Tema 2. Resolución de problemas mediante búsqueda

Sistemas Inteligente

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Contents

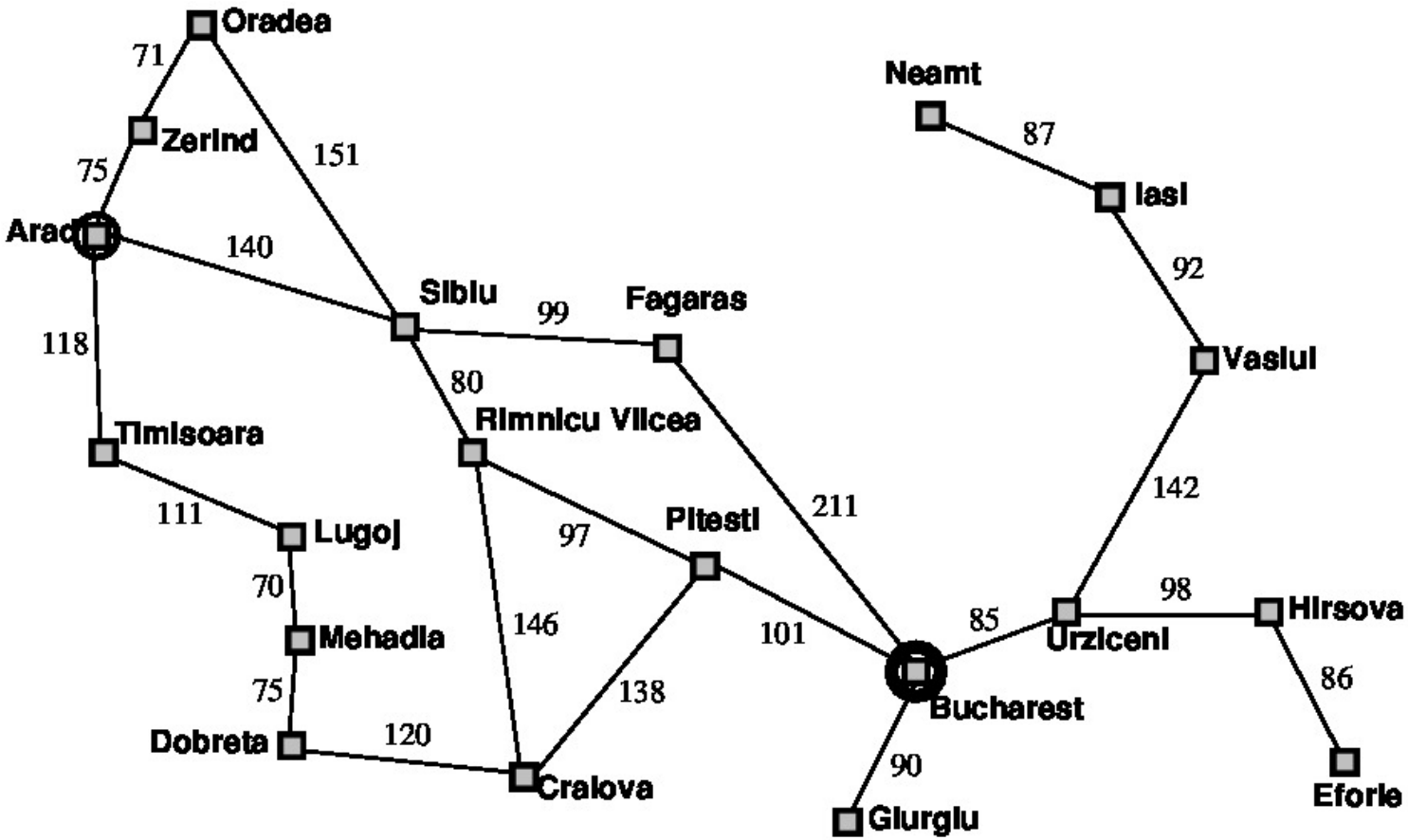
Introducción

- El objetivo **principal** es construir **agentes** que puedan crear un **plan** "hacía adelante" para resolver **problemas**.
- En esta clase de **problemas** hay muchos **estados**.

Ejemplo: Rumanía

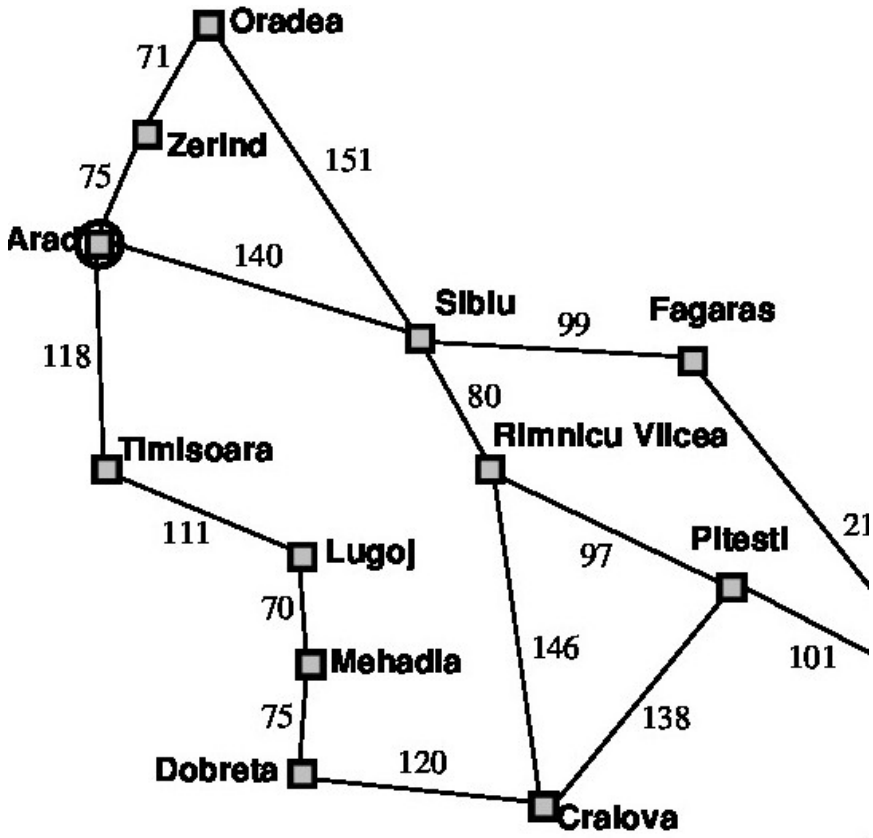
- De vacaciones en **Rumanía**; actualmente en **Arad**.
- El **vuelo** despega mañana de **Bucarest**
- *Formula el objetivo*: estar en **Bucarest**
- *Formula el problema*:
  - *estados*: varias **ciudades**
  - *acciones*: **conducir** entre ciudades.
- *Encontrar una solución*: **secuencia** de **acciones**, e.j., Arad-Sibiu, Sibiu-Fagaras, Fagaras-Bucharest

Ejemplo: Rumanía



Ejercicio

- ¿Existe una solución al **problema** de conducir desde **Arad** hasta **Bucharest**?



Agentes resolutores de problemas

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*

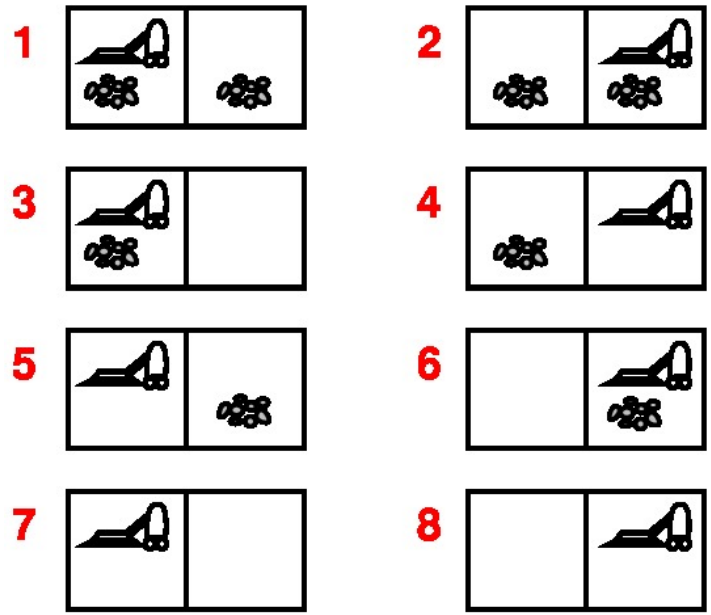
Contents

Tipos de problemas y espacios de estados.

- *Determinístico, Completamente observable*: *problema con un sólo estado*. **Agente** conoce **exactamente** en que **estado** se encuentra: la solución es una **secuencia de acciones**.
- *No-observable*: *problema conforme*. Agente puede **no saber** donde se encuentra; La solución (si existe) es una **secuencia**
- *No-determinístico* y/o *parcialmente observable*: *problema de contingencia*. **Las percepciones** proporcionan *nueva* información sobre el **estado** actual. La solución es un **plan de contingencia** o una **política**.
- *Espacio de estados desconocido*: *problema de exploración* ("**online**")

Ejemplo: mundo de la aspiradora

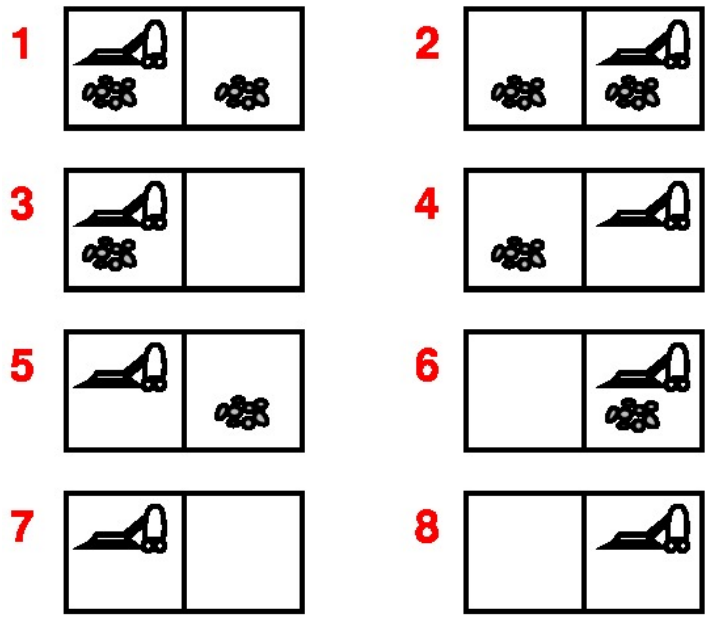
Single-state, start in #5. **Solution**



Ejemplo: mundo de la aspiradora

Single-state, start in #5. **Solution**
[*Right*, *Suck*]

Sensorless, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. **Solution**

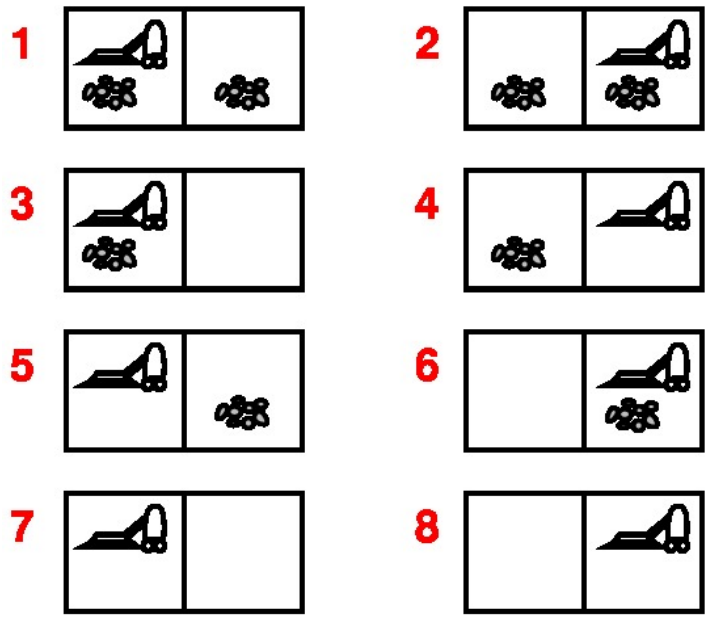


Ejemplo: mundo de la aspiradora

Single-state, start in #5. **Solution**
[*Right*, *Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. **Solution**
[*Right*, *Suck*, *Left*, *Suck*]

Contingency, start in #5
Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.
Solution

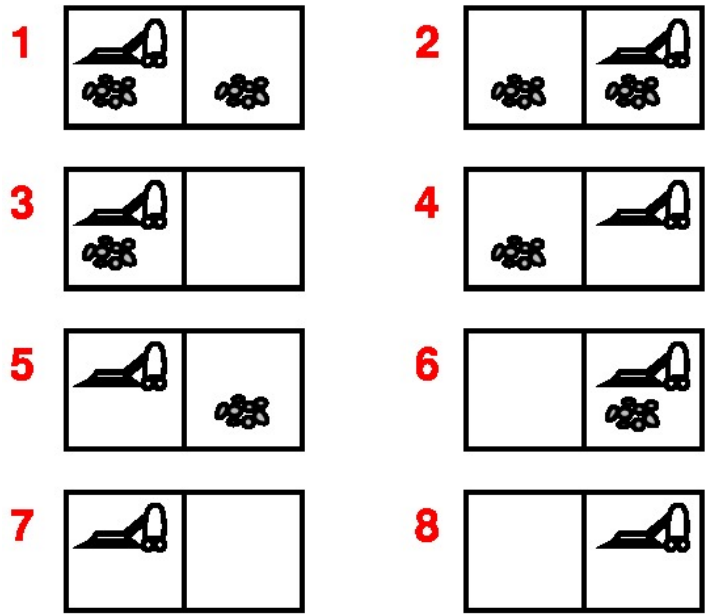


Ejemplo: mundo de la aspiradora

Single-state, start in #5. **Solution**
[*Right*, *Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., *Right* goes to $\{2, 4, 6, 8\}$. **Solution**
[*Right*, *Suck*, *Left*, *Suck*]

Contingency, start in #5
Murphy's Law: *Suck* can dirty a clean carpet
Local sensing: dirt, location only.
Solution
[*Right*, **if** *dirt* **then** *Suck*]



Contents

Formulación de un problema

- *estado inicial* e.g., "en **Arad**"
- *función sucesor* $S(x)$ = conjunto de tripletas **acción–estado–coste** e.g., $S(Arad) = \{(< Arad \rightarrow Zerind), Zerind, 75 >, \ldots\}$
- función objetivo, puede ser:
  - *explícita*, e.g., $x =$ "en **Bucharest**"
  - *implícita*, e.g., $NoDirt(x)$
- *coste del camino* (aditivo) p.e., suma de **distancias**, número de **acciones** ejecutadas, etc. $c(x, a, y)$ es el *coste de un paso*, asumiendo siempre que es ≥ 0

Solución

Una *solución* es una secuencia de acciones llevando del estado **inicial** a un estado **final**.

