

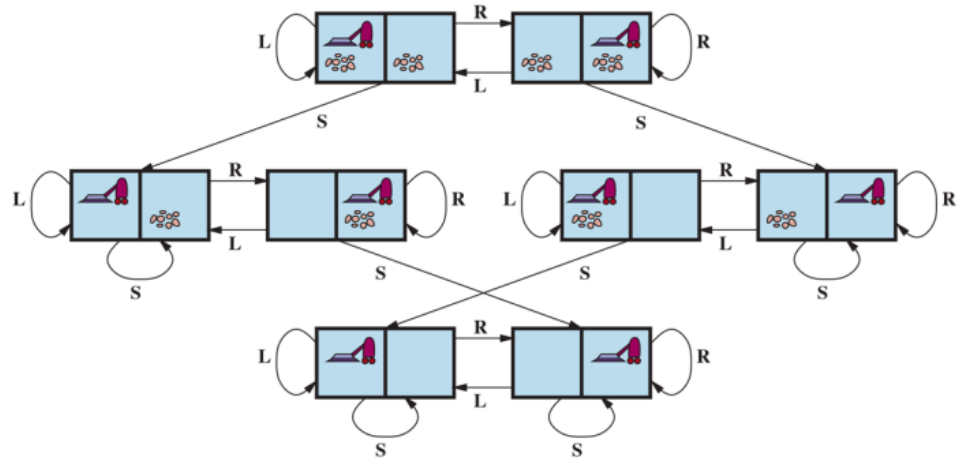
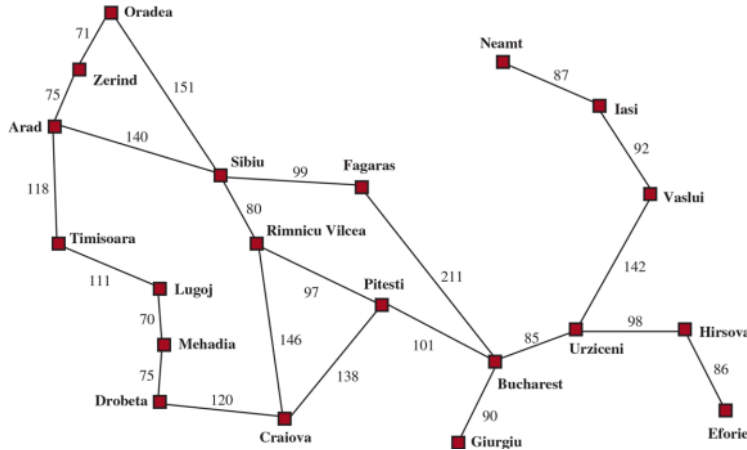
# ALGORITMOS DE BUSQUÉDA

## Inteligencia Artificial

Yomin E. Jaramillo Múnera, Msc. Automatizacion y Control Ind  
[vejaramilm@eafit.edu.co](mailto:vejaramilm@eafit.edu.co)

# REPASO: PROBLEMAS DE BUSQUEDA

- Un conjunto de posibles estados en los que el entorno puede estar. A esto lo llamamos el espacio de estados.



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

# REPASO: PROBLEMAS DE BUSQUEDA

- El estado inicial en el que el agente comienza.

**Ej1:** Arad,

**Ej2:** Cualquiera de los estados del problema de aspiradora

- Podemos tener uno o más estados objetivo. A veces hay un único estado objetivo, a veces hay un pequeño conjunto de estados objetivo alternativos

**Ej1:** Bucharest,

**Ej2:** Donde las 2 casillas estén limpias sin importar la ubicación de la aspiradora

# REPASO: PROBLEMAS DE BUSQUEDA

- Las acciones disponibles para el agente.

