Bloque 1 – Representación del conocimiento y búsqueda

Tema 4:
Resolución de problemas mediante búsqueda.
Búsqueda no informada.

Tema 4- Búsqueda no informada

1. Definición del problema: recordatorio y ejemplo
2. Búsqueda de soluciones
3. Estrategia en anchura
4. Estrategia de coste uniforme
5. Estrategia en profundidad
6. Búsqueda en CLIPS
7. Estrategia por profundización iterativa

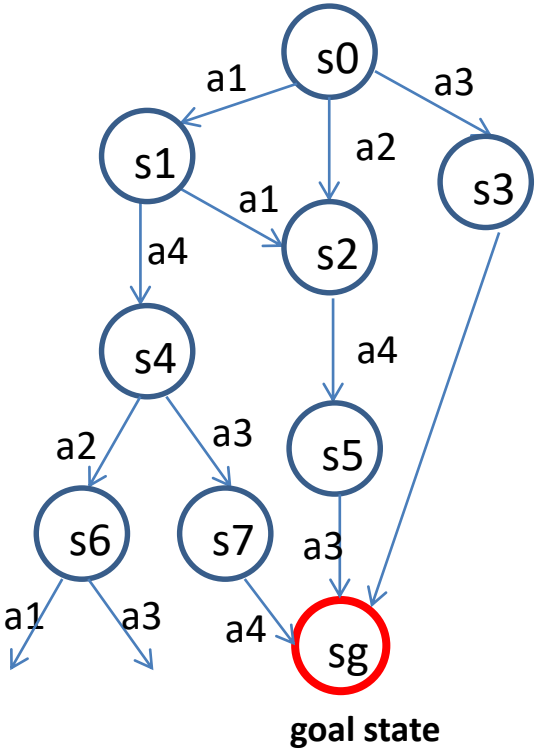
Bibliografía

S. Russell, P. Norvig. ***Inteligencia Artificial. Un enfoque moderno.*** Prentice Hall, 2nd edition, 2004 (Capítulo 3) http://aima.cs.berkeley.edu/2nd-ed/

1. Definición del problema: recordatorio

Conceptos:

- Representación basada en estados
- Estado inicial: *s0*
- Estado objetivo o final: *sg*
- Acciones del problema: {a1,a2,a3,a4}
- Modelo de transición: Result(s,a)
- Espacio de estados : grafo
- Camino



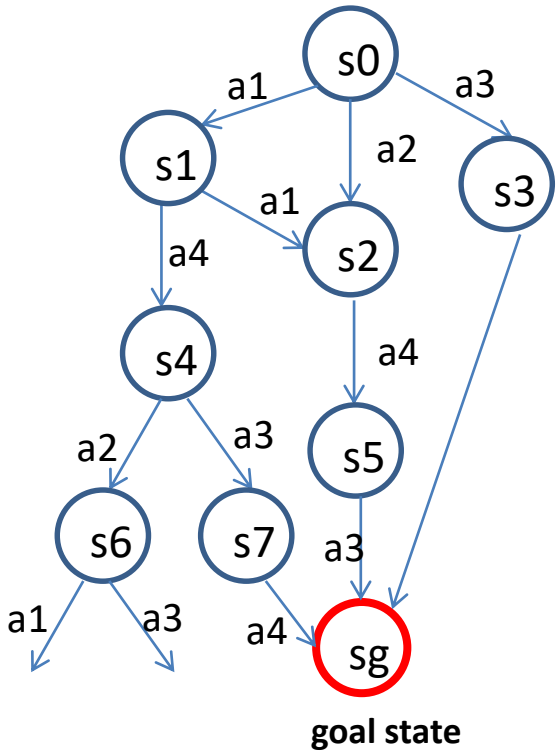
1. Definición del problema: recordatorio

Coste del camino: función que asigna un coste numérico a cada camino. Se describe como la suma de los costes de las acciones individuales del camino.

Generalmente, el coste de una acción es independiente del estado en el que se aplique (costes fijos de acciones):
$c(si,ai,sj)=c(ai)$

$g(sn)$: coste del estado sn; coste del camino desde el estado inicial $s0$ al estado sn

$$g(sn)=c(s0,a1,s1)+c(s1,a2,s2)+ \dots + c(sn-1,an, s_n)= c(a1)+c(a2)+\dots+c(an)$$



Ejemplos:

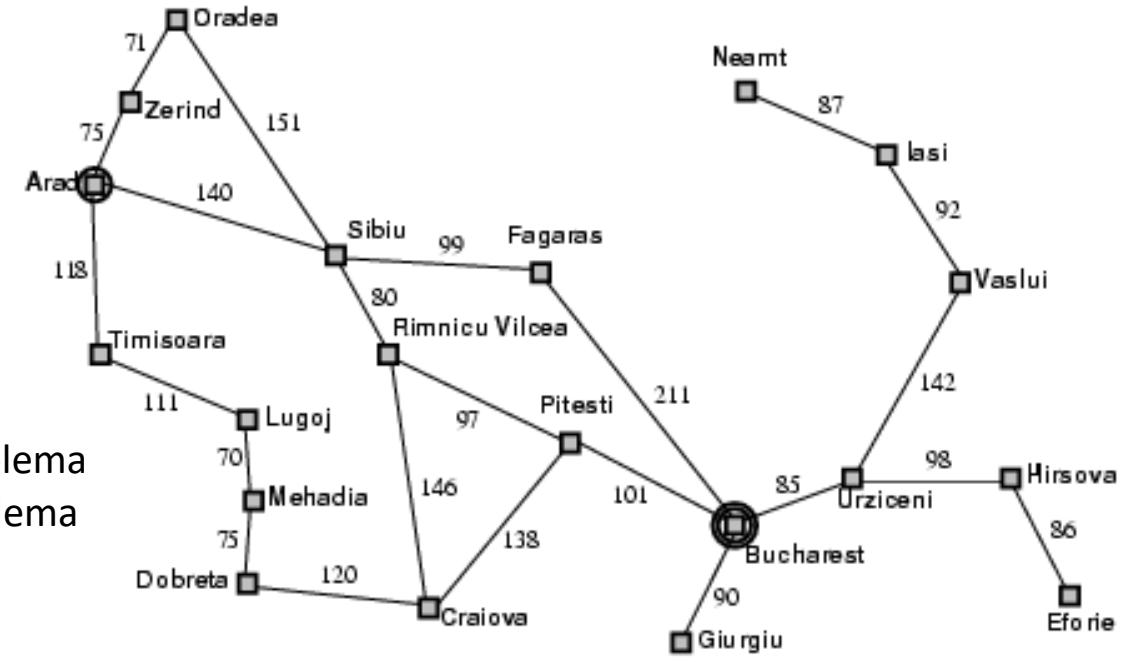
- $g(s5)= c(s0,a2,s2)+c(s2,a4,s5)=c(a2)+c(a4)$
- Hay dos caminos para alcanzar $s2$ desde $s0$:
  - $g(s2)=c(a2)$
  - $g(s2)=c(a1)+c(a1)$
  - Buscamos el camino de mínimo coste