Seleccionando un espacio de estados. I

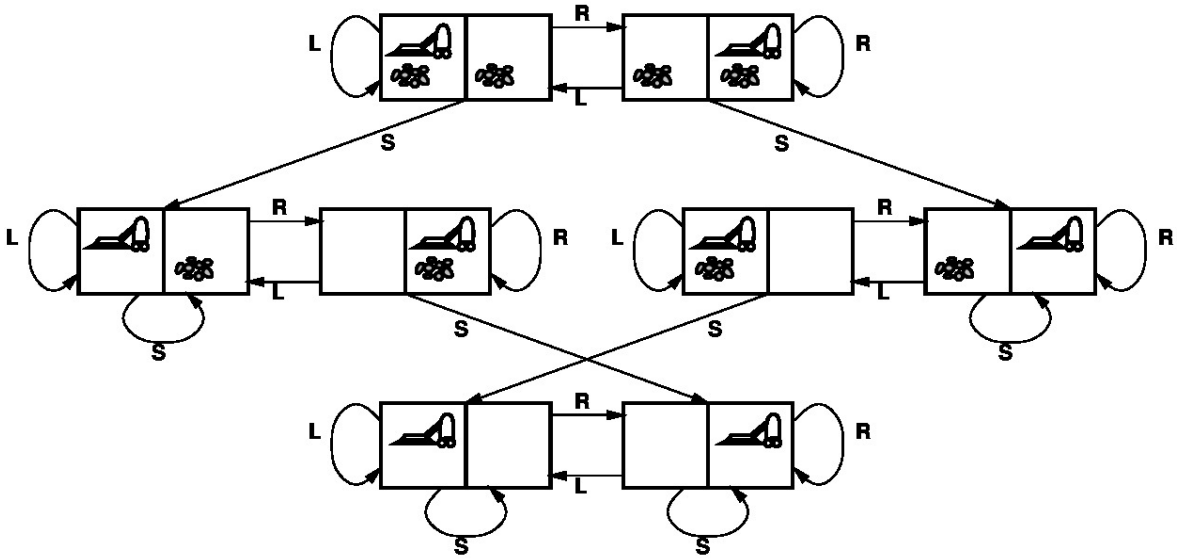
- **El mundo real es absurdamente complejo.**: el espacio de estado debe ser *abstraido* para solucionar un **problema**
- **(Abstracto) estado**: conjunto de **estados** reales.
- **(Abstracto) acción**: combinación de acciones reales: e.g., "Arad $\rightarrow$ Zerind" representa un conjunto complejo de posibles rutas, desvíos, paradas de descanso, etc. Para su garantizar su **realizabilidad**, *cualquier* estado real "in Arad" debe alcanzar a *algún* estado real 'in Zerind"
- **(Abstracto) Solución**: conjunto de caminos reales que son **soluciones** en el mundo real.

Seleccionando un espacio de estados. II

- Una acción **abstracta** debería ser "**más facil**" que el problema **original**!
- La elección de una buena **abstracción** debe derivar en una eliminación del máximo de **detalles** como sea posible mientras que se **mantenga** la validez garantizando que las acciones **abstractas** son fáciles de **llevar a cabo**.
- Si no es posible construir **abstracciones útiles**, los **agentes** inteligentes estarían completamente **abrumados** por el mundo real.

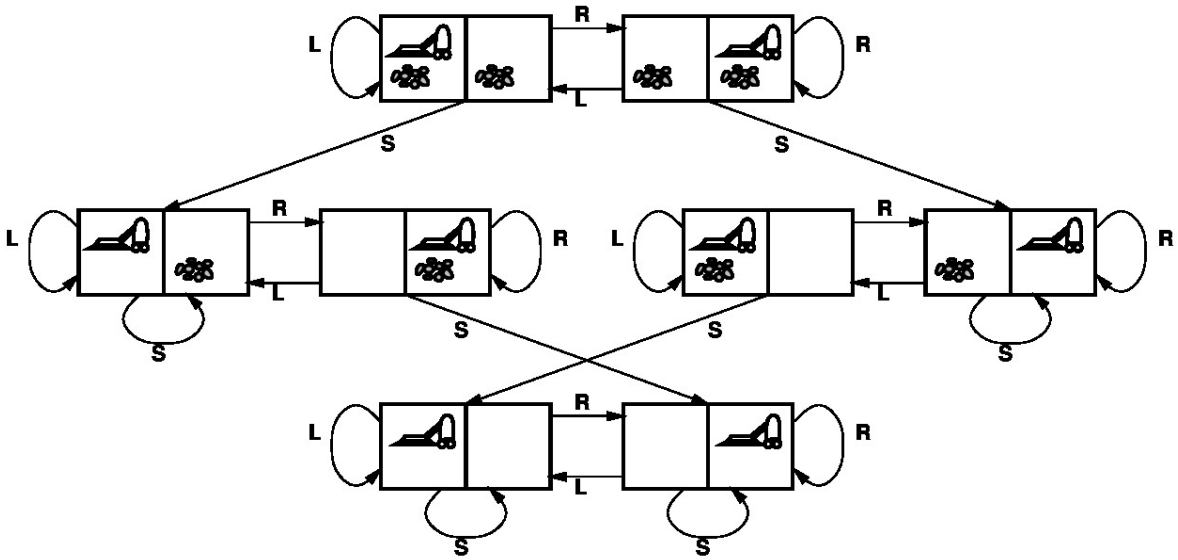
Contents

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



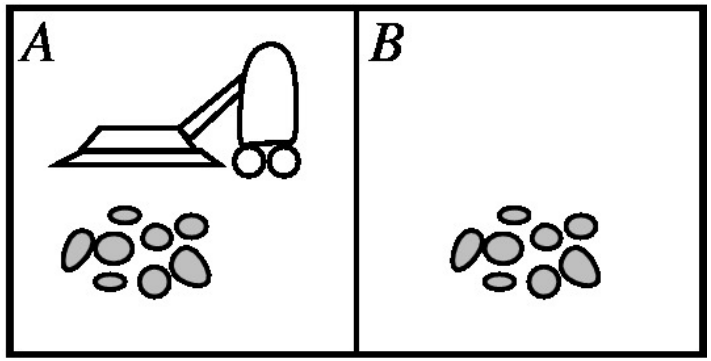
states: ?
actions: ?
goal test: ?
path cost: ?

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



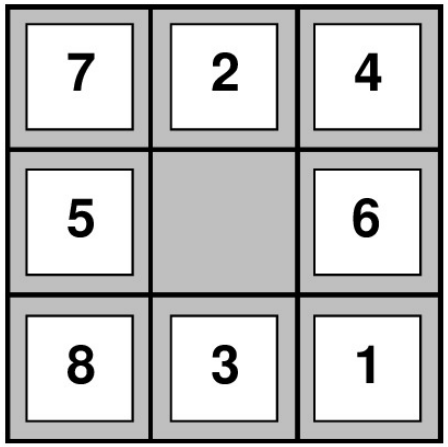
states: integer dirt and robot locations (ignore dirt *amounts* etc.)
actions: *Left*, *Right*, *Suck*, *NoOp*
goal test: no dirt
path cost: 1 per action (0 for *NoOp*)

Ejercicio I

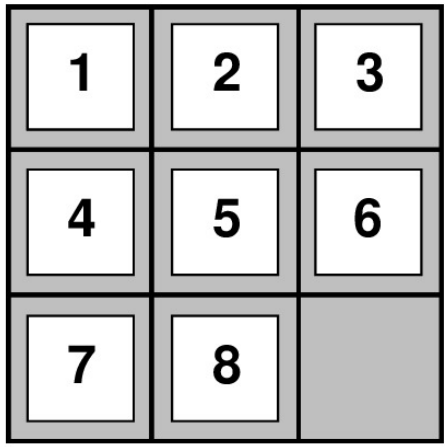


- **Power switch**: on/off/sleep
- **Dirt sensing camera**: on/off
- **Brushes height**: 1/2/3/4/5
- **Positions**: 10 (not only A and B)
- How many **states** in the **state space** are?

Example: El 8-puzzle I



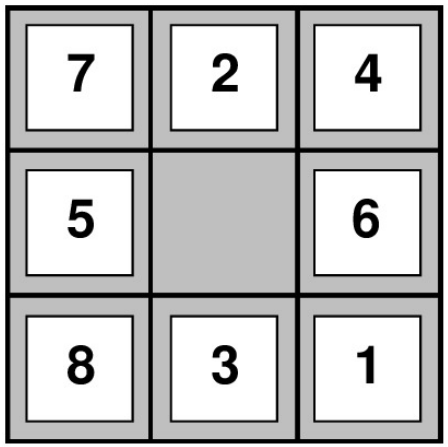
Start State



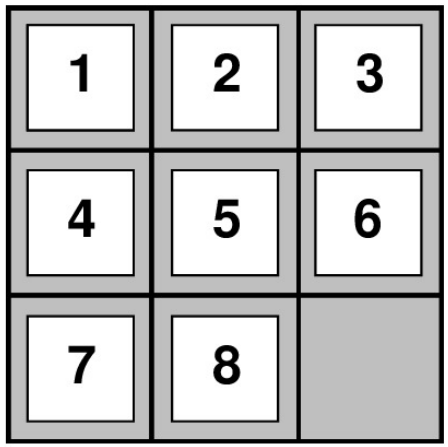
Goal State

states: ?
actions: ?
goal test: ?
path cost: ?

Example: El 8-puzzle II



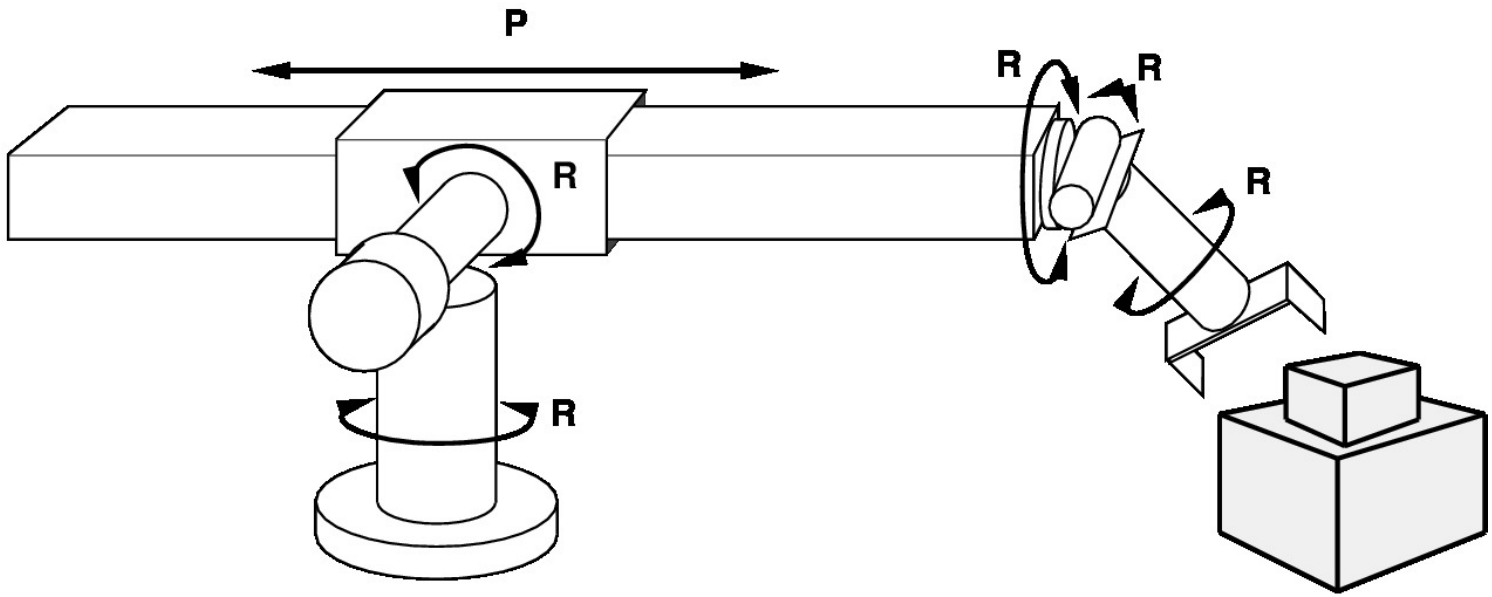
Start State



Goal State

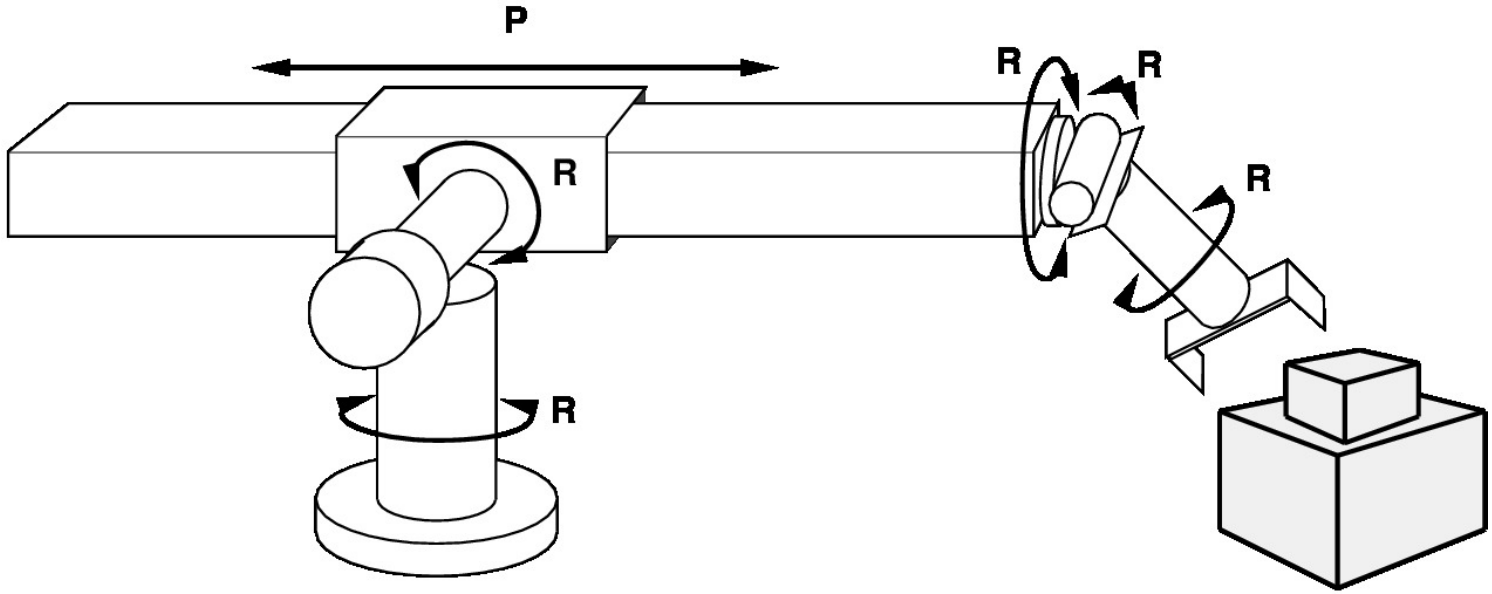
states: integer locations of tiles (ignore intermediate positions)
actions: move blank left, right, up, down (ignore unjamming etc.)
goal test: goal state (given)
path cost: 1 per move
[Note: optimal solution of *n*-Puzzle family is NP-hard]

Ejemplo: robot ensamblador



states:
actions:
goal test:
path cost:

Example: robot ensamblador



states: real-valued coordinates of robot joint angles parts of the object to be assembled
actions: continuous motions of robot joints
goal test: complete assembly *with no robot included!*
path cost: time to execute

El problema de los misioneros y los caníbales I

- Tres **misioneros** y tres **caníbales** están a un lado del **río**, junto a un **bote**.
- El bote puede **transportar** una o dos **personas** (y obviamente no puede desplazarse al otro lado con cero personas).
- El **objetivo** es transportar **a todo el mundo** al otro **lado**, sin dejar nunca un grupo de misioneros **superado en número** por los caníbales.
- La tarea es **formularlo** como un **problema** de búsqueda.

El problema de los misioneros y los caníbales II

1. Definir una representación de **estado**.
2. Dar los estados inicial y **objetivo** en esta representación.
3. Definir la función de **sucesores** en esta **representación**.
4. ¿Cuál es la **función** de **coste** en tu función **sucesor**?
5. ¿Cuál es el **número total** de estados **alcanzables**?

