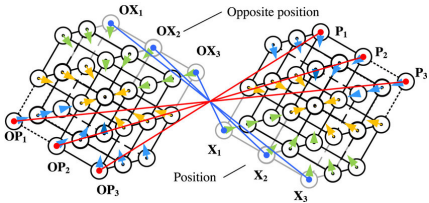
Resolver problemas mediante busqueda

Inteligencia Artificial



Marco Teran

2021 - Bogotá

Contenido

Agentes de resolución de problemas

Forma restringida de 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
```

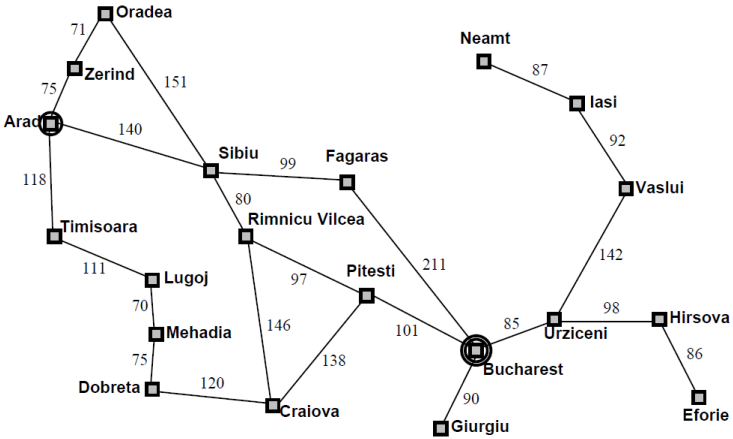
Nota: esto es o ine solución de problemas; solución ejecutada "ojos cerrados."
La resolución de problemas **en línea** implica actuar sin conocimiento completo.

Ejemplo: Rumania

De vacaciones en Rumanía; actualmente en Arad.
El vuelo sale mañana de Bucarest

- **Formular la meta:**
  - estar en Bucarest
- **Formular el problema:**
  - **estados:** varias ciudades
  - **acciones:** impulso entre ciudades
- **Encontrar una solución:**
  - secuencia de ciudades, por ejemplo, Arad, Sibiu, Fagaras, Bucarest

Ejemplo: Rumania



Tipos de problemas

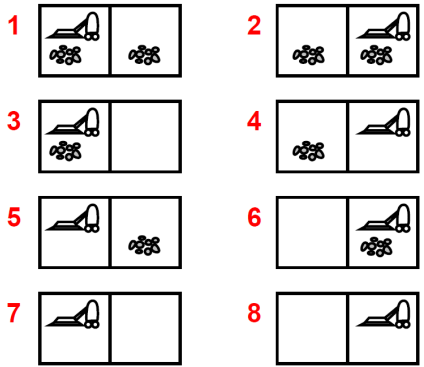
1. **Determinista, completamente observable** → problema de un solo estado
   - El agente sabe exactamente en qué estado estará; la solución es una secuencia
2. **No observable** → problema conforme
   - El agente puede no tener idea de dónde está; la solución (si la hay) es una secuencia
3. **No determinista 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")

Ejemplo: el mundo de la aspiradora

Un solo estado, comienza en $\#5$. **¿Solución?**
$[Derecha, Succionar]$

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

Contingencia, comienza en el $\#5$
Ley de Murphy: succionar puede ensuciar una alfombra limpia
Detección local: suciedad, sólo ubicación.
¿Solución?
$[Derecha,$ **si** $suciedad$ **entonces** $Succionar]$



Formulación del problema en un solo estado

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

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

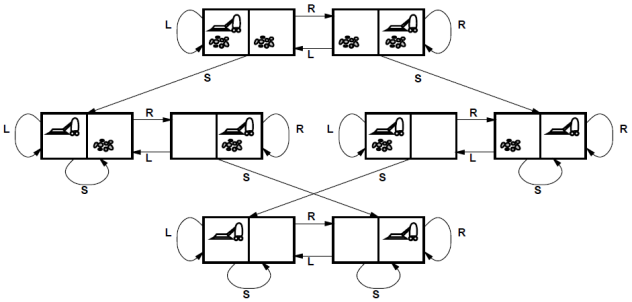
Una **solución** es una secuencia de acciones
