Búsqueda y resolución de problemas

Capítulo 3, Secciones 1–5

Contenido

- ♢ Agentes solucionadores de problemas
- ♢ Tipos de problemas
- ♢ Formulación de problemas
- ♢ Problemas de ejemplo
- ♢ Algoritmos básicos de búsqueda

Agentes solucionadores de problemas

Forma restringida de una agente universal:

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

Note que esta es una forma de resolución sin conexión con el mundo (offline).

Agentes solucionadores de problemas

La solución es ejecutada "con los ojos cerrados". El agente asume que ese es todo el conocimiento que se tiene (y se puede tener) del ambiente (conocimiento completo). Una suposición normalmente equivocada, pero suficiente en muchos casos.

El ejemplo de Rumanía

Estoy de vacaciones en Rumanía. En estos momentos estoy en Arad.
Mi avión de vuelta a Venezuela sale mañana de Bucarest.

Formule la meta:

Estar en Bucarest.

Formule el problema:

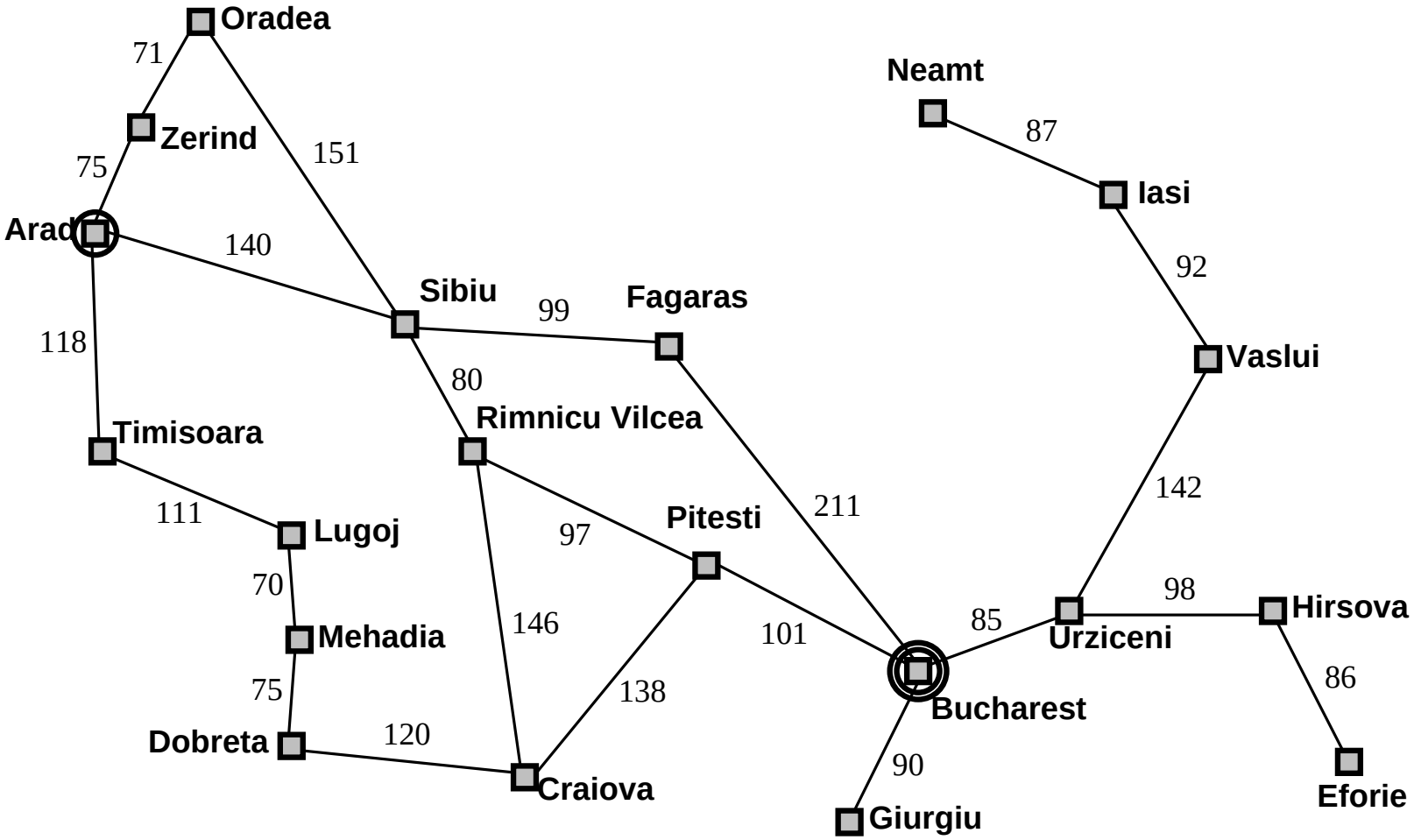
Estados: Las diferentes ciudades

Acciones: conducir mi auto entre las ciudades.

Encuentra la solución:

Una secuencia de ciudades, e.g. Arad, Sibiu, Fagaras, Bucarest.

El ejemplo de Rumanía



Tipos de problemas

Deterministico, completamente observable $\Longrightarrow$ *problema de un sólo estado*

El agente sabe exactamente a cual estado arribará. La solución es una secuencia de estados (o acciones).

No observable $\Longrightarrow$ *problema de múltiples estados*

El agente puede no tener idea de donde está. La solución, si existe, es una colección de estados posibles a donde arribaría. El agente si conoce el efecto de sus acciones. Piensen cuan exigente es esta última condición.

No-deterministico o parcialmente observable $\Longrightarrow$ *problem de contigencias*

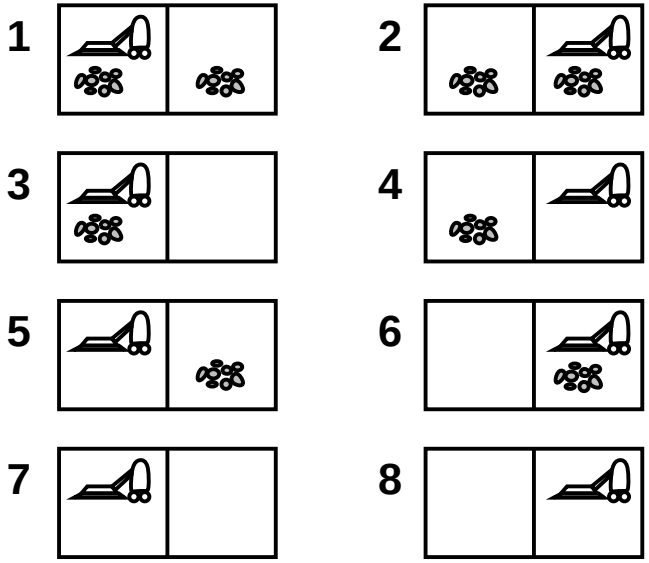
Los perceptos proveen *nueva* información sobre el estado actual

La solución es un *tree* or una *política* (varias ramas del árbol). Normalmente se solapan búsqueda y ejecución.

Espacio de estados desconocido $\Longrightarrow$ *problem de exploración* ("online")

El ejemplo del mundo de la aspiradora

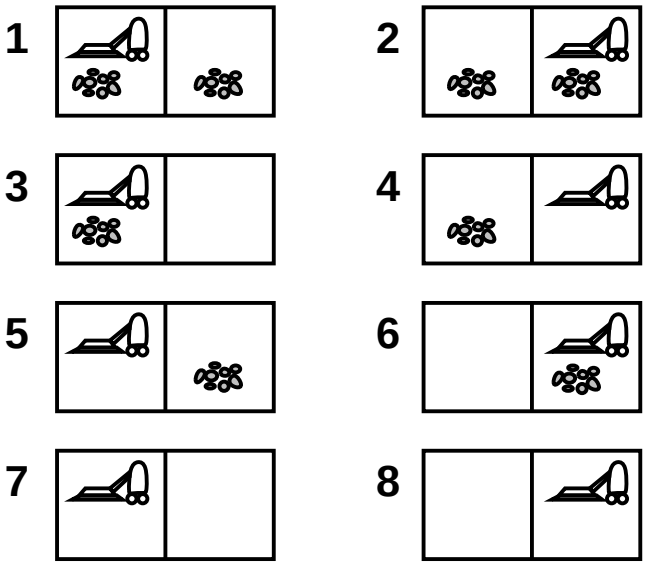
Estado simple, comienza en #5. Solución??



El ejemplo del mundo de la aspiradora

Estado simple, comienza en #5. Solución??
$[Right, Suck]$

Estado múltiple, comienza en $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ llega a $\{2, 4, 6, 8\}$. Solución??