1. Definición del problema: ejemplo

De vacaciones en Rumanía; estoy en Arad y quiero ir a Bucarest [Russell&Norvig, pp. 67-70]

Definición del problema

- Representación de estados: estar en una ciudad (Arad, Zerind, Fagaras, Sibiu, Bucarest …).
- Estado inicial: estoy en Arad
- Estado final: estar en Bucarest
- Acciones: conducir entre ciudades
- Solución: secuencia de ciudades de Arad a Bucarest; e.g. Arad, Sibiu, Fagaras, Bucarest,
- Coste del camino: número de km. de Arad a Bucarest (suma de km. de cada acción 'conducir')



El espacio de estados forma un grafo

- nodos representan los estados del problema
- arcos representan las acciones del problema

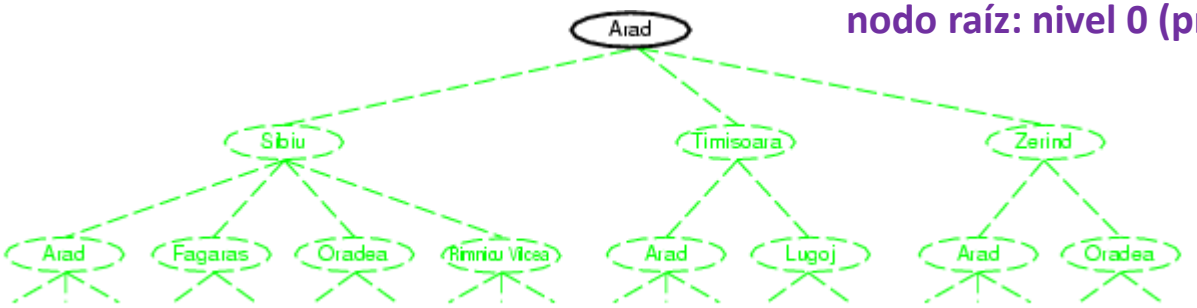
2. Búsqueda de soluciones

Proceso general de búsqueda

1. nodo-actual <- Estado inicial del problema
2. Comprobar si nodo-actual es el estado final del problema; en dicho caso, FIN.
3. Aplicar las acciones del problema en dicho estado y generar el conjunto de nuevos estados.
4. Escoger un nodo que no ha sido expandido todavía (nodo-actual)
5. Ir al paso 2

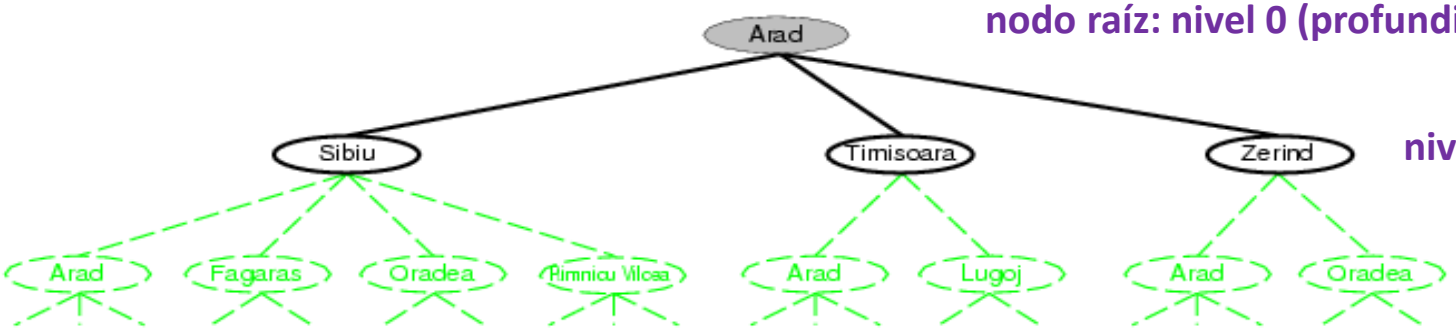
El conjunto de nodos no expandidos se denomina conjunto frontera, nodos hoja ó lista OPEN.

2. Búsqueda de soluciones: búsqueda en árbol



nodo raíz: nivel 0 (profundidad 0)

OPEN={Arad} ⇒ Escoger nodo y eliminarlo de la lista (expansión del nodo) ⇒ Arad objetivo?: NO ⇒ Generar hijos

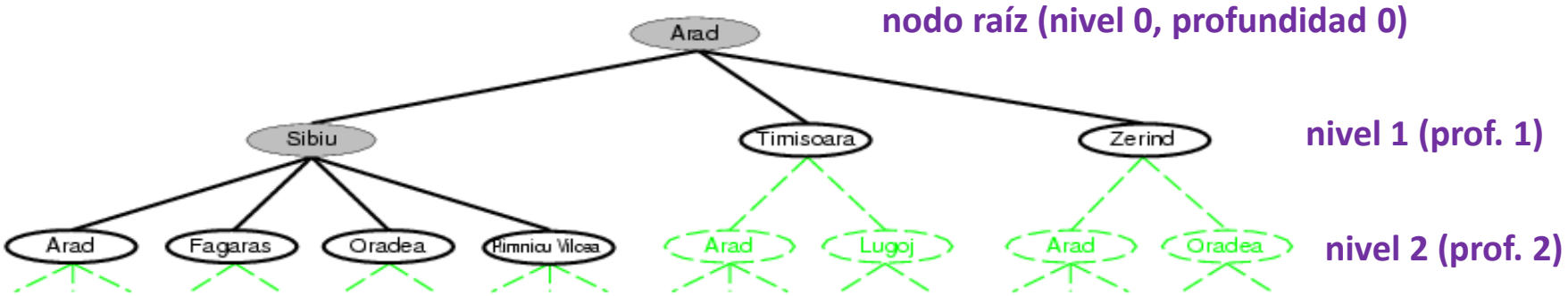


nodo raíz: nivel 0 (profundidad 0)

nivel 1 (prof. 1)

