Resolución de problemas y espacio de búsqueda

- **Tema 2: Resolución de problemas y espacio de búsqueda**
  - Representación de problemas según el paradigma del espacio de estados
  - Estrategias de búsqueda
    - Métodos no informados o ciegos
    - Métodos informados o heurísticos

Resolución de problemas y espacio de búsqueda

- **Representación de problemas según el paradigma del espacio de estados**
  - Representación de problemas y búsqueda
  - Paradigma del espacio de estados
  - Ejemplos
    - El 8-puzzle
    - Los misioneros
    - El granjero, el lobo, la col y la cabra
    - Las garrafas

Representación de problemas y búsqueda

- Los dos elementos básicos para resolver un problema son
  - su representación y
  - la búsqueda de la solución
- Representación de problemas
  - Muchos de los problemas de interés práctico tienen unos espacios tan grandes que no pueden ser representados explícitamente
  - Son necesarios métodos para representar estos espacios de forma implícita y métodos eficientes de búsqueda en estos espacios
- Paradigma del espacio de estados
  - Es el método más empleado en la resolución de problemas
  - Base de la mayoría de los métodos de resolución de problemas en IA
    - Permite una definición formal del problema: convertir una situación dada en otra deseada usando un conjunto de operaciones permitidas
    - Posibilita la resolución por búsqueda: exploración del espacio de estados para encontrar un camino del estado inicial a un estado objetivo

Representación de problemas y búsqueda

- Búsqueda
  - La búsqueda impregna toda la IA y, en particular, la resolución de problemas
  - Es un mecanismo general de resolución de problemas
  - Métodos no informados o ciegos
    - Exploración exhaustiva del espacio de búsqueda hasta encontrar una solución
    - No incorporan conocimiento que guíe la búsqueda
    - La búsqueda no incorpora información del dominio
  - Métodos informados o heurísticos
    - Exploración de los caminos más prometedores
    - Se incorpora conocimiento del problema: “pistas” para acotar el proceso de búsqueda y hacerlo más eficiente

Paradigma del espacio de estados

Representación de problemas como espacios de estados

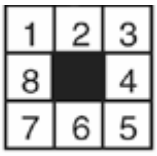
- Estado inicial
- Operadores (o función sucesor): descripción de las posibles acciones disponibles desde un estado (transformaciones de estados)
  - Espacio de estados (grafo dirigido: vértices-estados, arcos-acciones)
    - Estados alcanzables desde el estado inicial
    - Definido implícitamente por el estado inicial y los operadores
- Identificación de estados objetivo
- Función de coste de camino *(suma coste operadores empleados)*
  - Solución: camino desde el estado inicial a un estado objetivo
  - Pueden preferirse unos caminos a otros

Tipo de datos PROBLEMA:

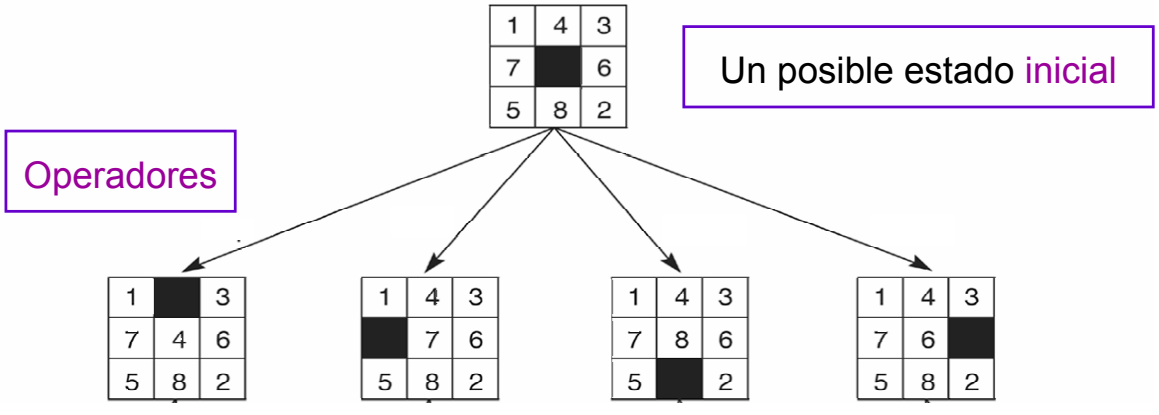
(estado_inicial, operadores, test_objetivo, coste_camino)

Ejemplo: el 8-puzzle

- Simplificación del 15-puzzle
  - Tablero de 3*3
  - 8 fichas numeradas
  - 1 hueco



Estado objetivo



Un posible estado inicial

Operadores

Estados alcanzables en un paso

Representación de problemas

- Para representar un problema, ya sea de juguete o del mundo real, es necesario aplicar abstracción
  - Eliminar detalles irrelevantes en la representación del estado y de las acciones de transformación de estados
  - Dependiendo de la representación, se simplifica o se complica el problema
- Representación del problema del 8-puzzle
  - El estado no debe incluir información sobre el material o el color del tablero, etc. Nada de esto es relevante
  - Describir lo estrictamente necesario para resolver el problema
  - Estados: especificación de la localización de cada ficha y del hueco en cada una de las 9 casillas
  - Estado inicial: puede ser cualquiera
  - Coste del camino: número de pasos
    - Se consigue asumiendo que cada operador tiene coste 1

Niveles de representación

- Niveles de representación
  - Nivel conceptual: se especifican estados y operadores, sin hacer referencia a estructuras de datos o algoritmos que vayan a usarse
  - Nivel lógico: se elige una estructura de datos para los estados y se determina el formato de codificación de los operadores
  - Nivel físico: para el lenguaje de programación elegido se concreta la implementación de estados y operadores determinada en el nivel lógico
- Representación del problema del 8-puzzle
  - Nivel conceptual
    - Estados: localización de cada ficha y del hueco en cada una de las 9 casillas
  - Nivel lógico
    - Muchas opciones: matriz 3*3, vector de longitud 9, conjunto de hechos {(superior_izda = 3), (superior_centro = 8), ...}
  - Nivel físico
    - Es más un problema de programación que de IA

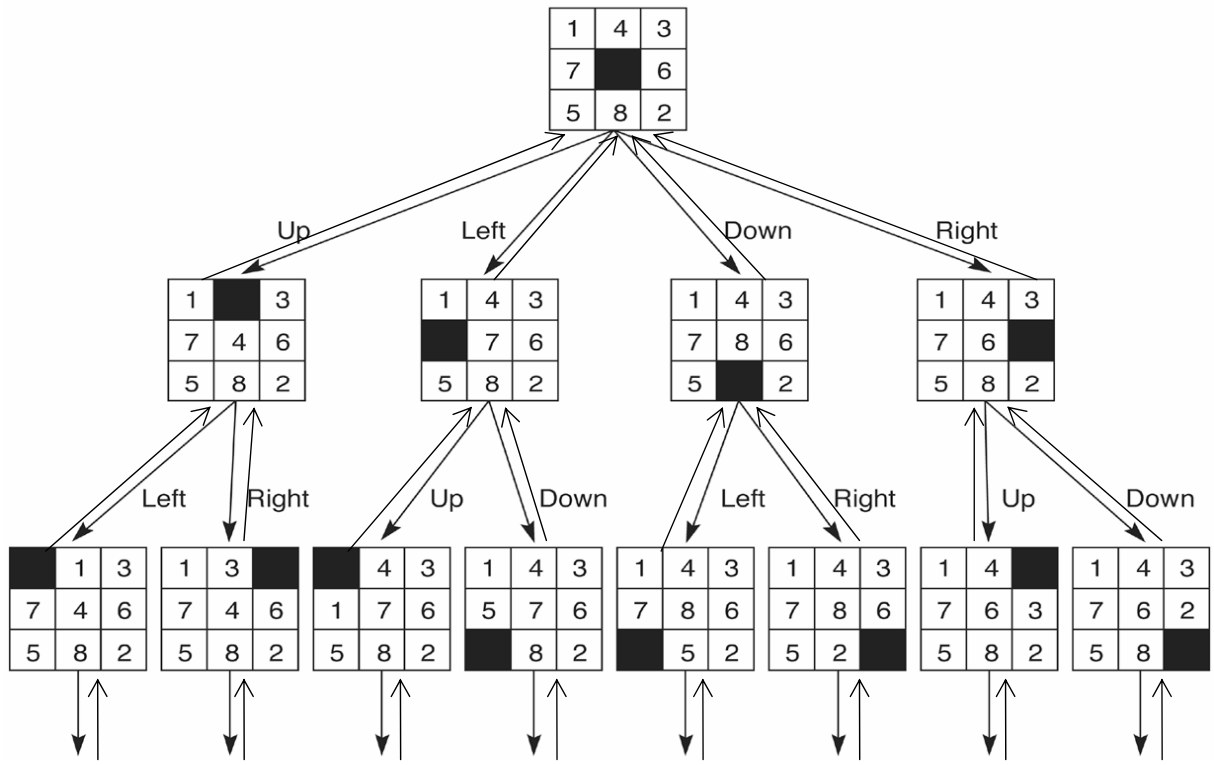
Paradigma del espacio de estados

- La elección de la representación para los estados está muy ligada a la representación de los operadores, puesto que ambos elementos deben "cooperar" para la resolución del problema

- Los operadores deben ser tan generales como sea posible para reducir el número de reglas distintas. Y deben ser deterministas *(ha de saberse de antemano cómo será el estado resultante después de aplicarlos)*
  - 9!*4 operadores para pasar de un estado cualquiera (hay 9! estados en el espacio de estados) a sus estados sucesores (máximo 4) *-- no general*
  - 8*4 operadores que mueven cualquier ficha (hay 8) arriba, abajo, derecha o izquierda *-- preferible pero puede mejorarse*
  - 4 operadores que mueven el hueco arriba, abajo, derecha o izquierda

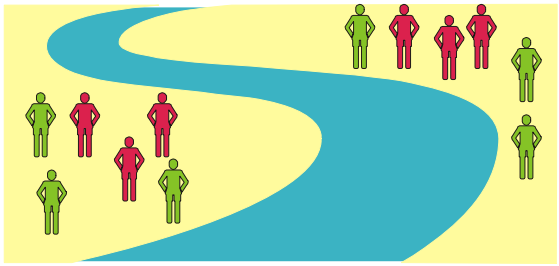
- La elección de una representación para los estados y los operadores define implícitamente una relación de adyacencia en el conjunto de estados factibles para el problema: el espacio de estados

Fragmento del espacio de estados del 8-puzzle



Ejemplo: los misioneros y los caníbales

- Descripción
  - 3 misioneros y 3 caníbales en la orilla de un río junto con 1 bote
  - El objetivo es que pasen todos a la otra orilla
  - Hay dos restricciones
    - Deben cruzar usando el bote en el que sólo pueden ir 1 o 2 personas
    - En ninguna de las orillas puede haber más caníbales que misioneros



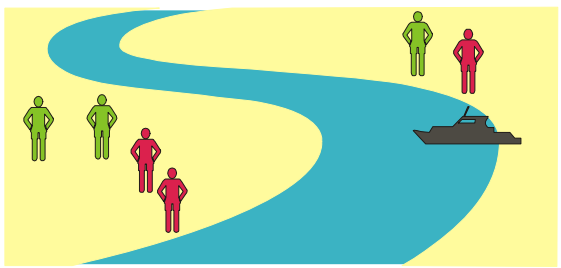
- Representar el problema según el paradigma del espacio de estados y dibujar el espacio de estados

Ejemplo: los misioneros y los caníbales

- Representación
  - Es necesario abstraer y dejar fuera características irrelevantes como intentar identificar a las personas concretas *M1, M2, M3, C1, C2, C3*
- Nivel conceptual
  - 7 "personajes" → ¿guardamos la posición de todos?
  - Estado = nº de misioneros, caníbales y bote en cada orilla
- Nivel lógico: posibilidades
  - *(M1, M2, M3, C1, C2, C3, B)*
  - *(NM_OI, NC_OI, NM_OD, NC_OD, B)*

+ = 3

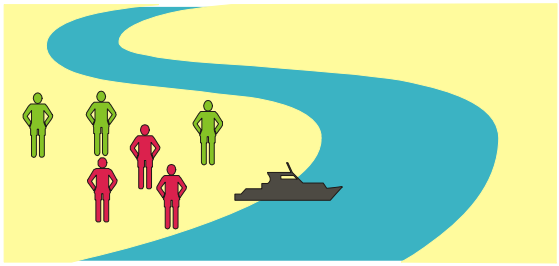
(2, 2, 1, 1, derecha)



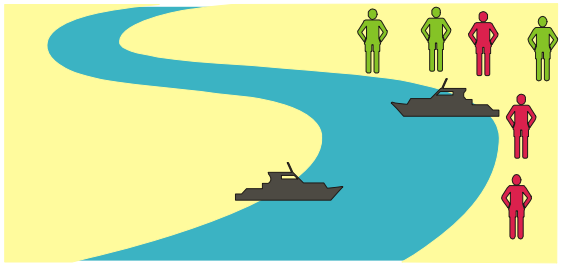
Ejemplo: los misioneros y los caníbales

- Nivel conceptual
  - Estado = nº de misioneros, caníbales y bote en la orilla de partida
  - Acordamos que la orilla inicial sea el margen izquierdo del río
- Nivel lógico
  - Estado = *(NM, NC, B)*
    - *NM* es el número de misioneros en la orilla izquierda *(0, 1, 2 o 3)*
    - *NC* es el número de caníbales en la orilla izquierda *(0, 1, 2 o 3)*
    - *B* es la posición del bote *(0 o 1)*
- El sitio donde está el bote es fundamental para los viajes. Determina si son o no aplicables ciertos operadores
  - $(2, 1, 0) \neq (2, 1, 1)$
- Función de coste de camino = número de veces que se cruza el río

Ejemplo: los misioneros y los caníbales

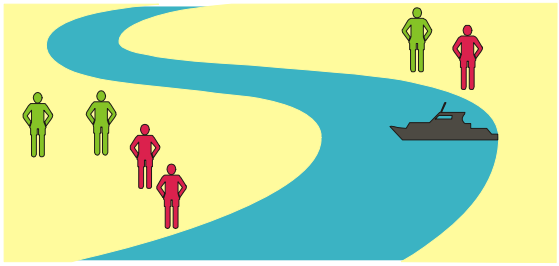


Estado inicial (3, 3, 1)



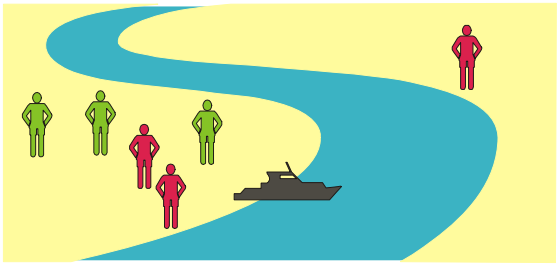
Estado objetivo (0, 0, 0)

(0, 0, 1) no es posible



(2, 2, 0)

Estados intermedios no peligrosos



(3, 2, 1)

Ejemplo: los misioneros y los caníbales

- Operadores *¿Qué determina un cambio de estado?*
  - Hay 5: el bote siempre cruza el río junto a 1 o 2 personas
    - 1 misionero: *M*
    - 2 misioneros: *MM*
    - 1 caníbal: *C*
    - 2 caníbales: *CC*
    - 1 misionero y 1 caníbal: *MC*
- Especificación de operadores
  - El sitio donde está el bote es fundamental
    - P.ej., no podría cruzar ningún misionero en los estados *(0, _, 1)* y *(3, _, 0)*
  - *cruzaM (NM, NC, B)*
    - Precondiciones: $\{ (NM > 0 \wedge B = 1) \vee (NM < 3 \wedge B = 0) \}$
    - Acciones: *si B = 1 entonces NM := NM-1* $\wedge$ *B := 0* $\rightarrow$ *(NM-1, NC, 0)*
      si B = 0 entonces NM := NM+1 $\wedge$ *B := 1* $\rightarrow$ *(NM+1, NC, 1)*

Ejemplo: los misioneros y los caníbales

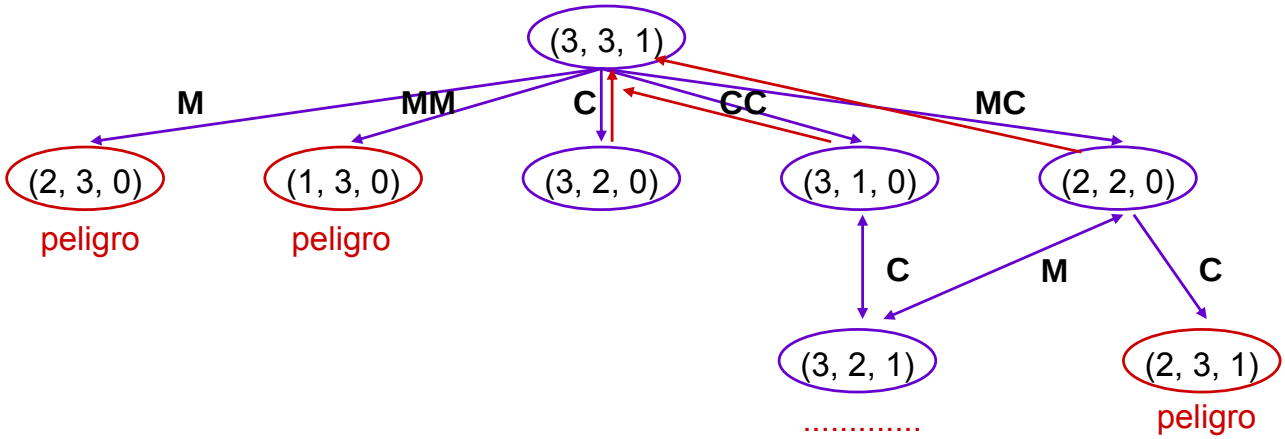
- Situaciones de peligro *¿en bote? ¿en orillas?*
  - En el bote no hay peligro (por viajar un máximo de 2 personas en él)
    - Si el máximo fuese 3 o 4, sí habría peligro *(y sería otro problema)*
  - Hay que comprobar la condición de peligrosidad en las orillas (en estados)
- Estudio de la peligrosidad de un estado *(NM, NC, B)*
  - *(3, 3, 1)* y *(2, 2, 0)* no son estados peligrosos
  - *(1, 2, 0)* y *(2, 3, 0)* son ejemplos de estados peligrosos
  - ¿Bastará *NM < NC* como condición de peligrosidad?
    - En *(2, 1, 0)* ¿no hay peligro?
      - ¡En la orilla derecha hay 1 misionero con 2 caníbales!
    - En *(0, 2, 1)* ¿hay peligro?
      - ¡Si no hay misioneros no hay peligro!
- Condición de peligrosidad
  - $(NM < NC \wedge NM \neq 0) \vee (NM > NC \wedge NM \neq 3)$

Ejemplo: los misioneros y los caníbales

- Espacio de estados
  - En principio, habría 4*4*2 = 32 estados posibles *(NM, NC, B)*
    - Hay 4 estados inalcanzables (por lo tanto, hay 28 estados alcanzables)
      - Obvios como (0, 0, 1) y (3, 3, 0)
      - Y no tan obvios como (3, 0, 1) y (0, 3, 0)
    - Hay 12 estados de peligro
      - (1, 2, _), (2, 3, _), (1, 3, _), (2, 1, _), (2, 0, _), (1, 0, _)
    - Hay 16 estados alcanzables seguros
  - Por lo tanto, el espacio de estados se compone de 12+16 = 28 estados
  - A veces la condición de peligrosidad aparece en la especificación de los operadores
    - Supondría que los estados de peligro no se generarían al aplicar un operador
      - Por lo tanto, en ese caso, el espacio de estados estaría compuesto por 16 estados
  - Nosotros la especificaremos aparte en la mayoría de las ocasiones
    - Impone condiciones sobre el nuevo estado y no sobre el estado actual

Ejemplo: los misioneros y los caníbales

- Espacio de estados



  - Desarrollad vosotros el resto del espacio de estados
    - A partir del estado (3, 2, 1)
    - Grafo dirigido con ciclos
    - Estados de peligro: peligro
      - No se resuelve el problema, por lo que no se sigue por ahí

Ejemplo: los misioneros y los caníbales

- Nivel físico o nivel de implementación: en Prolog
  - Estado inicial con predicado *inicial/1*

```
inicial(estado(3, 3, 1)).
```

  - Estado objetivo con predicado *objetivo/1*

```
objetivo(estado(0, 0, 0)).
```

  - Condición de peligrosidad con predicado *peligro/1*

```
peligro(estado(NM, NC, _)) :-
        (NM < NC, NM =\= 0) ; (NM > NC, NM =\= 3).
```

  - Operadores con predicado *movimiento/4*

```
movimiento(estado(NM, NC, B), estado(NNM, NC, NB), 1,
                                                cruzaM) :-
   ((NM > 0, B = 1) ; (NM < 3, B = 0)),
   ((B = 1) -> (NNM is NM-1) ; (NNM is NM+1)),
   opuesta(B, NB), \+(peligro(estado(NNM, NC, NB))).
```


Representación: pensar en estados y operadores

- La representación de los estados afecta a la facilidad/dificultad de la especificación de operadores
- Ejemplo de los misioneros
  - *(M1, M2, M3, C1, C2, C3, B)*
    - *cruzaM1 (M1, M2, M3, C1, C2, C3, B)*
      - Precondiciones: *{ M1 = B }*
      - Acciones: *si B = izquierda entonces M1 := derecha ∧ B := derecha*
        si B = derecha entonces M1 := izquierda ∧ B := izquierda
    - Habría que especificar 21 operadores
      - 3 para cruzar a un misionero
      - 3 para cruzar a un caníbal
      - 3 para cruzar a dos misioneros
      - 3 para cruzar a dos caníbales
      - 9 para cruzar a un caníbal y a un misionero
  - *(NM_OI, NC_OI, NM_OD, NC_OD, B)*
    - La información redundante supone hacer cambios en más componentes

Representación: pensar en estados y operadores

- La representación de los estados afecta a la facilidad/dificultad de la especificación de operadores
- Ejemplo del 8-puzzle
  - La representación de los estados a nivel conceptual estaba clara
  - Entre las posibilidades para los operadores
    - Centrarse en los estados era implanteable
      - 9!*4 = 1.451.520 operadores
    - Centrarse en la ficha a mover era factible
      - 8*4 operadores
      - Puede parametrizarse la especificación a nivel lógico dependiendo de cómo se haga la representación de estados
        - Debe facilitar localizar y cambiar cuál es la posición de una ficha concreta y del hueco
        - *izquierda(Ficha), derecha(Ficha), arriba(Ficha), abajo(Ficha)*
      - Aunque así sea más fácil, seguirían saliendo 32 operadores
    - Llegamos a la conclusión de que lo mejor era centrarse en el hueco
      - Salían 4 operadores

http://freeweb.siol.net/danej/riverIQGame.swf

http://freeweb.siol.net/danej/riverIQGame.swf

- Test japonés
  - Todo el mundo tiene que cruzar el río
  - Sólo 2 personas pueden cruzar a la vez
  - El padre no puede permanecer con ninguna de sus hijas sin que esté presente la madre
  - La madre no puede permanecer con ninguno de sus hijos sin que esté presente el padre
  - El ladrón no puede permanecer con ningún miembro de la familia sin que esté presente el policía
  - Sólo saben manejar la balsa la madre, el padre y el policía: sin uno de ellos a bordo, la balsa no se mueve
- Aplicación: para empezar haz clic sobre el círculo azul
  - Para mover las personas haz clic sobre ellos
  - Para que la balsa cruce el río, haz clic sobre la pala del otro lado

Ejercicio: el granjero, el lobo, la cabra y la col

- El problema del granjero, el lobo, la cabra y la col
  - Un granjero, un lobo, una cabra y una col se encuentran en la orilla izquierda de un río
  - El objetivo es que pasen a la orilla derecha del río
  - Las restricciones son:
    - deben cruzar en una barca
    - la barca debe ser tripulada por el granjero
    - la barca sólo tiene capacidad para un pasajero más
    - el lobo se comerá al la cabra si se los deja juntos sin compañía en una de las orillas (sin granjero; la col no lo evita)
    - la cabra se comerá la col si se los deja solos (sin granjero)
- Ejercicio:
  - Representación de estados (incluyendo el inicial y el objetivo)
  - Condición de peligrosidad
  - Especificación de operadores y dibujar espacio de estados

Ejercicio: las garrafas de 4 y 3 litros

- 2 garrafas vacías con capacidades de 4 y 3 litros, respectivamente
- Objetivo: la garrafa de 4 litros ha de contener exactamente 2 litros
- Medios: grifo para rellenarlas y posibilidad de trasvasar líquido de una garrafa a la otra, hasta que la 1ª se vacíe o la 2ª se llene

- Nivel conceptual
  - Estados: líquido que contienen las garrafas de 4 y 3 litros
  - ¿Operadores? *(p.e. llena-4: llenar del grifo la garrafa de 4 litros*)
- Nivel lógico
  - Estados: pares *(x, y)* donde *x* es el nº de litros que contiene la garrafa de 4 litros e *y* es el nº de litros en la garrafa de 3 litros
    - Estado inicial: *(0, 0)* ¿Objetivo?
  - Operadores como reglas (precondición y acción asociada)
    - *llena-4 (x, y)*: ¿Resto operadores?
      - precondición: *x < 4 (si no, sería un movimiento absurdo sin cambio de estado)*
      - acción: construir el estado *(4, y)*

Ejercicios de representación *(en el campus virtual)*

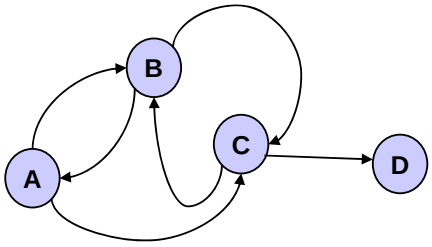
- Hoja 1 de ejercicios:
  - Las garrafas *(ejercicio 1)*
  - El granjero, el lobo, la cabra y la col *(ejercicio 2)*
  - El juego del 15 *(ejercicio 13)*
- Ejercicios extra de representación:
  - Os dejaré en el campus más ejercicios de representación, en concreto alguno más de exámenes para que tengáis más material para darle vueltas
- Resolved la parte de representación según el paradigma del espacio de estados de estos ejercicios
  - No dejaremos soluciones
    - Son para que los hagáis y discutirlos en clase

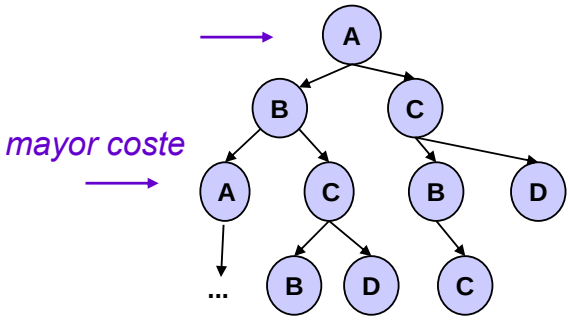
Resolución de problemas y espacio de búsqueda

- **Búsqueda**
  - Introducción
    - Espacio de búsqueda
    - Esquema general de búsqueda
    - Evaluación de estrategias de búsqueda
  - Métodos no informados o ciegos
  - Métodos informados o heurísticos

Espacio de búsqueda (o árbol de búsqueda)

- Árbol formado por los nodos generados en la búsqueda de la solución
  - Depende del algoritmo de búsqueda utilizado
  - Aunque el espacio de estados sea finito, el espacio de búsqueda puede ser infinito (grafo convertido a árbol) por los posibles ciclos del grafo
    - Nodos distintos del árbol de búsqueda pueden corresponderse con el mismo estado del espacio de estados





  - Un nodo representa un camino desde la raíz hasta un cierto estado

Tipo de datos NODO:

(estado, nodo_padre, operador, profundidad, coste_camino)

Espacio de búsqueda

- Evitar la repetición de estados (ciclos del grafo) tiene un coste. Puede hacerse a distintos niveles
  - Evitar aplicación sucesiva de operadores inversos *(si hay)* => bajo coste
  - Evitar ciclos en el camino actual (guardando estados del camino actual)
  - Evitar la repetición de cualquier estado
    => mayor coste espacio-tiempo para hacer las comparaciones
    - Marcar los estados que han sido generados para comprobar la no repetición
- Hay que llegar a un compromiso entre lo que se intenta evitar y el coste de evitarlo (cuanto más ciclos, más justificada)
- A diferencia de la teoría algorítmica de grafos, la búsqueda en espacios de estados
  - No supone que el grafo esté previamente generado
  - Ni asume tampoco que se tengan que generar todos los estados
    - Imprescindible para el tipo de problemas de IA
    - El grafo del 8-puzzle tiene 9! = 362.880 vértices...