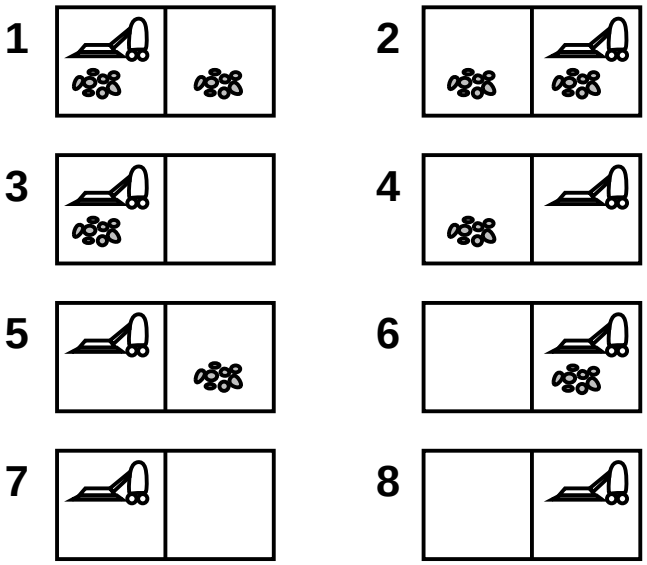


El ejemplo del mundo de la aspiradora

Estado simple, comienza en #5. Solución??
$[Right, Suck]$

Estado múltiple, comienza en $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ llega a $\{2, 4, 6, 8\}$. Solución??
$[Right, Suck, Left, Suck]$

Contingencia, comienza en #5
La ley de Murphy: $Suck$ puede ensuciar una alfombra limpia
Percepción local : sucio y ubicación solamente.
Solución??

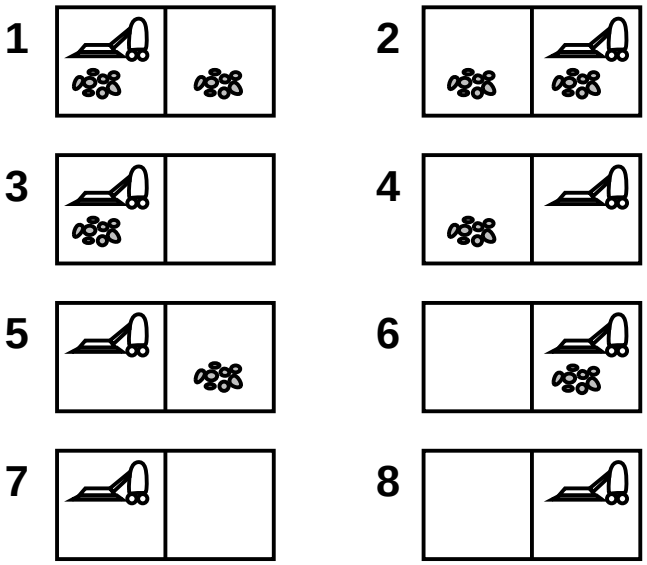


El ejemplo del mundo de la aspiradora

Estado simple, comienza en #5. Solución??
$[Right, Suck]$

Estado múltiple, comienza en $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ llega a $\{2, 4, 6, 8\}$. Solución??
$[Right, Suck, Left, Suck]$

Contingencia, comienza en #5
La ley de Murphy: $Suck$ puede ensuciar una alfombra limpia
Percepción local : sucio y ubicación solamente.
Solución??
$[Right, \textbf{si } dirt \textbf{ entonces } Suck]$



Formulación del problema de un solo estado

Un *problema* se define con los siguiente 4 items:

un estado inicial e.g., "en Arad"

una función succesor $S(x) =$ un conjunto de pares acción-estado asociados a un destino

e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

un test de meta, que puede ser *explícito*, e.g., $x =$ "at Bucharest"

o *implícito*, e.g., $NoDirt(x)$

función de costos (aditiva)

e.g., la suma de distancias, el número de acciones ejecutadas, etc.

$c(x, a, y)$ es el costo de paso, que se asume siempre ≥ 0

Una *solución* es una secuencia de acciones
que llevan al agente y a su ambiente desde el estado inicial hasta estado final

Seleccionando un espacio de estados

El mundo se presenta de muchas maneras y con muchos detalles.

Enfrentamos una enorme complejidad

$\Rightarrow$ el espacio de estados debe ser una *abstracción* de esa complejidad para poder emprender la resolución de los problemas.

Un estado (abstracto) en nuestro modelo = conjunto de estados reales.

Acción (abstracta) = agregado o combinación de acciones reales

e.g., "Arad $\rightarrow$ Zerind" representa un conjunto complejo de rutas posibles, desvios, paradas de descanso, etc.

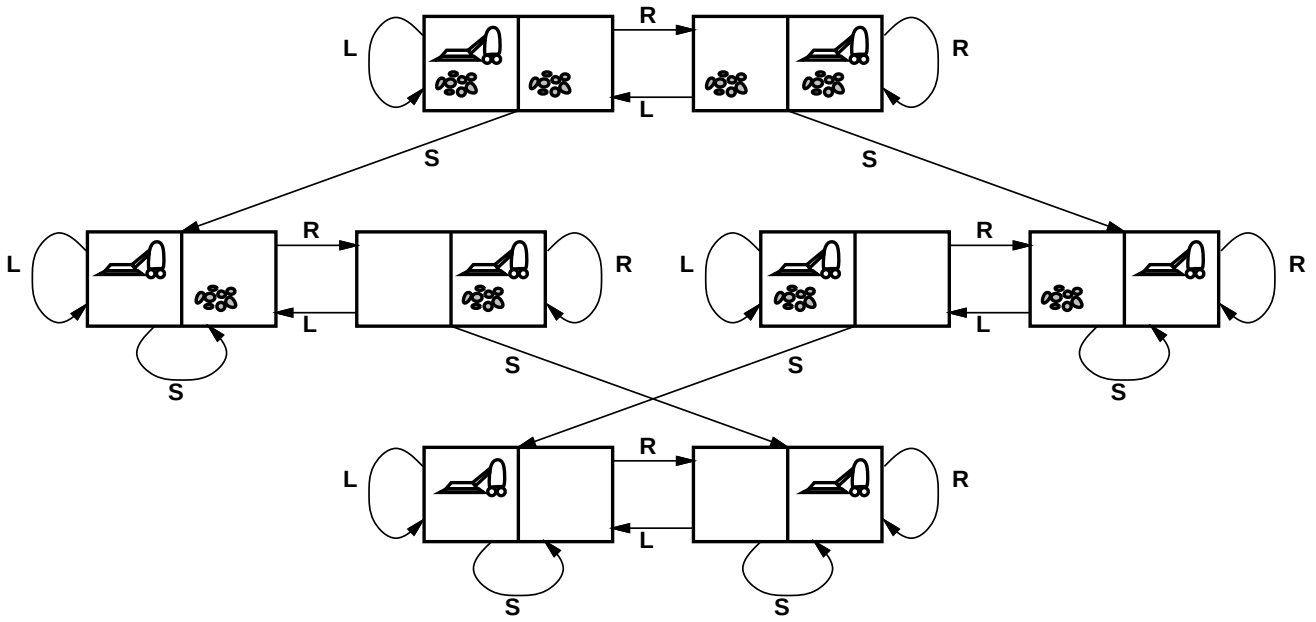
Para garantizar que tal acción se puede realizar, es preciso que cualquier estado real "en Arad" pueda conducir a *algún* estado real en "en Zerind"

Soluciones (abstractas) =

conjunto de caminos reales que son soluciones en el mundo real.

Cada una de las acciones abstractas debe ser "más fácil" que el problema original.