Problema de los exploradores

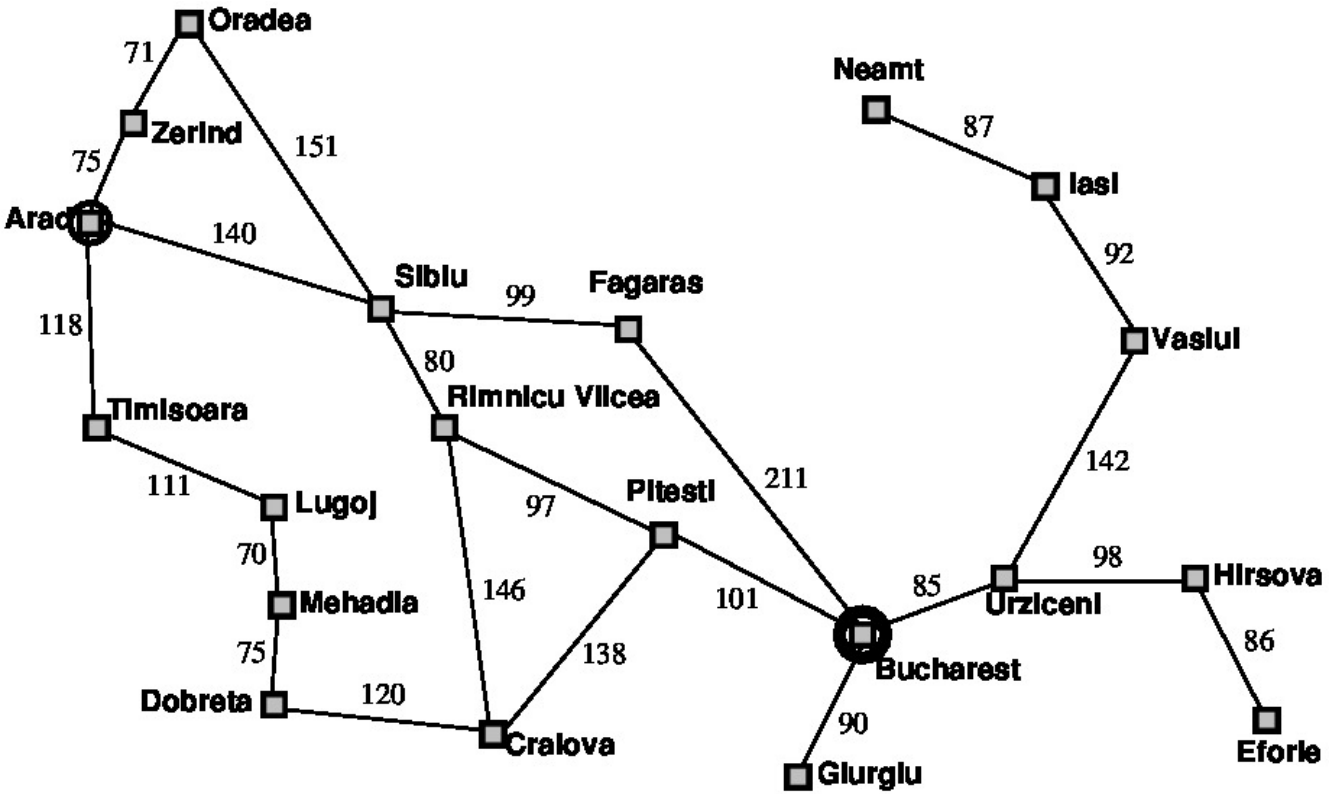
Cuatro exploradores: Alex, Brook Chris and Dusty necesitan cruzar un río en un pequeño bote que sólo puede transportar un peso de 100kg. Si Alex pesa 90kg, Brook pesa 80kg, Chris pesa 60kg, Dusty pesa 40kg y además llevan 20kg de comida y material. ¿Cómo podrían cruzar el río?

1. El estudiante debe definir el espacio de estados para este caso.
2. El estudiante debe definir el problema.

Contents

Ejemplo Buscar Ruta

- ¿Elementos del problema?
- ¿Frontera (frontier, fringe), estados explorados (expandidos, visitados) y no explorados?.



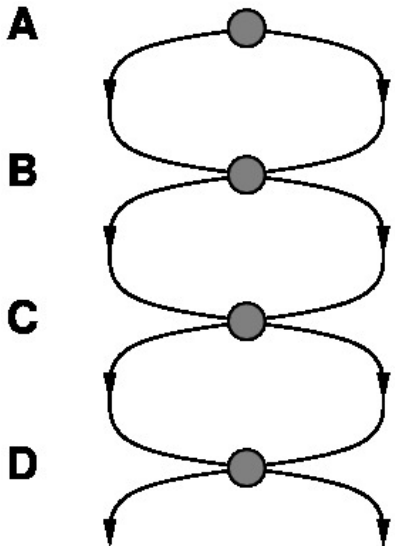
Algoritmos de búsqueda en Árbol

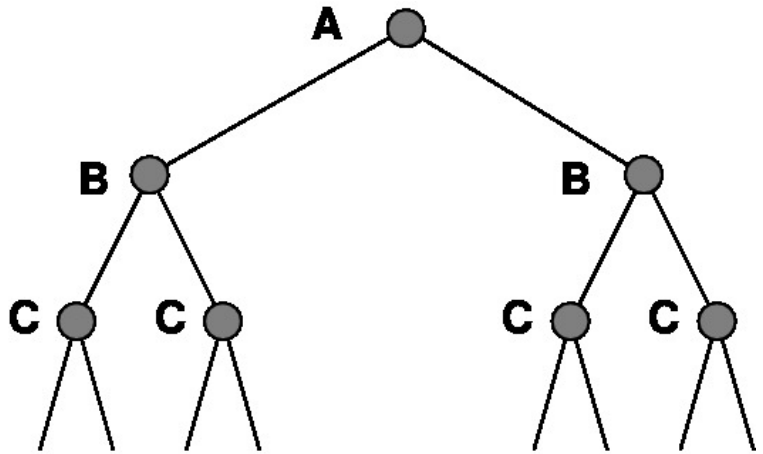
- Idea básica: offline, búsqueda en el espacio de estados generando sucesores de estados ya explorados (a.k.a. estados *expandidos* o *visitados*)

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

Estados repetidos

- Si se falla en la **detección** de **estados** repetidos un problema lineal puede volverse **exponencial**.



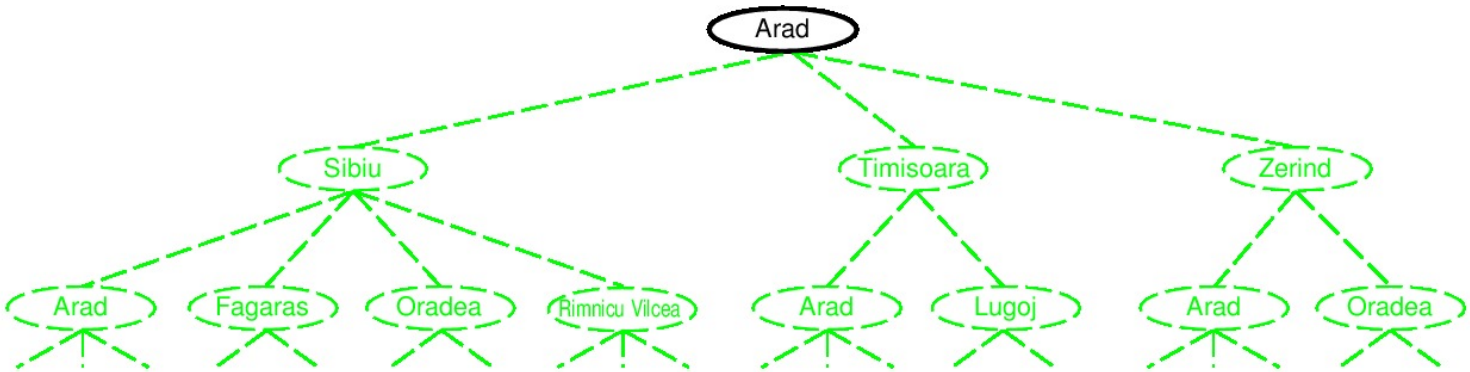


Búsqueda en Grafo

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

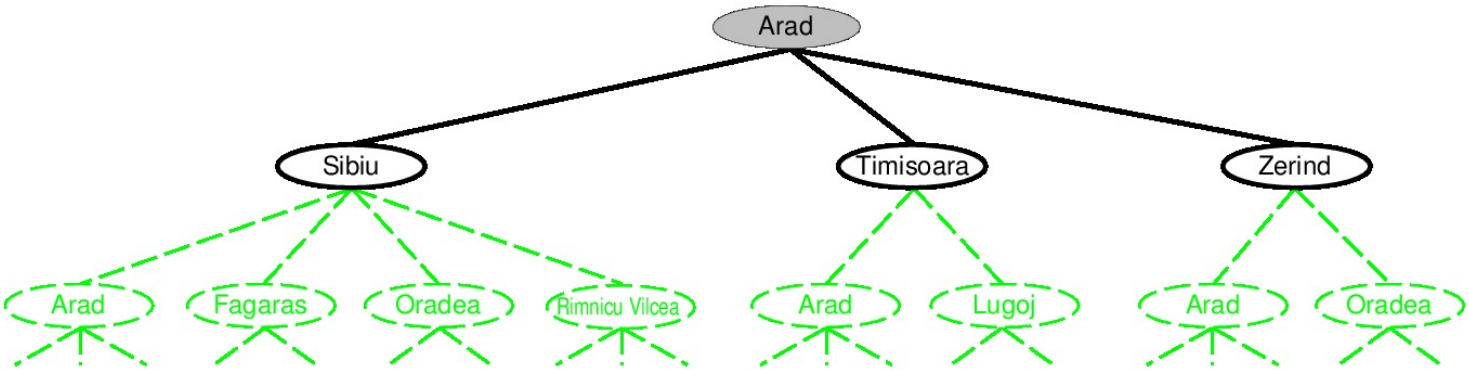
closed (visitados) ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
loop do
if *fringe* is empty **then return** failure
node ← REMOVE-FRONT(*fringe*)
if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*
if STATE[*node*] is not in *closed* **then**
add STATE[*node*]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

Ejemplo de árbol de búsqueda I



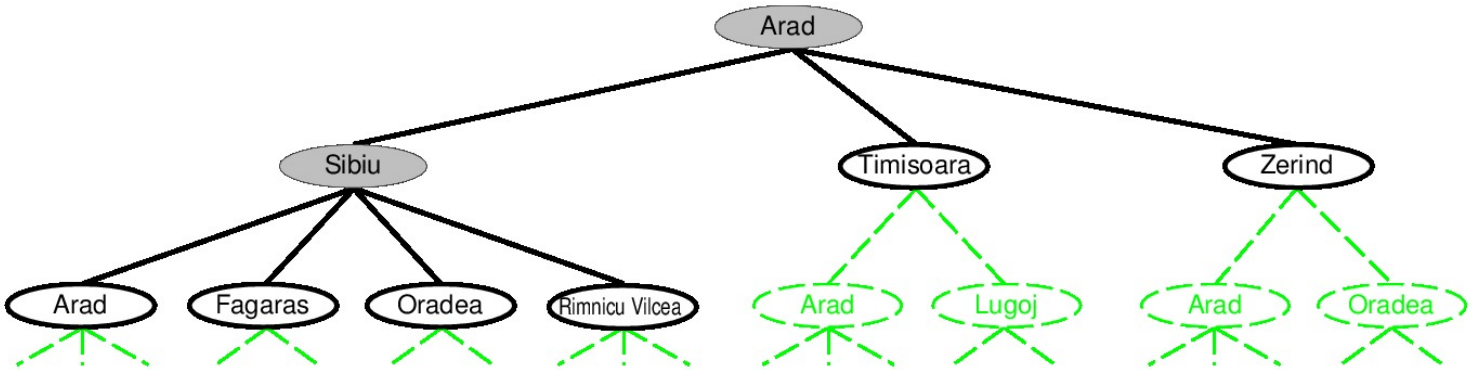
Frontera	Arad

Ejemplo de árbol de búsqueda II



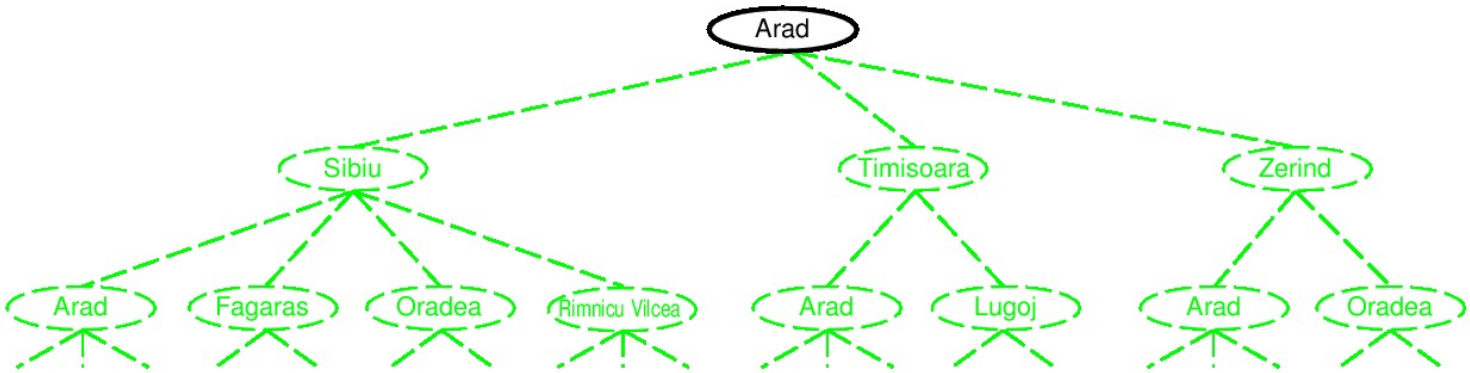
Frontera	Sibiu	Timisoara	Zerind

Ejemplo de árbol de búsqueda III



Frontera	Timisoara	Zerind	Arad	Fagaras	Oradea	Rimn. V.

