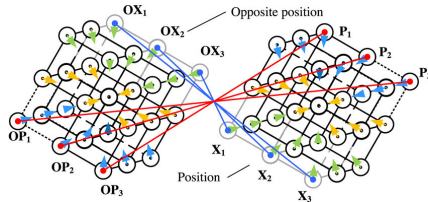


Resolver problemas mediante busqueda

Inteligencia Artificial



Marco Teran

Contenido

- 1 Agentes resolvente de problemas
- 2 Tipos de problemas
- 3 Formulación de problema bien definido
- 4 Ejemplo de problemas
- 5 Búsqueda de soluciones
- 6 Medir el rendimiento de la resolución del problema
- 7 Estrategias de búsqueda
- 8 Resumen

Agentes resolvente de problemas

Agentes resolvente de problemas

Forma sencilla de un agente general:

```
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
```

Nota: esto es una solución *offline* de problemas; solución ejecutada a "ojos cerrados."
La resolución de problemas **en línea** implica actuar sin conocimiento completo.

Ejemplo: Rumania

De vacaciones en Rumanía: actualmente en Arad.

El vuelo sale mañana de Bucarest

1 Formular el objetivo:

- Llegar a Bucarest

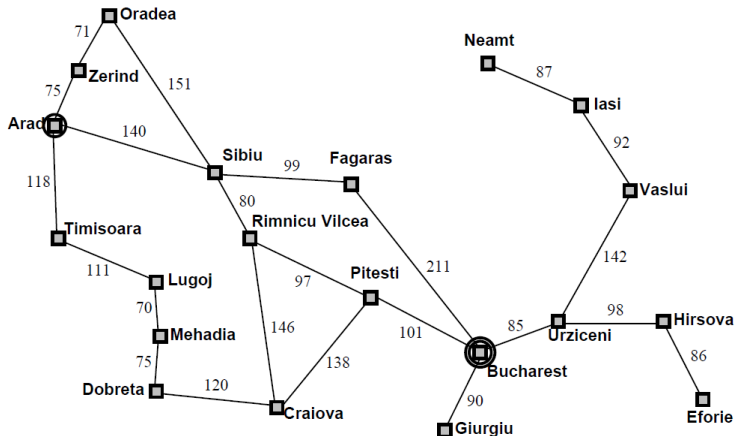
2 Formular el problema:

- **estados:** varias ciudades
- **acciones:** carreteras entre ciudades

3 Encontrar una solución:

- secuencia de ciudades, por ejemplo, *Arad, Sibiu, Fagaras, Bucarest*

Ejemplo: Rumania



Tipos de problemas

Tipos de problemas

- 1 **Determinista, completamente observable** → problema de un solo estado
 - El agente sabe exactamente en qué estado estará; la solución es una secuencia
- 2 **No observable** → problema conforme
 - El agente puede no tener idea de dónde está; la solución (si la hay) es una secuencia
- 3 **No determinista y/o parcialmente observable** → problema de contingencia
 - percepts proporcionan **nueva** información sobre el estado actual solución es un plan contingente o una política búsqueda a menudo **interleave**, ejecución
- 4 **Espacio de estado desconocido** → problema de exploración ("online")

Un solo estado, comienza en #5. **¿Solución?**

Conforme, comienza en $\{1, 2, 3, 4, 5, 6, 7, 8\}$ por ejemplo, **Derecha** $\{2, 4, 6, 8\}$. **¿Solución?**

Contingencia, comienza en el #5

Ley de Murphy: Aspirar puede ensuciar una alfombra limpia

Detección local: suciedad, sólo ubicación.

¿Solución?