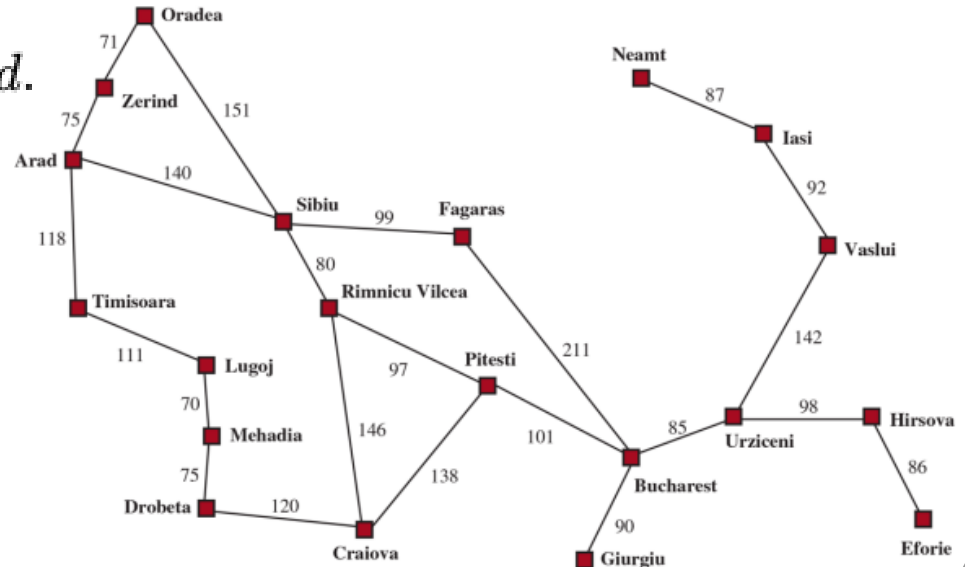
$ACTIONS(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}$ .

- Modelo de transición

$RESULT(Arad, ToZerind) = Zerind$ .

- Costo de la acción

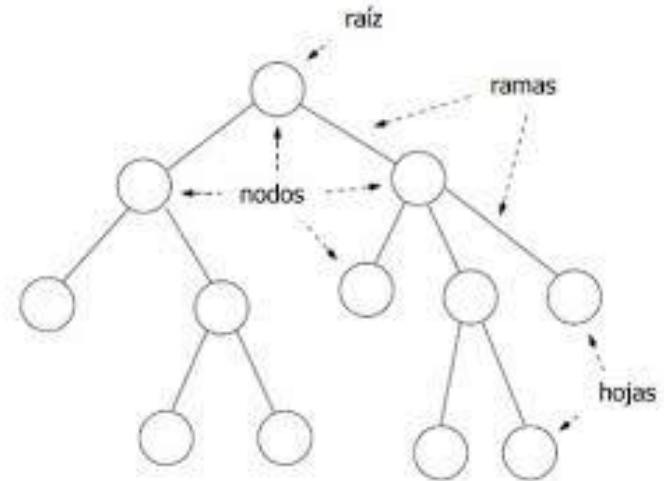
$ACTION-COST(s, a, s')$



# ALGORITMOS DE BUSQUEDA

*“Un algoritmo de búsqueda toma un problema de búsqueda como entrada y devuelve una solución, o una indicación de fracaso.”*

En esta clase vamos a revisar algoritmos donde podemos realizar una búsqueda basada en una estructura de ‘Arboles de búsqueda’ el cual nos permite explorar varios caminos para llegar a una solución



# ALGORITMOS DE BUSQUEDA

**Espacio de estados:** Describe el conjunto (posiblemente infinito) de estados en el mundo, y las acciones que permiten transiciones de un estado a otro.

**Árbol de búsqueda:** Describe los caminos entre estos estados, alcanzando el objetivo

# ALGORITMOS DE BUSQUEDA

- Dado un estado ( $s$ ), **ACCIONES**( $s$ ) devuelve un conjunto finito de acciones que se pueden ejecutar. Decimos que cada una de estas acciones es aplicable en ese estado. Un ejemplo:

