Resolución de problemas y espacio de búsqueda

- **Métodos informados o heurísticos**
  - Introducción
    - Métodos informados
    - Heurísticas
    - Función heurística
    - Ejemplo: 8-puzzle
  - Búsqueda primero el mejor
  - Algoritmos de mejora iterativa
  - Búsqueda con adversario

--- Métodos informados o heurísticos ---

- Métodos no informados
  - Muy ineficientes en la mayoría de los casos
  - Ante explosión combinatoria, la fuerza bruta es impracticable
- Métodos informados
  - Orientan 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”
      - elección informada del siguiente paso: qué nodo es más “prometedor”
    - Evita la explosión combinatoria podando el árbol de búsqueda
      - No genera nodos no prometedores y así mejora rendimiento
      - Puede encontrar una solución aceptablemente buena en tiempo razonable
  - Limitaciones
    - No garantizan encontrar solución, aunque existan soluciones
    - Algunos no garantizan soluciones con buenas propiedades
      - que sea de longitud mínima o de coste óptimo

Funciones Heurísticas

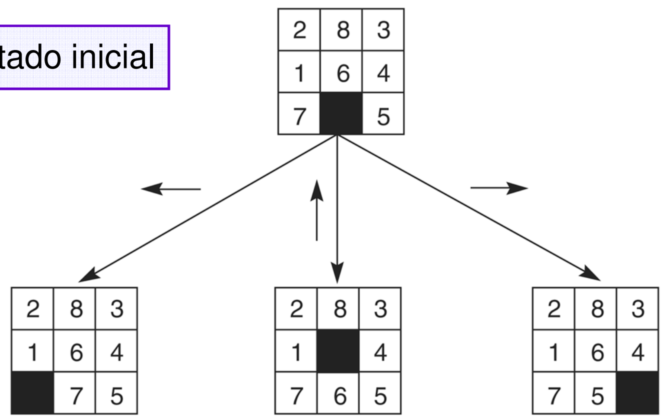
- Heurística: técnica que mejora la eficiencia de la búsqueda
  - Pudiendo sacrificar la completitud
    - posibilidad de fallar (no encontrar solución, habiéndola)
    - o de encontrar soluciones no óptimas
  - Utiliza conocimiento específico del dominio problema
    - más allá de la definición del problema en sí mismo
  - Propociona consejos
    - 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
- Método de búsqueda heurística
  - Algoritmo que usa una heurística en la búsqueda en el espacio de estados
- Importante seleccionar una función heurística adecuada.
  - Cómo escoger una buena?

Función heurística

- Función heurística *h'*
  - Asocia a cada estado del espacio de estados una cierta cantidad numérica que evalúa de algún modo lo prometedor que es ese estado para alcanzar un estado objetivo
  - Esto sirve para seleccionar 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 “cercanía” / distancia de un estado a un estado objetivo
      - Los estados de menor valor son los preferidos
    - Ambos puntos de vista son complementarios
      - Un cambio de signo permite pasar de una perspectiva a la otra
  - Convenio: asumiremos la 2ª interpretación
    - **Valores no negativos, el mejor es el menor, objetivos valor heurístico 0**

Ejemplo: heurísticas para el 8-puzzle

Estado inicial



Estado objetivo

1	2	3
8		4
7	6	5

Ejemplo: heurísticas para el 8-puzzle

- *Usamos* ***h*** *para indicar valores calculados y* ***h'*** *para valores estimados*
- 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 la importancia que merece a la dificultad de la inversión de fichas
  - Si 2 fichas están dispuestas de forma contigua y han de intercambiar sus posiciones, ponerlas en su sitio supone más de 2 movimientos

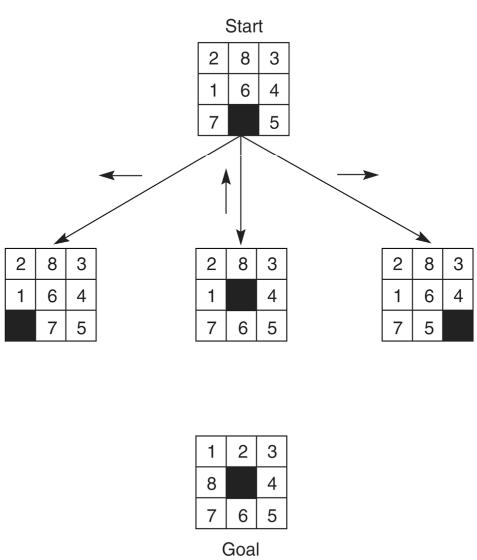
- h_c = el doble del nº de pares de fichas a "invertir entre sí"
  - Esta heurística también es pobre, puesto que se concentra demasiado en un cierto tipo de dificultad
  - En particular, tendrán valor 0 muchos tableros que no son el objetivo
  - Lo que suele hacerse con este tipo de heurísticas es incorporarlas a la función heurística final
    - Ejemplo: 2*0 = 0

2	8	3
1	6	4
	7	5

1	2	3
8		4
7	6	5

Ejemplo: heurísticas para el 8-puzzle

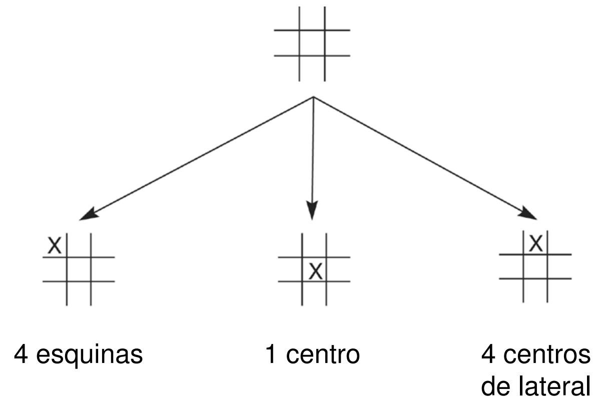
- $h_d(n) = h_a(n) + h_c(n)$
  - Es la heurística más fina, pero requiere un mayor esfuerzo de cálculo
  - Aún podría mejorarse...



	h_a	h_b	h_c	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

Ejemplo: las 3 en raya

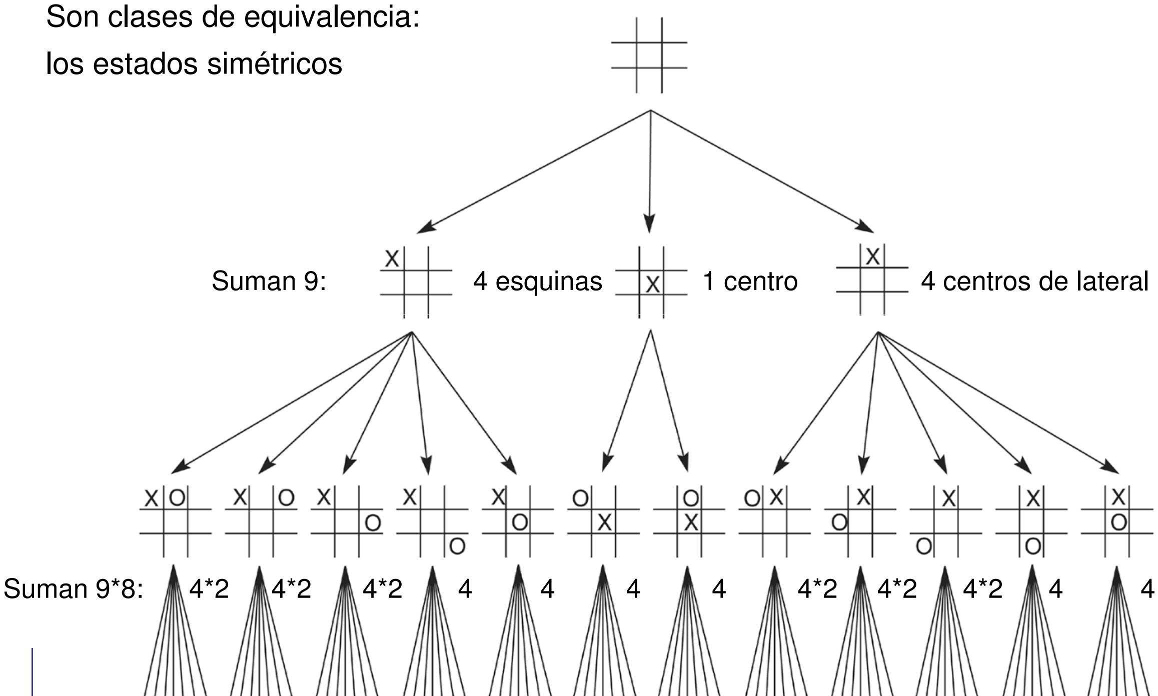
- Representación obvia
  - 9 movimientos iniciales, 8 posibles segundos movimientos, ...
  - Muchos operadores y “estados” distintos (configuraciones de tablero)
- Representación con reducción simétrica
  - 3 movimientos iniciales
    - esquina
    - centro del tablero
    - centro de un lateral
  - Configuraciones equivalentes por simetría
    - Suman 9 “movimientos”, aunque son sólo 3