OPEN={Sibiu Timisoara Zerind} ⇒ Escoger nodo y eliminarlo de la lista (expansión del nodo) ⇒ Sibiu objetivo?: NO ⇓ Generar hijos

2. Búsqueda de soluciones: búsqueda en árbol



OPEN = {Timisoara Zerind Arad Fagaras Oradea Rimnicu Vilcea}

2. Búsqueda de soluciones: algoritmo TREE-SEARCH

- Los algoritmos de búsqueda comparten la estructura básica vista anteriormente; se diferencian, básicamente, en la elección del siguiente nodo a expandir (**estrategia de búsqueda**).

function TREE-SEARCH (*problema*) **return** una solución ó fallo
    Inicializar la lista OPEN con el estado inicial del problema
    do
        if lista OPEN está vacía **then return** *fallo*
        expandir un nodo: escoger nodo hoja y eliminarlo de la lista OPEN
        if nodo escogido es el estado final **then return** la correspondiente *solución*
        generar hijos y añadir los nodos resultantes a la lista OPEN
    enddo

2. Búsqueda de soluciones: estados repetidos

Espacio de búsqueda ≠ Espacio de estados (el árbol de búsqueda puede contener estados repetidos y ciclos); e.g.: el espacio de estados del problema 'Rumanía' solo tiene 20 estados mientras que el árbol completo de búsqueda es infinito)

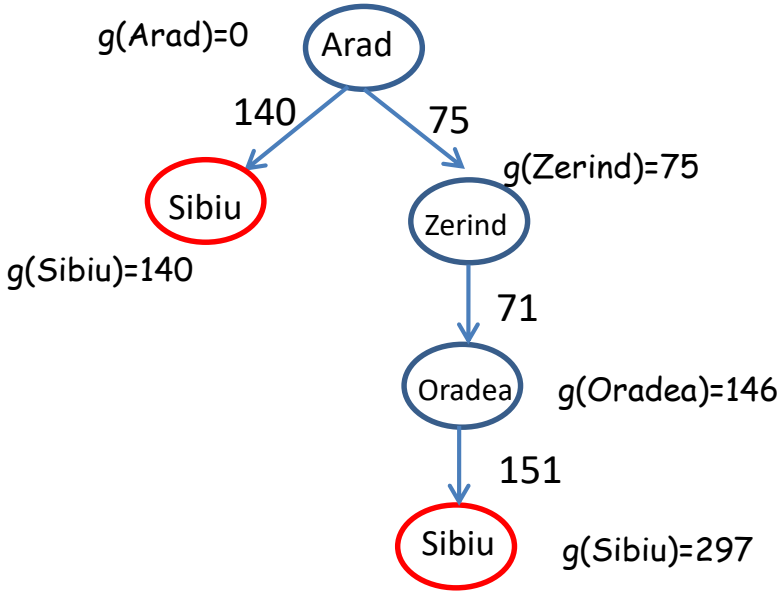
Estados repetidos:

- Acciones reversibles: ciclos

nodo repetido:
antecesor directo

nodo repetido: antecesor en el camino

caminos redundantes



2. Búsqueda de soluciones: estados repetidos

Como evitar estados repetidos

- Incluir en el algoritmo TREE-SEARCH una estructura de datos para guardar los nodos explorados ó expandidos (lista CLOSED)
- Cuando se genera un nuevo nodo, si éste se encuentra en la lista CLOSED (nodos ya expandidos) ó en la lista OPEN (nodos aún no expandidos) se puede eliminar en lugar de añadirlo a la lista OPEN
- Nuevo algoritmo: GRAPH-SEARCH, algoritmo que separa el grafo del espacio de estados en dos regiones: la región de nodos explorados/expandidos y la región de nodos no expandidos.

En general, evitar nodos repetidos y caminos redundantes es solo una cuestión de eficiencia.

2. Búsqueda de soluciones: algoritmo GRAPH-SEARCH

La lista OPEN se implementa como una cola de prioridades (priority queue): se extrae el elemento de la cola con la máxima prioridad de acuerdo a una función de evaluación (lista ordenada de modo que el primer elemento de la cola es el que tiene mayor prioridad)

Para cada estrategia de búsqueda se define una función de evaluación ($f(n)$) que devuelve un valor numérico para el nodo **n** tal que el nodo se inserta en la cola de prioridades en el mismo orden en el que sería expandido por la estrategia de búsqueda.

2. Búsqueda de soluciones: algoritmo GRAPH-SEARCH

function GRAPH-SEARCH (*problema*) **return** una solución ó un fallo
    Inicializar la lista OPEN con el estado inicial del problema
    Inicializar la lista CLOSED a vacío
    do
        if OPEN está vacía **then return** *fallo*
        p <- pop (lista OPEN)
        añadir p a la lista CLOSED
        if p = estado final **then return** *solución* p
        generar hijos de p
        para cada hijo n de p:
            aplicar f(n)
            if n no está en CLOSED **then**
                if n no está en OPEN o (n está repetido en OPEN y f(n) es mejor que el valor del nodo en OPEN) **then**
                    insertar n en orden creciente de f(n) en OPEN*
            else if f(n) es mejor que el valor del nodo repetido en CLOSED
                then escoger entre re-expandir n (insertarlo en OPEN y eliminarlo de CLOSED) o descartarlo
    enddo

* Como estamos interesados en encontrar únicamente la primera solución, se puede eliminar el nodo repetido de OPEN

2. Búsqueda de soluciones: algoritmo GRAPH-SEARCH

Búsqueda en árbol (TREE-SEARCH):

- mantiene la lista OPEN pero no la lista CLOSED con todos los nodos expandidos (menos memoria)
- puede evitar estados repetidos en la lista OPEN
- re-expande nodos ya explorados

Búsqueda en grafo (GRAPH-SEARCH):

- mantiene la lista OPEN y CLOSED (mayores requerimientos de memoria)
- control de estados repetidos y caminos redundantes (reducción de la búsqueda)

2. Búsqueda de soluciones: propiedades

Evaluaremos las estrategias de búsqueda de acuerdo a:

1. Completitud
2. Complejidad temporal
3. Complejidad espacial
4. Optimalidad

Soluciones que representen un balance entre:

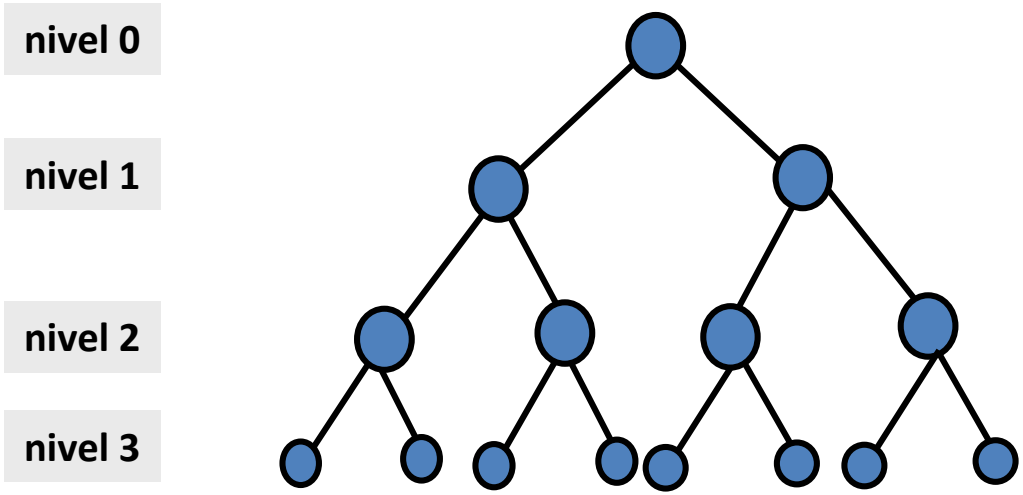
- coste del camino solución: $g(estado\_final)$
- coste de la búsqueda: coste de encontrar el camino solución

Tipos de estrategias de búsqueda:

1. no informada o búsqueda ciega (orden de expansión de los nodos)
2. Informada o búsqueda heurística (expansión 'inteligente' de los nodos)

3. Anchura

Expandir el nodo menos profundo (entre los nodos de la lista OPEN)



Función de evaluación (cola de prioridades): f(n)=nivel(n)=profundidad(n)

3. Anchura: propiedades

- Completa
- Óptima :
  - Anchura siempre devuelve el camino solución más corto (menos profundo)
  - El camino más corto es óptimo si todas las acciones tienen el mismo coste y el camino es una función no decreciente de la profundidad del nodo (costes no negativos)
- Complejidad temporal para factor de ramificación b y profundidad d:

  nodos expandidos $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d$ $O(b^d)$

  nodos generados $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d + (b^{d+1} - b)$ $O(b^{d+1})$
- Complejidad espacial para factor de ramificación b y profundidad d:
  - En el algoritmo GRAPH-SEARCH, todos los nodos residen en memoria
  - $O(b^d)$ en la lista CLOSED y $O(b^{d+1})$ en la lista OPEN (dominado por el tamaño de OPEN)
- Conclusiones:
  - Coste espacial más crítico que el coste temporal
  - Coste temporal inviable para valores altos de b y d

3. Anchura

Requerimientos de tiempo y memoria para anchura

b = 10
100.000 nodos generados/segundo
1000 bytes de almacenamiento/nodo

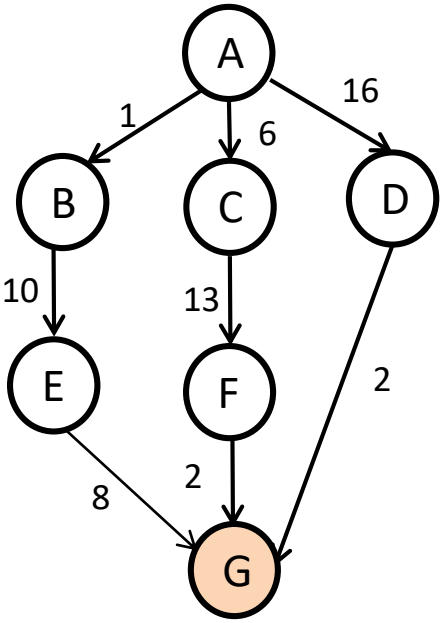
(Datos tomados del libro 3rd edition of the *Artificial Intelligence. A modern approach*)

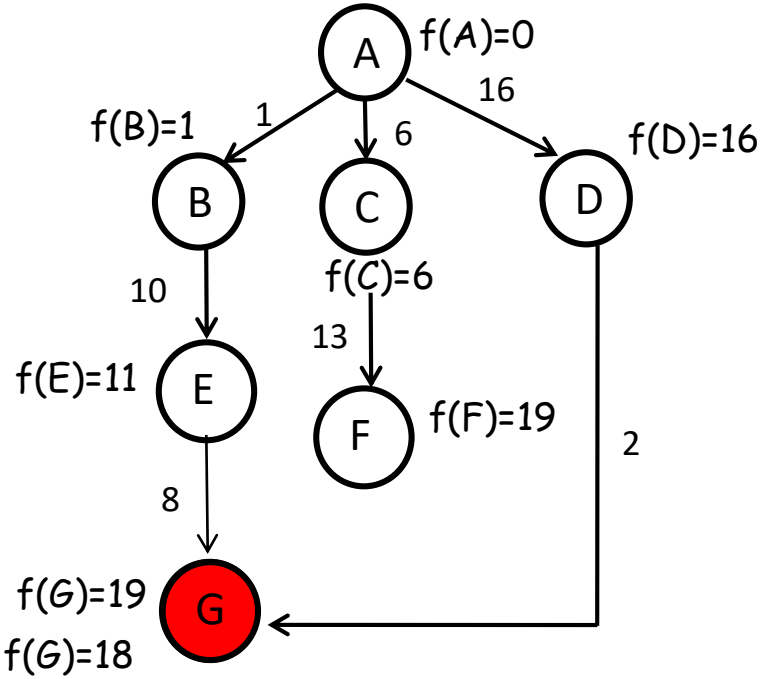
Profundidad	Nodos	Tiempo	Memoria
2	110	1.1 ms	107 kilobytes
4	11110	111 ms	10.6 megabytes
6	10^6	11 s.	1 gigabytes
8	10^8	19 min.	103 gigabytes
10	10^{10}	31 horas	10 terabytes
12	10^{12}	129 días	1 petabytes
14	10^{14}	35 años	99 petabytes
16	10^{16}	3500 años	10 exabytes

4. Coste uniforme

Expandir el nodo con el menor coste (menor valor de $g(n)$)

Función de evaluación (cola de prioridades): $f(n)=g(n)$





4. Coste uniforme

- Completa
  - si los costes de las acciones ≥ ε (constante positiva, costes no negativos)

- Óptima :
  - si los costes de las acciones son no negativos $g(sucesor(n)) > g(n)$
  - coste uniforme expande nodos en orden creciente de coste