Ejemplo: el grafo del espacio de estados del mundo de la aspiradora



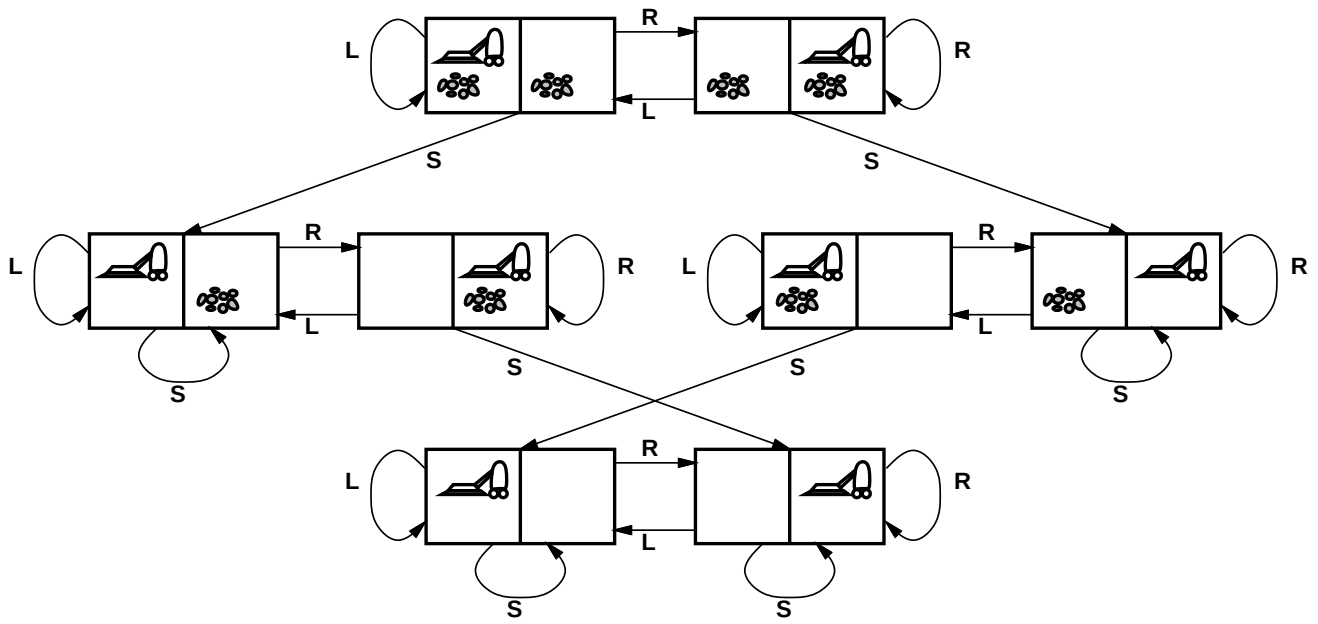
estados??

acciones??

test de meta??

costos??

Ejemplo: el grafo del espacio de estados del mundo de la aspiradora



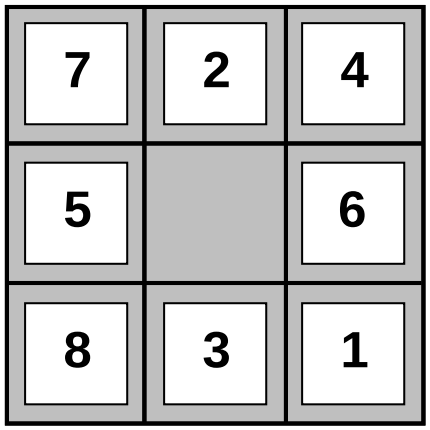
estados??: número entero para indicar sucio y ubicación del robot (pero ignora *cantidad* de sucio)

acciones??: $Left$, $Right$, $Suck$, $NoOp$

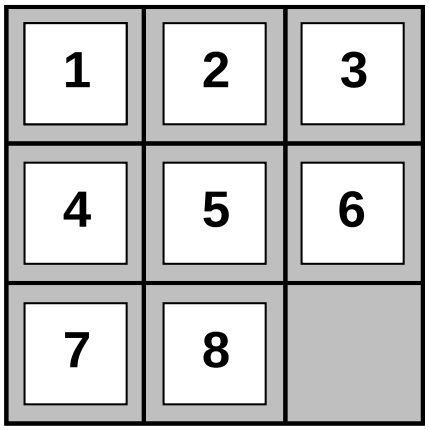
test de meta??: no hay sucio en ninguna ubicación

costo??: cuenta 1 por cada acción (0 por $NoOp$)

Example: The 8-puzzle



Start State



Goal State

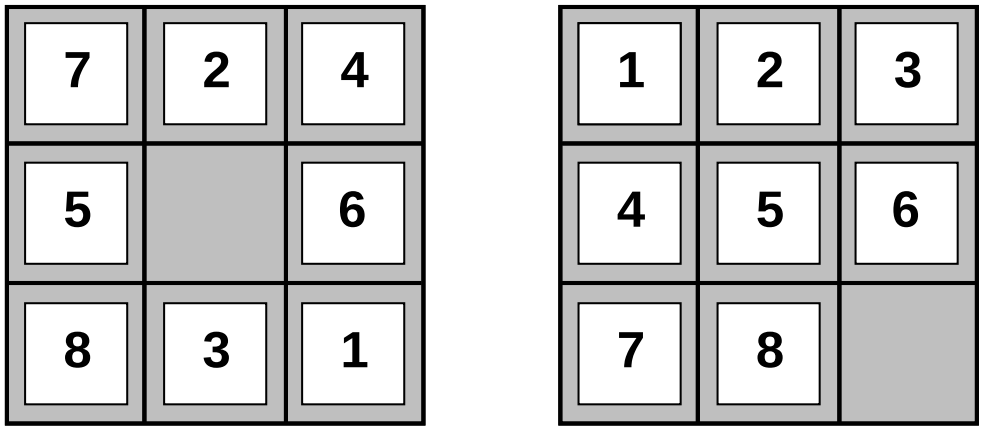
estados??

acciones??

test de meta??

costos??

Ejemplo: fichero de 8

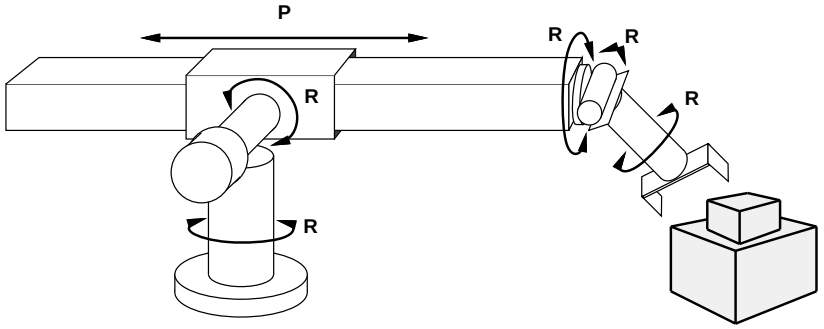


Start State **Goal State**

estados??: ubicación de las fichas en enteros (ignora posiciones intermedias)
acciones??: mover el vació izquierda, derecha, arriba, abajo (ignore destranque, etc.)
test de meta??: = cualquier configuración predefinida
costos??: 1 por movimiento.

[Nota: La solución óptima de problemas como el fichero de n es NP-Compleja]

Ejemplo: El robot de ensamblaje



estados??: las coordenadas (valores reales) de cada coyuntura del robot.
Las partes del objeto que serán ensambladas.

acciones??: movimientos continuos de las coyunturas del robot (más algunas acciones discretas para cambiar sus trayectorias).

test de meta??: ensamblaje completo.

costos??: tiempo de ejecución (energia empleada??)

Algoritmos de búsqueda en un árbol

Idea básica:

exploración simulada "offline", del espacio de estados

para lo cual se generan sucesores de los estados ya explorados

, una técnica tambíen conocida como *expandir* los estados.

function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure

initialize the search tree using the initial state of *problem*

loop do

if there are no candidates for expansion **then return** failure

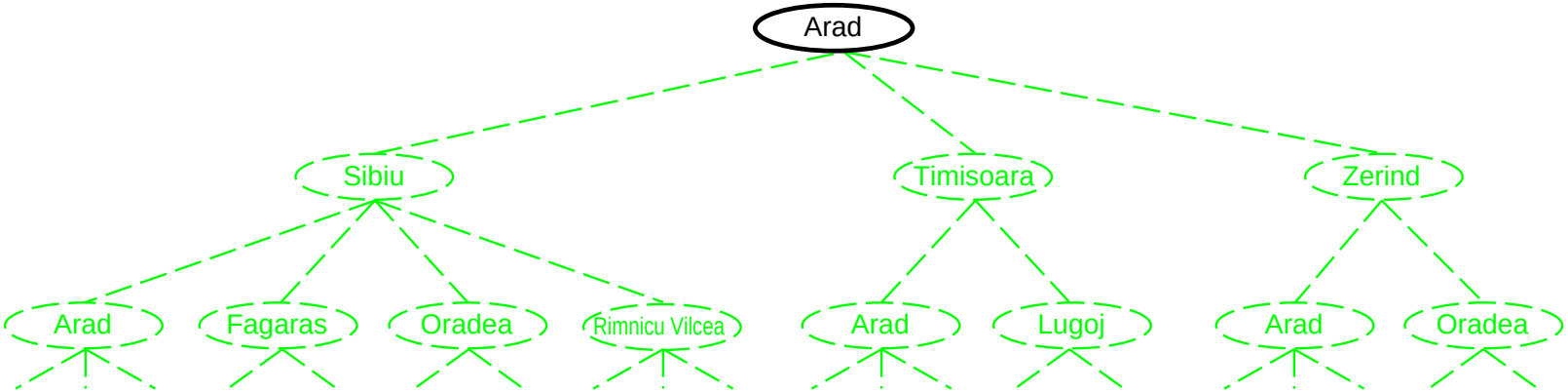
choose a leaf node for expansion according to *strategy*

