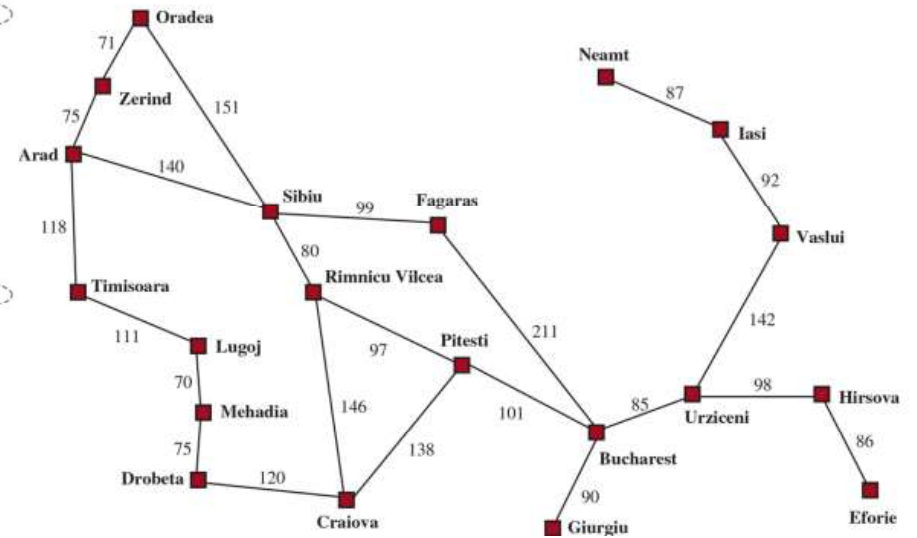
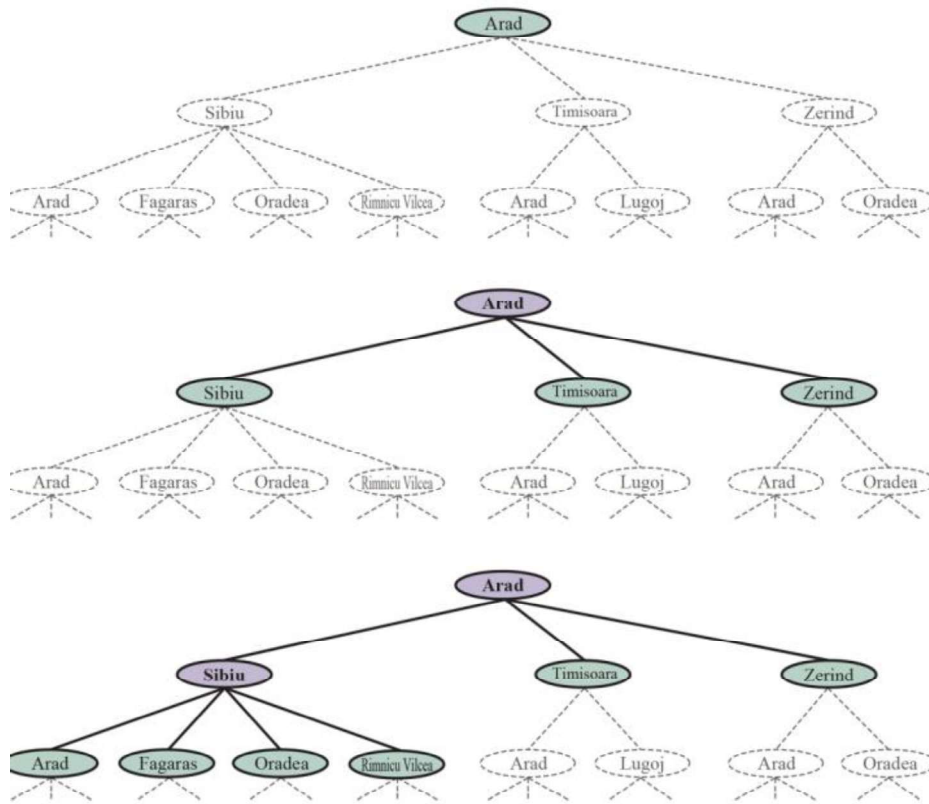