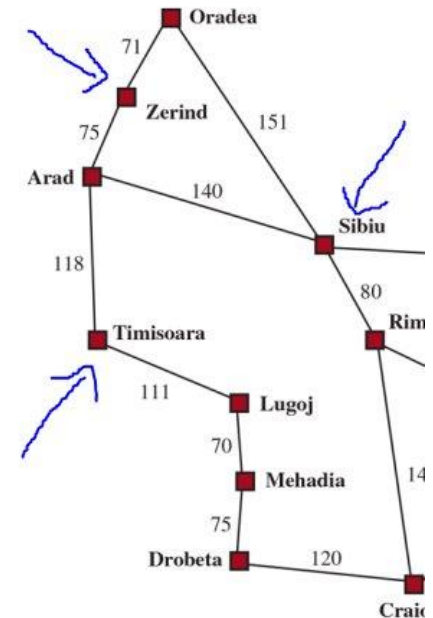
$$\text{ACTIONS}(\text{Arad}) = \{\text{ToSibiu}, \text{ToTimisoara}, \text{ToZerind}\}.$$

- Un modelo de transición, que describe lo que hace cada acción. **RESULTADO** ( $s, a$ ) devuelve el estado que resulta de realizar la acción  $a$  en el estado  $s$ . Por ejemplo:

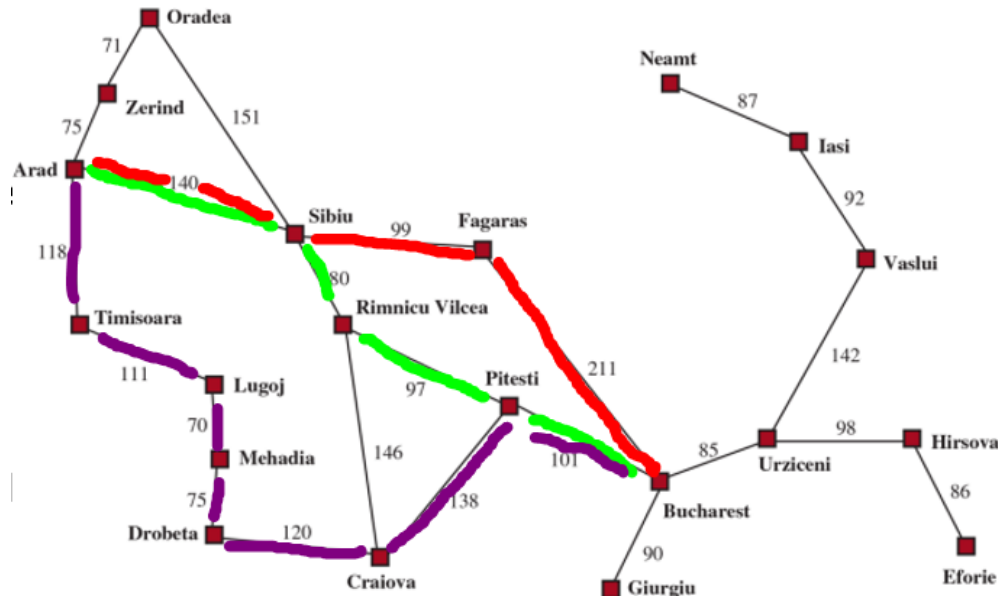
$$\text{RESULT}(\text{Arad}, \text{ToZerind}) = \text{Zerind}.$$

- Una función de costo asociada a la acción, denotada por:

