Bloque 1 – Representación del conocimiento y búsqueda

Tema 1:
Resolución de problemas mediante búsqueda.
Búsqueda no informada.

Bloque 1, Tema 1- Búsqueda no informada

1. Formulación del problema
2. Búsqueda de soluciones
3. Estrategia en anchura
4. Estrategia de coste uniforme
5. Estrategia en profundidad
6. Estrategia por profundización iterativa

Bibliografía

- S. Russell, P. Norvig. ***Artificial Intelligence. A modern approach.*** Prentice Hall, 4th edición, 2022 (Capítulo 3) http://aima.cs.berkeley.edu/
- S. Russell, P. Norvig. ***Artificial Intelligence. A modern approach.*** Prentice Hall, 3rd edición, 2010 (Capítulo 3)
- S. Russell, P. Norvig. ***Inteligencia artificial . Una aproximación moderna.*** Prentice Hall, 2ª edición, 2004 (Capítulos 3 y 4) http://aima.cs.berkeley.edu/2nd-ed/

1. Formulación del problema

Resolución de problemas mediante búsqueda en un espacio de estados:

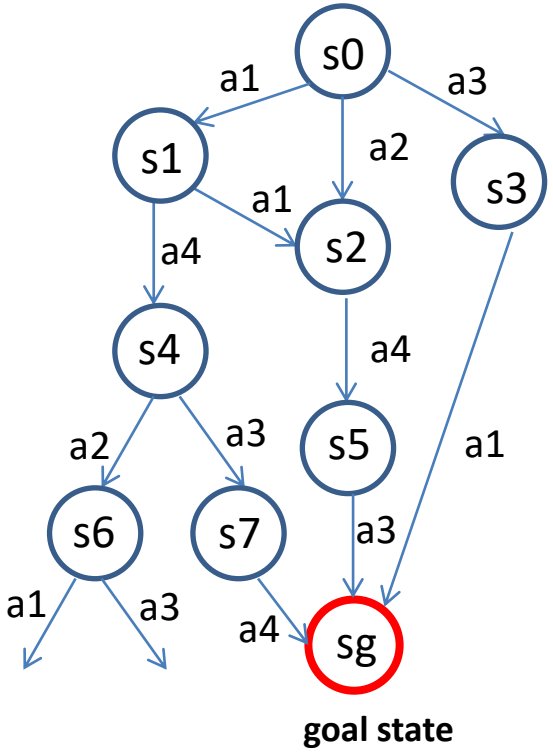
Problemas que se pueden resolver mediante un proceso que partiendo de una situación (estado) inicial, y utilizando un conjunto de acciones/reglas es capaz de determinar el conjunto de pasos que permiten alcanzar una situación final (estado objetivo).

Formulación del problema:

- **Espacio de estados del problema**: conjunto de posibles estados en los que puede encontrarse el dominio del problema
- **Estado inicial**
- **Uno o más estados objetivo**
- **Conjunto de acciones del problema**: *Actions*(**s**) devuelve un conjunto finito de acciones del problema que son aplicables en el estado **s**, es decir, que se pueden ejecutar en **s**
- **Modelo de transición**: *Result*(**s**,**a**) devuelve el estado resultante de ejecutar la acción **a** en el estado **s**
- **Función de coste de las acciones**: *Action-Cost*(**s**,**a**,**s'**)

1. Formulación del problema

- **Estado inicial**: *s0*
- **Estados objetivo**: {*sg*}
- **Conjunto de acciones del problema**: {a1,a2,a3,a4}
  Actions(s0) ={a1,a2,a3} Actions(s1)={a1,a4}, …
- **Modelo de transición**:
  Result(s0,a2)=s2, Result(s4,a3)=s7, …
- **Función de coste de las acciones**:
  Action-Cost(s0,a1,s1)=2, Action-Cost(s4,a3,s7)=1, …
- **Espacio de estados**: el grafo de la figura muestra el espacio de búsqueda que es un conjunto menor o igual que todo el espacio de estados del problema



1. Formulación del problema

Elementos del proceso de búsqueda:

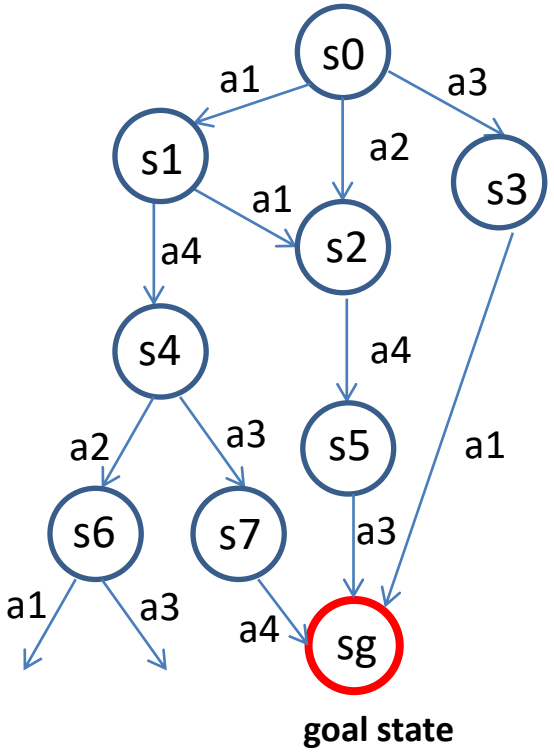
Camino: secuencia de acciones

Camino solución: secuencia de acciones desde el estado inicial *s0* hasta el estado objetivo *sg*

Coste del camino: 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):
Action-cost(s,a,s')=Action-cost(a)

Coste de un nodo (estado): *g(s)* devuelve el coste del camino desde el estado inicial *s0* hasta el estado *s*



- *g(s1)=Action-cost(a1)*
- *g(s2)=Action-cost(a2)* *g(s2)=Action-cost(a1)+Action-cost(a1)*
- *g(s7)=Action-cost(a1)+Action-cost(a4)+Action-cost(a3)*

1. Formulación del problema: ejemplo 1, Rumanía

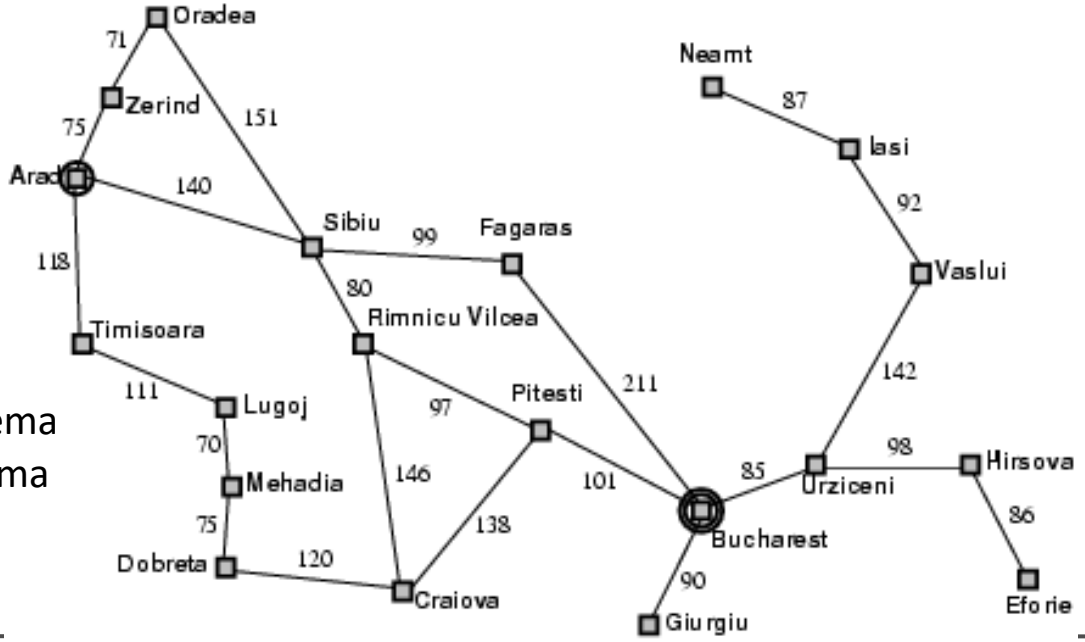
De vacaciones en Rumanía; estoy en Arad y quiero ir a Bucarest [Russell&Norvig, pp. 82]

Definición del problema

- Espacio de estados: conjunto de ciudades de Rumanía {Arad, Zerind, Fagaras, Sibiu, Bucarest …}.
- Estado inicial: estoy en Arad
- Estado final: estar en Bucarest
- Acciones: conducir entre ciudades
- Modelo de transición: el estado resultante es la ciudad destino de la acción de conducir
- 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')

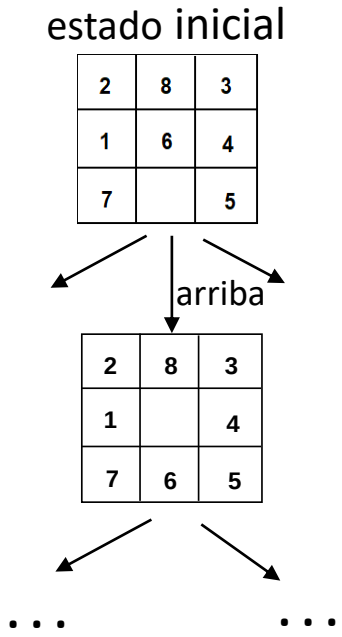
El espacio de estados forma un grafo

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



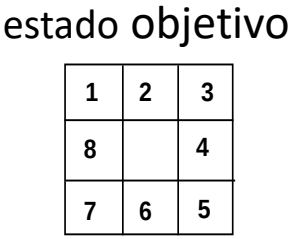
1. Formulación del problema: ejemplo 2, 8-puzzle

Tablero de 3*3 casillas. 8 de las casillas contienen una pieza o ficha que se puede deslizar a lo largo del tablero horizontal y verticalmente. Las fichas vienen marcadas con los números del 1 al 8. Hay una casilla libre en el tablero que permite los movimientos de las fichas.