  - Reduce el espacio de estados y, por lo tanto, el espacio de búsqueda

Ejemplo: reducción simétrica en las 3 en raya

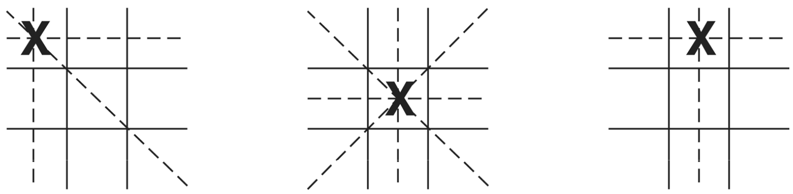
Son clases de equivalencia:
los estados simétricos



Ejemplo: heurística para las 3 en raya

- Heurística sencilla que casi elimina la búsqueda
  - **h'** Movimiento que maximice el número de líneas ganadoras
    - Para el nodo inicial

3 desde esquina



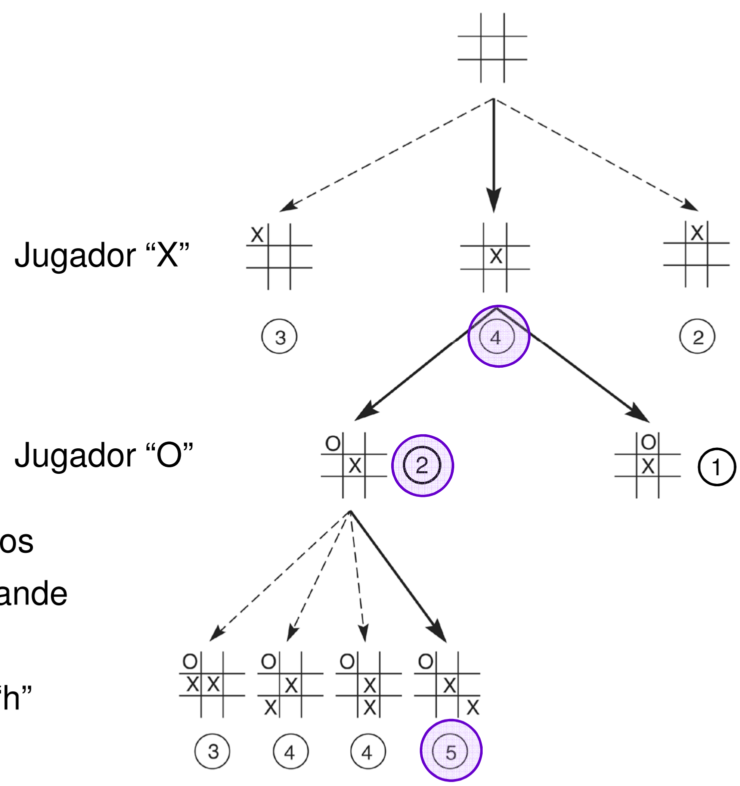
2 desde centro lateral

4 desde centro tablero

1. El movimiento inicial será al centro del tablero
   - Los demás no se considerarán, ni tampoco sus descendientes
   - Poda 2/3 del espacio de búsqueda con el primer movimiento
2. El oponente elegirá esquina o centro de un lateral
3. Siguiente movimiento: misma heurística a un único nodo

- La búsqueda exhaustiva no es necesaria

Ejemplo: reducción por heurística en las 3 en raya



En cada uno de estos niveles sólo se expande un nodo: el más prometedor según "h"

Resolución de problemas y espacio de búsqueda

- **Métodos informados o heurísticos → aplicar h**
  - Introducción
  - Búsquedas primero el mejor
    - Introducción
    - Búsqueda voraz *(greedy)*
    - Búsqueda óptima: A*
    - Más sobre heurísticas
      - Comparación de calidad
      - Generación
  - Algoritmos de mejora iterativa
  - Búsqueda con adversario

--- Búsquedas primero el mejor ---

- "Primero el aparentemente mejor", el nodo más prometedor
  - Si sabemos 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 en base a
  - una función de evaluación *f '(n)* que tenga en cuenta
  - una estimación del coste h'(n) necesario
    - para llegar a una solución a partir de ese nodo *n*
- Se usa una estimación en lugar de criterios fijos (1º profundidad o anchura)
  - El nodo seleccionado será aquel que minimice la función de evaluación
  - Gestión de *abiertos*: cola de prioridad, ordenada por el valor de la función de evaluación
- Asumiremos que el coste de los operadores nunca puede ser negativo

Búsquedas primero el mejor

La función f ' puede ser básicamente de dos tipos:

1. Una función que considere exclusivamente lo que "falta"
   - el coste mínimo estimado para llegar a una solución a partir del nodo *n*
   - *f '(n) = h'(n)* => búsqueda voraz

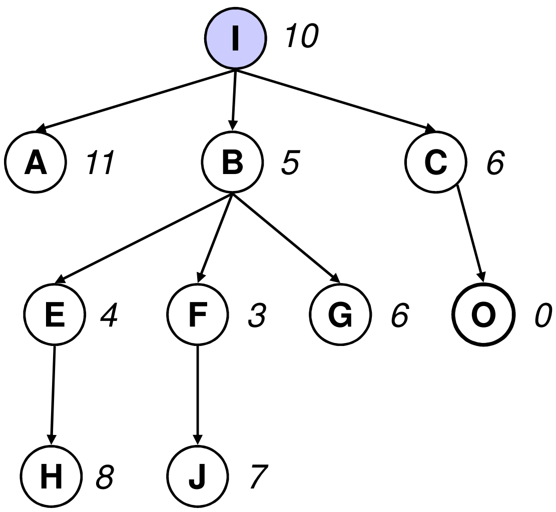
2. Una función que considere
   - el coste total estimado del camino
     - desde el nodo inicial a un nodo objetivo que pase el nodo *n*
   - La función ***f '*** está formada por dos componentes:
     - *g(n)* = coste del camino desde nodo inicial hasta *n*
       - No es una estimación, sino un coste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

--- 1. Búsqueda voraz *(greedy)*---

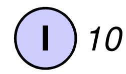
- ***f '(n) = h'(n)***
  - Representa el coste mínimo estimado para
  - llegar desde *n* a un nodo objetivo (por el camino más barato)
    - *Donde h'(n) = 0* para los nodos objetivo
- Voraz: en cada paso trata de ponerse lo más cerca posible del objetivo
- Propiedades
  - No es óptima ni completa, como primero en profundidad *(caminos ∞)*
    - No tiene en cuenta el coste real; sólo el coste estimado
  - Caso peor: si la heurística es muy mala, más bien desinforma *(no realista)*
    - 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, las complejidades podrían reducirse mucho
    - Esto depende del problema y de la calidad de la heurística
    - En general, expande pocos nodos: escaso coste de búsqueda

Ejemplo: búsqueda voraz

Espacio de estados



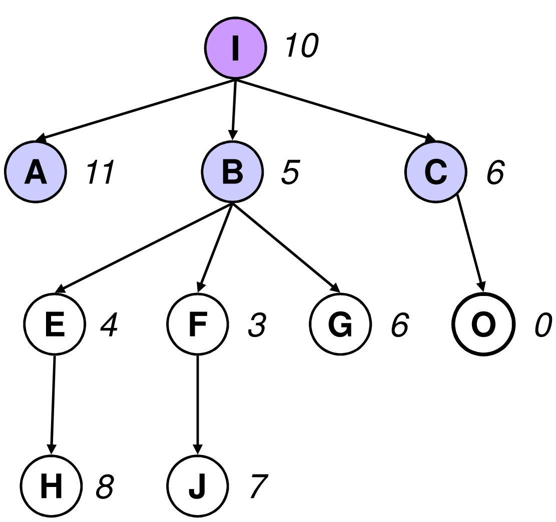
Espacio de búsqueda



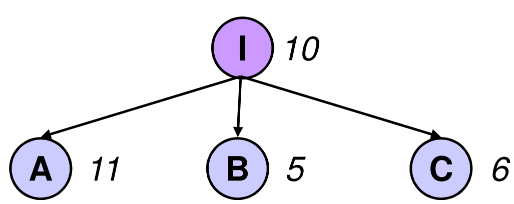
Cola de prioridad de nodos abiertos: I (10)