[*Derecha*, **si** *suciedad* entonces *Aspirar*]



Formulación de problema bien definido

Formulación de problema bien definido

Un **problema** se define por cuatro elementos:

- 1 **Estado inicial**, p.ej., "en Arad"
- 2 **función sucesor** $S(x)$ = conjunto de pares de estado-acción por ejemplo,
 $S(Arad) = \{\langle Arad \rightarrow Zerind; Zerind \rangle, \dots\}$
- 3 **test objetivo**, puede ser
 - *explícito*, por ejemplo, $x = \text{"en Bucarest"}$
 - *implícito*, por ejemplo, $NoDirt(x)$
- 4 **costo del camino** (aditivo)
 - por ejemplo, suma de distancias, número de acciones ejecutadas, etc.
 - $c(x, a, y)$ es el costo escalonado, que se supone es ≥ 0

Una **solución** es una secuencia de acciones que lleva desde el estado inicial a un estado objetivo

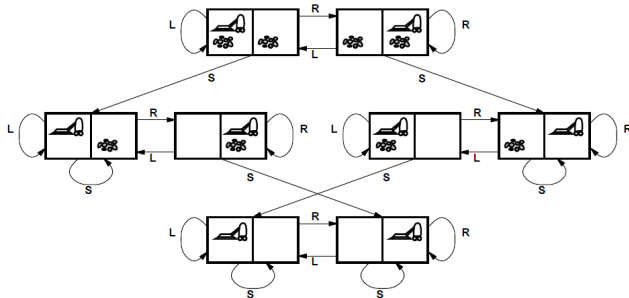
Selección de un espacio de estado

- El mundo real es absurdamente complejo
 - → espacio de estado debe ser abstraído para la resolución de problemas
- (Abstracto) estado = conjunto de estados reales
- (Abstracto) acción = combinación compleja de acciones reales
 - por ejemplo, "*Arad* → *Zerind*" representa un conjunto complejo de posibles rutas, desvíos, paradas de descanso, etc.
- Para la realizabilidad garantizada, cualquier estado real "*en Arad*"
 - debe llegar a algún estado real "*en Zerind*"
- (Abstracción) solución =
 - conjunto de caminos reales que son soluciones en el mundo real

¡Cada acción abstracta debería ser "más fácil" que el problema original!

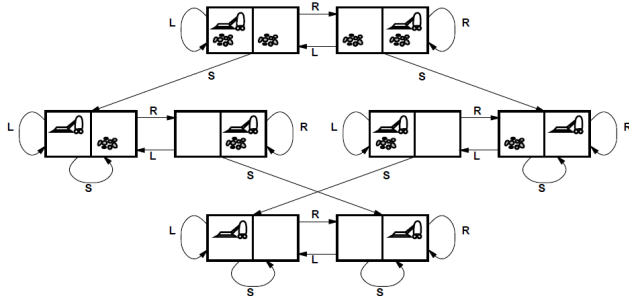
Ejemplo de problemas

Ejemplo: gráfico de espacio de estado mundial de la aspiradora



- ¿estados?
- ¿acciones?
- ¿test objetivo?
- ¿costo del camino?

Ejemplo: gráfico de espacio de estado mundial de la aspiradora



- **¿estados?** suciedad *discreta* y ubicaciones de robots (ignorar las cantidades de suciedad, etc.)
- **¿acciones?** Izquierda, derecha, aspirar, NoOp
- **¿test objetivo?** no hay suciedad
- **¿costo del camino?** 1 por acción (0 para NoOp)

Ejemplo: El rompecabezas de 8

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- ¿estados?
- ¿acciones?
- ¿test objetivo?
- ¿costo del camino?

Ejemplo: El rompecabezas de 8

7	2	4
5		6
8	3	1

Start State

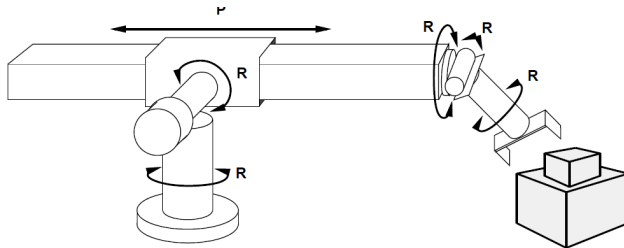
1	2	3
4	5	6
7	8	

Goal State

- **¿estados?** ubicaciones *discretas* de las fichas (ignorar posiciones intermedias)
- **¿acciones?** mover el blanco a la izquierda, derecha, arriba, abajo
- **¿test objetivo?** = estado del objetivo (dado)
- **¿costo del camino?** 1 por movimiento

[Nota: solución óptima de la familia de n-Puzzle es *NP-hard*]

