Sistemas Inteligentes

Tema 2:
Resolución de problemas mediante búsqueda

Contenido

- Introducción
- Búsqueda no informada
- Búsqueda heurística
- Búsqueda entre adversarios

1. Introducción

- La resolución de problemas es una capacidad que se considera inteligente.
  - Encontrar el camino en un laberinto.
  - Resolver un crucigrama.
  - Jugar a un juego.
  - Diagnosticar una enfermedad.
  - Decidir si invertir en bolsa.
  - ...
- El objetivo es que un programa sea capaz de resolver estos problemas.

1. Introducción

- Es necesario definir cualquier tipo de problema de manera que se pueda resolver automática-mente.
- Para ello hace falta:
  - Una representación común para todos los problemas.
  - Algoritmos que usen alguna estrategia para resolver problemas definidos en esa representación común.

1. Introducción

Definición de un problema

- En un problema se pueden identificar los siguientes elementos:
  - Punto de partida.
  - Objetivo a alcanzar.
  - Acciones disponibles.
  - Restricciones sobre el objetivo.
  - Elementos relevantes definidos por el tipo de dominio.

1. Introducción

Representación de problemas

- Representaciones generales:
  - Espacio de estados: un problema se divide en un conjunto de pasos de resolución desde el inicio hasta el objetivo.
  - Reducción a subproblemas: un problema se descompone en subproblemas.
- Representaciones para problemas específicos:
  - Resolución de juegos.
  - Satisfacción de restricciones.

1. Introducción

Representación de problemas: estados

- Un estado es la representación de los elementos que describen el problema en un determinado momento.
- Hay dos estados especiales:
  - Estado inicial: punto de partida
  - Estado final: objetivo del problema

1. Introducción

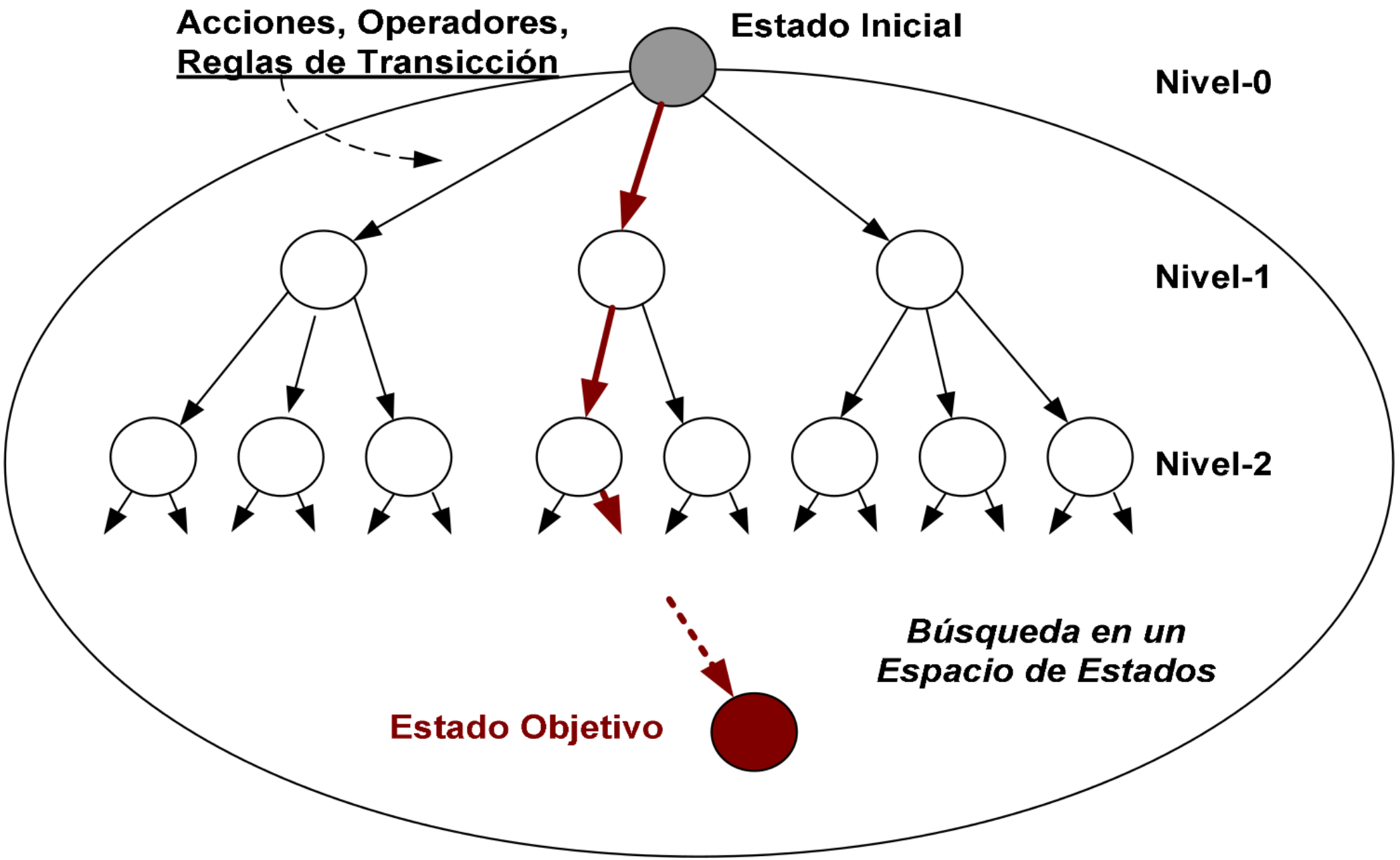
Modificación del estado: operadores

- Para moverse entre los estados hacen falta operadores de transformación.
- Operador: función de transformación sobre un estado que lo convierte en otro estado.
- Los operadores definen una relación de accesibilidad entre estados.
- Representación de un operador:
  - Condiciones de aplicabilidad.
  - Función de transformación.

Espacio de estados

- Los estados y su relación de accesibilidad definen el espacio de estados.
- Representa todos los caminos que hay entre todos los estados posibles de un problema.
- Es parecido a un mapa de carreteras de un problema.
- La solución del problema está dentro de ese mapa.

1. Introducción



Solución de un problema en un espacio de estados

- Solución: secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o también el estado final.
- Tipos de solución: una cualquiera, la mejor, todas.
- Coste de una solución: gasto en recursos de la aplicación de los operadores a los estados. Puede ser importante según el tipo de problema y la solución buscada.

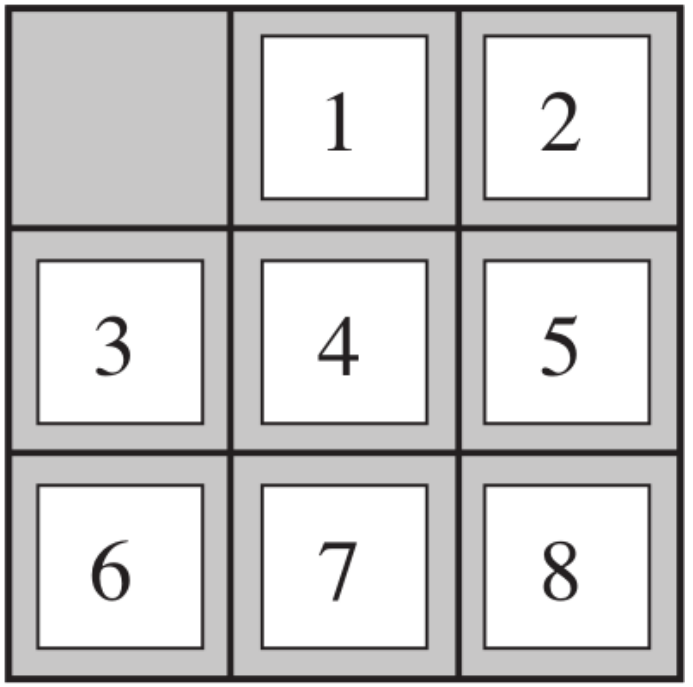
1. Introducción

Descripción de un problema en un espacio de estados

- Conjunto de estados del problema.
- Estado inicial.
- Estado final o las condiciones que cumple.
- Operadores de cambio de estado (condiciones de aplicabilidad y función de transformación).
- Tipo de solución:
  - Secuencia de operadores o estado final.
  - Una solución cualquiera, la mejor, ...

1. Introducción

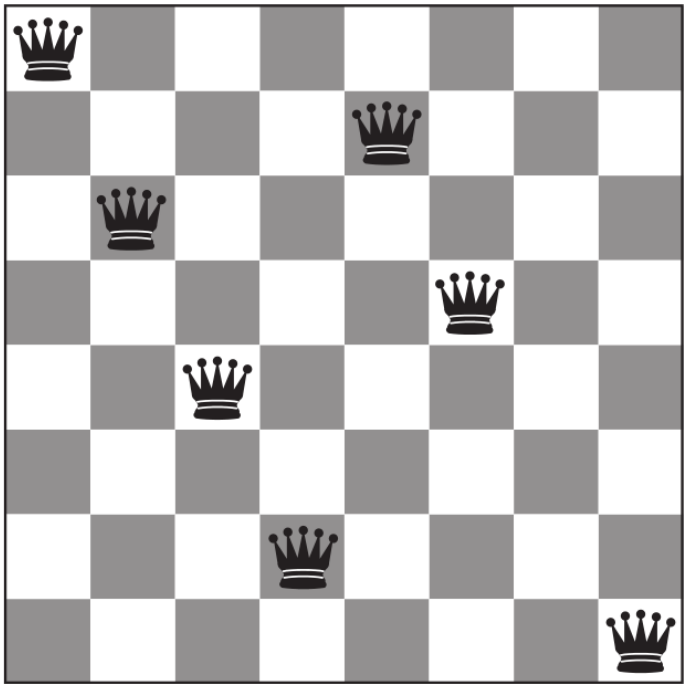
Ejemplo: 8 puzzle



- Espacio de estados: configuraciones de 8 fichas en el tablero.
- Estado inicial: cualquier configuración.
- Estado final: fichas ordenadas.
- Operadores: mover hueco.
  - Condiciones: el movimiento está dentro del tablero.
  - Transformación: intercambio entre el hueco y la ficha en la posición del movimiento.
- Solución: los pasos + el mejor número.

1. Introducción

Ejemplo: 8 reinas



- Espacio de estados: configuraciones de 0 a 8 reinas en el tablero.
- Estado inicial: tablero vacío.
- Estado final: 8 reinas que no se matan entre ellas.
- Operadores: colocar una reina.
  - Condiciones: la reina no debe ser matada por otra ya colocada.
  - Transformación: colocar una reina más en el tablero en una fila y columna.
- Solución: una cualquiera, no importan los pasos.

1. Introducción

Búsqueda en el espacio de estados

- Para resolver un problema de búsqueda hay que explorar el espacio de estados.
- Se parte del estado inicial evaluando cada paso hasta encontrar un estado final.
- En el caso peor se explorarán todos los posibles caminos entre el estado inicial y el final.

1. Introducción

Estructura del espacio de estados

- Estructuras de datos: árboles y grafos
- Estados: nodos
- Operadores: arcos dirigidos entre nodos
- Árboles: sólo un camino lleva a un nodo
- Grafos: varios caminos pueden llevar a un nodo

Algoritmo básico

- El espacio de estados puede ser infinito.
- Es necesaria una aproximación diferente para buscar y recorrer árboles y grafos (no podemos tener la estructura en memoria).
- La estructura se construye durante la búsqueda

Algoritmo básico

función: búsqueda_en_espacio_de_estados
datos: estado_inicial
resultado: una_solución

estado_actual ← estado_inicial

mientras estado_actual ≠ estado_final

    expansión: generar y guardar sucesores del estado actual.

    selección: escoger el siguiente estado entre los pendientes.

fin

Algoritmo básico

- Nodos abiertos (frontera): estados generados pero todavía no visitados.
- Nodos cerrados: estados visitados y que ya se han expandido.
- Hace falta una estructura para almacenar los nodos abiertos.
- La política de inserción en la estructura determina el tipo de búsqueda.
- Si se explora un grafo puede ser necesario tener en cuenta los estados repetidos.

Características de los algoritmos

- Completitud: ¿encontrará la solución?
- Complejidad temporal: ¿cuánto tardará?
- Complejidad espacial: ¿cuánta memoria será necesaria?
- Optimalidad: ¿encontrará la mejor solución?

Algoritmo general de búsqueda

```
función: búsqueda_general
abiertos.insertar(estado_inicial)
actual ← abiertos.primero()
mientras !esFinal(actual) && !abiertos.esVacia()
        abiertos.borrarPrimero()
        cerrados.insertar(actual)
        hijos ← generarSucesores(actual)
        hijos ← tratarRepetidos(hijos, cerrados, abiertos)
        abiertos.insertar(hijos)
        actual ← abiertos.primero()
fin
```

Algoritmo general de búsqueda

- La estructura de abiertos determina el orden de visita de los nodos.
- La función generarSucesores seguirá el orden de generación de sucesores definido en el problema.
- El tratamiento de los nodos repetidos dependerá de cómo se visiten los nodos.

2. Búsqueda no informada

- Las estrategias de búsqueda no informada (búsqueda a ciegas) no saben si un estado no objetivo es mejor que otro.
- Siguen un orden de visitas establecido por la estructura del espacio de búsqueda hasta encontrar un estado objetivo.
- Algoritmos de búsqueda no informada:
    - Búsqueda primero en anchura.
    - Búsqueda primero en profundidad.
    - Búsqueda en profundidad iterativa.

2. Búsqueda no informada

Búsqueda primero en anchura

- Visita los nodos y los genera por niveles.
- La estructura para los nodos abiertos es una cola (FIFO).
- Un nodo es visitado cuando todos los nodos de los niveles superiores y sus hermanos precedentes han sido visitados.

Búsqueda primero en anchura

- Completitud: el algoritmo encuentra una solución.
- Complejidad temporal: exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^p)$
- Complejidad espacial: $O(r^p)$
- Optimalidad: la solución que se encuentra es óptima en número de niveles desde la raíz, en problemas de coste unitario.

Búsqueda primero en profundidad

- Visita los nodos y los genera profundizando y retrocede cuando no encuentra sucesores.
- La estructura para los nodos abiertos es una pila (LIFO).
- Para garantizar que el algoritmo acaba hay que imponer un límite en la profundidad de exploración.

Búsqueda primero en profundidad

- Completitud: el algoritmo encuentra una solución si ésta está dentro del límite de profundidad.
- Complejidad temporal: exponencial respecto al factor de ramificación y la profundidad de la solución $O(r^p)$
- Complejidad espacial:
  - controlando repetidos: $O(r^p)$
  - no controlando repetidos: $O(rp)$
  - recursivo sin controlar repetidos: $O(p)$
- Optimalidad: no se garantiza.

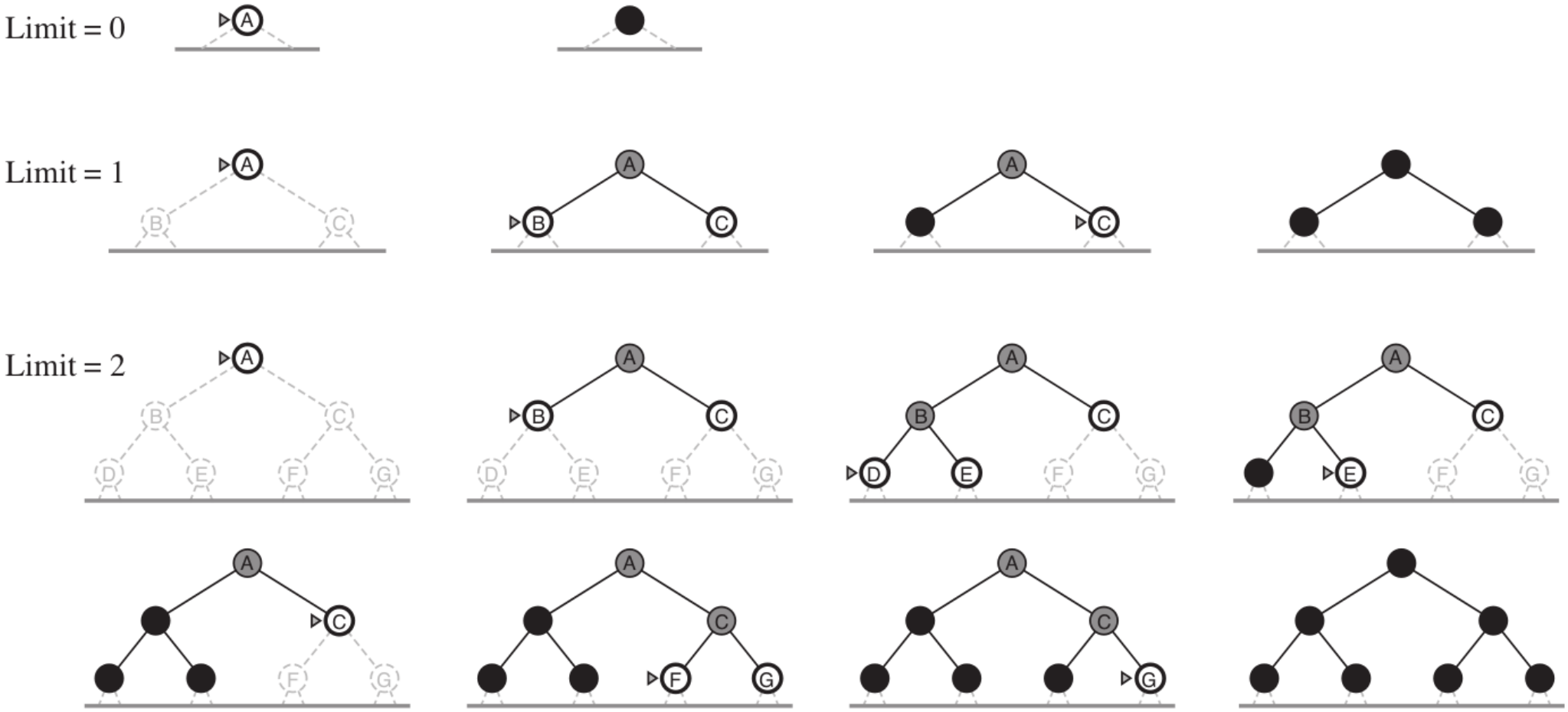
Búsqueda primero en profundidad

```
función: búsqueda_profundidad_limitada(límite: entero)
abiertos.insertar(estado_inicial)
actual ← abiertos.primero()
mientras !esFinal(actual) && !abiertos.esVacia()
    abiertos.borrarPrimero()
    cerrados.insertar(actual)
    si profundidad(actual) ≤ límite
        hijos ← generarSucesores(actual)
        hijos ← tratarRepetidos(hijos, cerrados, abiertos)
        abiertos.insertar(hijos)
    fin
    actual ← abiertos.primero()
fin
```