# ALGORITMOS DE BUSQUEDA- TECNICAS CIEGAS

## Inteligencia Artificial



# ALGORITMOS DE BUSQUEDA-PATHS REDUNDANTES



## ALGORITMOS DE BUSQUEDA-SOLVING PERFORMANCE

- **COMPLETITUD (COMPLETENESS):** ¿Está garantizado que el algoritmo encontrará una solución cuando exista una, y reportará correctamente el fallo cuando no la haya?
- **OPTIMALIDAD DE COSTO (COST OPTIMALITY):** ¿Encuentra una solución con el costo de camino más bajo entre todas las soluciones?
- **COMPLEJIDAD DE TIEMPO (TIME COMPLEXITY):** ¿Cuánto tiempo tarda en encontrar una solución? Esto se puede medir en segundos, o de manera más abstracta, por el número de estados y acciones considerados.
- **COMPLEJIDAD ESPACIAL (SPACE COMPLEXITY):** ¿Cuánta memoria se necesita para realizar la búsqueda?

## ALGORITMOS DE BUSQUEDA-SOLVING PERFORMANCE

La complejidad de tiempo y espacio se considera con respecto a alguna medida de la dificultad del problema. En la informática teórica, la medida típica es el tamaño del grafo del espacio de estados,  $|V|+|E|$

donde  $|V|$  es el número de vértices (nodos de estado) del grafo y  $|E|$  es el número de aristas (pares de estado/acción distintos).

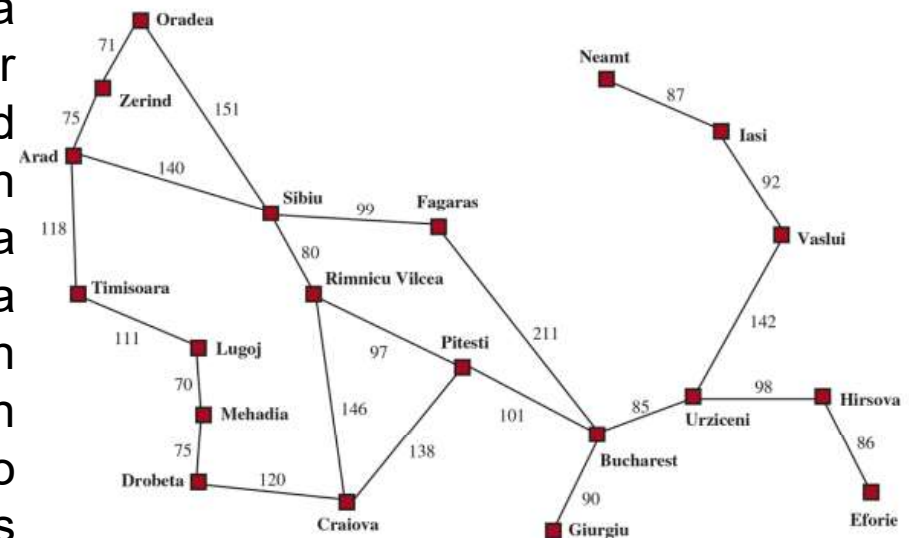
# ALGORITMOS DE BUSQUEDA-SOLVING PERFORMANCE


Para un espacio de estados implícito, la complejidad se puede medir en términos de **d**, la profundidad o número de acciones en una solución óptima; **m**, el número máximo de acciones en cualquier camino; y **b**, el factor de ramificación o número de sucesores de un nodo que necesitan ser considerados.