$$\text{ACTION-COST}(s, a, s')$$



# ALGORITMOS DE BUSQUEDA

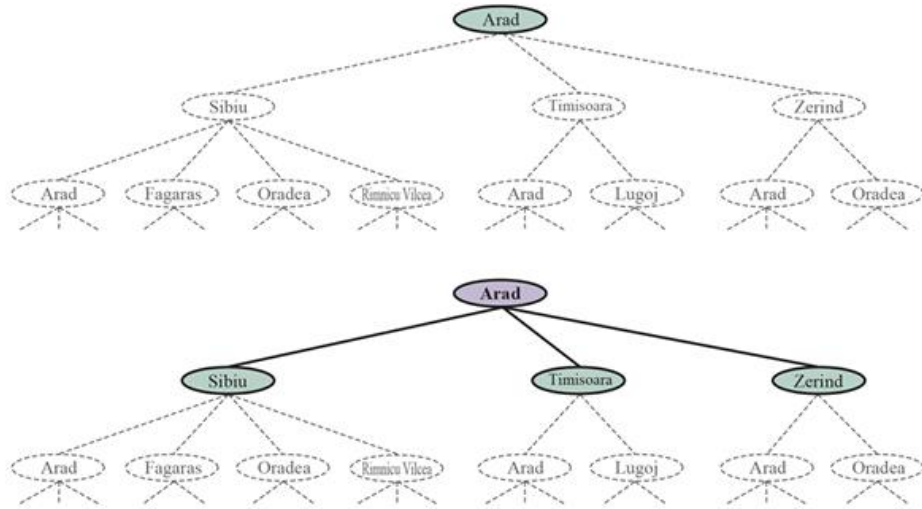
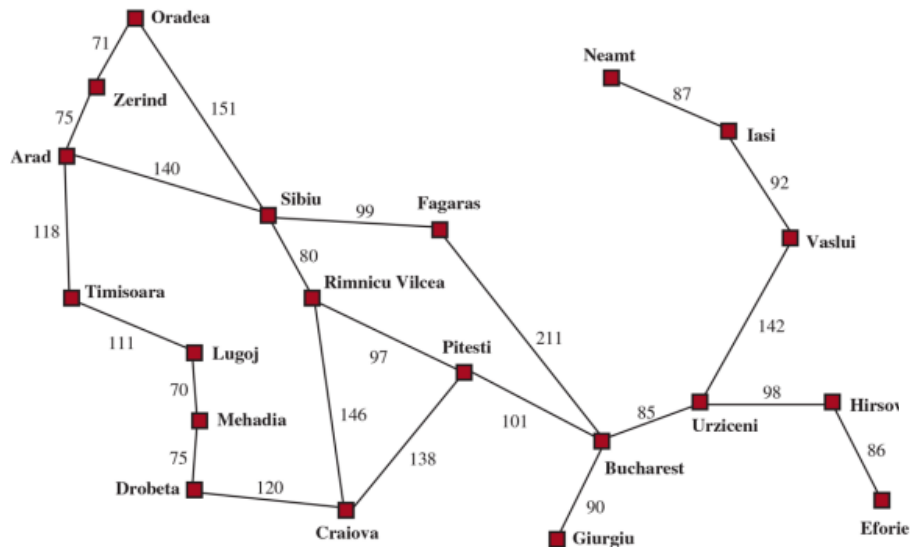


El árbol de búsqueda puede tener múltiples caminos hacia (y, por lo tanto, múltiples nodos para) cualquier estado dado, pero cada nodo en el árbol tiene un camino único de regreso a la raíz (como en todos los árboles).



# ALGORITMOS DE BUSQUEDA

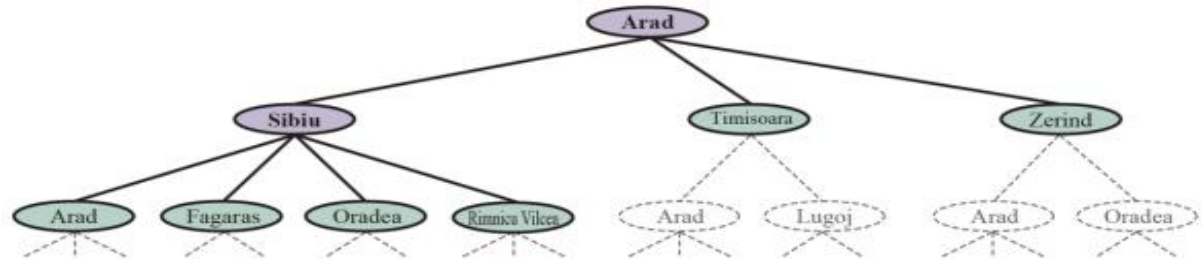
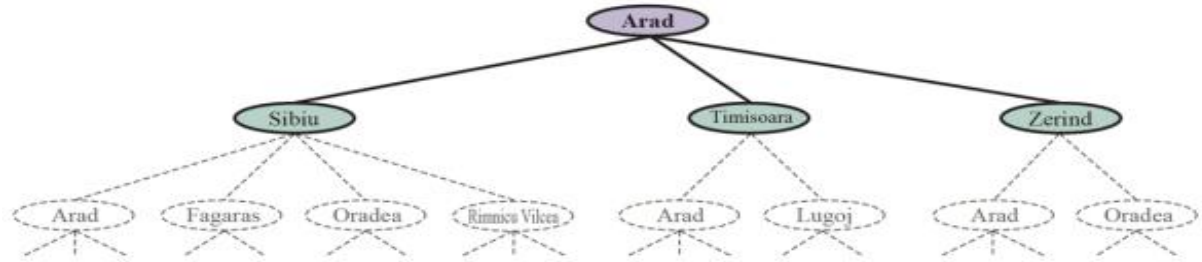
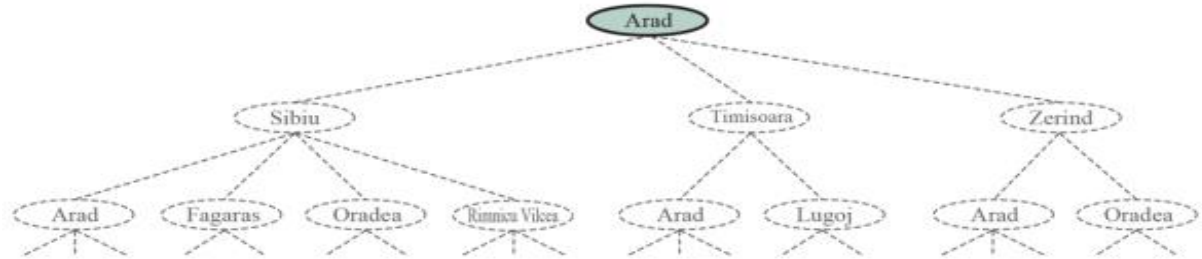
Volvamos a nuestro problema de búsqueda de Rumania