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

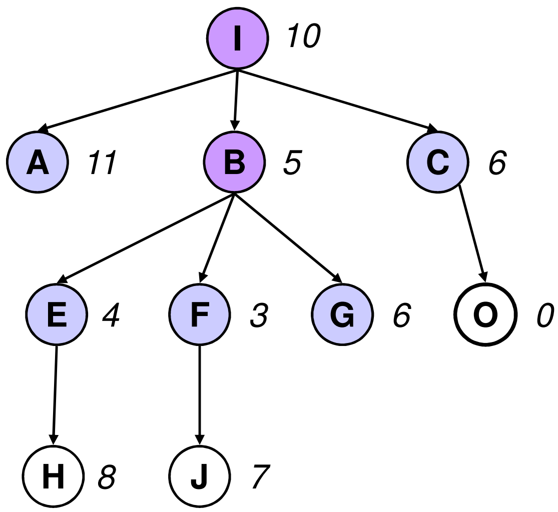


Cola de prioridad de nodos abiertos: A (11) C (6) B (5)

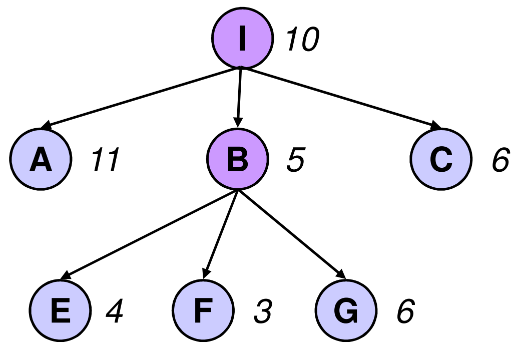
Nodos expandidos (por orden): I

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

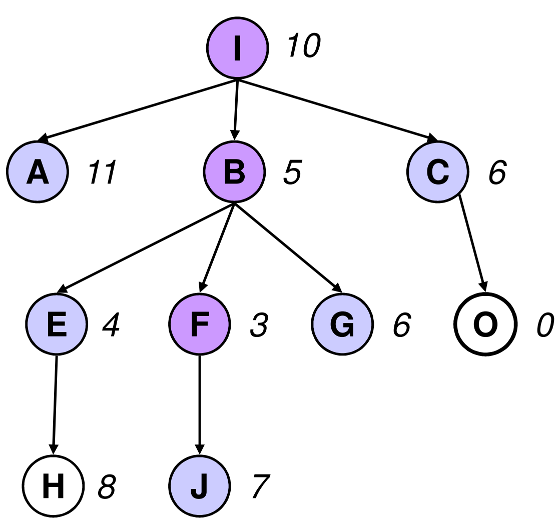


Cola de prioridad de nodos abiertos: A (11) G(6) C (6) E (4) F (3)

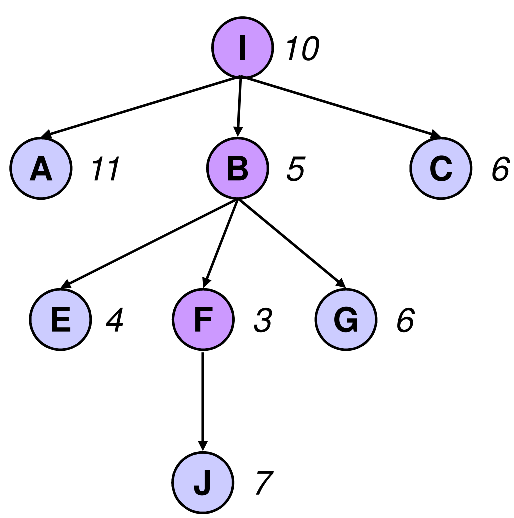
Nodos expandidos (por orden): I B

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

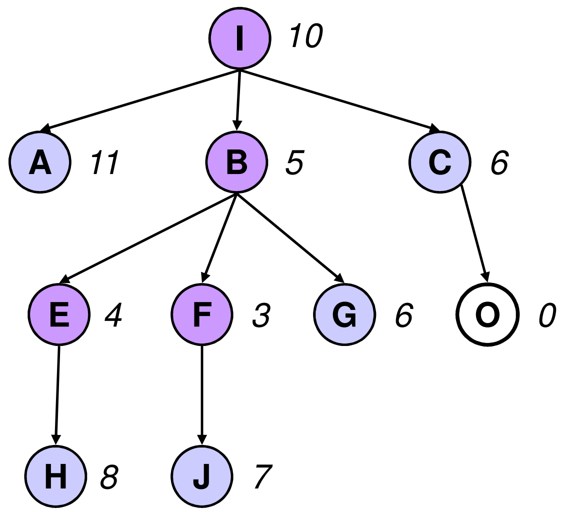


Cola de prioridad de nodos abiertos: A (11) J (7) G (6) C (6) E (4)

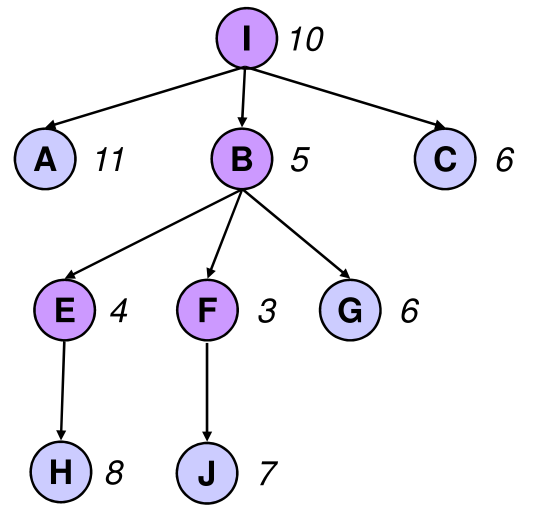
Nodos expandidos (por orden): I B F

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

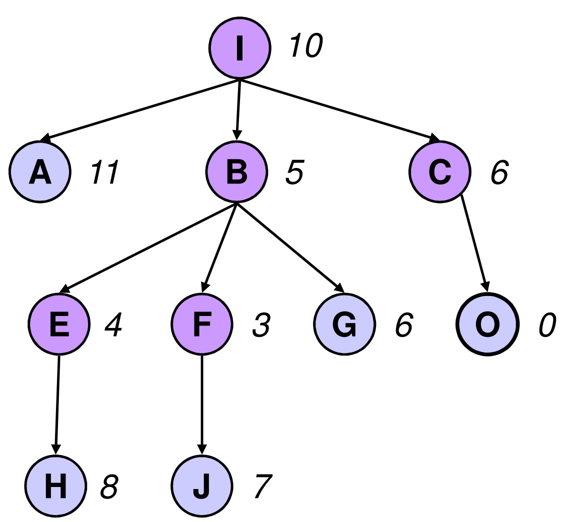


Cola de prioridad de nodos abiertos: A (11) H (8) J (7) G (6) C (6)

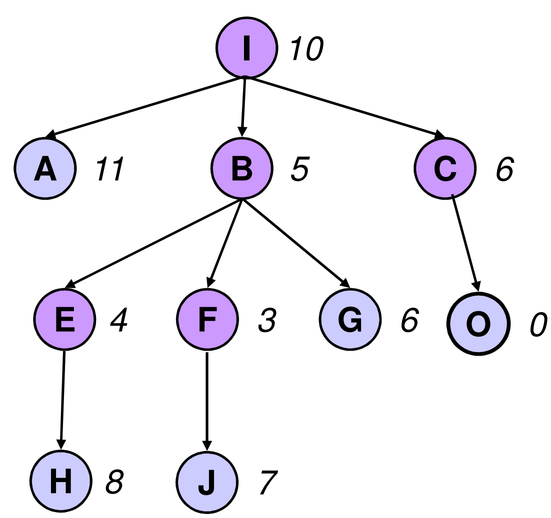
Nodos expandidos (por orden): I B F E

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

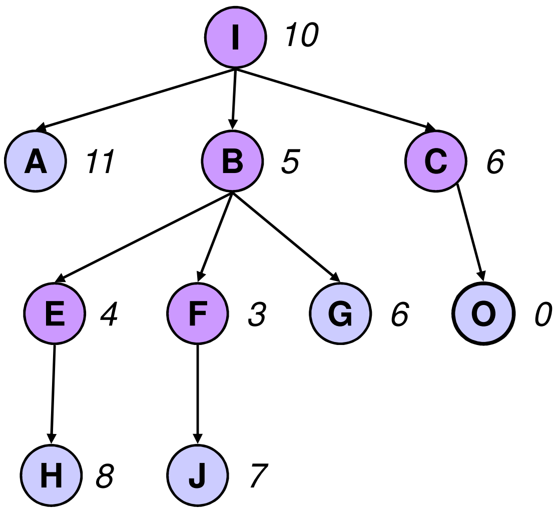


Cola de prioridad de nodos abiertos: A (11) H (8) J (7) G (6) O (0)