Búsqueda en profundidad iterativa

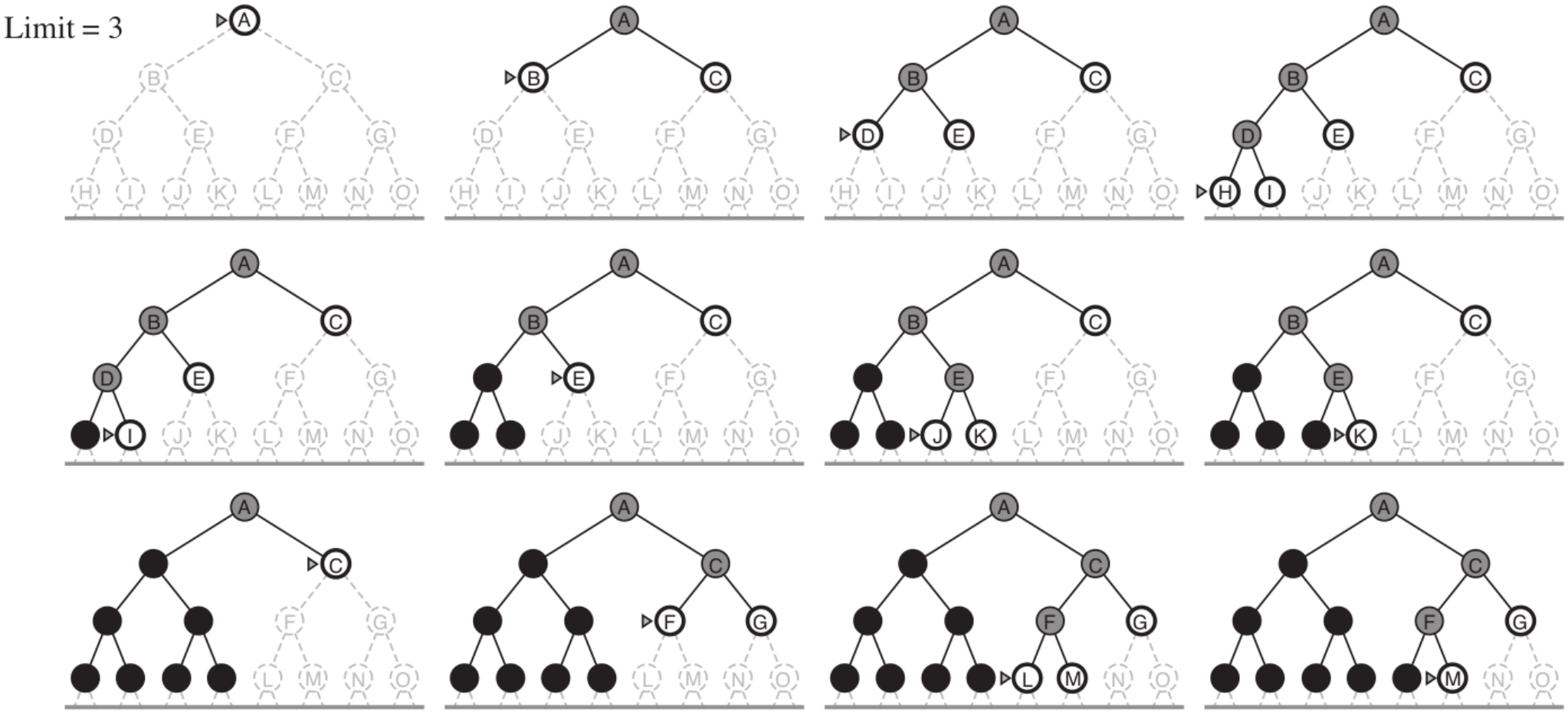
- Combina las ventajas de la búsqueda primero en profundidad y primero en anchura.
- Realiza búsquedas en profundidad sucesivas con un límite de profundidad creciente en cada iteración.
- Consigue la optimalidad de la búsqueda en anchura pero sin su coste espacial.
- Para garantizar que el algoritmo acaba si no hay solución, hay que definir una cota máxima de profundidad.

Búsqueda en profundidad iterativa



2. Búsqueda no informada

Búsqueda en profundidad iterativa



2. Búsqueda no informada

```
función: busqueda_profundidad_iterativa(límite: entero)
prof ← 1
actual ← estado_inicial
mientras !esFinal(actual) && prof < límite
        abiertos.inicializar()
        abiertos.insertar(estado_inicial)
        actual ← abiertos.primero()
        mientras !esFinal(actual) && !abiertos.esVacia()
                abiertos.borrarPrimero()
                cerrados.insertar(actual)
                si profundidad(actual) ≤ prof
                        hijos ← generarSucesores(actual)
                        hijos ← tratarRepetidos(hijos, cerrados, abiertos)
                        abiertos.insertar(hijos)
                fin
                actual ← abiertos.primero()
        fin
        prof ← prof + 1
fin
```

Búsqueda en profundidad iterativa

- Completitud: el algoritmo siempre encontrará la solución.
- Complejidad temporal: como en la búsqueda en anchura. $O(r^p)$
  - → Generar el árbol en cada iteración sólo añade un factor constante a la función de coste.
- Complejidad espacial: como en la búsqueda en profundidad.
- Optimalidad: la solución que se encuentra es óptima, igual que en la búsqueda en anchura.

3. Búsqueda heurística

- Utiliza conocimiento específico del problema para encontrar soluciones de manera más eficiente.
- Función de evaluación de los nodos, f(n) = distancia desde n hasta un objetivo.
- Expande el nodo con f(n) más baja.
  - → Búsqueda primero el mejor.
- El cálculo de f(n) tiene dos efectos:
  - Positivo: ahorro de esfuerzo de búsqueda.
  - Negativo: coste del cálculo de la función.
  - → Para que sea útil: ahorro > coste

3. Búsqueda heurística

- Función heurística, h(n) = coste estimado del camino más barato desde n hasta un objetivo.
  - → Si n es un nodo objetivo: h(n) = 0
- Las funciones heurísticas son la forma más común de transmitir el conocimiento adicional al algoritmo de búsqueda.
- Estrategias:
  - Voraz primero el mejor
  - A*, A* de profundidad iterativa
  - Búsqueda recursiva del primero mejor

Búsqueda voraz primero el mejor

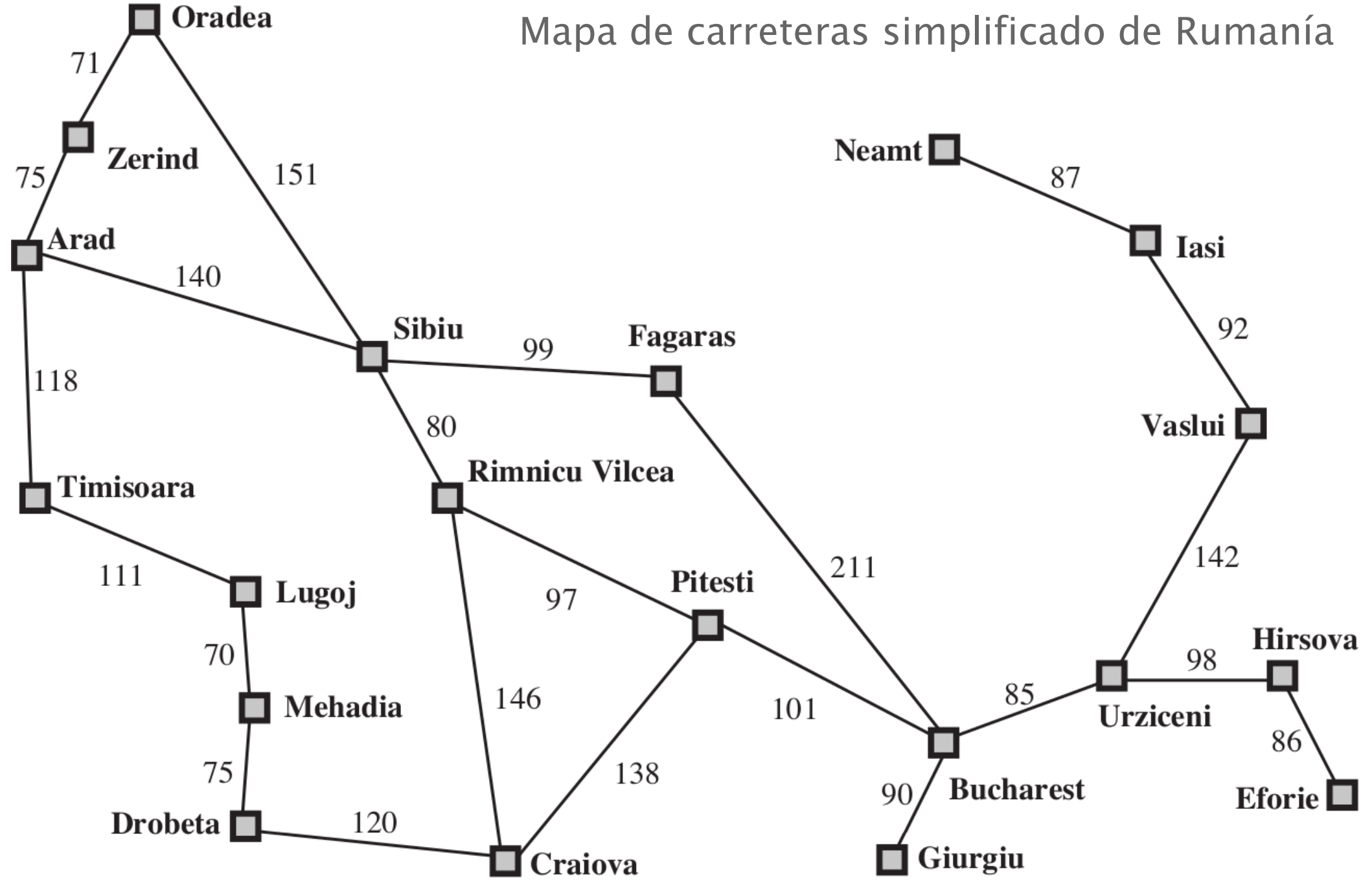
- Evalúa los nodos sólo con la función heurística:

$$f(n) = h(n)$$

- Expande el nodo más cercano al objetivo, con la esperanza de que conduzca rápidamente a la solución → no es óptima y es incompleta.
- La estructura de nodos abiertos es una cola de prioridad donde los nodos están ordenados de menor a mayor según $h(n)$.
- La complejidad temporal y espacial es en el peor de los casos: $O(r^p)$

3. Búsqueda heurística

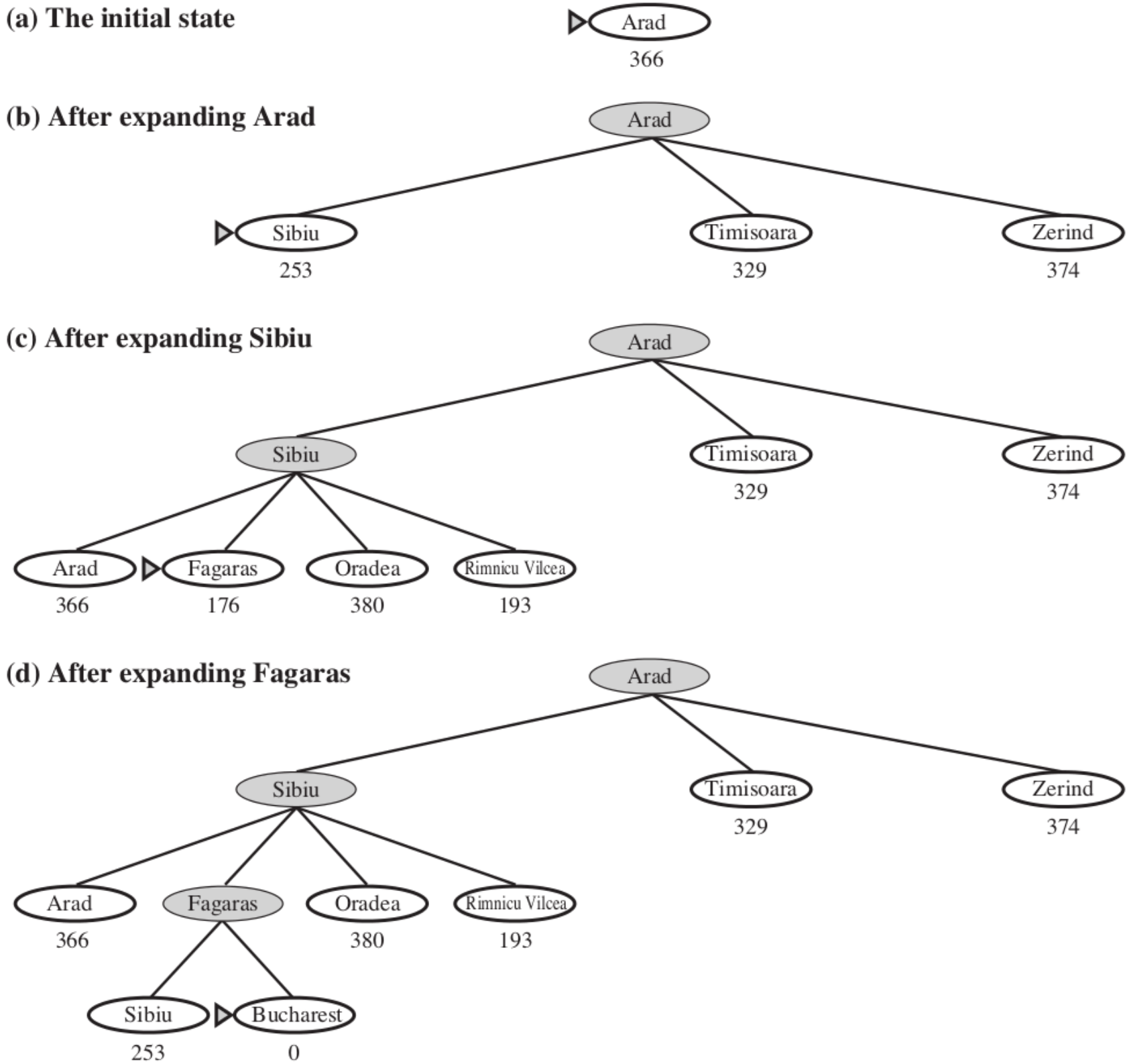
Mapa de carreteras simplificado de Rumanía



Búsqueda voraz primero el mejor

Para encontrar el camino mas corto desde Arad hasta Bucarest se puede utilizar como heurística la distancia en línea recta.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374


(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
253
Timisoara
329
Zerind
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vilcea
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vilcea
193
Sibiu
253
Bucharest
0