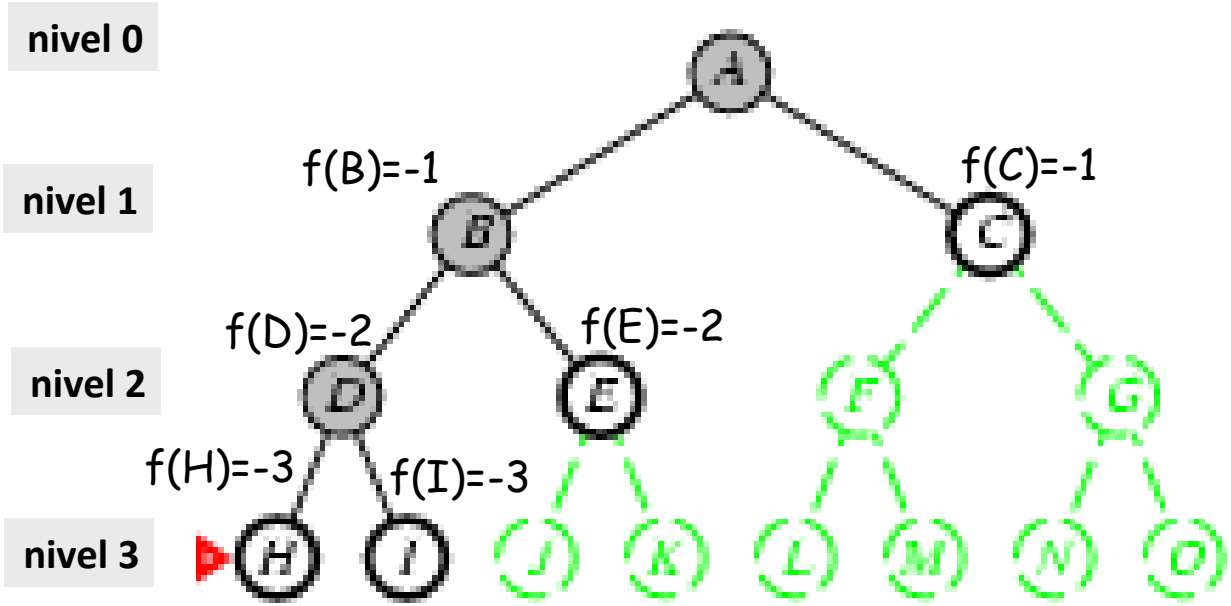
- Complejidad temporal y espacial:

  - sea C^* el coste de la solución óptima
  - asumimos que todas las acciones tienen un coste mínimo de ε
  - complejidad temporal y espacial: $O(b^{C^*/\varepsilon})$

5. Profundidad: búsqueda en árbol (TREE-SEARCH)

Expandir el nodo más profundo (entre los no expandidos)

Función de evaluación (cola de prioridades): f(n)=-nivel(n)



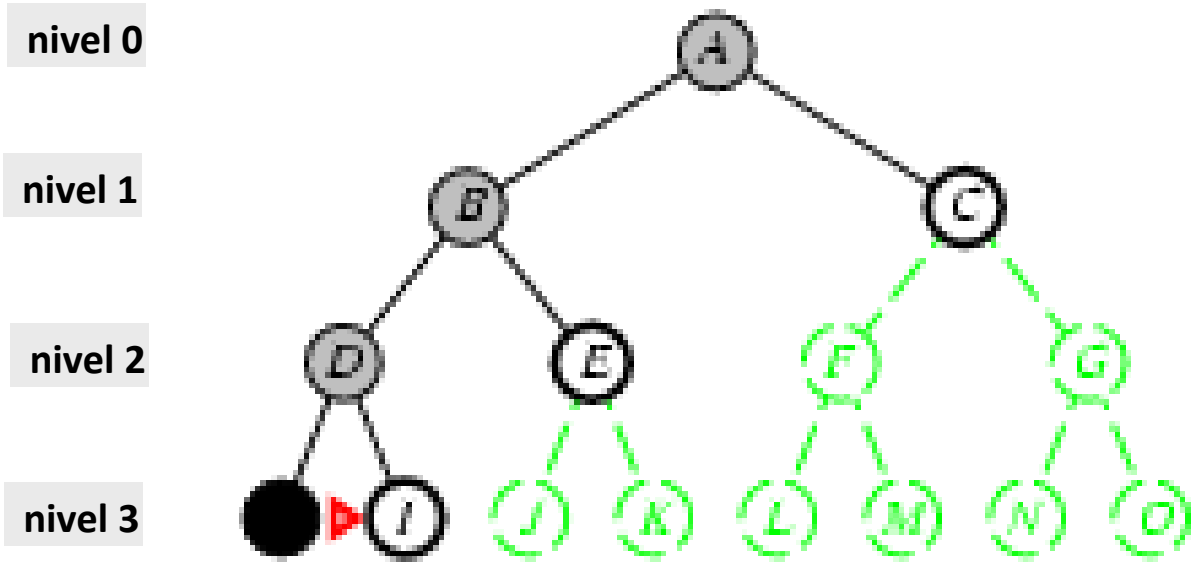
Backtracking :

1. nodo muerto, nodo no objetivo y no expandible (no hay acciones aplicables)
2. límite de profundidad máximo (definido por el usuario, m=3 en este ejemplo)
3. estado repetido (opcional si se aplica control de nodos repetidos)

Se mantiene una lista PATH para aplicar Backtracking: almacena los nodos expandidos del camino actual y se eliminan cuando se llama a la función Backtracking

5. Profundidad: búsqueda en árbol (TREE-SEARCH)

BACKTRACKING(n):
1. Eliminar n de la lista PATH
2. Si parent(n) no tiene más hijos en OPEN => BACKTRACKING (parent(n))
3. Si parent(n) tiene más hijos en OPEN => escoger siguiente nodo de la lista OPEN