Definición del problema

- Espacio de estados: 9! = 362.880 estados
- Estado inicial: una configuración del puzzle
- Estado objetivo: una configuración del puzzle
- Acciones: movimientos de las fichas (o movimientos de casilla libre) en 4 direcciones {arriba, abajo, izquierda, derecha}
- Función de coste de las acciones: Action-cost(a)=1 para todas las acciones del problema
- Modelo de transición: estado resultante del movimiento de la ficha
- Solución: secuencia de movimientos de las fichas desde estado inicial a estado objetivo
- Coste del camino: número de movimientos del camino



2. Búsqueda de soluciones

Una solución es una secuencia de acciones, por lo que los algoritmos de búsqueda funcionan considerando varias secuencias de acciones posibles.

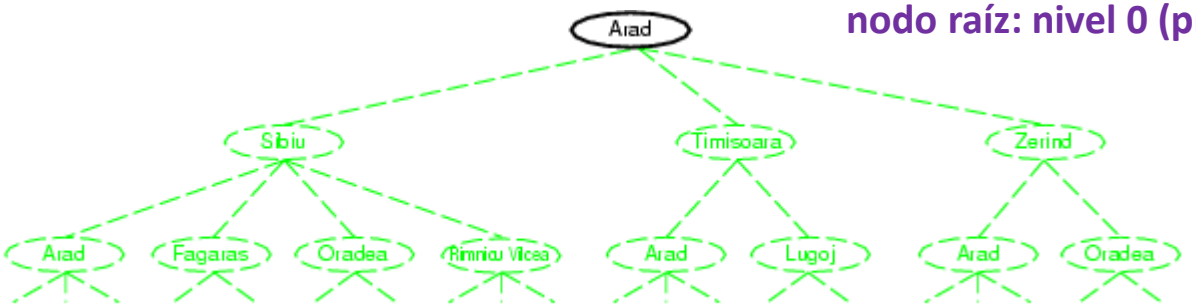
Las posibles secuencias de acciones a partir del estado inicial forman un árbol de búsqueda con el estado inicial en la raíz; las ramas son acciones y los nodos corresponden a estados en el espacio de estados del problema

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. Determinar las acciones del problema que son aplicables en nodo-actual y generar el conjunto de nuevos estados hijo; añadir una rama del nodo padre a cada nodo hijo
4. nodo-actual <- escoger un nodo del conjunto de nodos que no ha sido expandido todavía (expansión del nodo)
5. Ir al paso 2

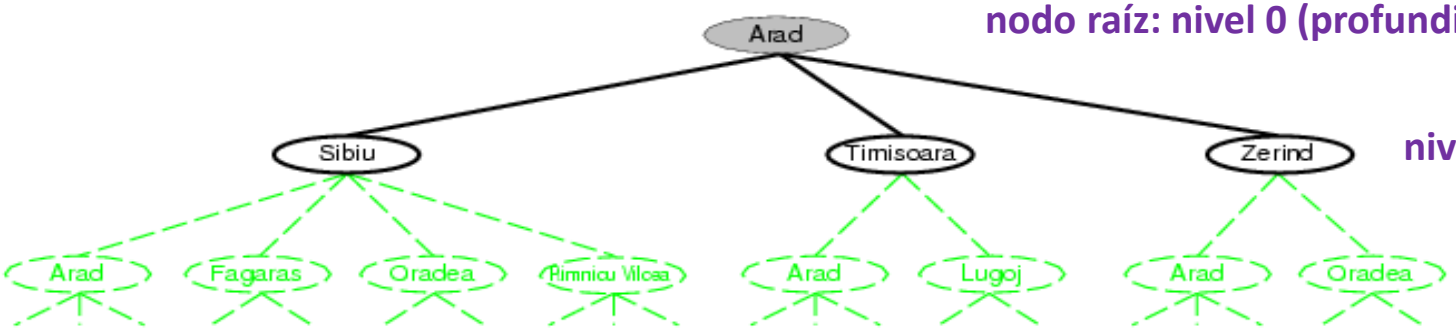
El conjunto de nodos no expandidos se denomina conjunto frontera, nodos hoja ó lista OPEN.

2. Búsqueda de soluciones: búsqueda en árbol



nodo raíz: nivel 0 (profundidad 0)

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

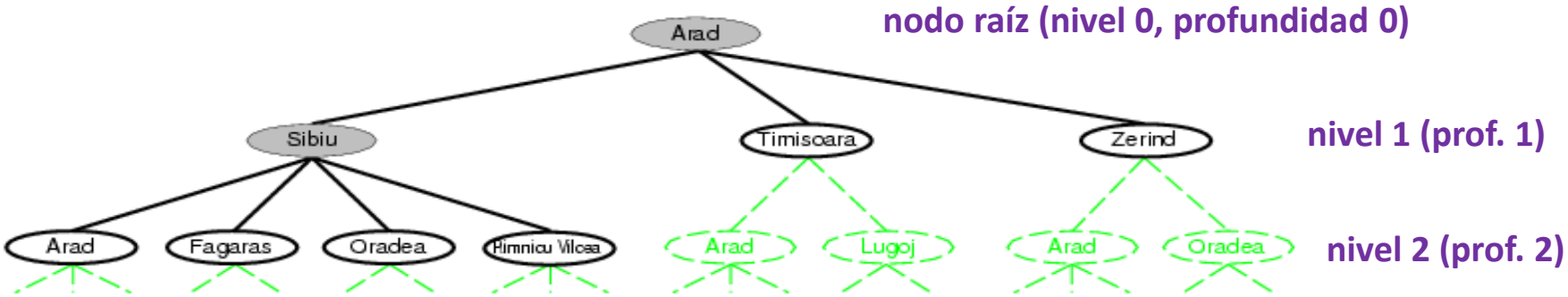


nodo raíz: nivel 0 (profundidad 0)

nivel 1 (prof. 1)

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

2. Búsqueda de soluciones: búsqueda en árbol

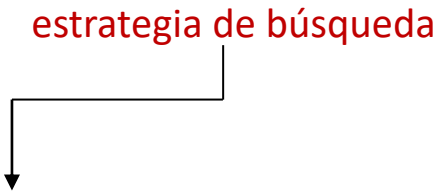


OPEN = {Timisoara Zerind Arad Fagaras Oradea Rimnicu Vilcea}

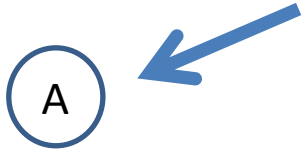
2. Búsqueda de soluciones: algoritmo TREE-SEARCH

- Un árbol de búsqueda (tree-search) es el conjunto de nodos que se genera durante la resolución de un problema mediante búsqueda: árbol de búsquda = espacio de búsqueda.
- Los algoritmos de búsqueda comparten la estructura básica vista anteriormente en el proceso general de búsqueda; se diferencian, básicamente, en como seleccionan el siguiente nodo que se va 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



Traza del algoritmo TREE-SEARCH



Estado inicial = nodo raíz
OPEN ={A}

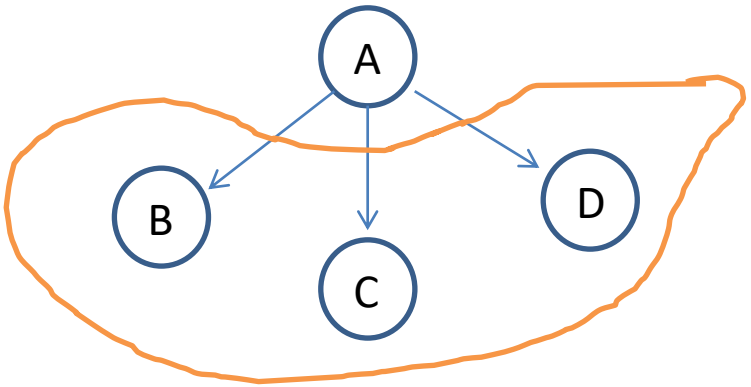
Seleccionamos primer nodo de lista OPEN y lo eliminamos de la lista: **A**

Expandimos el nodo seleccionado

1) Comprobamos si **A** es el objetivo

2) sino, generamos los hijos de **A** (reglas o acciones aplicables en el nodo A)