que lleva desde el estado inicial a un estado objetivo

Selección de un espacio de estado

- El mundo real es absurdamente complejo
  - $\rightarrow$ 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* $\rightarrow$ *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"*
- (Abstracto) solución =
  - conjunto de caminos reales que son soluciones en el mundo real

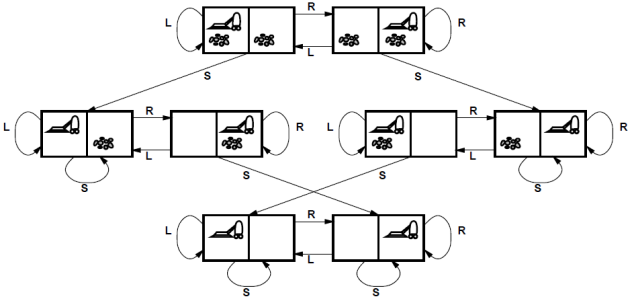
¡Cada acción abstracta debería ser "más fácil" que el problema original!

Ejemplo: gráfo de espacio de estado mundial de vacío



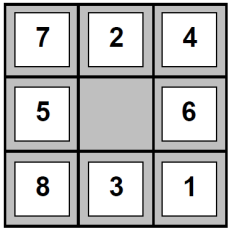
- ¿estados?
- ¿acciones?
- ¿test de objetivos?
- ¿costo de la ruta?

Ejemplo: gráfo de espacio de estado mundial de vacío

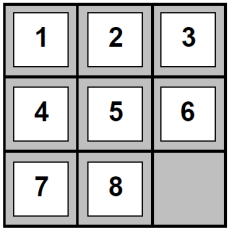


- **¿estados?** suciedad entera y ubicaciones de robots (ignorar las cantidades de suciedad, etc.)
- **¿acciones?** Izquierda, derecha, chupar, NoOp
- **¿test de objetivos?** no hay suciedad
- **¿ruta de costo?** 1 por acción (0 para NoOp)

Ejemplo: El rompecabezas de 8



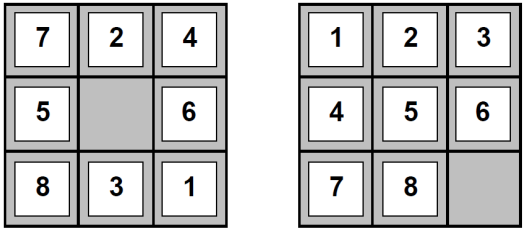
Start State



Goal State

- ¿estados?
- ¿acciones?
- ¿test de objetivos?
- ¿costo de la ruta?

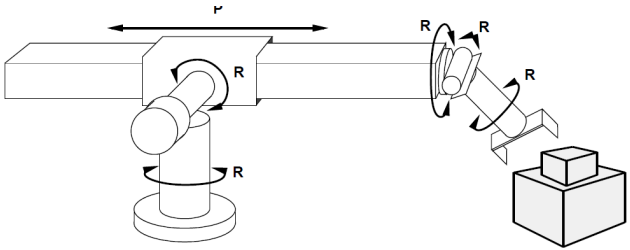
Ejemplo: El rompecabezas de 8



- **¿estados?** ubicaciones enteras de las baldosas (ignorar posiciones intermedias)
- **¿acciones?** mover en blanco a la izquierda, derecha, arriba, abajo (ignorar unjamming etc.)
- **¿test de objetivos?** = estado del objetivo (dado)
- **¿costo de la ruta?** 1 por movimiento

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

Ejemplo: ensamblaje robótico



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

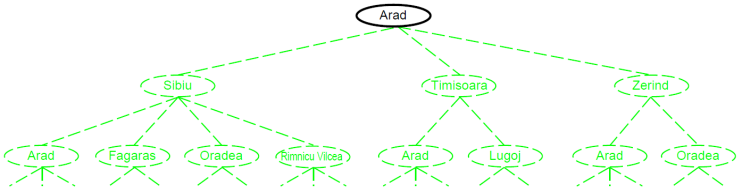
Algoritmos de búsqueda en árbol

Idea básica:

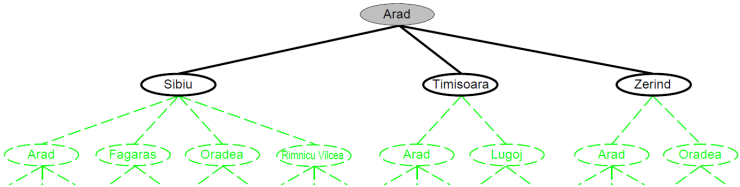
offline, exploración simulada del espacio de estado generando sucesores de Estados ya explorados (a.k.a. estados en expansión)