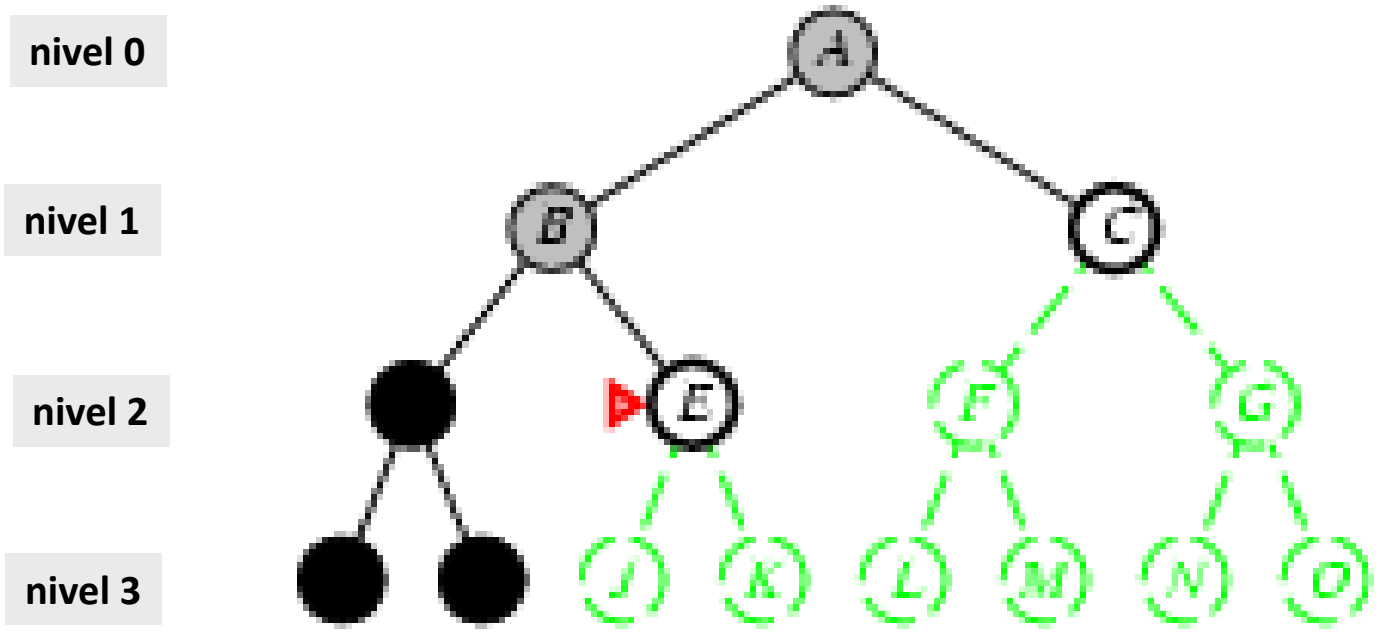


Eliminar H de la lista OPEN:
1. poner H en la lista PATH
2. comprobar si H es objetivo: NO
3. comprobar si H está en el máximo nivel de profundidad (m=3): SI => BACKTRACKING (H)

lista OPEN = {I(-3),E(-2),C(-1)}

lista PATH = {A,B,D}

5. Profundidad: búsqueda en árbol (TREE-SEARCH)



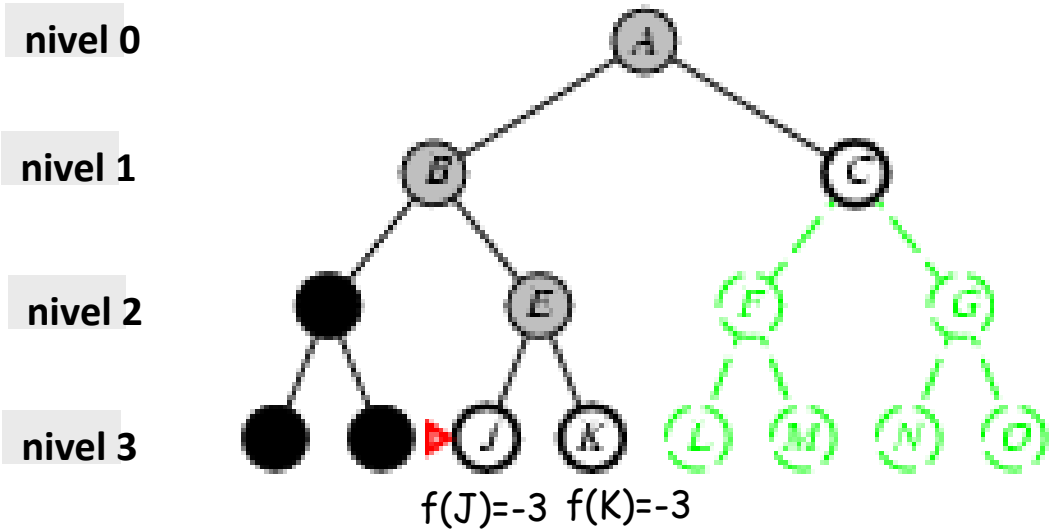
Eliminar I de la lista OPEN:

1. poner I en la lista PATH
2. comprobar si I es objetivo: NO
3. comprobar si I está en el máximo nivel de profundidad (m=3): SI => BACKTRACKING (I)

lista OPEN = {E(-2),C(-1)}

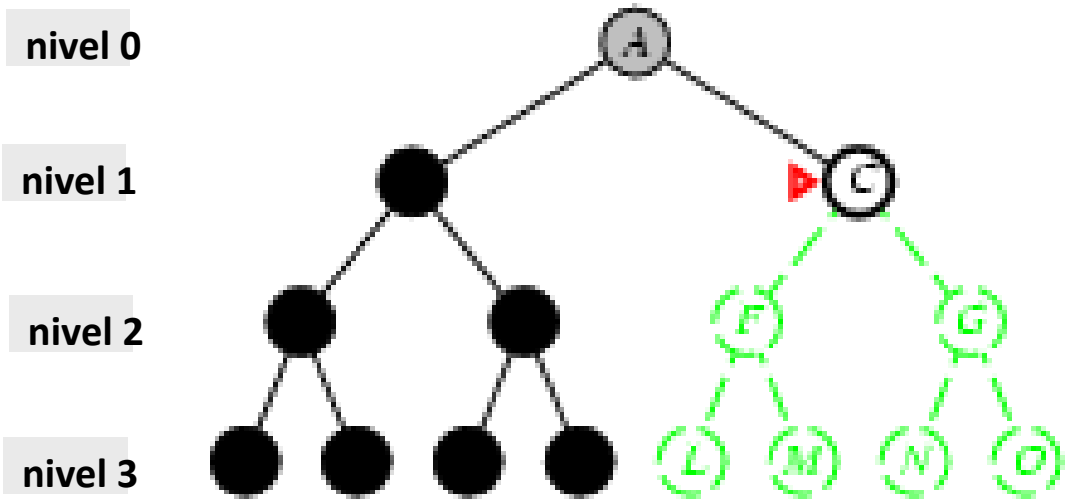
lista PATH= {A,B}

5. Profundidad: búsqueda en árbol (TREE-SEARCH)



lista OPEN = {J(-3),K(-3),C(-1)}

lista PATH = {A,B,E}



lista OPEN = {C(-1)}

lista PATH = {A}

5. Profundidad: propiedades

- Complejidad temporal:
  - Para un árbol de máxima profundidad **m**, $O(b^m)$
  - si **m=d** entonces profundidad explora tantos nodos como anchura. Pero **m** puede ser un valor mucho más grande que **d**