Ejemplo: ensamblaje robótico



- **¿estados?** coordenadas del valor real de ángulos robóticos conjuntos. Partes del objeto a montar
- **¿acciones?** movimientos continuos de articulaciones robóticas
- **¿test objetivo?** montaje completo sin robot incluido!
- **¿costo del camino?** tiempo de ejecución

Búsqueda de soluciones

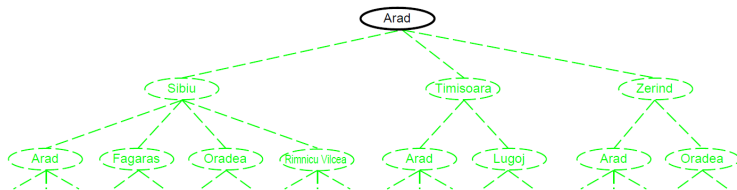
Árbol de búsqueda

Idea básica:

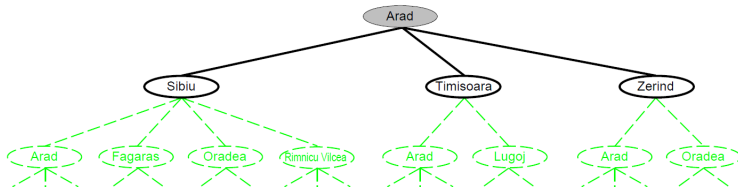
offline, exploración simulada del espacio de estado generando sucesores de Estados ya explorados (a.k.a. estados en expansión)

```
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
```

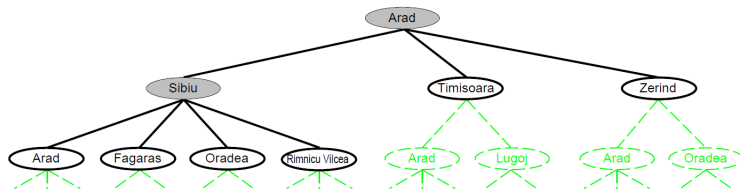
Ejemplo de búsqueda en árbol



Ejemplo de búsqueda en árbol

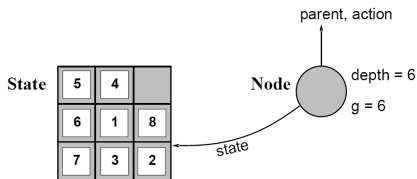


Ejemplo de búsqueda en árbol



Implementación: estados vs. nodos

- Un estado es una (representación de una) configuración física
- Un nodo es una estructura de datos que constituye parte de un árbol de búsqueda
 - incluye padre, hijos, profundidad, costo del camino $g(x)$
- ¡Los estados no tienen padres, hijos, profundidad, o costo de camino!



La función *Expand* crea nuevos nodos, rellenando los diferentes campos y utilizando el *SuccessorFn* del problema para crear los estados correspondientes.

Implementación: búsqueda general de árboles

```

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, STATE(node)) then 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

```

**Medir el rendimiento de la resolución del
problema**

Medir el rendimiento de la resolución del problema

- Una estrategia se define seleccionando el orden de expansión del nodo
- Las estrategias se evalúan en las siguientes formas:
 - 1 **completitud:** ¿siempre garantizado que encuentra una solución si existe?
 - 2 **complejidad en tiempo:** cuánto tarda, número de nodos generados/expandidos
 - 3 **complejidad en espacio:** número máximo de nodos en memoria posibles
 - 4 **optimización:** ¿siempre encuentra una solución de menor costo?
- La complejidad del tiempo y del espacio se mide en términos de
 - b -factor máximo de ramificación del árbol de búsqueda
 - d -profundidad del nodo objetivo más superficial
 - m -longitud máxima de cualquier camino en el espacio de estado (puede ser ∞)

Estrategias de búsqueda

Estrategias de búsqueda no informadas

Estrategias no informadas utilizan sólo la información disponible en la definición del problema (búsquedas a ciegas) (*no heurísticas*)

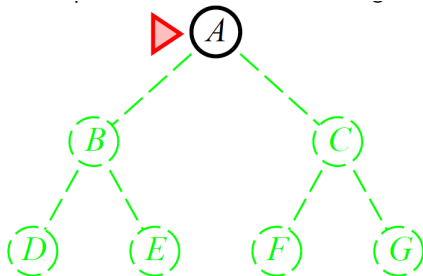
- 1 Búsqueda primero en anchura
- 2 Búsqueda de costo uniforme
- 3 Búsqueda primero en profundidad
- 4 Búsqueda de profundidad limitada
- 5 Búsqueda primero en profundidad con profundidad iterativa