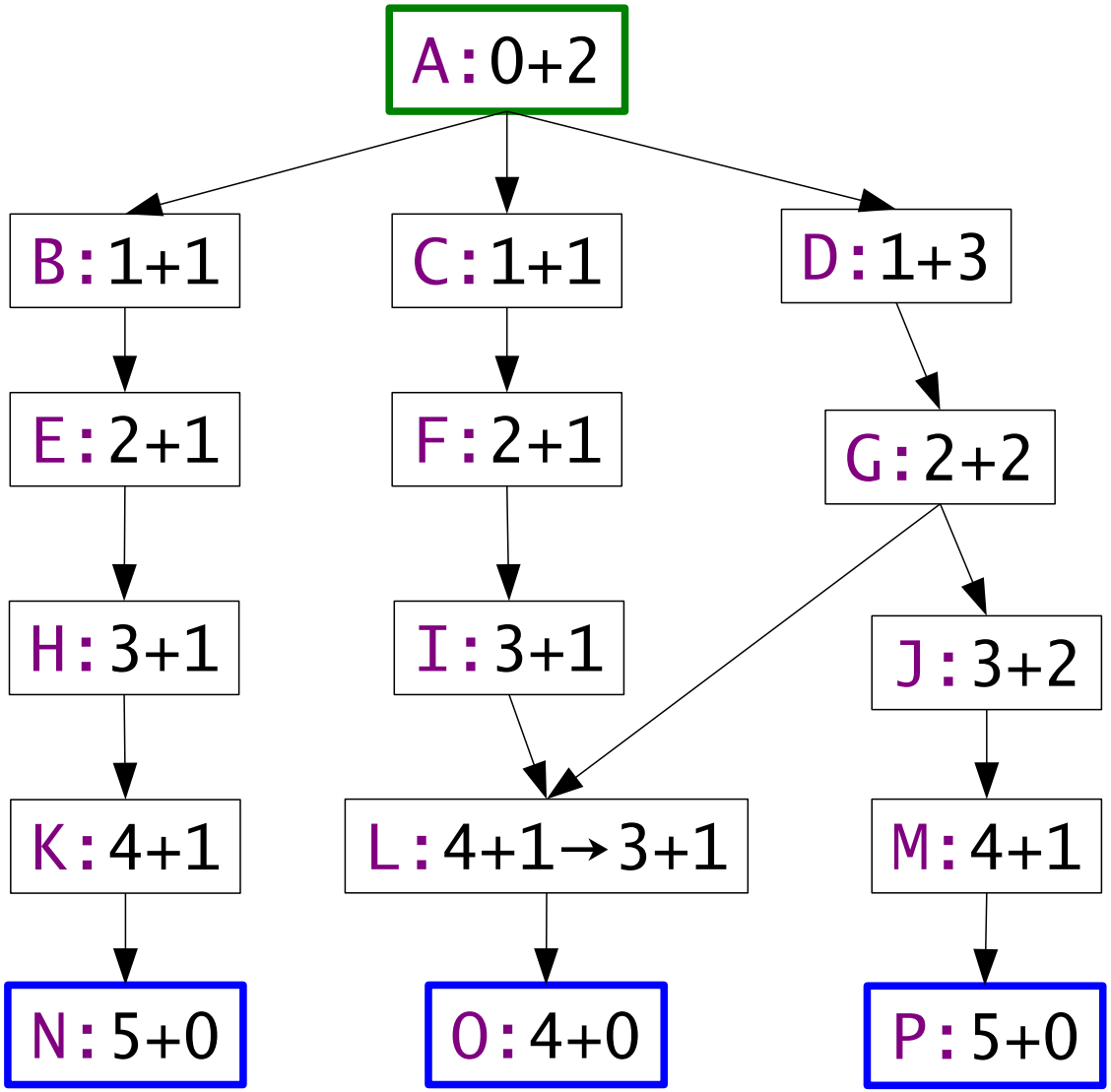
Búsqueda A*

- Es la forma más conocida de búsqueda primero el mejor.
- Minimiza el coste estimado de la solución.
  - $f(n) = g(n) + h(n)$ Coste estimado de la solución a través de n.
  - $g(n)$ = coste real hasta el nodo n
  - $h(n)$ = coste estimado hasta el objetivo
- En caso de empate se escoge el nodo con menor h.

Búsqueda A* – **Tratamiento de nodos repetidos**

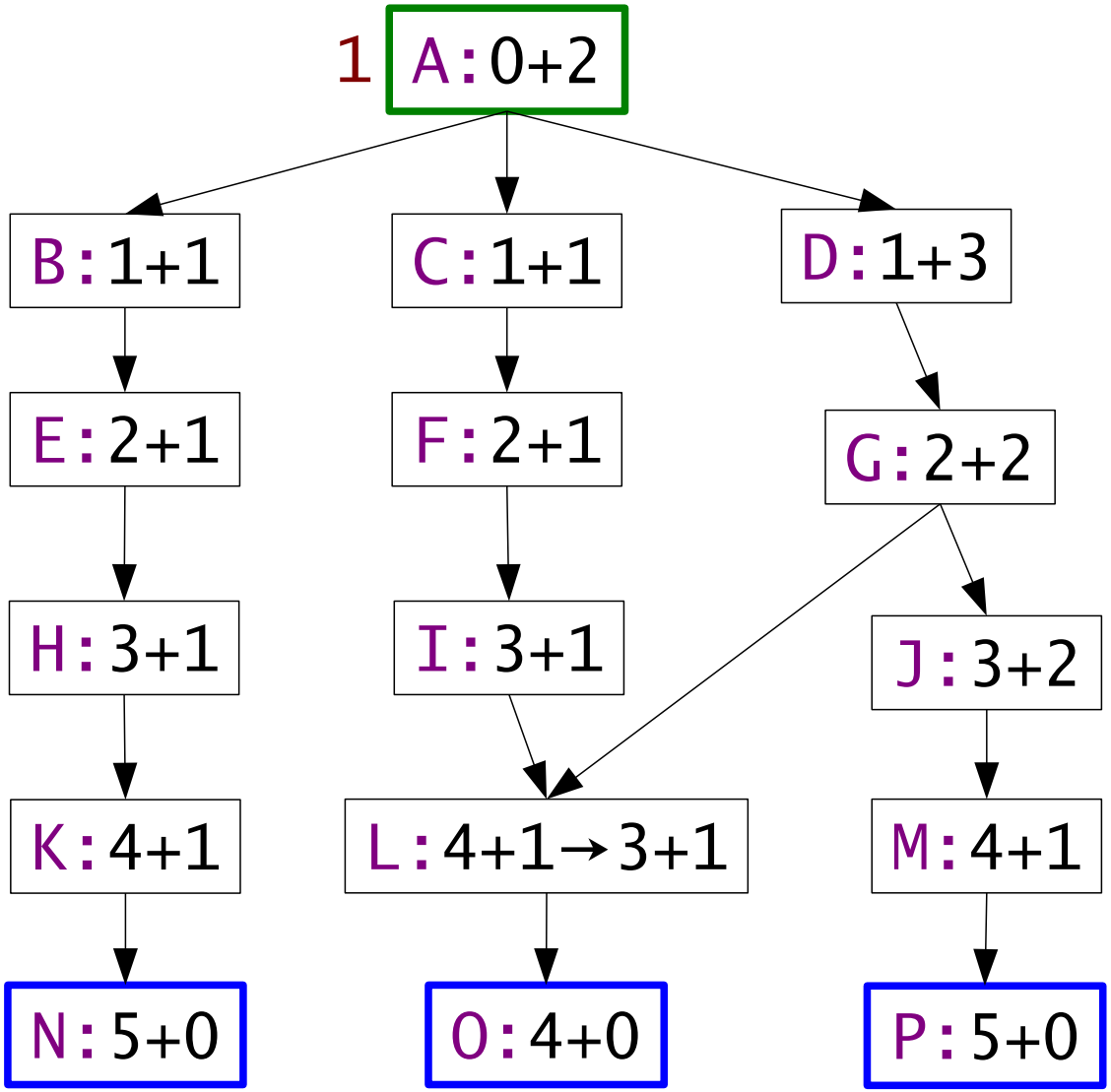
- Si un nodo repetido está en la est. de abiertos:
  - Si su coste es menor, sustituimos el coste por el menor. Esto podrá variar su posición en la estructura.
  - Si su coste es igual o mayor se olvida el nodo.
- Si un nodo repetido está en la est. de cerrados:
  - Si su coste es menor
    → el nodo es reabierto con el coste menor.
  - Si su coste es mayor o igual
    → el nodo es olvidado.

Búsqueda A* – Tratamiento de nodos repetidos



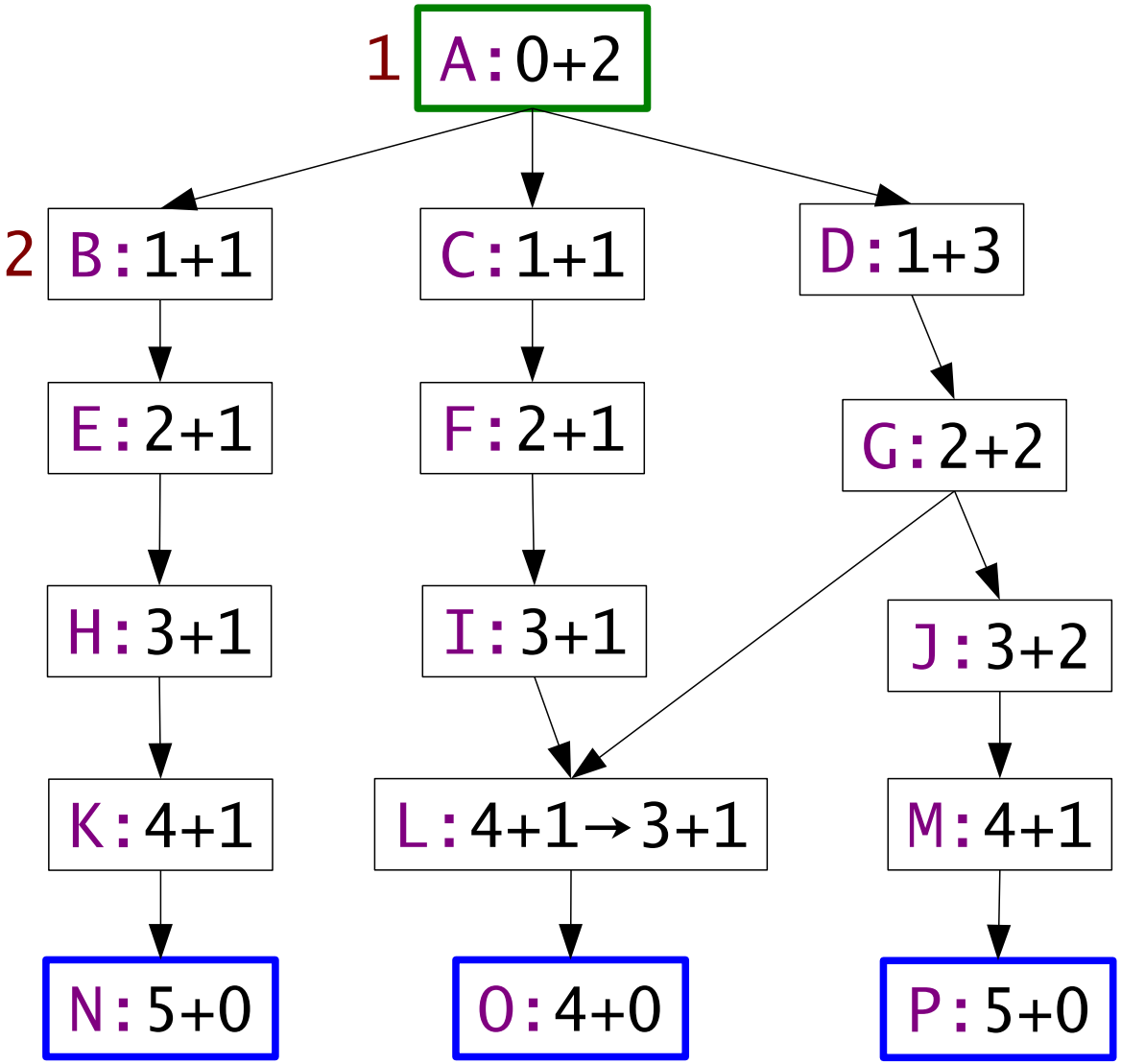
Abiertos	Cerrados
A:0+2	

Búsqueda A* – Tratamiento de nodos repetidos



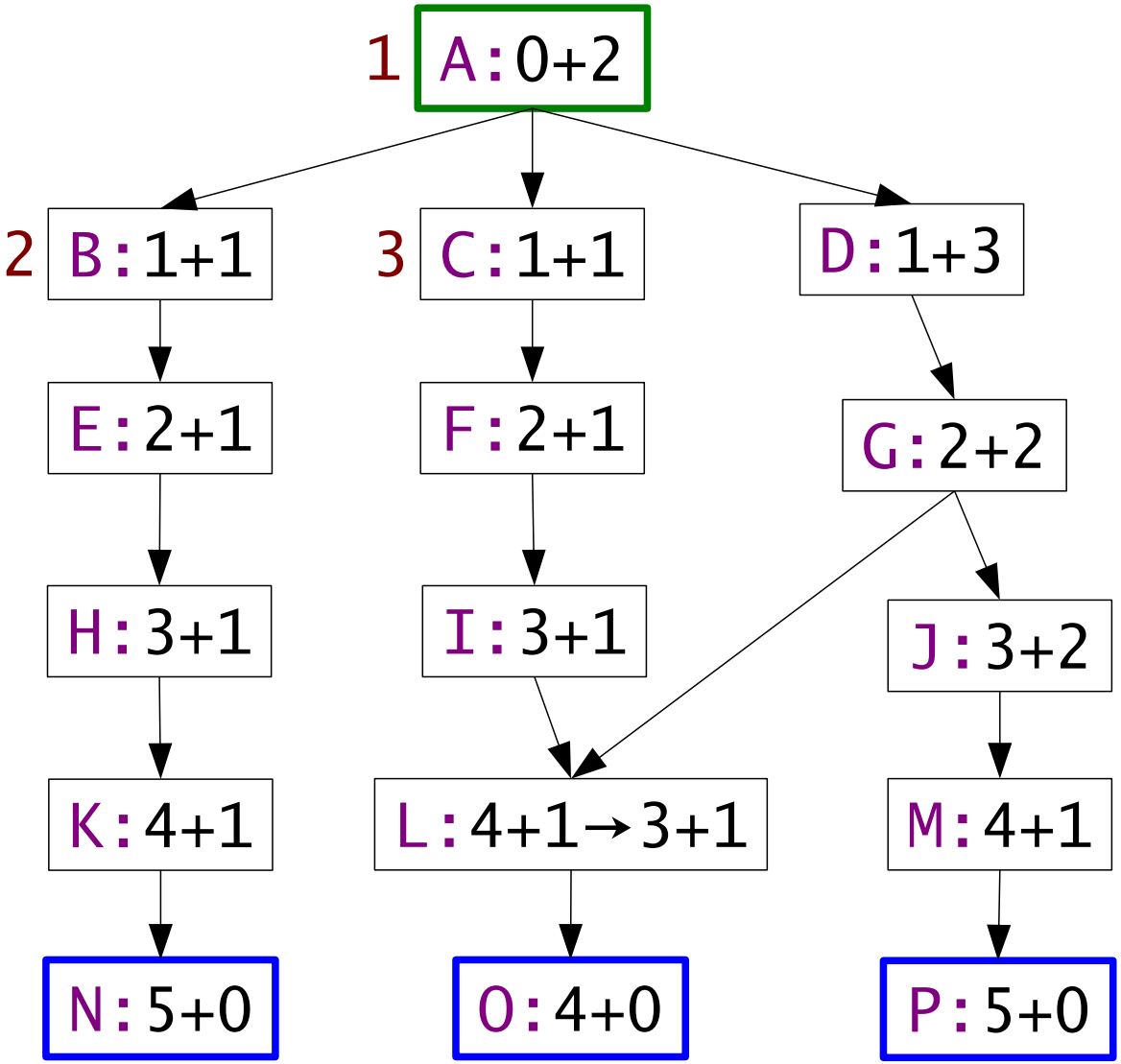
Abiertos	Cerrados
B:1+1	A:0+2
C:1+1	
D:1+3	

Búsqueda A* – Tratamiento de nodos repetidos



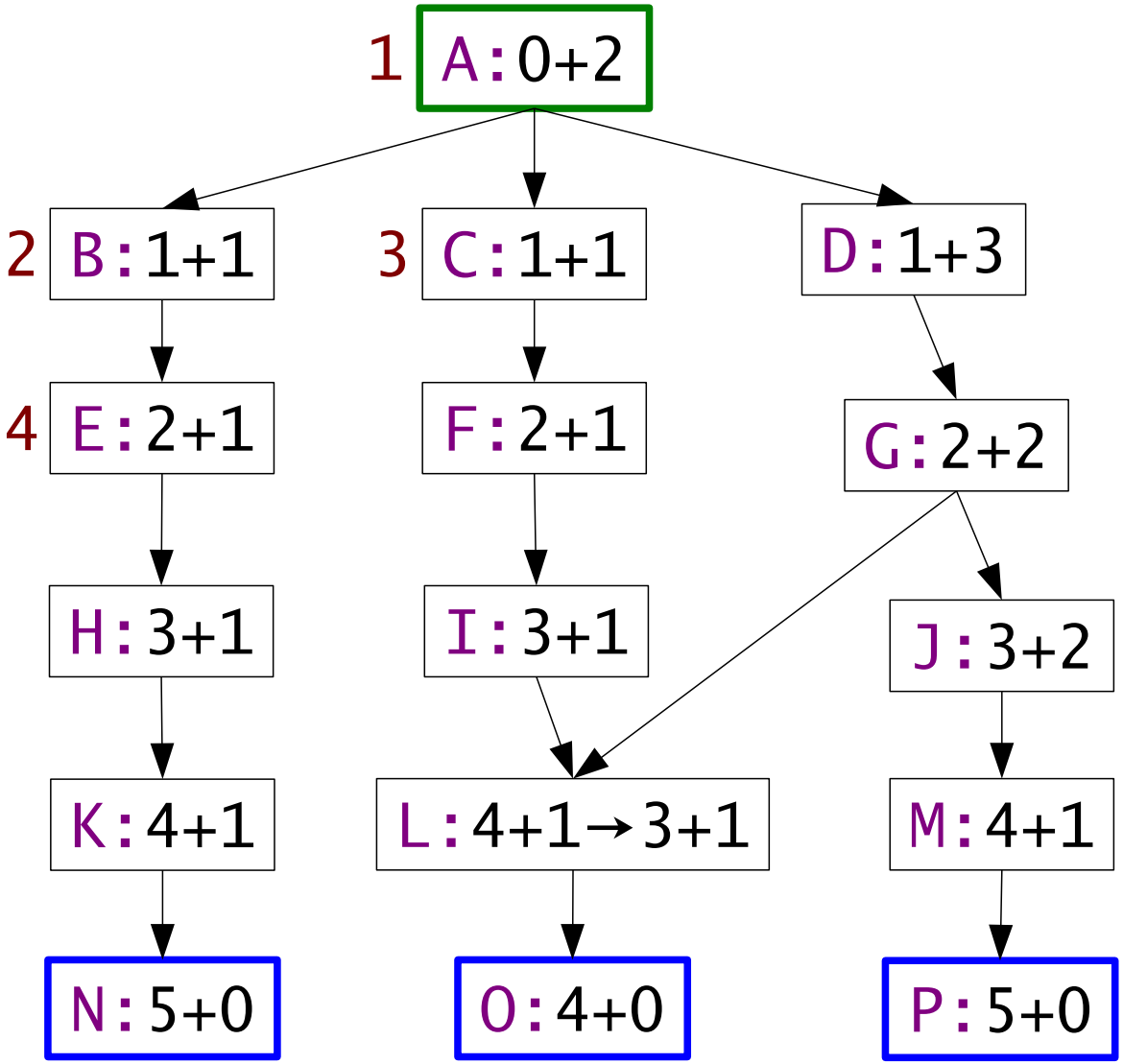
Abiertos	Cerrados
C:1+1	A:0+2
E:2+1	B:1+1
D:1+3	