if the node contains a goal state **then return** the corresponding solution

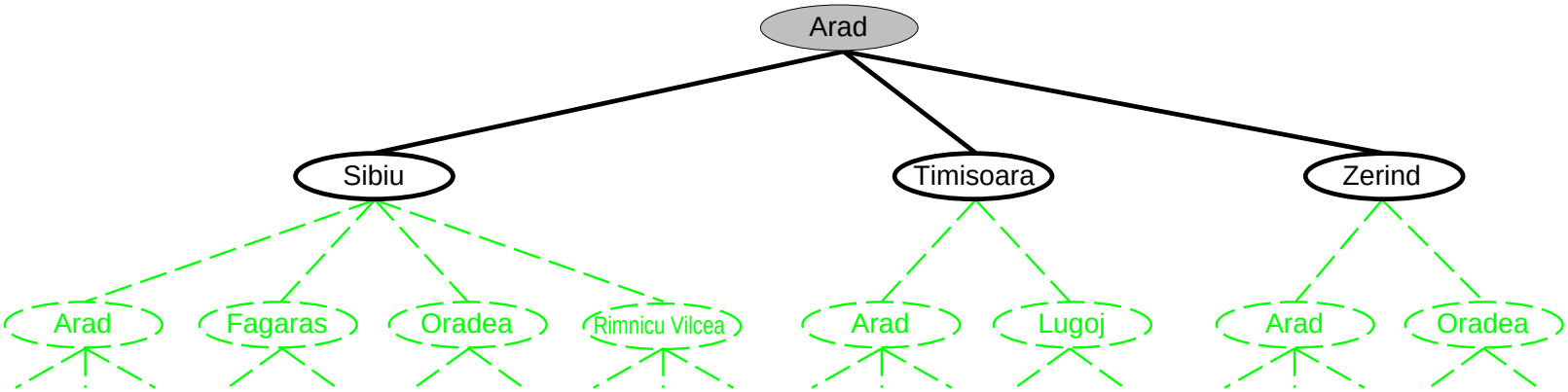
else expand the node and add the resulting nodes to the search tree

end

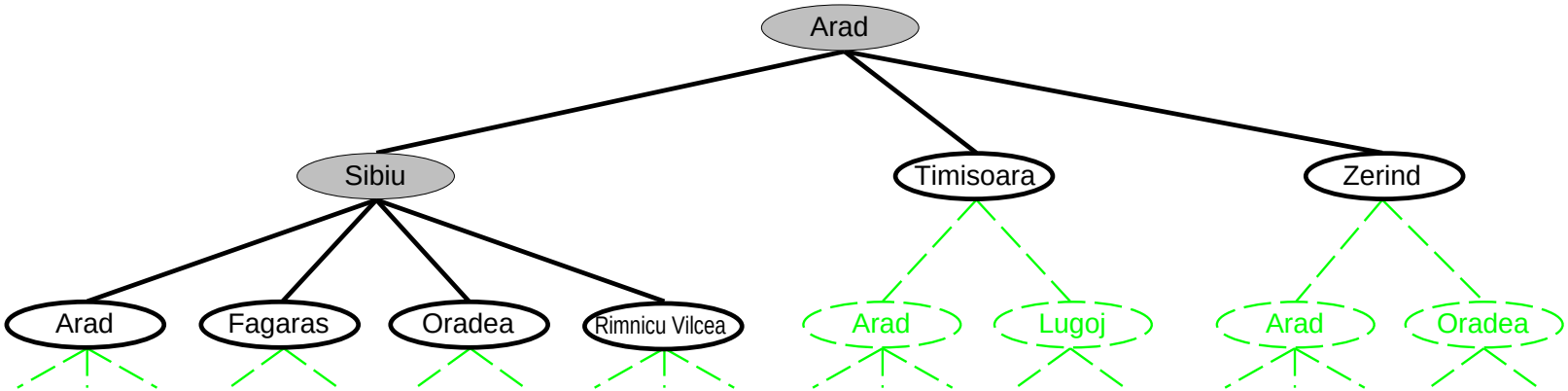
Ejemplo de búsqueda en un árbol



Ejemplo de búsqueda en un árbol



Ejemplo de búsqueda en un árbol

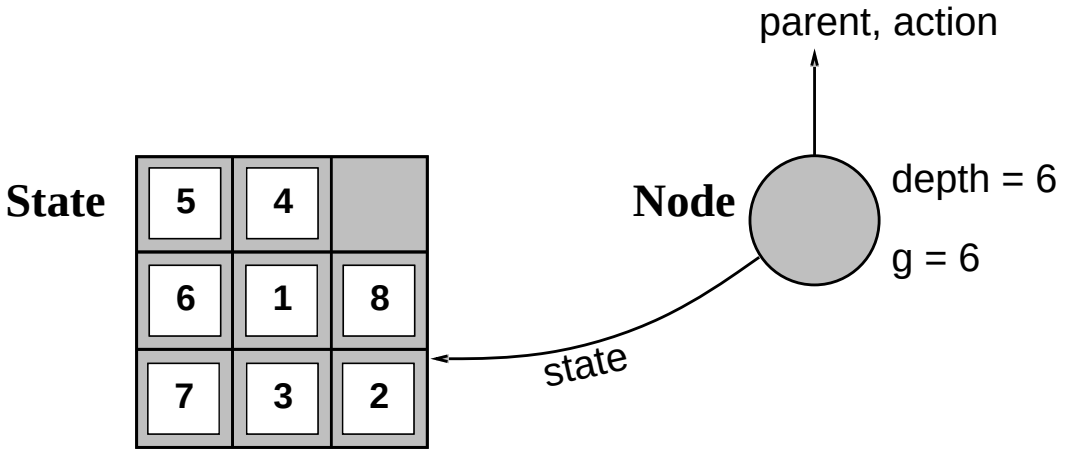


Implementación: estados vs.ñnodos

Un *estado* es una (representación de) una configuración física

Un *node* es una estructura de data parte constitutiva de un árbol de búsqueda que incluye *padres*, *hijos*, *profundidad* y *costo del camino* $g(x)$

Los *estados* no tienen padres, hijos, profundidad o costo del camino!.



La función EXPAND crea nuevos nodos, completando los diversos campos de la estructura y usando SUCCESORFN del problema para crear los estados correspondientes.

Implementación: búsqueda general en un árbol

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Estrategias de búsqueda

Cada estrategia se define en la forma de seleccionar *el orden el que se "expanden" los (caminos de) nodos*

Las estrategias se evaluan de acuerdo a los siguientes aspectos:

completitud— ¿Acaso consigue siempre una solución, si nuna existe?

complejidad en tiempo— Número de nodos generados/expandidos

complejidad en memoria— Máximo número de nodos en memoria

optimalidad—¿Consigue siempre una solución de mínimo costo?

Estrategias de búsqueda

Las complejidades de tiempo y espacio se miden en términos de:

b—factor de "enramajé' (ramificación) máximo del árbol de búsqueda

d—profundidad de la solución de menor costo

m—profundidad máxima del espacio de estados (que puede ser ∞)

Estrategias de búsqueda no informadas

Las estrategias *no informadas* sólo usan la información disponible en la definición del problema

Búsqueda primero en anchura/Breadth-first search

Búsqueda de costo uniforme/Uniform-cost search

Búsqueda primero en profundidad/Depth-first search

Búsqueda de profundidad limitada/Depth-limited search

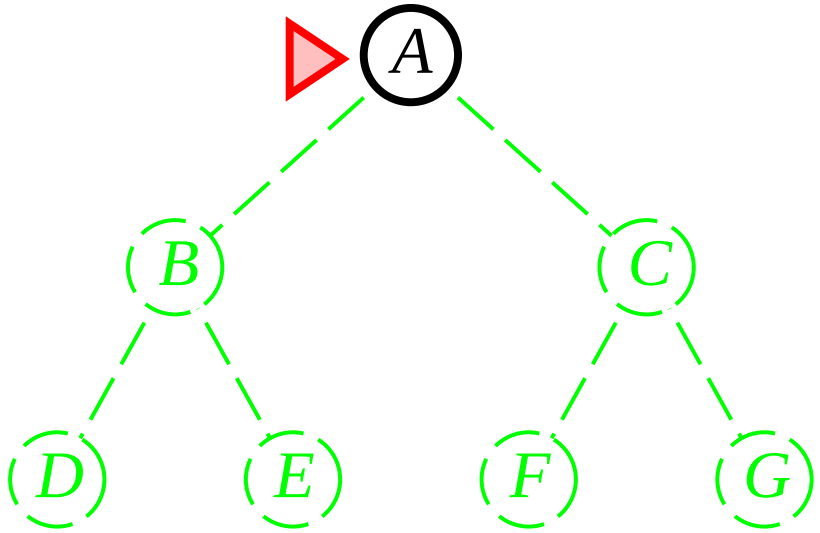
Búsqueda de profundizamiento iterativo/Iterative deepening search

Búsqueda primero en anchura

Expanda el nodo no expandido menos profundo.