```
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
```

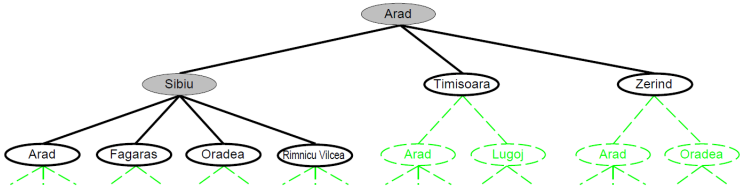
Ejemplo de búsqueda en árbol



Ejemplo de búsqueda en árbol

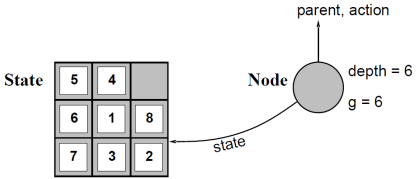


Ejemplo de búsqueda en árbol



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 de la ruta $g(x)$
- ¡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
```

Estrategias de búsqueda

- Una estrategia se define seleccionando el orden de expansión del nodo
- Las estrategias se evalúan en las siguientes dimensiones:
  - completitud: ¿siempre encuentra una solución si existe?
  - tiempo complejidad-número de nodos generados/expandidos
  - complejidad espacial-número máximo de nodos en memoria
  - optimality-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 de la solución de menor coste
  - m-profundidad máxima del espacio de estado (puede ser $\inf$)

Estrategias de búsqueda no informadas

Estrategias desinformadas utilizar sólo la información disponible en la definición del problema

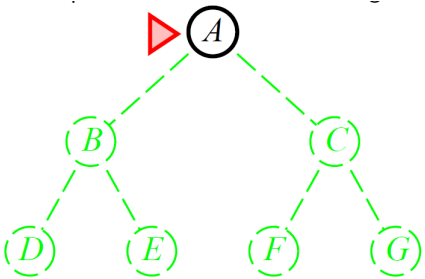
1. Búsqueda por amplitud inicial
2. Búsqueda de costos uniformes
3. Búsqueda a fondo
4. Búsqueda de profundidad limitada
5. Búsqueda iterativa de profundización

Búsqueda por amplitud inicial

Expandir el nodo poco profundo no expandido

Aplicación:

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

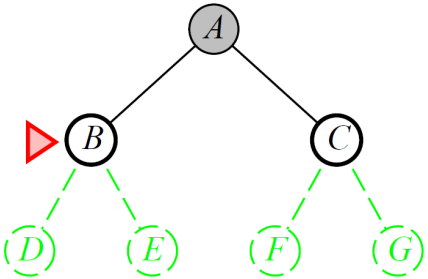


Búsqueda por amplitud inicial

Expandir el nodo poco profundo no expandido

Aplicación:

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

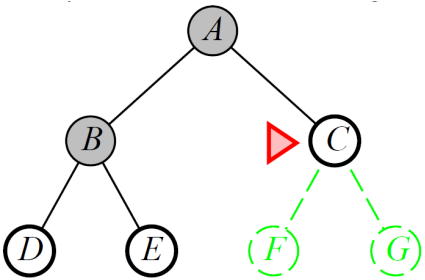


Búsqueda por amplitud inicial

Expandir el nodo poco profundo no expandido

Aplicación:

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

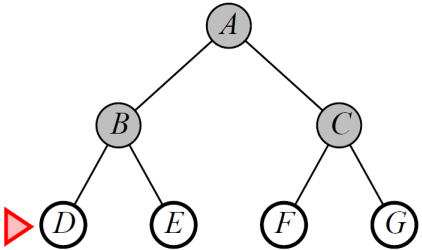


Búsqueda por amplitud inicial

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 primera búsqueda de amplitud