Búsqueda A* – Tratamiento de nodos repetidos



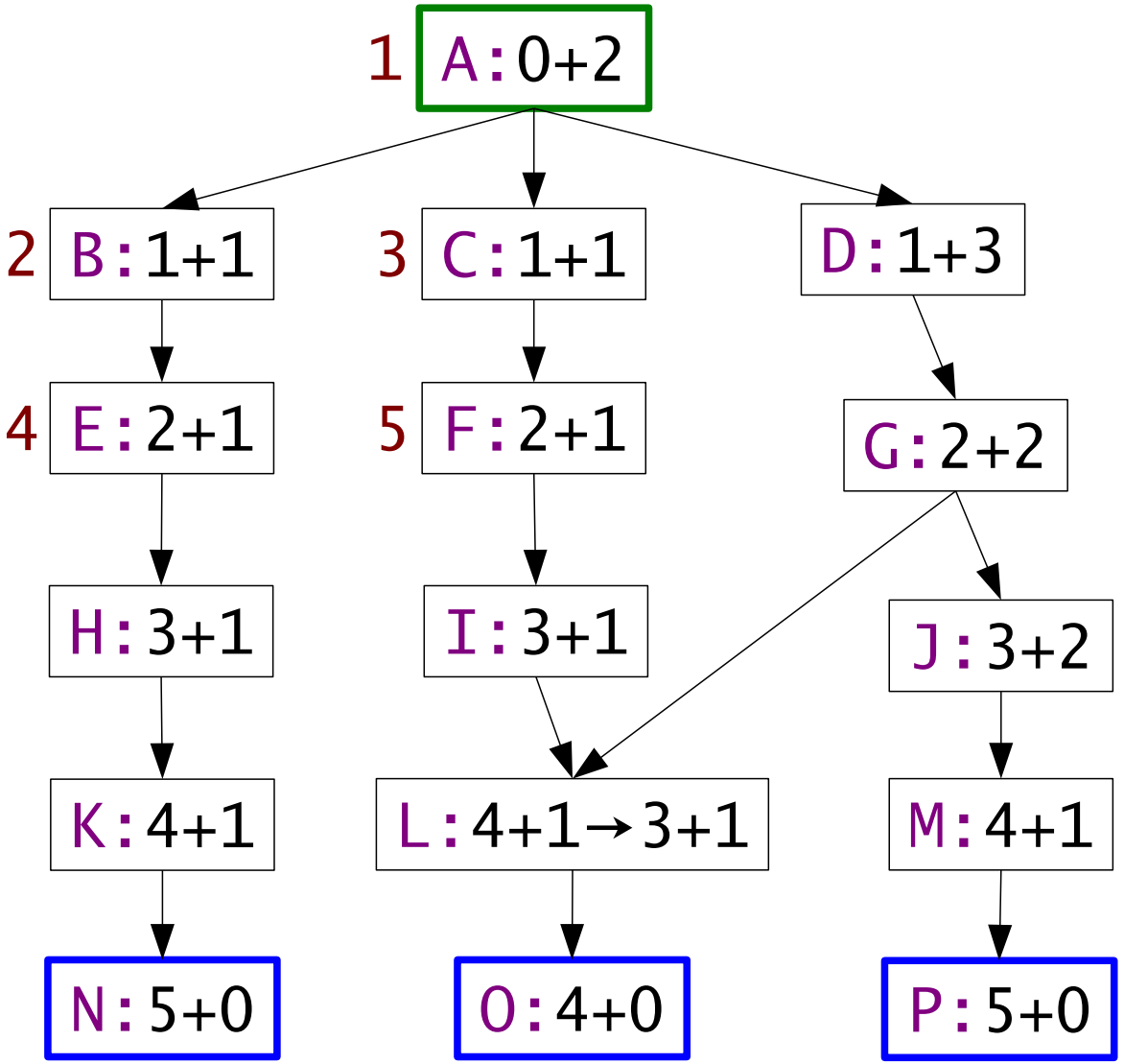
Abiertos	Cerrados
E:2+1	A:0+2
F:2+1	B:1+1
D:1+3	C:1+1

Búsqueda A* – Tratamiento de nodos repetidos



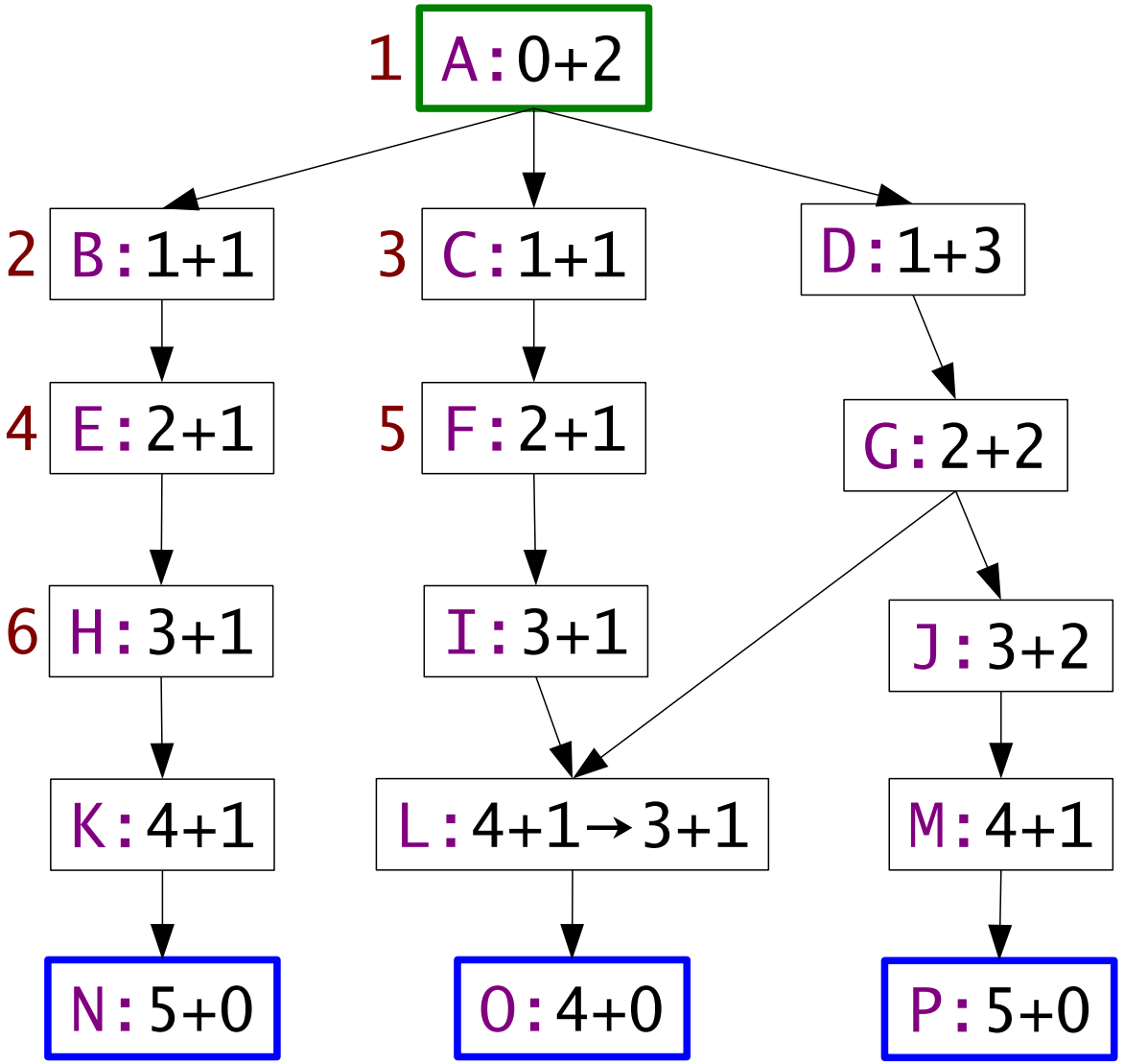
Abiertos	Cerrados
F:2+1	A:0+2
H:3+1	B:1+1
D:1+3	C:1+1
	E:2+1

Búsqueda A* – Tratamiento de nodos repetidos



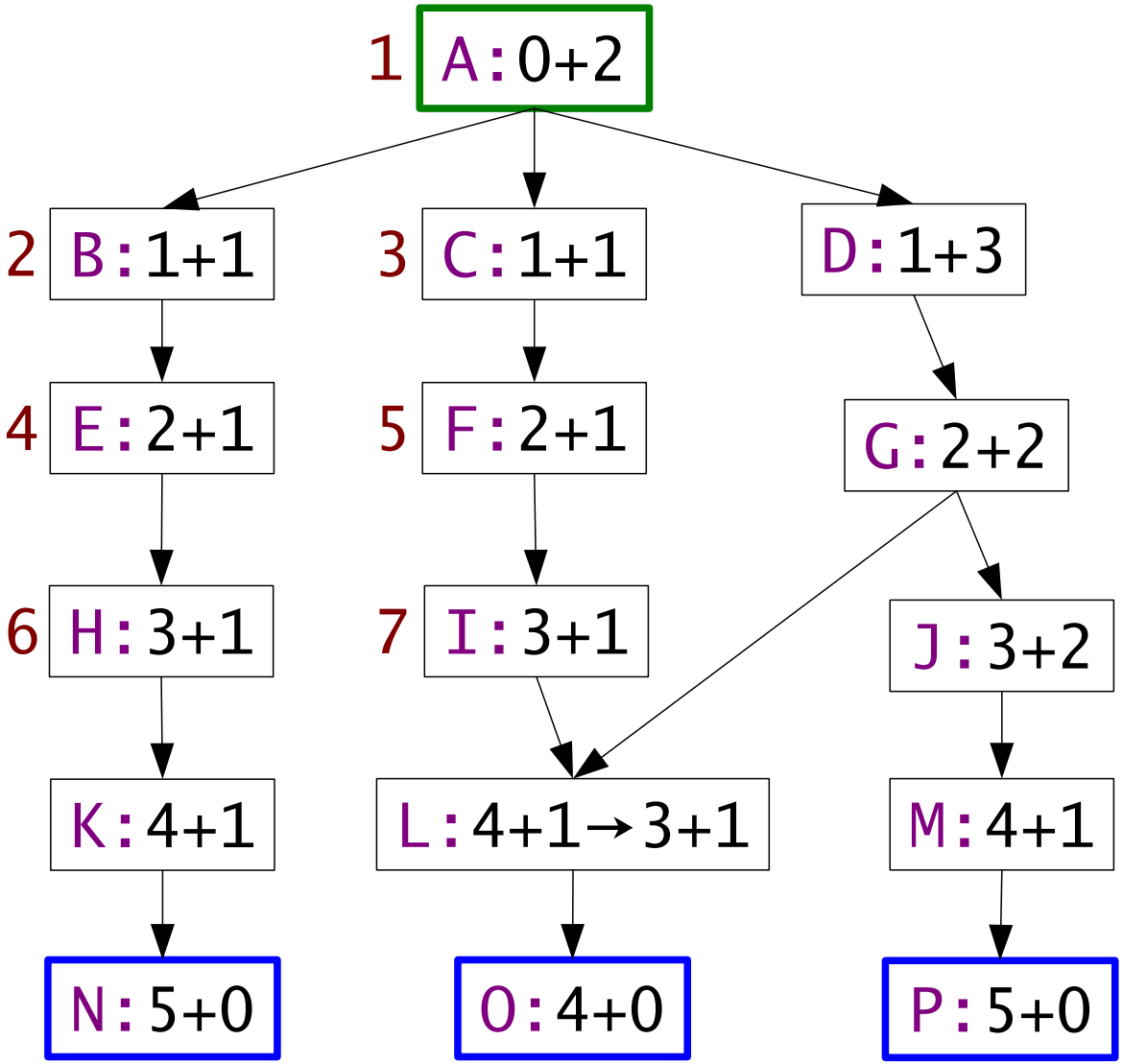
Abiertos	Cerrados
H:3+1	A:0+2
I:3+1	B:1+1
D:1+3	C:1+1
	E:2+1
	F:2+1

Búsqueda A* – Tratamiento de nodos repetidos



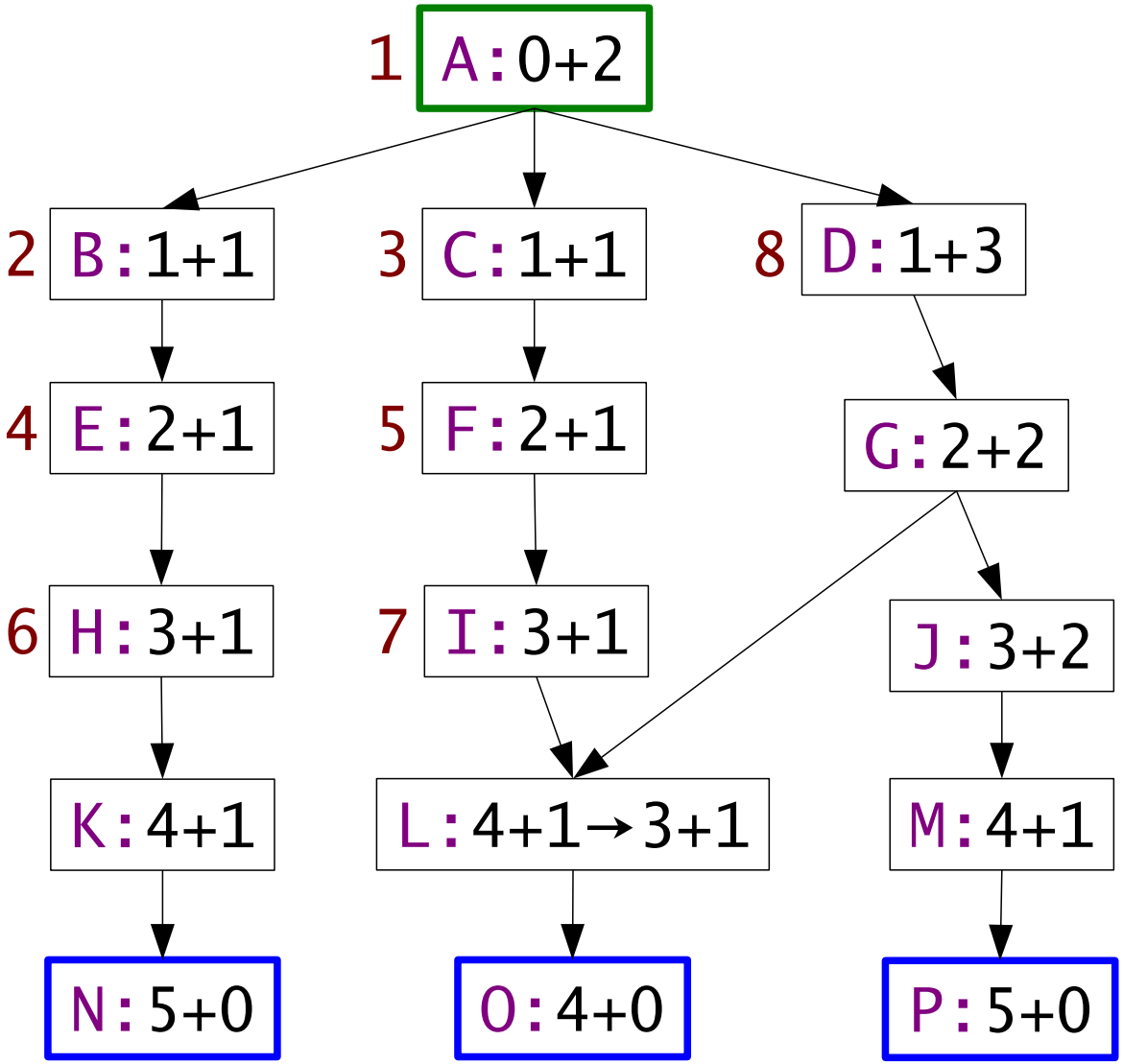
Abiertos	Cerrados
I:3+1	A:0+2
D:1+3	B:1+1
K:4+1	C:1+1
	E:2+1
	F:2+1
	H:3+1