# TÉCNICAS CIEGAS

# ALGORITMOS DE BUSQUEDA-TÉCNICAS CIEGAS

Un algoritmo de búsqueda no informada no recibe ninguna pista sobre qué tan cerca está un estado del objetivo o los objetivos. Por ejemplo, considera nuestro agente en Arad con el objetivo de llegar a Bucarest. Un agente no informado, sin conocimiento de la geografía rumana, no tiene idea de si ir a Zerind o a Sibiu es un mejor primer paso. En contraste, un agente que conoce la ubicación de cada ciudad sabe que Sibiu está mucho más cerca de Bucarest y, por lo tanto, es más probable que esté en el camino más corto.





## TÉCNICAS CIEGAS- Dijkstra o Búsqueda de Costo Uniforme



## TÉCNICAS CIEGAS- Dijkstra o Búsqueda de Costo Uniforme

Cuando las acciones tienen costos diferentes, una elección obvia es usar la búsqueda del mejor primero (**best-first search**) donde la función de evaluación es el costo del camino desde la raíz hasta el nodo actual. Esto es conocido como el **algoritmo de Dijkstra** en la comunidad de ciencias de la computación teórica, y como **búsqueda de costo uniforme** en la comunidad de IA.

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure  
return BEST-FIRST-SEARCH(problem, PATH-COST)
```

## TÉCNICAS CIEGAS- Dijkstra o Búsqueda de Costo Uniforme

### Algoritmo de Dijkstra (Uniform cost) vs Breadth first Search

Mientras que la búsqueda en anchura se expande en ondas de profundidad uniforme primero profundidad 1, luego profundidad 2, y así sucesivamente la búsqueda de costo uniforme se expande en ondas de costo de camino uniforme.

$$O(b^{1+\lceil C^*/\epsilon \rceil})$$

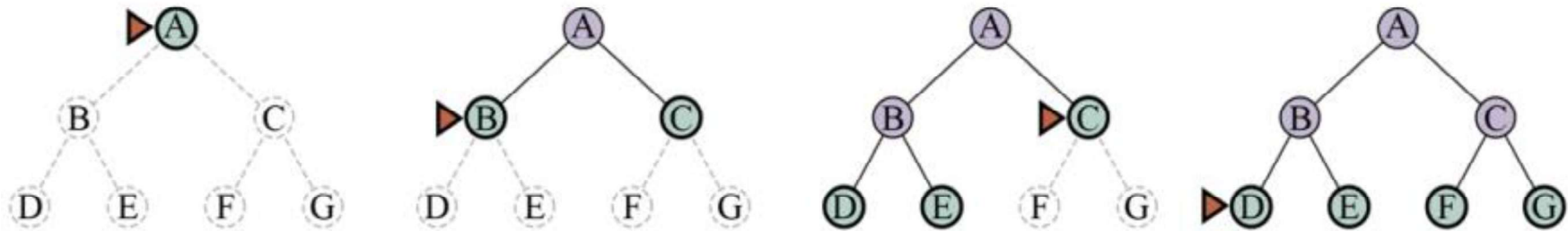
$C^*$ , el costo de la solución óptima, y  $\epsilon$ , un límite inferior en el costo de cada acción, con  $\epsilon > 0$



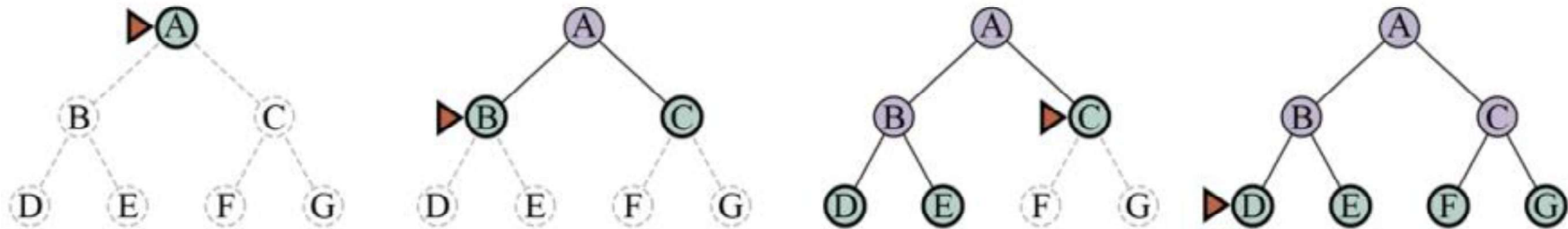