Esquema general de búsqueda

```
función BÚSQUEDA_GENERAL (problema, estrategia)
   devuelve solución o fallo    % o cicla
     inicializar abiertos con
        nodo(estado_inicial del problema)        % generación nodo
     repetir
          si abiertos = Ø
             entonces devolver fallo
          extraer nodo de abiertos según estrategia
             % el orden de extracción lo determina la estrategia
          si nodo contiene estado_objetivo
             entonces devolver solución
             si no                    % aquí se controlaría si repetidos
                  expandir nodo % generar los nodos hijos
                  añadir todos sus hijos a abiertos
                  % orden de aplicación de operadores (generación)
```

Terminología

- Abiertos
  - En la estructura *abiertos* se van guardando los nodos correspondientes a estados generados pero pendientes de ser expandidos
    - Al expandir un nodo se saca de *abiertos*
    - Dependiendo de la estrategia, la gestión de *abiertos* cambiará
- Tipos de estados/nodos
  - Generados
    - Son aquéllos que aparecen o han aparecido en nodos de *abiertos*
  - Generados pero no expandidos
    - Son aquéllos que aparecen en nodos de *abiertos*
  - Expandidos
    - Un estado se expande cuando un nodo que lo representa sale de *abiertos*, se generan todos sus descendientes y éstos se añaden a abiertos
    - Un estado puede ser expandido varias veces pero un nodo se expande una única vez

Evaluación de estrategias de búsqueda

- Criterios para la evaluación de estrategias de búsqueda
  - Completitud: ¿garantiza encontrar solución si la hay?
  - Optimalidad: ¿encuentra la solución de coste mínimo?
  - Complejidad en tiempo: ¿cuánto tarda en encontrar una solución? (simplificando, nº de nodos expandidos)
  - Complejidad en espacio: ¿cuánta memoria se necesita? (simplificando, máximo nº de nodos simultáneamente en memoria)
- Coste total: 2 componentes
  - Coste de la solución: coste del camino encontrado (a veces, de la solución en sí misma –problemas de optimización)
  - Coste de la búsqueda de la solución: complejidad del algoritmo utilizado
- Hay que llegar a un compromiso entre ambos costes
  - Obtener la mejor solución posible con los recursos disponibles

Resolución de problemas y espacio de búsqueda

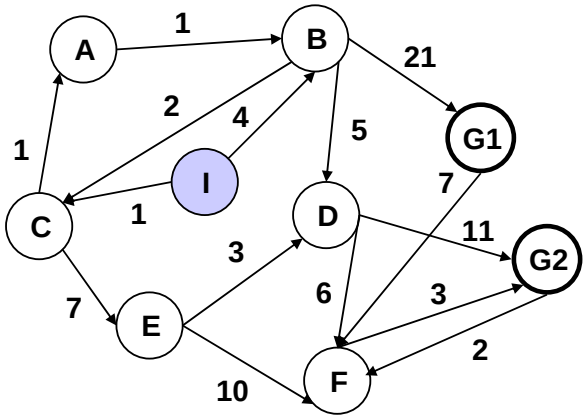
- **Métodos no informados o ciegos**
  - Primero en anchura
  - Coste uniforme
  - Primero en profundidad
  - Profundidad limitada
  - Profundización iterativa
  - Bidireccional

Búsqueda primero en anchura (Moore, 1959)

- Los nodos se expanden por orden no decreciente de profundidad
  - Los nodos de profundidad *p* se expanden antes que los nodos de profundidad *p+1 (por lo que no hay vuelta atrás)*
  - La estructura *abiertos* se implementa con una cola
- Propiedades
  - Completa y *óptima si el coste del camino es una función no decreciente de la profundidad del nodo (el camino de menor longitud puede no ser óptimo)*
  - Complejidad en tiempo y en espacio: $O(r^p)$
    - Suponiendo un factor de ramificación *máximo r* (nº de hijos de un nodo) y un camino hasta la solución de profundidad mínima *p*, el nº de nodos expandidos en el caso peor es $r^0 + r^1 + r^2 + ... + r^p + (r^{p+1} - r)$
  - Problema mayor: la memoria (sólo viable para casos pequeños)
    - Todos los nodos generados han de mantenerse en memoria
- Ciclos: no se pierden soluciones, aunque suponen ineficiencia
  - Si no hay solución, podrían afectar a la terminación

Ejemplo: búsqueda primero en anchura

Espacio de estados

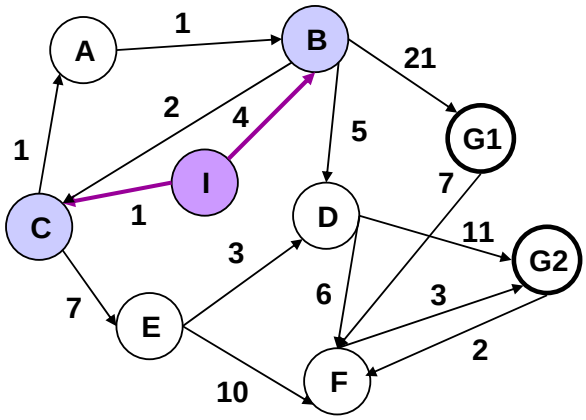


Espacio de búsqueda

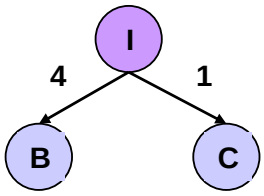
Cola de nodos abiertos: I

Ejemplo: búsqueda primero en anchura

Espacio de estados



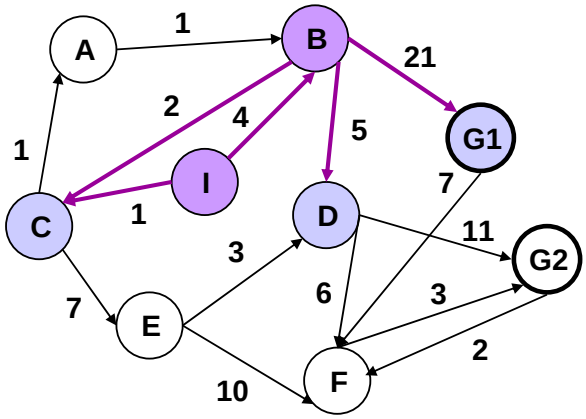
Espacio de búsqueda



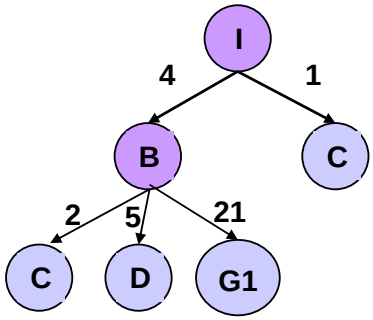
Cola de nodos abiertos: C B

Ejemplo: búsqueda primero en anchura

Espacio de estados



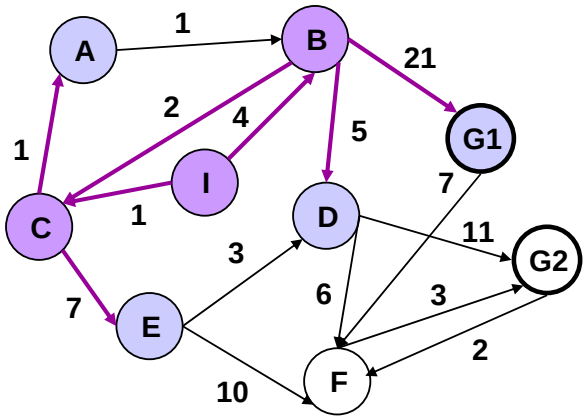
Espacio de búsqueda



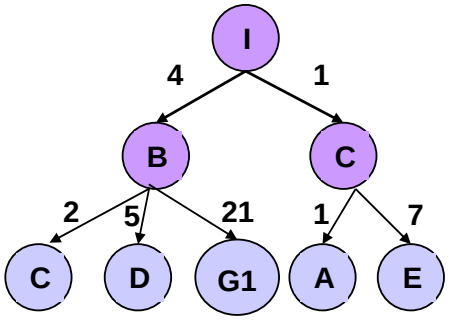
Cola de nodos abiertos: G1 D C C

Ejemplo: búsqueda primero en anchura

Espacio de estados



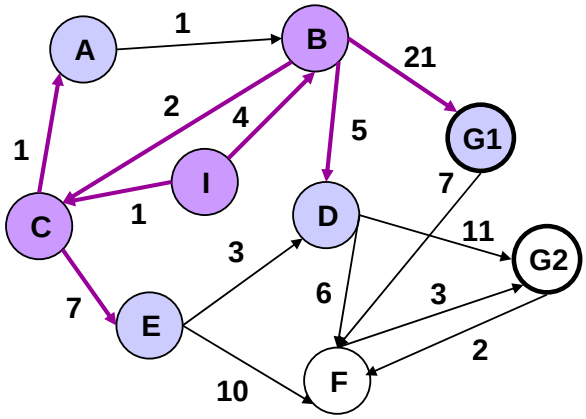
Espacio de búsqueda



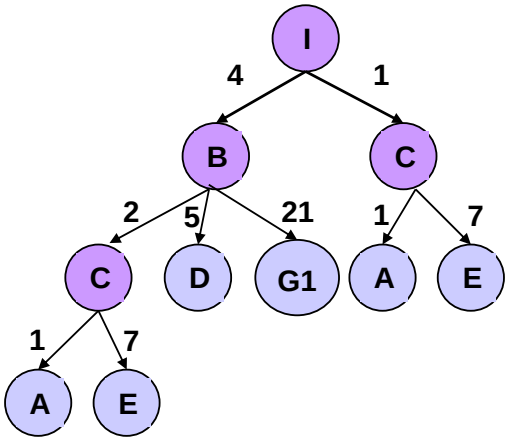
Cola de nodos abiertos: E A G1 D C

Ejemplo: búsqueda primero en anchura

Espacio de estados



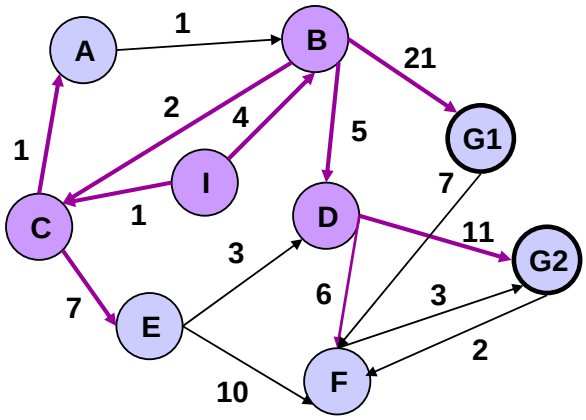
Espacio de búsqueda



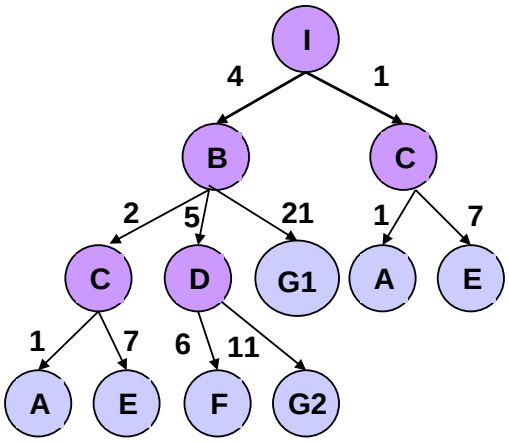
Cola de nodos abiertos: E A E A G1 D

Ejemplo: búsqueda primero en anchura

Espacio de estados



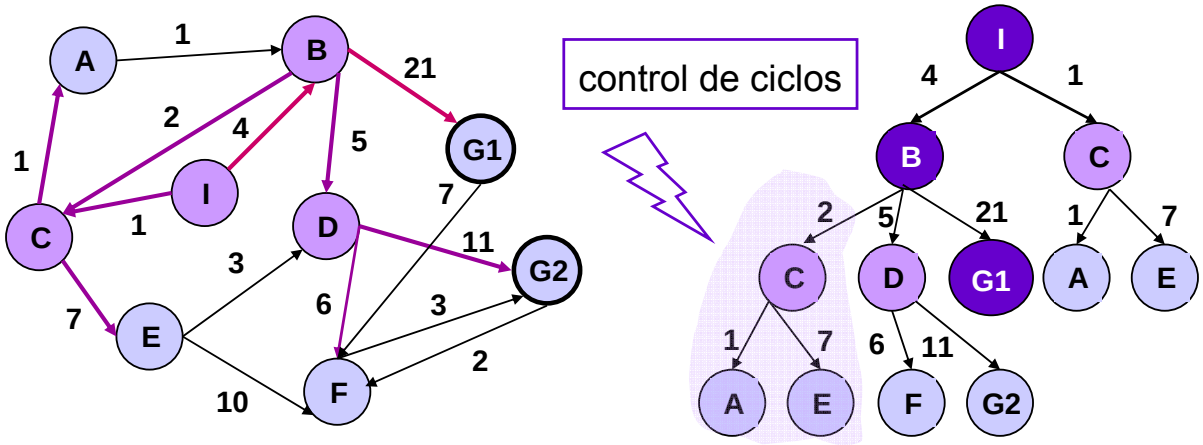
Espacio de búsqueda



Cola de nodos abiertos: G2 F E A E A G1

Ejemplo: búsqueda primero en anchura

Espacio de estados **Espacio de búsqueda**



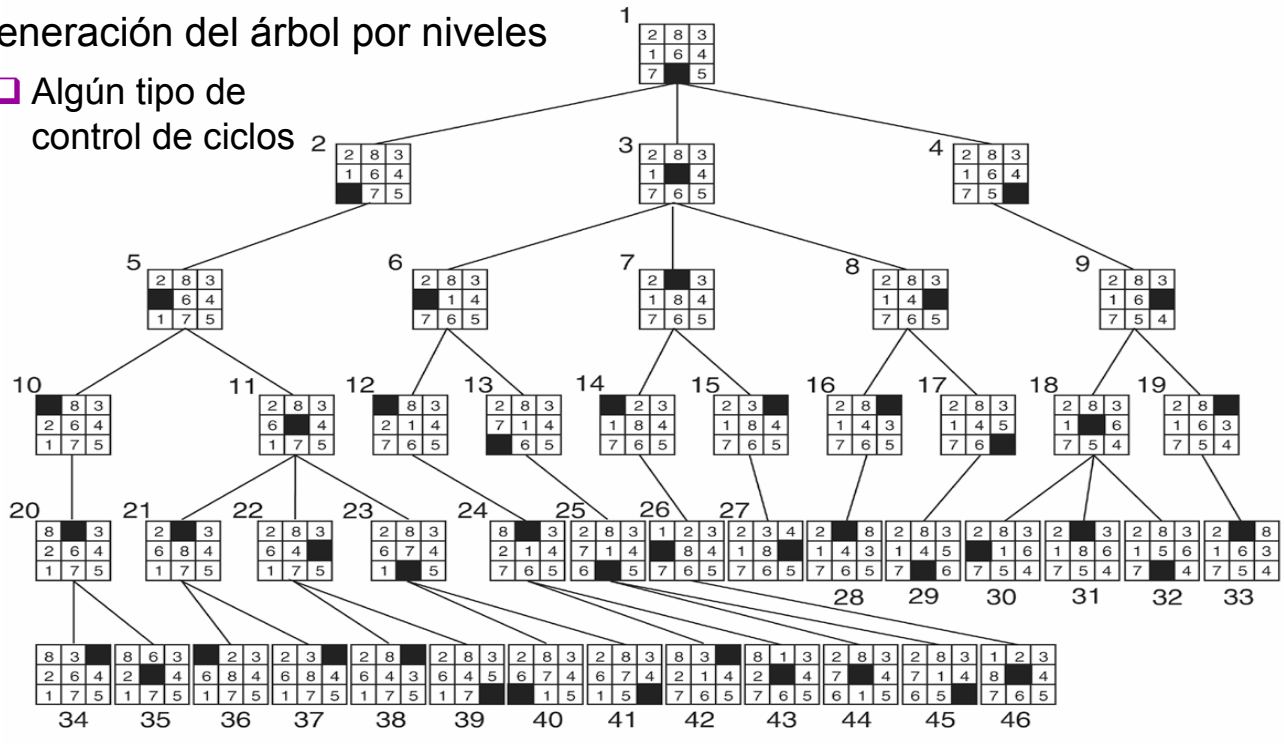
Nodos expandidos (por orden): I B C C D G1

Nodos generados (por orden): I B C C D G1 A E A E F G2

Camino a la solución: I B G1 Coste: 4+21 = 25

Búsqueda primero en anchura (8-puzzle)

- Generación del árbol por niveles
  - Algún tipo de control de ciclos



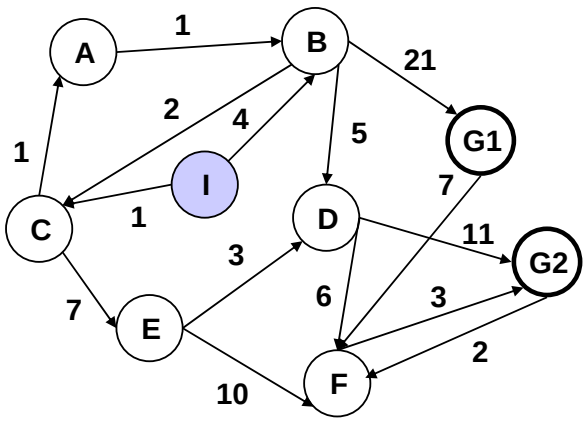
Objetivo

Búsqueda de coste uniforme (Dijkstra, 1959)

- Variante: los nodos se expanden por orden no decreciente de coste (camino de coste menor)
  - Se expanden los nodos de igual coste en lugar de expandir los nodos de igual profundidad (implementación con cola de prioridad)
  - Coste del camino frente al número de pasos
  - Si el coste del camino a un nodo coincide con su profundidad (o es directamente proporcional a la profundidad) entonces esta búsqueda equivale a primero en anchura
- Propiedades:
  - Completa si no existen caminos infinitos de coste finito
  - Óptima si *coste(sucesor(n)) >= coste(n)*
    - Se satisface cuando todos los operadores tienen *coste >= 0*
  - Complejidad en espacio y tiempo equivalente a primero en anchura: $O(r^p)$

Ejemplo: búsqueda de coste uniforme

Espacio de estados

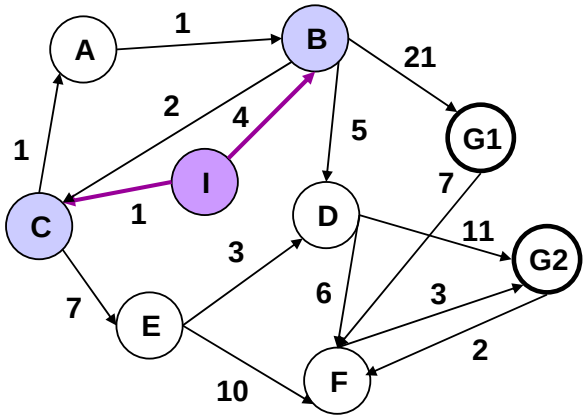


Espacio de búsqueda

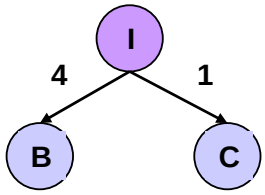
Cola de prioridad de nodos abiertos: I(0)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



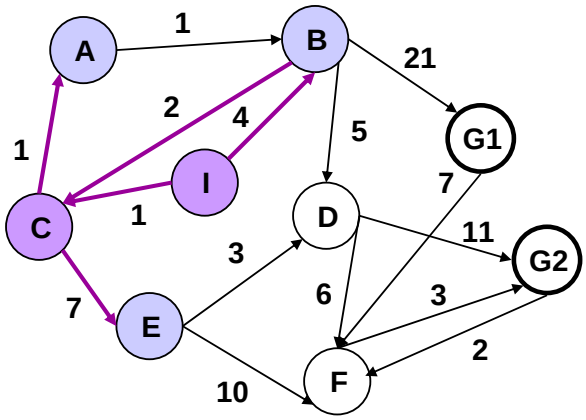
Espacio de búsqueda



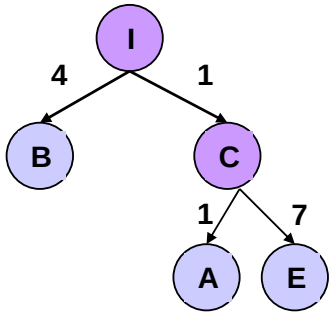
Cola de prioridad de nodos abiertos: B(4) C(1)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



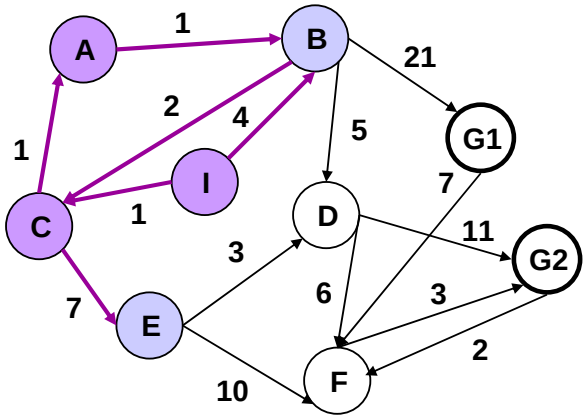
Espacio de búsqueda



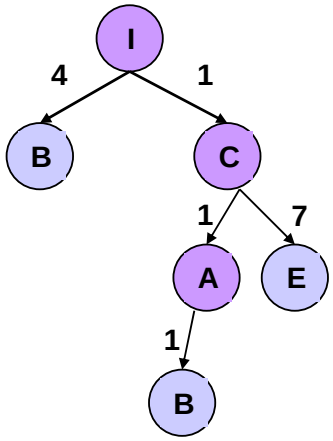
Cola de prioridad de nodos abiertos: E(8) B(4) A(2)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



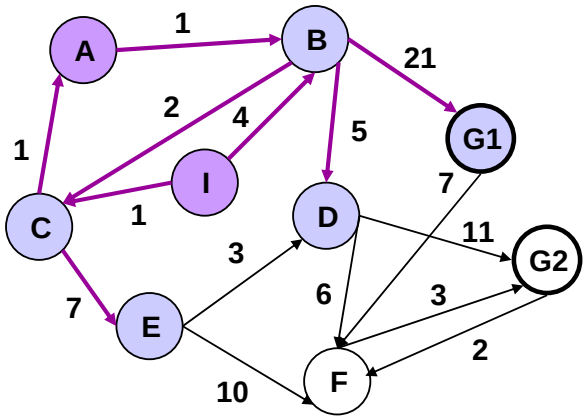
Espacio de búsqueda



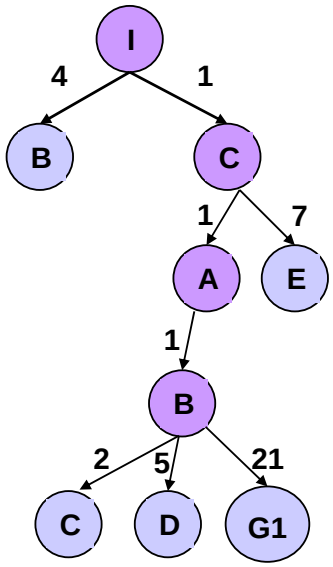
Cola de prioridad de nodos abiertos: E(8) B(4) B(3)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



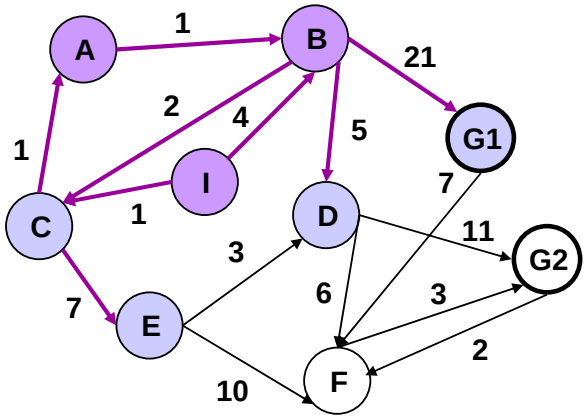
Espacio de búsqueda



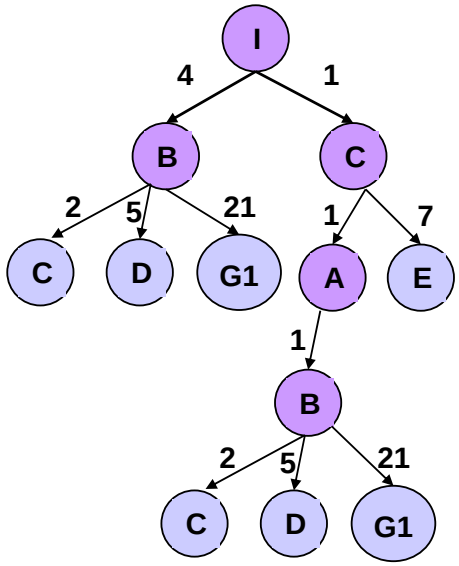
Cola de prioridad de nodos abiertos: G1(24) E(8) D(8) C(5) B(4)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



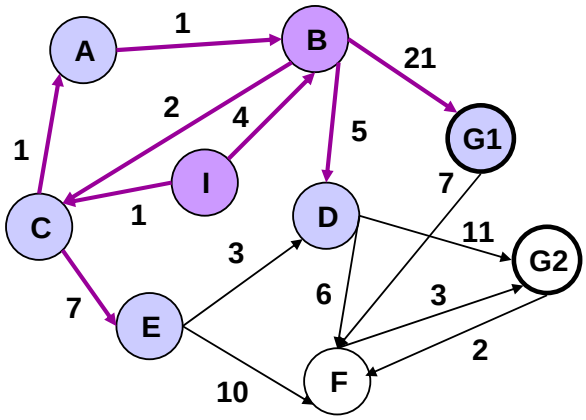
Espacio de búsqueda



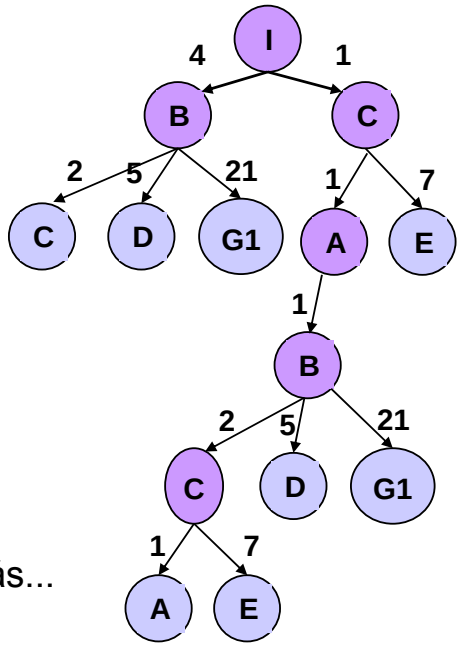
Cola de prioridad de nodos abiertos: G1(25) G1(24) D(9) E(8) D(8) C(6) C(5)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



Espacio de búsqueda

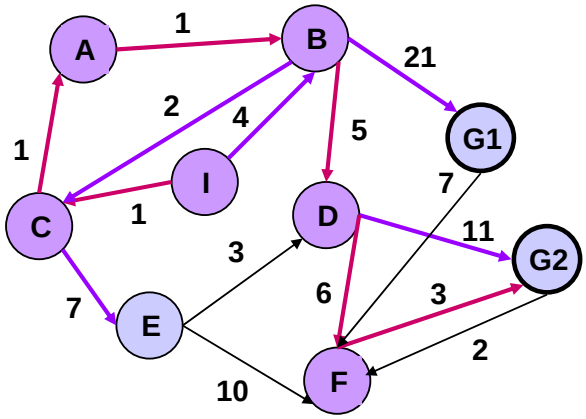


Y así seguiría unos cuantos pasos más...