Búsqueda primero en anchura

Expandir el nodo poco profundo no expandido

Aplicación:

- *fringe* es una cola de FIFO, es decir, los nuevos sucesores van al final

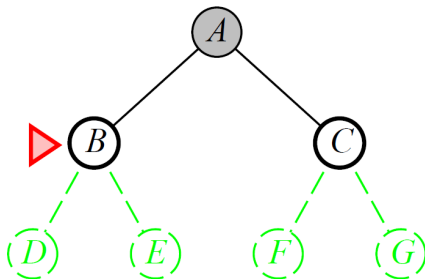


Búsqueda primero en anchura

Expandir el nodo poco profundo no expandido

Aplicación:

- *fringe* es una cola de FIFO, es decir, los nuevos sucesores van al final

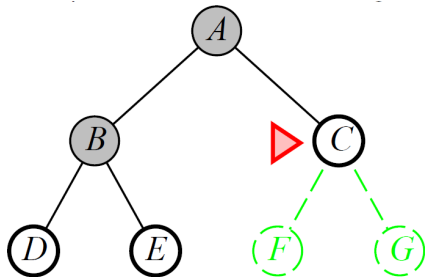


Búsqueda primero en anchura

Expandir el nodo poco profundo no expandido

Aplicación:

- *fringe* es una cola de FIFO, es decir, los nuevos sucesores van al final

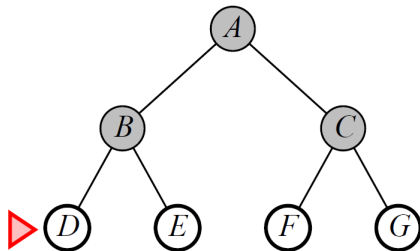


Búsqueda primero en anchura

Expandir el nodo poco profundo no expandido

Aplicación:

- *fringe* es una cola de FIFO, es decir, los nuevos sucesores van al final



Propiedades de la búsqueda primero en anchura

- **¿Completo?** Sí (si b es finito)
- **¿Tiempo?** $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, es decir, exp. en d
- **¿Espacio?** $O(b^{d+1})$ (mantiene cada nodo en memoria)
- **¿Óptimo?** Sí (si el costo = 1 por paso); no es óptimo en general

El espacio es el gran problema; puede generar fácilmente nodos a 100MB/ seg por lo
24 horas = 8640GB.

Búsqueda de costo uniforme

Expandir el nodo de menor costo no expandido

Aplicación:

- *fringe* = cola ordenada por coste del surco, primero el más bajo

Equivalente a ancho-primer si el paso cuesta todos iguales

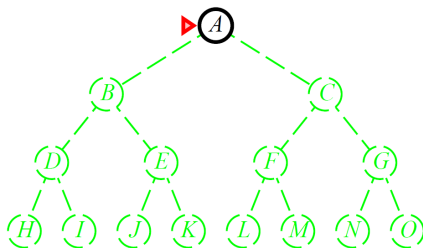
- **¿Completo?** Sí, si el costo del paso $\geq \epsilon$
- **¿Tiempo?** # de nodos con $g \leq$ costo de solución óptima, $O(b^{\lceil C^*/\epsilon \rceil})$ donde C^* es el coste de la solución óptima
- **¿Espacio?** # de nodos con $g \leq$ costo de solución óptima, $O(b^{\lceil C^*/\epsilon \rceil})$
- **¿Óptimo?** Sí-nodos expandidos en orden creciente de $g(n)$

