algorithms. These include:

- A*: f = g+h
  This is the classical best-first heuristic search algorithm that combines g+h evenly to find the shortest path
- Weighted A*: f = g+w*h
  This is a simple modification to weighted A* that finds w-optimal solutions. Weighted A* does not need to re-open states to find w-optimal solutions.
- Weighted A* (XDP): $f = (1/2w)[g+(2w-1)h+sqrt((g-h)^2+4wgh)]$
  XDP stands for convex downward parabola. This is a small modification to weighted A*. In Weighted A*, the set of states with the same priority are on a straight line. XDP changes this straight line to a parabola. XDP causes the best-first search to find near-optimal paths near the start and paths that are up to (2w-1) supoptimal near the goal. Overall the paths found are still w-optimal, and re-openings are not required.
- Weighted A* (XUP): $f = (1/2w)(g + h + sqrt((g+h)^2+4w(w-1)h^2))$
  XUP stands for convex upward parabola. XUP is similar to XDP, except it looks for (2w-1)-optimal paths near the start and optimal paths near the start.
- A* epsilon: focal list f = h; open list f = g + h
  A* epsilon introduced the idea of a focal list - the subset of states within the open that have a given property. A* epsilon interleaves greedy search to the goal with A* expansions to prove that the solution is w-optimal. (This implementation doesn't quite handle the focal list properly, but is a very similar approach.)
- Optimistic Search: optimistic: f = g + (2w-1)h; proof: f = g+h
  Optimistic search relies on the fact that WA* often finds paths that are better than w-suboptimal. It searches with a larger weight to find the solution and then tries to prove that the solution is still w-optimal. Optimistic search re-opens states with a shorter path is found. (Note that in the demo below you can choose the w-bound and exactly which bound to use for the optimistic search.)
- Dynamic Potential Search: $f = (w*f_{min}-g(n))/h(n)$ where f_{min} is the minimum f-cost on open
  Dynamic Potential Search searches dynamically based on the minimum f-cost in the open list. It often performs quite well, but not on grid domains, because it must re-open states when a shorter path is found, and the open list must be re-sorted when f_{min} changes.
- Improved Optimistic Search
  Improved Optimistic Search (IOS) is very similar to Optimistic search except (1) it doesn't re-open closed states in the optimistic portion of the search, (2) it has a better method of keeping track of improved solution quality, (3) it has better termination conditions, and (4) can use the alternate XDP/XUP priority functions.
- Improved Optimistic Search (XDP)
  This is IOS with the optimistic search using the XDP search.
- Improved Optimistic Search (XUP)
  This is IOS with the optimistic search using the XDP search.

Instructions

Aprendizaje de heurísticas

- Se supone que una función heurística h (n) estima el coste de una solución comenzando desde el estado en el nodo n
- Definir una heurística no siempre es fácil
  - Problemas relajados
  - Aprender de la experiencia → resolver muchos puzles proporciona muchos ejemplos a partir de los cuales se puede aprender h (n).
  - Cada ejemplo consiste en un estado de la ruta de la solución y el costo real de la solución desde ese punto.
  - A partir de estos ejemplos, se puede utilizar un algoritmo de aprendizaje para construir una función h (n) que pueda predecir los costos de solución para otros estados que surgen durante la búsqueda.
- Aprender heurísticas con métodos de aprendizaje por refuerzo (reinforcement learning) o Montecarlo.
- Otras técnicas para hacer esto utilizando redes neuronales, árboles de decisión y otros métodos que se verán en el 2ª cuatrimestre (IA2)

Aprendizaje de heurísticas

- Los métodos de aprendizaje inductivo funcionan mejor cuando se proporcionan características de un estado que son relevantes para predecir el valor del estado
  - Computo la solución con cualquier búsqueda.
  - 100 configuraciones de 8 puzles generadas aleatoriamente y recopilar estadísticas sobre los **costes reales de su solución**.
  - Calcular funciones en las configuraciones
    - f1 (n) = "número de casillas mal colocadas"
    - f2 (n) podría ser "el número de pares de fichas adyacentes que no son adyacentes en el estado del objetivo".

    Ej: Si f1(n)=5 ➔ el coste promedio de la solución es 14. [Aprendizaje inductivo]
  - En vez de usar f1 directamente como h' se puede usar el valor de f1 para predecir h'(n).
- Combinar f1 (n) y f2 (n) para predecir h (n)
  - Un enfoque común es usar una combinación lineal: h (n) = c1*f1 (n) + c2*f2 (n).
  - Las constantes c1 y c2 se ajustan para dar el mejor ajuste a los datos reales sobre los costes de la solución.
  - Uno espera que tanto c1 como c2 sean positivos porque las fichas mal ubicadas y los pares adyacentes incorrectos hacen que el problema sea más difícil de resolver.
  - Esta heurística cumple la condición de que h (n) = 0 para los estados objetivo, pero no es necesariamente admisible o consistente.

Más sobre búsqueda heurística

- **Búsquedas con adversario**
  - **Minimax y poda alfa beta**
  - **Búsqueda Monte Carlo** (**MCTS**) - se usa mucho en videojuegos y juegos no deterministas (tipo póquer)
    - Aprendizaje de heurísticas: selección de movimientos al azar + aprendizaje inductivo
    - Se elige el movimiento que llevó a mayor número de victorias en juegos anteriores.
    - Juegos en los que se pueden hacer simulaciones
    - Se ha usado en GO

  https://int8.io/monte-carlo-tree-search-beginners-guide/
- **Planificación**
- **Búsqueda local**
- **Aprendizaje de heurísticas con RL**

Bibliografía

- **Meseguer, P.** Búsqueda heurística I. http://www.iiia.csic.es/~pedro/busqueda1-introduccion.pdf
- **Russell, S., Norvig, P.** Artificial Intelligence. A Modern Approach. Prentice Hall, 2013, 3ª edición.
- **Vázquez-Salceda, J.** Apuntes de UPC

  http://www.lsi.upc.edu/~jvazquez/teaching/iag/apuntes/ApuntesIA-G-Busc.pdf
- **Apuntes de Inteligencia Artificial. UPC** https://www.cs.upc.edu/~bejar/ia/material/teoria/ApuntesIA.pdf
- **Recursos de planning**
  - https://planning.wiki/