Cola de prioridad de nodos abiertos: G1(25) G1(24) E(12) D(9) E(8) D(8) C(6) A(6)

Ejemplo: búsqueda de coste uniforme

Espacio de estados



Búsqueda de coste uniforme

- Completa y óptima *(en este caso)*
- Expande muchos nodos
- Aquí es mucho mayor el problema de los ciclos
  - Pero un control de ciclos sobre *abiertos* supondría que la estrategia dejaría de ser óptima...
  - Puede verse en este ejemplo *(el B hijo de A no se generaría y no se encontraría esta solución óptima)*
- La mejor solución se encuentra a profundidad 6

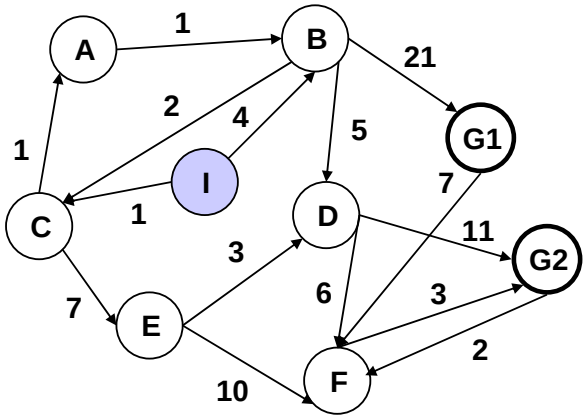
Camino a la solución: I C A B D F G2 Coste: 1+1+1+5+6+3 = 17

Búsqueda primero en profundidad

- Se visita un sucesor del nodo actual en cada paso *(camino)*
  - En cada nodo se tienen que dejar marcados los sucesores no explorados, por si hiciese falta volver a ellos *(vuelta atrás)*
  - Si el nodo actual no tiene sucesores se hace backtracking
  - Implementación: se usa una pila para la estructura *abiertos* *(o bien se implementa usando recursión)*
- Propiedades:
  - No completa: puede meterse en caminos infinitos
  - No óptima: puede haber soluciones mejores por otros caminos; no recomendable si *m* es grande
  - Espacio: si *m* es la máxima profundidad del árbol y *r* el factor de ramificación ➔ $O(r*m)$
    - requisitos modestos: basta el camino actual y los nodos no expandidos
  - Tiempo: $O(r^m)$ *(si hay muchas soluciones, puede ser más rápida que primero en anchura; depende del orden de aplicación de operadores)*

Ejemplo: búsqueda primero en profundidad

Espacio de estados

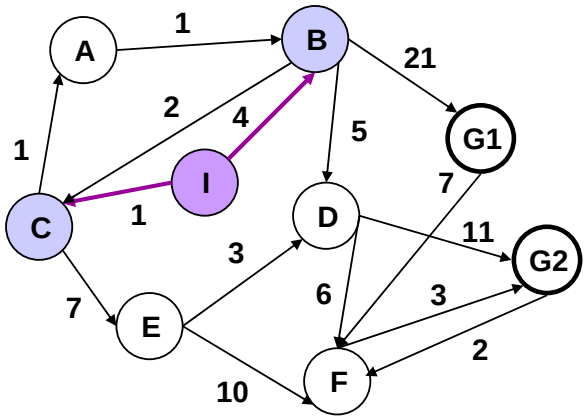


Espacio de búsqueda

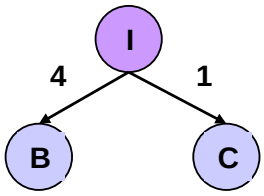
Pila de nodos abiertos: I

Ejemplo: búsqueda primero en profundidad

Espacio de estados



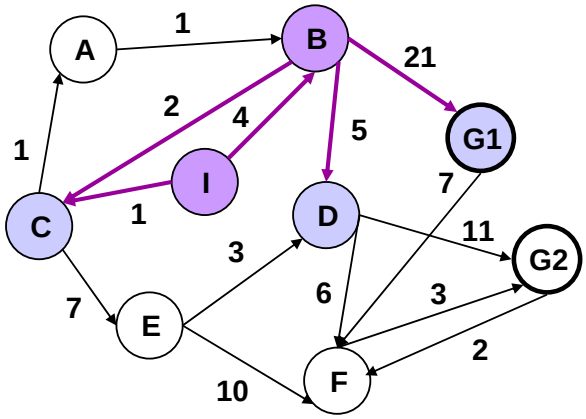
Espacio de búsqueda



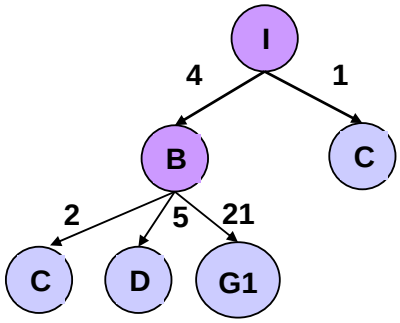
Pila de nodos abiertos: B C

Ejemplo: búsqueda primero en profundidad

Espacio de estados



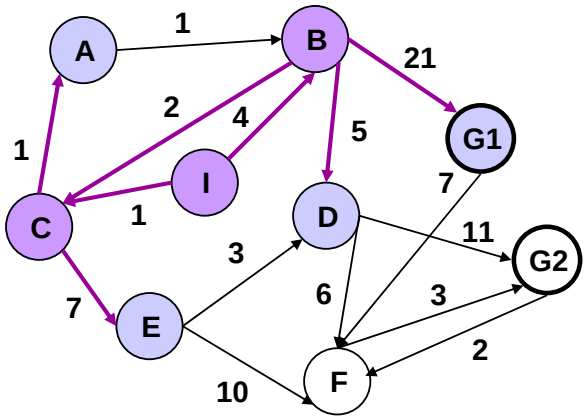
Espacio de búsqueda



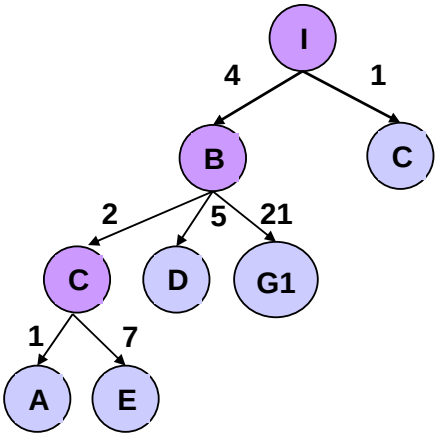
Pila de nodos abiertos: C D G1 C

Ejemplo: búsqueda primero en profundidad

Espacio de estados



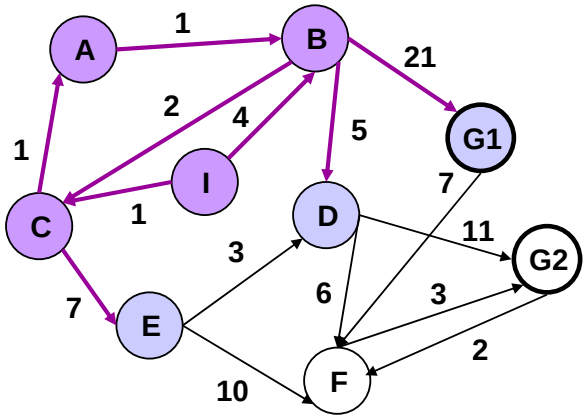
Espacio de búsqueda



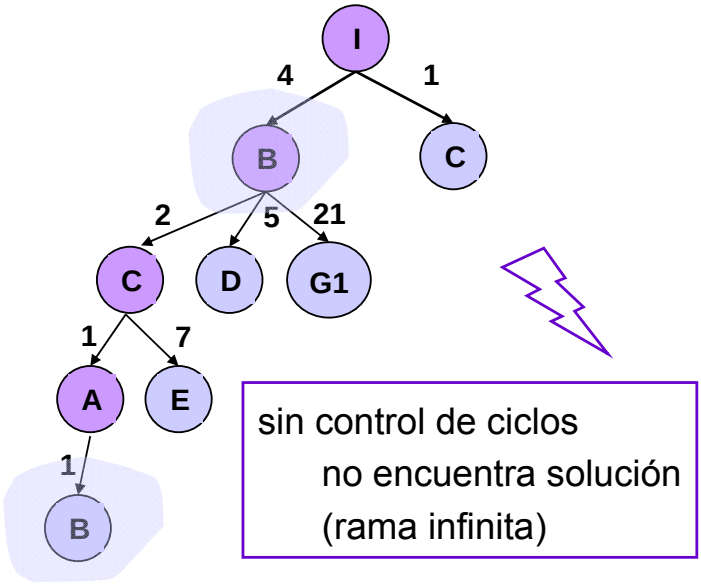
Pila de nodos abiertos: A E D G1 C

Ejemplo: búsqueda primero en profundidad

Espacio de estados



Espacio de búsqueda



sin control de ciclos
no encuentra solución
(rama infinita)

Pila de nodos abiertos: B E D G1 C

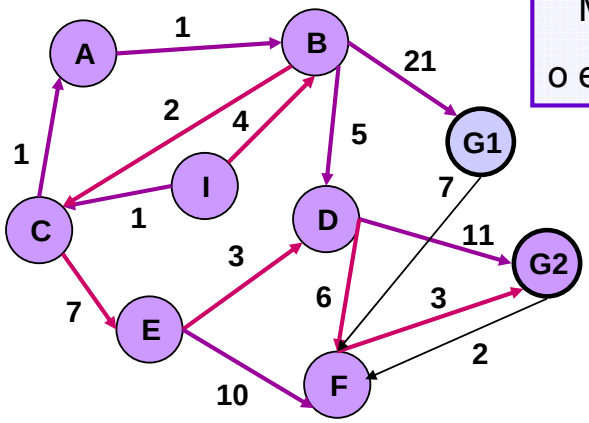
Nodos expandidos: I B C A

Control de ciclos: evitar repeticiones de estados

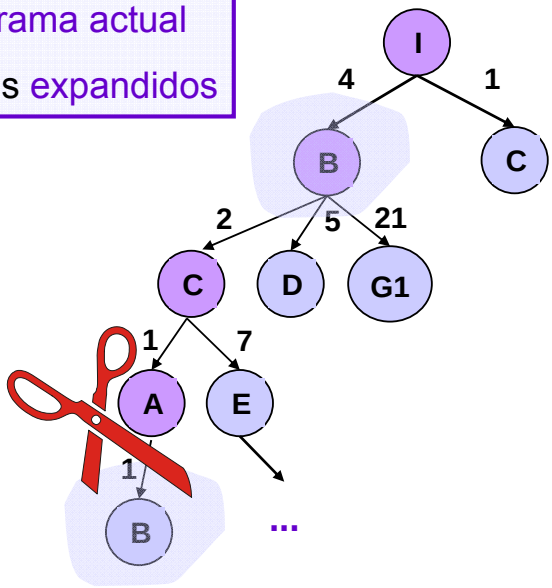
- Cuando el espacio de estados tiene ciclos conviene tener cuidado con ellos (bien sea por eficiencia o por terminación)
  - *"Los algoritmos que olvidan su historia están condenados a repetirla"*
- Pero su control empeora la complejidad de los algoritmos utilizados
  - Mirar los nodos del camino actual es lo más sencillo, lo menos costoso *(cada nodo tiene acceso a su padre)* y lo más seguro
  - Mirar los nodos de abiertos *(los generados aún pendientes de expandir)*
  - Mirar los estados ya expandidos: cerrados
    - Para ello es necesario tenerlos almacenados (estructura *cerrados*)
  - Combinar las 2 últimas comprobaciones
    - Así, cada estado del espacio de estados es examinado, a lo más, una vez
    - Puede suponer una sobrecarga inaceptable o un derroche *(si no es necesario)*
      - Gestión de cerrados *(además, siempre aumenta; nunca se eliminan elementos)*
      - Aplicación de algoritmos de búsqueda en abiertos y cerrados
    - ¡Cuidado! Pueden perderse propiedades de las estrategias

Ejemplo: primero en profundidad + control ciclos

Espacio de estados **Espacio de búsqueda**



Mirar en la rama actual
o en los nodos expandidos



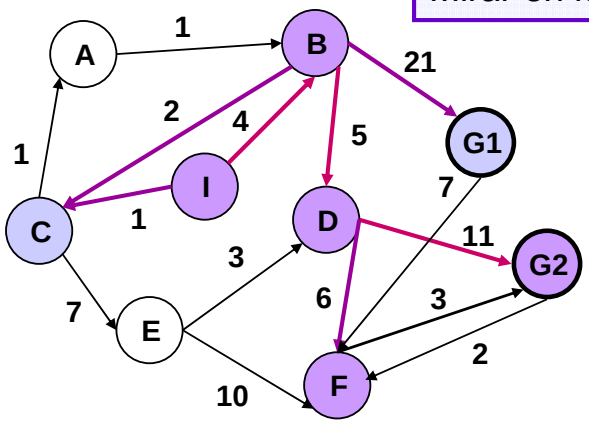
Nodos expandidos: I B C A E D F G2

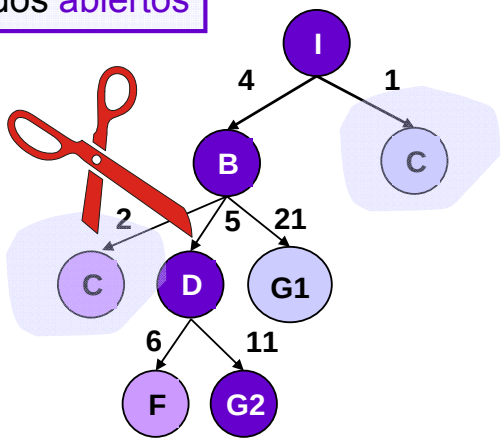
Camino a la solución: I B C E D F G2 Coste: 4+2+7+3+6+3 = 25

Ejemplo: primero en profundidad + control ciclos

Espacio de estados **Espacio de búsqueda**

Mirar en los nodos abiertos





Nodos expandidos (por orden): I B D F G2

Nodos abiertos (por orden): I B C D G1 F G2

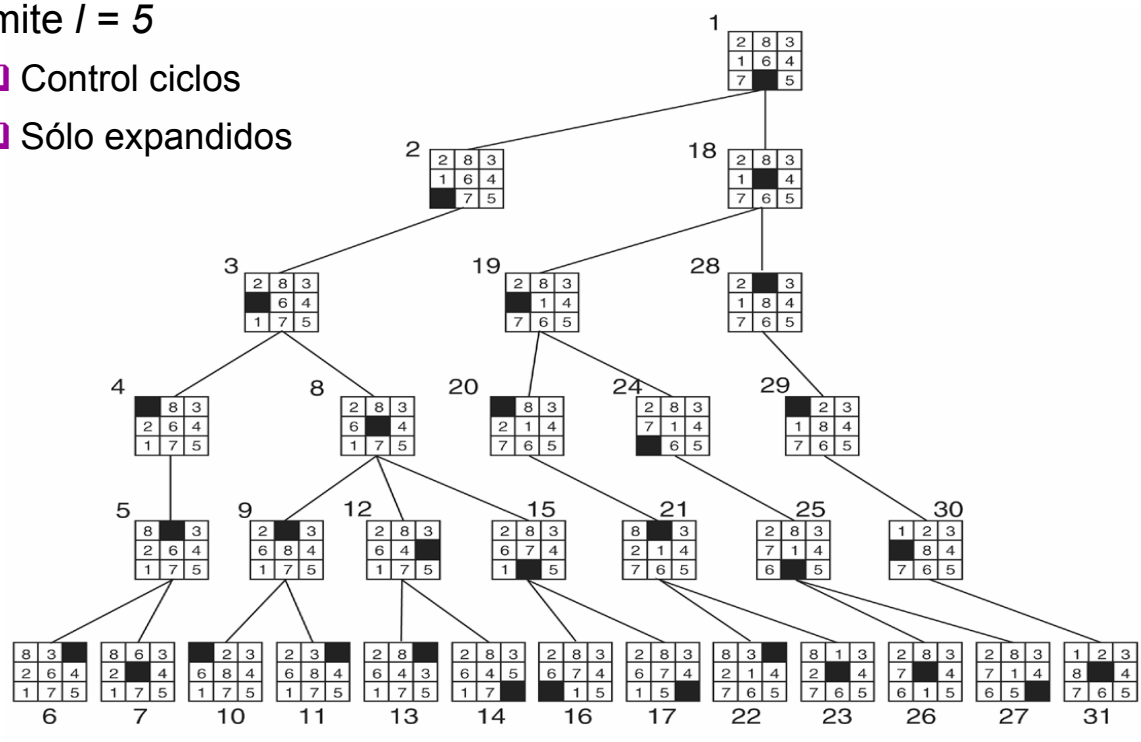
Camino a la solución: I B D G2 Coste: 4+5+11 = 20

Búsqueda de profundidad limitada

- Como la búsqueda en profundidad, pero se fija un límite *l* de profundidad en la búsqueda, para evitar descender indefinidamente por el mismo camino
  - El límite permite desechar caminos en los que se supone que no encontraremos un nodo objetivo lo suficientemente cercano al nodo inicial
- Propiedades:
  - *Completa sólo si l >= p* (*p* profundidad mínima de "la solución")
    - Si *p* es desconocido, la elección de *l* es una incógnita
    - Hay problemas en los que este dato (diámetro del espacio de estados) es conocido de antemano, pero en general no se conoce a priori
  - No óptima
    - No puede garantizarse que la primera solución encontrada sea la mejor
  - Tiempo: $O(r^l)$
  - Espacio: $O(r*l)$

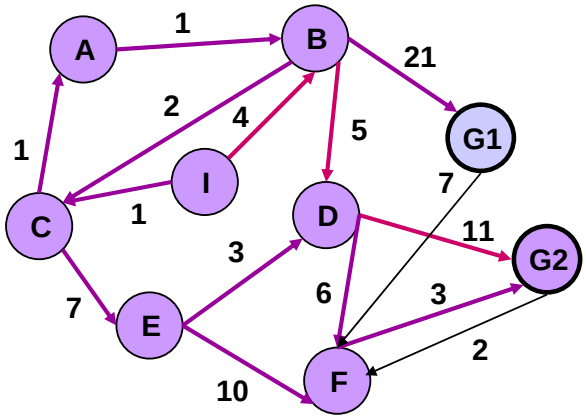
Búsqueda de profundidad limitada

- Límite *l* = *5*
  - Control ciclos
  - Sólo expandidos

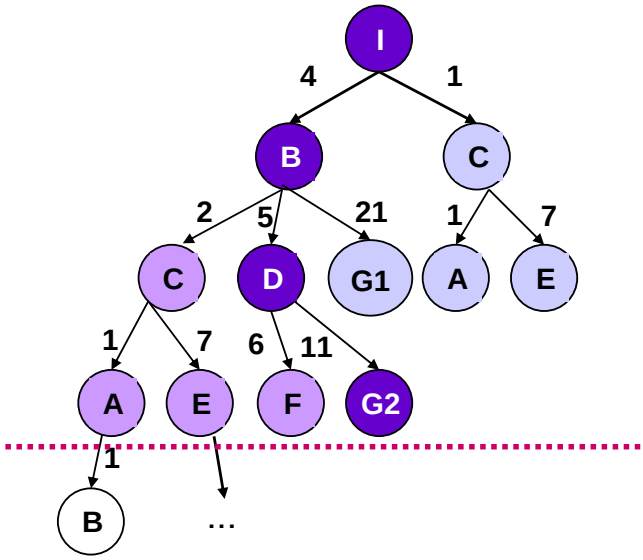


Ejemplo: búsqueda de profundidad limitada *(l = 3)*

Espacio de estados



Espacio de búsqueda



Nodos expandidos: I B C A E D F G2

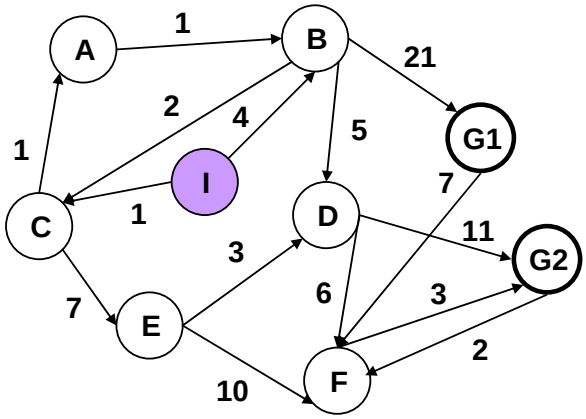
Camino a la solución: I B D G2 Coste: 4+5+11 = 20

Búsqueda con profundización iterativa

- Aplicación iterativa del algoritmo de búsqueda de profundidad limitada, con límite de profundidad variando de forma creciente (0,1,...)
  - Evita el problema de la elección del límite
- Combina las ventajas de los algoritmos primero en profundidad y primero en anchura
- Suele ser el método no informado preferido cuando el espacio de estados es grande y la profundidad de la solución se desconoce
- Propiedades:
  - *Óptima y completa (como primero en anchura: la de menor profundidad)* pero ocupando poca memoria (como primero en profundidad)
  - Tiempo: $O(r^p)$ *(algo mejor que primero en anchura)*
    - Nodos expandidos: r^p * *1 vez* + ... + r^2 * *(p-1)* + r^1 * *p* + r^0 * *(p+1 veces)*
    - La repetición de cálculos afecta principalmente a niveles superiores por lo que no es excesivamente importante
  - Espacio: $O(r*p)$ (como primero en profundidad)

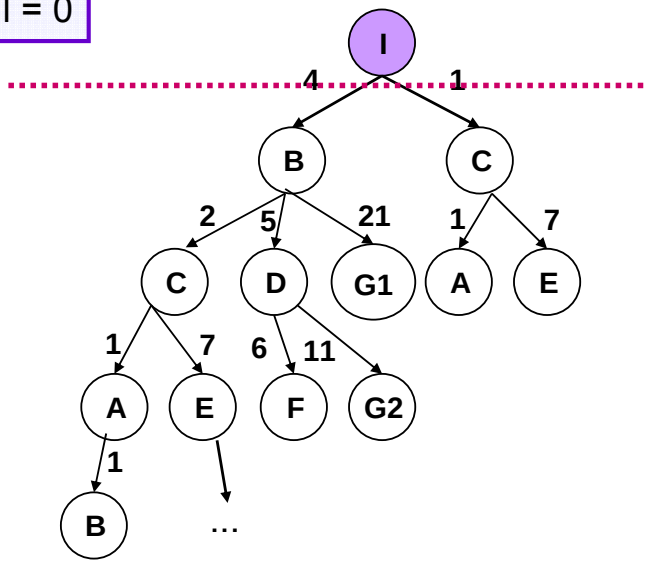
Ejemplo: búsqueda con profundización iterativa

Espacio de estados



Espacio de búsqueda

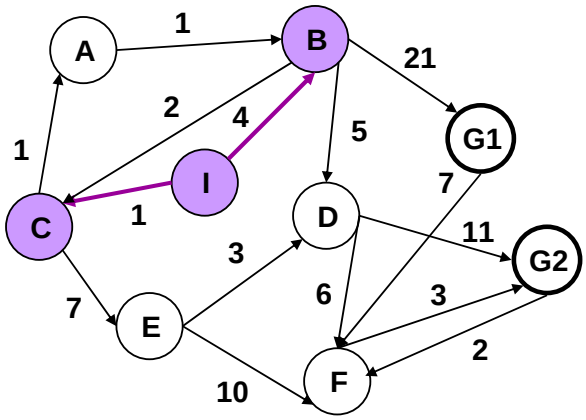
I = 0



Nodos expandidos: I

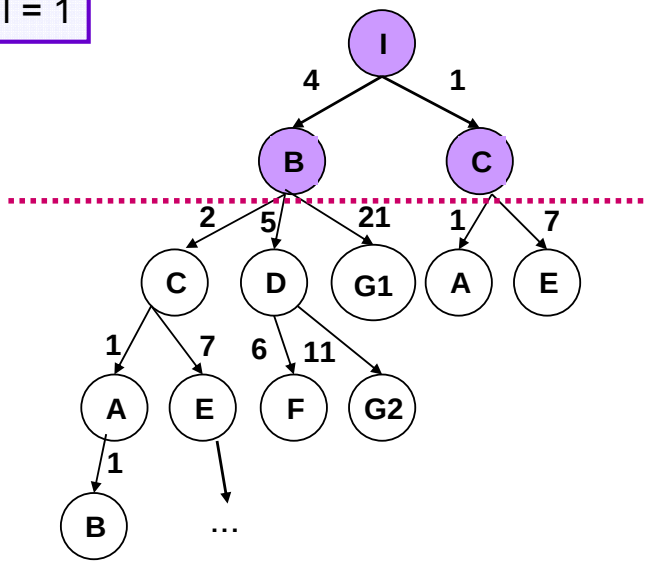
Ejemplo: búsqueda con profundización iterativa

Espacio de estados



Espacio de búsqueda

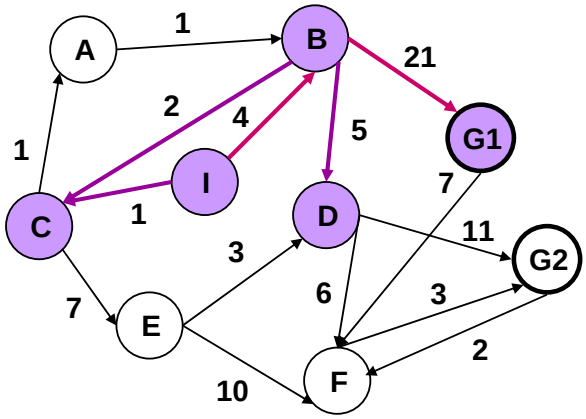
I = 1



Nodos expandidos: I I B C

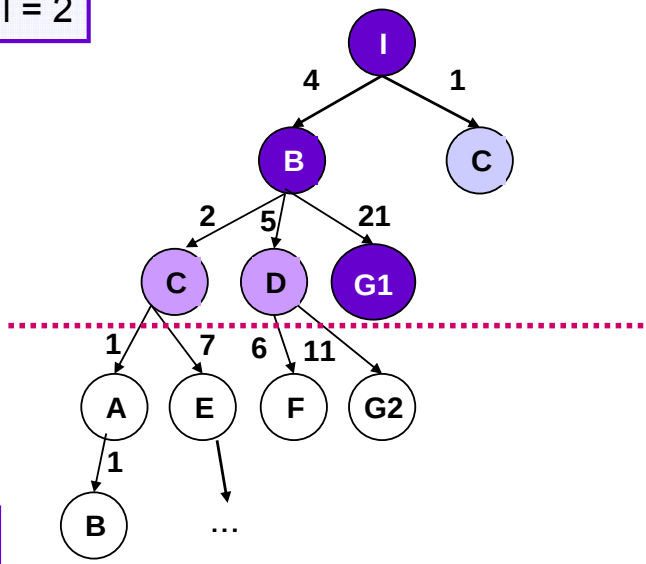
Ejemplo: búsqueda con profundización iterativa