Búsqueda primero en profundidad

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

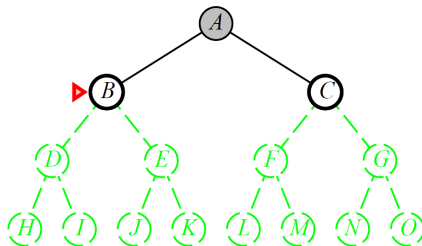


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

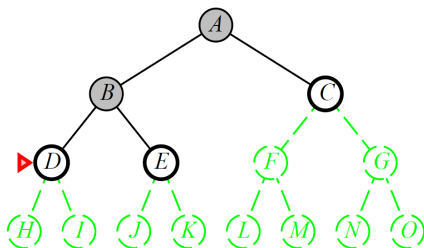


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

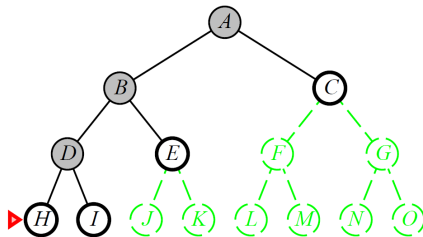


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

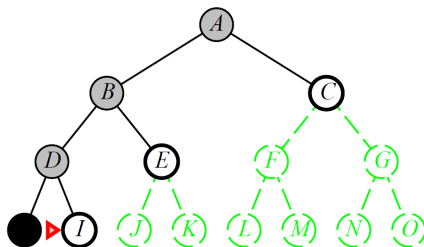


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

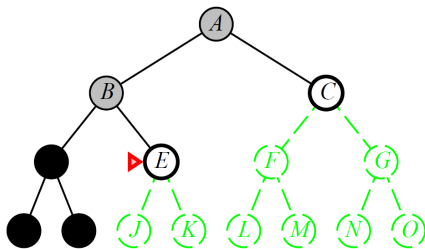


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

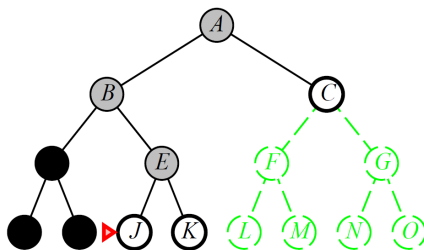


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

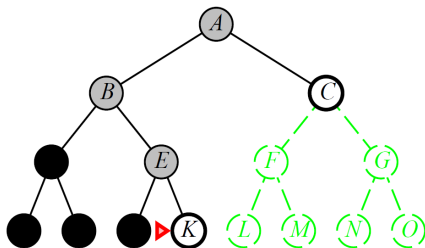


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

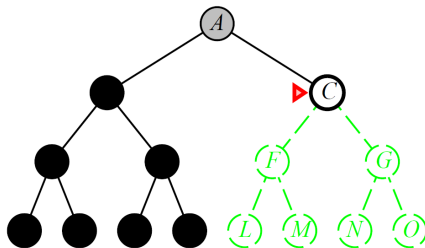


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

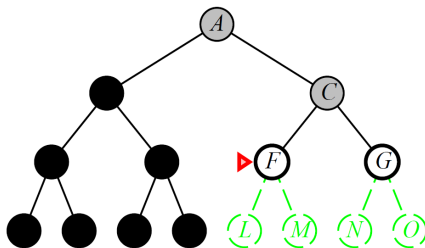


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

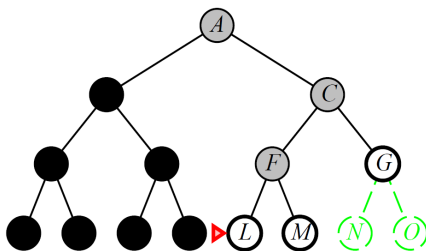


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante

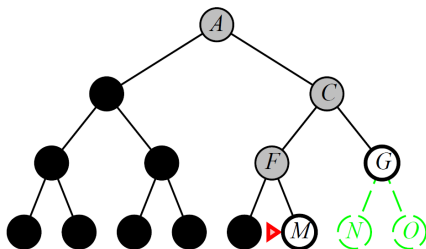


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

- *fringe* = cola LIFO, es decir, poner sucesores delante



Propiedades de la búsqueda de profundidad inicial

- **¿Completo?** No: falla en espacios de profundidad infinita, espacios con bucles. Modificar para evitar estados repetidos a lo largo de la ruta → completar en espacios finitos
- **¿Tiempo?** $O(b^m)$: terrible si m es mucho más grande que d , pero si las soluciones son densas, puede ser mucho más rápido que la breadth-first
- **¿Espacio?** $O(bm)$, es decir, espacio lineal!
- **¿Óptimo?** No