## TÉCNICAS CIEGAS- Breadth-First Search

## TÉCNICAS CIEGAS-Breadth-first search

Cuando todas las acciones tienen el mismo costo, una estrategia adecuada es la **Breadth-first Search**, en la que primero se expande el nodo raíz, luego se expanden todos los sucesores del nodo raíz, luego los sucesores de estos, y así sucesivamente.



## TÉCNICAS CIEGAS-Breadth-first search



En este caso, una cola First In, First Out (FIFO) será más rápida que una cola de prioridad, y nos dará el orden correcto de los nodos: los nodos nuevos (que siempre son más profundos que sus padres) van al final de la cola, y los nodos antiguos, que son menos profundos que los nuevos nodos, se expanden primero.

## TÉCNICAS CIEGAS-Breadth-first search

En este caso '**reached**' puede ser un conjunto de estados en lugar de un mapeo de estados a nodos, porque una vez que hemos alcanzado un estado, nunca podremos encontrar un mejor camino hacia ese estado. Esto también significa que podemos hacer una prueba temprana del objetivo, verificando si un nodo es una solución tan pronto como se genera.

La búsqueda siempre encuentra una solución con un número mínimo de acciones, porque cuando está generando nodos en la profundidad  $d$ , ya ha generado todos los nodos en la profundidad  $d-1$  por lo que, si uno de ellos fuera una solución, habría sido encontrada.

## TÉCNICAS CIEGAS-Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure  
  node  $\leftarrow$  NODE(problem.INITIAL)  
  if problem.IS-GOAL(node.STATE) then return node  
  frontier  $\leftarrow$  a FIFO queue, with node as an element  
  reached  $\leftarrow$  {problem.INITIAL}  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    for each child in EXPAND(problem, node) do  
      s  $\leftarrow$  child.STATE  
      if problem.IS-GOAL(s) then return child  
      if s is not in reached then  
        add s to reached  
        add child to frontier  
  return failure
```