Espacio de estados



Espacio de búsqueda

I = 2



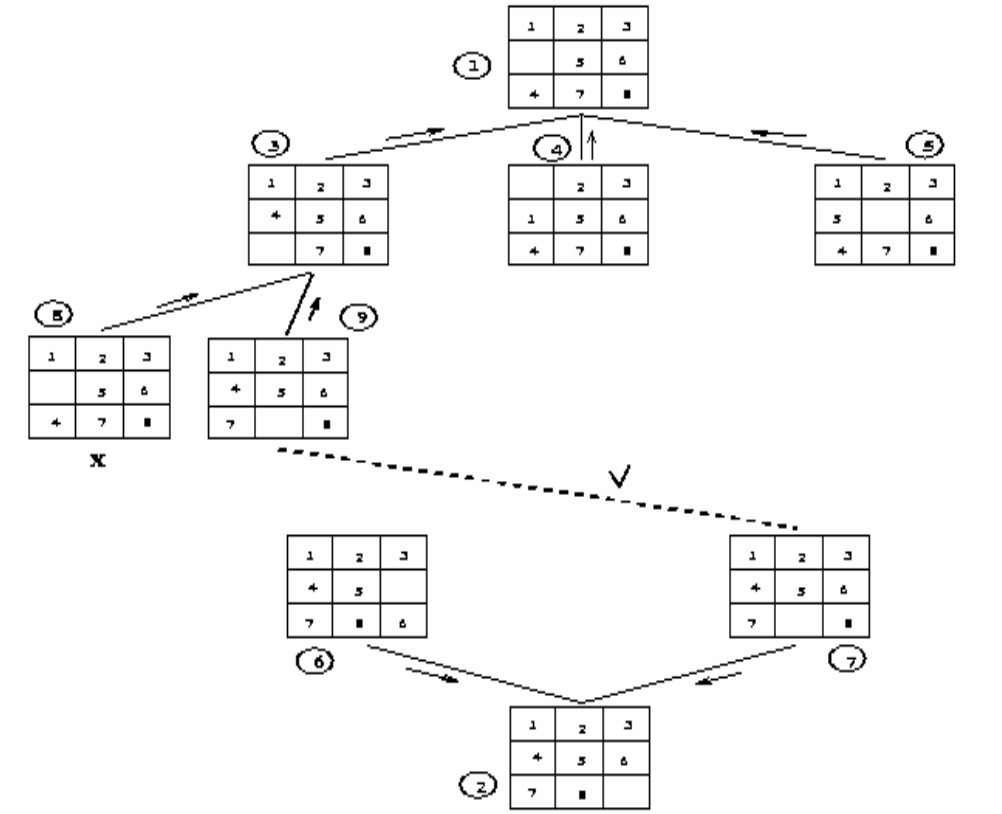
Nodos expandidos: I I B C I B C D G1

Camino a la solución: I B G1 Coste: 4+21 = 25

Búsqueda bidireccional (Pohl, 1969)

- La idea es ejecutar dos búsquedas simultáneas: una hacia delante desde el estado inicial y la otra hacia atrás desde el estado objetivo, parando cuando las dos búsquedas se encuentren
  - Motivación: $r^{p/2} + r^{p/2}$ es mucho menor que r^p
  - Es necesario conocer explícitamente el estado objetivo (si hay varios, puede ser problemático) y poder obtener los predecesores de un estado
- Propiedades:
  - *Óptima y completa si las búsquedas son en anchura y los costes son uniformes*. Otras combinaciones no lo garantizan: ¡no encontrarse!
  - Tiempo: $O(2*r^{p/2}) = O(r^{p/2})$
    - Si la comprobación de la coincidencia puede hacerse en tiempo constante
  - Espacio: $O(r^{p/2})$ *(su mayor debilidad)*
    - Al menos, los nodos de una de las dos partes se deben mantener en memoria para la comparación *(suele usarse una tabla hash para guardarlos)*

Búsqueda bidireccional



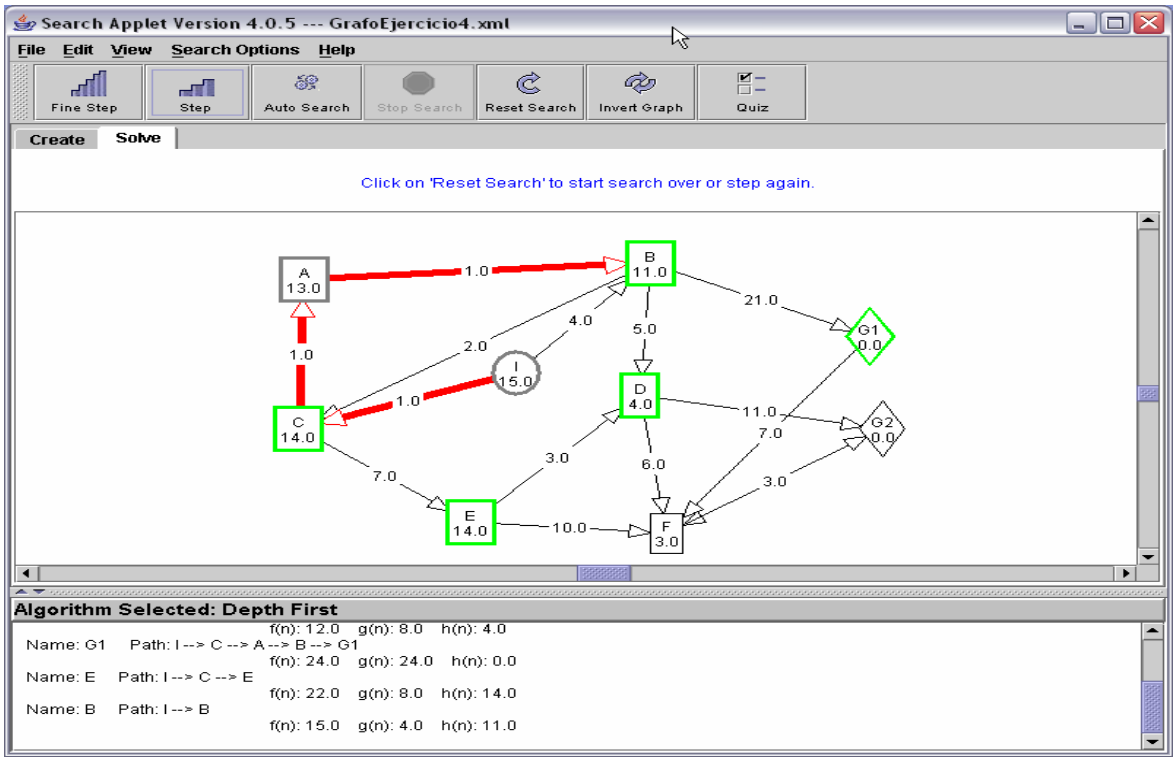
Resumen de métodos no informados

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)	Sí	$O(r^{p/2})$	$O(r^{p/2})$

r = factor de ramificación
m = máxima profundidad del árbol

p = profundidad mínima solución
l = profundidad límite

Criterio de desempate: menor coste (y no orden alfabético)



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
  - Disponen de alguna información sobre la proximidad de cada estado a un estado objetivo, guiando así la búsqueda hacia el camino más "prometedor" (elección informada del siguiente paso)
    - No considerando estados no prometedores ni sus descendientes (podando el árbol de búsqueda), un método informado *podría* desafiar la explosión combinatoria y encontrar una solución aceptable
  - No garantizan encontrar solución, aunque existan soluciones
  - Si encuentran una solución, no aseguran que ésta tenga buenas propiedades (que sea de longitud mínima o de coste óptimo)
  - En *algunas ocasiones* (que, en general, no se podrán determinar a priori), encontrarán una solución aceptablemente buena en tiempo razonable

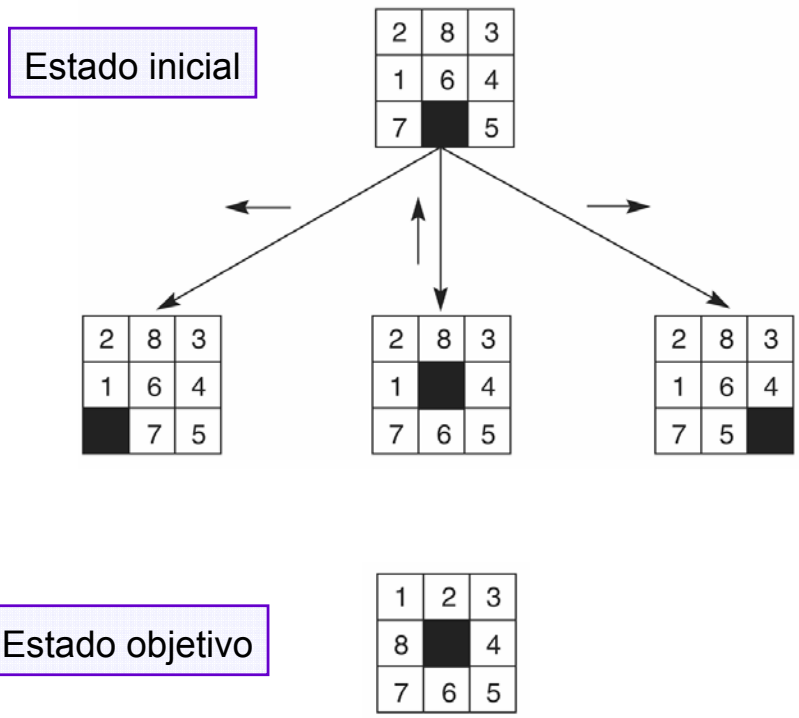
Heurísticas

- Heurística
  - Técnica que puede resolver un problema pero sin ofrecer garantía de que vaya a hacerlo
    - Newell, Shaw y Simon, 1963
  - Técnica que mejora la eficiencia de la búsqueda pudiendo sacrificar la completitud
    - Limitación inherente: posibilidad de fallar (no encontrar solución, habiéndola) o de encontrar soluciones no óptimas
  - Utiliza conocimiento específico del problema (más allá de la definición del problema en sí mismo)
  - Consejo acerca del orden de los sucesores de un estado para la búsqueda
    - Tiene efectos locales: elección del sucesor
- Método de búsqueda heurística
  - Algoritmo que hace uso de alguna heurística durante la búsqueda en el espacio de estados

Función heurística

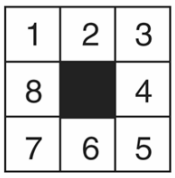
- Función heurística *h'*
  - Asocia a cada estado del espacio de estados una cierta cantidad numérica que evalúa de algún modo lo prometedor que es ese estado para alcanzar un estado objetivo
  - Esto sirve para seleccionar el estado con mejor valor heurístico
  - Dos posibles interpretaciones
    - Puede ser una estimación de la "calidad" de un estado
      - Los estados de mayor valor heurístico son los preferidos
    - Puede ser una estimación de lo próximo que se encuentra un estado de un estado objetivo (distancia, coste estimado del camino más barato, predicción de futuro...)
      - 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



Ejemplo: heurísticas para el 8-puzzle

- h_a = suma de 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

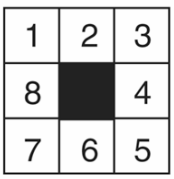


- h_b = nº de fichas descolocadas (respecto al tablero objetivo)
  - Es la heurística más sencilla y parece bastante intuitiva
    - Ejemplo: 5
  - Pero no usa la información relativa al esfuerzo (nº de movimientos) necesario para llevar una ficha a su sitio

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



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...

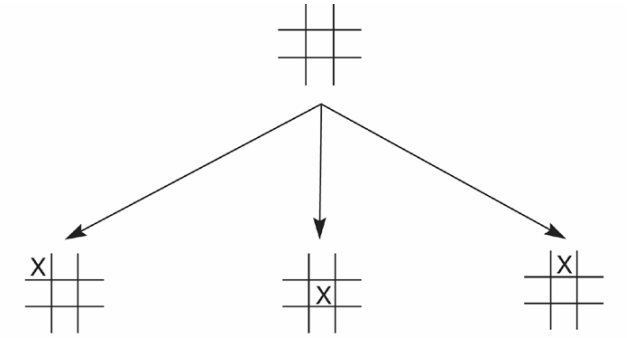
1	2	3
8		4
7	6	5

Estado objetivo

	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

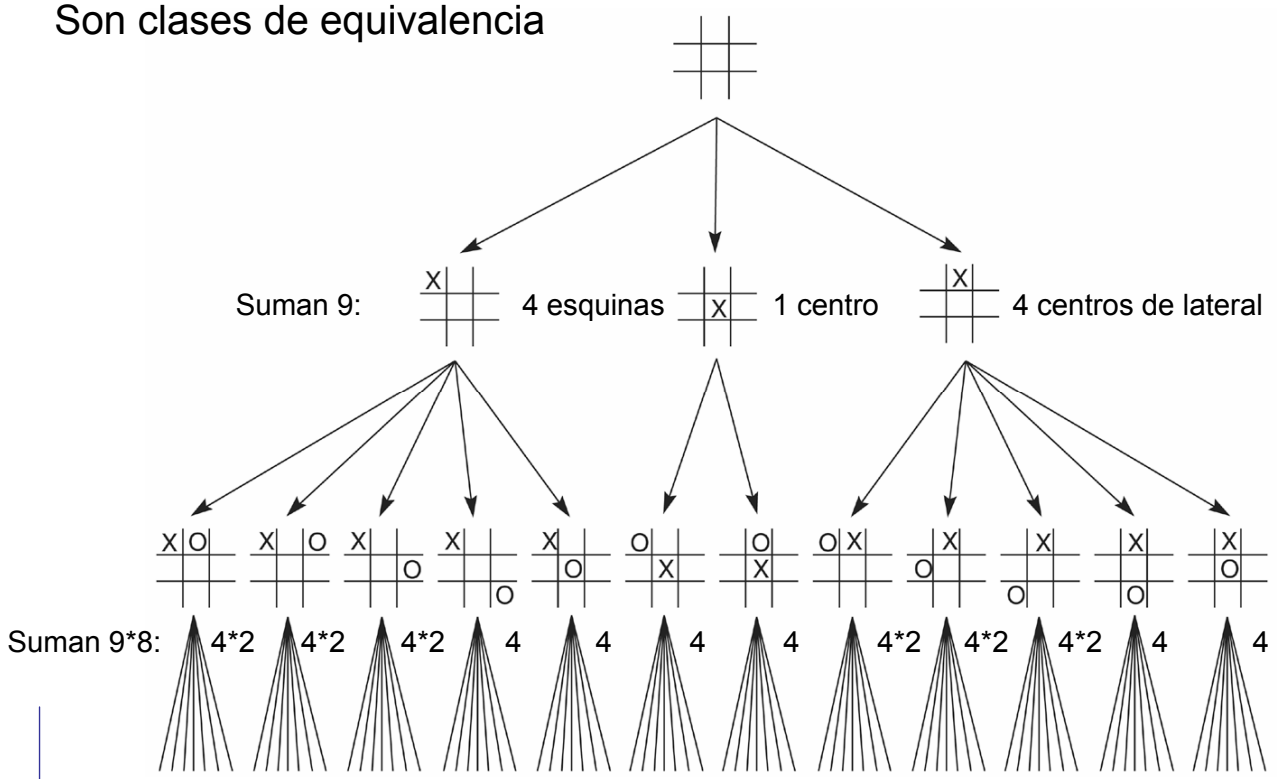
- Representación obvia
  - 9 movimientos iniciales, 8 posibles segundos movimientos, ...
  - Muchos operadores y "estados" distintos (configuraciones de tablero)
- Representación con reducción simétrica
  - 3 movimientos iniciales
    - esquina
    - centro del tablero
    - centro de un lateral
  -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



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

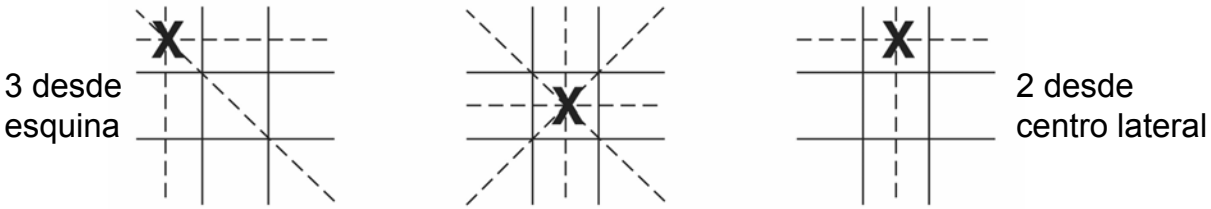
Ejemplo: reducción simétrica en las 3 en raya

Son clases de equivalencia



Ejemplo: heurística para las 3 en raya

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

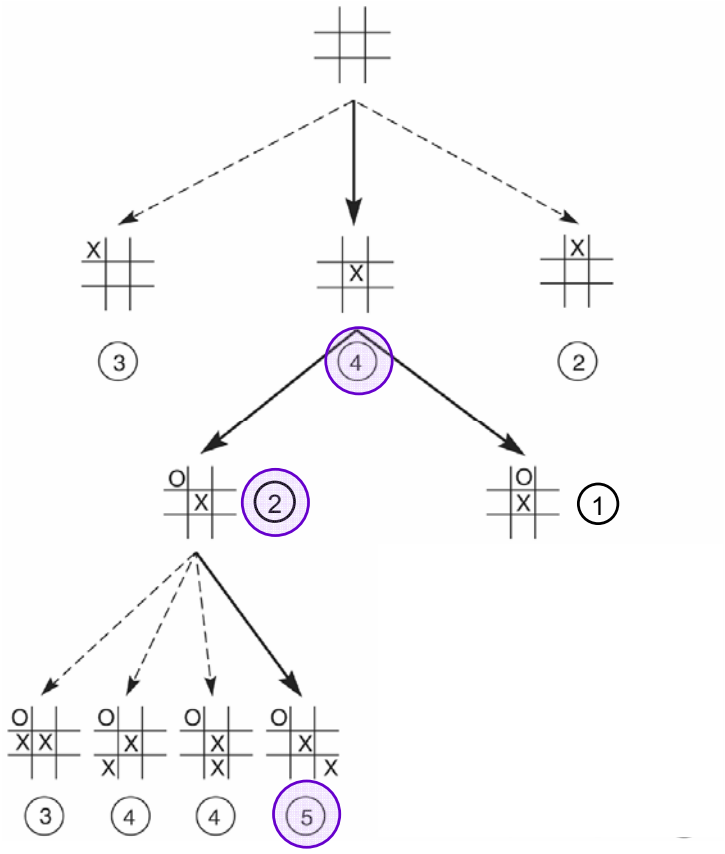


4 desde centro tablero

  - 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
  - El oponente elegirá esquina o centro de un lateral
  - 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



Resolución de problemas y espacio de búsqueda

- **Métodos informados o heurísticos**
  - 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

- Realmente debería llamarse “primero el aparentemente mejor”
  - Si supiéramos realmente 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 necesario para llegar a una solución a partir de ese nodo *n*
  - En lugar de criterios fijos como en 1º en profundidad o 1º en anchura, se usará una estimación
  - 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

f ' puede ser básicamente de dos tipos

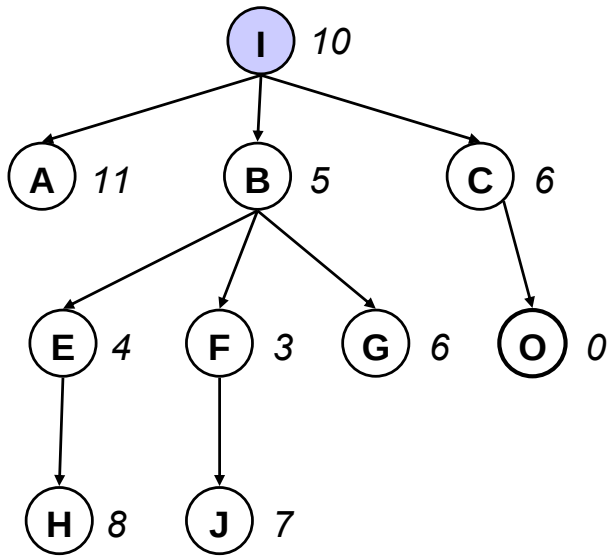
- Una función que considere exclusivamente el coste mínimo estimado para llegar a una solución a partir del nodo *n*
  - *f '(n) = h'(n)* => búsqueda voraz
- Una función que considere el coste total estimado del camino a un nodo objetivo que pase por el nodo *n*
  - En este caso, la función está formada por dos componentes:
    - *g(n)* = coste del camino hasta *n*
      - No es una estimación, sino un coste calculado exactamente
    - *h'(n)* = coste mínimo estimado para llegar a una solución desde *n*
  - *f '(n)= g(n) + h'(n)* => búsqueda A* sólo si *h'(n)* cumple ciertas condiciones (calcula el coste estimado más barato de la solución)
    - *h'(n)* ha de satisfacer ciertas condiciones para que la búsqueda sea A*
      - Heurística admisible: no ha de sobrestimar nunca el coste de alcanzar el objetivo *(ser optimista por naturaleza)*

Búsqueda voraz *(greedy)*

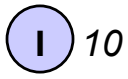
- ***f '(n) = h'(n)***
  - Representa el coste mínimo estimado para llegar desde *n* a un nodo objetivo (por el camino más barato)
    - *h'(n) = 0* para los nodos objetivo
- Voraz: en cada paso trata de ponerse lo más cerca posible del objetivo
- Propiedades
  - No es óptima ni completa, como primero 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



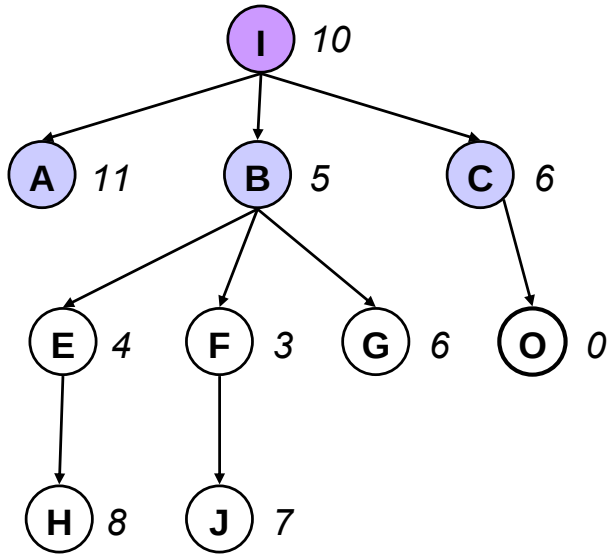
Espacio de búsqueda



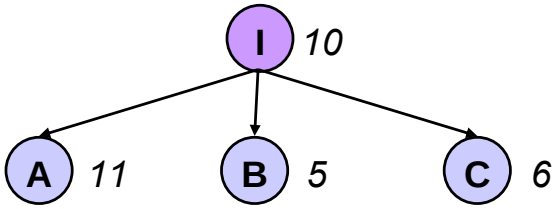
Cola de prioridad de nodos abiertos: I (10)

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

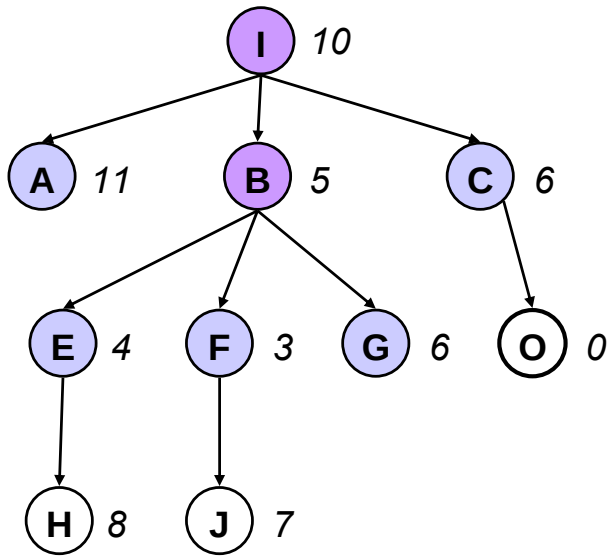


Cola de prioridad de nodos abiertos: A (11) C (6) B (5)

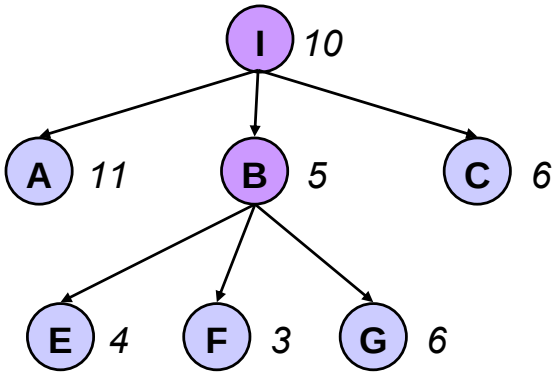
Nodos expandidos (por orden): I

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

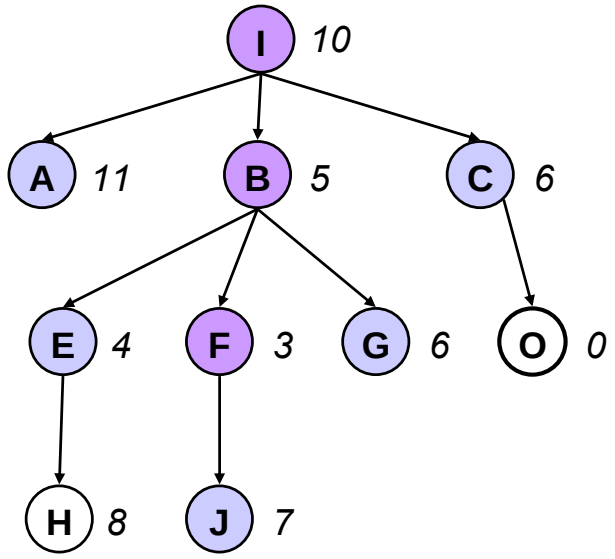


Cola de prioridad de nodos abiertos: A (11) G(6) C (6) E (4) F (3)

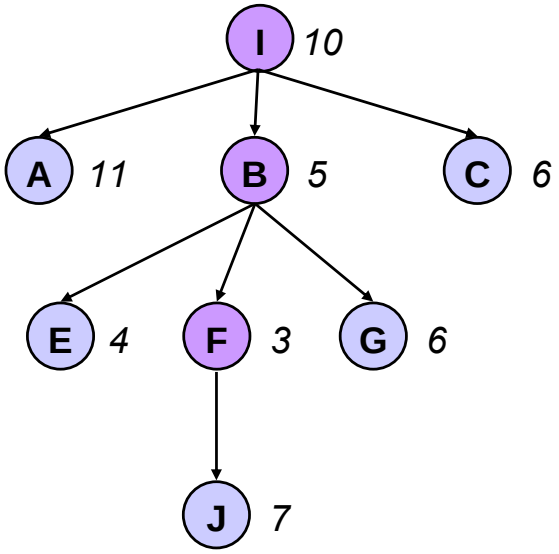
Nodos expandidos (por orden): I B

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

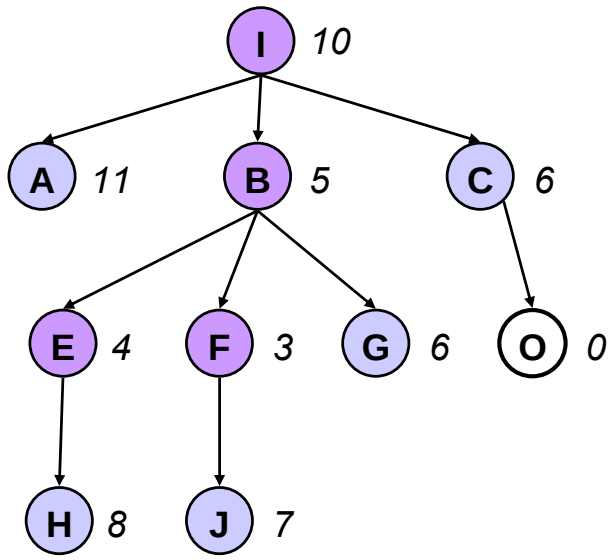


Cola de prioridad de nodos abiertos: A (11) J (7) G (6) C (6) E (4)

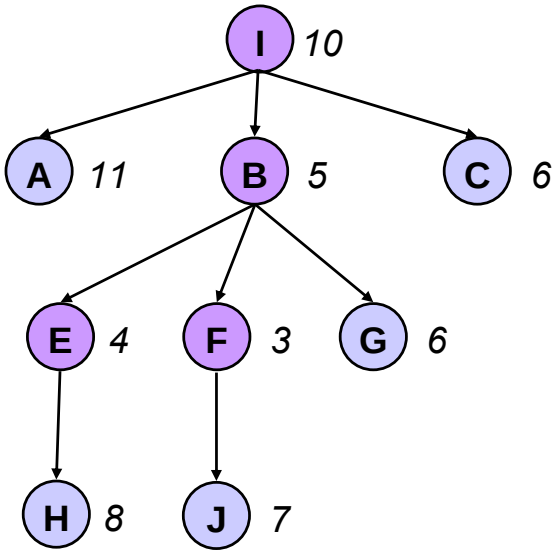
Nodos expandidos (por orden): I B F

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

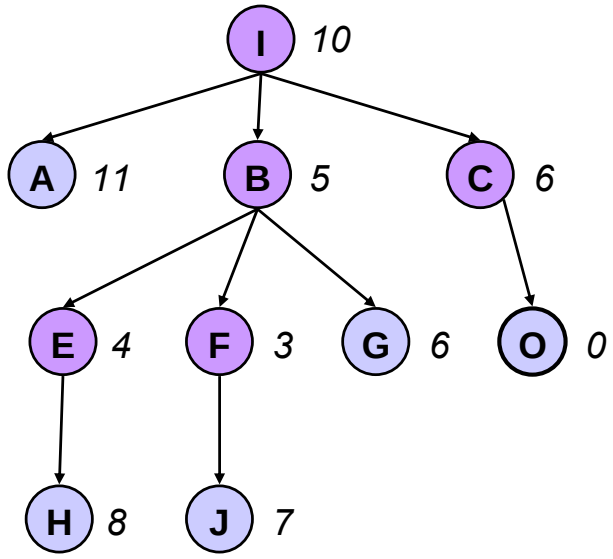


Cola de prioridad de nodos abiertos: A (11) H (8) J (7) G (6) C (6)