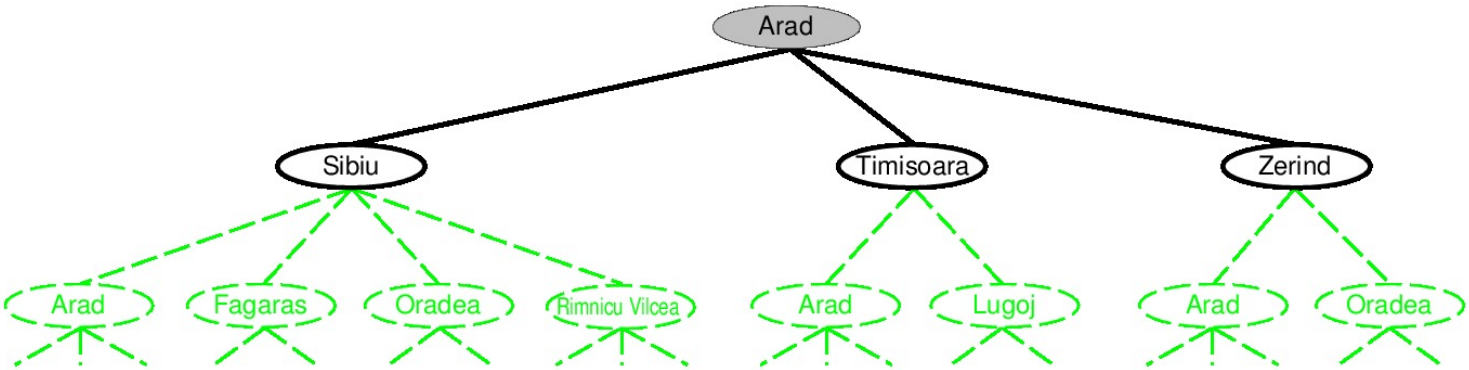
Ejemplo de búsqueda en Grafo (Errónea) I



Frontera	Arad

Visitados	

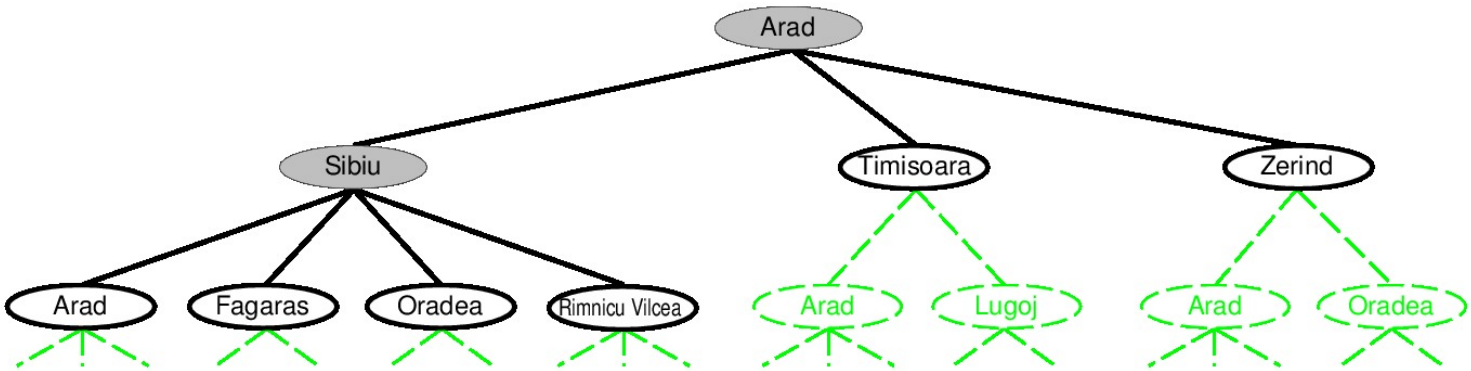
Ejemplo de búsqueda en Grafo (Errónea) II



Frontera	Sibiu	Timisoara	Zerind

Visitados	Arad

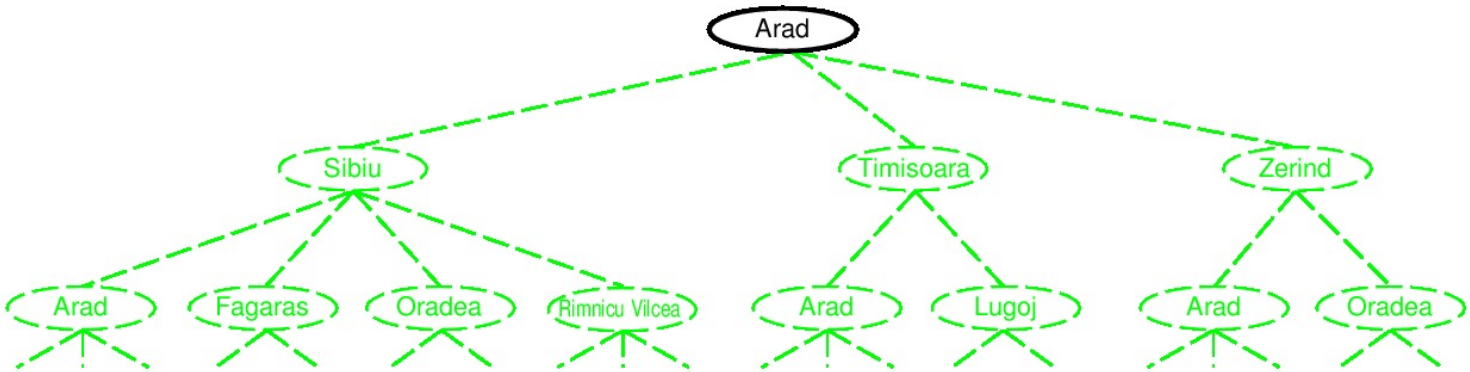
Ejemplo de búsqueda en Grafo (Errónea) III



Frontera	Timisoara	Zerind	Fagaras	Oradea	Rimnicu Vlicea

Visitados	Arad	Sibiu

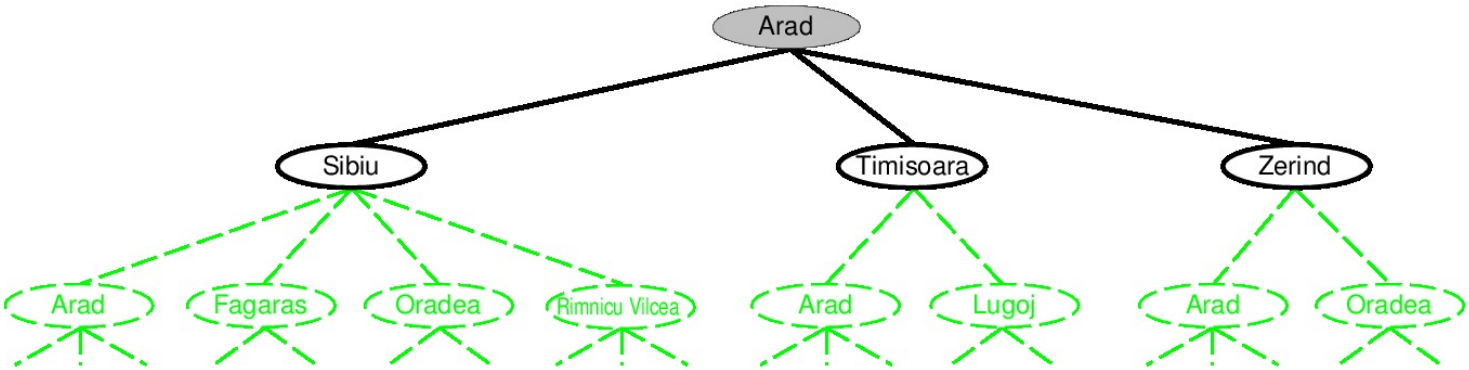
Ejemplo de búsqueda en Grafo I



Frontera	Arad

Visitados	

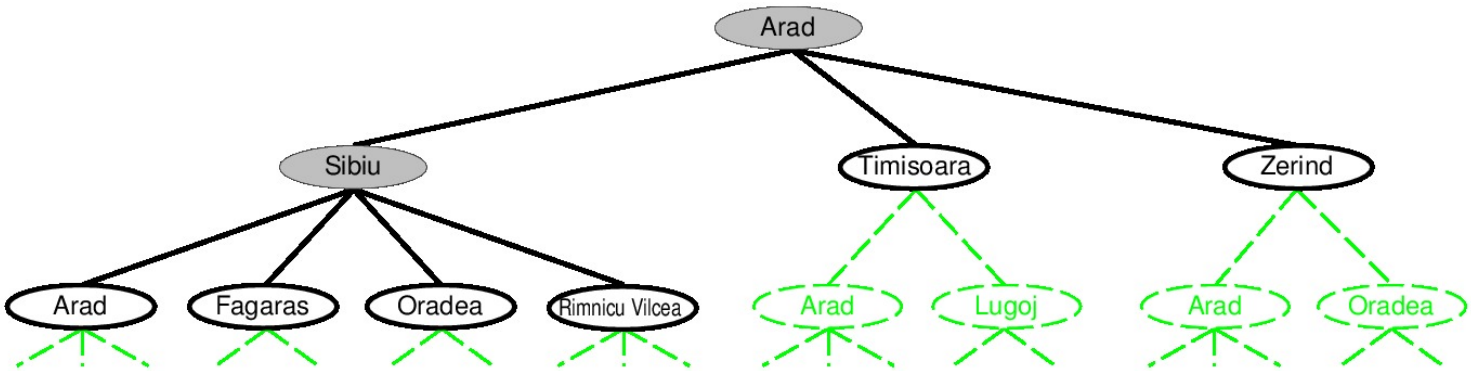
Ejemplo de búsqueda en Grafo II



Frontera	Sibiu	Timisoara	Zerind

Visitados	Arad

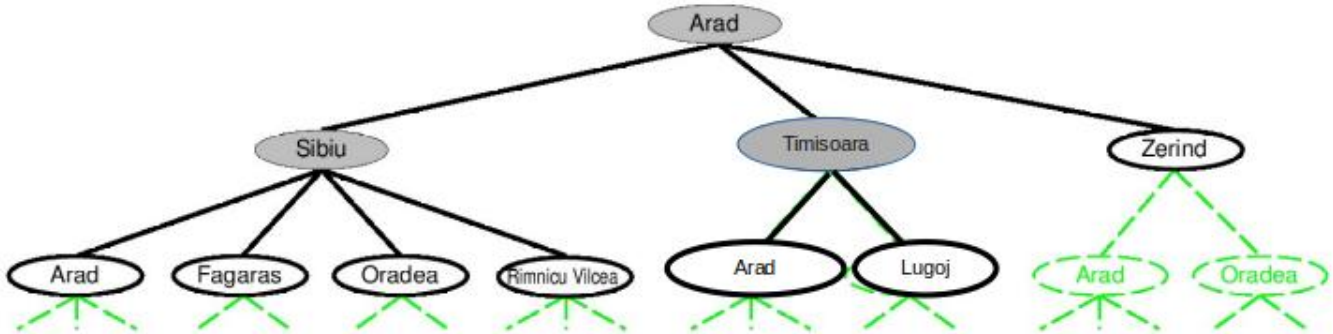
Ejemplo de búsqueda en Grafo III



Frontera	Timisoara	Zerind	Arad	Fagaras	Oradea	R. Vlicea

Visitados	Arad	Sibiu

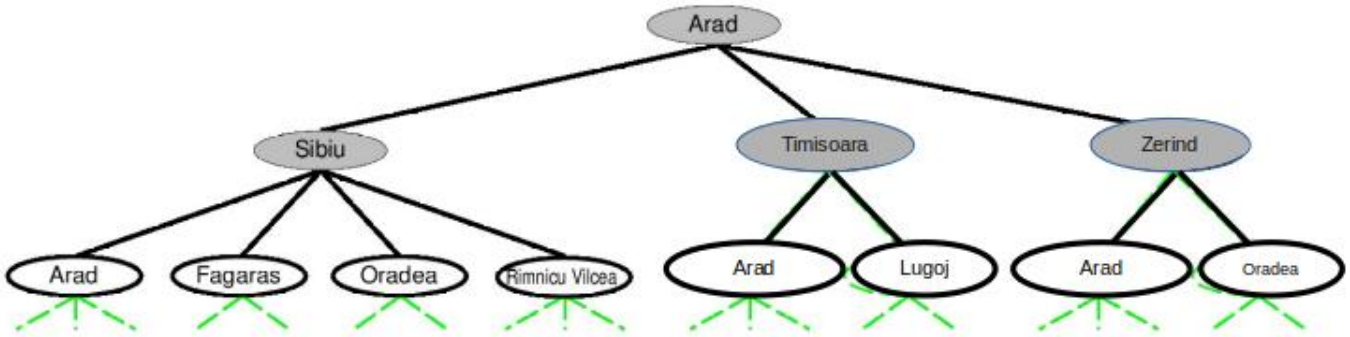
Ejemplo de búsqueda en Grafo IV



Zerind	Arad	Fagaras	Oradea	R. Vlicea	Arad	Lugoj

Visitados	Arad	Sibiu	Timisoara

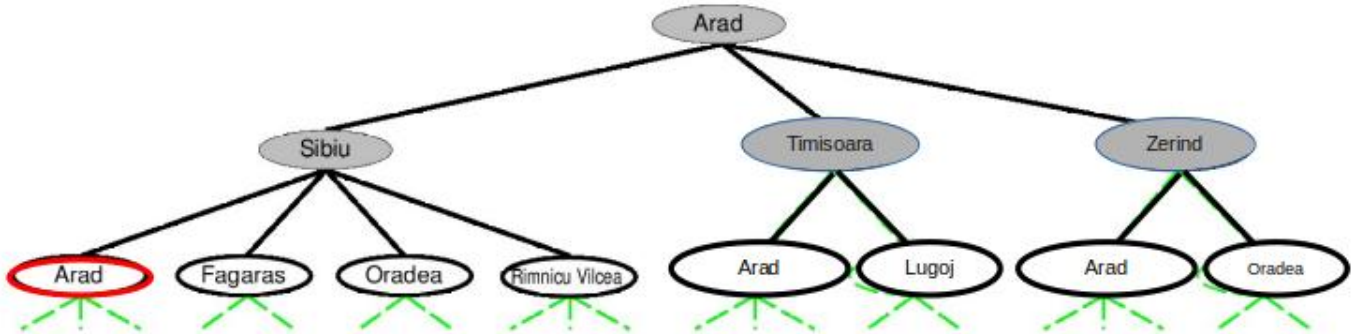
Ejemplo de búsqueda en Grafo V



Arad	Fagaras	Oradea	R. Vlicea	Arad	Lugoj	Arad	Oradea

Visitados	Arad	Sibiu	Timisoara	Zerind

Ejemplo de búsqueda en Grafo VI



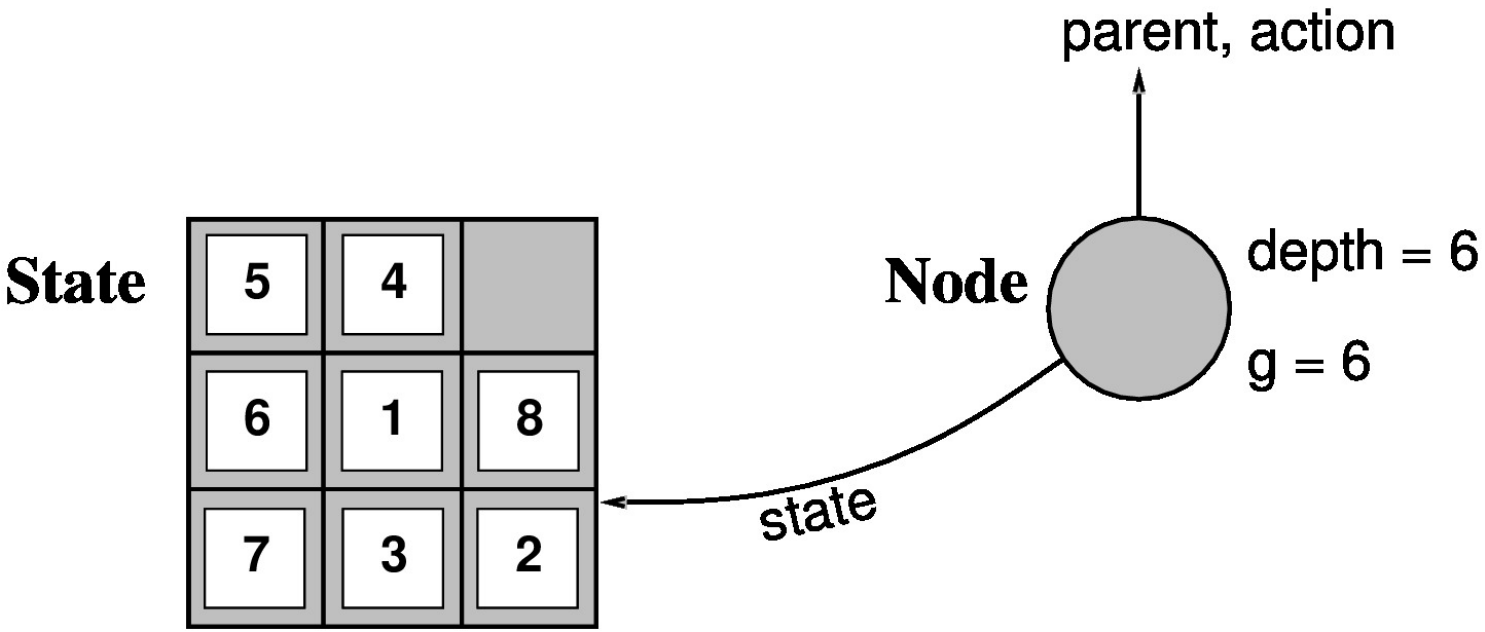
Fagaras	Oradea	R. Vlicea	Arad	Lugoj	Arad	Oradea

Visitados	Arad	Sibiu	Timisoara	Zerind

Implementación: estados vs. nodos I

- Un *estado* es una (representación de) una **configuración física**
- Un *nodo* es una estructura de datos parte **constitutiva** de un árbol de búsqueda e incluye:
  - *padre* y **acción** ejecutada por el padre que lo genera.
  - *profundidad*
  - *coste del camino* $g(x)$
- Los estados **no** tienen **padres**, hijos, **profundidad**, o coste del camino!