Búsqueda A* – Tratamiento de nodos repetidos



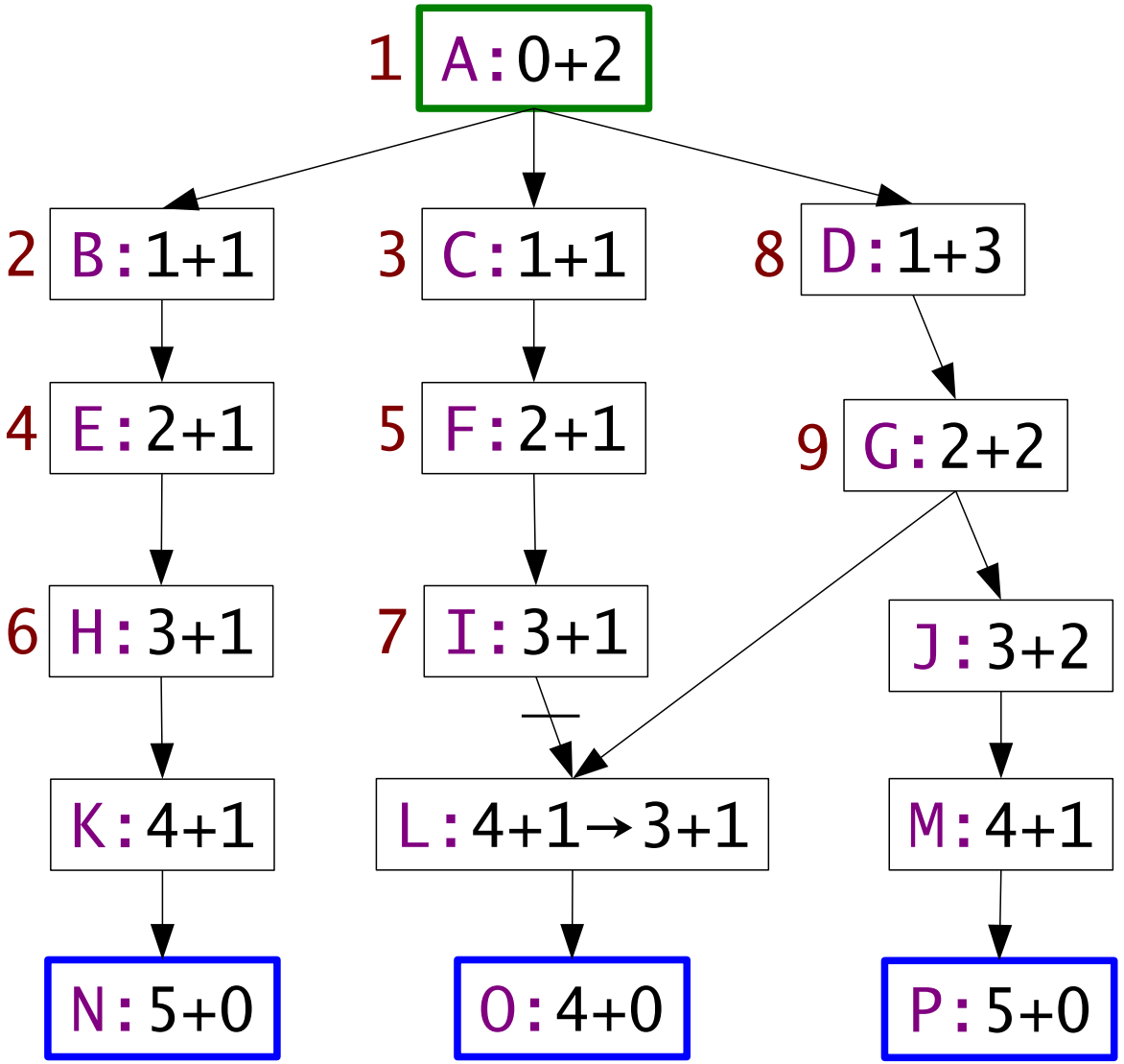
Abiertos	Cerrados
D:1+3	A:0+2
K:4+1	B:1+1
L:4+1	C:1+1
	E:2+1
	F:2+1
	H:3+1
	I:3+1

Búsqueda A* – Tratamiento de nodos repetidos



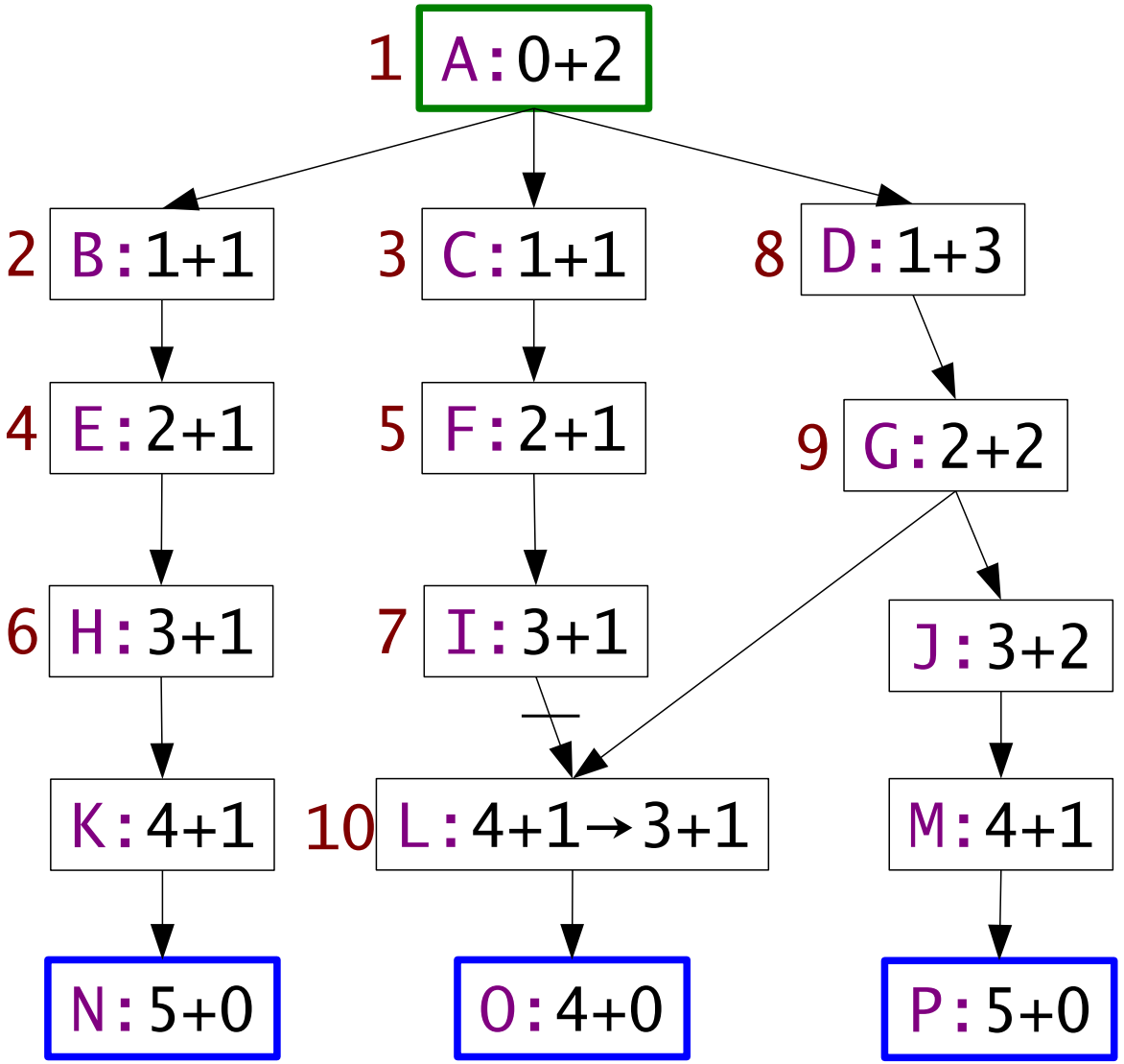
Abiertos	Cerrados
G:2+2	A:0+2
K:4+1	B:1+1
L:4+1	C:1+1
	E:2+1
	F:2+1
	H:3+1
	I:3+1
	D:1+3

Búsqueda A* – Tratamiento de nodos repetidos



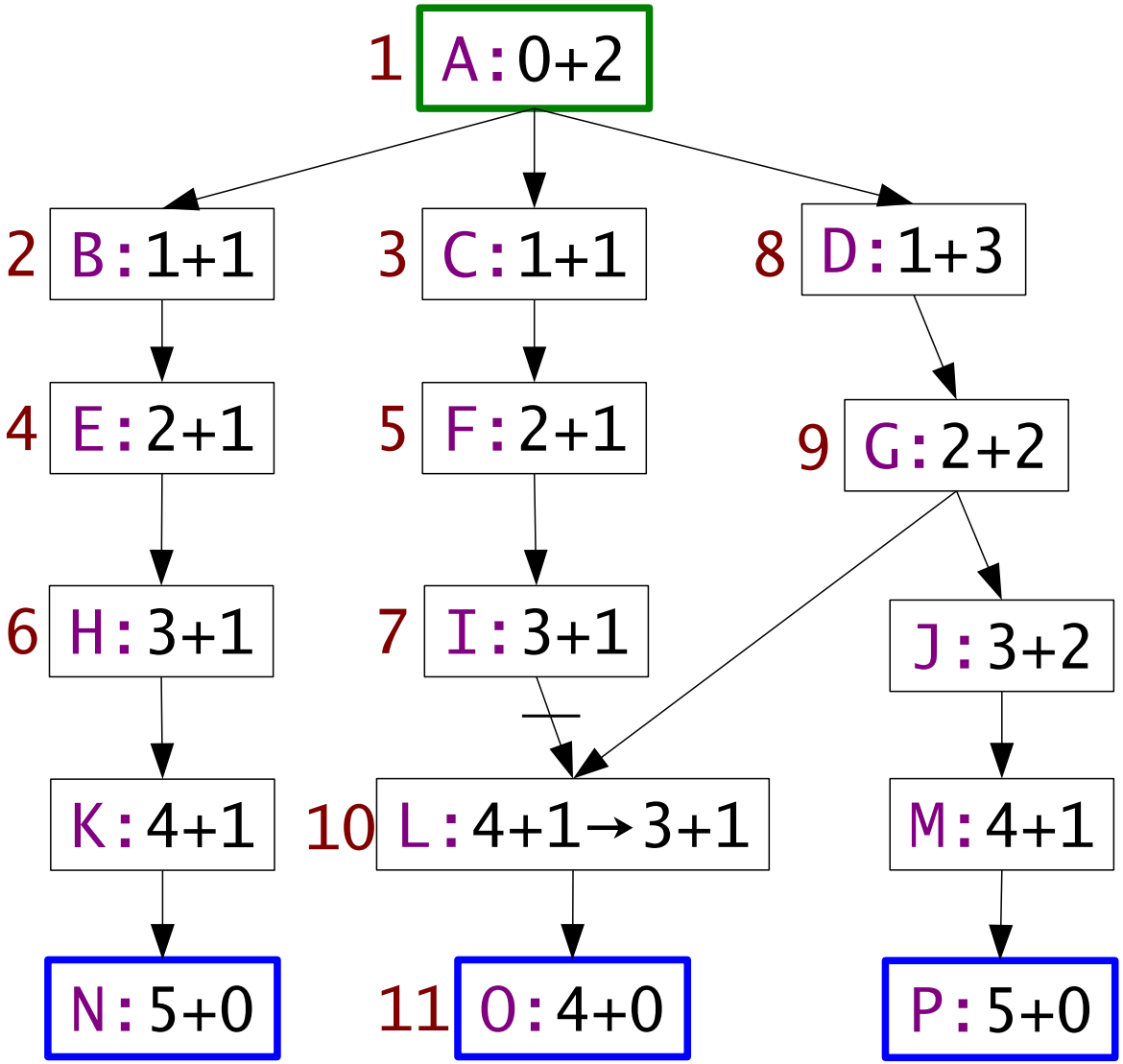
Abiertos	Cerrados
L:3+1	A:0+2
K:4+1	B:1+1
J:3+2	C:1+1
	E:2+1
	F:2+1
	H:3+1
	I:3+1
	D:1+3
	G:2+2

Búsqueda A* – Tratamiento de nodos repetidos



Abiertos	Cerrados
O:4+0	A:0+2
K:4+1	B:1+1
J:3+2	C:1+1
	E:2+1
	F:2+1
	H:3+1
	I:3+1
	D:1+3
	G:2+2
	L:3+1

Búsqueda A* – Tratamiento de nodos repetidos



Abiertos	Cerrados
K:4+1	A:0+2
J:3+2	B:1+1
	C:1+1
	E:2+1
	F:2+1
	H:3+1
	I:3+1
	D:1+3
	G:2+2
	L:3+1
	O:4+0

Búsqueda A* – **Admisibilidad**

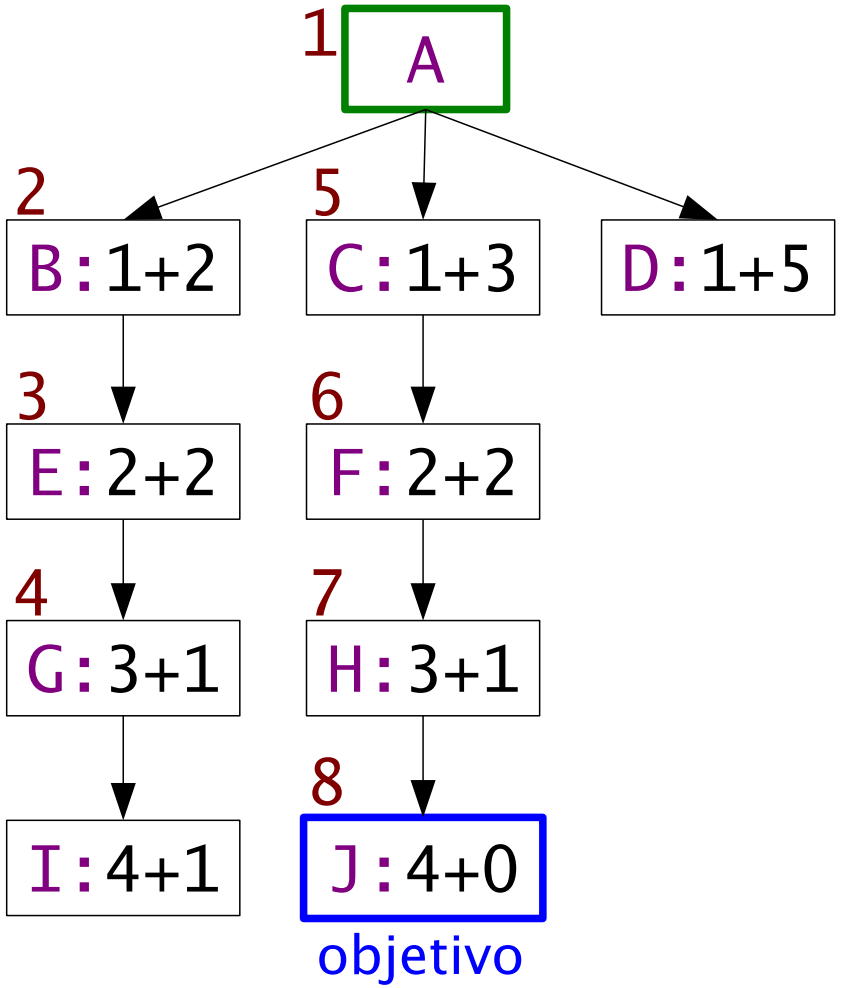
- A* es óptima si h(n) es una heurística admisible:
  - → $\forall n: 0 \leq h(n) \leq h^*(n)$
    Por tanto, h debe ser un estimador optimista, nunca debe sobrestimar el coste verdadero h^*.
  - Como g(n) es el coste exacto hasta n,
    - → $f(n) \leq$ coste verdadero.
- Cuanto más se aproxime h a h^* mayor será la tendencia a explorar en profundidad.
- Si $h = h^* \rightarrow A^*$ converge directamente hacia el objetivo.