## TÉCNICAS CIEGAS-Breadth-first search

Un árbol uniforme donde cada estado tiene  $b$  sucesores. La raíz del árbol de búsqueda genera  $b$  nodos, cada uno de los cuales genera  $b$  nodos más, para un total de  $b^2$  en el segundo nivel. Cada uno de estos genera  $b$  nodos más, produciendo  $b^3$  nodos en el tercer nivel, y así sucesivamente. Ahora supón que la solución está en la profundidad  $d$ . Entonces, el número total de nodos generados es:

$$1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$$

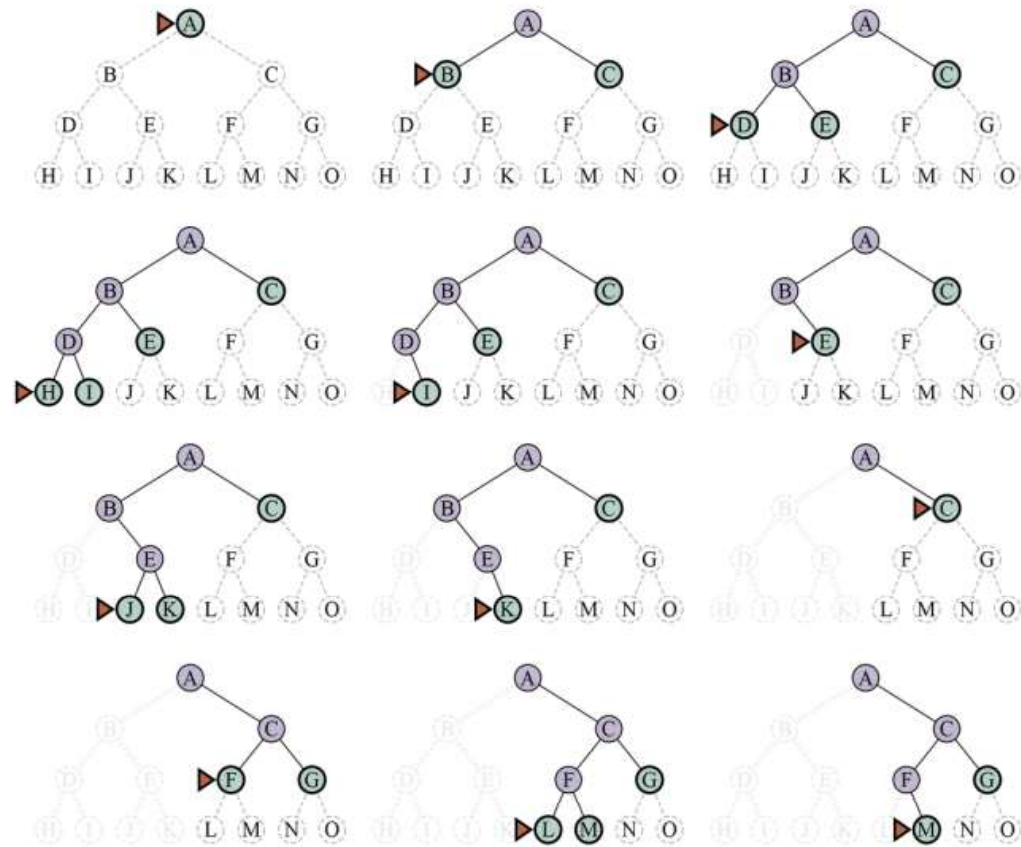


## TÉCNICAS CIEGAS- Depth-first search

## TÉCNICAS CIEGAS- Depth-first search

La búsqueda en profundidad siempre expande primero el nodo más profundo en la frontera. generalmente se implementa no como una búsqueda en grafo, sino como una búsqueda en forma de árbol que no mantiene una tabla de estados alcanzados

# TÉCNICAS CIEGAS- Depth-first search



## TÉCNICAS CIEGAS- Depth-first search

Para un espacio de estados con forma de árbol finito, una búsqueda en profundidad con forma de árbol toma un tiempo proporcional al número de estados, y tiene una complejidad de memoria de solo  **$O(bm)$**  donde  **$b$**  es el factor de ramificación y  **$m$**  es la profundidad máxima del árbol.



## TÉCNICAS CIEGAS- Depth limited-search

## TÉCNICAS CIEGAS- Depth-limited search

Para evitar que la búsqueda en profundidad se desvíe por un camino infinito, podemos usar la Depth-limited search, una versión de la búsqueda en profundidad en la que se proporciona un límite de profundidad,  $\ell$ , y se tratan todos los nodos a la profundidad  $\ell$  como si no tuvieran sucesores

```
function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff  
  frontier  $\leftarrow$  a LIFO queue (stack) with NODE(problem.INITIAL) as an element  
  result  $\leftarrow$  failure  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    if problem.IS-GOAL(node.STATE) then return node  
    if DEPTH(node) >  $\ell$  then  
      result  $\leftarrow$  cutoff  
    else if not IS-CYCLE(node) do  
      for each child in EXPAND(problem, node) do  
        add child to frontier  
  return result
```