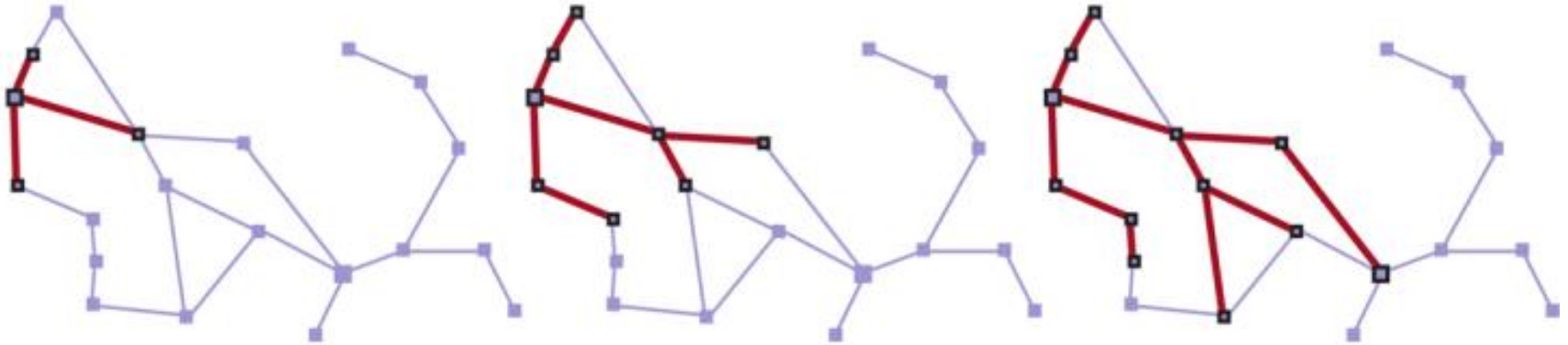
# ALGORITMOS DE BUSQUEDA

Decimos que cualquier estado para el cual se haya generado un nodo ha sido alcanzado (ya sea que ese nodo haya sido expandido o no).