Traza del algoritmo TREE-SEARCH



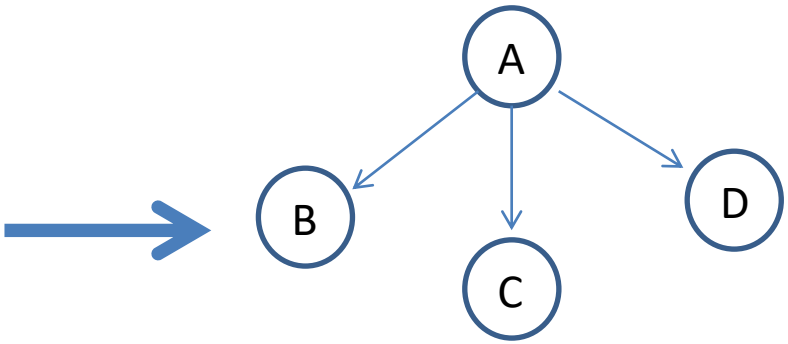
nodos hoja

La lista **OPEN** contiene los nodos hojas de un árbol

OPEN list = {B, C, D}

Los nodos se insertan en OPEN en el orden que determine la estrategia de búsqueda: anchura, profundidad, heurística …

Traza del algoritmo TREE-SEARCH



OPEN list = {B, C, D}

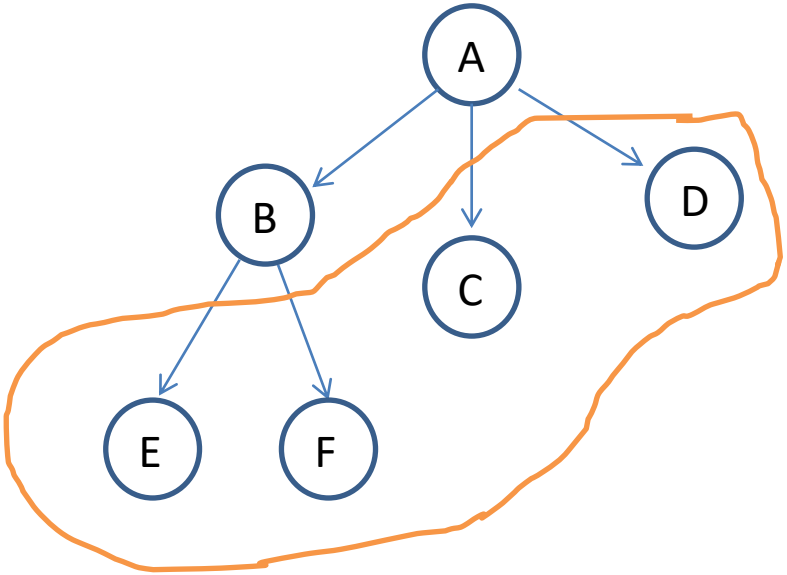
seleccionamos el primer nodo de la lista OPEN y lo eliminamos de la lista: **B**

expandimos el nodo **B**

1) comprobamos si **B** es el estado objetivo

2) sino, generamos los hijos de **B** (reglas o acciones aplicables en el nodo B)

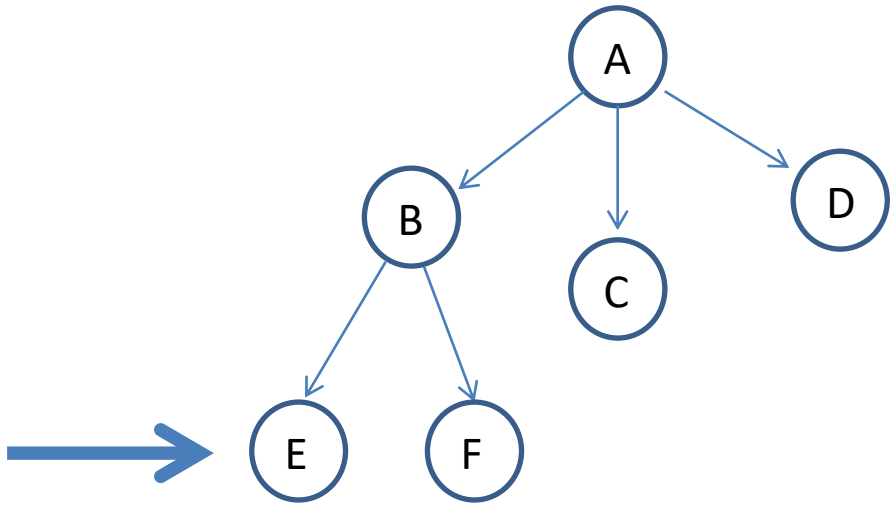
Generación de un árbol de búsqueda



OPEN list = {E, F, C, D}

estoy usando estrategia en profundidad (LIFO)

Traza del algoritmo TREE-SEARCH



OPEN list = {E, F, C, D}

Seleccionamos el primer nodo de OPEN y lo eliminamos: **E**

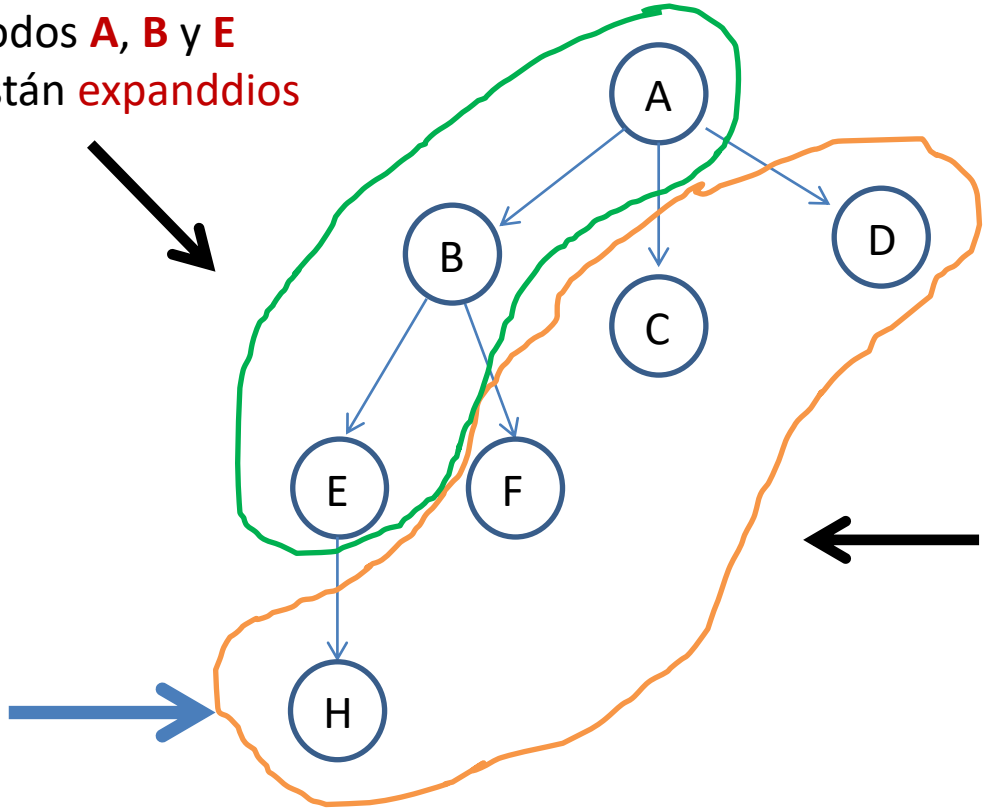
expandimos el nodo **E**

1) comprobamos si **E** es el estado objetivo

2) sino, generamos los hijos de **E** (reglas o acciones aplicables en el nodo E)

Traza del algoritmo TREE-SEARCH

Los nodos **A**, **B** y **E**
ya están expanddios



OPEN list = {H, F, C, D}

Y los nodos **H**, **F, C** y **D**
están generados
pero no expandidos

Seleccionamos el primer nodo de la lista OPEN y lo eliminamos: **H**

Y así seguimos iterando ….

2. Búsqueda de soluciones: estados repetidos

Espacio de búsqueda ≠ Espacio de estados : el árbol de búsqueda puede contener estados repetidos y ciclos; por ejemplo, el espacio de estados del problema 'Rumanía' solo tiene 20 estados o ciudades mientras que el árbol de búsqueda contendrá estados repetidos y caminos redundantes (árbol completo de búsqueda es infinito).

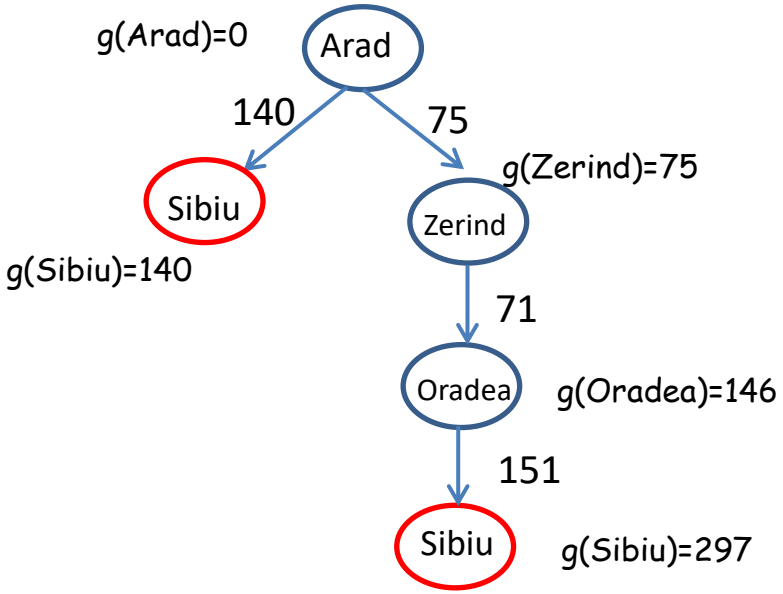
Estados repetidos:

- Acciones reversibles: ciclos

nodo repetido:
antecesor directo

nodo repetido: antecesor en el camino

caminos redundantes: más de un camino entre dos estados



2. Búsqueda de soluciones: estados repetidos

Dado que la resolución de un problema consiste en alcanzar el objetivo, no hay razón para mantener más de un camino hacia un estado dado porque cualquier objetivo que sea alcanzable extendiendo un camino también lo será extendiendo al otro.