Búsqueda A* – Admisibilidad

h subestima h*

h(B)=2, no es real

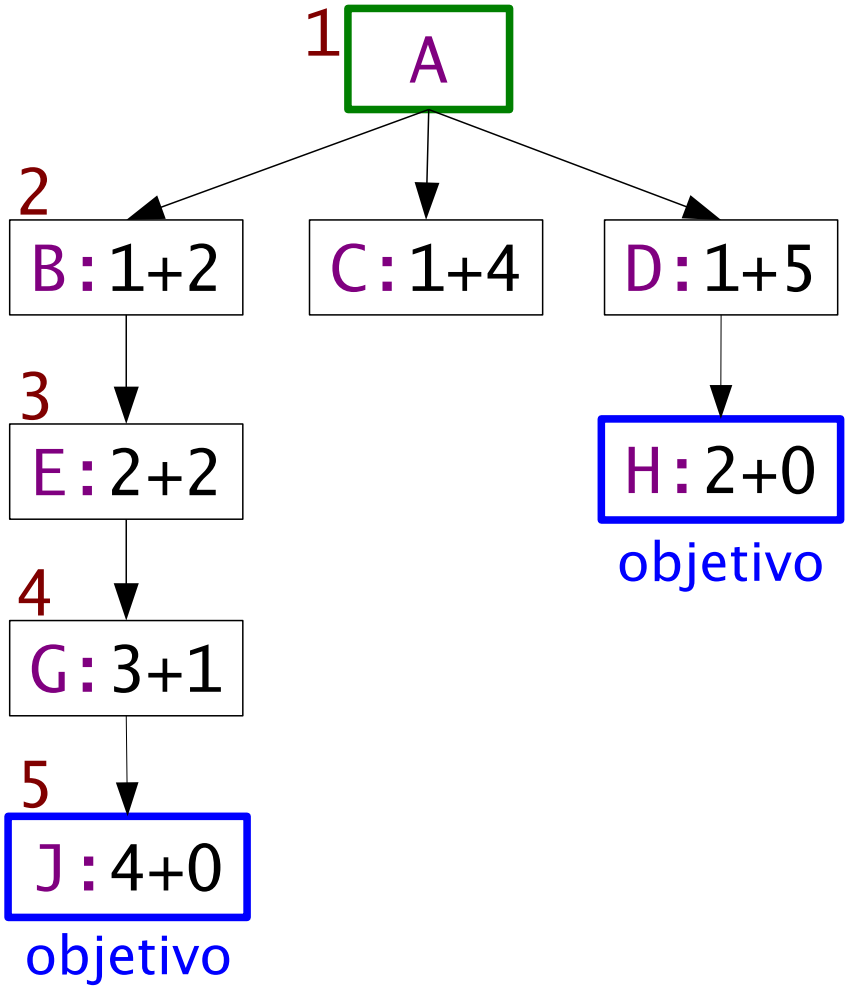
Se pierde tiempo pero se consigue llegar al objetivo



Búsqueda A* – Admisibilidad

h sobrestima h*

Partiendo de D existe un camino más corto, pero no se alcanza.



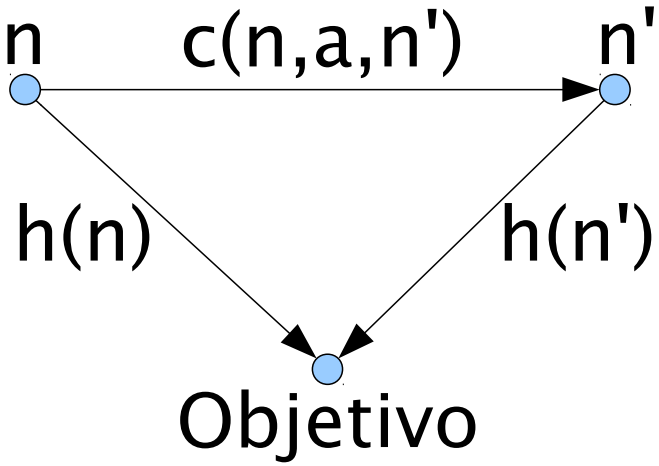
Búsqueda A* – **Consistencia (monotonía)**

Sea n un nodo.
Sea n' un sucesor de n generado por una acción a.
Sea c(n,a,n') el coste de aplicar la acción a.

Una heurística h(n) es consistente si se cumple:

$$h(n) \leq c(n,a,n') + h(n')$$ (desigualdad triangular)

- La distancia en línea recta es una heurística consistente para el problema del mapa de carreteras.



Búsqueda A* – Consistencia (monotonía)

- Toda heurística consistente es también admisible.
- Con una heurística consistente se llega a los nodos por el camino mínimo, por tanto, ya no se podrán reexpandir. → No es necesario tratar los nodos duplicados cerrados.
- Los valores de f(n) a lo largo de cualquier camino no disminuyen:

$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n')$$
$$f(n') \geq g(n) + h(n) = f(n)$$

Búsqueda A* – **Algoritmos más informados**

- Dado un problema, existen tantos A^* para resolverlo como estimadores se puedan definir.
- Sean h_1 y h_2 admisibles, si se cumple:

  $\forall n \neq final: 0 \leq h_2(n) < h_1(n) \leq h^*(n)$

  entonces A_1^* es más informado que A_2^*.
- Si el nodo n es expandido por A_1^*

  $\Rightarrow$ n es expandido por A_2^* (pero no al revés)
- A_1^* expande menos nodos que A_2^*

  → no expandir un nodo (subárbol) se llama poda.

Búsqueda A* – Algoritmos más informados

- ¿Siempre convienen algoritmos más informados?
- Compromiso entre:
  - Tiempo de cálculo de h:
    - $h_1(n)$ requerirá más tiempo que $h_2(n)$.
  - Número de reexpansiones:
    - A_1^* puede que reexpanda más que A_2^*.
    - Pero si A_1^* es consistente no lo hará.

Búsqueda A* – Algoritmos más informados

Las siguientes heurísticas del 8 puzzle son admisibles:

- $h_0(n) = 0$
  - Equivale a la búsqueda en anchura.
- $h_1(n) =$ nº de piezas mal colocadas
  - A_1^* está más informado que A_0^*.
- $h_2(n) = \Sigma_{i\in[1,8]} d_i$
  - $d_i \equiv$ distancia de la pieza i hasta su pos. final.
  - A_2^* está más informado que A_1^*.

Búsqueda A* – **Memoria acotada**

- A* resuelve problemas en los que hay que encontrar la mejor solución.
- Su coste en espacio y tiempo en el caso medio es mejor que los algoritmos no informados.
- Hay problemas en los que el tamaño del espacio de búsqueda no permite la aplicación de A*.
- Para que el nº de nodos a almacenar no crezca exponencialmente:

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

4. Búsqueda entre adversarios

- Permite decidir la mejor jugada en cada momento para cierto tipo de juegos.
- Los juegos tienen diferentes características:
  - Nº de jugadores, información conocida por todos los jugadores, cooperación/competición, azar, recursos limitados, ...
- Se analizarán los juegos con:
  - 2 jugadores (MIN y MAX).
  - Movimientos alternos.
  - Información perfecta.
  - Por ejemplo: ajedrez, damas, go, ...

Representación del juego

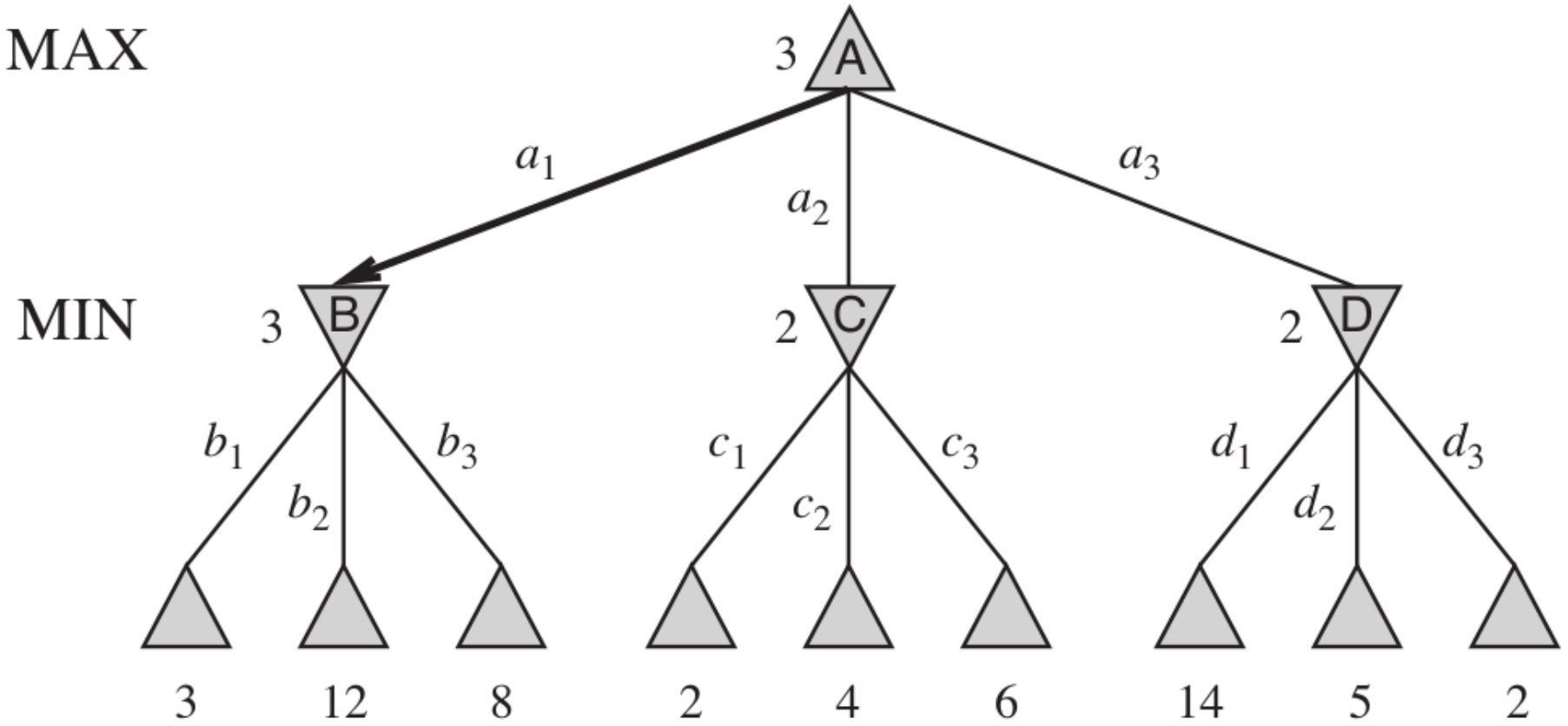
- Se puede definir como un problema de espacio de estados:
  - Estado = elementos del juego
  - Estados finales = estados ganadores
  - Acciones/operadores = reglas del juego
- La accesibilidad de los estados depende de las acciones elegidas por el contrario.
- No hay optimalidad. Todas las soluciones son iguales sin importar la longitud del camino.

4. Búsqueda entre adversarios

- La aproximación trivial es generar todo el árbol de jugadas.
- Las jugadas terminales se etiquetan con el valor +1 si gana MAX, -1 si gana MIN.
- El objetivo es encontrar un conjunto de movimientos accesible para que gane MAX.
- Los valores se propagan hasta la raíz para elegir una rama de una hoja ganadora accesible.
- Una búsqueda en profundidad minimiza el espacio.
- En muchos juegos esta búsqueda es impracticable. Por ejemplo en el ajedrez $O(2^{35})$, go $O(2^{300})$.

4. Búsqueda entre adversarios

- Valor-minimax(n) =
  - utilidad(n) si n es un estado terminal
  - $\max_{s \in \text{sucesores}(n)}$ valor-minimax(s) si n es MAX
  - $\min_{s \in \text{sucesores}(n)}$ valor-minimax(s) si n es MIN



4. Búsqueda entre adversarios

- Aproximación heurística:
  - Función que indique lo cerca que se está de una jugada ganadora (o perdedora).
  - Intervendrá información del dominio.
  - No representa coste ni distancia en pasos.
  - Las jugadas ganadoras se evalúan a $+\infty$ y las perdedoras a $-\infty$.
- El algoritmo busca con profundidad limitada y decide la siguiente jugada a partir del nodo raíz.
- Cada nueva jugada implicará repetir la búsqueda.
- Cuanta más profundidad, mejor juego.

4. Búsqueda entre adversarios

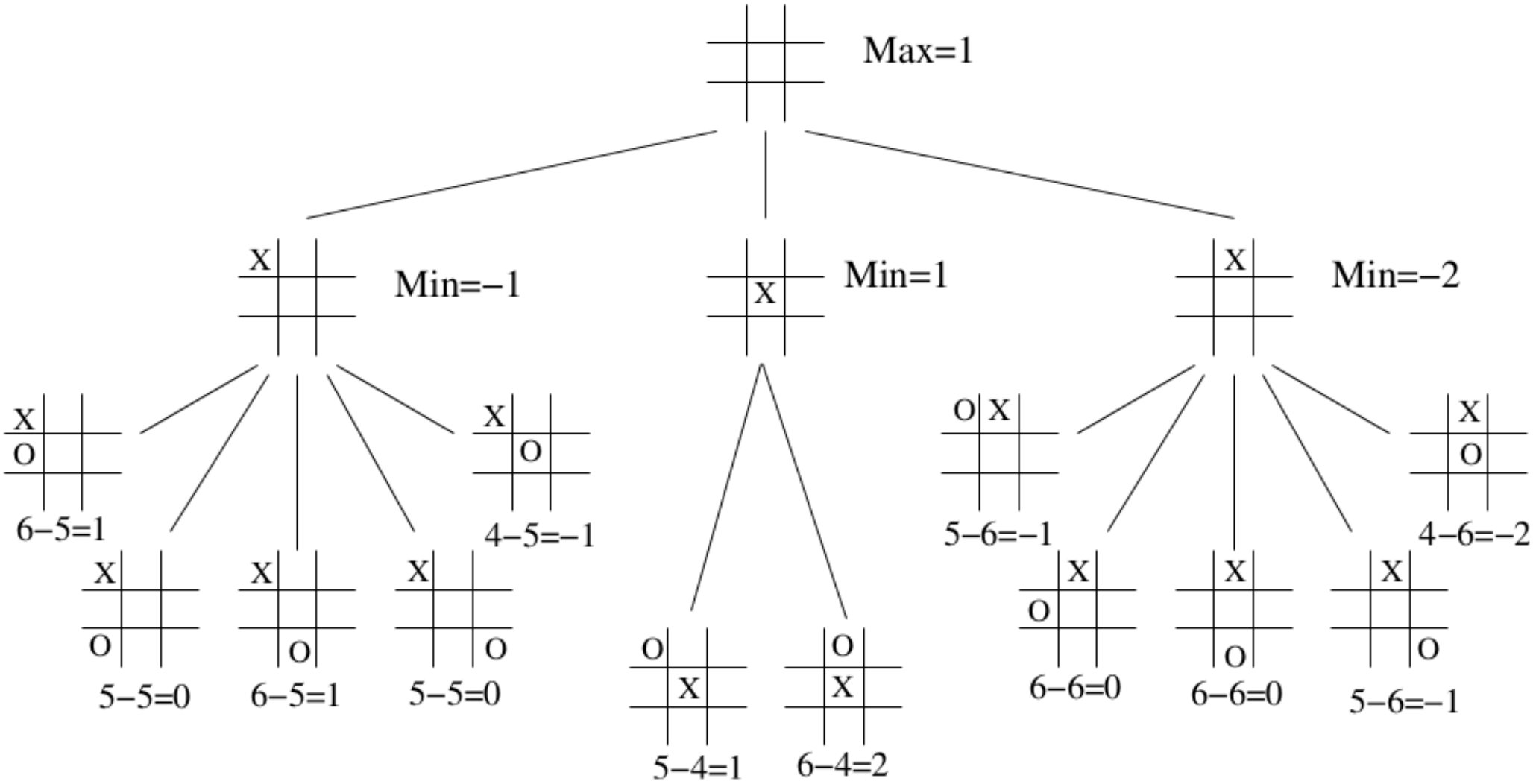
```
función: minimax(estado) devuelve: una acción
        v ← maxValor(estado)
        devolver la acción de sucesores(estado) con valor v
función: maxValor(estado) devuelve: valor de utilidad
        si esTerminal(estado) devolver utilidad(estado)
        v ← -∞
        para s ∈ sucesores(estado) hacer
                v ← max(v, minValor(s))
        devolver v
función: minValor(estado) devuelve: valor de utilidad
        si esTerminal(estado) devolver utilidad(estado)
        v ← +∞
        para s ∈ sucesores(estado) hacer
                v ← min(v, maxValor(s))
        devolver v
```

Ejemplo: 3 en raya

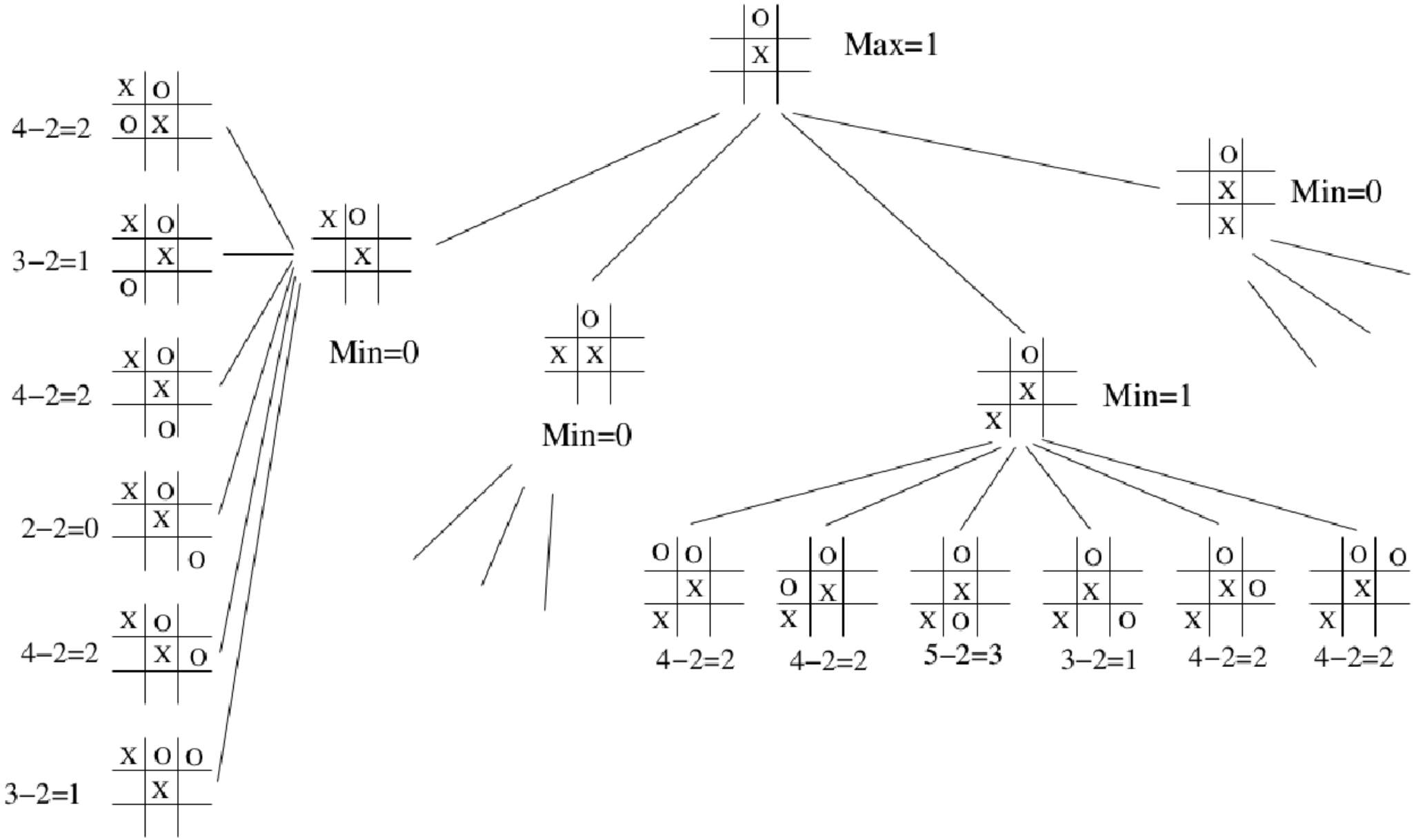
- e = nº de filas, columnas y diagonales completas disponibles para MAX – las disponibles para MIN.
- MAX juega con X y desea maximizar e.
- MIN juega con O y desea minimizar e.
- Se pueden controlar las simetrías para reducir el tamaño del árbol.
- Se establece la profundidad de parada 2.