- Complejidad espacial:
  - La versión TREE-SEARCH solo almacena el camino del nodo raíz al nodo hoja actual (lista PATH), junto con los nodos hermanos no expandidos de los nodos del camino (lista OPEN).
  - Para un factor de ramificación **b** y máxima profundidad **m**, $O(b.m)$.
  - Listas OPEN y PATH contienen muy pocos nodos: apenas existe control de nodos repetidos. En la práctica, profundidad genera el mismo nodo varias veces.
  - Aún así, la versión TREE-SEARCH de profundidad puede ser más rápida que anchura

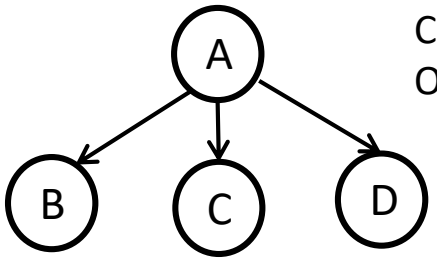
- Completitud:
  - Si no hay máximo nivel de profundidad y no hay control de nodos repetidos => no es completa
  - Si no hay máximo nivel de profundidad y hay control de nodos repetidos => es completa
  - Si hay máximo nivel de profundidad (**m**) podría perder la solución si ésta no se encuentra en el espacio de búsqueda definido por **m** => no es completa
- No óptima

6. Búsqueda en CLIPS

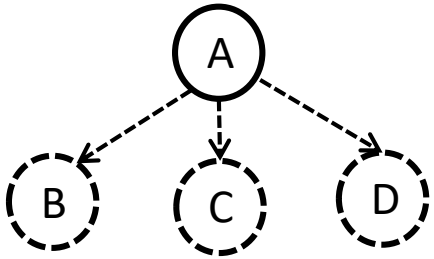
- Ver práctica 1 (boletín del puzzle) para detalles de la implementación de la búsqueda en CLIPS

RESUMEN:

- No hay control de estados repetidos (hechos que representan el mismo estado son diferentes por campos como el *nivel*).
- Reglas aplicables = Nodos generados = lista OPEN
- Reglas ejecutadas = Nodos expandidos = lista CLOSED



CLOSED List= {A}
OPEN List={B, C, D}



WM= {A}
Agenda={B, C, D}

- Anchura: seleccionar opción *Breadth* en la agenda CLIPS
- Profundidad: seleccionar opción *Depth* en la agenda CLIPS
- CLIPS implementa la búsqueda en grafo (GRAPH-SEARCH) de Anchura y Profundidad.

7. Profundización iterativa

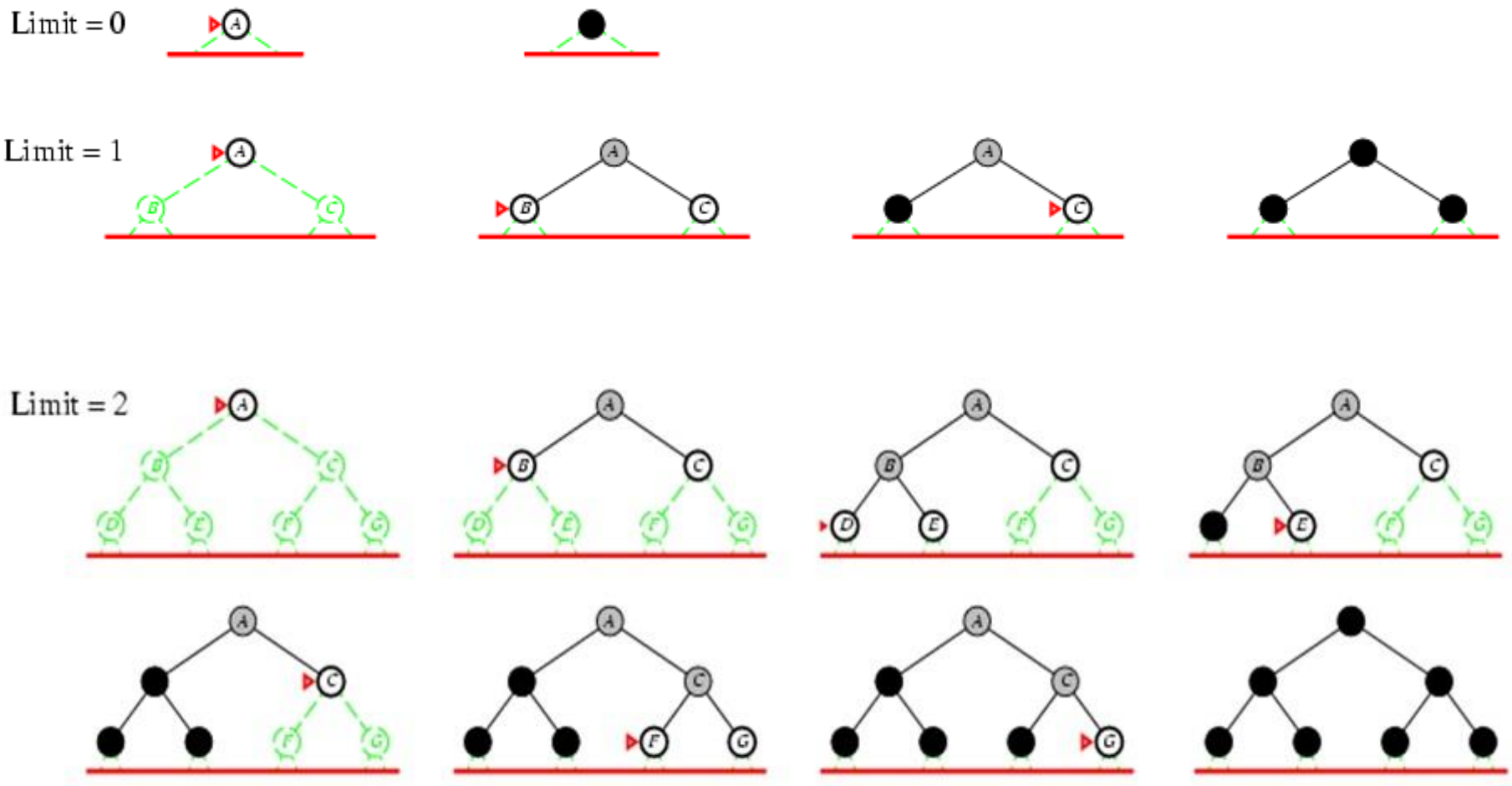
```
function Iterative_Deepening_Search (problem) returns (solution, failure)
    inputs: problem /*a problem*/
      for depth = 0 to ∞ do
          result = depth_limited_search (problem, depth)
          if result <> failure return result
      end
      return failure
end function
```

```
depth_limited_search  (problem, limit)
.....
   if  goal_test(node) then return SOLUTION(node)
        else  if depth(node) = limit then backtracking
                   else   generate_successors (node)
....
```