Por otro lado, los nodos no expandidos se conocen como 'Frontera del árbol de búsqueda'



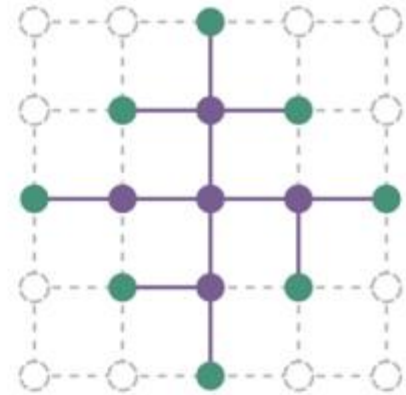
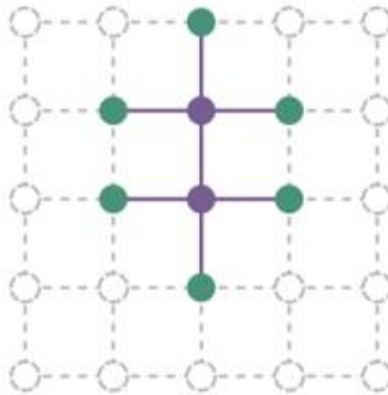
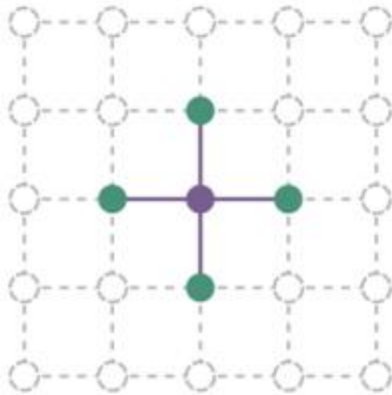
# ALGORITMOS DE BUSQUEDA



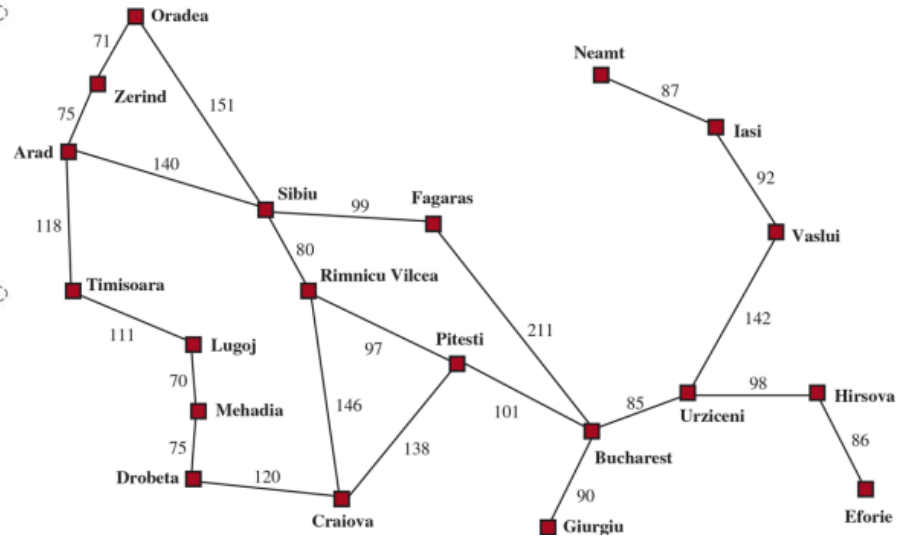
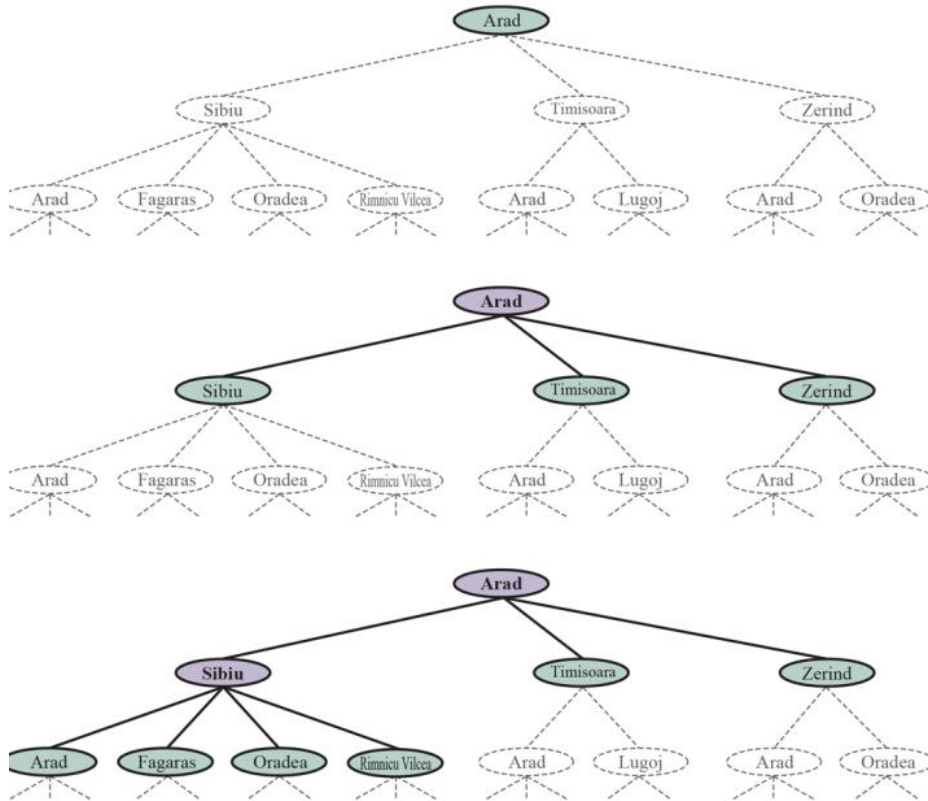
Observa que en la tercera etapa, la ciudad más alta (Oradea) tiene dos sucesores, ambos de los cuales ya han sido alcanzados por otros caminos, por lo que no se extienden caminos desde Oradea.