4. Búsqueda entre adversarios

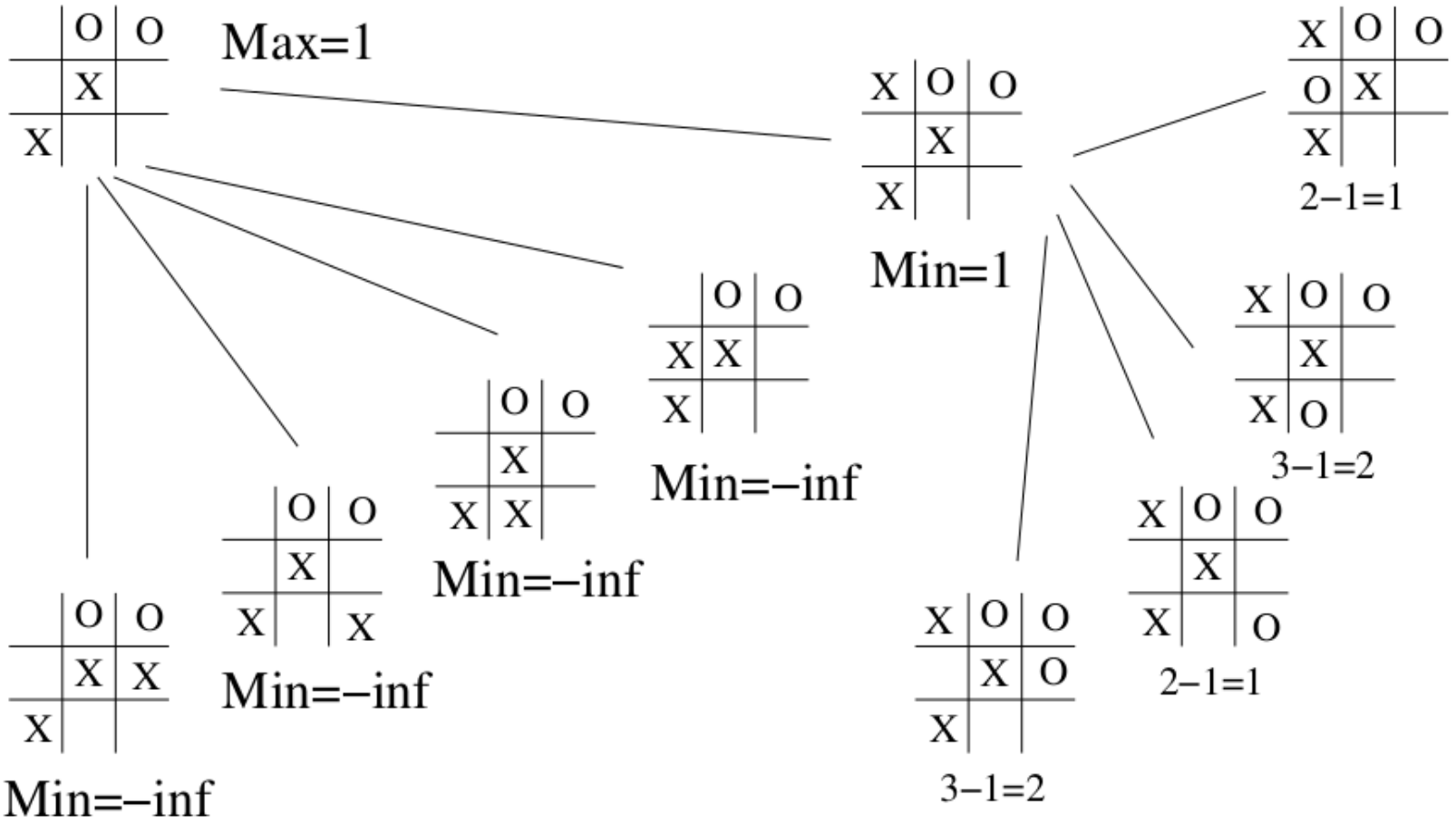
Ejemplo: 3 en raya



4. Búsqueda entre adversarios



Ejemplo: 3 en raya



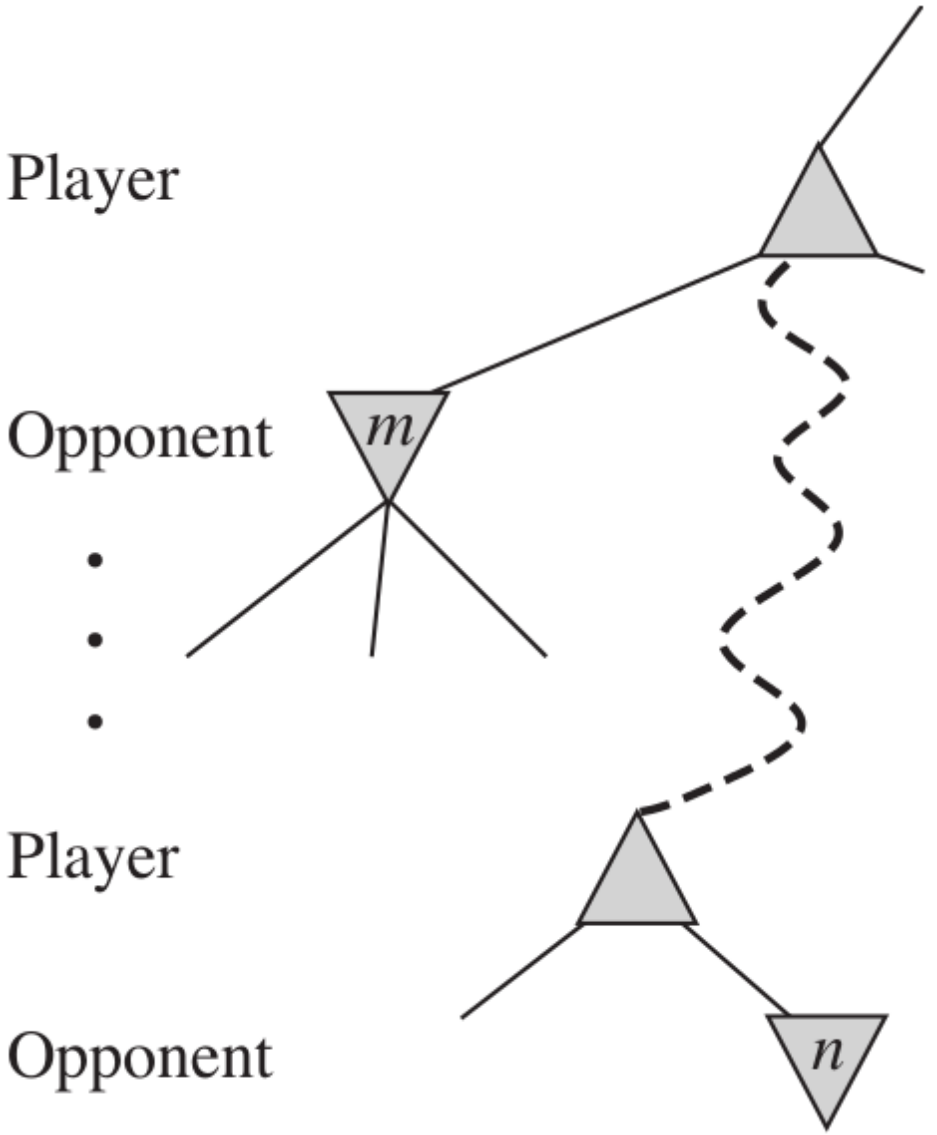
Minimax con poda αβ

- El problema de Minimax es que el nº de nodos a explorar es exponencial con el número de movimientos.
- Es posible calcular la decisión minimax correcta sin mirar todos los nodos en el árbol de juegos.
- α = valor de la mejor opción para MAX a lo largo del camino.
- β = valor de la mejor opción para MIN a lo largo del camino.

Minimax con poda αβ

Caso general:

Si m es mejor que n para el jugador, nunca hay que ir a n en el juego.



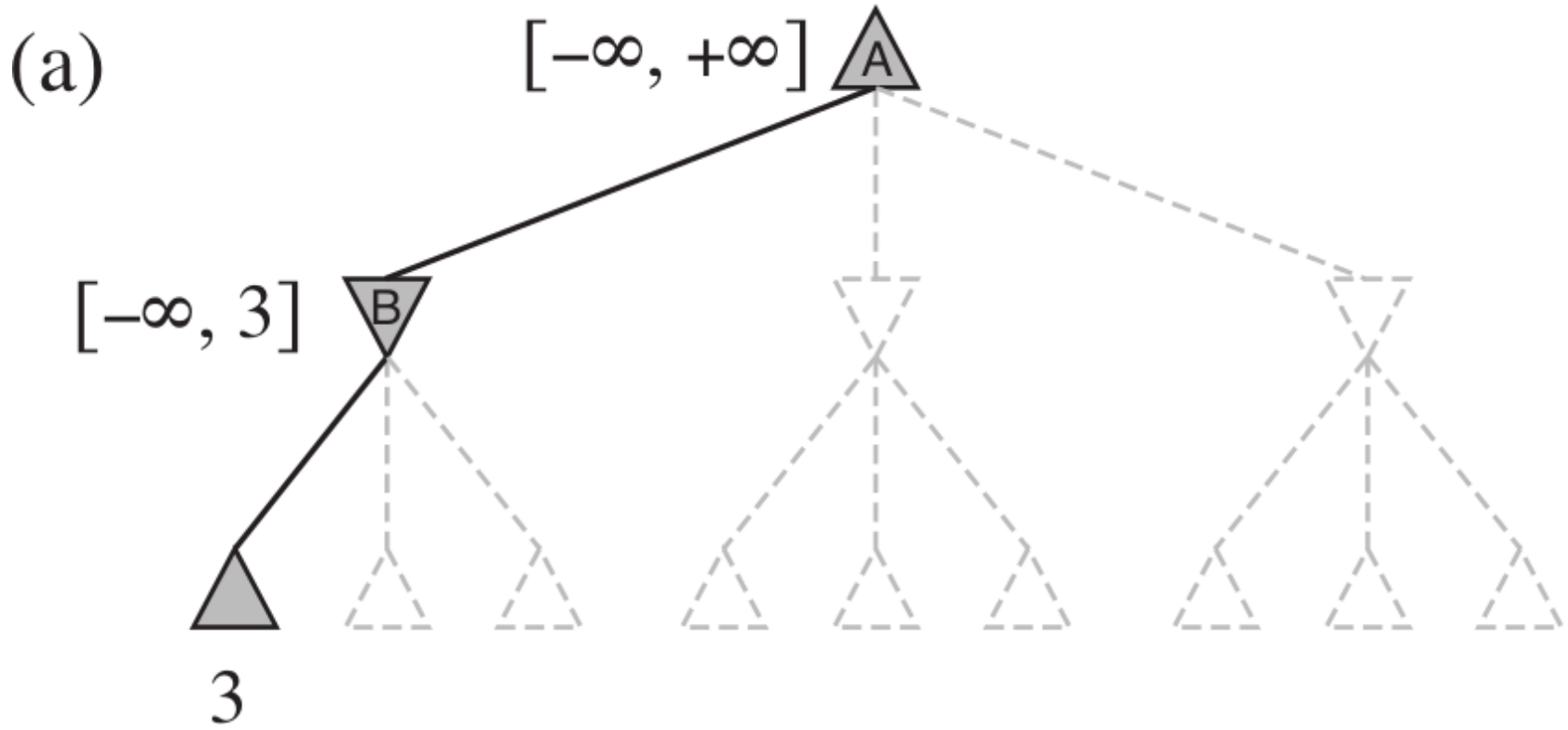
4. Búsqueda entre adversarios

```
función: minimaxAlfaBeta(estado) devuelve: una acción
        v ← maxValor(estado, -∞, +∞)
        devolver la acción de sucesores(estado) con valor v

función: maxValor(estado, α, β) devuelve: valor de utilidad
        si esTerminal(estado) devolver utilidad(estado)
        para s ∈ sucesores(estado) hacer
                α ← max(α,  minValor(s, α, β))
                si α ≥ β devolver α
        devolver α

función: minValor(estado, α, β) devuelve: valor de utilidad
        si esTerminal(estado) devolver utilidad(estado)
        para s ∈ sucesores(estado) hacer
                β ← min(β,  maxValor(s, α, β))
                si α ≥ β devolver β
        devolver β
```

Minimax con poda αβ

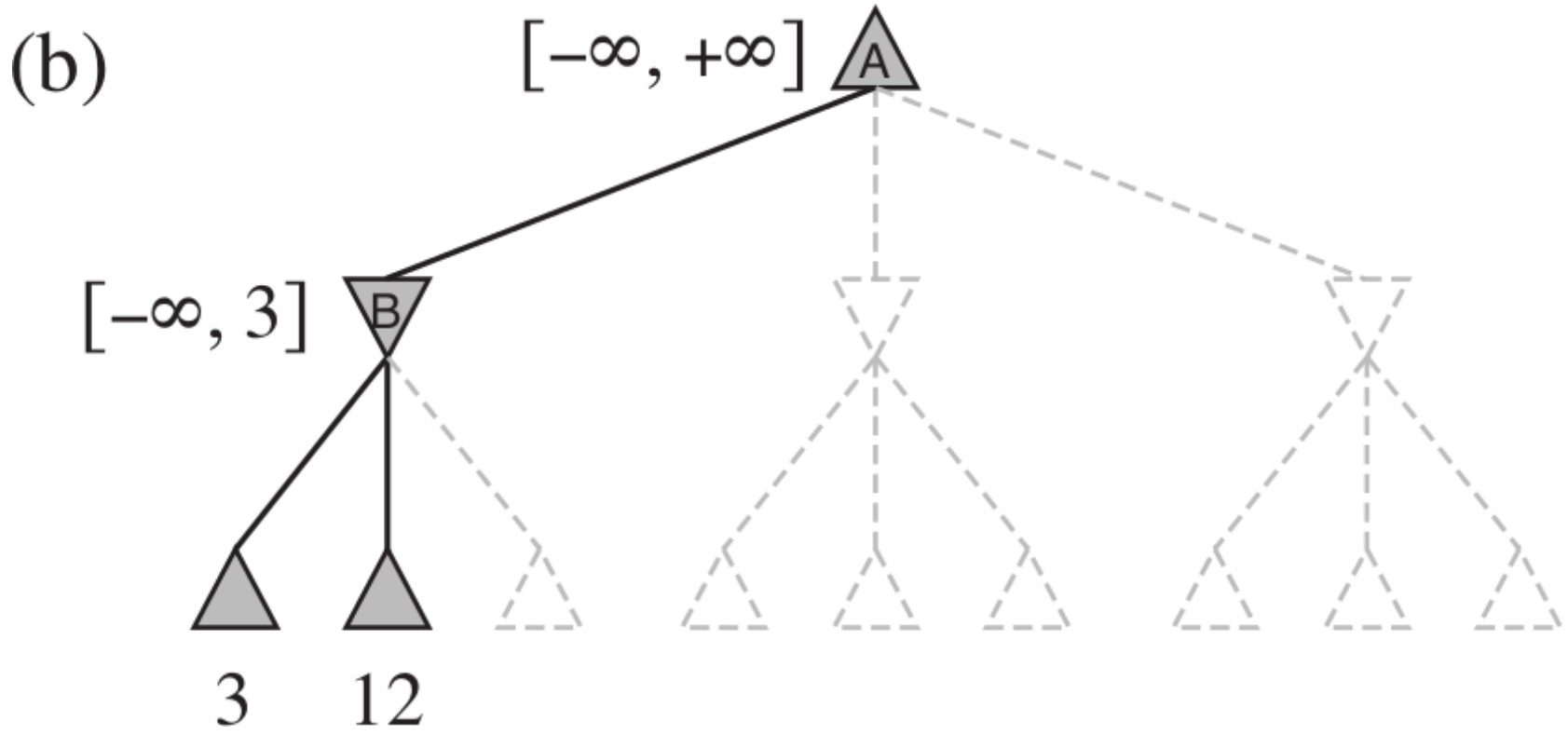
- La eficacia de la poda αβ es muy dependiente del orden en el que se examinan los sucesores.
- Si se generan primero los sucesores peores no se podrá realizar podas.
- Es conveniente tratar de examinar primero los sucesores que probablemente sean los mejores.
- En el mejor caso se explorarán $O(r^{p/2})$ nodos en lugar de $O(r^p)$ para minimax.