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.

La existencia de nodos repetidos y caminos redundantes afecta exclusivamente a la eficiencia del algoritmo.

2. Búsqueda de soluciones: algoritmo GRAPH-SEARCH

La lista OPEN se implementa como una cola de prioridades (priority queue):

- los elementos se ordenan siempre en la cola según una función de ordenación (en orden creciente, minimización de la función)
- los nodos con menor valor se sitúan al principio de la cola; los nodos con mayor valor al final de la cola
- siempre se extrae el elemento de la cola con mayor prioridad, es decir, el elemento con menor valor de la función de ordenación que será el primer elemento de la cola

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 (orden creciente de $f(n)$).

2. Búsqueda de soluciones: algoritmo GRAPH-SEARCH

function GRAPH-SEARCH (*problema*) **return** una solución o 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 es un nodo 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 menor que el valor del nodo repetido en CLOSED
                then escoger entre re-expandir n (insertarlo en OPEN y eliminarlo de CLOSED) o descartar n
    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)
- solo puede evitar estados repetidos en la lista OPEN
- re-expande nodos ya explorados (consumo de tiempo y memoria repitiendo la misma búsqueda)

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: ¿se garantiza que la estrategia encuentra una solución para el problema cuando la hay?
2. Complejidad temporal: ¿cuánto tiempo necesita la estrategia para encontrar una solución?
3. Complejidad espacial: ¿cuánta memoria necesita la estrategia para realizar la búsqueda?
4. Optimalidad: ¿devuelve la estrategia la solución de mayor calidad, esto es, el camino solución de menor coste?

Soluciones que representen un balance entre:

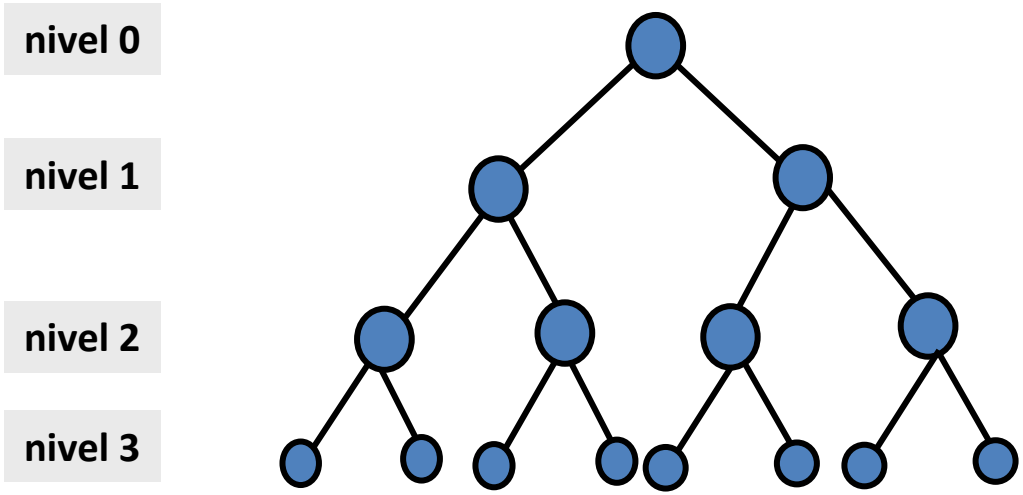
- coste del camino solución: suma del coste de las acciones individuales del camino solución (*g*(estado_final)) (camino óptimo, cercano al óptimo, lejos del óptimo, …)
- coste de la búsqueda: coste de encontrar el camino solución

Tipos de estrategias de búsqueda:

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

3. Anchura

Expande el nodo menos profundo entre los nodos no expandidos de la lista OPEN:

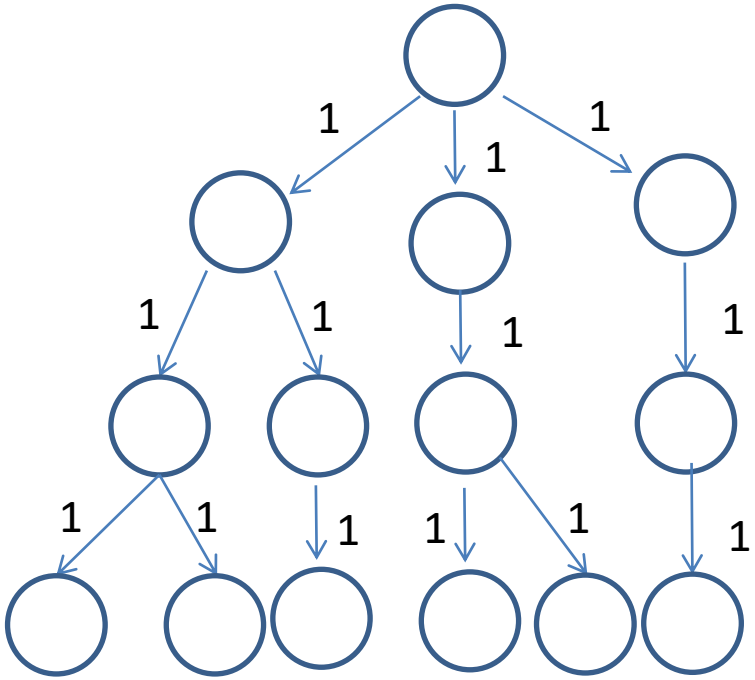


Función de evaluación (cola de prioridades): ¿qué función de ordenación/evaluación puede utilizarse para insertar los nodos menos profundos al comienzo de la cola?

f(n)=nivel(n)=profundidad(n)

3. Anchura

Si todos los operadores/acciones del problema tienen el mismo coste entonces para implementar **Anchura** podemos usar $f(n)=g(n)$



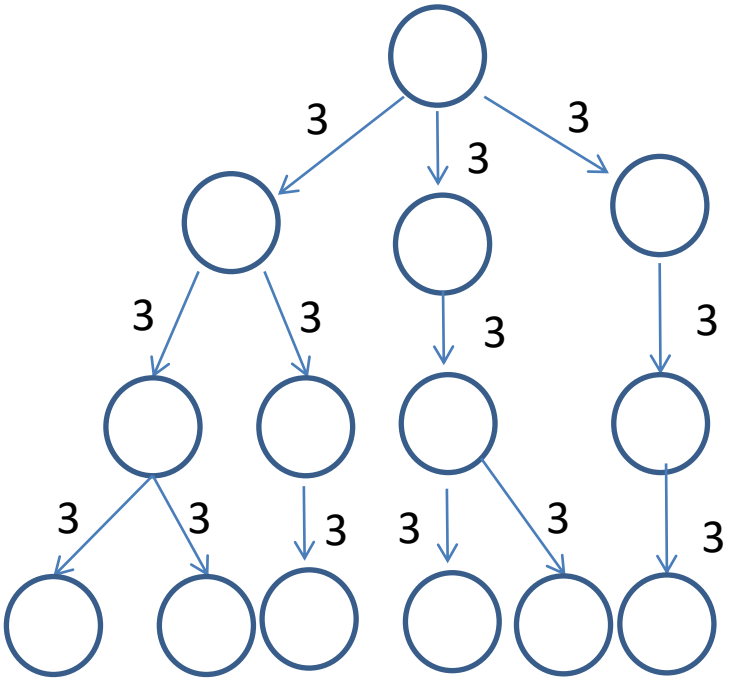
nivel= 1 f(n)=g(n)=1

nivel= 2 f(n)=g(n)=2

nivel= 3 f(n)=g(n)=3

3. Anchura

Si todos los operadores/acciones del problema tienen el mismo coste entonces para implementar anchura podemos usar $f(n)=g(n)$



nivel= 1 f(n)=g(n)=3

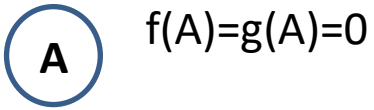
nivel= 2 f(n)=g(n)=6

nivel= 3 f(n)=g(n)=9

3. Anchura

f(n)=g(n)