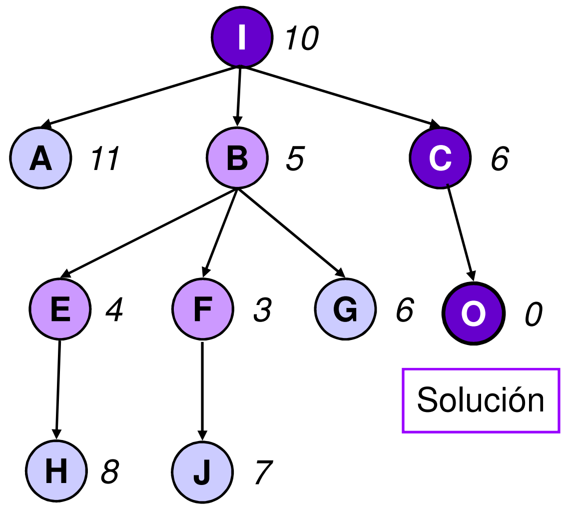
Nodos expandidos (por orden): I B F E C

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda



Cola de prioridad de nodos abiertos: A (11) H (8) J (7) G (6)

Nodos expandidos (por orden): I B F E C O

--- 2. Búsqueda óptima: 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

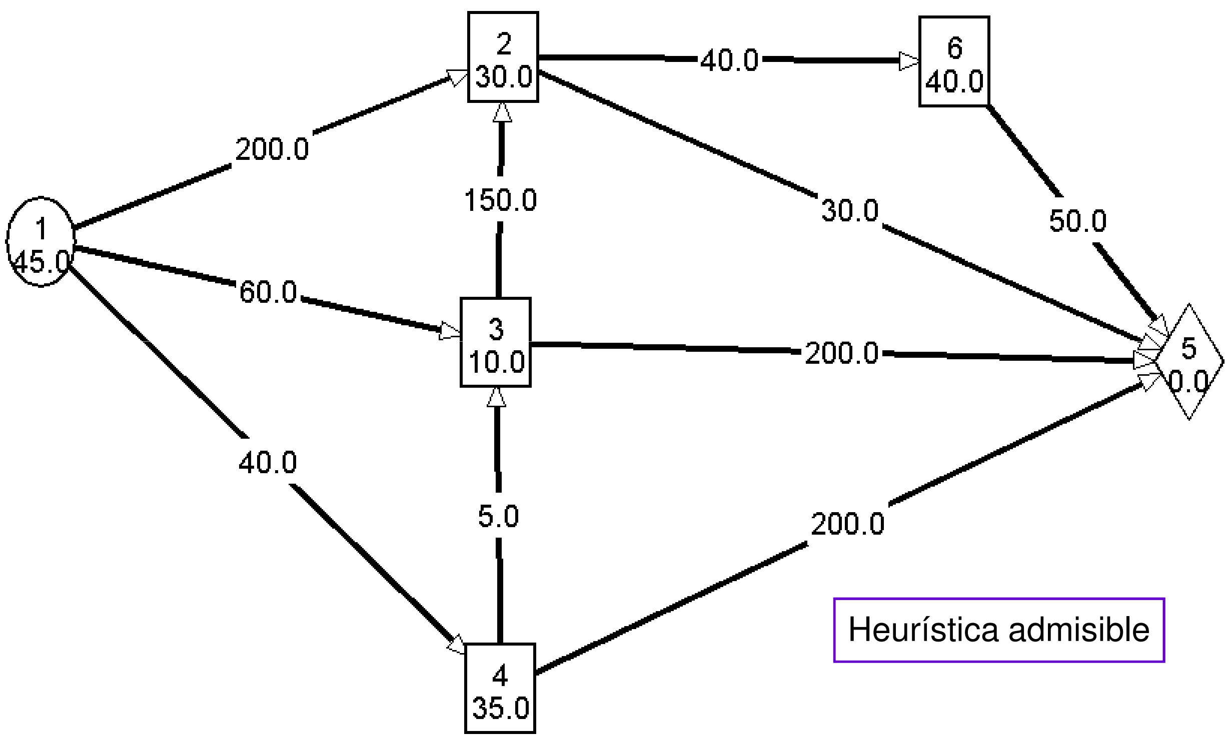
Condiciones sobre *h'* para garantizar la optimalidad

- *h'* es una heurística admisible si $h'(n) \leq h(n)$ para todo *n,* donde
  - *h(n)* representa el coste real para ir desde *n* a un nodo objetivo por el camino de menor coste
  - No ha de sobrestimar nunca el coste de alcanzar el objetivo
  - Son heurísticas optimistas por naturaleza
- **Si *h'* es admisible**, la búsqueda con $f'(n) = g(n) + h'(n)$
  - se denomina A* o búsqueda óptima (si no, se llama A)
- Si *h'* es admisible, entonces $f'(n) = g(n) + h'(n)$ cumple:

  1) $f'(n) <= f(n)$ para todo *n*
  - $f(n)$ = coste mínimo real de cualquier solución que pase por *n*

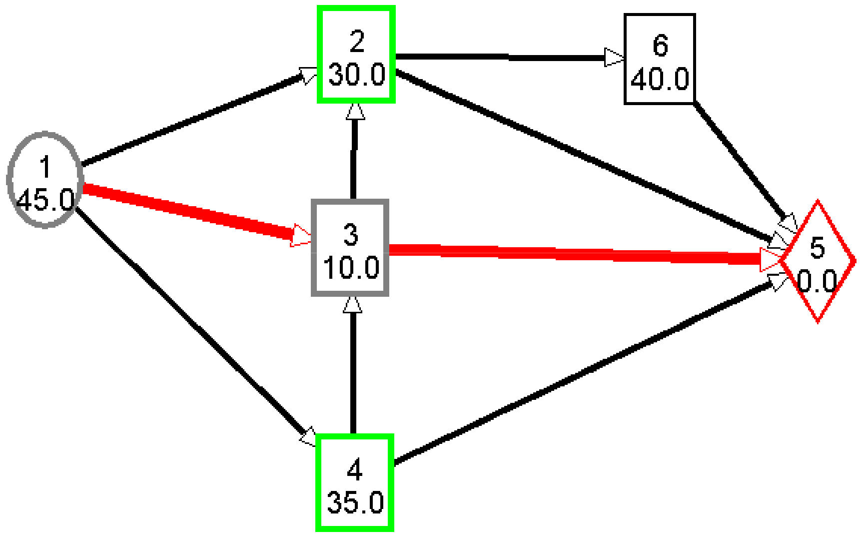
  2) $f'(n) = f(n)$ para los nodos objetivo *n* (puesto que $h'(n) = h(n) = 0$)
- Un algoritmo que utiliza una función *f '* que cumple *1)* y *2)* es óptimo
  - La solución que encuentra es óptima *(la de menor coste real)*