Implementación:

fringe es una cola FIFO: los nuevos sucesores se agregan al final.

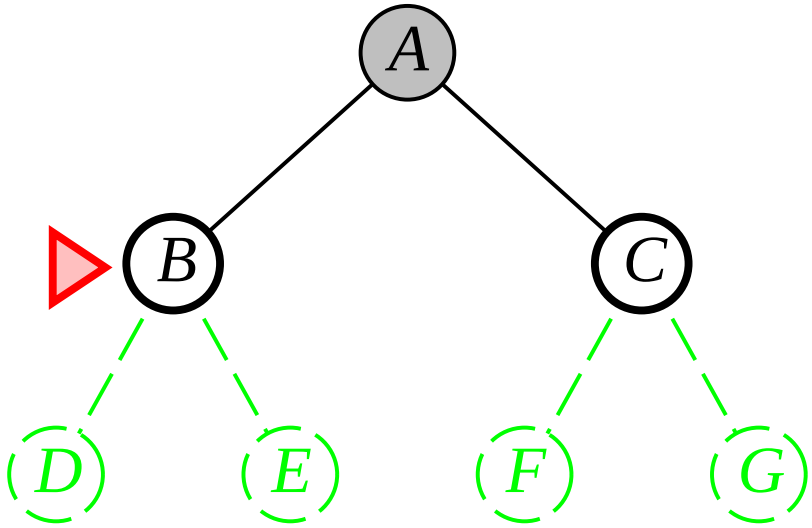


Búsqueda primero en anchura

Expanda el nodo no expandido menos profundo.

Implementación:

fringe es una cola FIFO: los nuevos sucesores se agregan al final.

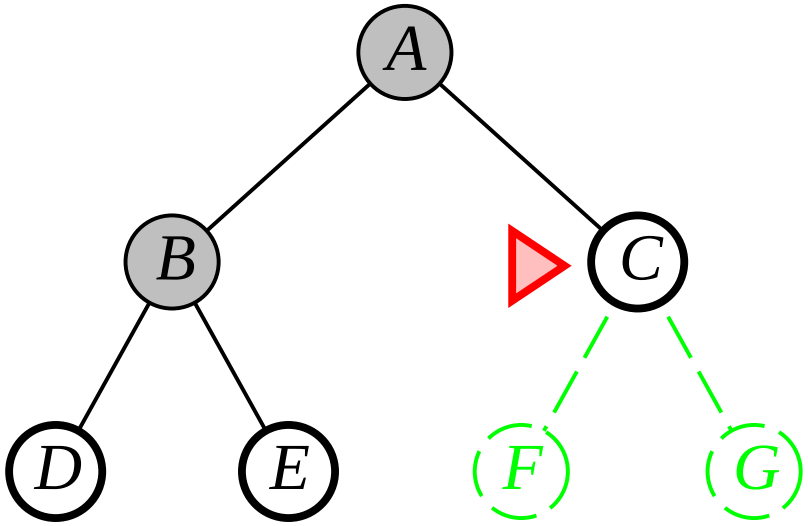


Búsqueda primero en anchura

Expanda el nodo no expandido menos profundo.

Implementación:

fringe es una cola FIFO: los nuevos sucesores se agregan al final.

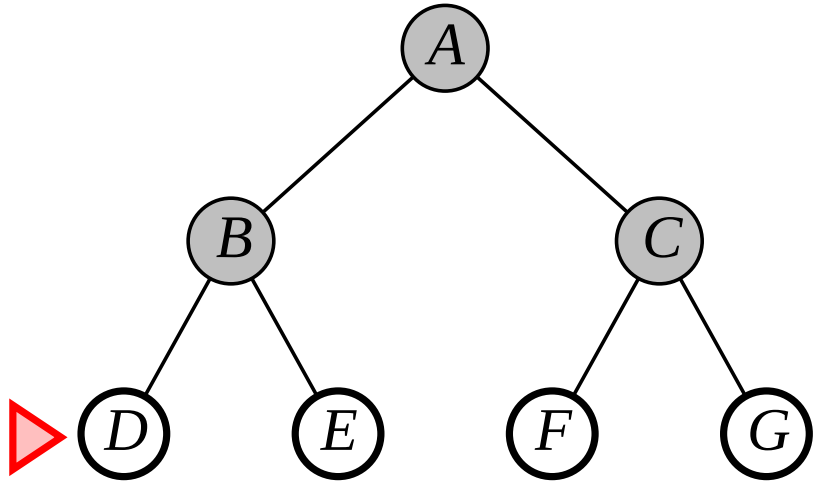


Búsqueda primero en anchura

Expanda el nodo no expandido menos profundo.

Implementación:

fringe es una cola FIFO: los nuevos sucesores se agregan al final.



Propiedades de la búsqueda primero en anchura

Completa??

Propiedades de la búsqueda primero en anchura

Completa?? Si (si b is finito)

Tiempo??

Propiedades de la búsqueda primero en anchura

Completa?? Si (si b is finito)

Tiempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, es decir, es exponencial.ñen d

Espacio??

Propiedades de la búsqueda primero en anchura

Completa?? Si (si b is finito)

Tiempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, es decir, es exponencial.ñen d

Espacio?? $O(b^{d+1})$ (Guarda todos los nodos en memoria)

Optimalidad??

Propiedades de la búsqueda primero en anchura

Completa?? Sí (si b is finito)

Tiempo?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, es decir, es exponencial.ñen d

Espacio?? $O(b^{d+1})$ (Guarda todos los nodos en memoria)

Optimalidad?? Sí (is el cost $= 1$ por paso); en general no es óptima.

El *espacio* es el gran problema aquí. La estrategia puede generar nodos a una taza de 10MB/seg

Es decir, en 24 horas son 860 GB.

Búsqueda de costo uniforme

Expande el nodo no expandido de menor costo.

Implementación:

$fringe =$ cola ordenada por el costo del camino.

Es equivalente a la búsqueda primero en anchura si todos los pasos son del mismo costo.

Completa?? Sí, si el costo del paso es $\geq \epsilon$

Tiempo?? $\#$ de nodos con $g \leq$ costo de la solución óptima, $O(b^{\lceil C^*/\epsilon \rceil})$
donde C^* es el costo de la solución óptima.