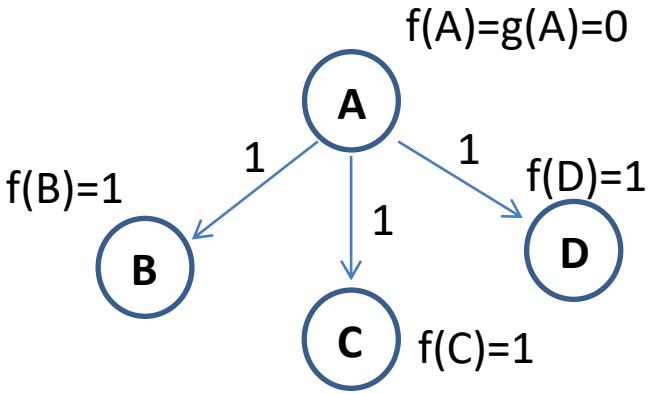
OPEN list = {A}



OPEN list = {B}

OPEN list = {B,C}

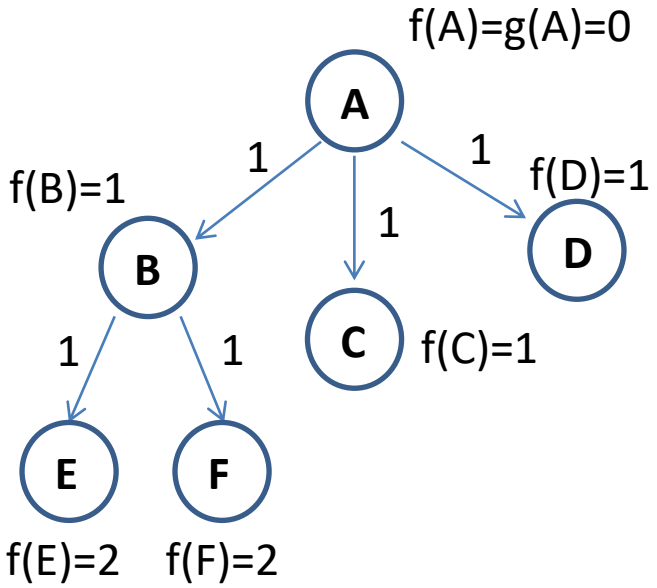
OPEN list = {B,C,D}



3. Anchura

OPEN list = {C,D,E}

OPEN list = {C,D,E,F}



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 $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d$ $\boldsymbol{O(b^d)}$

  nodos generados $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d + (b^{d+1} - b)$ $\boldsymbol{O(b^{d+1})}$

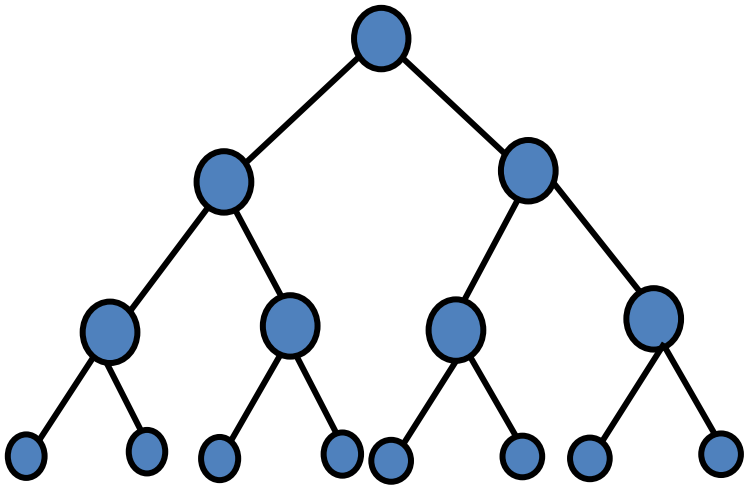
- Complejidad espacial para factor de ramificación b y profundidad d:
  - En el algoritmo GRAPH-SEARCH, todos los nodos residen en memoria
  - $\boldsymbol{O(b^d)}$ en la lista CLOSED y $\boldsymbol{O(b^{d+1})}$ en la lista OPEN (dominado por el tamaño de OPEN)

- Conclusiones:
  - Coste espacial más crítico que el coste temporal
  - Coste temporal inviable para valores altos de b y d

Número de nodos en un árbol de búsqueda

b=2

número de nodos en cada nivel



nivel 0	d=0	$b^d=2^0=1$ nodo
nivel 1	d=1	$b^d=2^1=2$ nodos
nivel 2	d=2	$b^d=2^2=4$ nodos
nivel 3	d=3	$b^d=2^3=8$ nodos

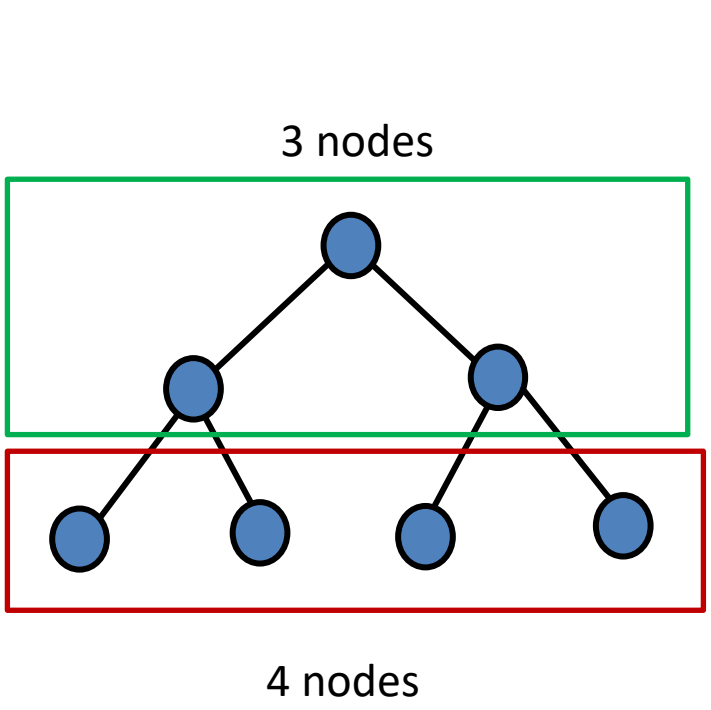
Número de nodos en un árbol de búsqueda

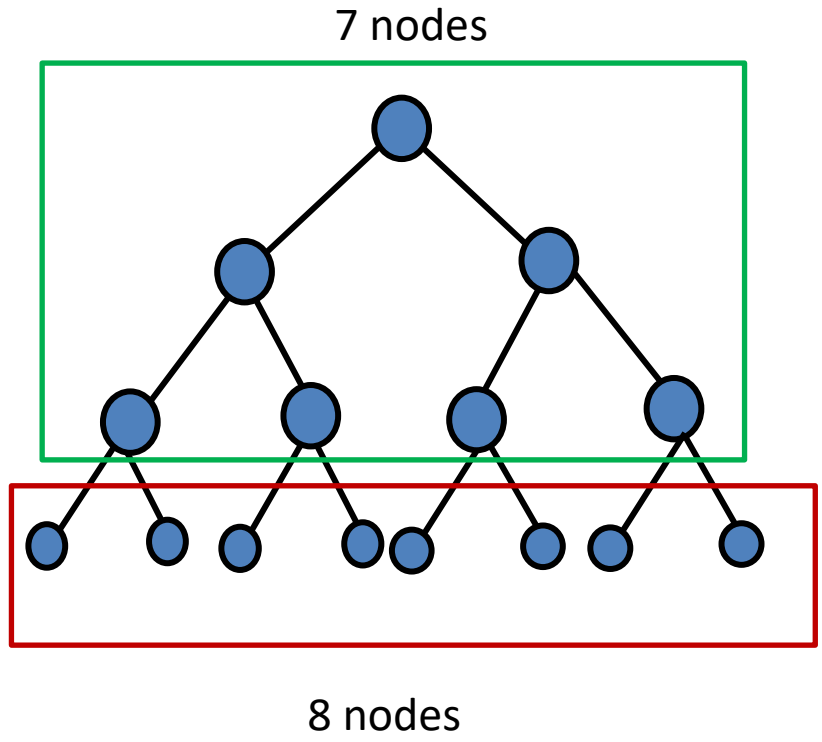
El número total de nodos de un árbol con factor de ramificación **b** y nivel de profundidad d es:

$$1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d$$

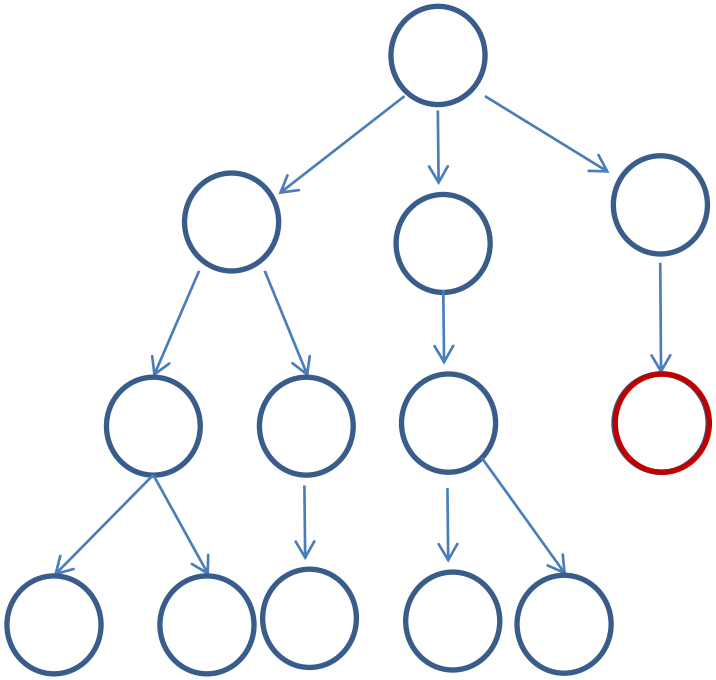
En un árbol de búsqueda con el mismo (o casi el mismo) **b** en cada nivel, la mayoría de nodos están en el nivel inferior.

El nivel inferior contiene tantos nodos como el resto del árbol.