Implementación: estados vs. nodos II



- La **función de expansión** crea nuevos nodos, **rellenando** los campos del nodo y usando la función **Sucesores** del **problema** para crear los correspondientes **estados**.

Versión del Algoritmo a Implementar I

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

closed (visitados) ← 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

Versión del Algoritmo a Implementar II

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(STATE[*node*], *action*, *result*)
    DEPTH[*s*] ← DEPTH[*node*] + 1
    add *s* to *successors*
  end for
return *successors*

Cuestiones Pendientes I

Hasta ahora se plantean gran cantidad de **conceptos** nuevos, **estructuras**, ideas, etc. que quedan algo **inconexas** y que deben generar al alumno una serie de **dudas o cuestiones**.

- ¿Por qué el **algoritmo** permite **añadir** estados ya **visitados** en la frontera? ¿No sería más eficiente comprobar si están en la lista de **visitados** antes de añadirlos?
- ¿Si realmente tengo ya mi **estructura** de **árbol** con nodos que tienen un puntero o **referencia** al padre? ¿Para qué necesito la **frontera**?

Cuestiones Pendientes II

- ¿Por qué el **algoritmo** sí especifica que hay que **tomar** (y **eliminar**) el **primer** elemento de la frontera pero **no especifica** donde se deben insertar estos elementos?
- ¿Por qué el algoritmo **testea** si un nodo contiene un **estado** que satisface la **función objetivo** sólo cuando lo extrae de la **frontera** y no cuándo lo introduce? ¿No sería esto **último** mucho más **eficiente**?

Cuestiones Pendientes III

- ¿Cuál de los **dos algoritmos** vamos a usar **finalmente**? ¿La búsqueda en **grafo** o la búsqueda en **árbol**?
- ¿Por qué habla de **algoritmos de búsqueda** y no de **algoritmo de búsqueda** si sólo vamos a utilizar el algoritmo de búsqueda en grafo?
- Si tengo que resolver un problema de **búsqueda**, ¿Tengo que **especificar** todos los **campos** de cada **nodo** del árbol?. En este caso, ¿Qué **formato** de nodo de búsqueda o qué **campos** deben aparecer?

Cuestiones Pendientes IV

Por ahora sólo se pueden resolver estas cuestiones:

- ¿Por qué el **algoritmo** permite **añadir** estados ya **visitados** en la frontera? ¿No sería más eficiente comprobar si están en la lista de **visitados** antes de añadirlos?
  Si hacemos comprobaciones al introducir en la frontera hay que hacerla **para todos los sucesores**. Esta consulta, si la lista de visitados tiene muchos elementos (como ocurre en este tipo de problemas) puede ser muy **costosa temporalmente**. Si la hacemos al extraer de la frontera sólo se hace una única consulta. Además, para algunas estrategias (¿Qué es una estrategia?) los nodos con estados repetidos que se añaden a la frontera "suelen quedar en posiciones finales"por lo que difícilmente llegan a salir de la frontera y realmente esa única consulta **ni se llega a realizar**.

Cuestiones Pendientes V

- ¿Si realmente tengo ya mi **estructura** de **árbol** con nodos que tienen un puntero o **referencia** al padre? ¿Para qué necesito la **frontera**?
  Nos faltan las **primitivas** para **generar** y **recorrer** el árbol, por lo que se necesita de una estructura auxiliar que permite no sólo acceder a los nodos sino además poder establecer **diferentes criterios de orden** para la generación y/o exploración del árbol.

Cuestiones Pendientes VI

- ¿Cuál de los **dos algoritmos** vamos a usar **finalmente**? ¿La búsqueda en **grafo** o la búsqueda en **árbol**?
  El que incorpora **control de estados visitados** o explorados.
  Es decir la búsqueda en Grafo. Si no es así problemas triviales pueden llegar a ser **muy complejos** de resolver.

Estrategias de búsqueda I

Estrategia

Una estrategia se define seleccionando el **orden de expansión** del nodo

Las estrategias tienen que ser evaluadas desde las siguientes dimensiones:

- *Completitud*: ¿Encuentra **siempre** una **solución** si esta existe?
- *Complejidad temporal*: número de **nodos** generados/**expandidos**
- *Complejidad espacial*: máximo **número** de nodos en **memoria**
- *Optimalidad*: ¿Encuentra **siempre** la **solución** de menor coste?

Estrategias de búsqueda II

La complejidad espacial y temporal se mide en términos de:

- b: máximo **factor** de ramificación del **árbol** de búsqueda.
- d: **Profundidad** de la **solución** con menor coste.
- m: **Profundidad** máxima del **espacio de estados** (puede ser ∞)

Estrategias de búsqueda Sin Información

Estrategias no informadas

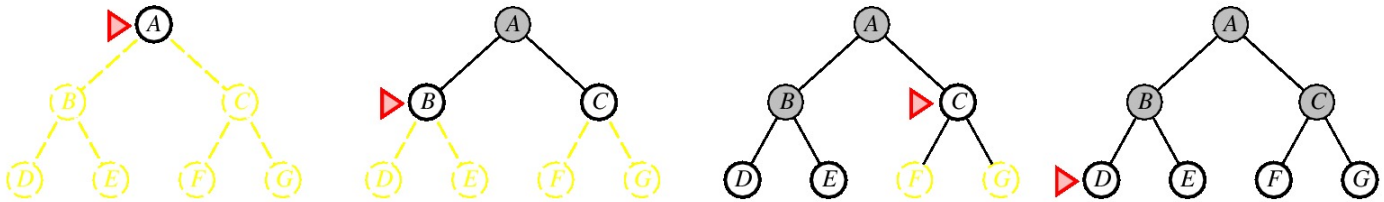
Sólo utilizan la información disponible en la definición del problema.

- Primero en **Anchura**
- Primero en **Profundidad**
- Profundidad **Acotada**
- Profundidad **Iterativa**
- Costo **Uniforme**

Primero en Anchura I

- Expandir el nodo no visitado **más superficial** (camino más corto)
- *Implementación*: frontera es una cola **FIFO**, i.e., los nuevos **sucesores** van al final.

Primero en Anchura II



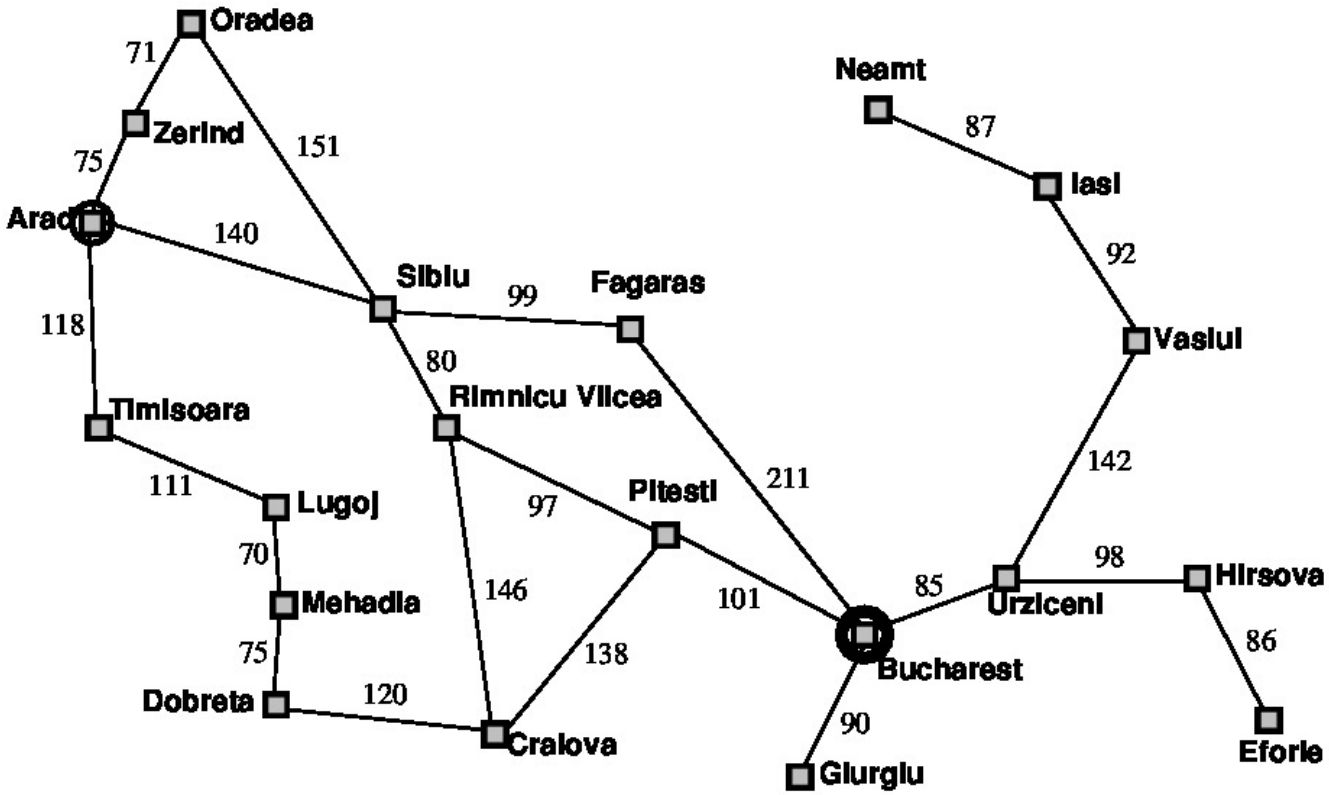
Frontera	A

Frontera	B	C

Frontera	C	D	E

Frontera	D	E	F	G

Ejemplo: Primero en Anchura



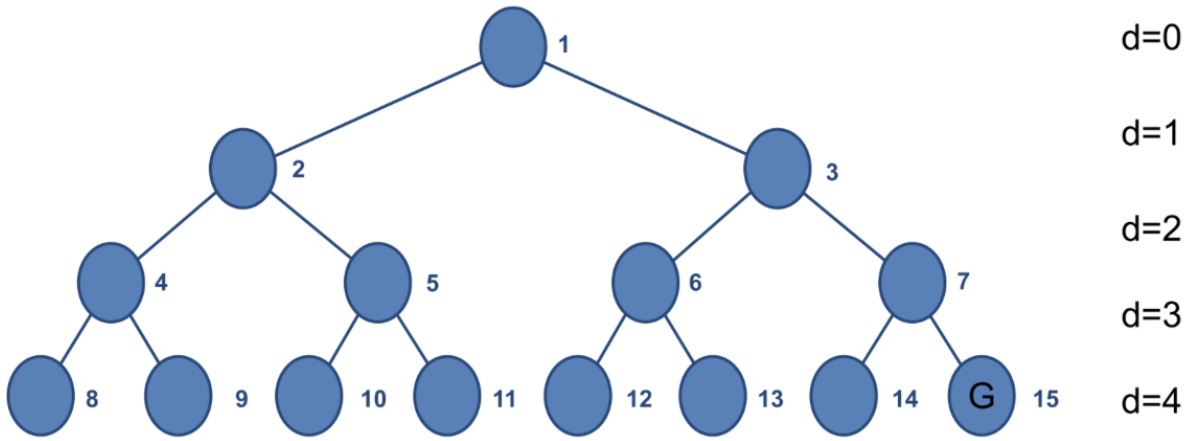
Propiedades de primero en anchura I

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

- **Completo**: Si (si b es finito)
- **Tiempo**: $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. en d
- **Espacio**: $O(b^{d+1})$ (guarda todos los nodos en memoria)
- **Óptimo**: Si (Si el coste $= 1$ por paso); no óptimo en general.
- *Espacio* es el mayor problema; puede fácilmente generar nodos a 100MB/seg por tanto 24hrs $=$ 8640GB.

Propiedades de primero en anchura II

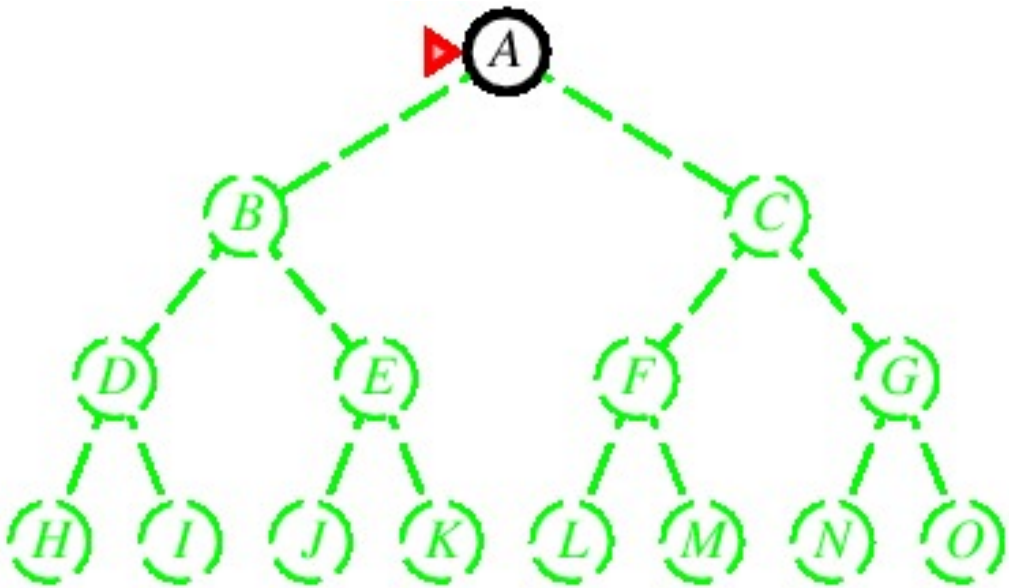
- En el análisis anterior se asume la peor situación posible que sería la siguiente (solución en el nodo más profundo a la derecha del árbol):



Primero en profundidad I

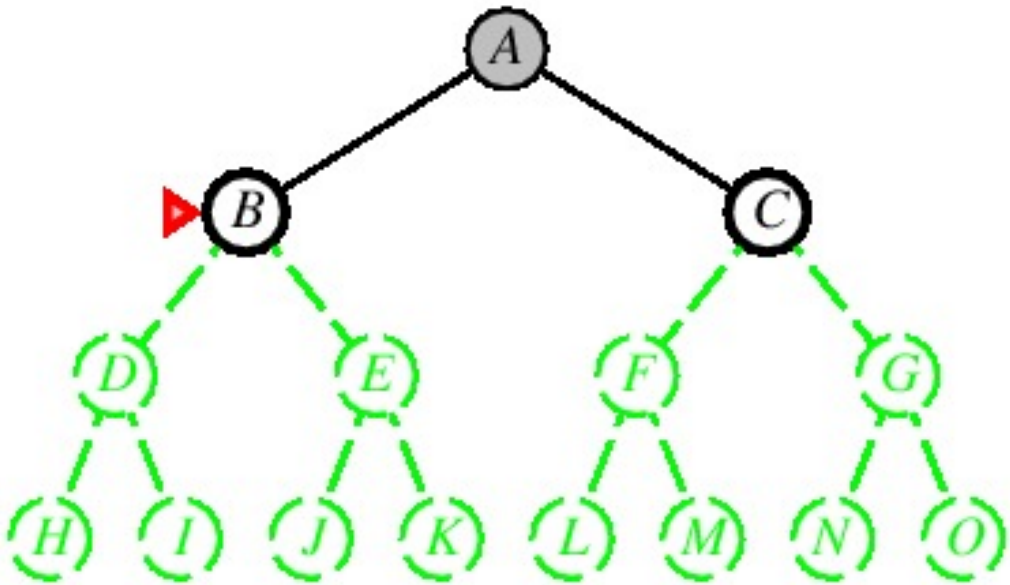
- Expande el nodo no expandido **más profundo**
- *Implementación*: *frontera* = cola **LIFO**, i.e., pone **sucesores** al inicio.