Espacio?? $\#$ de nodos con $g \leq$ costo de la solución óptima $O(b^{\lceil C^*/\epsilon \rceil})$

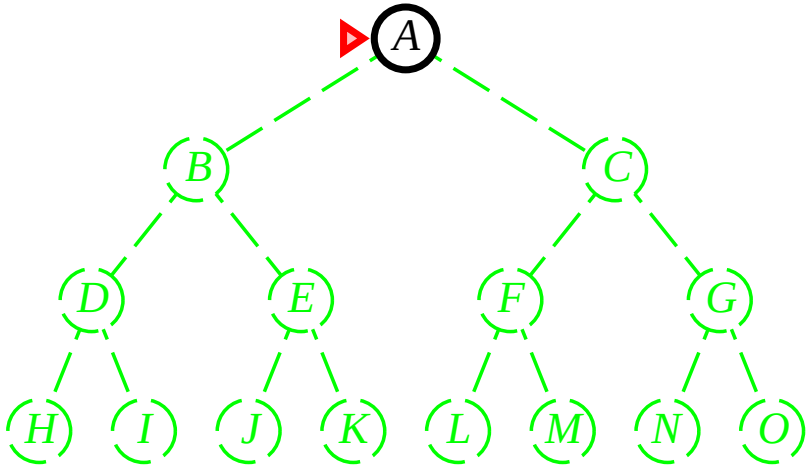
Optimalidad?? Sí—los nodos se expanden en el orden creciente de $g(n)$

Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

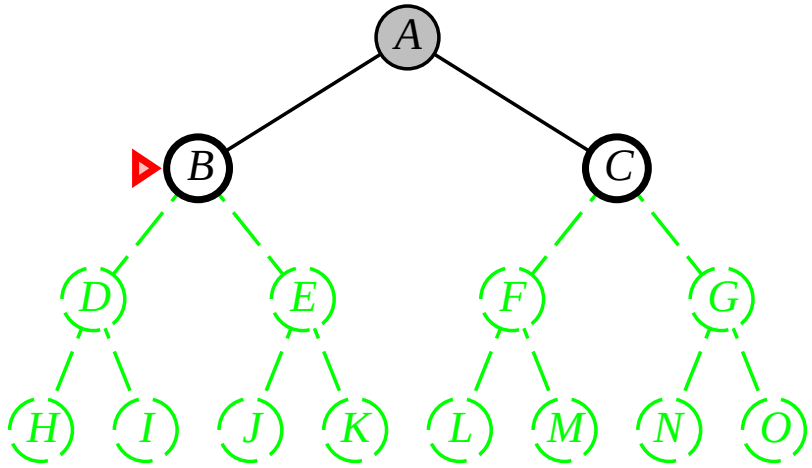


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe$ = cola LIFO , es decir, coloca los sucesores al frente