ANCHURA



f(n)=nivel(n)

solución en nivel 2

Hemos generado ya
nodos a nivel 3

3. Anchura

Requerimientos de tiempo y memoria para anchura

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

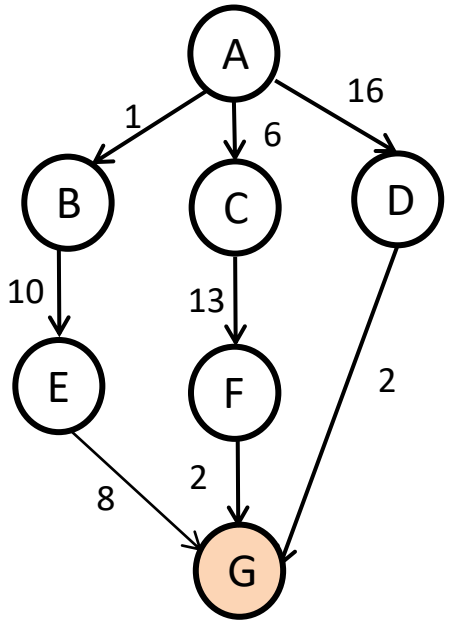
(Datos tomados del libro 3rd edition of the *Artificial Intelligence. A modern approach*)

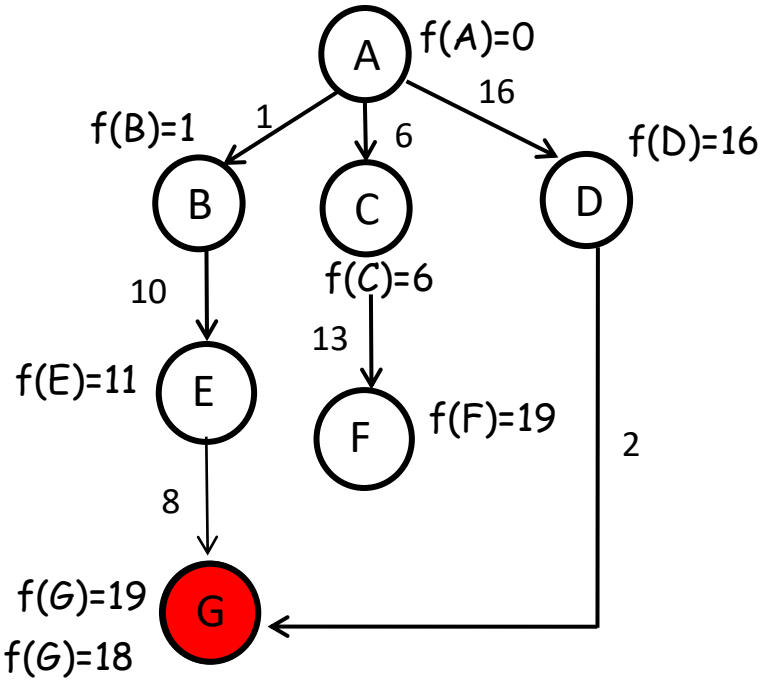
Profundidad	Nodos	Tiempo	Memoria
2	110	1.1 ms	107 kilobytes
4	11110	111 ms	10.6 megabytes
6	10^6	11 s.	1 gigabytes
8	10^8	19 min.	103 gigabytes
10	10^{10}	31 horas	10 terabytes
12	10^{12}	129 días	1 petabytes
14	10^{14}	35 años	99 petabytes
16	10^{16}	3500 años	10 exabytes

4. Coste uniforme

Expande el nodo con el menor coste entre los nodos no expandidos de OPEN (menor valor de $g(n)$)

Función de evaluación para ordenar elementos en lista OPEN: $f(n)=g(n)$





4. Coste uniforme

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

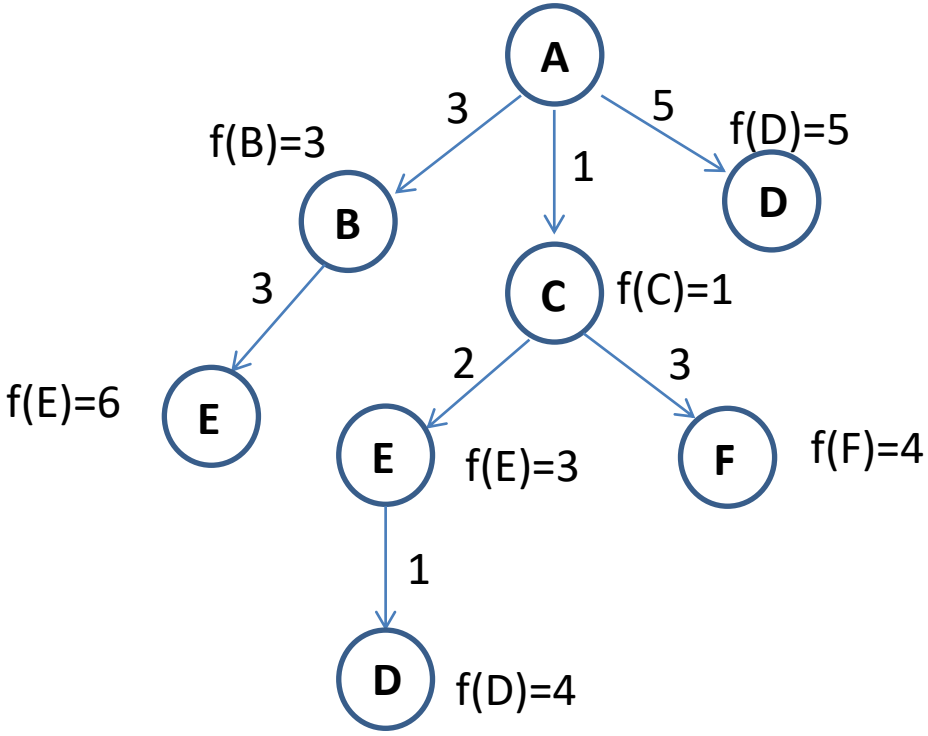
- Óptima :
  - si los costes de las acciones son no negativos $g(sucesor(n)) > g(n)$
  - coste uniforme expande nodos en orden creciente de coste

- 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: $O(b^{C^*/\varepsilon})$

4. Coste uniforme

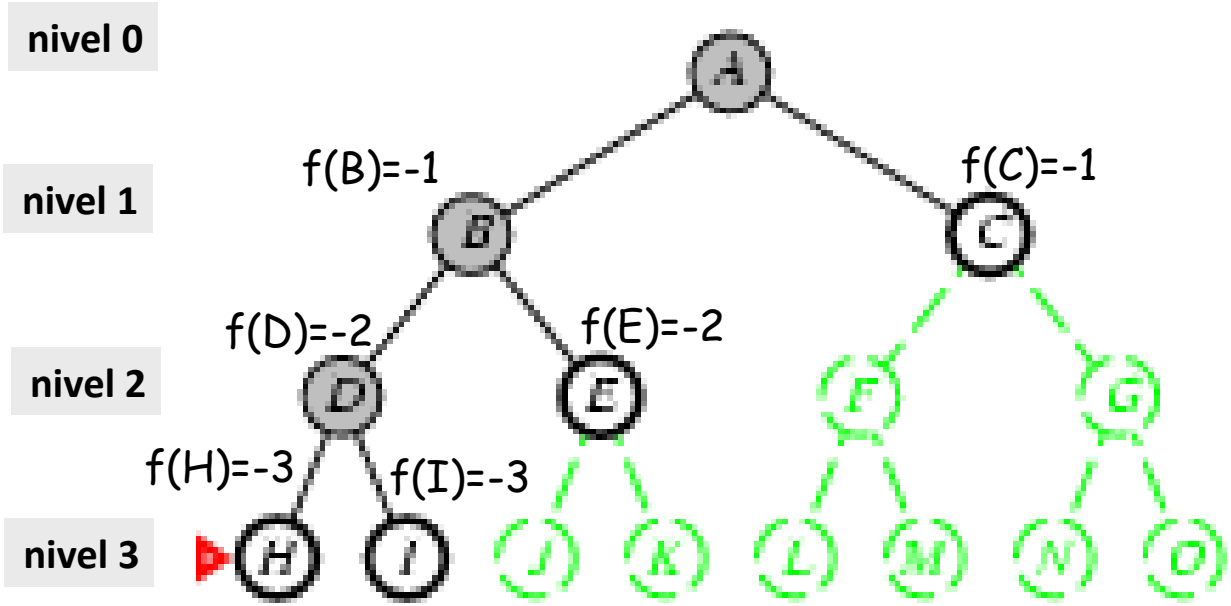
Resolver con el algoritmo de GRAPH SEARCH aplicando f(n)=g(n)



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

Expande el nodo más profundo entre los no expandidos de OPEN.