# ALGORITMOS DE BUSQUEDA

La frontera es el conjunto de nodos (y los estados correspondientes) que han sido alcanzados pero aún no expandidos; el interior es el conjunto de nodos (y los estados correspondientes) que han sido expandidos; y el exterior es el conjunto de estados que no han sido alcanzados.



# 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.

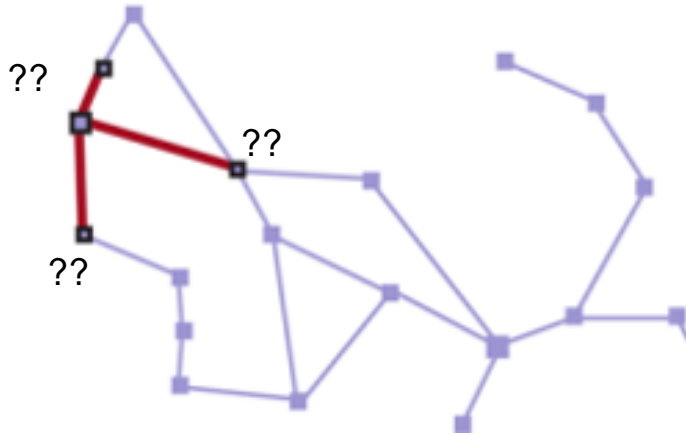




# ALGORITMO DE BEST- FIRST SEARCH

# ALGORITMOS DE BUSQUEDA-BEAST FIRST SEARCH

¿Cómo decidimos qué nodo de la frontera expandir a continuación? Un enfoque muy general se llama **búsqueda del mejor primero** (*best-first search*), en el cual elegimos un nodo  $n$  con el valor mínimo de alguna función de evaluación  $f(n)$ .



En cada iteración, elegimos un nodo en la frontera con el valor mínimo de  $f(n)$ , lo devolvemos si su estado es un estado objetivo, y de lo contrario aplicamos una función de expansión.

# BEST-FIRST SEARCH

Los algoritmos de búsqueda requieren una estructura de datos para mantener un registro del árbol de búsqueda. Un nodo en el árbol se representa mediante una estructura de datos con cuatro componentes:

- **node.STATE:** el estado al que corresponde el nodo.
- **node.PARENT:** el nodo en el árbol que generó este nodo.
- **node.ACTION:** la acción que se aplicó al estado del padre para generar este nodo.
- **node.PATH-COST:** el costo total del camino desde el estado inicial hasta este nodo.