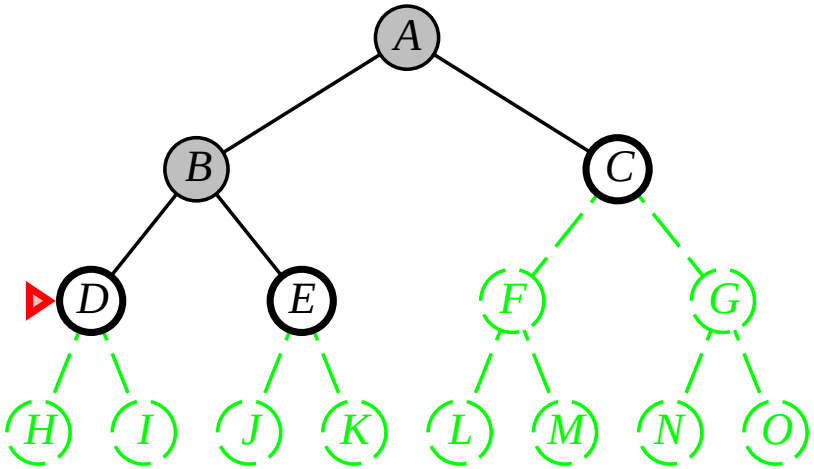


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

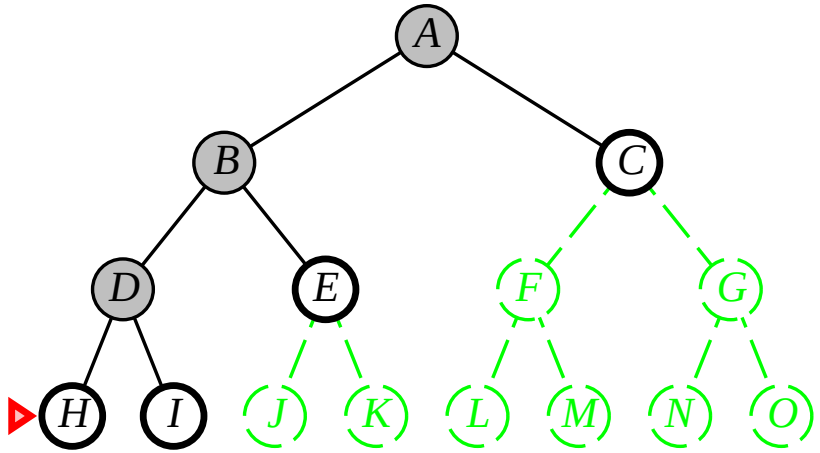


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

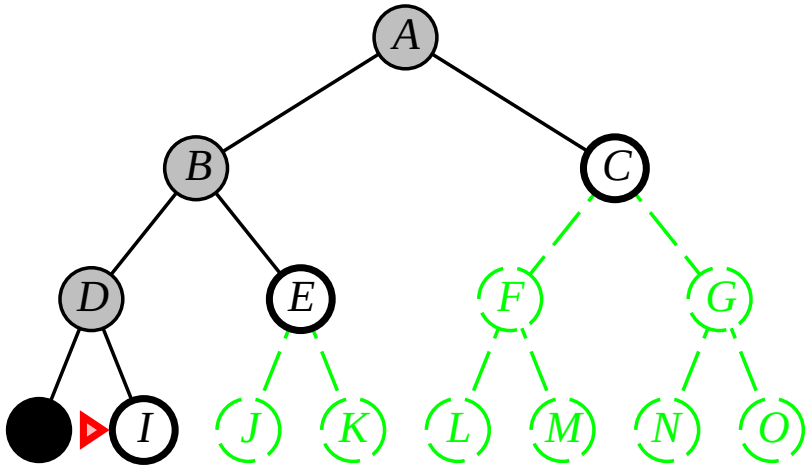


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

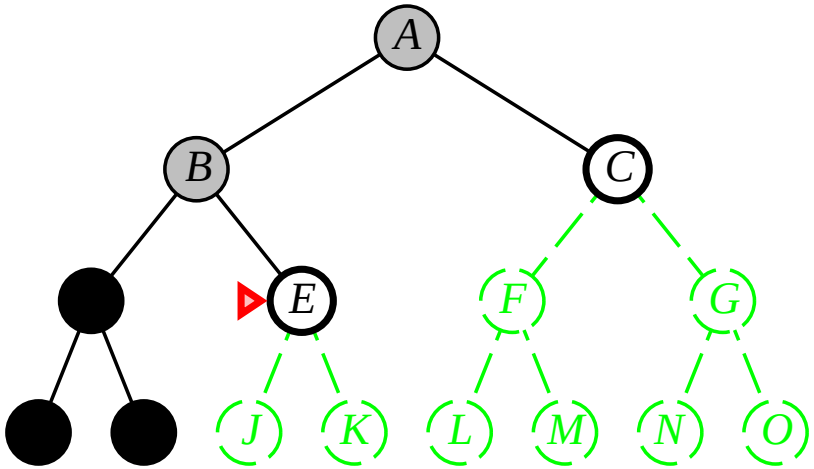


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

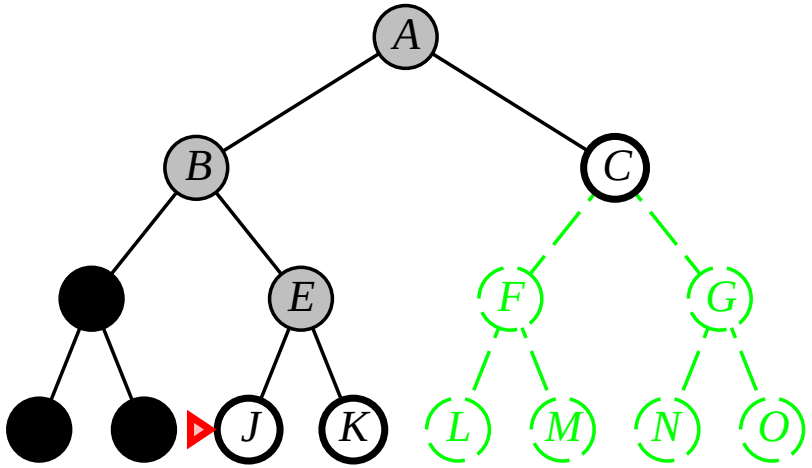


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

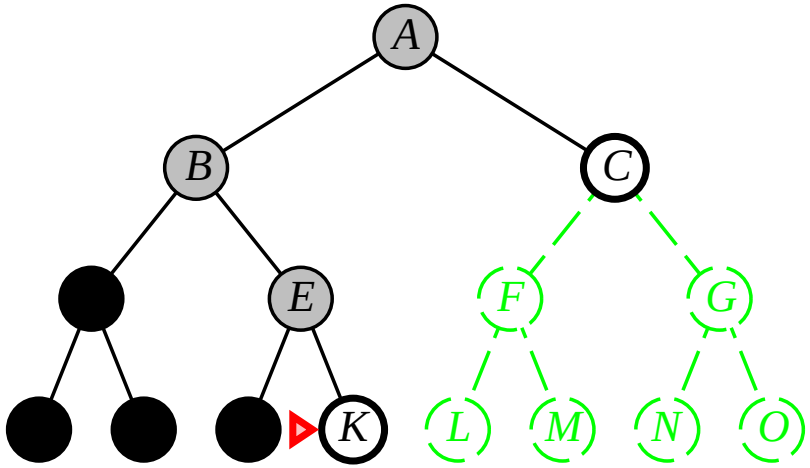


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

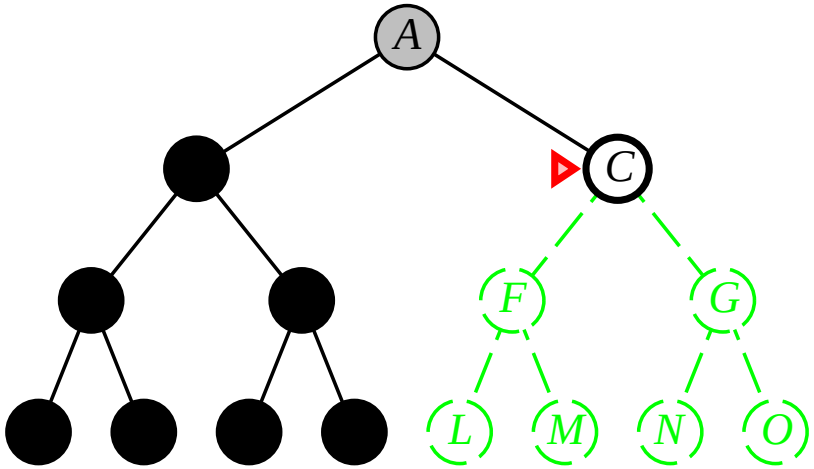


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe$ = cola LIFO , es decir, coloca los sucesores al frente

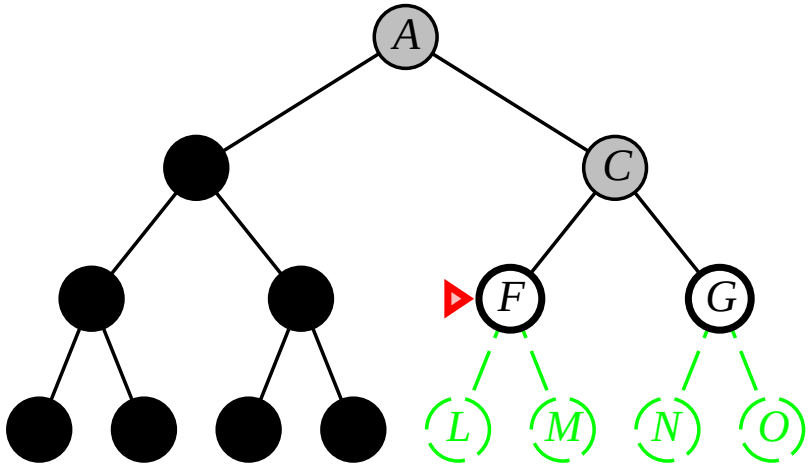


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe =$ cola LIFO , es decir, coloca los sucesores al frente

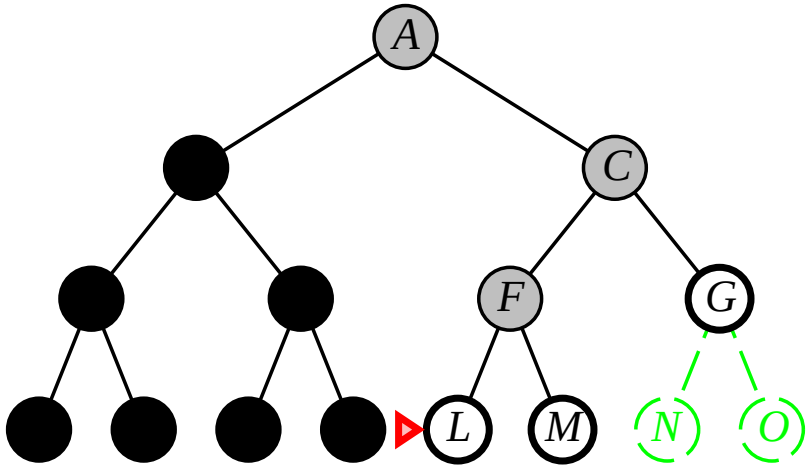


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe$ = cola LIFO , es decir, coloca los sucesores al frente

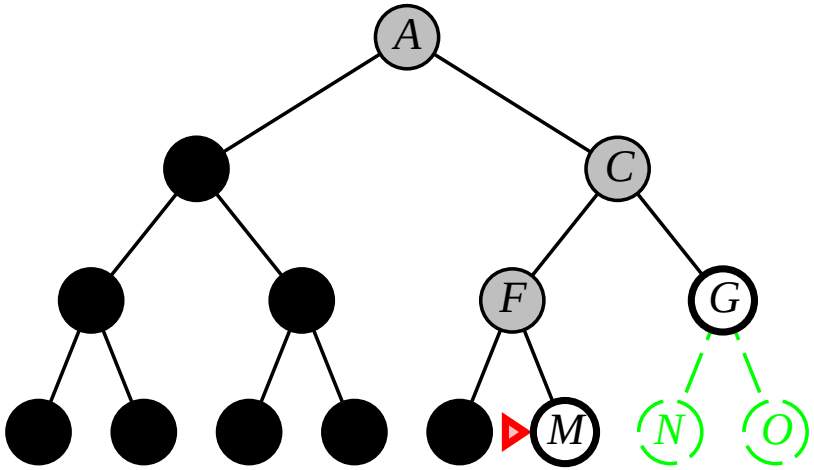


Búsqueda primero en profundidad

Expande el nodo no expandido de major profundidad.

Implementación:

$fringe$ = cola LIFO , es decir, coloca los sucesores al frente



Propiedades de la búsqueda primero en profundidad

Completa??

Completa?? No: falla en espacios de estados infinitos y espacios con ciclos.
Se puede modificar para evitar repetir estados a lo largo de un camino
$\Rightarrow$ lo cual hace que sea completa en espacios finitos (aún con ciclos).

Tiempo??

Completa?? No: falla en espacios de estados infinitos y espacios con ciclos.

Se puede modificar para evitar repetir estados a lo largo de un camino

$\Rightarrow$ lo cual hace que sea completa en espacios finitos (aún con ciclos).

Tiempo?? $O(b^m)$: muy malo si m es mucho mayor que d

pero si las soluciones son "densas" en el espacio, puede ser mucho más rápida que primero en anchura.

Espacio??

Completa?? No: falla en espacios de estados infinitos y espacios con ciclos.

Se puede modificar para evitar repetir estados a lo largo de un camino

$\Rightarrow$ lo cual hace que sea completa en espacios finitos (aún con ciclos).

Tiempo?? $O(b^m)$: muy malo si m es mucho mayor que d

pero si las soluciones son "densas" en el espacio, puede ser mucho más rápida que primero en anchura.

Espacio?? $O(bm)$, es decir, lineal en el espacio de estados!

Optima??

Completa?? No: falla en espacios de estados infinitos y espacios con ciclos.

Se puede modificar para evitar repetir estados a lo largo de un camino

$\Rightarrow$ lo cual hace que sea completa en espacios finitos (aún con ciclos).

Tiempo?? $O(b^m)$: muy malo si m es mucho mayor que d

pero si las soluciones son "densas" en el espacio, puede ser mucho más rápida que primero en anchura.

Espacio?? $O(bm)$, es decir, lineal en el espacio de estados!

Optima?? No

Búsqueda de profundidad limitada

= búsqueda primero en profundidad con límite de profundidad l,
es decir, los nodos a profundidad l no tienen sucesores.