Ejemplo



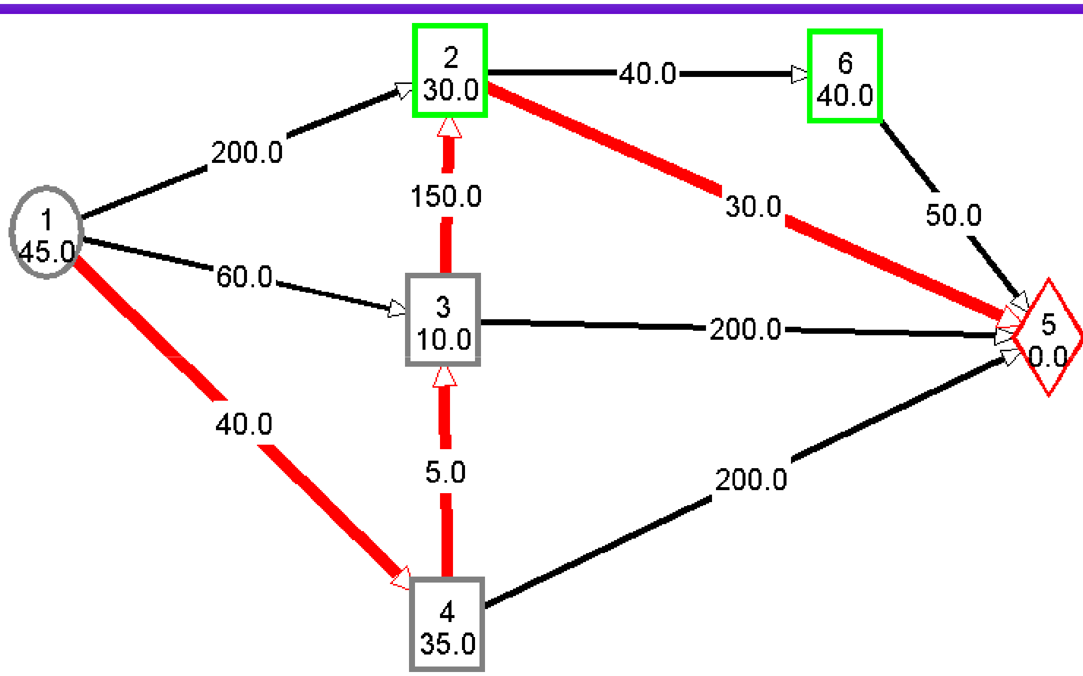
Heurística admisible

Solución con voraz



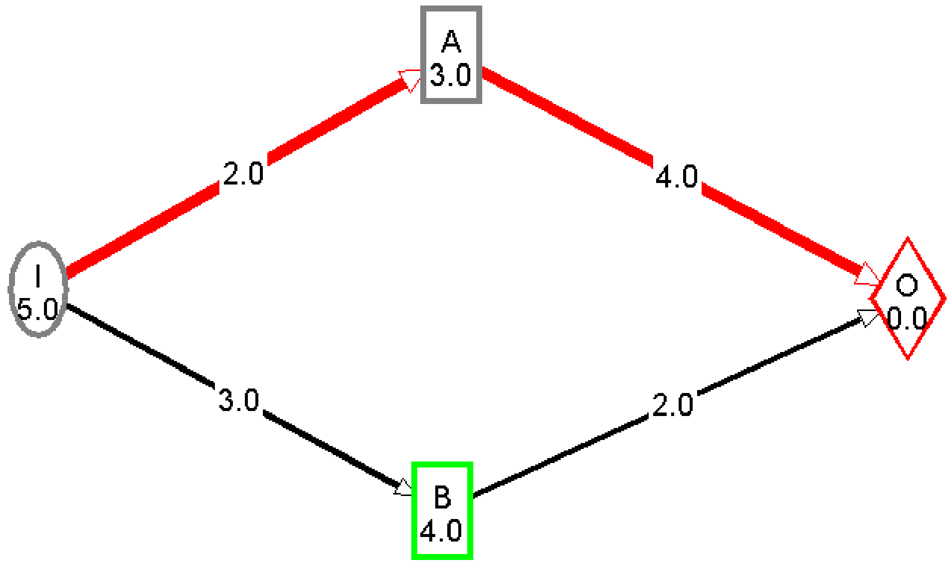
Solución con voraz: 1-3-5 Coste (no tenido en cuenta): 60+200 = 260

Solución con A*



Solución con A*: 1-4-3-2-5 Coste mínimo: 40+5+150+30 = 225

Ejemplo: A, no A* (heurística no admisible)



Heurística no admisible: no garantiza coste mínimo; no es A*

Implementación de A*

- Información almacenada en un nodo ***n***:
  - Descripción del estado
  - Puntero al nodo padre
    - Para la reconstrucción del camino al encontrar solución
  - Valores de *f '(n), g(n)* y *h'(n)*
  - Acceso a sus sucesores inmediatos
    - Cuando se encuentra un camino mejor, la mejora ha de transmitirse a sus sucesores (*g(n)* depende del camino)
- Se usan dos estructuras para almacenar los nodos:
  - Abiertos: nodos generados y evaluados pero no expandidos
    - Gestión de abiertos: cola de prioridad
  - Cerrados: nodos ya expandidos

Algoritmo A*

```
cerrados := [ ] ;      abiertos   := [INICIAL];     f'(INICIAL) := h'(INICIAL)
repetir
    si abiertos = [] entonces fallo % fallo: termina sin encontrar solución
    si no                                              % quedan nodos
        extraer MEJOR_NODO de abiertos (con f' mínima)
                                                       % cola de prioridad
        mover MEJOR_NODO de abiertos a cerrados
        si MEJOR_NODO contiene estado_objetivo entonces
                solución_encontrada := true
        si no
                generar lista SUCESORES de MEJOR_NODO % es el padre
                para cada SUCESOR hacer
                        TRATAR_SUCESOR (SUCESOR)
hasta solución_encontrada o fallo
```


TRATAR_SUCESOR (SUCESOR)

```
% Creamos nodo para SUCESOR
SUCESOR.ANTERIOR := MEJOR_NODO        % nodo padre
g(SUCESOR):= g(MEJOR_NODO) + coste(MEJOR_NODO, SUCESOR)
    % coste del camino hasta SUCESOR pasando por MEJORNODO

% DISTINCIÓN DE CASOS sobre SUCESOR:
% ¿está el estado ya en algún nodo de abiertos o cerrados?
% caso 1: ni en abiertos ni en cerrados
% caso 2: ya en abiertos
% caso 3: ya en cerrados
```

TRATAR_SUCESOR: ni en *cerrados* ni en *abiertos*

caso SUCESOR no estaba en *abiertos* ni *cerrados*
  añadir SUCESOR a ABIERTOS
  añadir SUCESOR a MEJOR_NODO.SUCESORES
  f'(SUCESOR) := g(SUCESOR) + h'(SUCESOR)

TRATAR_SUCESOR: ya en *abiertos*

VIEJO.ESTADO := SUCESOR.ESTADO
Caso VIEJO.ESTADO está en *abiertos*
  si g(SUCESOR) < g(VIEJO) **entonces**
    % nos quedamos con el nodo con menor coste
    % actualizando el que ya está en *abiertos*
    VIEJO.ANTERIOR := MEJOR_NODO
    actualizar g(VIEJO) y f'(VIEJO)
  eliminar SUCESOR

  añadir VIEJO a MEJOR_NODO.SUCESORES

TRATAR_SUCESOR: ya en *cerrados*

VIEJO.ESTADO := SUCESOR.ESTADO

Caso VIEJO.ESTADO está en cerrados

si g(SUCESOR) < g(VIEJO) entonces % (no si monotonía)

% nos quedamos con el nodo con menor coste

% actualizando el que ya está en *cerrados*

VIEJO.ANTERIOR := MEJOR_NODO

actualizar g(VIEJO) y f'(VIEJO)

propagar g a sucesores de VIEJO

eliminar SUCESOR

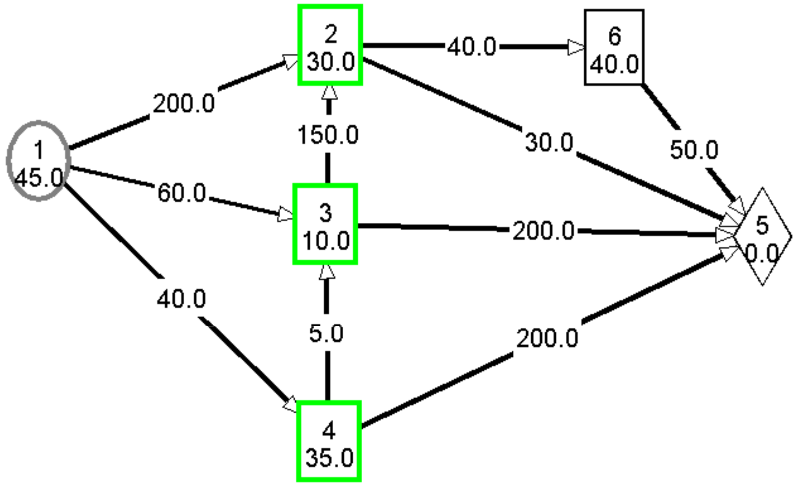
añadir VIEJO a MEJOR_NODO.SUCESORES

propagar g a sucesores de VIEJO

- Generar sucesores de viejo
- Para cada sucesor_viejo
  - % si se actualiza un nodo de cerrados hay que recorrer
  - % sus hijos para propagar el nuevo camino
  - Si sucesor_viejo.estado en cerrados => nodo_cerrados
    - si g(VIEJO) + coste(VIEJO, sucesor_viejo) < g(nodo_cerrados)
      - actualizar camino, g y f’
    - Propagar g a sucesores de sucesor_viejo
  - Si sucesor_viejo.estado en abiertos => nodo_abiertos
    - si g(VIEJO) + coste(VIEJO, sucesor_viejo) < g(nodo_abiertos)
      - actualizar camino, g y f’