# ALGORITMOS DE BUSQUEDA

Necesitamos una estructura de datos para almacenar la frontera. La elección adecuada es una cola (**queue**) de algún tipo, porque las operaciones en una frontera son:

- **IS-EMPTY(frontier)**: devuelve verdadero solo si no hay nodos en la frontera.
- **POP(frontier)**: elimina el nodo superior de la frontera y lo devuelve.
- **TOP(frontier)**: devuelve (pero no elimina) el nodo superior de la frontera.
- **ADD(node, frontier)**: inserta el nodo en su lugar adecuado en la cola.

# ALGORITMOS DE BUSQUEDA

## Tipos de colas se utilizan en los algoritmos de búsqueda:

- **Priority queue:** Este tipo de cola extrae primero el nodo con el costo mínimo de acuerdo con alguna función de evaluación. Se utiliza en la búsqueda del mejor primero (*best-first search*).
- **FIFO queue (First-In-First-Out)**, o cola de primero en entrar, primero en salir, extrae primero el nodo que se agregó a la cola en primer lugar; veremos que se utiliza en la búsqueda en anchura (*breadth-first search*).
- **LIFO queue (Last-In-First-Out)**, también conocida como pila (*stack*), extrae primero el nodo que se agregó más recientemente; veremos que se utiliza en la búsqueda en profundidad (*depth-first search*).

# ALGORITMOS DE BUSQUEDA-BEAST FIRST SEARCH

```
function BEST-FIRST-SEARCH(problem,f) returns a solution node or failure  
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)  
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element  
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node  
  while not IS-EMPTY(frontier) do  
    node  $\leftarrow$  POP(frontier)  
    if problem.IS-GOAL(node.STATE) then return node  
    for each child in EXPAND(problem, node) do  
      s  $\leftarrow$  child.STATE  
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then  
        reached[s]  $\leftarrow$  child  
        add child to frontier  
  return failure
```

```
function EXPAND(problem,node) yields nodes  
  s  $\leftarrow$  node.STATE  
  for each action in problem.ACTIONS(s) do  
    s'  $\leftarrow$  problem.RESULT(s, action)  
    cost  $\leftarrow$  node.PATH-COST + problem.ACTION-COST(s, action, s')  
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Cada nodo hijo se agrega a la frontera si no ha sido alcanzado antes, o se vuelve a agregar si ahora se alcanza con un costo de camino menor que cualquier camino anterior. El algoritmo devuelve una indicación de fallo o un nodo que representa un camino hacia un objetivo.


# TECNICAS CIEGAS O NO INFORMADAS

# TECNICAS CIEGAS O NO INFORMADAS

Un algoritmo de búsqueda no informado no tiene ninguna pista sobre qué tan cerca está un estado del objetivo.



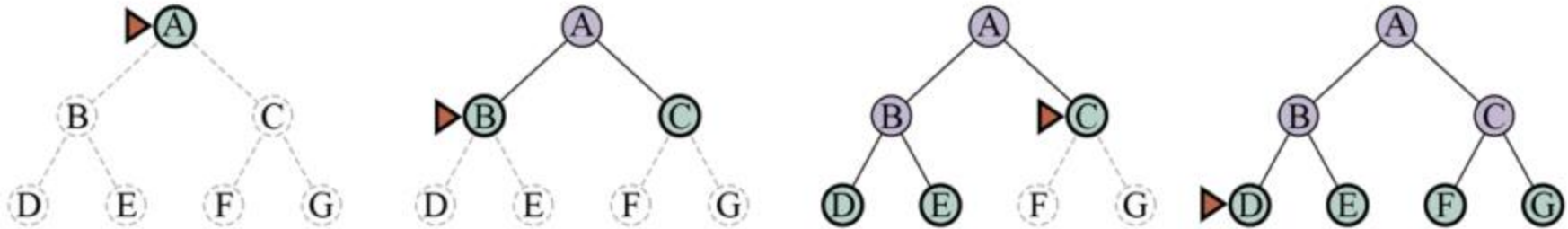




## **TECNICAS CIEGAS O NO INFORMADAS BREADTH-FIRST SEARCH**

# 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 '**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.*

# TÉCNICAS CIEGAS-Breadth-first search

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

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



**GRACIAS**