## TÉCNICAS CIEGAS- Iterative Deepening search

## TÉCNICAS CIEGAS- Iterative Deepening search

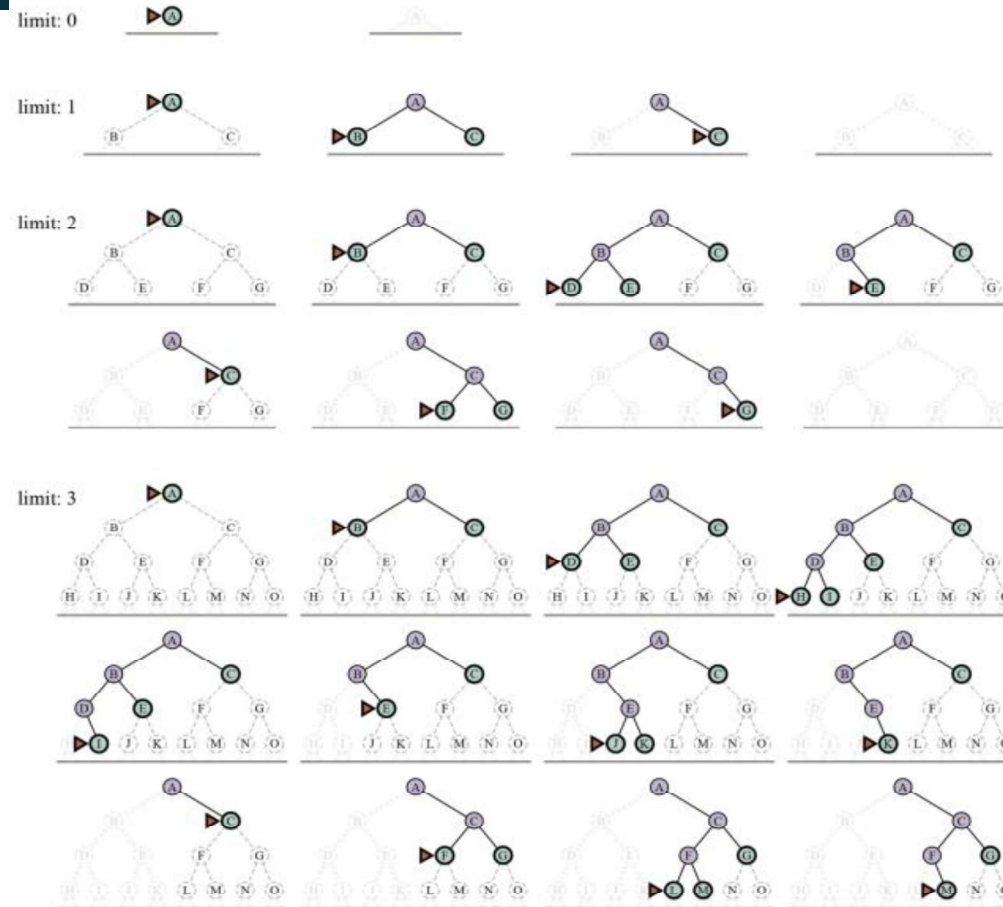
Resuelve el problema de elegir un buen valor para  $\ell$  probando todos los valores: primero 0, luego 1, luego 2, y así sucesivamente, hasta que se encuentre una solución, o la búsqueda en profundidad limitada devuelva el valor de falla en lugar del valor de corte.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```

Al igual que la búsqueda en anchura, la búsqueda en profundidad iterativa es óptima para problemas donde todas las acciones tienen el mismo costo, y es completa en espacios de estados acíclicos finitos



# TÉCNICAS CIEGAS- Depth-first search





## TÉCNICAS CIEGAS- Comparación

## TÉCNICAS CIEGAS- Depth-first search

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes <sup>1</sup>	Yes <sup>1,2</sup>	No	No	Yes <sup>1</sup>
Optimal cost?	Yes <sup>3</sup>	Yes	No	No	Yes <sup>3</sup>
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$

**b:** factor de ramificación.

**m:** es la profundidad máxima del árbol de búsqueda.

**d:** es la profundidad de la solución más superficial, o es m cuando no hay solución.

**l:** es el límite de profundidad..

1. completo si **b** es finito y el espacio de estados tiene una solución o es finito.
2. Completo si todos los costos de acción son  $E > 0$ .
3. Es óptimo en costos si todos los costos de acción son idénticos