Ejemplo A*: se expande 1

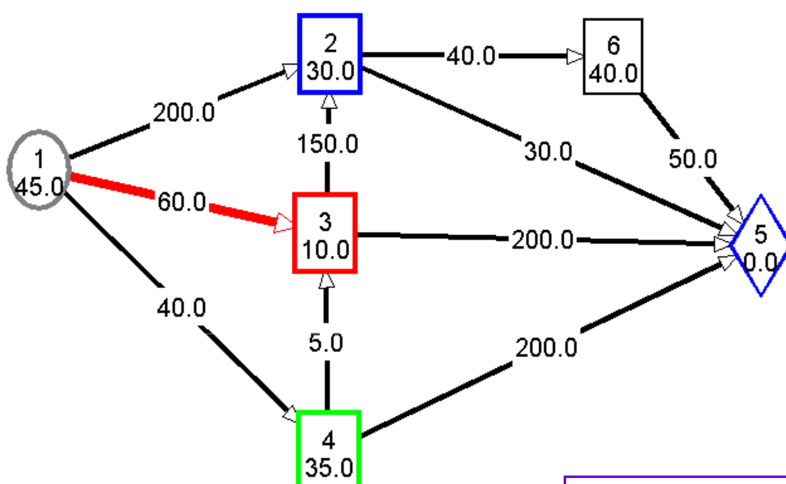
Abiertos				
padre		1	1	1
estado	~~**1**~~	**2**	**3**	**4**
h'	~~45~~	30	10	35
g	~~0~~	200	60	40
f'	~~45~~	230	70	75



Cerrados: 1

Ejemplo A*: se expande 3

Abiertos				
padre		1	1	1
estado	~~**1**~~	**2**	**3**	**4**
h'	~~45~~	30	10	35
g	~~0~~	200	60	40
f'	~~45~~	230	70	75

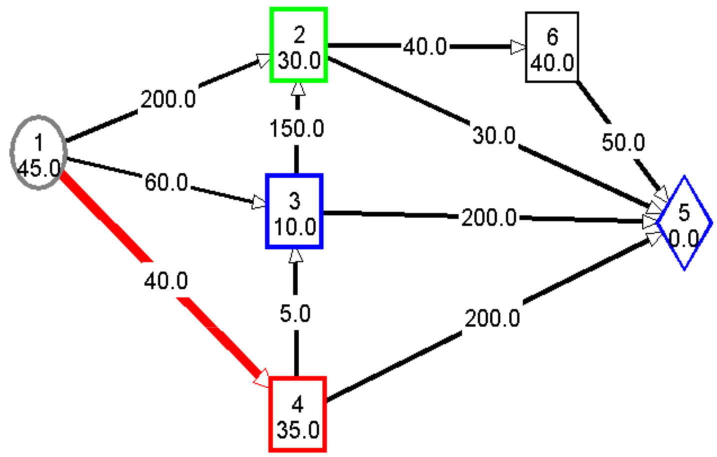


Cerrados: 1, 3

- Camino
  - 1-3 Coste: 60
- Sucesores de 3: 2 y 5
  - 5 es nuevo: se añade a *abiertos*
  - 2 ya está en *abiertos*
  - ¿Es mejor este nuevo camino 1-3-2 que 1-2?
    - No: tiene peor coste (60+150 frente a 200)
      - Nos quedamos con el que teníamos

Ejemplo A*: se expande 4

Abiertos					
padre		1	1	1	3
estado	**1**	**2**	**3**	**4**	**5**
h'	45	30	10	35	0
g	0	200	60	40	60+200
f'	45	230	70	75	260



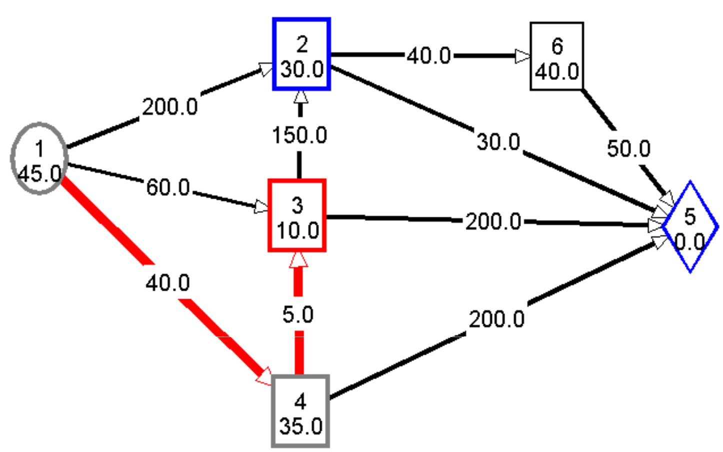
- Cambio de camino
  - 1-4 Coste: 40

Cerrados: 1, 3, 4

- Sucesores de 4: 3 y 5
  - 3 está en *cerrados* (ya había sido expandido)
  - ¿Es mejor este nuevo camino 1-4-3 que 1-3?
    - Sí: tiene mejor coste (40+5 frente a 60)
    - Nos quedamos con éste y propagamos el cambio a los sucesores de 3 (2 y 5)

Ejemplo A*

Abiertos					
padre		1	4	1	3
estado	**1**	**2**	**3**	**4**	**5**
h'	45	30	10	35	0
g	0	200	45	40	45+200
f'	45	230	55	75	245

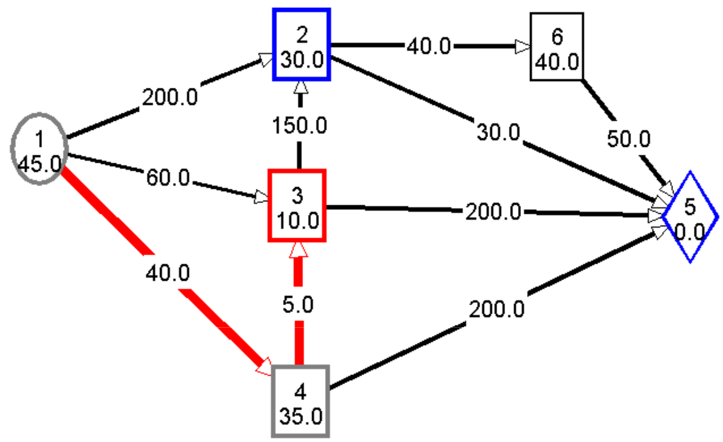


- Actualizamos 3 y su hijo 5
- Falta su hijo 2
  - Está en *abiertos* como hijo de 1
  - ¿Es mejor 1-4-3-2 que 1-2?
    - Sí: tiene mejor coste (40+5+150 frente a 200)
    - Nos quedamos con este nuevo camino

Cerrados: 1, 3, 4

Ejemplo A*

Abiertos					
padre		3	4	1	3
estado	**1**	**2**	**3**	**4**	**5**
h'	45	30	10	35	0
g	0	195	45	40	45+200
f'	45	225	55	75	245

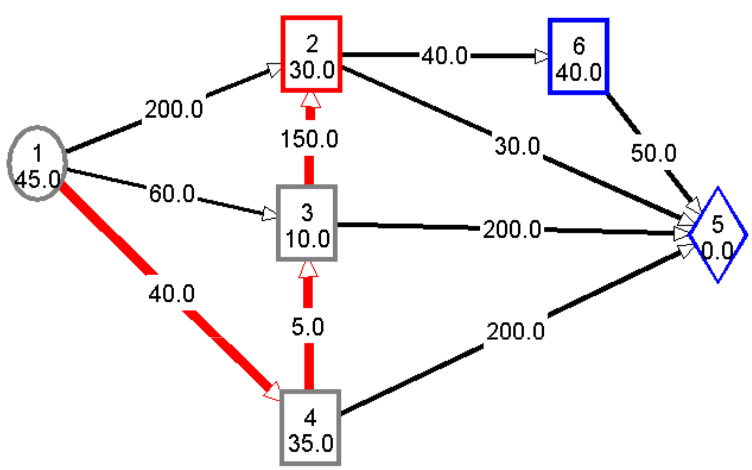


- Falta 5 como sucesor de 4
  - 5 ya está en *abiertos*
  - ¿Es mejor 1-4-5 que 1-4-3-5?
    - Sí: tiene mejor coste (40+200 frente a 245)
    - Nos quedamos con este nuevo camino

Cerrados: 1, 3, 4

Ejemplo A*: se expande 2

Abiertos					
padre		3	4	1	4
estado	**1**	**2**	**3**	**4**	**5**
h'	45	30	10	35	0
g	0	195	45	40	240
f'	45	225	55	75	240