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



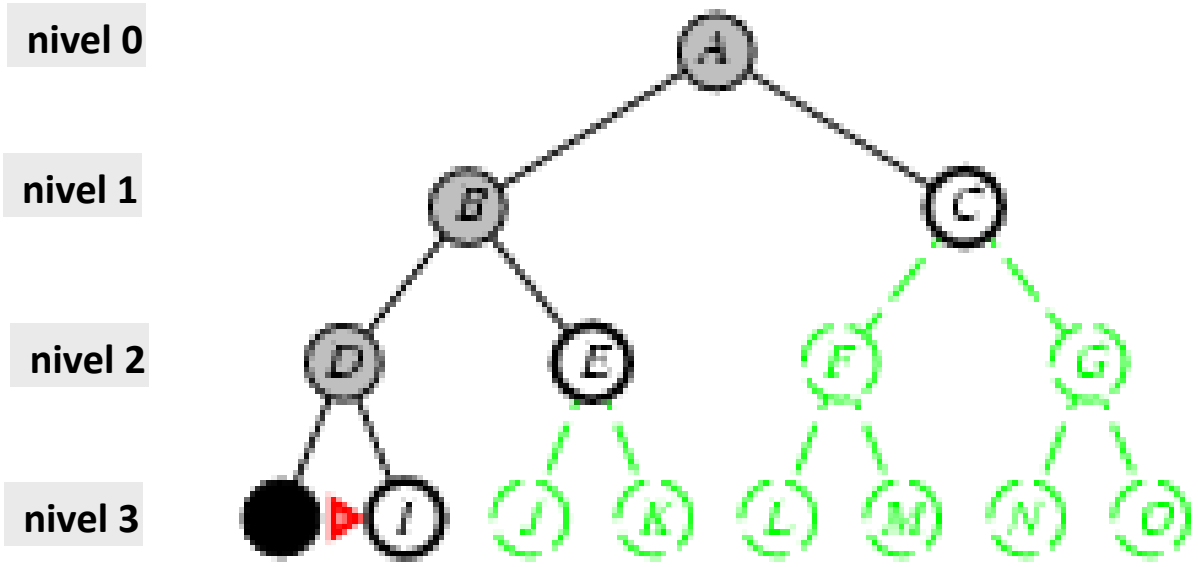
Backtracking :

1. nodo muerto, nodo no objetivo sin acciones aplicables (no hijos)
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

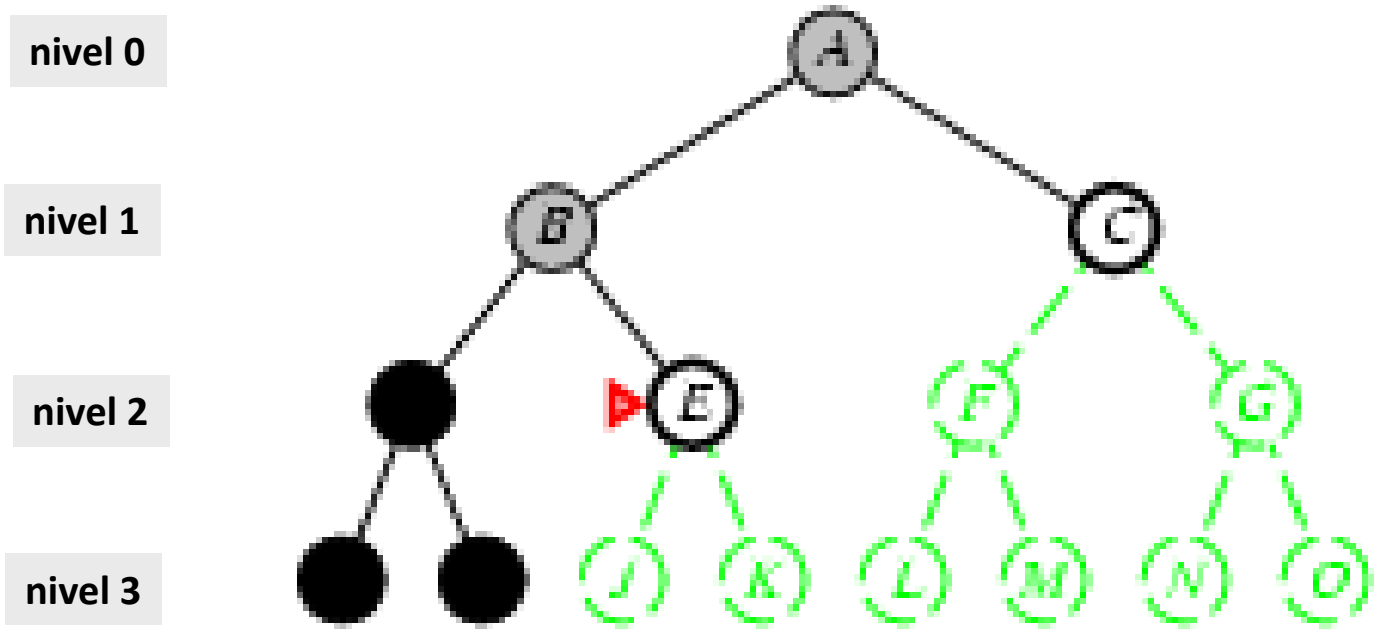


Eliminar H de la lista OPEN:
1. poner H en la lista PATH
2. comprobar si H es objetivo: NO
3. comprobar si H está en el máximo nivel de profundidad (m=3): SI => BACKTRACKING (H)

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

lista PATH = {A,B,D}

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



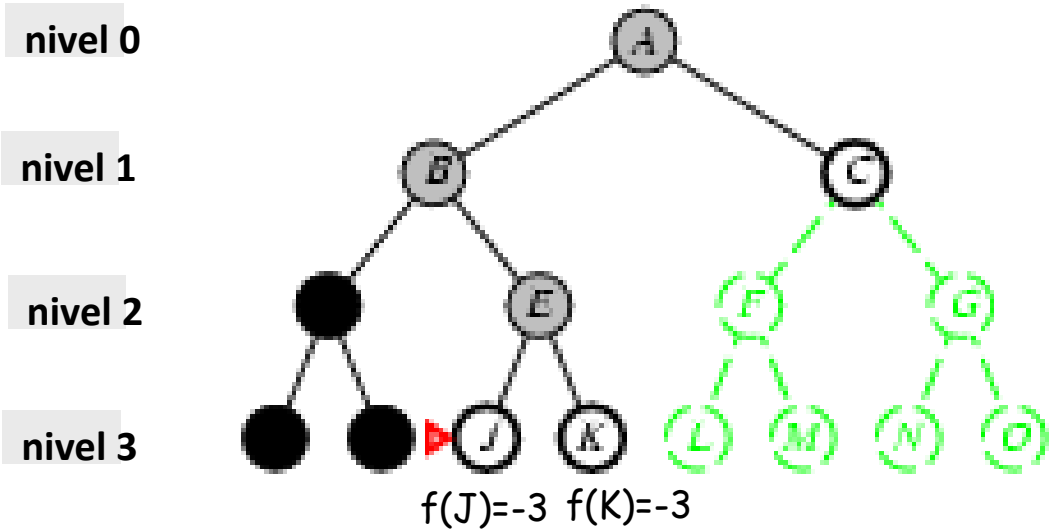
Eliminar I de la lista OPEN:

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)}

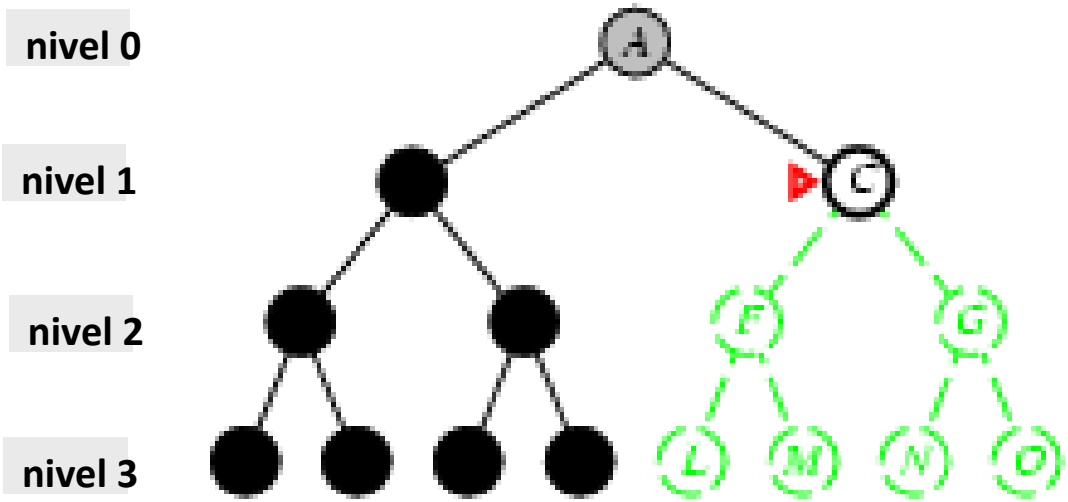
lista PATH= {A,B}

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**, $O(b \cdot m)$**. Complejidad espacial lineal!**
  - Listas OPEN y PATH contienen muy pocos nodos: manejo eficiente de las listas
  - Listas OPEN y PATH contienen muy pocos nodos: apenas existe control de nodos repetidos. En la práctica, esto significa que profundidad genera el mismo nodo mútiples veces.
  - Aún así, la versión TREE-SEARCH de profundidad puede ser más rápida que anchura
  - La versión GRAPH-SEARCH de profundidad no ofrece ninguna ventaja sobre anchura

5. Profundidad: propiedades

- Completitud:
  - Si no hay máximo nivel de profundidad y no hay control de nodos repetidos => no es completa porque la búsqueda puede quedarse estancada en bucles infinitos
  - Si no hay máximo nivel de profundidad pero sí hay control de nodos repetidos se pueden evitar ciclos infinitos => es completa porque eventualmente encontrará la solución
  - Si hay máximo nivel de profundidad (**m**) se podría perder la solución si ésta no se encuentra en el espacio de búsqueda definido por **m** => no es completa