- **¿Completo?** Sí (si b es finito)
- **¿Tiempo?** $1+b+b^2+b^3+...+b^d+b(b^d-1)=O(b^{d+1})$, es decir, exp. in 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 costos uniformes

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

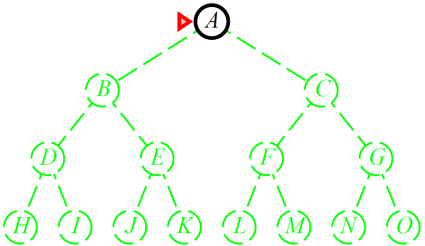
- **¿Completo?** Sí, si el costo del paso $\geq \epsilon$
- **¿Tiempo?** # de nodos con $g \leq$ costo de solución óptima, $O(b^{\lceil C*/\epsilon \rceil})$ donde $C*$ es el coste de la solución óptima
- **¿Espacio?** # de nodos con $g \leq$ costo de solución óptima, $O(b^{\lceil C*/\epsilon \rceil})$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

Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

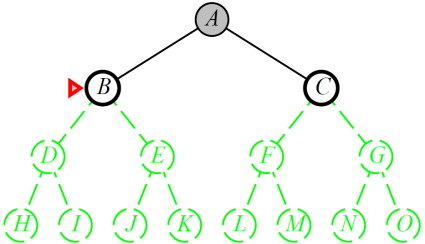


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

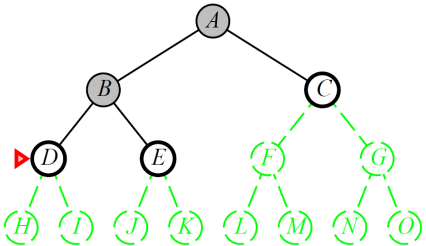


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

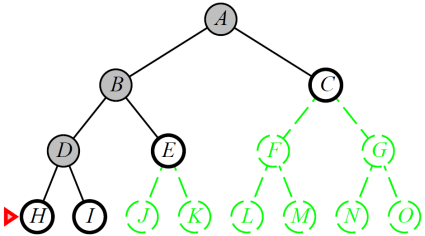


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

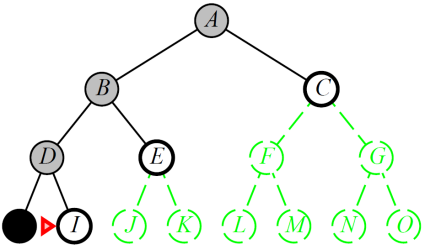


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

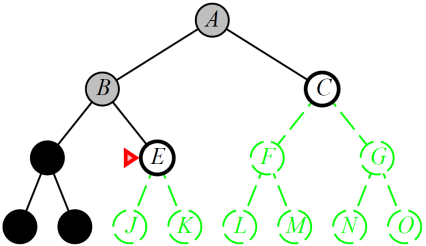


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

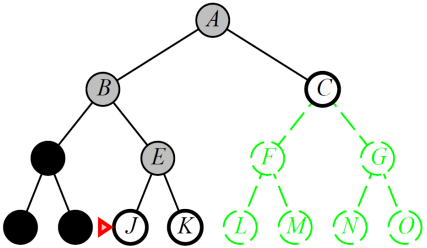


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

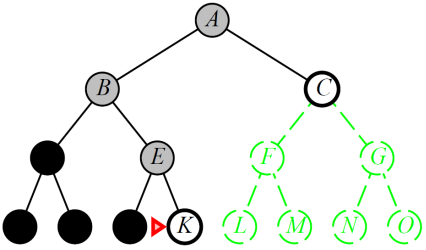


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

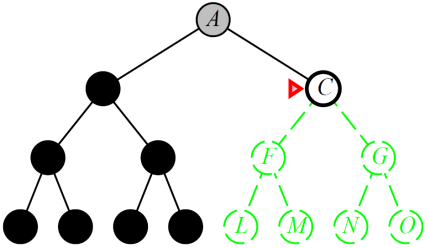


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

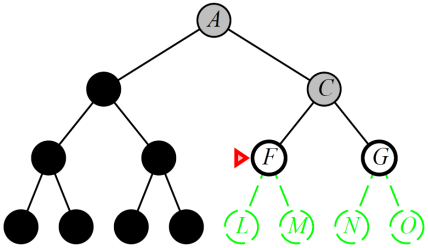


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

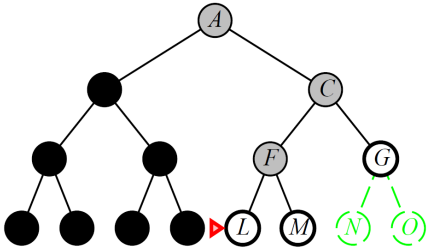


Búsqueda a fondo

Expandir el nodo más profundo no expandido

Aplicación:

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

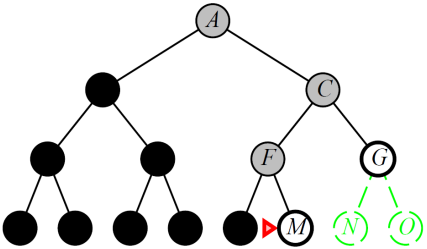


Búsqueda a fondo

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 $\rightarrow$ 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 iterativa de profundización

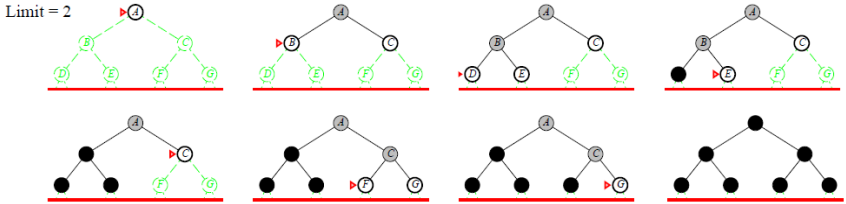
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

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

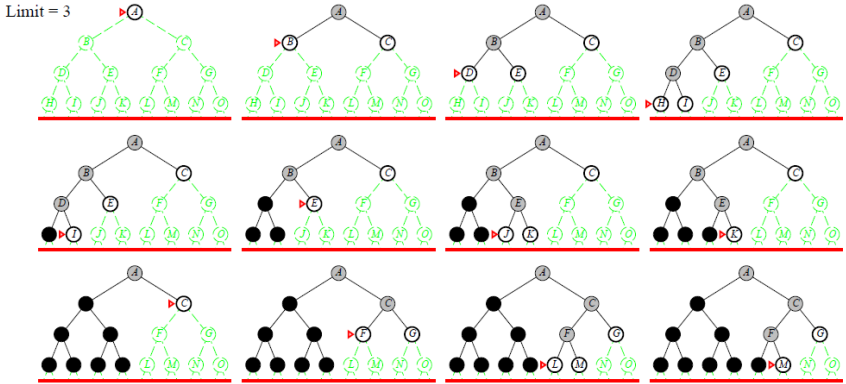
Búsqueda iterativa de profundización $l = 0$

Búsqueda iterativa de profundización $l = 1$

Búsqueda iterativa de profundización $l = 2$



Búsqueda iterativa de profundización $l = 3$



Propiedades de la búsqueda iterativa de profundización

- **¿Completo?** Sí
- **¿Tiempo?** $(d+1)b^0 + db^1 + (d-1)b^2 + ... + b^d = O(b^d)$
- **¿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:

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

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

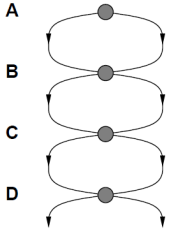
- IDS funciona mejor porque otros nodos en profundidad d no se expanden
- BFS se puede modificar para aplicar la prueba de objetivos cuando se **genera** un nodo

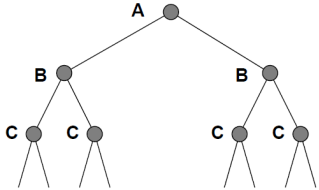
Resumen de los algoritmos

Criterio	Ancho	Uniforme	Profundidad	Profundidad limitada	Profundización 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*

Estados repetidos

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





Búsqueda por gráfos

```
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 gráfos puede ser exponencialmente más eficiente que la búsqueda en árbol

Muchas gracias por su atención

¿Preguntas?

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