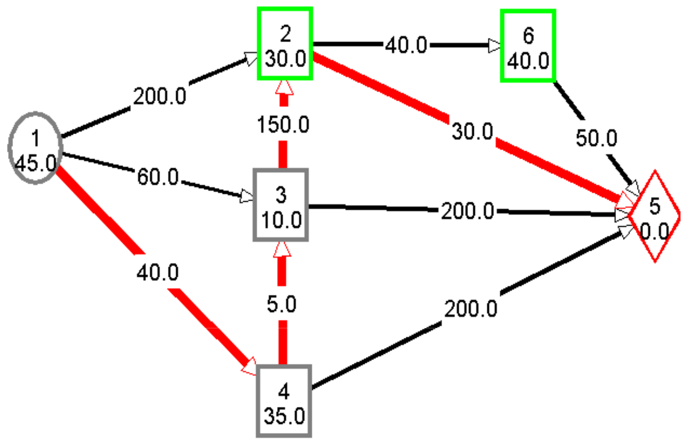


- Camino
  - 1-4-3-2 Coste: 195
- Sucesores de 2: 6 y 5
  - 5 está en *abiertos* y 6 es nuevo (lo añadimos a *abiertos)*
  - ¿Es mejor 1-4-3-2-5 que 1-4-5?
    - Sí: tiene mejor coste (195+30 frente a 240)
    - Nos quedamos con este nuevo camino

Cerrados: 1, 3, 4, 2

Ejemplo A*: se expande 5

Abiertos						
padre		3	4	1	2	2
estado	**1**	**2**	**3**	**4**	**5**	**6**
h'	45	30	10	35	0	40
g	0	195	45	40	225	235
f'	45	225	55	75	225	275

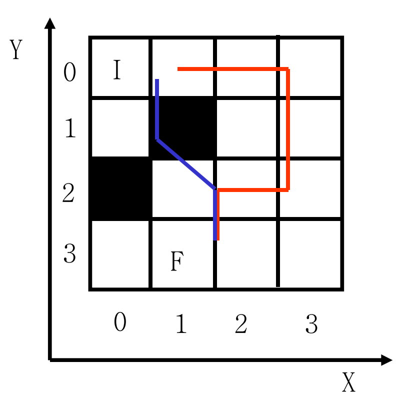


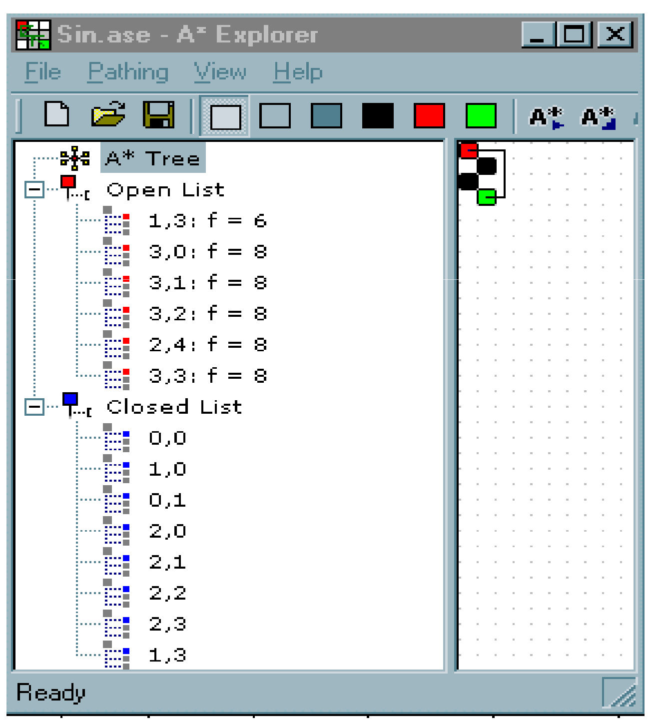
- Camino
  - 1-4-3-2-5 Coste: 225
- 5 es estado objetivo
  - Solución encontrada
  - Al ser admisible la heurística, hay garantía de optimalidad
    - Hemos encontrado el camino de menor coste

Cerrados: 1, 3, 4, 2, 5

Ejemplo: A* Explorer

→ http://www.generation5.org/content/2002/ase.asp

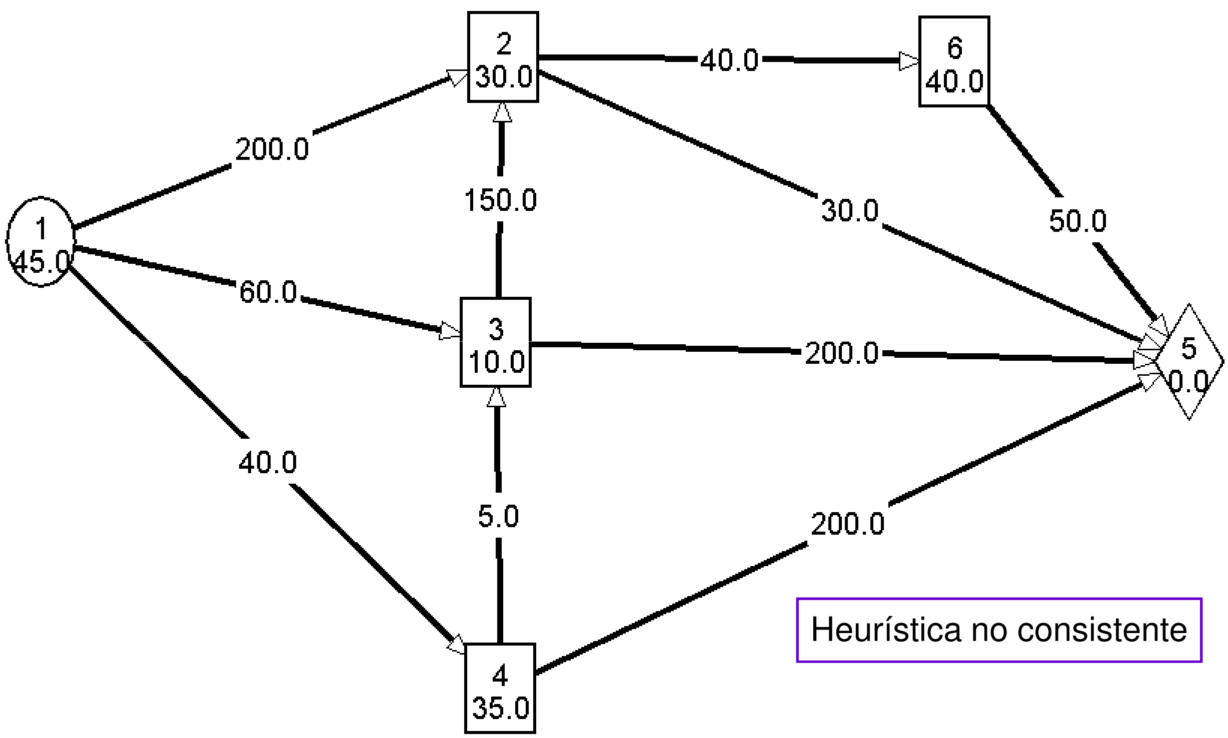




Mejorar eficiencia de A* con buenas h’ y f’

- **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 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 *(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
  - 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



Heurística no consistente

Mejora del A* y comportamiento

- Si *h'* no es consistente, pero sí admisible, mejoramos un poco el A*
  - puede modificarse *h'* dinámicamente durante la búsqueda para
  - que cumpla la condición de consistencia *h' (n) <= c(n, n') + h' (n')*
    - En cada paso, comprobamos los valores de *h'*
      - para los sucesores (los h') del nodo *n* que acaba de ser expandido
    - Si para alguno de estos valores de *h' se cumple que*
    - *Si 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*
  - 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 optimalidad del A*

- 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) <= 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 ***r*** es finito y
    - existe una constante *ε > 0* tal que
      - el coste de cualquier operador es siempre *>= ε*
- 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*

- 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 $O(r^p)$
- El crecimiento exponencial no occure 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 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 crecimiento exponencial

- Por ello, a menudo es poco práctico 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

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 **r*** (método experimental)
  - Es mejor la que menor **r*** tenga

Dominancia

- Dadas dos heurísticas admisibles h'_1 y h'_2,
  - se dice que h'_2 domina a h'_1 si $h'_2(n) \geq h'_1(n)$ para todo ***n***
    - Aproxima más a $h(n)$: $h(n) \geq h'_2(n) \geq h'_1(n)$
- La dominancia se traslada directamente a la eficiencia
  - A* usando h'_2 nunca expandirá más nodos que A* usando h'_1
    - Excepto quizás algunos nodos n con $f'(n) = f^*$
  - A* usando h'_2 estará más informada que A* usando h'_1
    - A* con cualquier heurística no nula está más informada (y normalmente expandirá menos nodos) que la búsqueda de coste uniforme
- Será preferible elegir h'_2 siempre que
  - Siga siendo admisible
  - El coste de cómputo de h'_2 no debe superar la potencial ganancia

Factor de ramificación efectivo r* (r-estrella)