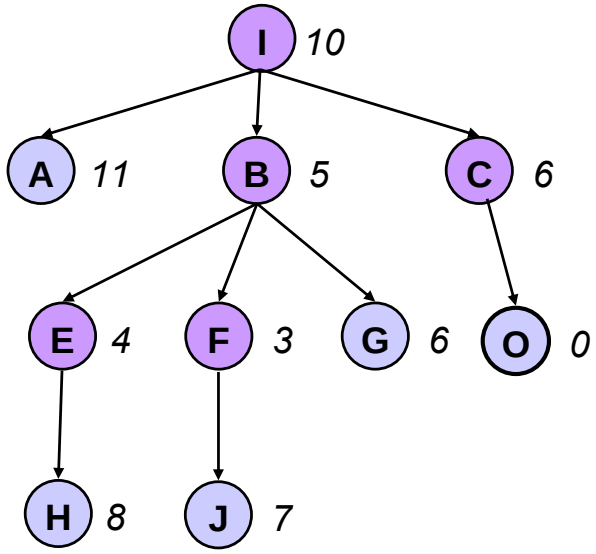
Nodos expandidos (por orden): I B F E

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda

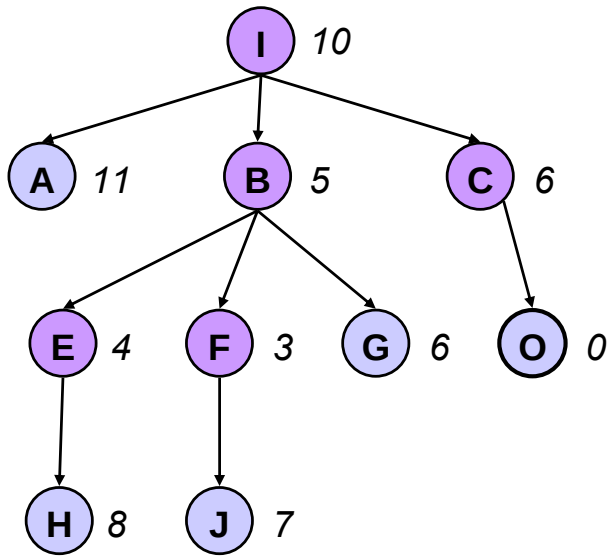


Cola de prioridad de nodos abiertos: A (11) H (8) J (7) G (6) O (0)

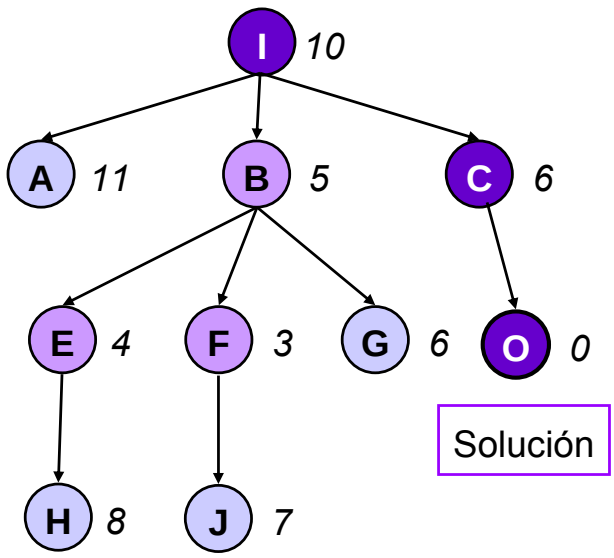
Nodos expandidos (por orden): I B F E C

Ejemplo: búsqueda voraz

Espacio de estados



Espacio de búsqueda



Solución

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

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

Búsqueda óptima: algoritmo A* (1968)

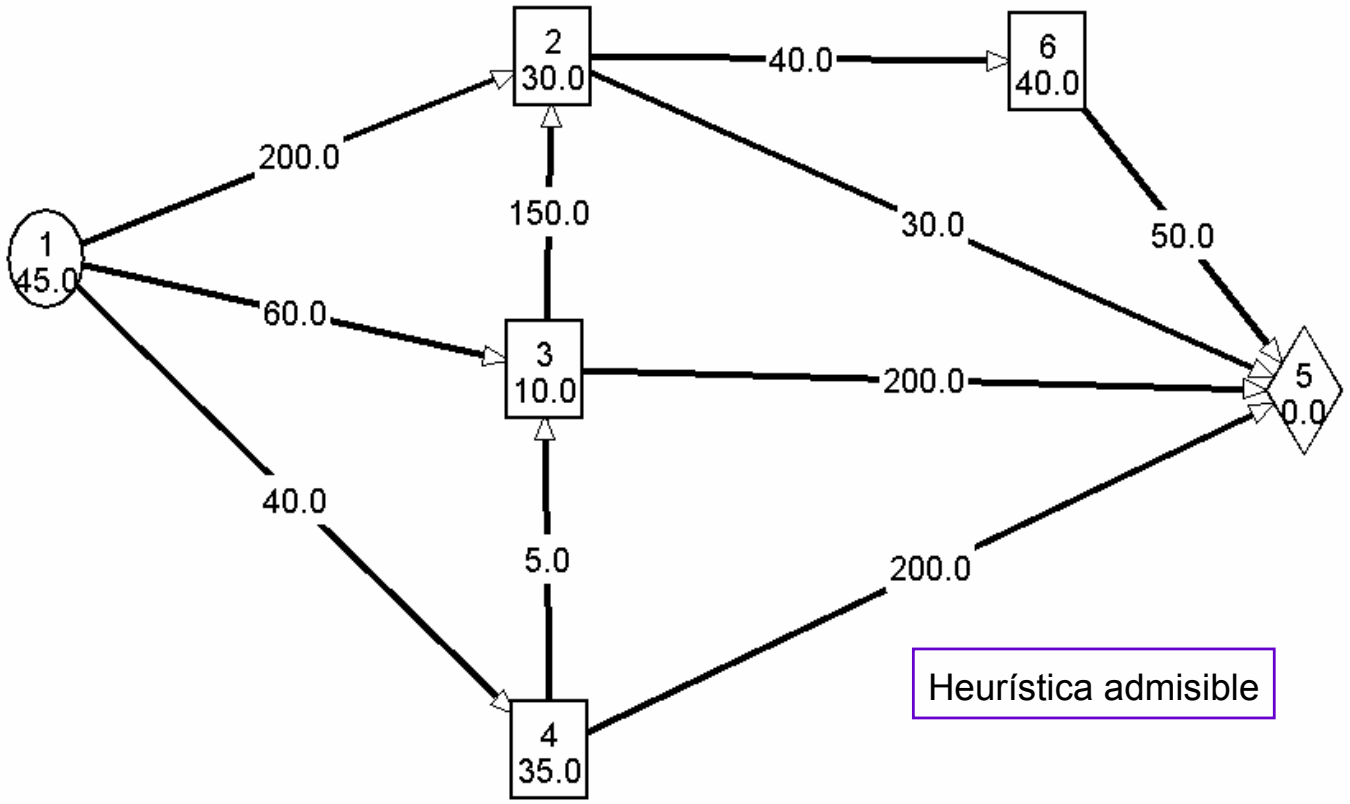
- ***f '(n) = g(n) + h'(n)***
  - *f '(n)* = estimación del coste mínimo total (desde el inicial hasta un objetivo) de cualquier solución que pase por el nodo *n*
  - *g(n)* = coste real del camino hasta *n*
  - *h'(n)* = estimación del coste mínimo desde *n* a un nodo objetivo
    - Si *h'* = 0 => búsqueda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

- Combinación de una estrategia de tipo primero en anchura con una estrategia de tipo primero en profundidad
  - La componente *g* de *f '* le da el toque realista, impidiendo que la búsqueda 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

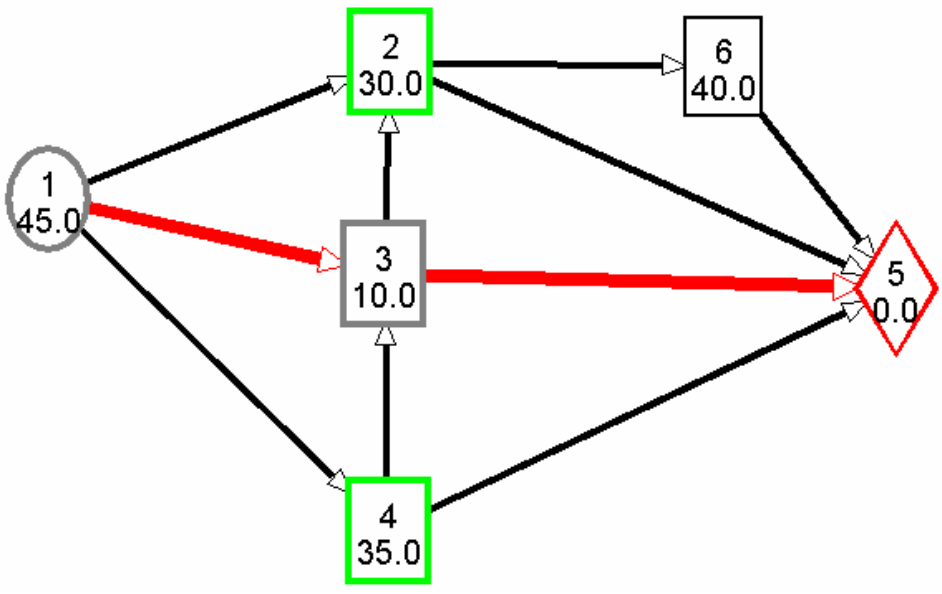
- *h'* es una heurística admisible si *h'(n)* ≤ *h(n)* para todo *n,* donde
  - *h(n)* representa el coste real para ir desde *n* a un nodo objetivo por el camino de menor coste
  - No ha de sobrestimar nunca el coste de alcanzar el objetivo
  - Son heurísticas optimistas por naturaleza
- **Sólo si *h'* es admisible**, la búsqueda primero el mejor 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 de búsqueda primero el mejor que utiliza una función de evaluación *f '* que cumple *1)* y *2)* es óptimo
  - La solución que encuentra es óptima *(la de menor coste real)*

Ejemplo



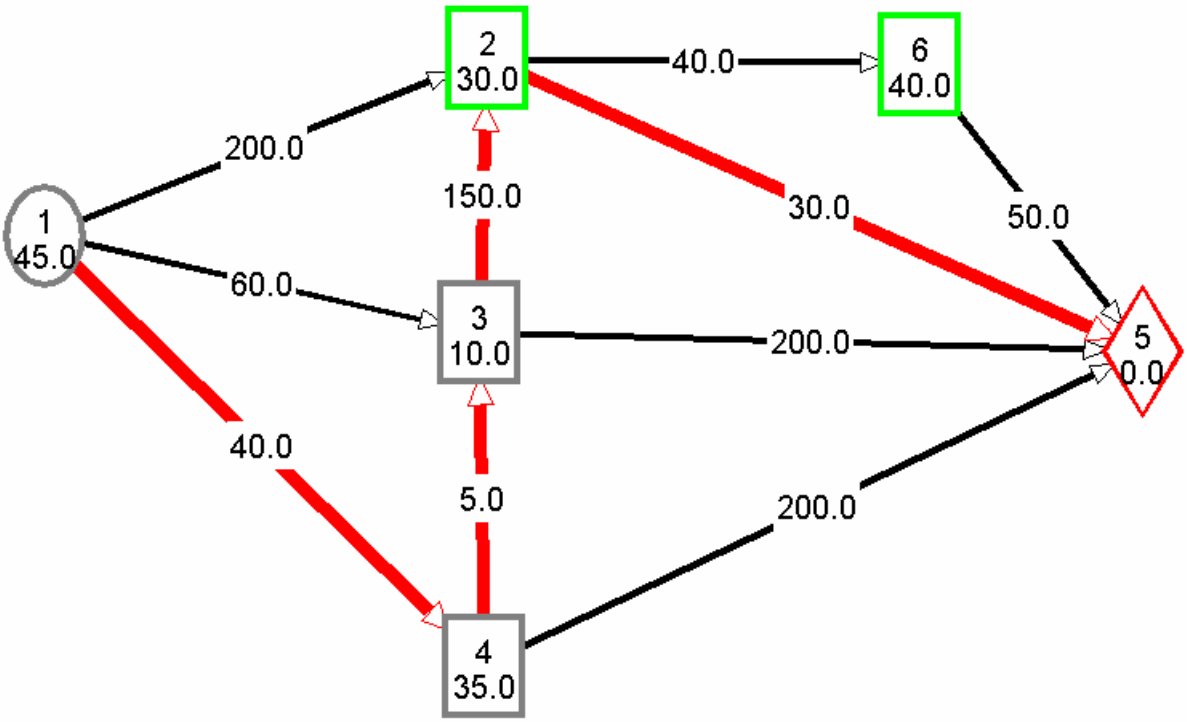
Heurística admisible

Solución con voraz



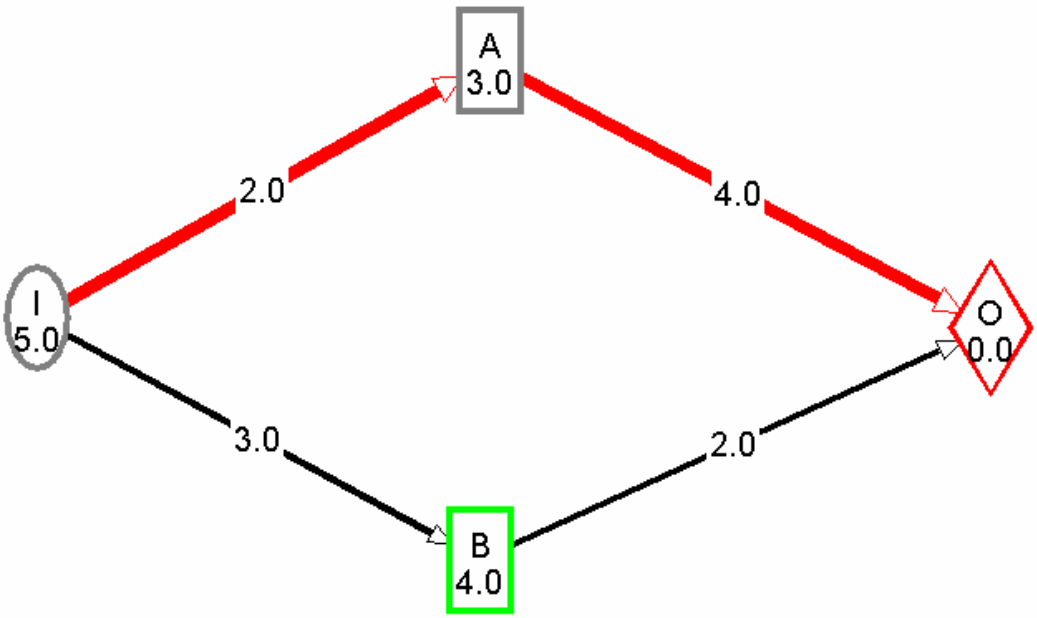
Solución con voraz: 1-3-5 Coste (no tenido en cuenta): 60+200 = 260

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*

A*: demostración de su optimalidad

- Sea ***OO*** un nodo objetivo óptimo ***g(OO) = f****
  (el coste del camino a *OO* desde el nodo inicial es óptimo)
- Sea ***OS*** un nodo objetivo sub-óptimo ➔ ***g(OS) > f****
- Si seleccionásemos *OS* como siguiente nodo a expandir, al ser un objetivo, acabaría la búsqueda y obtendríamos esa solución sub-óptima. Veamos cómo esta suposición conduce a una contradicción
  - Sea *X* el último nodo hoja obtenido que forma parte del camino hacia *OO*
    - Tiene que existir *X* porque, en caso contrario, hubiéramos expandido completamente ese camino y generado el nodo *OO* que, al tener menor coste que *OS*, habría sido seleccionado en su lugar
  - Como ***h'* es admisible** ➔ $f'(X) \leq f^*$
  - Como no hemos elegido expandir *X* sino *OS* ➔ $f'(OS) \leq f'(X)$
  - Por lo tanto: $\boldsymbol{f'(OS) \leq f^*}$
- Como *OS* es un nodo objetivo ➔ $h'(OS) = 0$ ➔ $f'(OS) = g(OS)$
  ➔ $\boldsymbol{g(OS) \leq f^*}$ ***¡contradicción!***

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        := []                          % inicialización
abiertos        := [INICIAL]
f'(INICIAL) := h'(INICIAL)
repetir
    si abiertos = [] entonces fallo
    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 SUCESORES de MEJOR_NODO
                para cada SUCESOR hacer TRATAR_SUCESOR
hasta solución_encontrada o fallo
```


TRATAR_SUCESOR *(distinción de casos)*

```
SUCESOR.ANTERIOR := MEJORNODO          % nodo padre
g(SUCESOR):= g(MEJOR_NODO) + coste(MEJOR_NODO->SUCESOR)
                % coste del camino hasta SUCESOR

caso "SUCESOR = VIEJO" perteneciente a cerrados
    si g(SUCESOR) < g(VIEJO) entonces      % (no si monotonía)
        % nos quedamos con el camino de menor coste
        VIEJO.ANTERIOR := MEJOR_NODO
        actualizar g(VIEJO) y f'(VIEJO)
        propagar g a sucesores de VIEJO
    eliminar SUCESOR
    añadir VIEJO a SUCESORES_MEJOR_NODO
```

TRATAR_SUCESOR *(continuación)*

```
caso "SUCESOR = VIEJO" perteneciente a abiertos
    si g(SUCESOR) < g(VIEJO) entonces
        % nos quedamos con el camino de menor coste
        VIEJO.ANTERIOR := MEJOR_NODO
        actualizar g(VIEJO) y f'(VIEJO)
    eliminar SUCESOR
    añadir VIEJO a SUCESORES_MEJOR_NODO

caso SUCESOR no estaba en abiertos ni cerrados
    añadir SUCESOR a abiertos
    añadir SUCESOR a SUCESORES_MEJOR_NODO
    f'(SUCESOR) := g(SUCESOR) + h'(SUCESOR)
```


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

```
% 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 SUCESOR no estaba en abiertos ni cerrados
    añadir SUCESOR a ABIERTOS
    añadir SUCESOR a SUCESORES_MEJOR_NODO
    f'(SUCESOR) := g(SUCESOR) + h'(SUCESOR)
```

TRATAR_SUCESOR: ya en *abiertos*

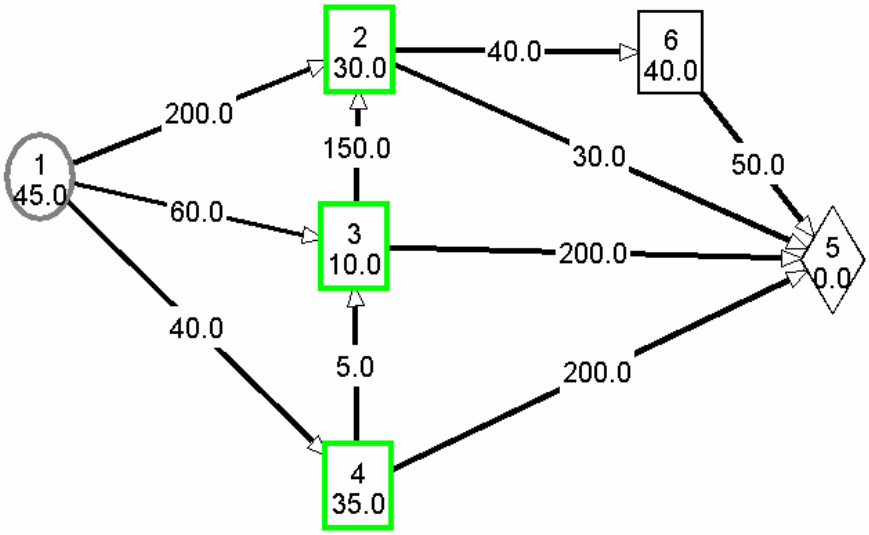
```
caso "SUCESOR = VIEJO" perteneciente a abiertos
    si g(SUCESOR) < g(VIEJO) entonces
        % nos quedamos con el nodo con menor coste
        % actualizando el que ya está en abiertos
        VIEJO.ANTERIOR := MEJOR_NODO
        actualizar g(VIEJO) y f'(VIEJO)
    eliminar SUCESOR
    % no añadimos nada nuevo
    añadir VIEJO a SUCESORES_MEJOR_NODO
```


TRATAR_SUCESOR: ya en *cerrados*