- No óptima

6. Profundización Iterativa (Iterative Deepening – ID)

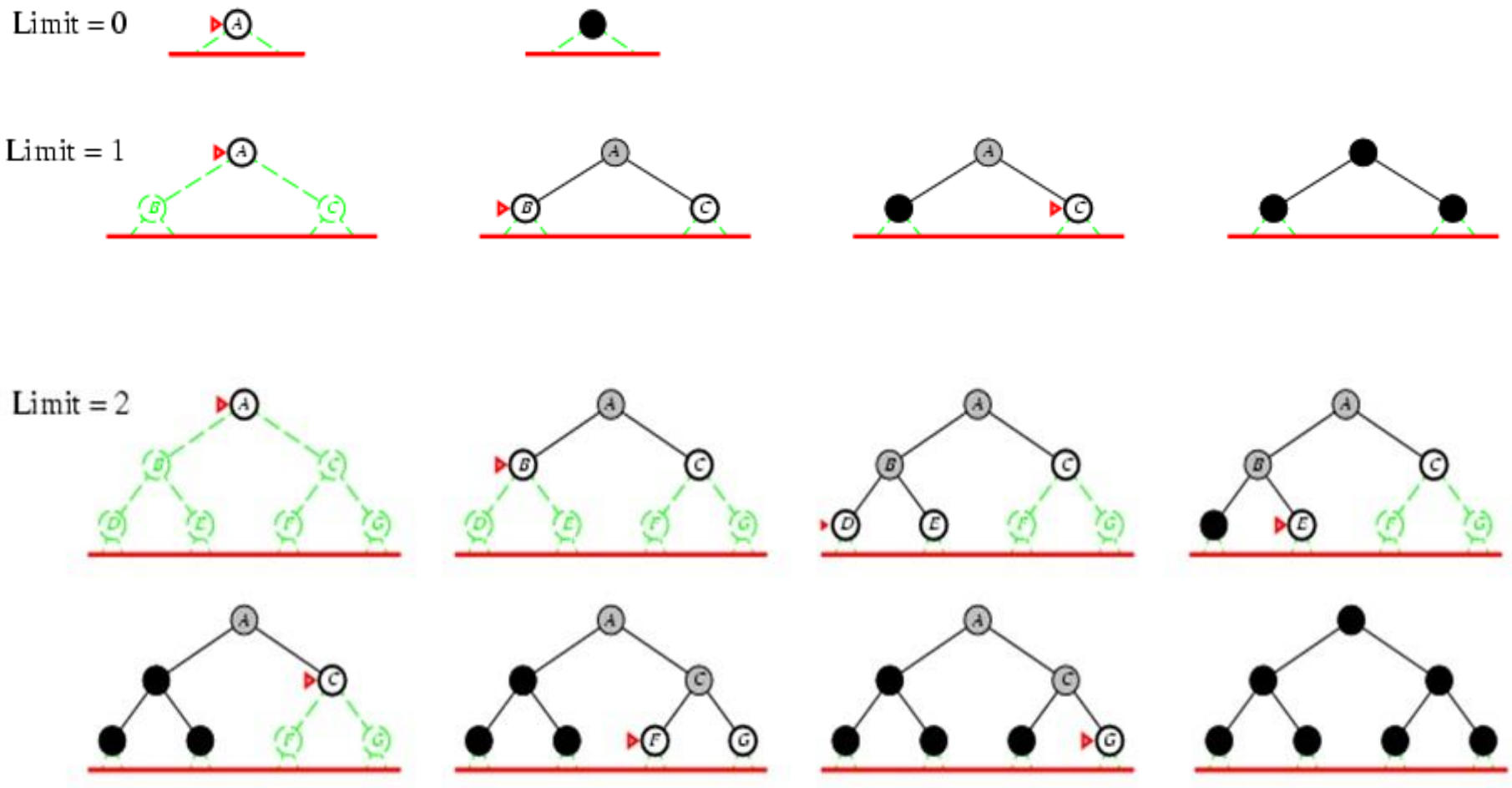
```
function Iterative_Deepening_Search (problem) returns (solution, failure)
    inputs: problem /*a problem*/
      for limit = 0 to ∞ do
          result = depth_limited_search (problem, limit)
          if result <> failure return result
      end
      return failure
end function
```

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

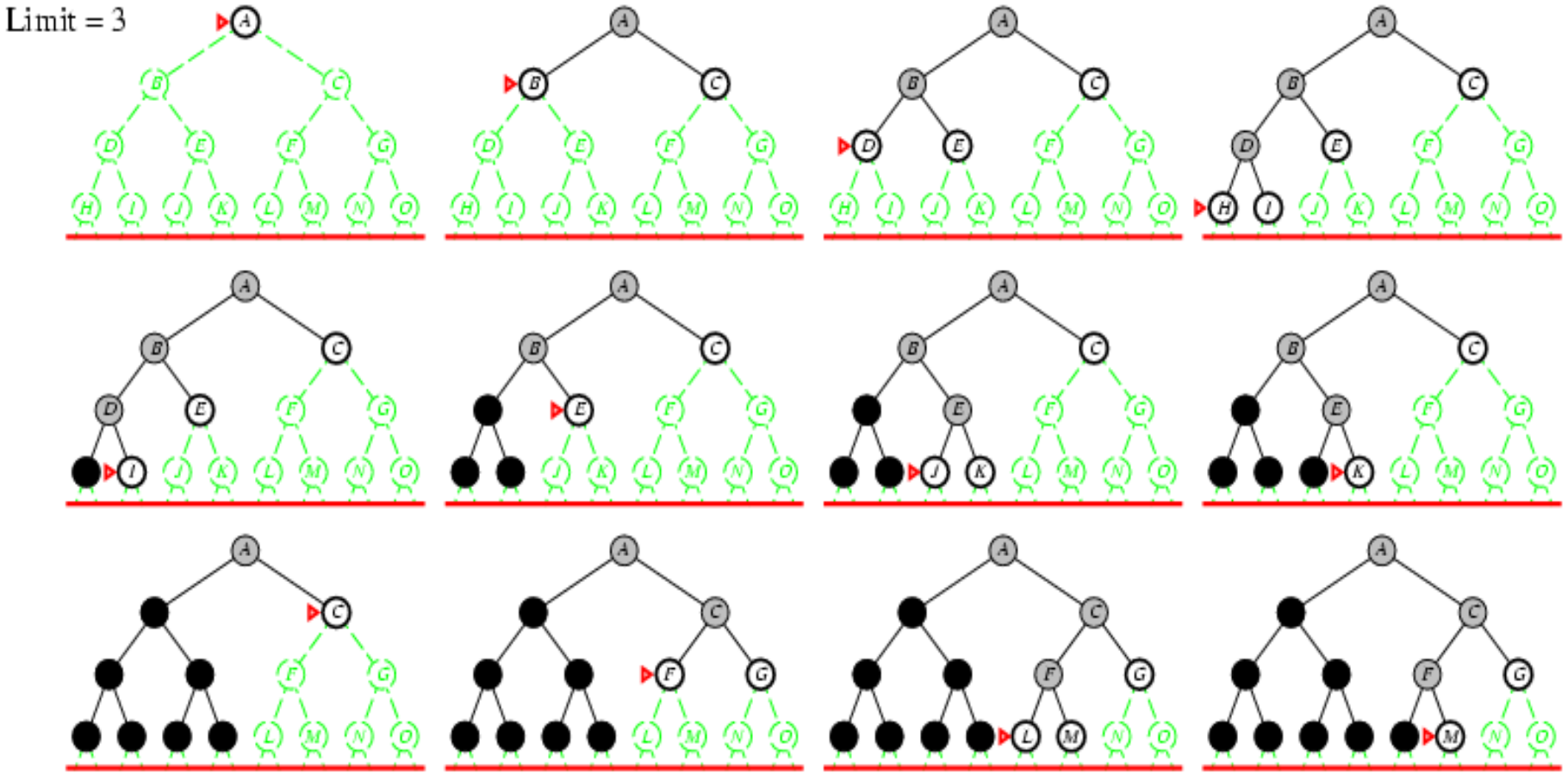
Realiza **iterativamente** una **búsqueda limitada en profundidad**, desde una profundidad-máxima 0 hasta ∞.

- 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** (bajo las mismas condiciones que anchura)
- Complejidad temporal $O(b^d)$, complejidad espacial $O(b \cdot d)$
- Método de búsqueda preferido cuando el espacio de búsqueda es muy grande y la profundidad de la solución no es conocida

6. Profundización iterativa (Iterative Deepening – ID)



6. Profundización iterativa (Iterative Deepening – ID)



6. Profundización iterativa (Iterative Deepening – ID)

- Número de nodos generados para b=10, d=5:
  - ID: $(d)\cdot b + (d-1)\cdot b^2 + (d-2)\cdot b^3 + \ldots\ldots\ldots + 1\cdot b^d$ 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

En ID, los nodos del nivel inferior (d) son generados una sola vez, los anteriores dos veces, etc. Los hijos de la raíz se generan d veces.

En anchura, cuando se selecciona un nodo a nivel d para ser expandido, el árbol puede contener nodos del siguiente nivel d+1; sin embargo, esto no ocurre en ID. ID puede ser más rápida que Anchura a pesar de la generación repetida de estados por los altos costes de memoria de Anchura y dificultad de acceso a los nodos almacenados.

ID 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.

ID 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 f(n)=nivel(n)	Coste uniforme f(n)=g(n)	Prof. Tree-Search f(n)=-nivel(n)	Prof. Iterativa
Temporal	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^d)$
Espacial	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b \cdot m)$	$O(b \cdot d)$
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 son $\geq \varepsilon$ para un ε positivo