Primero en profundidad II



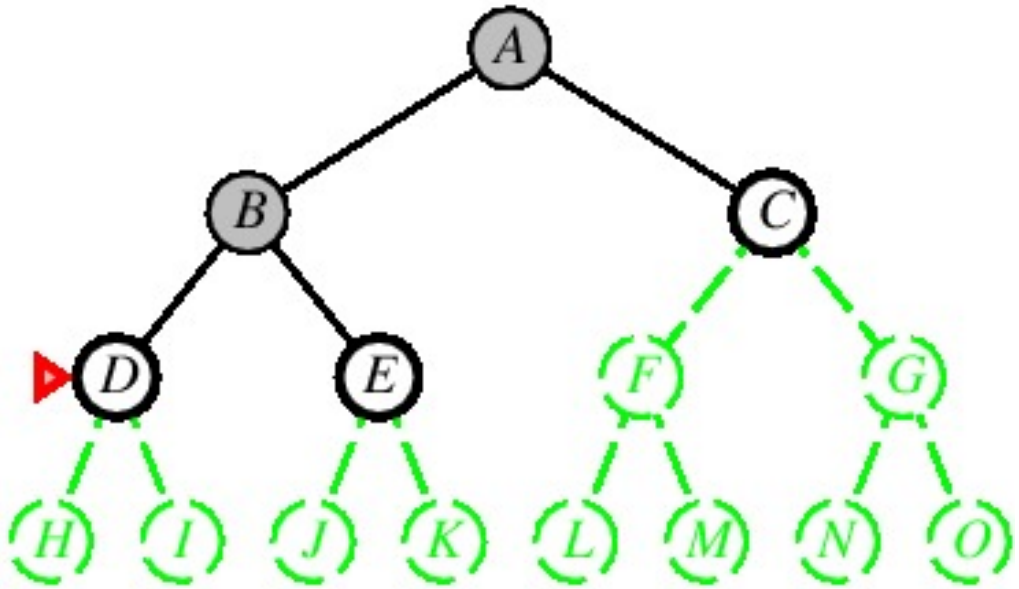
Frontera	A

Primero en profundidad III



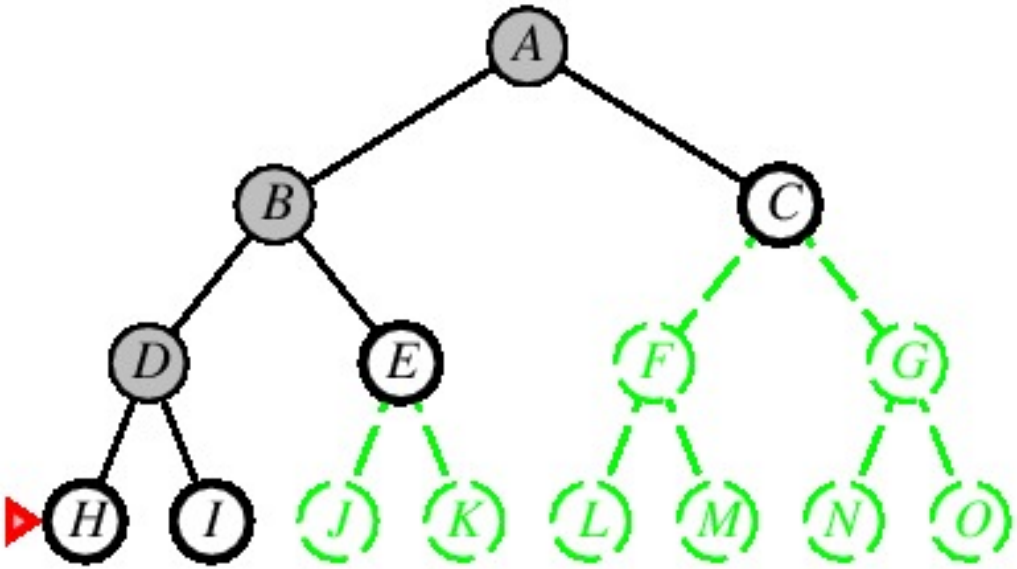
Frontera	B	C

Primero en profundidad IV



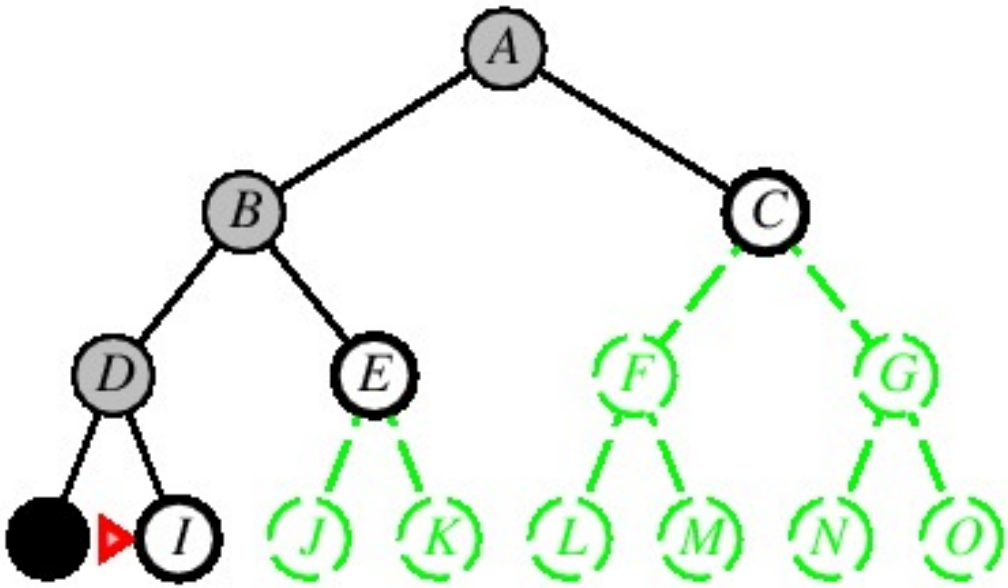
Frontera	D	E	C

Primero en profundidad V



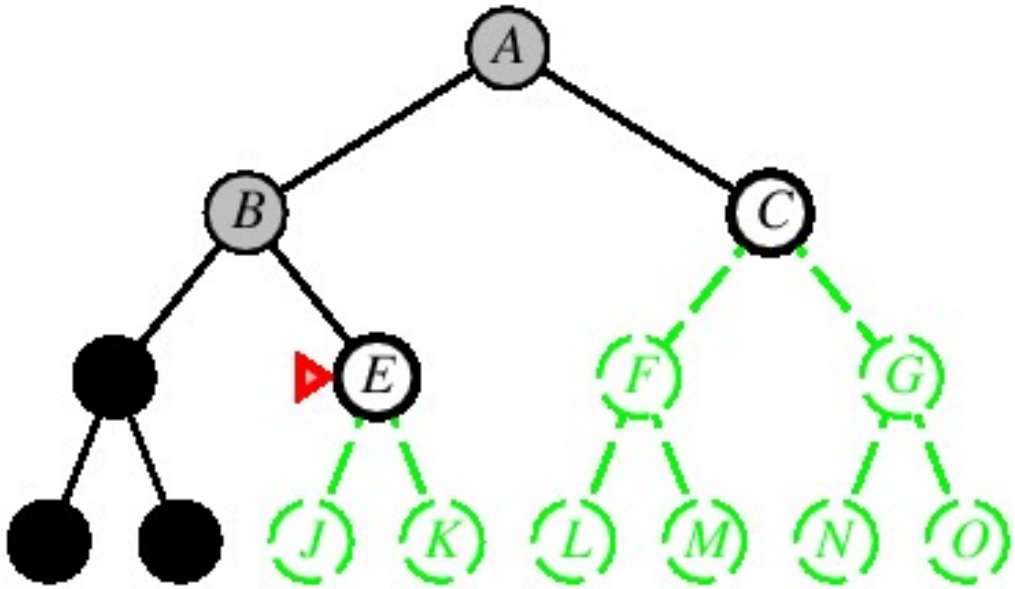
Frontera	H	I	E	C

Primero en profundidad VI



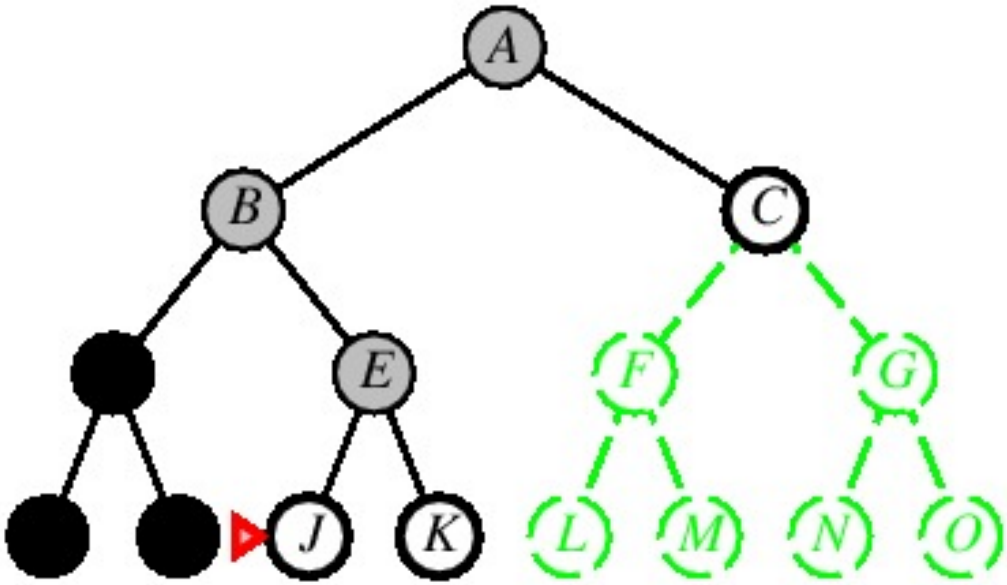
Frontera	I	E	C

Primero en profundidad VII



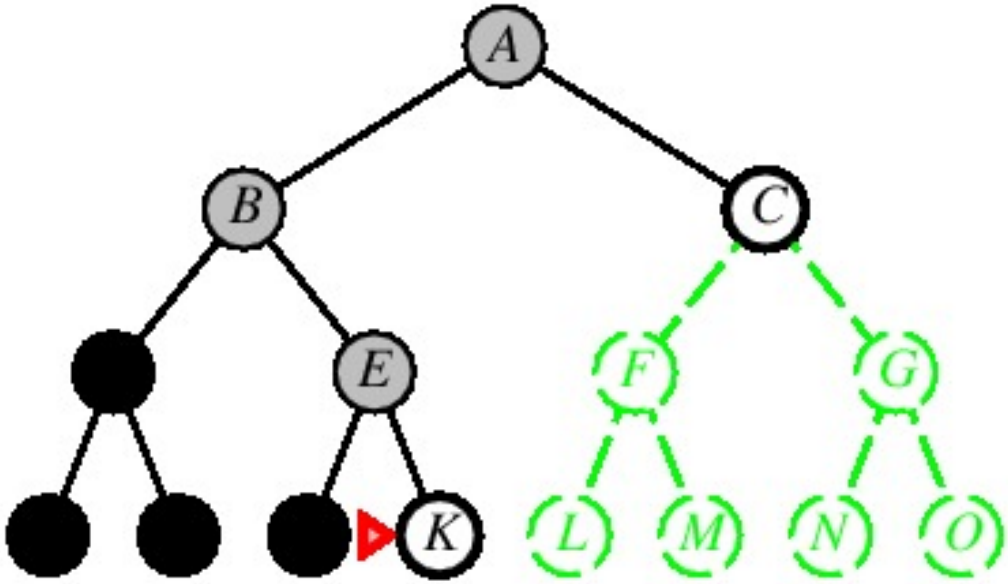
Frontera	E	C

Primero en profundidad VIII

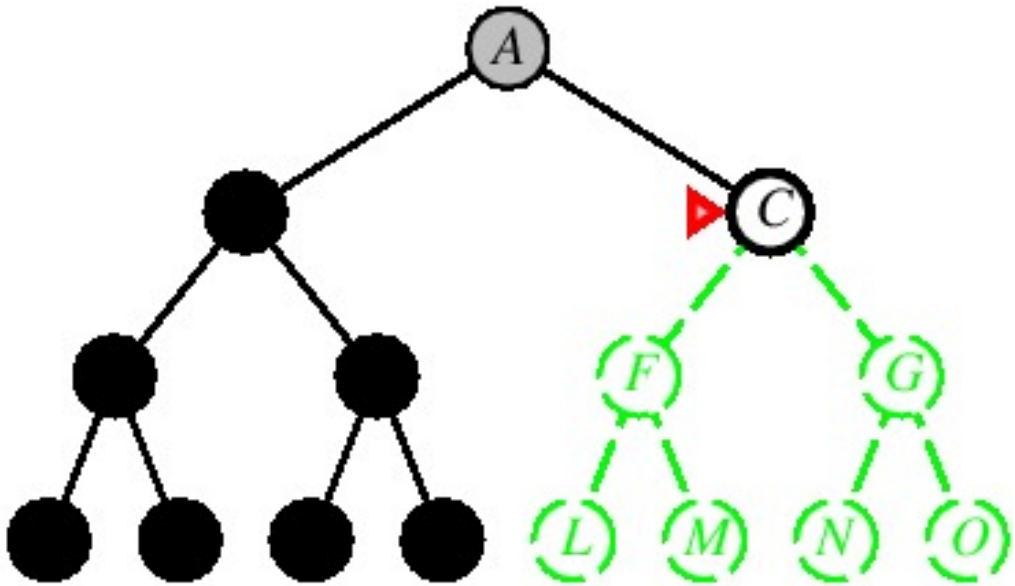


Ejercicio: Simula la frontera durante el resto de la búsqueda

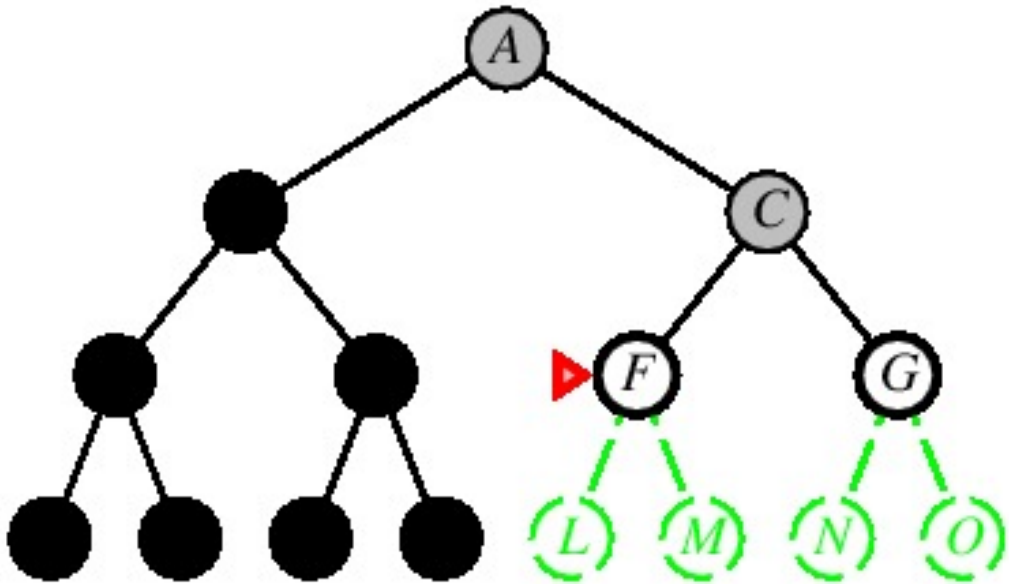
Primero en profundidad IX



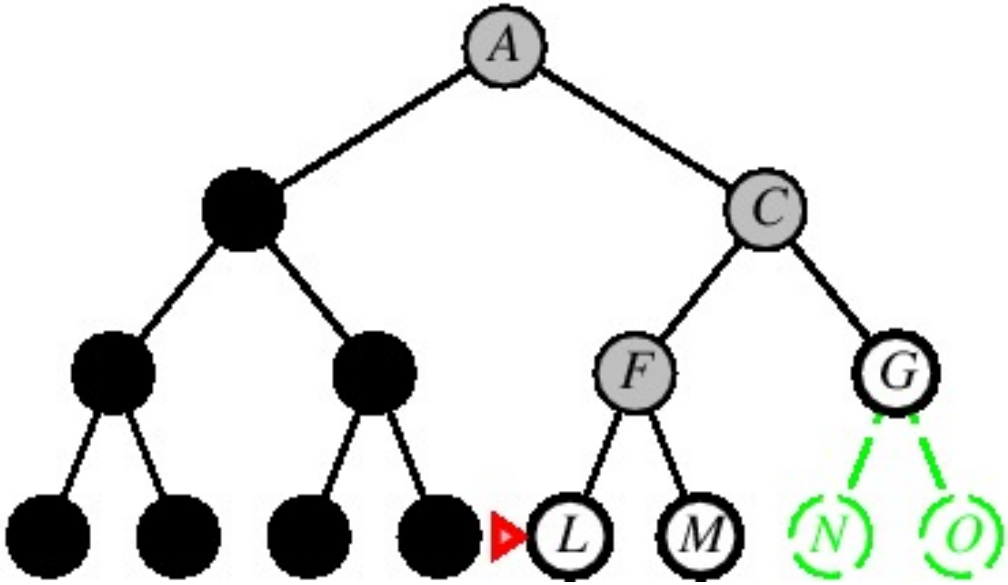
Primero en profundidad X



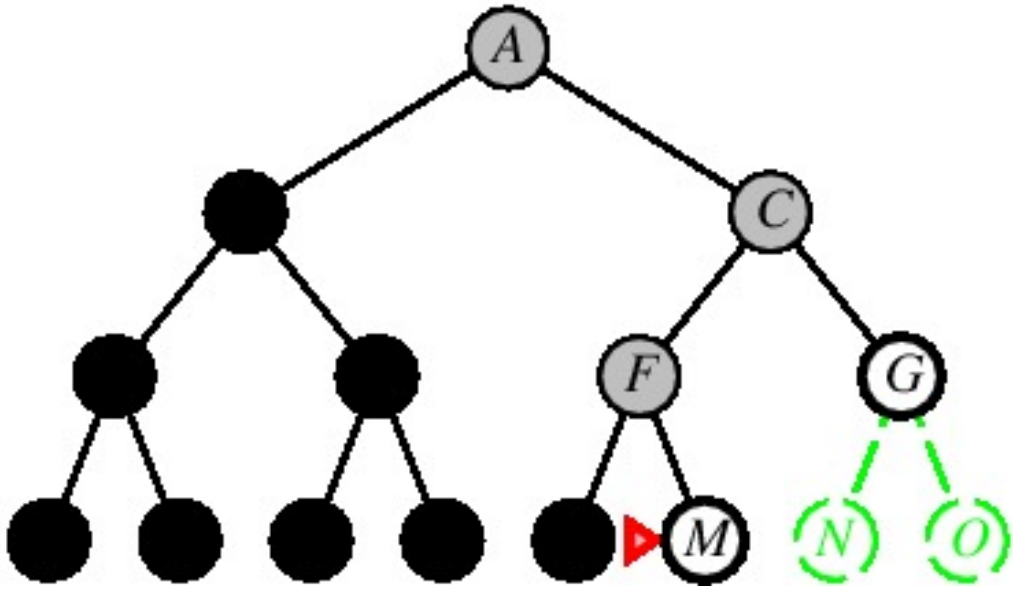
Primero en profundidad XI



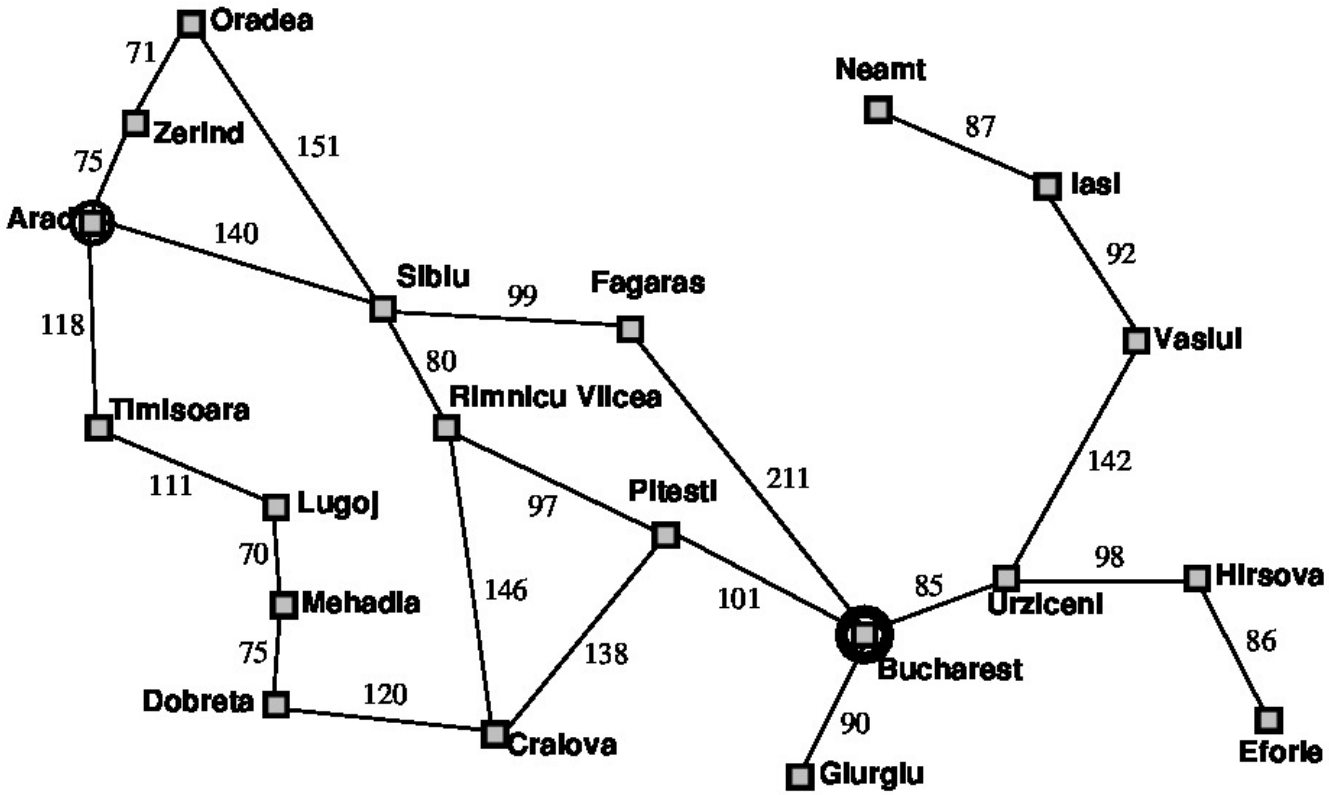
Primero en profundidad XII



Primero en profundidad XIII

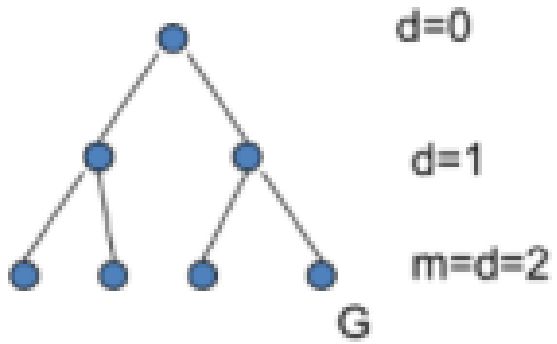


Ejemplo: primero en profundidad



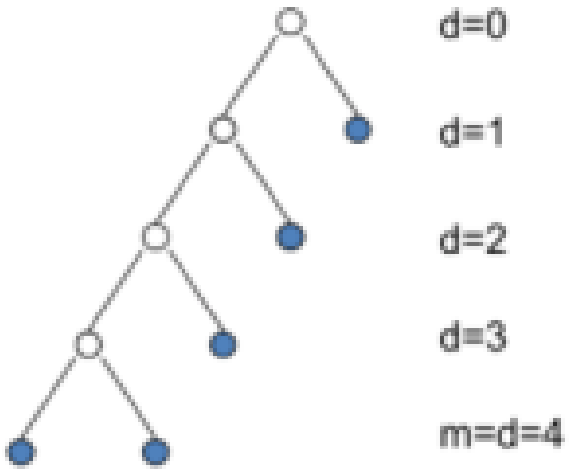
Propiedades de primero en profundidad I

- **Completa**: No, **cae** en espacios con **profundidad infinita**, espacios con ciclos. Si se modifica para evitar estados repetidos en el camino $\Rightarrow$ **completa en espacios finitos**.
- **Tiempo**: $O(b^m)$: terrible si m es mucho más **grande** que d pero si las **soluciones** son profundas, puede ser mucho **más rápido** que primero en anchura.



Propiedades de primero en profundidad II

- **Espacial**: $O(bm)$, i.e., **Lineal!**



- **Óptimo**: **No**

Profundidad acotada

- Como **primero en profundidad** con un límite de profundidad *l*, i.e., los nodos en profundidad *l* no tienen **sucesores**.

Prof. Acotada: Problemas con la Poda "Booleana" I

- La profundidad máxima **limita** el máximo número de acciones que un **agente puede ejecutar** para alcanzar un estado que satisfaga la **función objetivo**.