Minimax con poda αβ



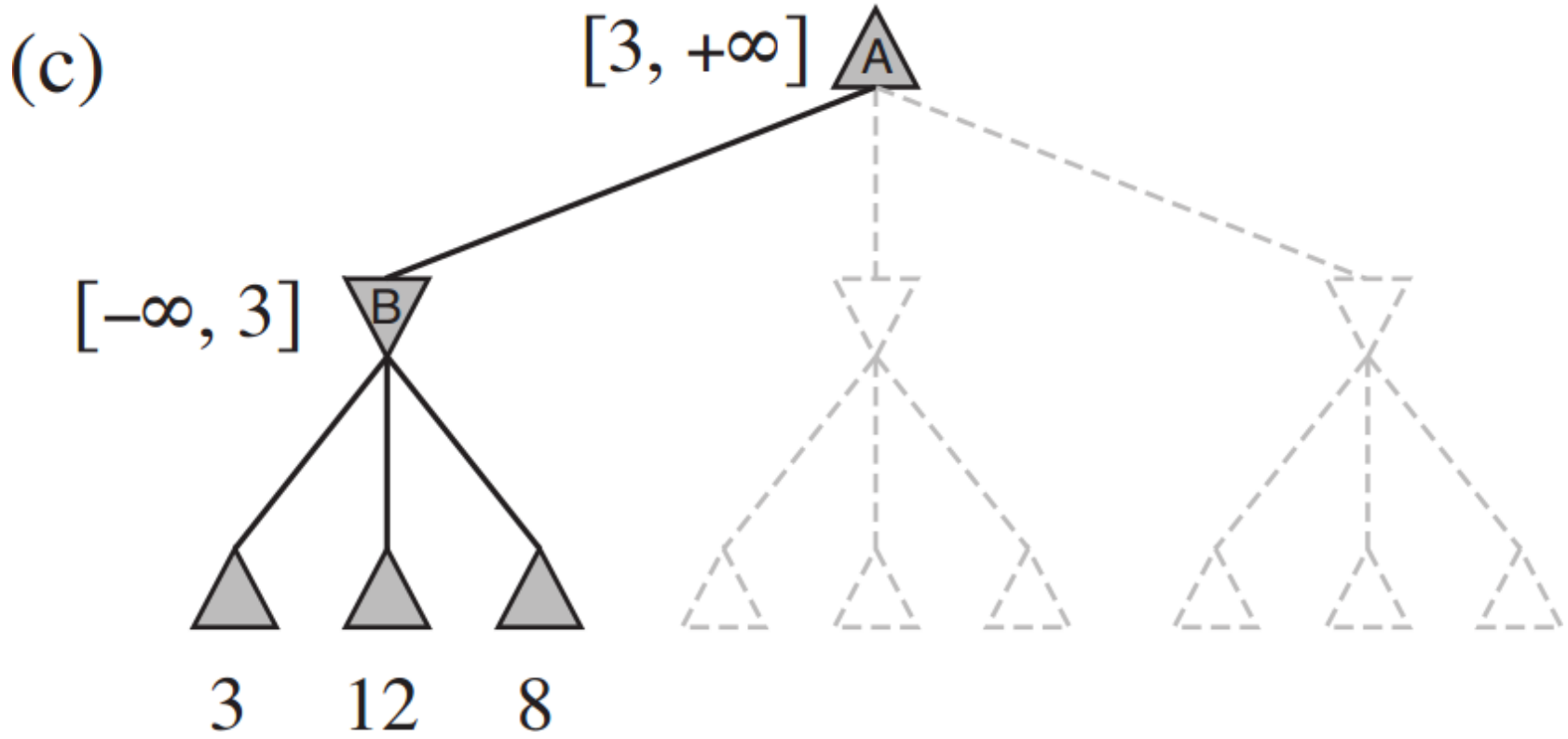
B es un nodo MIN, su valor es 3 como máximo.

Minimax con poda αβ



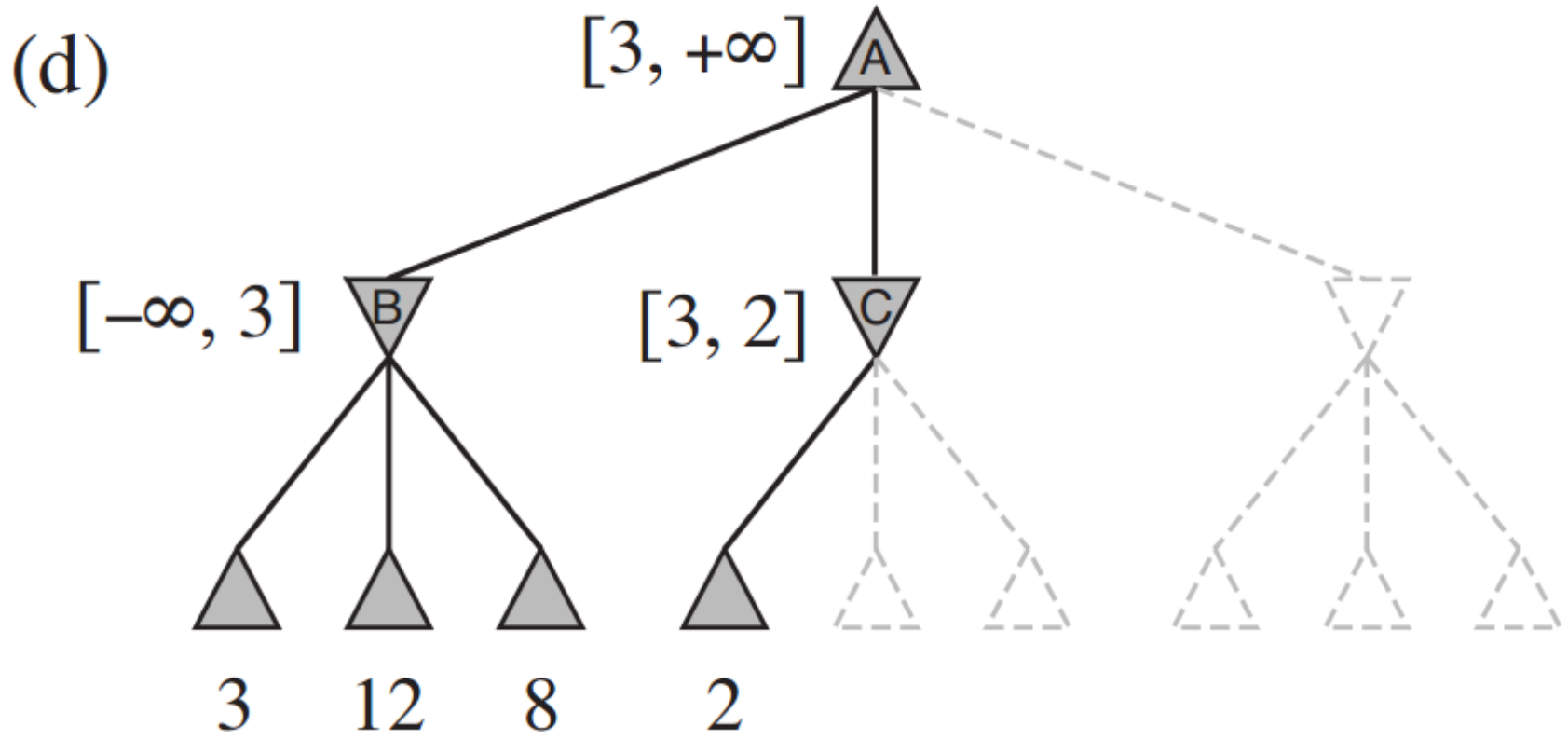
La segunda hoja de B vale 12. MIN evitará este movimiento. B todavía vale 3 como máximo.

Minimax con poda αβ



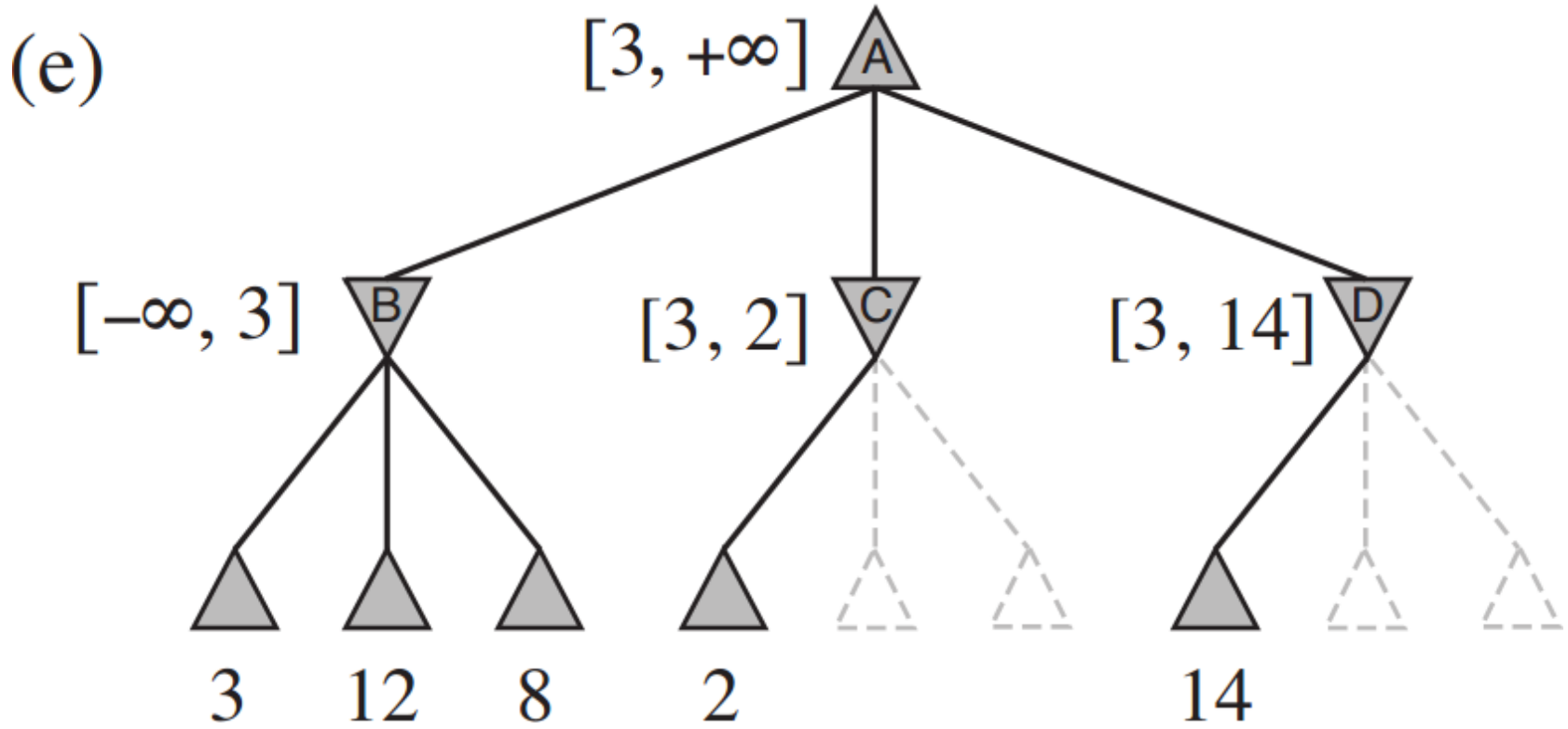
La tercera hoja de B vale 8. Ya se han visitado todos los sucesores, así que B vale 3 exactamente.
El valor de la raíz es al menos 3.

Minimax con poda αβ



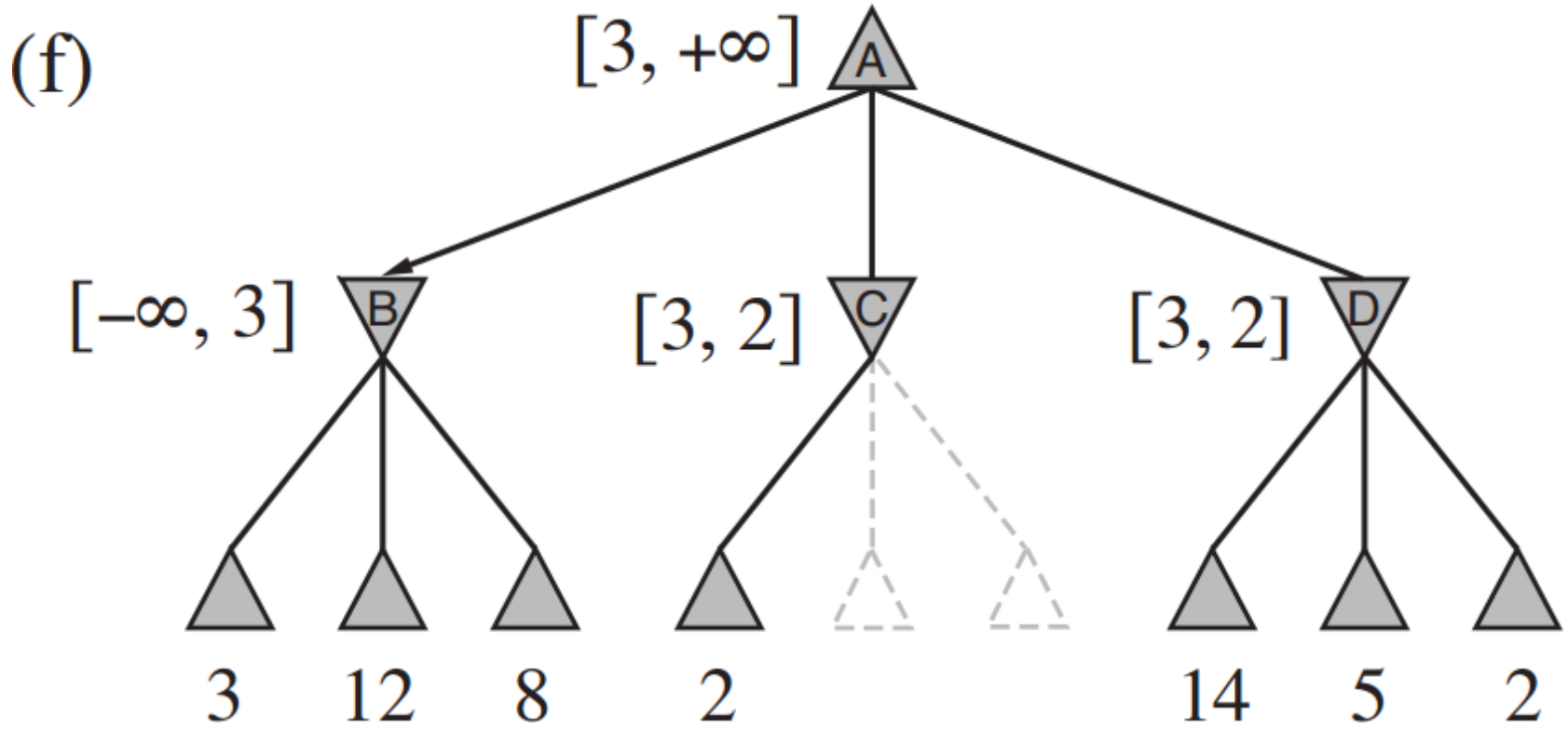
La primera hoja de C vale 2. Como C es un nodo MIN, vale 2 como máximo. Como B vale 3, MAX nunca elegiría C. No hay que seguir explorando C.

Minimax con poda αβ



La primera hoja de D vale 14. D vale 14 como máximo. Como es mayor que 3, hay que seguir explorando D. La raíz también es 14 como máximo.

Minimax con poda αβ



La segunda hoja de D vale 5, así que hay que seguir.
La 3ª hoja vale 2. D vale exactamente 2.
MAX decide moverse a B dando el valor 3.

Minimax con poda αβ

Otra forma de verlo:

MiniMax(raíz)

= max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))

= max(3, min(2, x, y), 2)

= max(3, z, 2), donde $z \leq 2$

= 3