```
caso "SUCESOR = VIEJO" perteneciente a 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
    % no añadimos nada nuevo
    añadir VIEJO a SUCESORES_MEJOR_NODO
```

Ejemplo A*: se expande 1

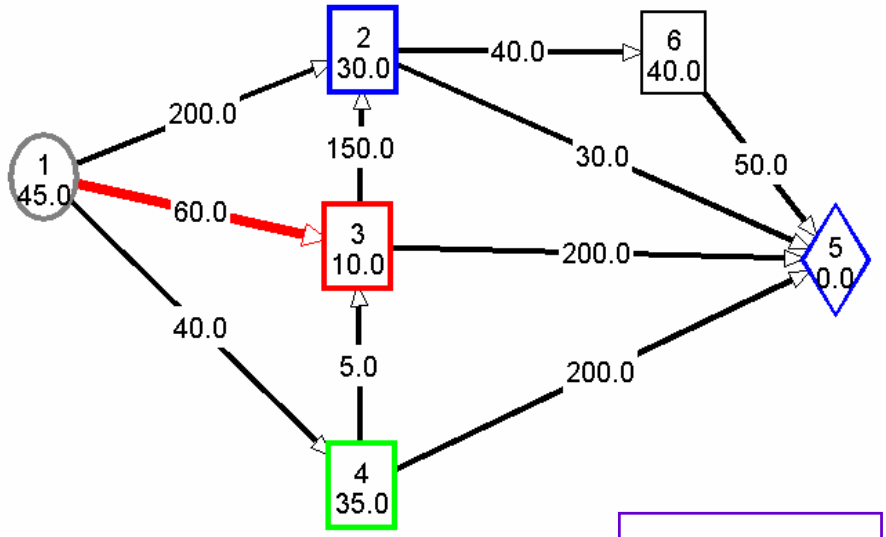
Abiertos				
padre		1	1	1
estado	**1**	**2**	**3**	**4**
h'	45	30	10	35
g	0	200	60	40
f'	45	230	70	75



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

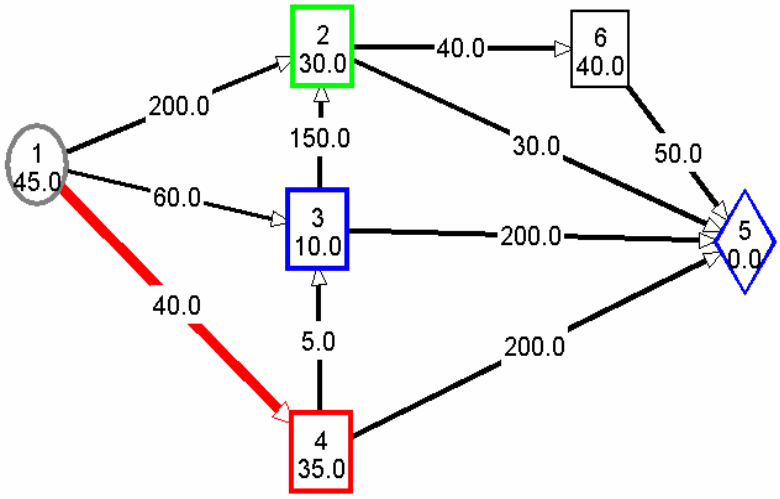


- Camino
  - 1-3 Coste: 60
- Sucesores de 3: 2 y 5
  - 5 es nuevo: se añade a *abiertos*
  - 2 ya está en *abiertos*
  - ¿Es mejor este nuevo camino 1-3-2 que 1-2?
    - No: tiene peor coste (60+150 frente a 200)
      - Nos quedamos con el que teníamos

Cerrados: 1, 3

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

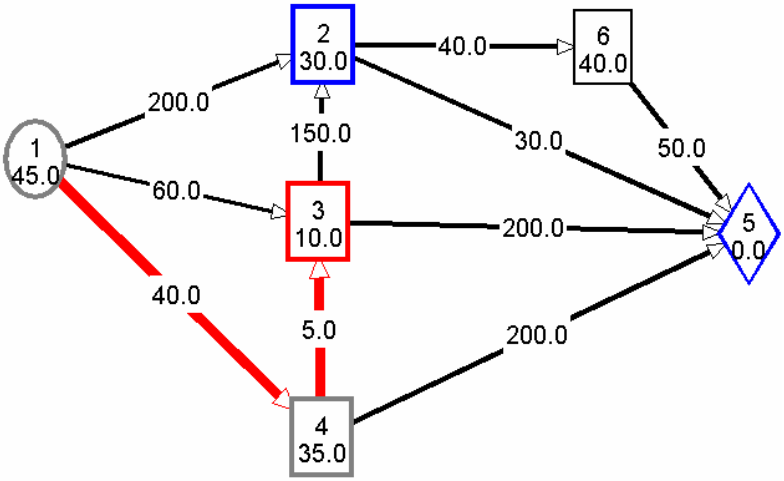


- Cambio de camino
  - 1-4 Coste: 40
- Sucesores de 4: 3 y 5
  - 3 está en *cerrados* (ya había sido expandido)
  - ¿Es mejor este nuevo camino 1-4-3 que 1-3?
    - Sí: tiene mejor coste (40+5 frente a 60)
    - Nos quedamos con éste y propagamos el cambio a los sucesores de 3 (2 y 5)

Cerrados: 1, 3, 4

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

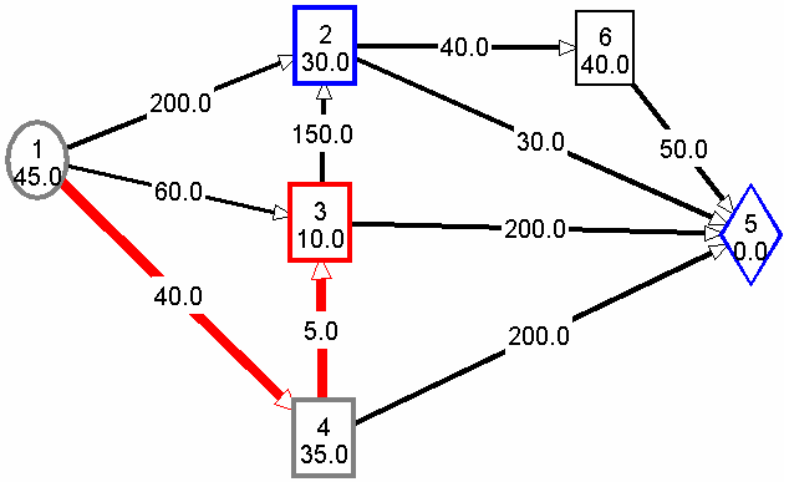


- Actualizamos 3 y su hijo 5
- Falta su hijo 2
  - Está en *abiertos* como hijo de 1
  - ¿Es mejor 1-4-3-2 que 1-2?
    - Sí: tiene mejor coste (40+5+150 frente a 200)
    - Nos quedamos con este nuevo camino

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

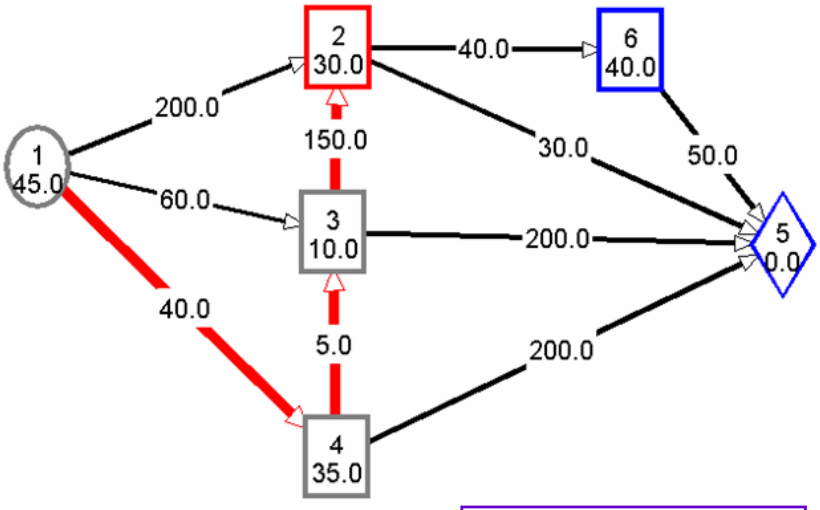


- Falta 5 como sucesor de 4
  - 5 ya está en *abiertos*
  - ¿Es mejor 1-4-5 que 1-4-3-5?
    - Sí: tiene mejor coste (40+200 frente a 245)
    - Nos quedamos con este nuevo camino

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

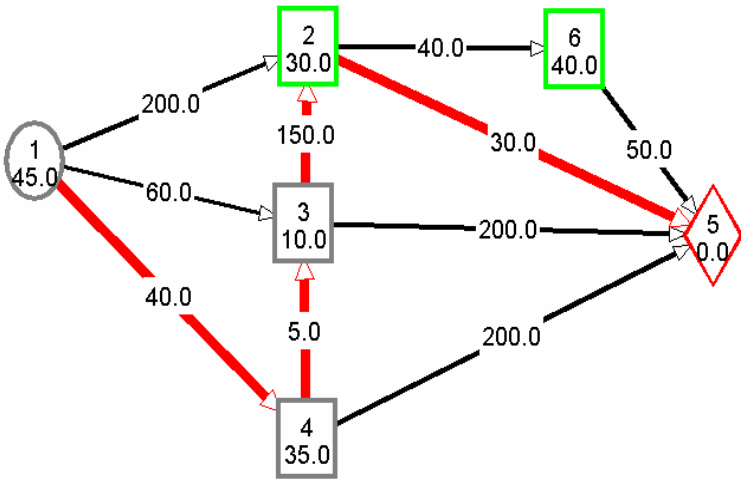


- Camino
  - 1-4-3-2 Coste: 195
- Sucesores de 2: 6 y 5
  - 5 está en *abiertos* y 6 es nuevo (lo añadimos a *abiertos)*
  - ¿Es mejor 1-4-3-2-5 que 1-4-5?
    - Sí: tiene mejor coste (195+30 frente a 240)
    - Nos quedamos con este nuevo camino

Cerrados: 1, 3, 4, 2

Ejemplo A*: se expande 5

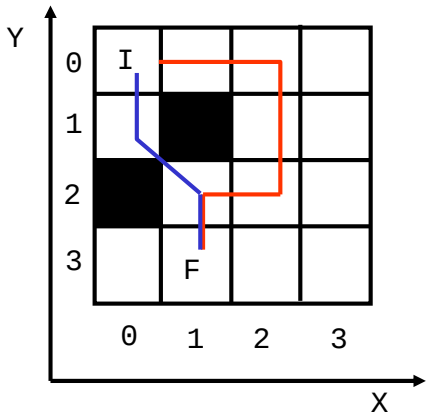
Abiertos						
padre		3	4	1	2	2
estado	**1**	**2**	**3**	**4**	**5**	**6**
h'	45	30	10	35	0	40
g	0	195	45	40	225	235
f'	45	225	55	75	225	275

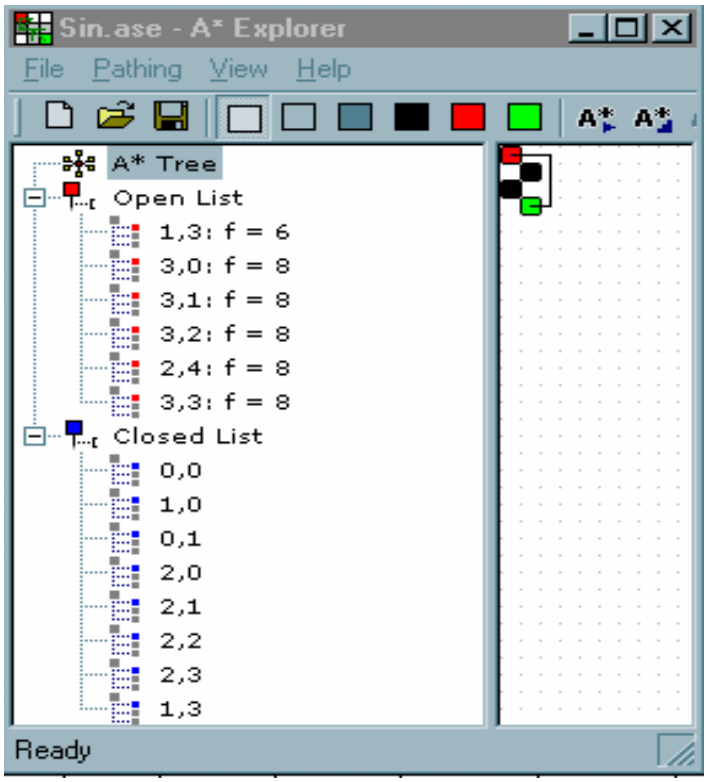


- Camino
  - 1-4-3-2-5 Coste: 225
- 5 es estado objetivo
  - Solución encontrada
  - Al ser admisible la heurística, hay garantía de optimalidad
    - Hemos encontrado el camino de menor coste

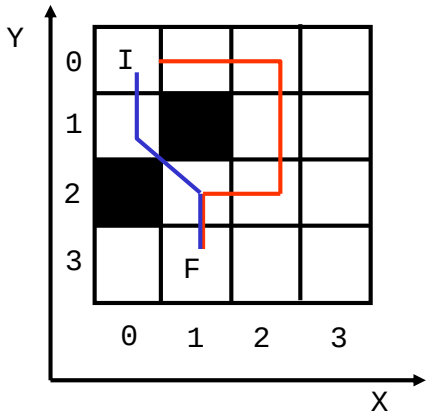
Cerrados: 1, 3, 4, 2, 5

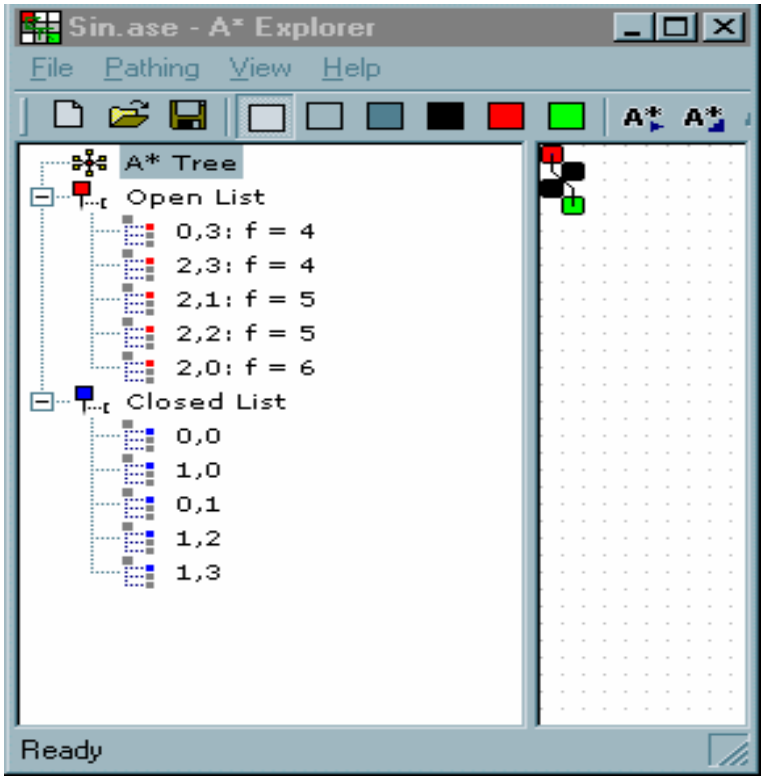
Ejemplo: A* Explorer





Ejemplo: A* Explorer

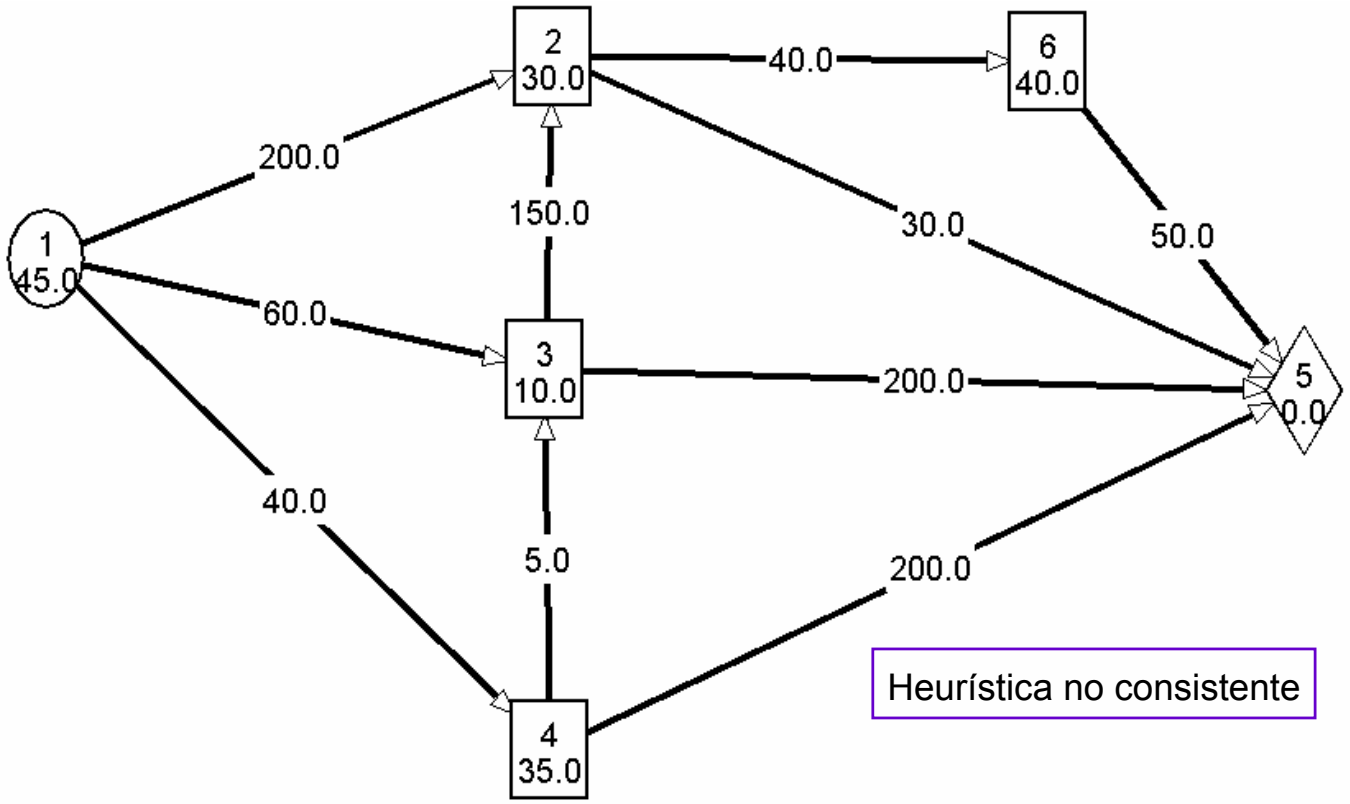




Otras propiedades de A*

- **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 se satisface la condición de consistencia, cada vez que A* expanda un nodo, habrá encontrado un camino óptimo al mismo
    - Incrementa la eficiencia al no necesitar revisitar nodos: 1ª expansión, la mejor

Ejemplo



Otras propiedades de A*

- Si la función heurística no es consistente, pero sí admisible, puede irse modificando de forma dinámica durante el proceso de búsqueda para que cumpla la condición de consistencia
  - En cada paso de A*, comprobamos los valores de *h'* para los sucesores del nodo *n* que acaba de ser expandido
  - Si alguno de estos valores de *h'* es menor que *h'(n)*, menos el coste del operador aplicado, habría que ajustar los valores de *h'* (durante el proceso de búsqueda) para que sea igual a *h'(n)* menos el coste del correspondiente operador
    - *h' (3) = h' (4) – c(4, 3) = 35 – 5 = 30 en el ejemplo*

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 expande ningún nodo con *f '(n) > f** (ahí está la poda)

Otras propiedades de 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
    - Hay nodos con factor de ramificación infinito
    - O 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 $f'(n) < f^*$ corre el riesgo de omitir la solución óptima

Complejidad de A*

- La búsqueda A* es completa, óptima y óptimamente eficiente, pero lamentablemente A* no es la respuesta a todas las necesidades de búsqueda
  - En el caso peor sigue siendo exponencial con respecto a la profundidad de la solución
  - El crecimiento exponencial ocurrirá a no ser que el error en la heurística no crezca más rápido que el logaritmo del coste de camino 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 (a *h(n)* y no a su logaritmo)
  - El crecimiento exponencial que resulta desborda la capacidad de cualquier computador

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

Alternativas propuestas

- 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*)
    - Tareas de tiempo real: obligan a tomar una decisión cada cierto tiempo
  - IDA* (Iterative Deepening A*)
    - Límite con f' (no la profundidad): expande sólo estados con coste inferior
  - SMA* (Simplified Memory-bounded A*)
    - Si al generar un sucesor falta memoria, se libera el espacio de los nodos de *abiertos* menos prometedores

Comparación de la calidad de heurísticas

- El diseño de buenas heurísticas no es sencillo
  - Se trata de un problema empírico
  - El buen juicio y la intuición pueden ayudar, pero para determinar su calidad es imprescindible evaluar su rendimiento en instancias del problema
- La precisión de una heurística afecta al rendimiento
- Hay dos maneras de comparar la calidad de dos heurísticas
  - Por demostración de dominancia (método teórico)
    - Es mejor la más dominante
  - Por factor de ramificación efectivo (método experimental)
    - Es mejor la que menor factor de ramificación efectivo 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

- Factor de ramificación efectivo r^*
  - Si N es el número de nodos generados por 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
    - El hipotético árbol con factor de ramificación r^* cumpliría:
      $$N = 1 + r^* + (r^*)^2 + \ldots + (r^*)^p$$
    - Conocemos p y N, entonces podemos despejar r^*
    - r^* puede variar según los ejemplos del problema, pero por lo general es constante para problemas lo suficientemente difíciles
    - Las medidas experimentales de r^* sobre un pequeño conjunto de problemas pueden proporcionar una buena guía para la utilidad total de la heurística
    - Una heurística bien diseñada tendría un valor de r^* cercano a 1 y permitiría resolver problemas bastante grandes
    - Ejemplo: 8-puzzle con $p = 12$
      - *Búsqueda ciega (profundización iterativa): $N = 3.644.035$, $r^* = 2.78$*
      - *$A^*(h_b)$: $N = 227$, $r^* = 1.42$* *$A^*(h_a)$: $N = 73$, $r^* = 1.24$*

Generación de heurísticas

- Hemos visto que h_b (piezas mal colocadas) y h_a (distancia de Manhattan) son heurísticas bastante buenas para el 8-puzzle y ya sabemos que h_a es mejor
  - ¿Cómo ha podido surgir h_a?
  - ¿Es posible que un programa invente mecánicamente una heurística?
- Ambas son estimaciones de la longitud del camino restante para el 8-puzzle, pero también son longitudes de caminos exactos para versiones *simplificadas* del puzzle (cambio de reglas):
  - Una ficha puede moverse a cualquier casilla (ocupada o no): h_b
  - Una ficha puede moverse 1 casilla en horizontal o en vertical aunque la casilla estuviera ocupada: h_a
- 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

Generación de heurísticas

- Si la definición de un problema está escrita en un lenguaje formal, es posible construir problemas relajados automáticamente
  - 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
  - Es crucial que los problemas relajados puedan resolverse esencialmente sin búsqueda
  - ABSOLVER (1993): programa que genera heurísticas automáticamente

Generación de heurísticas

- 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 $h'(n) = max \{ h'_1(n), h'_2(n), \ldots , h'_n(n) \}$
- También se pueden obtener heurísticas admisibles a partir del coste de la solución de un subproblema de un problema dado
- Otra posibilidad es aprender de la experiencia
  - La "experiencia" aquí significaría resolver muchos 8-puzzles, por ejemplo
  - 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 aisladas de cada estado que sean relevantes para su evaluación. Suelen utilizar combinación lineal de estas características: $c_1x_1(n) + c_2x_2(n)$
- En cualquier caso, una heurística debe ser fácil de calcular
  - Computacionalmente no debe ser más costosa que *expandir un nodo*

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

- En muchos problemas el camino al objetivo es irrelevante
  - Por ejemplo, en las 8-reinas lo que importa es la configuración final
    - Y en muchas aplicaciones importantes: diseño de circuitos integrados, disposición del suelo, planificación del trabajo, programación automática, optimización de redes, dirección de vehículos, gestión de carteras...
- Si no importa el camino al objetivo, podemos considerar una clase diferente de algoritmos que no se preocupan en absoluto de los caminos *(ignoran el coste del camino, en particular)*
  - Los 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 pegan 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
    - Pueden encontrar soluciones razonables en espacios de estados grandes o infinitos para los cuales son inadecuados algoritmos sistemáticos

Algoritmos de mejora iterativa

- Además de poder encontrar los objetivos, los algoritmos de búsqueda local son útiles para resolver problemas de optimización puros
  - El objetivo es encontrar el mejor estado según una cierta función objetivo
- Métodos informados de optimización local
  - En algunos problemas de optimización, la solución en sí tiene un coste asociado *(por ejemplo, en el problema de la mochila)* que se quiere optimizar (el coste del camino es indiferente)
  - Planteamiento habitual como búsqueda en el espacio de soluciones
  - Extrapolable a búsqueda en espacio de estados *(usando heurística)*
- Algoritmos de escalada: consumen muy 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)*

- Técnica de búsqueda local más básica
  - En cada paso, el nodo actual se intenta sustituir por el primer vecino mejor
    - Primer vecino con un valor más alto que el nodo actual
    - *O primer sucesor con una medida heurística más baja que el nodo actual*
  - Intenta moverse en dirección de un valor creciente, es decir, cuesta arriba *(cuesta abajo, si busca un valor decreciente)*
  - Termina cuando encuentra una solución o alcanza un "pico" en el que ningún vecino tiene valor más alto *(más bajo)*
  - El algoritmo no mantiene un árbol de búsqueda, sino una estructura de datos del nodo actual que necesita sólo el registro del estado y su valor según la función objetivo –en optimización *(función heurística)*
  - No mira adelante más allá del vecino inmediato del estado actual (ni hermanos, ni otros hijos)
- Es como un genera y prueba, matizado por una función heurística para inyectarle conocimiento específico del problema
  - Se suele usar a menudo cuando sólo se dispone de una buena heurística

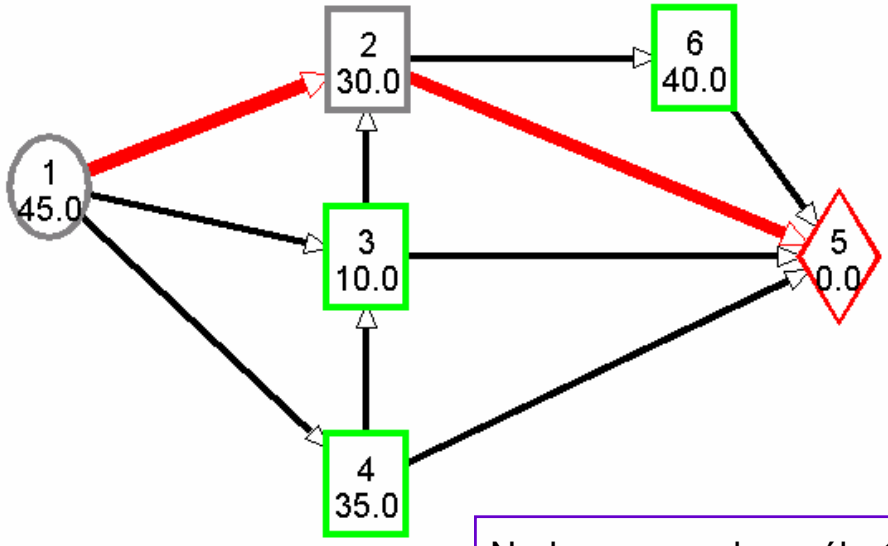
Escalada simple *(hill climbing)*

```
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 se haya
   encontrado 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 es mejor 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: numérico



Nodos generados: sólo 1, 2 y 5

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

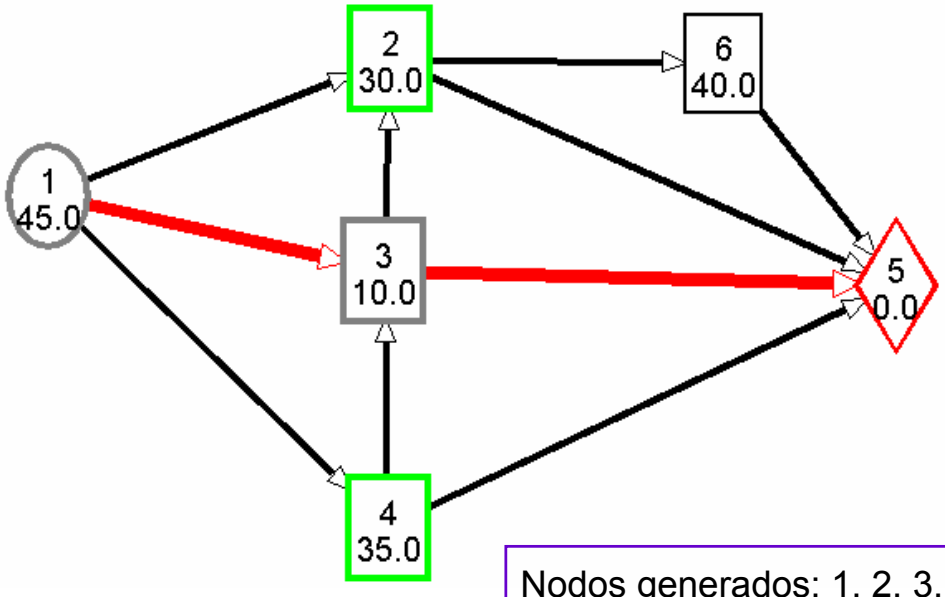
Escalada por máxima pendiente

- Variante: estudia todos los vecinos del nodo actual
  - En cada paso, el nodo actual se sustituye por el mejor vecino
    - Vecino con el valor más alto
    - *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 (y no por uno simplemente con pendiente)
  - Continuamente se mueve en dirección del valor creciente, es decir, cuesta arriba *(cuesta abajo, si busca un valor decreciente)*
  - Termina cuando encuentra una solución o alcanza un "pico" en el que ningún vecino tiene valor más alto *(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

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 hacer
        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:

  - Un máximo local (*mínimo local* u óptimo local)
    - Un estado mejor que sus vecinos pero peor que otros estados más alejados
  - Una meseta
    - Todos los estados vecinos tienen el mismo valor heurístico
    - Es imposible determinar el mejor movimiento: sería búsqueda ciega
  - Una cresta
    - Mezcla de los anteriores: región del espacio de estados en la que la dirección que determina la heurística no guía hacia ningún estado objetivo. Puede terminar en un máximo local o tener un efecto muy similar al de la meseta

Variantes de los algoritmos de escalada

- Muchas variantes que los mejoran procurando resolver los bloqueos
  - Profundidad + escalada: los nodos de igual profundidad son ordenados poniendo al comienzo los más prometedores y se permite *backtracking*
    - Recupera completitud y exhaustividad
  - Reiniciar toda o parte de la búsqueda
  - Dar un paso más: generar sucesores de sucesores y ver qué pasa
  - Si aparece un óptimo local, volver a un nodo anterior y probar en una dirección distinta
  - Si aparece una meseta, hacer un salto grande para salir de la meseta
    - ¿Cómo escaparse?
      - Reinicio aleatorio: comenzar la 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 de los algoritmos de escalada depende muchísimo de la forma del paisaje del espacio de estados
  - Problemas NP-duros: suelen tener un nº exponencial de óptimos locales
    - IA: último reducto para resolverlos en un tiempo aceptable y de forma aproximada
- 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)
- Un algoritmo puramente aleatorio es completo pero muy ineficiente
- Parece razonable intentar combinar la escalada con elección aleatoria de caminos de algún modo que produzca tanto eficiencia como completitud: ese algoritmo es el enfriamiento o temple simulado
  - 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

- Se plantea como un problema de minimización de la energía, que es la función a optimizar
  - Al comienzo del proceso, la temperatura T es alta y se permiten movimientos contrarios al criterio de optimización
  - 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 de la temperatura T) va lo bastante lento se alcanza un óptimo global con probabilidad cercana a 1
- Se ha usado mucho en diseño VLSI, en planificación de fábricas y en otras 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
mientras haya operadores aplicables a ACTUAL y no se haya encontrado
   solución hacer
   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)

```
                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^-ΔE/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]}
                        si N < P' {se acepta el movimiento}
                        entonces ACTUAL := NUEVOESTADO
   actualizar T de acuerdo con la planificación del
        enfriamiento
devolver MEJOR_HASTA_AHORA como solución
```

Resolución de problemas y espacio de búsqueda

- **Métodos informados o heurísticos**
  - Introducción
  - Búsqueda primero el mejor
  - Algoritmos de mejora iterativa
  - Búsqueda con adversario
    - Introducción
    - Minimax
    - Poda alfa-beta
    - Consideraciones prácticas

Búsqueda con adversario

- Búsqueda en un entorno hostil, competitivo, impredecible
  - Adversario(s) contrario(s) a nuestros objetivos
- A los problemas de búsqueda con adversario (conflicto de intereses) se les suele denominar juegos
- De cara a simplificar, consideraremos principalmente juegos con las siguientes características:
  - 2 jugadores cuyas jugadas se alternan
  - Al acabar, cada jugador pierde, gana o empata
    - Juegos de suma cero (o nula)
      - Lo que “gana” uno es lo que “pierde” el otro
  - Totalmente observables, todo a la vista
    - Información perfecta (o completa)
      - No interviene el azar

Búsqueda con adversario

- Un árbol de juego es una representación explícita de todas las secuencias de jugadas posibles en una partida
  - Cada nivel representa, alternativamente, las movimientos posibles de cada jugador
  - Las hojas corresponden a estados GANAR, PERDER o EMPATAR
  - Cada camino desde la raíz (el estado inicial del juego) hasta una hoja representa una partida completa
  - El espacio de estados se suele representar como un árbol
- Los algoritmos de búsqueda vistos hasta ahora no sirven
- El problema ya no es encontrar un camino en el árbol de juego (puesto que esto depende de los movimientos futuros que hará el oponente), sino decidir el mejor movimiento dado el estado actual del juego
  - Se podría usar escalada para elegir el siguiente movimiento
    - Lo limitaría a un solo nivel

Minimax

- Se suele denominar MAX al jugador que mueve primero, y al otro MIN
- Son nodos MAX (MIN) aquéllos en los que tiene que jugar MAX (MIN)
  - Si identificamos la raíz con el nivel 0, y comienza jugando MAX, los nodos de nivel par le corresponden a MAX y los de nivel impar a MIN
- En el árbol de juego completo se asigna a los nodos terminales un valor de 1, -1 o 0 según si gana MAX (G), gana MIN (P) o empatan (E)
- A partir de los valores asignados a los nodos terminales, podemos "ascender" los valores hasta la raíz *(propagar hacia arriba)*:
  - A cada nodo MAX se le asigna el máximo de los valores de sus hijos
    - MAX intenta maximizar su ventaja
      - Buscamos el "mejor" movimiento para MAX
  - A cada nodo MIN se le asigna el mínimo de los valores de sus hijos
    - MIN procura minimizar la puntuación de MAX
      - Asumimos que siempre intentará elegir el "peor" movimiento para MAX, es decir, el "mejor" para sus intereses *(siempre jugará óptimamente)*