- Por ejemplo, si l=4 siendo el estado inicial Arad y el estado objetivo Bucharest, hay dos posibles soluciones: (**Ir a Sibiu, Ir a Fagaras, Ir a Bucharest**) e (**Ir a Sibiu, Ir a RV, Ir a Pitesti, Ir a Bucharest**) con profundidad=3 y profundidad=4, respectivamente.

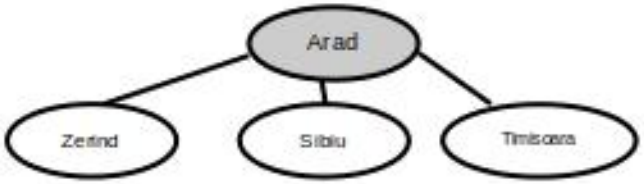
Prof. Acotada: Problemas con la Poda "Booleana" II

- Por tanto, si se ejecuta el algoritmo de búsqueda con **prof. acotada** y l=4, se debería **alcanzar una de las dos soluciones**.
- Pero, ¿Qué ocurriría si el **orden de exploración** de los sucesores de Arad es **Zerind, Sibiu y Timisoara**?
- **Simula** el algoritmo de búsqueda con control de **estados visitados**.

Frontera	Arad (0)

Visitados	

Prof. Acotada: Problemas con la Poda "Booleana" III



Frontera	Zerind(1)	Sibiu(1)	Timisoara(1)

Visitados	Arad

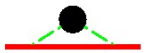
Profundidad iterativa incremental

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

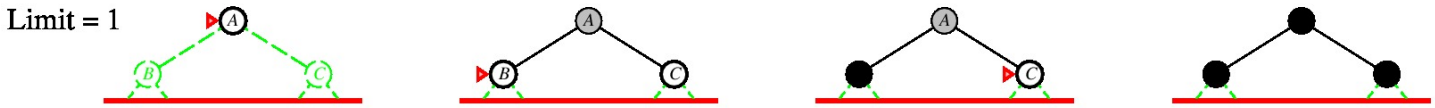
  for *depth* $\leftarrow 0$ **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*
end

Profundidad iterativa incremental $l = 0$

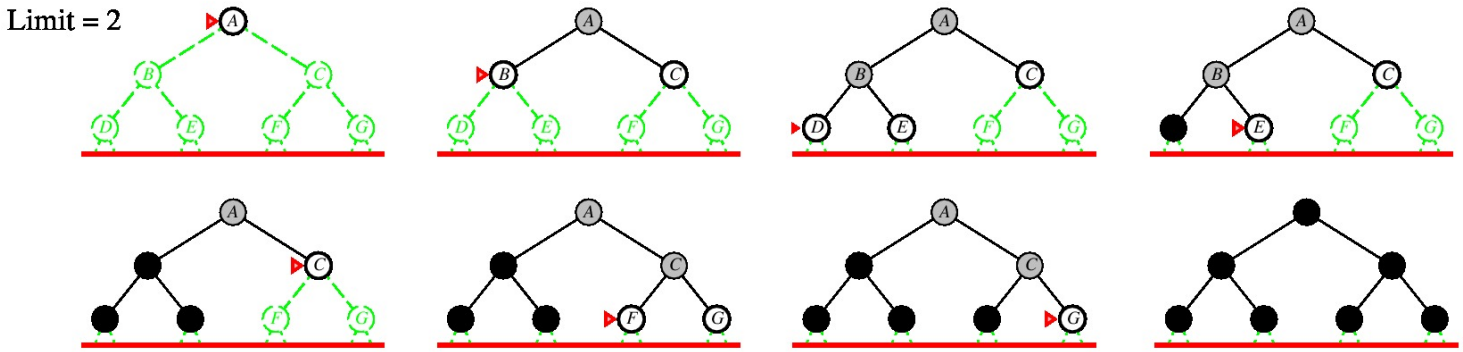
Limit = 0



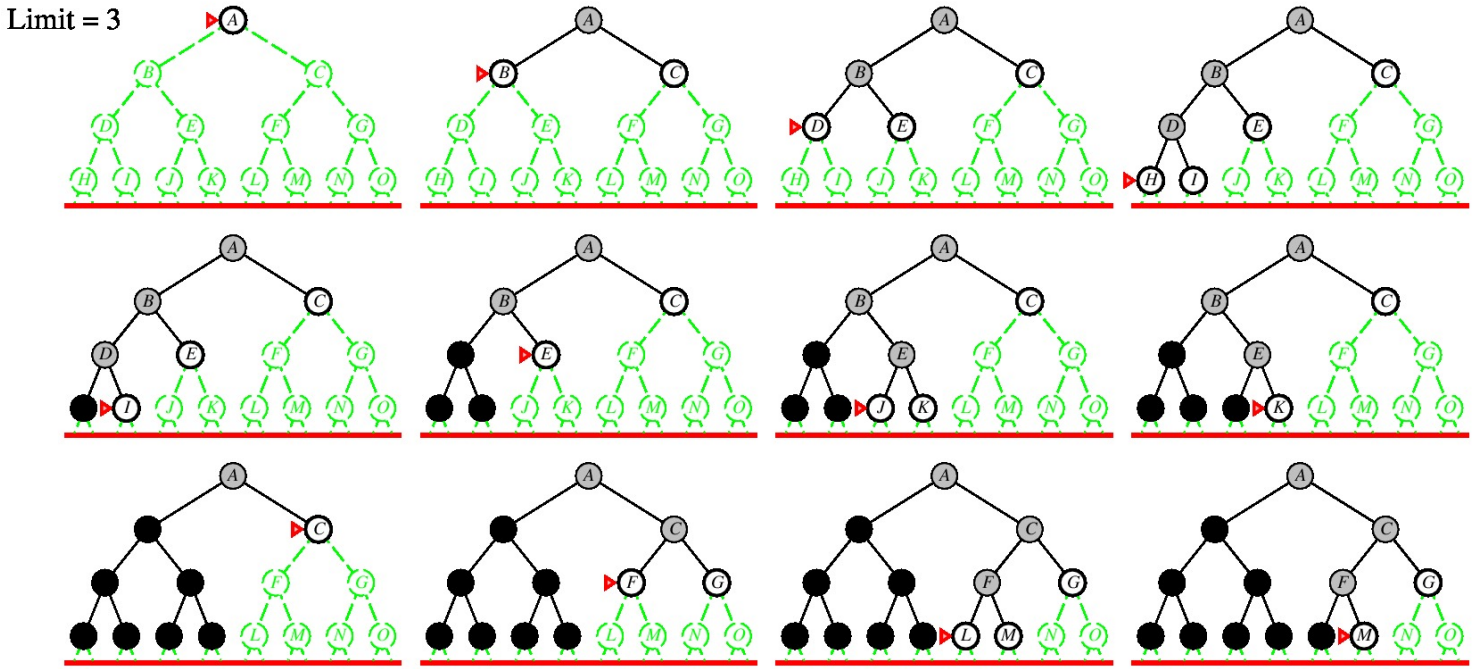
Profundidad iterativa incremental $l = 1$



Profundidad iterativa incremental $l = 2$



Profundidad iterativa incremental $l = 3$



Propiedades de Profundidad iterativa incremental I

- A nivel de complejidad tiene las **ventajas de Anchura y Profundidad**. La complejidad temporal va en función de d ya que siempre encuentra la solución *menos profunda* y a nivel espacial puede dejar de almacenar partes del árbol **ya exploradas**.
- **Completa**: Si
- **Tiempo**: $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$. Aquí es d y no $d+1$ porque en esta estrategia al no generar los sucesores a profundidad mayor que la cota d no tiene que almacener ni generar los mismos.
- **Espacial** $O(bd)$
- **Óptima** Si, si coste de cada paso $= 1$.