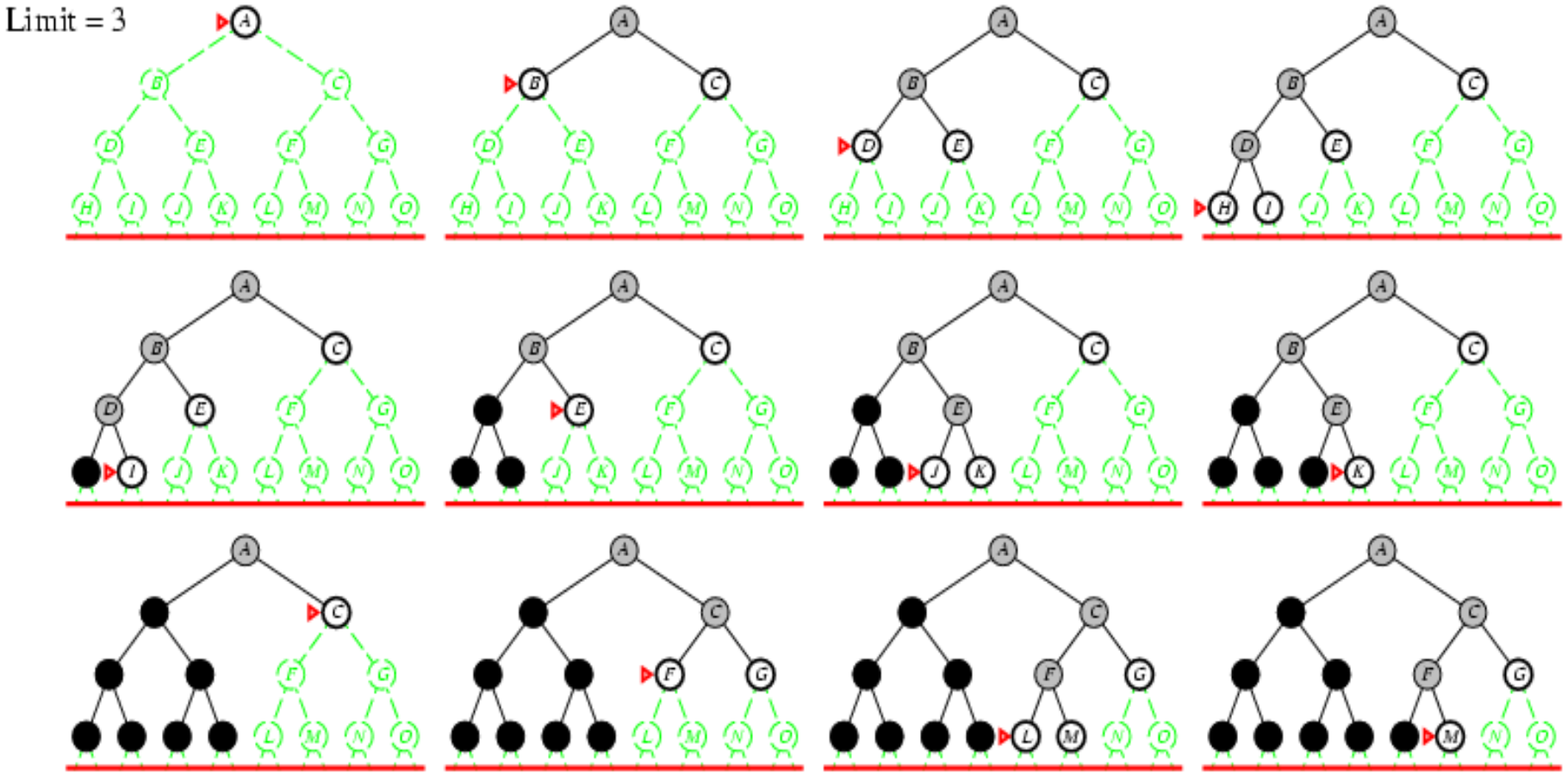
Realiza **iterativamente** una **búsqueda limitada en profundidad**, desde una profundidad-máxima 0 hasta ∞.

- Resuelve la dificultad de elección del límite adecuado de la búsqueda en profundidad.
- Combina ventajas de búsqueda primero en amplitud y primero en profundidad.
- **Completa y admisible.**
- Complejidad temporal $O(b^d)$, complejidad espacial $O(b \cdot d)$

7. Profundización iterativa



7. Profundización iterativa



7. Profundización iterativa

- Número de nodos generados para b=10, d=5:
  - Profundización iterativa: $(d)\cdot b + (d-1)\cdot b^2 + (d-2)\cdot b^3 + \ldots\ldots\ldots + 1\cdot b^d$ 123.456
  - Anchura: $1 + b + b^2 + \ldots\ldots\ldots\ldots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b)$ 1.111.100

PI puede parecer ineficiente porque genera estados repetidamente, pero realmente no es así:

- En un árbol de búsqueda con ramificación similar en cada nivel, la mayor parte de los nodos está en el nivel inferior.
- Los nodos de nivel inferior (d) son generados una sola vez, los anteriores dos veces, etc. Los hijos de la raíz se generan d veces.

La búsqueda en anchura generará algunos nodos en profundidad d+1, mientras que PI no lo hace. Por ello, PI es en realidad más rápida que anchura.

PI es el método de búsqueda no informada preferido cuando el espacio de búsqueda es grande y no se conoce a priori la profundidad de la solución.

Resumen de búsqueda no informada

Criterio	Anchura	Coste uniforme	Prof.	Profundización iterativa
Temporal	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^d)$
Espacial	$O(b^d)$	$O(b^{C^*/\varepsilon})$	$O(b.m)$	$O(b.d)$
Optima?	Sí*	Sí	No	Sí*
Completa?	Sí	Sí**	No	Sí

* Óptima si los costes de las acciones son todos iguales

** Completa si los costes de las acciones ≥ ε para un ε positivo