Minimax

- El valor ascendido a la raíz indica el valor del mejor estado que el jugador MAX puede aspirar a alcanzar
  - La etiqueta de un nodo nos indica lo mejor que podría jugar MAX en el caso de que se enfrentase a un oponente perfecto
- Resolver un árbol de juego significa asignar una etiqueta a la raíz con el método anterior
- Un árbol solución (o estrategia de juego) para MAX es un subárbol del árbol de juego que
  - contiene a la raíz
  - contiene un sucesor de cada nodo MAX no terminal que aparezca en él
  - contiene todos los sucesores de cada nodo MIN que aparezca en él

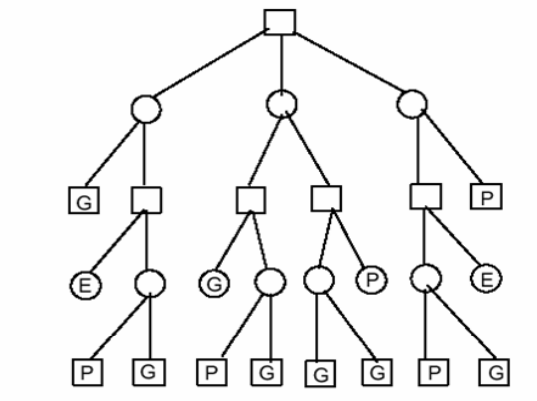
Minimax

- Un árbol solución representa un plan (estrategia) de los movimientos que debe realizar MAX ante cualquier movimiento posible de MIN
  - Si se dispone de un árbol solución para MAX, puede usarse para diseñar un programa que juegue con un contrincante MIN (humano o máquina). No asegura que vaya a ganar
- Un árbol solución para MAX se llamará árbol ganador para MAX (o estrategia ganadora para MAX) si todos los nodos terminales del árbol solución tienen etiqueta 1
  - Un árbol ganador para MAX asegura que el jugador MAX ganará, haga lo que haga MIN

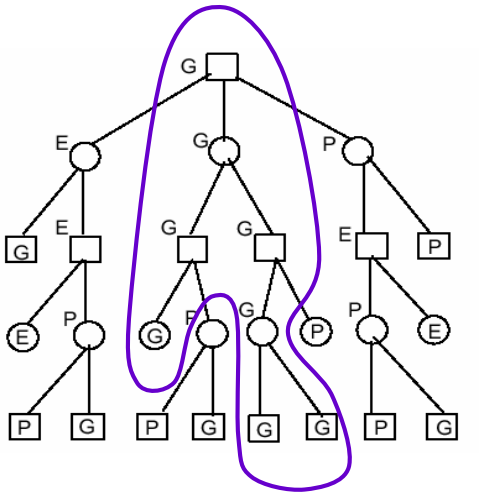
Existirá un árbol ganador para MAX
si y sólo si
al resolver el árbol de juego la etiqueta de la posición inicial es 1

Ejemplo

Árbol de juego sin resolver



Árbol de juego resuelto



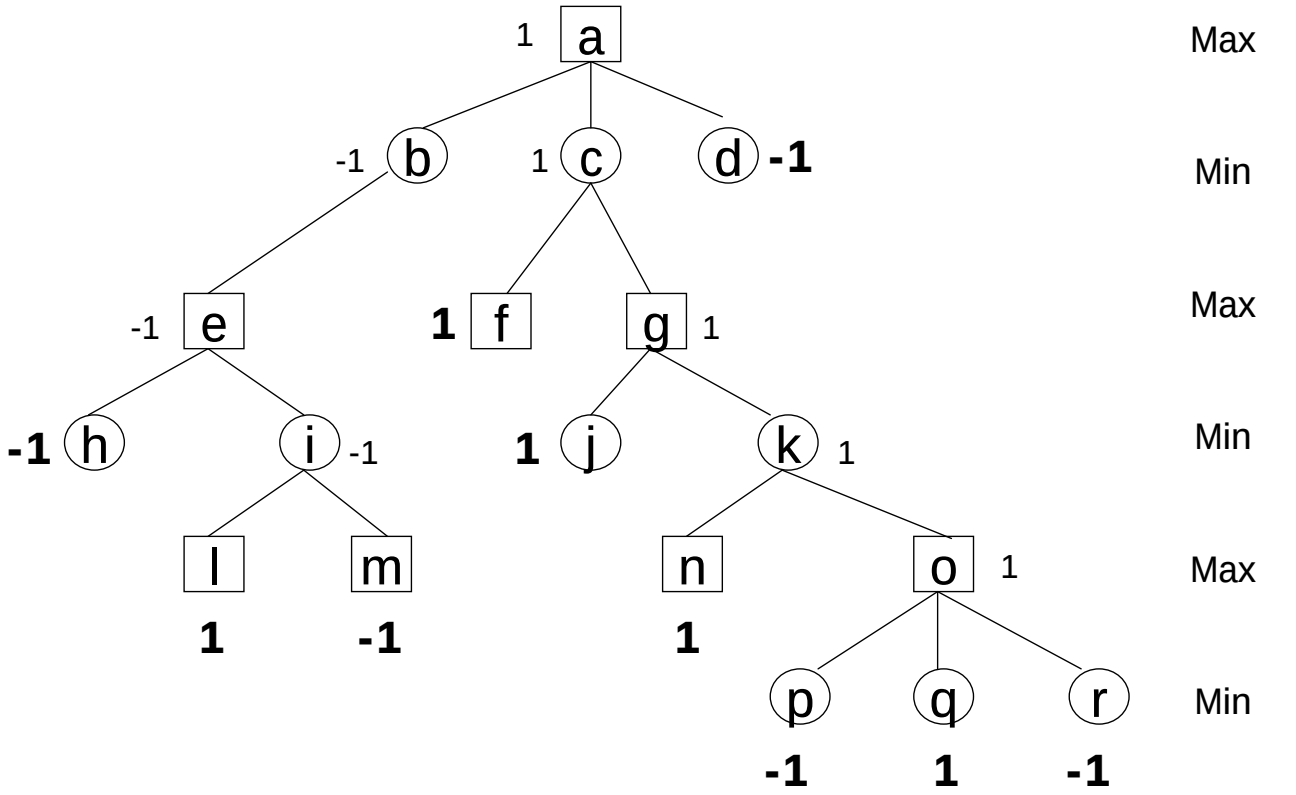
G = 1, E = 0, P = -1
Nodos MAX: cuadrados
Nodos MIN: círculos

Árbol ganador para MAX

Minimax

- El árbol de juego debe ser generado dinámicamente
  - No es necesario mantener todo el árbol en memoria *(generación 1º en profundidad)*
- El método de etiquetado descrito requiere un árbol de juego completo
  - En la práctica, para la mayoría de los juegos, desarrollar todo el árbol de juego es una tarea impracticable
    - Muchos posibles movimientos en cada paso hacen que el árbol crezca enormemente
  - Damas $\approx 10^{40}$ nodos no terminales
    - Generar el árbol completo requeriría 10^{21} siglos (3 billones nodos/seg.)
  - Ajedrez $\approx$ unos 10^{120} nodos y unos 10^{101} siglos
    - Incluso si un adivino nos proporcionase un árbol ganador sería imposible almacenarlo o recorrerlo
- Minimax es exponencial en tiempo por lo que sólo resulta aplicable directamente en juegos muy simples *(árboles de juego manejables)*

Ejemplo



Minimax con estimación en nodos límite

- Aproximación práctica: generar el árbol de juego sólo hasta un cierto nivel límite *(horizonte limitado)*
  - El que permitan los recursos disponibles (tiempo y/o espacio)
  - Cuanto más profundo sea el límite, mayor será el horizonte
- Las "hojas" de ese subárbol no se corresponden con finales de partida
  - No se les puede asignar un valor que refleje si llevarán a ganar o no
- Aproximación heurística: se asigna un valor a esos nodos "hoja" del subárbol de juego según alguna función heurística
  - Estimación heurística de la "bondad" del estado correspondiente (tendencia a ganar, perder o empatar)
    - Nodos terminales (finales de partida) ➔ función de evaluación
    - Nodos límite (nivel de exploración) ➔ función de estimación
      - Valores positivos grandes a los nodos más favorables para MAX
  - Ése es el valor que se propaga hacia arriba

Minimax con estimación en nodos límite

- Se asume que el valor ascendido hasta la raíz, obtenido mediante una profundización hasta un cierto límite *n*, va a ser una estimación mejor que si sólo aplicáramos directamente la función de estimación a los sucesores del nodo raíz
  - Una estrategia de tipo escalada no tendría en cuenta la secuencia de futuras respuestas del oponente y dependería demasiado de la calidad de la función de estimación
- A la raíz le llega la medida heurística del mejor estado alcanzable en *n* movimientos
  - Cuanto mayor sea el horizonte, más seguros son los elementos de decisión para elegir la mejor jugada
  - Así se preven las consecuencias de la jugada a más largo plazo
  - Pero ese "mirar hacia delante limitado" no ofrece garantías

Implementación de minimax

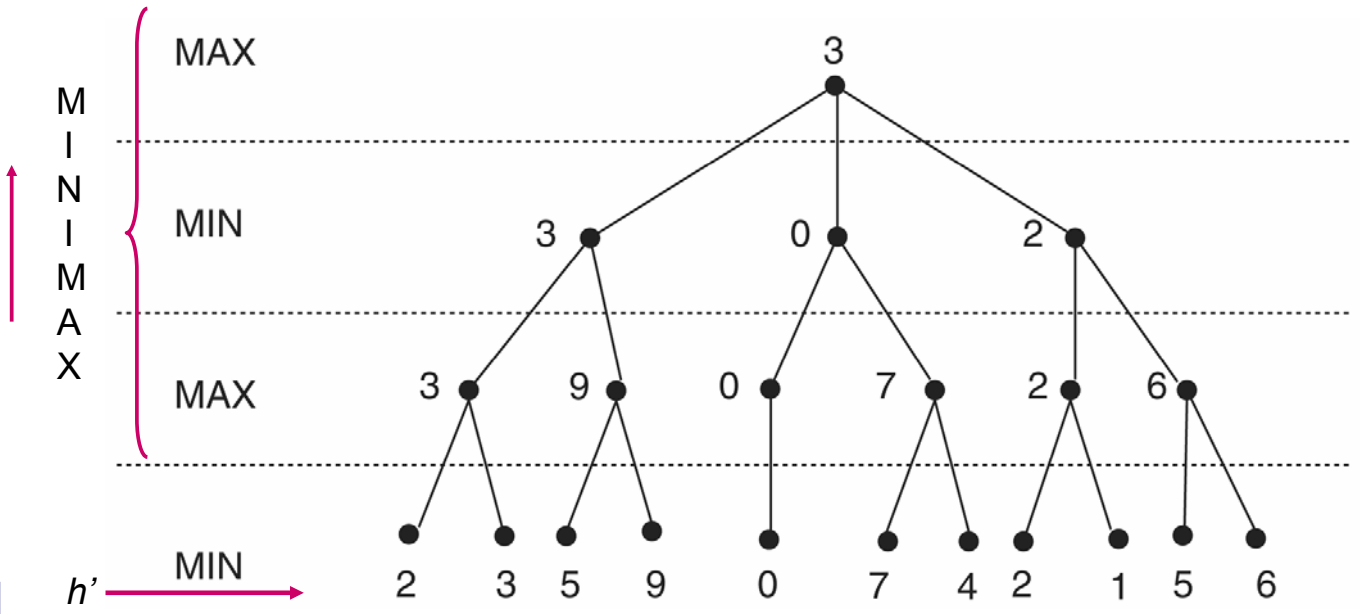
```
/* Llamada inicial: MAX_VALOR (estado, límite) */

función MAX_VALOR (estado, prof) devuelve valor
    si prof = 0 entonces
        devolver estimar(estado)
    si estado es nodo terminal entonces
        devolver evaluar(estado)
    si no
        valor = -∞
        para cada SUCESOR s de estado hacer
            valor = maximo(valor, MIN_VALOR(s, prof-1))
        devolver valor

/* función MIN_VALOR análoga */
```

Ejemplo: valor minimax de 3 capas

Cálculo del valor minimax mirando hacia delante 3 capas (niveles)



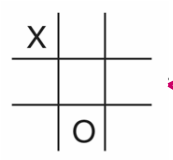
Ejemplo: función de estimación para el tres en raya

Heurística: intentar medir el conflicto del juego $h'(n) = M(n) - O(n)$

- $M(n)$ = nº de mis posibles líneas ganadoras *(contando vacías)*
- $O(n)$ = nº de sus posibles líneas ganadoras *(contando vacías)*

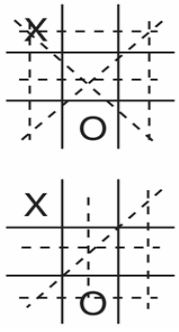
- $+\infty$ si final: MAX gana
- $-\infty$ si final: MIN gana

Para evitar confusión con los valores de *h'*, se distinguen los finales de partida con valores fuera del rango de *h'*



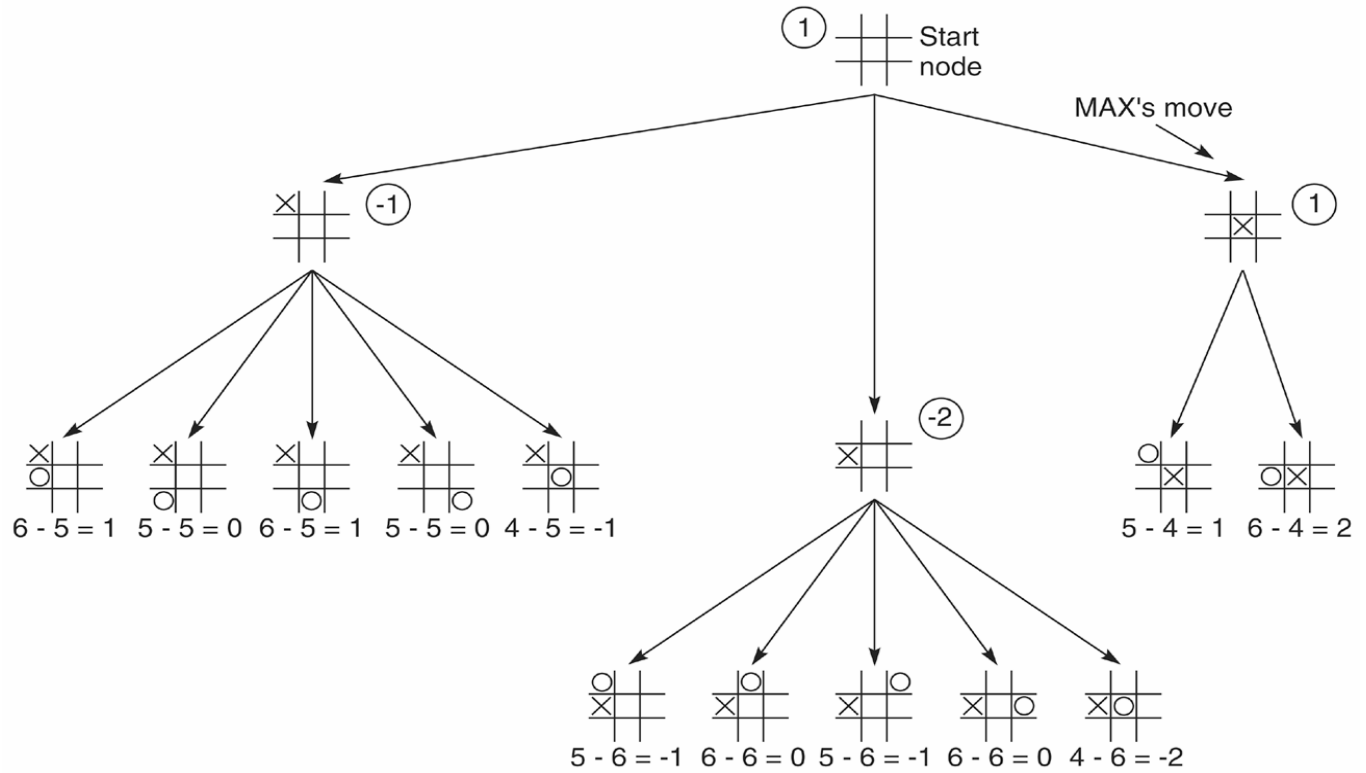
X tiene 6 posibles líneas ganadoras

O tiene 5 posibles líneas ganadoras

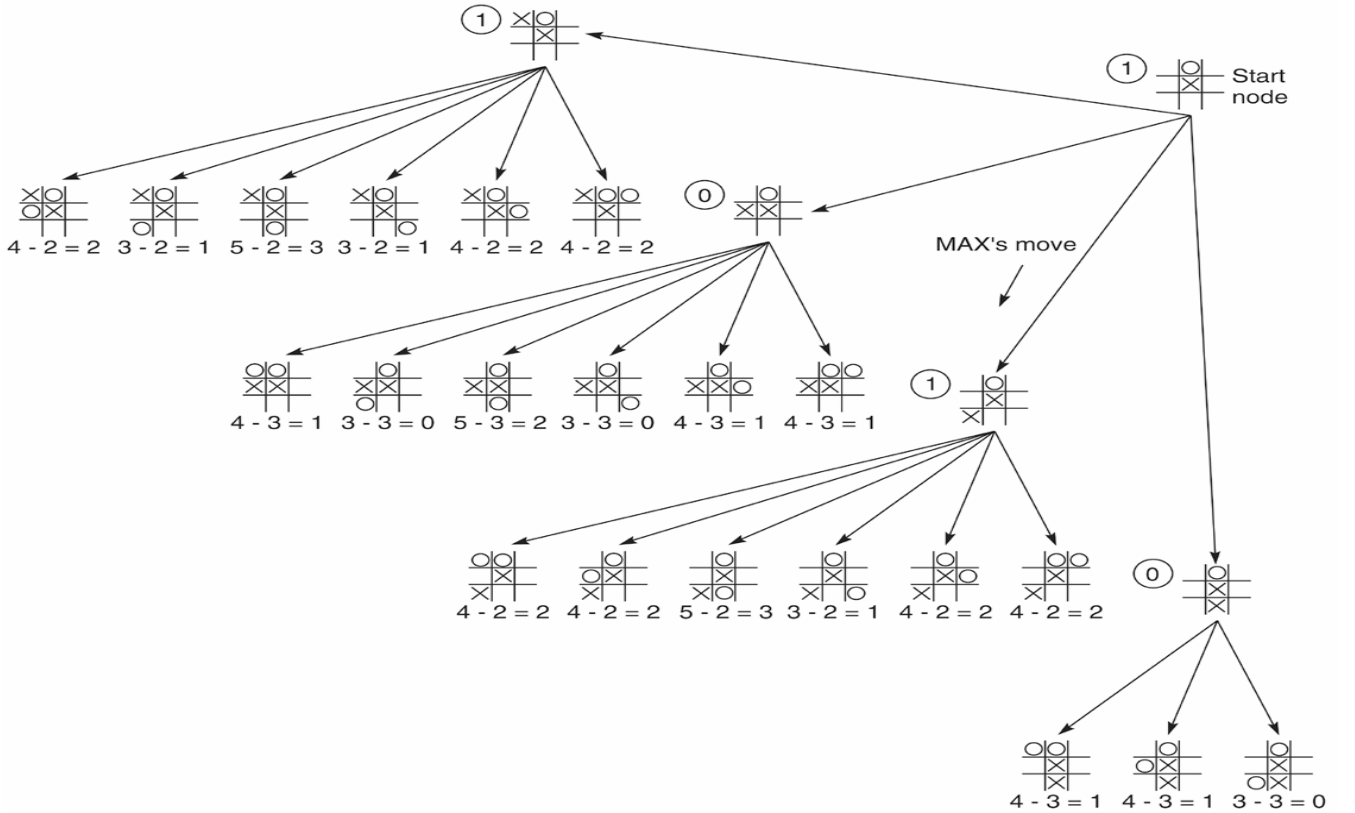


$h'(n) = 6 - 5 = 1$

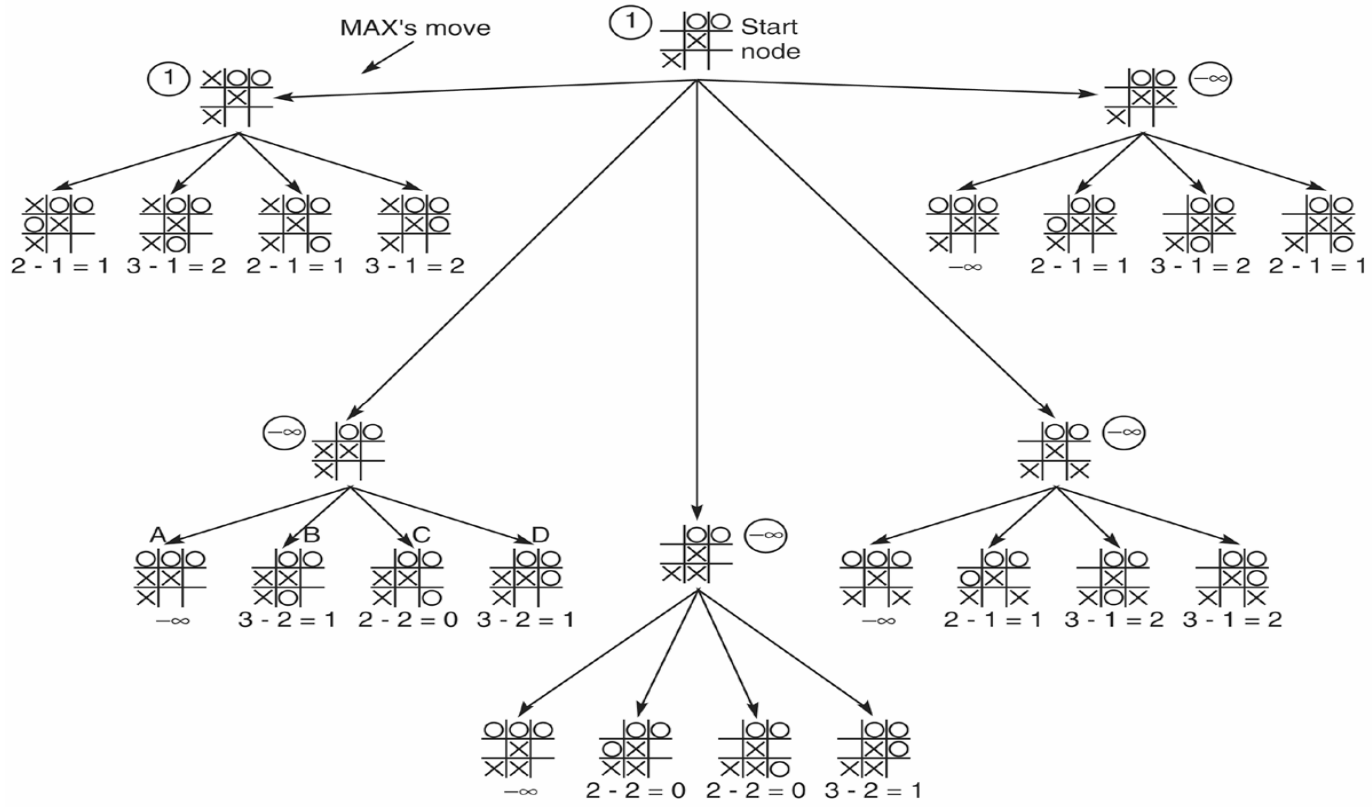
Movimiento inicial mirando 2 capas por delante



Segundo movimiento de MAX (2 capas)



Movimiento de MAX cercano al final (2 capas)



La función de estimación

- En los nodos terminales (de final de partida), el valor de la función de estimación debe coincidir con el de la función de evaluación
- La función de estimación *(heurística)* suele tener en cuenta distintas características del estado del juego ponderadas por distintos pesos:
  - Número de piezas propias y del contrario (ventaja de uno sobre el otro)
  - Ponderación de la importancia de las piezas en ajedrez
  - Posiciones de las piezas en el tablero
  - Puntos débiles (peón aislado, etc.)
  - Características posicionales (protección del rey, capacidad de maniobra, control del centro del tablero, etc.)
- Cuanto más tiempo se gaste calculando la función de estimación, menos tiempo se puede dedicar a la búsqueda

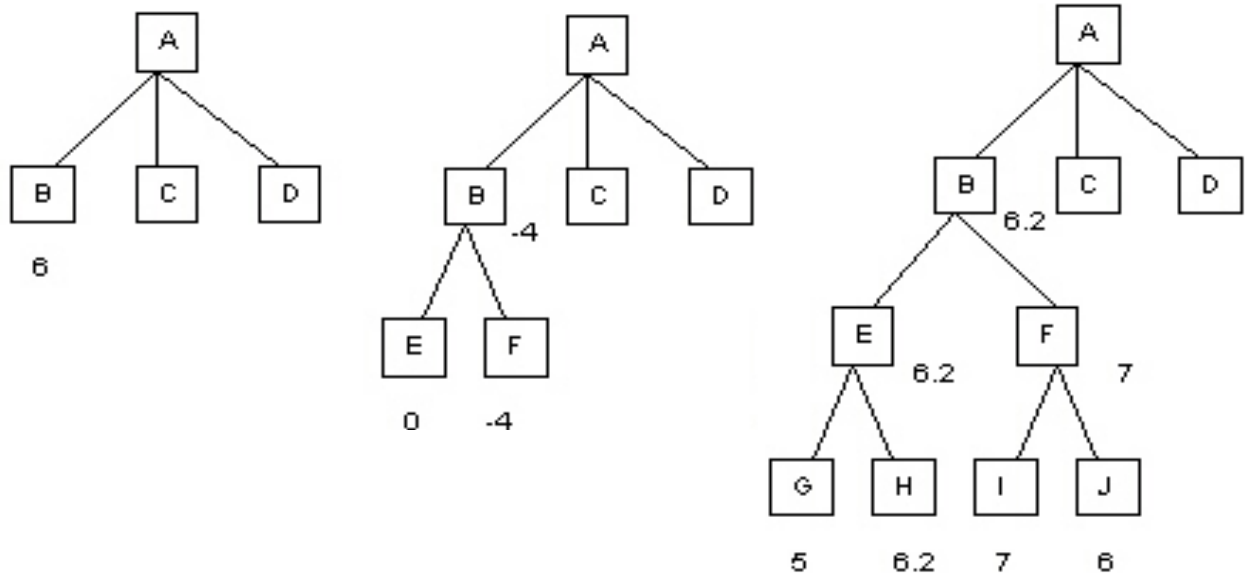
Hay que llegar a un compromiso entre
la calidad de la estimación y su coste

La profundidad límite de exploración

- Lo más sencillo es realizar búsquedas hasta una profundidad fija (la máxima que nos permitan nuestros recursos)
  - Pero puede darse el efecto horizonte: se evalúa como bueno o malo un nodo sin saber que en la siguiente jugada la situación se revierte
- Resulta más efectivo elegir una profundidad menor, explorar todas las ramas hasta esa profundidad y dedicar los recursos ahorrados a profundizar más en ciertas ramas

  ➔ Búsquedas secundarias:
  - Búsqueda de la quietud (o espera del reposo)
    - Continuar la búsqueda hasta alcanzar una situación estable *(aparentemente)*
  - Extensiones singulares
    - Si un nodo hoja es muy diferente a sus hermanos, el nodo se expande una capa más por si se da una situación de captura inminente (jugada forzada)
  - Otros tipos
    - Movimientos de libro (aperturas y finales de partida)...