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

  $52 = 1 + 1.91 + (1.91)^2 + (1.91)^3 + (1.91)^4 + (1.91)^5 \rightarrow$ **r*** = 1.91
- Conocemos ***p*** y ***N***. Entonces **N = O(r*)^P** , un cálculo aproximado $r^* = \sqrt[p]{N}$ → 2,2
- ***r**** es constante para problemas lo suficientemente difíciles
- Las medidas experimentales de ***r**** sobre un pequeño conjunto de problemas
  - proporcionar una buena guía para la utilidad total de la heurística
- Una buena heurística tendrá un valor de ***r**** cercano a **1**
- Ejemplo: 8-puzzle con *p = 12*
  - *Búsqueda ciega (profundización iterativa): N = 3.644.035, r* = 2.78*
  - *A*(h_b): N = 227, r* = 1.42* *A*(h_a): N = 73, r* = 1.24*

    descolocadas *Manhattan*

Generación de heurísticas: Tres Métodos

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

  1.- Relajación: Relajar las restricciones o reglas del problema

  2.- Abstracción: Definir un subproblema del original

  a) Abstraer un estado eliminando lo que diferenciaba a otros parecidos conservando la transición entre estados (operadores originales)

  EJ: ignorar la mitad de las fichas en el Puzzle-8

  b) Las distancias al objetivo desde esos estados se usan con h'

  c) Calcular a) y b) debe ser eficiente (barato)

  d) otro modo es salvar en una BD lo ya calculado en forma de patrones empezando desde el obejtivo hacia atrás.

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

Generación de heurísticas: 1.- Relajación

- A un problema simplificado con menos restricciones en las acciones
  - *(relajación de restricciones)* se le llama problema relajado
- El coste de una solución óptima en un problema relajado es
  - Una heurística admisible para el problema original
  - Es crucial que los problemas relajados se resuelvan sin búsqueda
  - Así, el coste se calcula en vez de estimarlo
  - Computacionalmente no debe ser más costoso que *expandir un nodo*
- Ejemplo: 8-puzzle obtenemos dos heurísticas h_b *descolocadas y* h_a *Manhattan*
  - Ambas son estimaciones de la longitud del camino restante
  - También son longitudes de caminos exactos
  - para problemas *simplificados* del puzzle (quitando algunas restricciones):
    - Una ficha puede moverse a cualquier casilla (ocupada o no): h_b
    - Una ficha puede moverse 1 casilla aunque la casilla estuviera ocupada: h_a

Generación de heurísticas: automáticamente

- Se construyen automáticamente problemas relajados con un lenguaje formal
  - Descripción de acciones del 8-puzzle:
    - una ficha puede moverse de la casilla A a la casilla B
      - si A es horizontal o verticalmente adyacente a B y
      - B es la casilla vacía
  - Generación de problemas relajados (quitando una o ambas condiciones)
    a) una ficha puede moverse de la casilla A a la casilla B si A es horizontal o verticalmente adyacente a B
    b) una ficha puede moverse de la casilla A a la casilla B
    c) una ficha puede moverse de la casilla A a la casilla B si B es la casilla vacía
  - Generación de heurísticas
    a) Distancia de Manhattan: moviendo cada ficha en dirección a su destino
    b) Número de fichas descoladas: moviendo cada ficha a su destino en un paso
- ABSOLVER (1993): programa que genera heurísticas automáticamente

Generación de heurísticas: 2.- Abstracción

- Definir un subproblema del original se le llama Abstracción

- Tenemos varios pasos
  1.- Abstraer un estado eliminando lo que diferenciaba a otros parecidos
    - conservando la transición entre estados (operadores originales)
      EJ: ignorar la mitad de las fichas en el Puzzle-8

  2.- La h' se hace con las distancias calculadas al objetivo desde esos estados
    otro modo es salvar en una BD lo ya calculado en forma de patrones
      empezando desde el objetivo hacia atrás.

  3.- Calcular 1. y 2. debe ser eficiente (barato)

Generación de heurísticas: 3.- Aprendizaje

3.- Aprendizaje: Otra posibilidad es aprender de la experiencia
- EJ: La "experiencia" aquí significaría resolver muchos 8-puzzles
- Cada solución óptima proporciona ejemplos a partir de los cuales
  - se puede utilizar aprendizaje inductivo para construir una heurística
- Estos métodos trabajan mejor cuando se les suministran
  - características de cada estado que sean relevantes para su evaluación.
- Suelen utilizar combinación lineal de estas características: $c_1x_1(n) + c_2x_2(n)$

- Un problema con la generación de nuevas heurísticas es que a menudo no se consigue encontrar una heurística "claramente mejor"
  - Si tenemos varias heurísticas y ninguna domina a todas las demás, podemos definir otra $h'(n) = max \{ h'_1(n), h'_2(n), \ldots , h'_n(n) \}$

Funciones heurísticas: Pasos para diseñar h’

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

- Características
  - Dínamicas : qué cambia entre estados
  - Estáticas: se mantienen igual entre estados
  - Si las relajamos, qué carácterísticas 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
    - en que podamos resolver y calcular 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 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 admisible? (o consistente al menos)
5. Si puedes encontrar otra función que “domina” a h’ : usalá
6. Si puedes hacer experimentos estudia que r* sea cercano a 1

Resolución de problemas y espacio de búsqueda

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

-- 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
    - para los cuales son inadecuados algoritmos sistemáticos (A*)

Métodos informados de optimización local

- Algoritmos de búsqueda local reuelven 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 o ascensión de colinas)*--

- 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

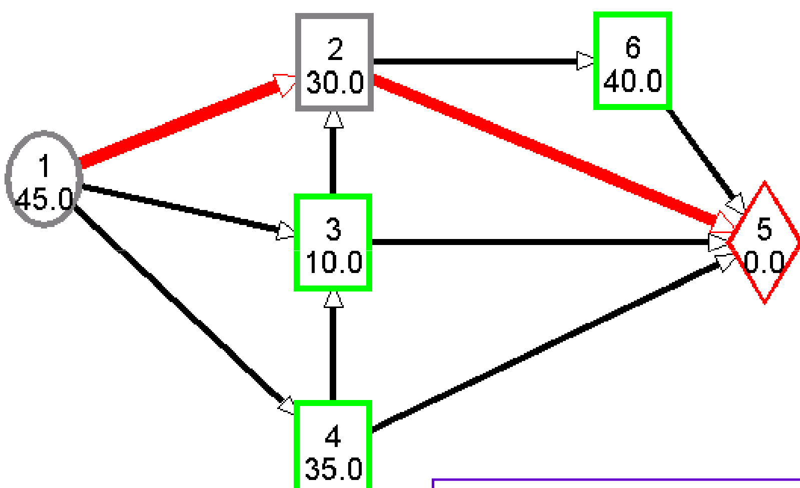
Escalada simple *(hill climbing)*

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

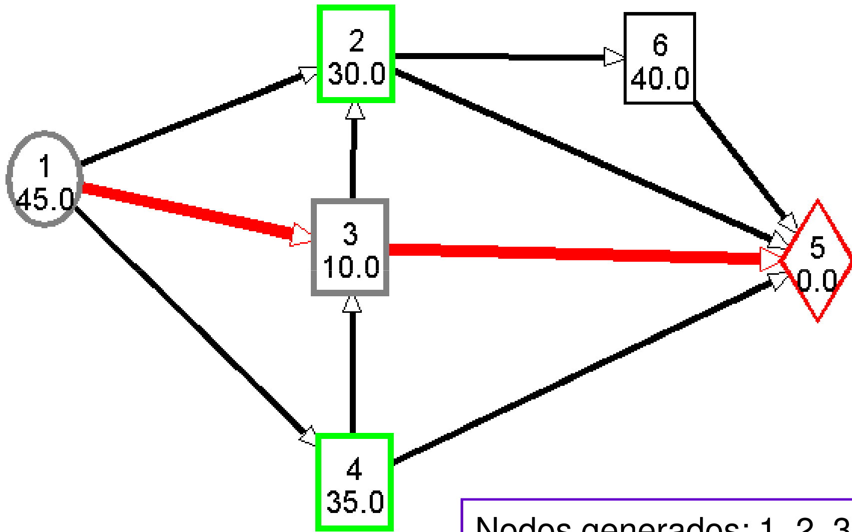
--- Escalada por máxima pendiente ---

- 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 la llama búsqueda local 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

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 máximo local (*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 máximo 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

Enfriamiento simulado *(simulated annealing, 1983)*

- 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 *(h_b)*

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 funda en un estado cristalino de energía baja

Enfriamiento simulado

- Es como un problema de minimización (de energía): la función a optimizar
  - 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
- 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

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**