Implementación recursiva:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

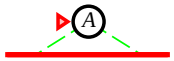
Búsqueda de profundizamiento iterativo

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

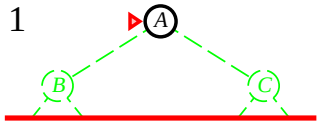
Búsqueda de profundizamiento iterativo $l = 0$

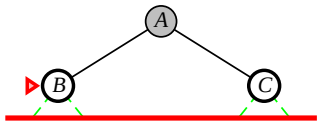
Limit = 0

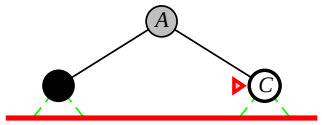


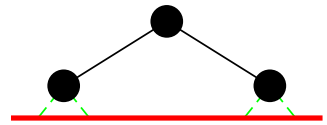
Búsqueda de profundizamiento iterativo $l = 1$

Limit = 1



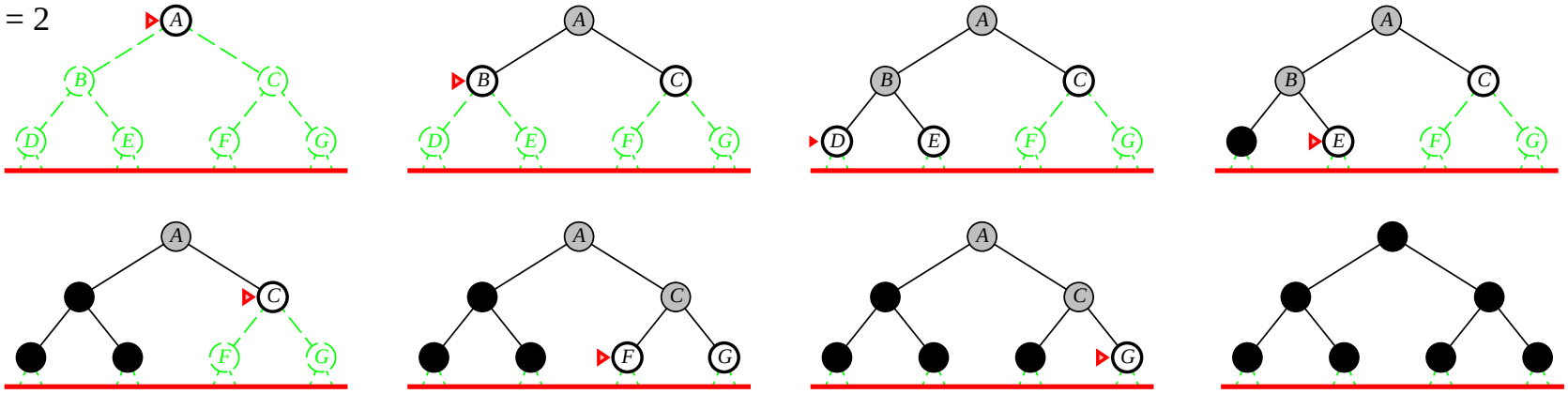




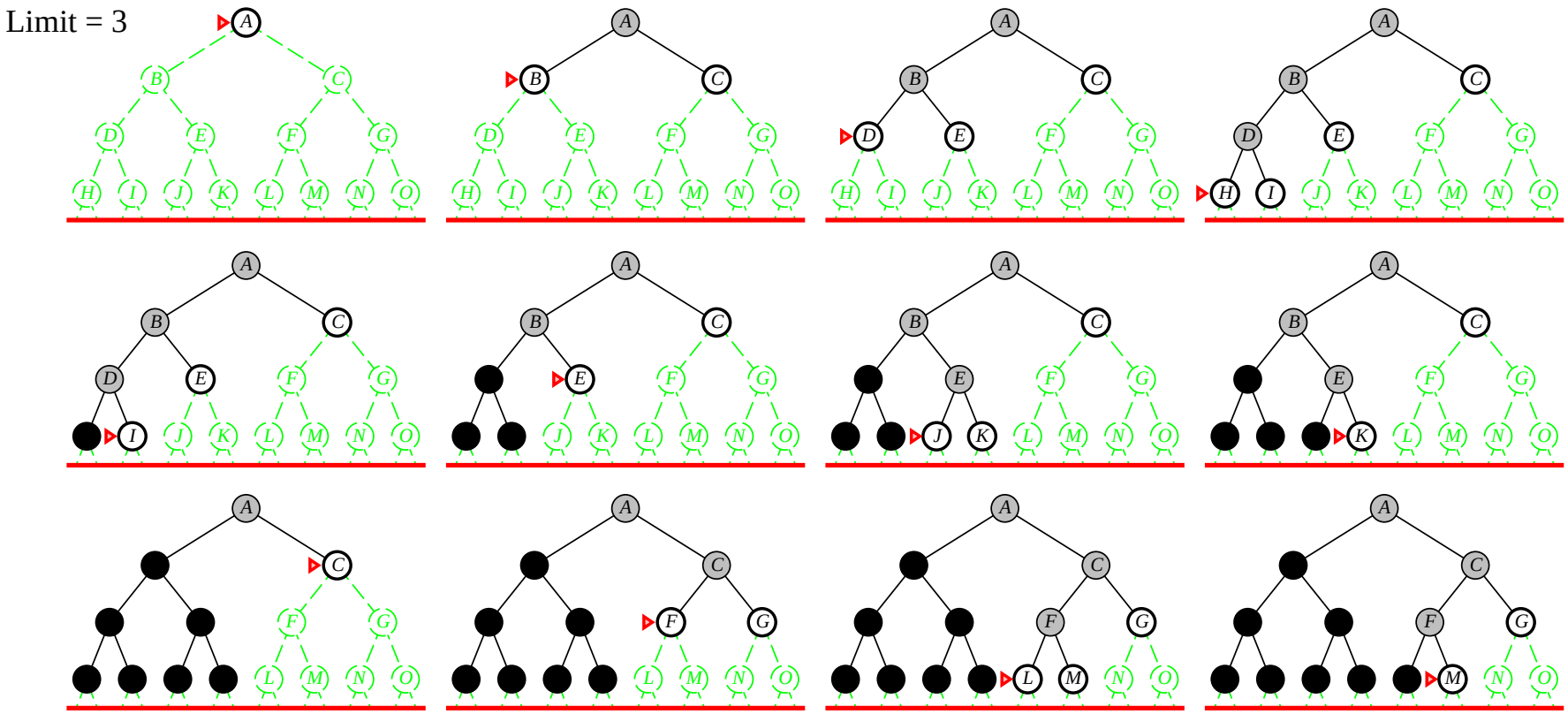


Búsqueda de profundizamiento iterativo $l = 2$

Limit = 2



Búsqueda de profundizamiento iterativo $l = 3$



Propiedades de la búsqueda de profundizamiento iterativo

Completa??

Propiedades de la búsqueda de profundizamiento iterativo

Completa?? Sí

Tiempo??

Propiedades de la búsqueda de profundizamiento iterativo

Completa?? Sí

Tiempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espacio??

Propiedades de la búsqueda de profundizamiento iterativo

Completa?? Sí

Tiempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espacio?? $O(bd)$

Optima??

Completa?? Sí

Tiempo?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Espacio?? $O(bd)$

Optima?? Sí, si el costo del paso es $= 1$

Puede ser modificada para que explore el árbol de costo uniforme.

Comparación numérica para $b = 10$ y $d = 5$. La solución a la derecha extrema:

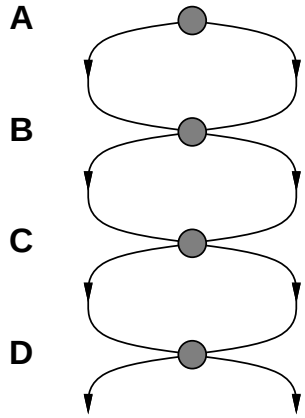
$$N(\text{Prof. iterativo}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{Primero anchura}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111$$

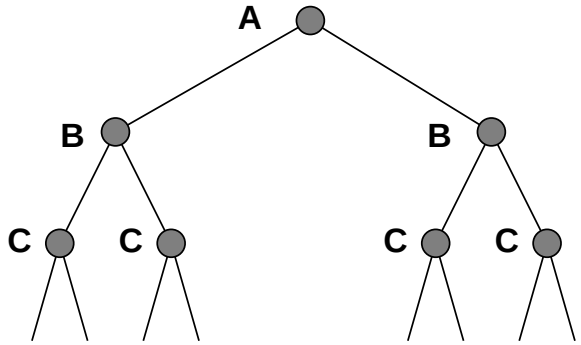
Resumen de los algoritmos

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes*	No	No	Yes

El super problema con los estados repetidos

La incapacidad de detectar los estados repetidos puede convertir un problema lineal en uno exponencial!.





Búsqueda en grafos

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Resumen

La formulación de un problema requiere usualmente que el agente abstraiga detalles de la realidad tal como se le presenta, de manera que pueda definir un espacio de estados factible para la búsqueda.

Hay una variedad de estrategias de búsqueda no informadas.

La técnica de profundizamiento iterativo usa solamente espacio lineal y no mucho más tiempo que los otros algoritmos no informados.

Volvamos con la formulación del problema. ¿Qué nos falta? ¿ Qué otra clase de conocimiento se puede usar en las búsquedas?