Búsqueda de la quietud

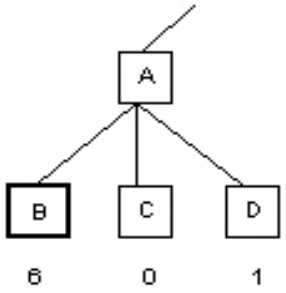


Explorando un nivel la estimación de B es 6

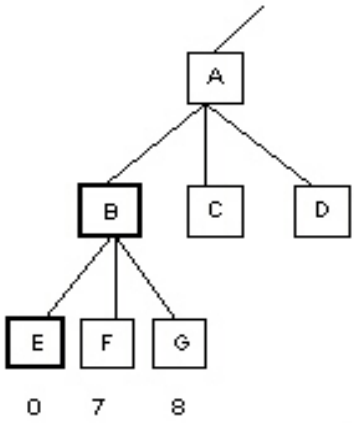
Se explora 1 nivel más y hay un cambio drástico

Reposo alcanzado: estimación similar a la inicial

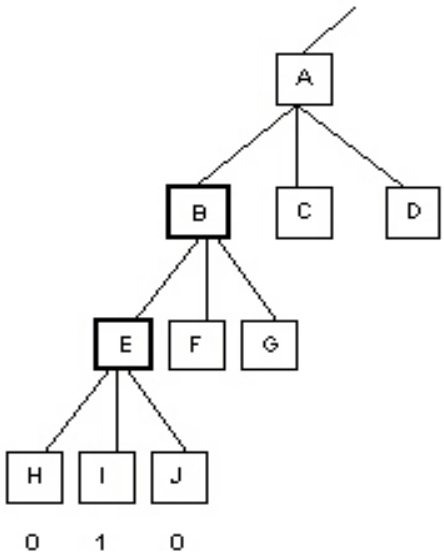
Extensiones singulares



B tiene un valor superior al resto de sus hermanos. ¿Captura inminente?



¿Captura por parte del oponente?



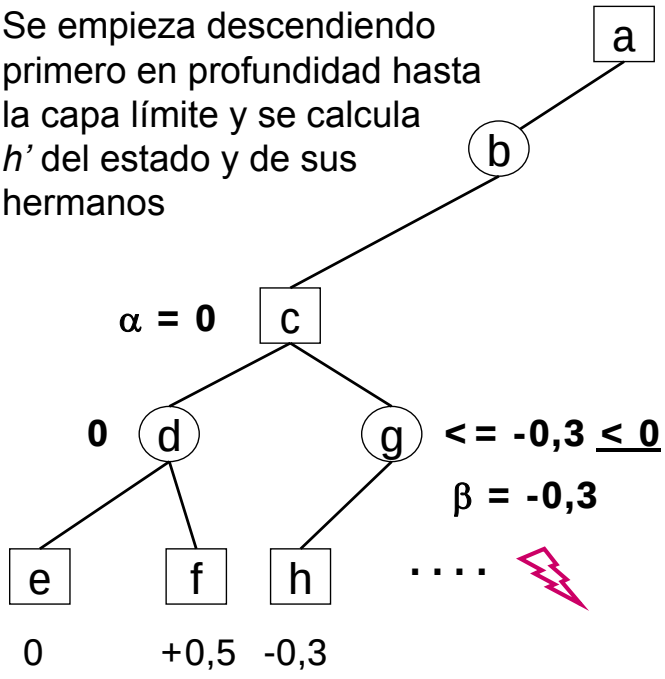
No hay captura: valores en un intervalo estrecho. Se interrumpe la búsqueda secundaria

Poda alfa-beta

- Optimización de minimax, que genera el espacio de búsqueda entero y después evalúa y propaga (recorrido doble)
  - Un solo recorrido: generación y evaluación simultáneas
  - Poda: algunas ramas del árbol no necesitan ser analizadas
    - Abandono de soluciones parciales por ser peores que otras
- Alfa-beta realiza una búsqueda de tipo primero en profundidad
- Para cada nodo MAX se calcula un valor α que representa el valor máximo de los sucesores de MAX generados hasta ese momento
  - El valor α representa una cota inferior para el valor final que pueda alcanzar el nodo MAX (lo peor que le podría ir a MAX)
    - Se inicializa a $-\infty$ y nunca puede decrecer
- Para cada nodo MIN se calcula un valor β que representa el valor mínimo de los sucesores de MIN generados hasta ese momento
  - El valor β representa una cota superior para el valor final que pueda alcanzar el nodo MIN (lo mejor que le podría ir a MIN)
    - Se inicializa a $+\infty$ y nunca puede crecer

Poda alfa

Se empieza descendiendo primero en profundidad hasta la capa límite y se calcula *h'* del estado y de sus hermanos



¡MAX nunca elegirá g!

Si son nodos MAX, el mínimo va a al padre y este valor se ofrece al abuelo como valor provisional α

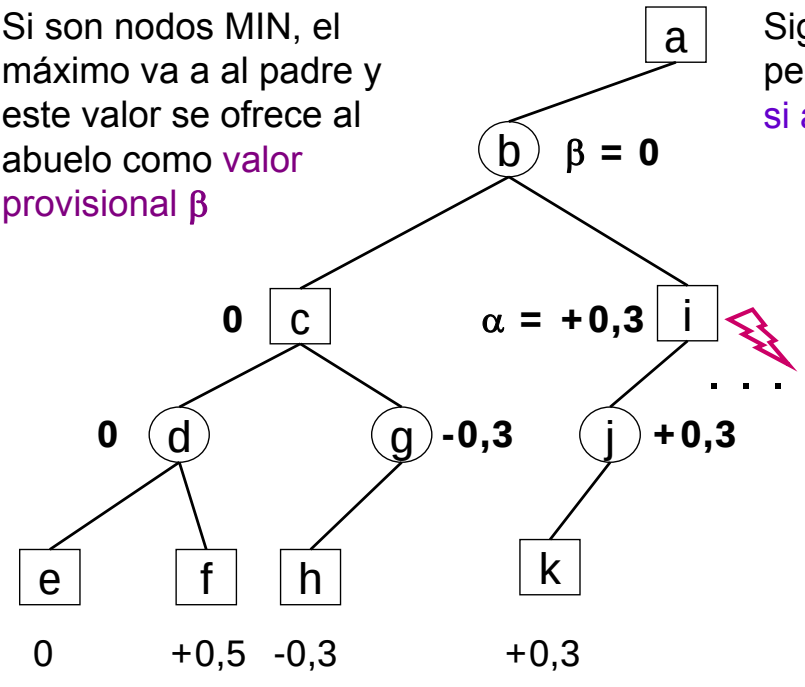
Sigue descendiendo a otros nietos, pero termina la exploración del padre si alguno de sus valores es <= α

Se puede cortar la búsqueda por debajo de cualquier nodo MIN cuyo valor β sea menor o igual que el valor α de algún antecesor MAX

A ese nodo MIN se le asigna definitivamente su valor β que puede no coincidir con el que habría obtenido minimax *(podría ser <)* pero esto no afecta a la elección de movimiento

Poda beta

Si son nodos MIN, el máximo va a al padre y este valor se ofrece al abuelo como valor provisional β



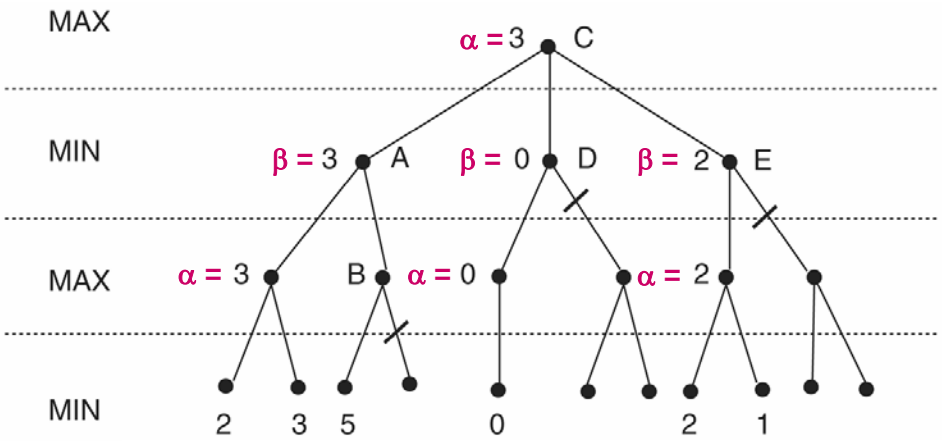
¡MIN nunca elegirá i!

Sigue descendiendo a otros nietos, pero termina la exploración del padre si alguno de sus valores es >= β

Se puede cortar la búsqueda por debajo de cualquier nodo MAX cuyo valor α sea mayor o igual que el valor β de algún antecesor MIN

A ese nodo MAX se le asigna definitivamente su valor α que puede no coincidir con el que habría obtenido minimax *(podría ser >)* pero esto no afecta a la elección de movimiento

Ejemplo



A tiene β = 3 (A <= 3)

B es β-podado (B >= 5 y 5 > 3)

C tiene α = 3 (C >= 3)

D es α-podado (D <= 0 y 0 < 3)

E es α-podado (E <= 2 y 2 < 3)

C tiene valor 3 definitivo (valor minimax)

Alfa-beta (inicialización y poda beta)

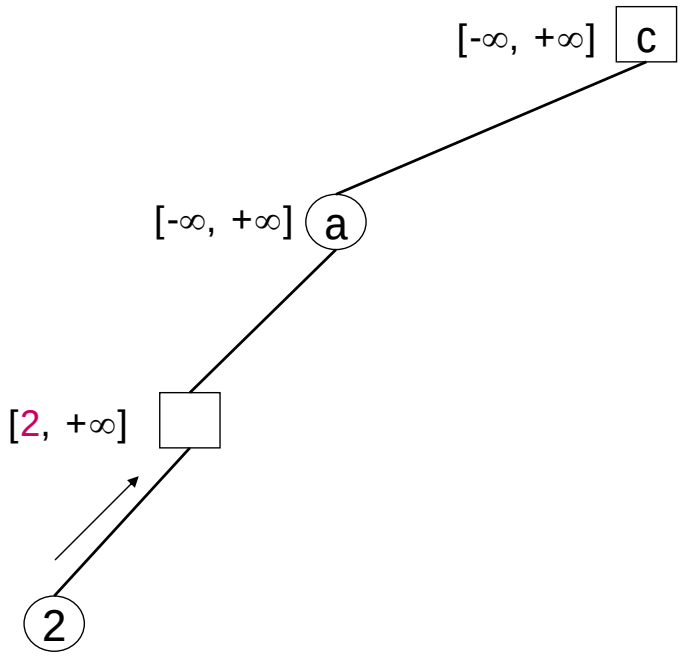
```
función BÚSQUEDA_ALFA_BETA (estado) devuelve valor
    {permite seleccionar un movimiento con ese valor}
      valor = MAX_VALOR (estado, -∞, +∞) {estado es un nodo MAX}
      devolver valor

función MAX_VALOR (estado, alfa, beta) devuelve valor
    si estado es un nodo terminal entonces
          devolver evaluar(estado)
    si no
          valor = -∞ {antes de empezar a descender por el primer hijo}
          para cada SUCESOR s de estado hacer
                valor = maximo(valor, MIN_VALOR(s, alfa, beta))
                si valor >= beta entonces devolver valor {poda}
                si no alfa = maximo(alfa, valor)
          devolver valor
```

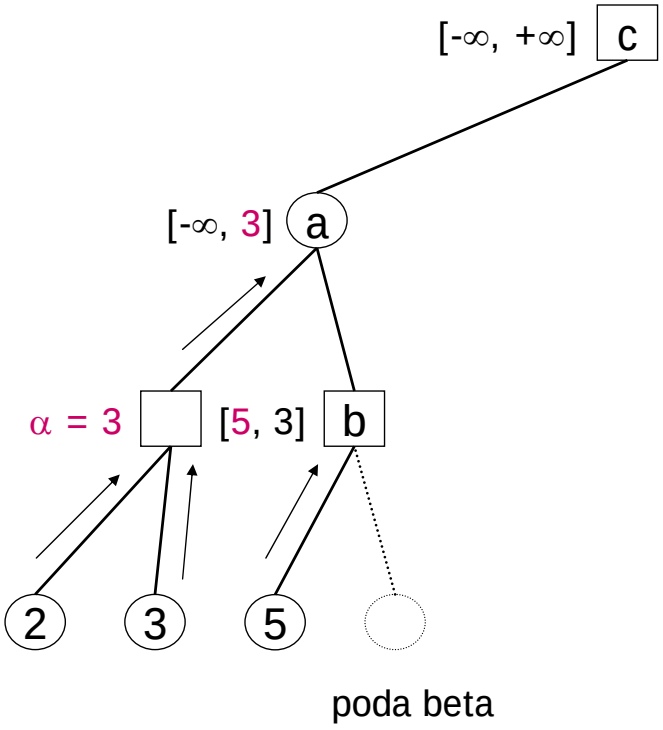
Alfa-beta (poda alfa)

```
función MIN_VALOR (estado, alfa, beta) devuelve valor
    si estado es un nodo terminal entonces
        devolver evaluar(estado)
    si no
        valor = +∞ {antes de empezar a descender por el primer hijo}
        para cada SUCESOR s de estado hacer
            valor = minimo(valor, MAX_VALOR(s, alfa, beta))
            si valor <= alfa entonces devolver valor {poda}
            si no beta = minimo(beta, valor)
        devolver valor
```

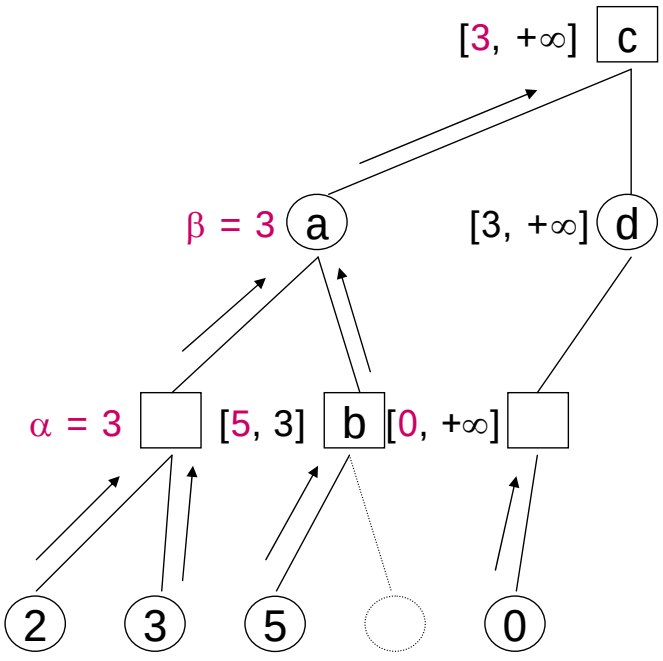

Ejemplo paso a paso



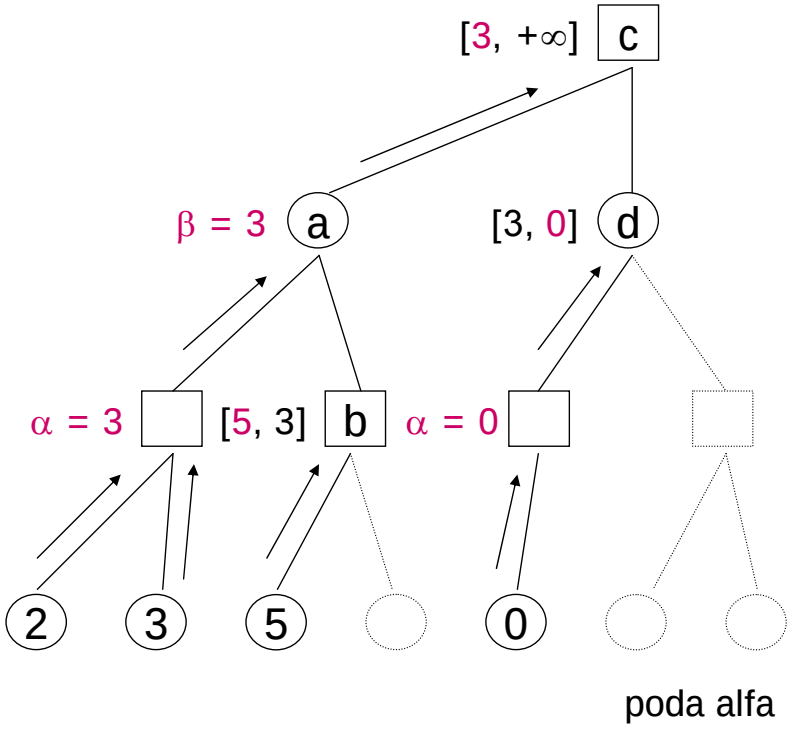
Ejemplo paso a paso



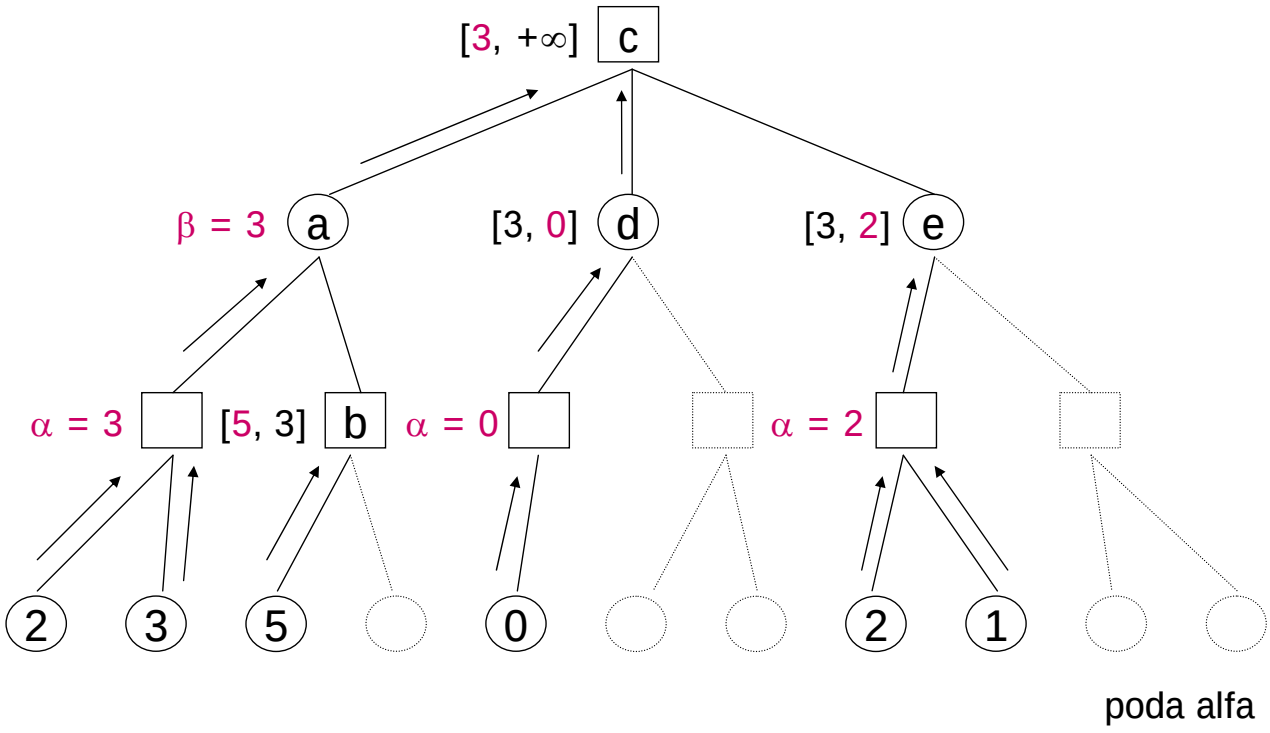
Ejemplo paso a paso



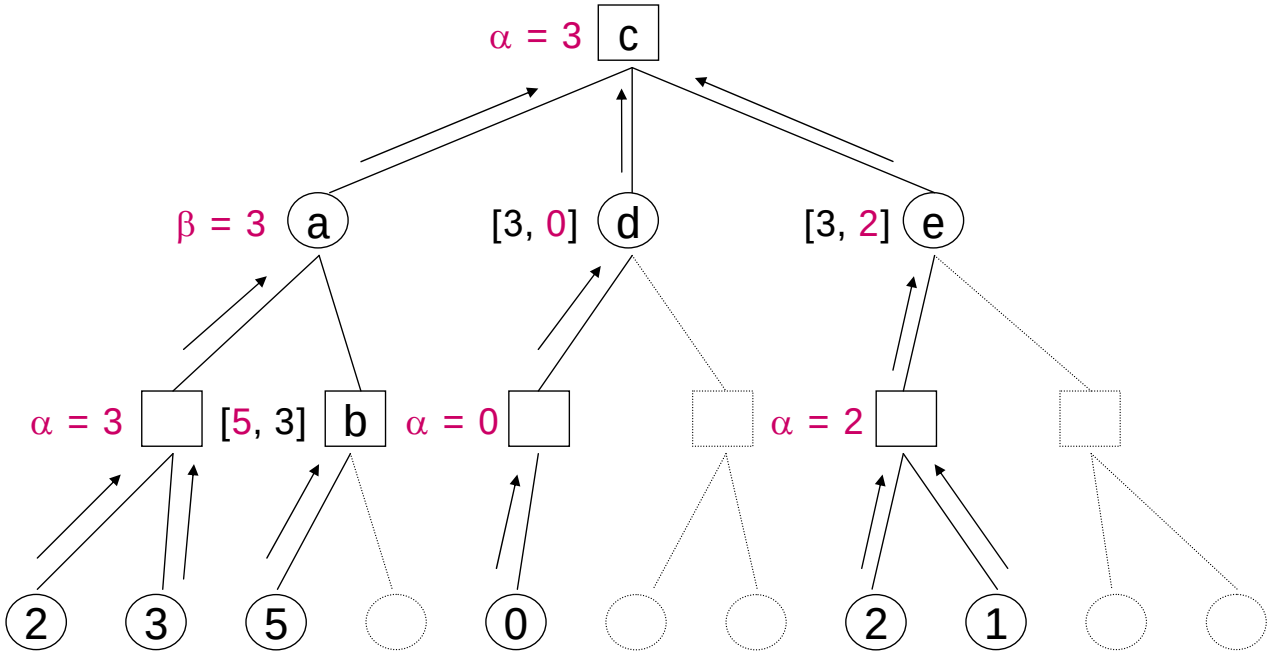
Ejemplo paso a paso



Ejemplo paso a paso



Ejemplo paso a paso

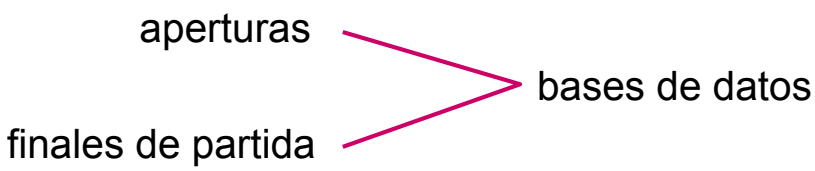


Propiedades de la poda alfa-beta

- Este algoritmo es de tipo ramificación y poda *(branch & bound)*
- Garantiza encontrar el mismo mejor movimiento (u otro equivalente con el mismo valor) que minimax, pero de forma más eficiente
- Si el mejor nodo límite es el generado primero, las podas serán máximas y habrá que generar y evaluar el mínimo número de nodos
  - Influye mucho el orden de generación de los sucesores
  - En esas condiciones, alfa-beta tiene que examinar sólo $O(r^{p/2})$ nodos para escoger el mejor movimiento, en vez de $O(r^p)$ con minimax
    - El factor de ramificación efectivo sería $\sqrt{r}$ en lugar de r
    - Permite profundizar aproximadamente el doble (= tiempo)
- Si pudiera utilizarse una función de ordenación en la generación de nodos, se garantizaría *(si la tuviéramos, jugaríamos perfecto...)*
  - Para un nodo MAX debería generar primero el hijo de mayor valor
  - Para un nodo MIN el de menor valor
  - Al menos podemos usar la función de estimación...

Ajedrez: consideraciones prácticas

	Tiempo
1. Generación de movimientos (ordenados por *h'*)	50%
❑ para máximo aprovechamiento de la poda alfa-beta	
2. Evaluación estática	40%
❑ función de estimación a las hojas	
3. Búsqueda	¡10%!



- ❑ Todas estas ideas estaban ya claras a finales de los 60
  - ❑ Mejoras por incremento en la potencia de cómputo de los ordenadores
  - ❑ Y por unas pocas nuevas ideas

Ajedrez: consideraciones prácticas

- ❑ Factor de ramificación altamente variable (poda alfa-beta)
  - ❑ Con límite fijo en la exploración, unas veces va rápido, otras muy lento
- ❑ Se usa profundización iterativa
  - ❑ Así siempre se tiene disponible un movimiento (tiempo limitado)
  - ❑ Ordenación de movimientos en función de los resultados de la última iteración
  - ❑ Uso de los resultados alfa y beta de la última iteración para inicializar los valores en la siguiente
    - ❑ Ayuda a efectuar más podas de forma temprana
- ❑ Efecto horizonte
  - ❑ Con límite fijo en la exploración, es más fácil que se produzca
  - ❑ Búsqueda de la quietud
    - ❑ Se continúa la búsqueda en esos nodos hoja
    - ❑ Deep Blue los explora hasta 30 capas por delante
- ❑ Paralelización *(resultó complicadísimo; > innovación de Deep Blue)*

Juegos: estado actual

- Programas que superan a los mejores jugadores humanos
  - Damas: CHINOOK (1994)
    - Derrotó al campeón mundial humano durante 40 años Marion Tinsley
    - BD de finales de partida: juego perfecto para configuraciones de tablero con 8 o menos piezas
  - Othello: LOGISTELLO (1997)
    - Los campeones humanos se niegan a medirse con programas tan buenos
  - Scrabble: MAVEN (1998)
- Programas competitivos con los mejores jugadores humanos
  - Ajedrez: DEEP BLUE (1997)
  - Backgammon: TD-GAMMON (1995, *aprendizaje por refuerzo*)
    - Complicación: aleatoriedad (tiradas de dados)
    - Los programas desarrollados por humanos son muy malos
    - En cambio, un sistema de aprendizaje máquina (mucha búsqueda y uso de los resultados para construir una muy buena función heurística) lo consiguió

Juegos: estado actual

- Juegos que presentan dificultades con los métodos actuales
  - Bridge
    - Información oculta (las cartas de los otros jugadores)
    - Comunicación con el compañero mediante un lenguaje restringido
    - Los jugadores máquina no son nada buenos en la fase del juego que involucra comunicación humana
  - Go
    - Como el ajedrez: información perfecta, no aleatoriedad, ni comunicación
    - Problema: el enorme factor de ramificación ($r > 300$)
      - Los métodos de búsqueda que funcionan bien en ajedrez no son susceptibles de aplicarse en el go
      - Los jugadores humanos parecen basarse en algo mucho más complejo: comprensión de patrones espaciales
      - Planteamiento de métodos basados en mejores heurísticas y menos en búsqueda por fuerza bruta, con bases de conocimiento de patrones
  - Poker
- Para más info, visitar: www.gameai.com/clagames.html

Observaciones

- Observaciones sobre juegos aplicables a la aproximación simbólica a la IA
  - Las máquinas superan a los humanos en áreas perfectamente definidas en las que las reglas están claras
    - Ajedrez
    - Matemáticas
  - Las áreas que siguen resultando extremadamente complicadas en la IA son más nebulosas
    - Lenguaje, visión y sentido común
    - La mayoría de la investigación actual está centrada en estas actividades no tan bien definidas
  - Los éxitos se han obtenido tras muchos años de refinamientos graduales incluso en actividades bien definidas como el ajedrez
    - No debemos esperar éxitos en el corto plazo en los grandes retos de la IA

Bibliografía

- **Russell, S. y Norvig, P.**
  Inteligencia Artificial: Un Enfoque Moderno.
  Prentice Hall, 2004, 2ª edición.
  - **Capítulos 3, 4 y 6**
- **Luger, G.F.**
  Artificial Intelligence.
  Addison-Wesley, 2005, 5ª edición.
  - **Capítulos 2, 3, 4 y 6**
- **Rich, E. y Knight, K.**
  Artificial Intelligence.
  McGraw-Hill, 1991, 2ª edición.
  - **Capítulos 2 y 3**

Bibliografía

- **Nilsson, J.**
  Artificial Intelligence: A New Synthesis.
  Prentice Hall, 2004, 2ª edición.
  - **Capítulos 7, 8, 9, 10, 11 y 12**
- **J. J. Rubio García, P. R. Muro Medrano y J. A. Bañares Bañares**
  Apuntes de búsqueda para IAIC 1.
  Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, 1998, versión 1.0.
- **T. Lozano-Perez y L. Kaelbling**
  Artificial Intelligence.
  Electrical Engineering and Computer Science, MIT OpenCourseWare, Massachusetts Institute of Technology, 2003.
  - **Capítulo 2: *Search Handout***

Minimax dependiente de adversario

¡NO CONTAR!