Búsqueda de profundidad limitada

= búsqueda de profundidad inicial con límite de profundidad l , es decir, los nodos en la profundidad l no tienen sucesores

Aplicación 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 primero en profundidad con profundidad iterativa

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to  $\infty$  do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
  end
```

Búsqueda primero en profundidad con profundidad iterativa

$l = 0$

Limit = 0



Búsqueda primero en profundidad con profundidad iterativa

$l = 1$

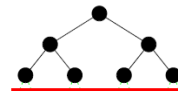
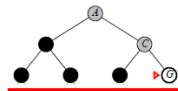
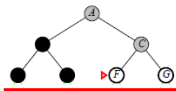
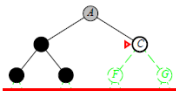
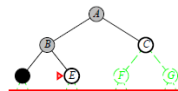
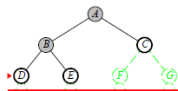
Limit = 1



Búsqueda primero en profundidad con profundidad iterativa

$l = 2$

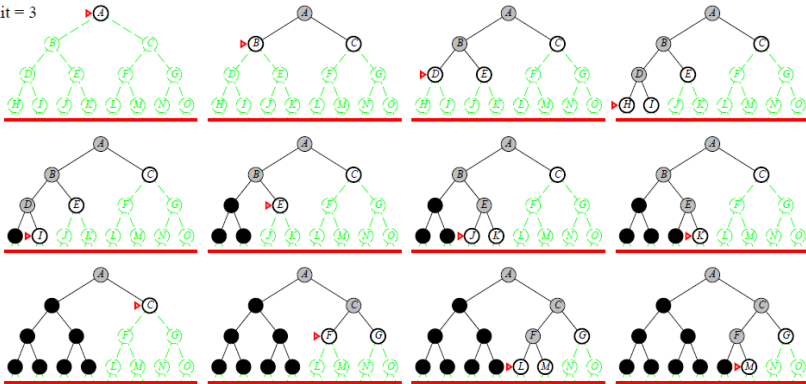
Limit = 2



Búsqueda primero en profundidad con profundidad iterativa

$l = 3$

Limit = 3



Propiedades de la búsqueda iterativa de profundización

- **¿Completo?** Sí
- **¿Tiempo?** $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- **¿Espacio?** $O(bd)$
- **¿Óptimo?** Sí, si el costo del paso = 1. Se puede modificar para explorar el árbol de costo uniforme

Comparación numérica para $b = 10$ y $d = 5$, solución a la derecha:

$$N(BPI) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BPA) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

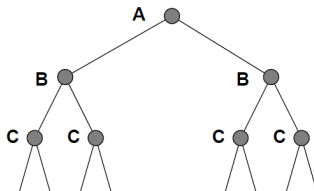
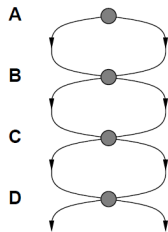
- IDS funciona mejor porque otros nodos en profundidad d no se expanden
- BFS se puede modificar para aplicar la prueba de objetivos cuando se **genera** un nodo

Resumen de los algoritmos

Criterio	Primero en anchura	Costo uniforme	Primero en profundidad	Profundidad limitada	Profundidad iterativa
¿Completo?	Sí*	Sí*	No	Sí, si $l \geq d$	Sí
Tiempo	b^{d+1}	$b^{\lceil C*/\epsilon \rceil}$	b^m	b^l	b^d
Espacio	b^{d+1}	$b^{\lceil C*/\epsilon \rceil}$	bm	bl	bd
¿Óptimo?	Sí*	Sí	No	No	Sí*

Evitar estados repetidos

¡La falla en detectar estados repetidos puede convertir un problema lineal en uno exponencial!



Búsqueda por gráficos

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

Resumen

- La formulación de problemas generalmente requiere abstraer detalles del mundo real para definir un espacio de estado que pueda explorarse de forma factible
- Variedad de estrategias de búsqueda desinformadas
- La búsqueda de profundización iterativa utiliza sólo espacio lineal y no mucho más tiempo que otros algoritmos desinformados
- La búsqueda por gráficos puede ser exponencialmente más eficiente que la búsqueda en árbol

Muchas gracias por su atención

¿Preguntas?



Contacto: Marco Teran
webpage: marcoteran.github.io/