Propiedades de Profundidad iterativa incremental II

- Comparación **numérica** para $b = 10$ y $d = 5$, **solución** en la hoja derecha más lejana:

$$\begin{aligned} N(IDS) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(BFS) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

- IDS lo completa **mejor** porque otros **nodos** a profundidad d no son **expandidos**

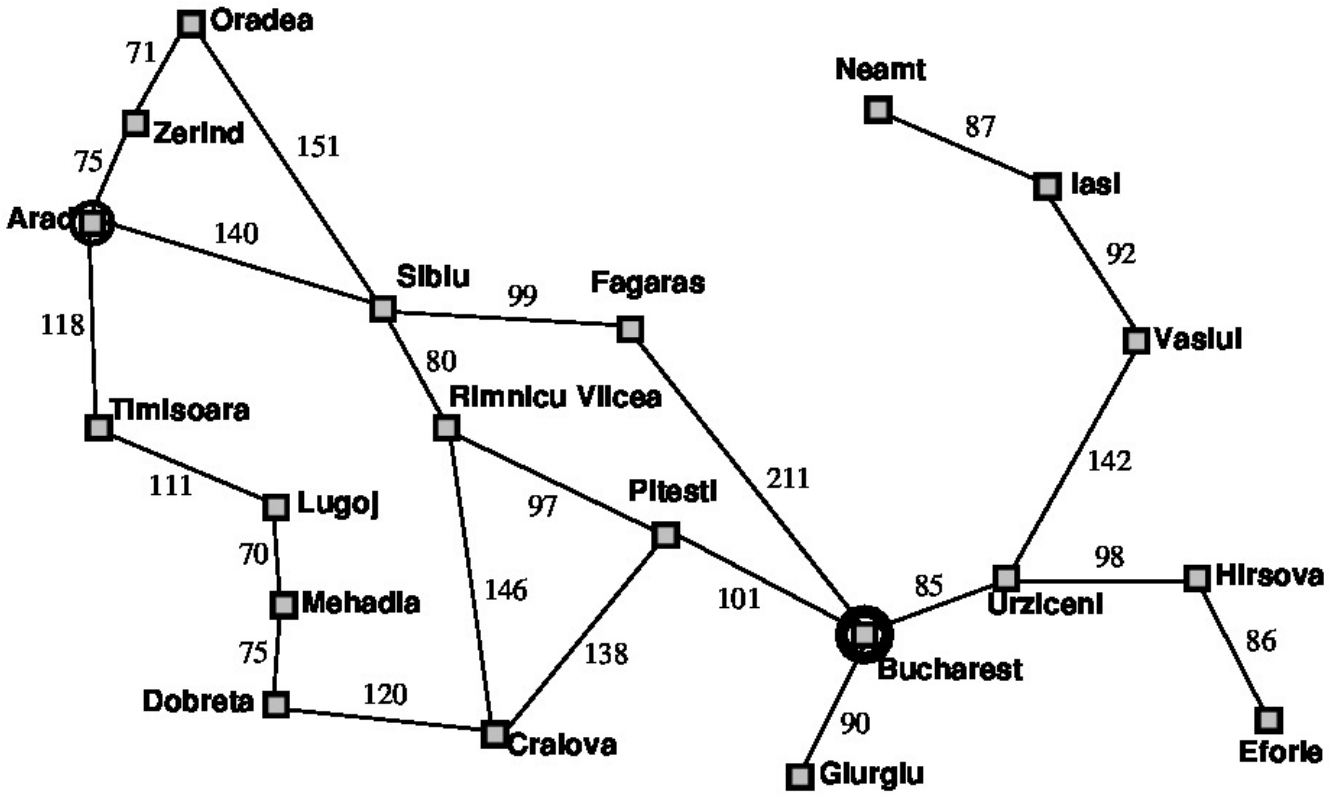
Coste uniforme I

- Expandir el nodo no expandido **de menor coste**.
- *Implementación*: *frontera* = cola **ordenada** por el **coste del camino**, **el menor** primero.
- **Equivalente** a primero en anchura si los costes de cada paso son todos **iguales**.

Coste uniforme II

- **Completa**: **Si**, si el coste de los pasos es $\geq \epsilon$
- **Tiempo** de **nodos** con $g \leq$ coste de la **solución óptima**, $O(b^{C^*/\epsilon})$ donde C^* es el coste de la solución óptima.
- **Tiempo** $O(log(b^d)b^d) \rightarrow O(b^d)$
- **Espacial** de **nodos** con $g \leq$ coste de la solución **óptima**, $O(b^{C^*/\epsilon})$
- **Espacial**: $O(b^d)$ peor caso es como **primero en anchura**.
- **Optimal**: Si, los nodos se **expanden** en orden **creciente** de $g(n)$

Ejemplo: costo uniforme



Resumen de Algoritmos

Criterio	Anchura	Uniforme	Profundidad	Acotada	Iterativa
¿Completa?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Tiempo	b^{d+1}	$b^{C^*/\epsilon}$	b^m	b^l	b^d
Espacio	b^{d+1}	$b^{C^*/\epsilon}$	bm	bl	bd
¿Optima?	Yes*	Yes	No	No	Yes*

Unificación de Algoritmos I

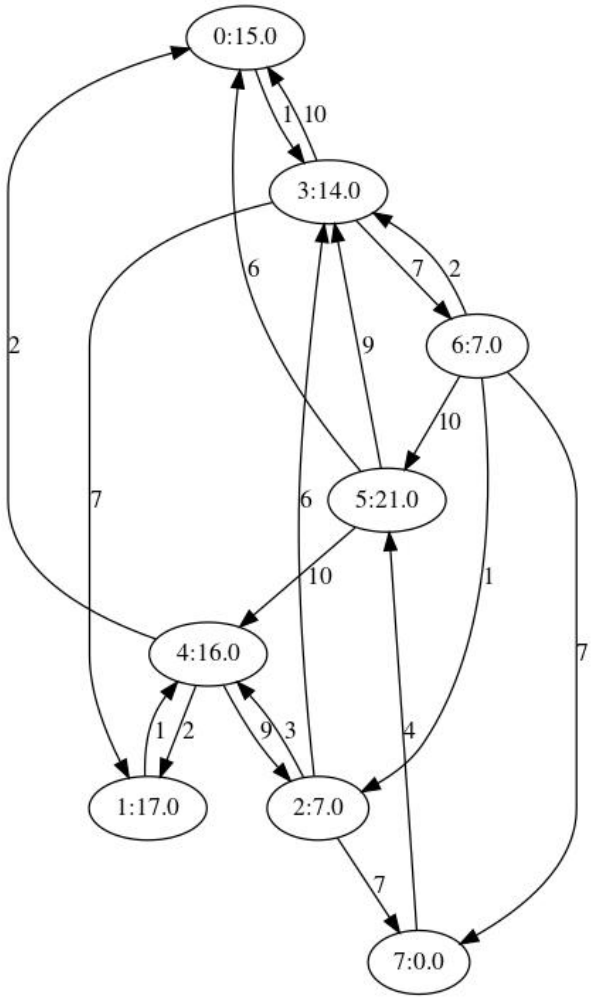
function GRAPH-SEARCH(*problem, fringe*) **returns** a solution, or failure

fringe ← an empty list ordered by f (value) in ascendent order
closed (visitados) ← 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

Unificación de Algoritmos II

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
    ID[*s*] ← *LastID+1*;
    PARENT-NODE[*s*] ← *node*;
    ACTION[*s*] ← *action*;
    STATE[*s*] ← *result*
    PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(STATE[*node*], *action*, *result*)
    DEPTH[*s*] ← DEPTH[*node*] + 1
    VALUE[*s*] ← DEPTH[*s*](anchura)
    VALUE[*s*] ← -DEPTH[*s*](profundidad)
    VALUE[*s*] ← PATH-COST[*s*](costo uniforme)
    add *s* to *successors*
  end for
return *successors*

Resolución de Ejercicios Teóricos I



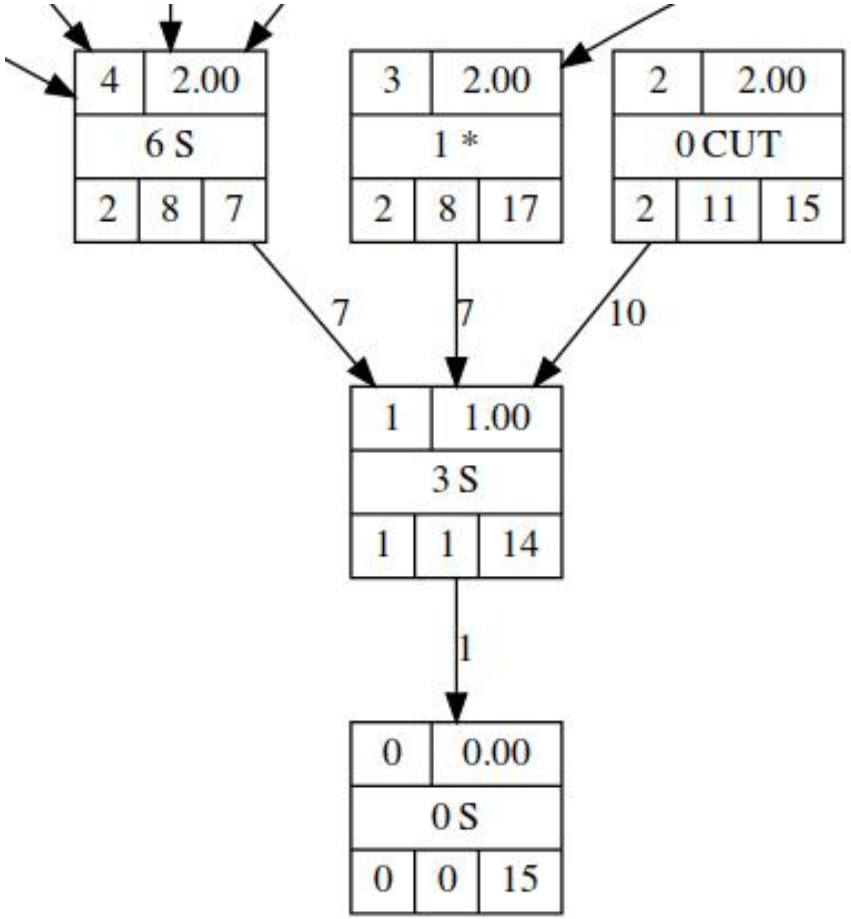
Resolución de Ejercicios Teóricos II

- **Estado Inicia**l: 0
- **Estado Fina**l: 7
- **Coste de la acción**: Etiqueta del arco entre dos nodos.
- **Objetivo**: Generar el árbol de búsqueda para una estrategia concreta.

Resolución de Ejercicios Teóricos III

Un nodo tiene los siguientes campos. Con respecto a la definición teórica se simplifica **eliminando la acción**.
Primera Fila: Id; Value
Segunda Fila: Estado
Tercera Fila: Prof; Coste; Heurística.
Etiquetas que se van **añadiendo** en el proceso de **ejecución** del algoritmo:
CUT: Nodo extraído de la frontera y no expandido por estar en visitados.
*: Nodo expandido.
S: Nodo parte del camino solución. Es lo último que se marca y sobreescribe el *.

Resolución de Ejercicios Teóricos IV



Cuestiones Pendientes I

Se continua intentando dar respuesta a las preguntas planteadas.

- ¿Por qué habla de **algoritmos de búsqueda** y no de **algoritmo de búsqueda** si sólo vamos a utilizar el algoritmo de búsqueda en grafo?
  En la literatura cada estrategia tiene su propio algoritmo. Nosotros en esta asignatura **los unificamos todos en uno** realizando pequeñas modificaciones siendo la principal de ella la gestión de la frontera.

Cuestiones Pendientes II

- Si tengo que resolver un problema de **búsqueda**, ¿Tengo que **especificar** todos los **campos** de cada **nodo** del árbol?. En este caso, ¿Qué **formato** de nodo de búsqueda o qué **campos** deben aparecer?
  Efectivamente se deben especificar todos los campos del nodo del árbol. Aparte de los teóricos a nivel de implementación se añade un **ID único para cada nodo** y además el **valor o f que es el criterio de ordenación en la frontera.**

Cuestiones Pendientes III

- ¿Por qué el **algoritmo** sí especifica que hay que **tomar** (y **eliminar**) el **primer** elemento de la frontera pero **no especifica** donde se deben insertar estos elementos?
  Es un **algoritmo general** que sirve para todas las estrategias.
  Ya se ha estudiado que depende de la estrategia los nuevos nodos se añaden en una posición concreta de la frontera.

Cuestiones Pendientes IV

- ¿Por qué el algoritmo **testea** si un nodo contiene un **estado** que satisface la **función objetivo** sólo cuando lo extrae de la **frontera** y no cuándo lo introduce? ¿No sería esto **último** mucho más **eficiente**?
  Para profundidad y anchura es válido detener el algoritmo cuando se obtiene un sucesor que satisface la función objetivo. Pero para costo uniforme no lo es ya que podemos añadir un nodo que satisface la función objetivo pero representa un camino de mayor coste que otro que aparece posteriormente. Como nuestro objetivo es mediante un único algoritmo implementar todas las estrategias y aunque es un poco más ineficiente para anchura y profundidad sólo detenemos el algoritmo si un nodo extraído de la frontera satisface la función objetivo.

Conclusiones

- La **formulación** de un problema normalmente requiere **abstraer** detalles del mundo real para definir un espacio de **estados** que puede ser explorado **de manera viable**.
- variedad de estrategias de búsqueda **sin información**.
- Profundidad **iterativa** usa espacio lineal y no tarda mucho más tiempo que otras estrategias **no informadas**.
- La búsqueda en **grafo** puede ser **exponencialmente** más eficiente que la búsqueda en **árbol**.