



# Inteligencia Artificial

UTN – FRVM

5º Año Ing. en Sistemas de  
Información



# Agenda

- Problem-solving Agents. Problem Types and Formulation. Example Problems.
- Basic Search Algorithms.





# Problem-solving Agents

- Goal-based agent called a **problem-solving agent**.
- Problem-solving agents use **atomic** representations, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms.
- Definitions of **problems** and their **solutions**. General-purpose search algorithms that can be used to solve these problems.
- **Uninformed** search algorithms — algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.
- **Informed** search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions.
- The simplest kind of task environment, for which the solution to a problem is always a *fixed sequence* of actions.



# Problem-solving Agents

- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.



# The Environment

- **Observable**, so the agent always knows the current state.
- **Discrete**, so at any given state there are only finitely many actions to choose from.
- **Known**, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.)
- **Deterministic**, so each action has exactly one outcome.
- *Under these assumptions, the solution to any problem is a fixed sequence of actions.*



# Search, Solution, Execution

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A **search algorithm** takes a problem as input and returns a **solution** in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.



# Problem-solving agents

```
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  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action
```

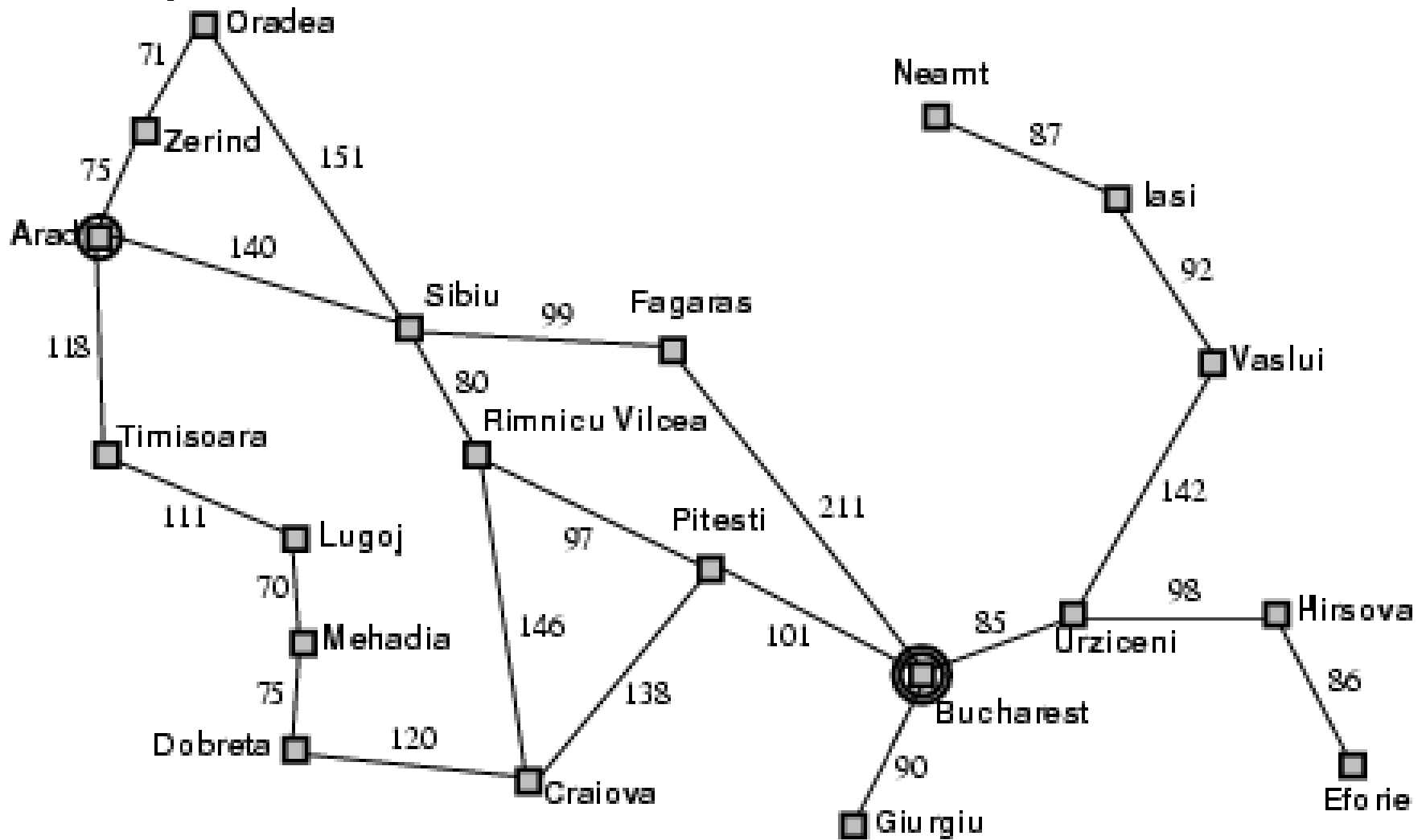


# Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulate problem:
  - **states**: various cities
  - **actions**: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest



# Example: Romania





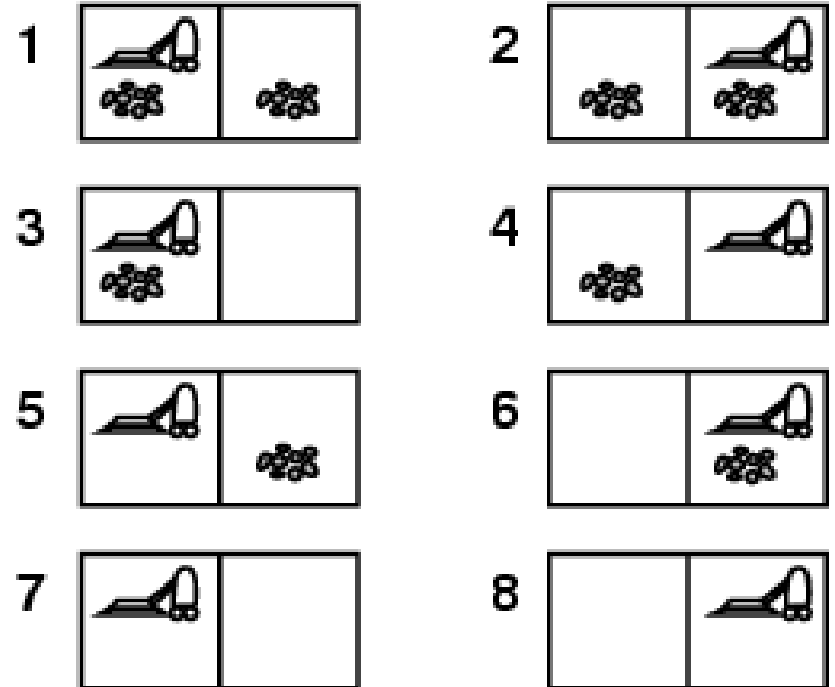
# Problem types

- Deterministic, fully observable → **single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → **sensorless problem (conformant problem)**
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → **contingency problem**
  - percepts provide **new** information about current state
  - often **interleave** } search, execution
- Unknown state space → **exploration problem**

# Example: vacuum world

- Single-state, start in #5.

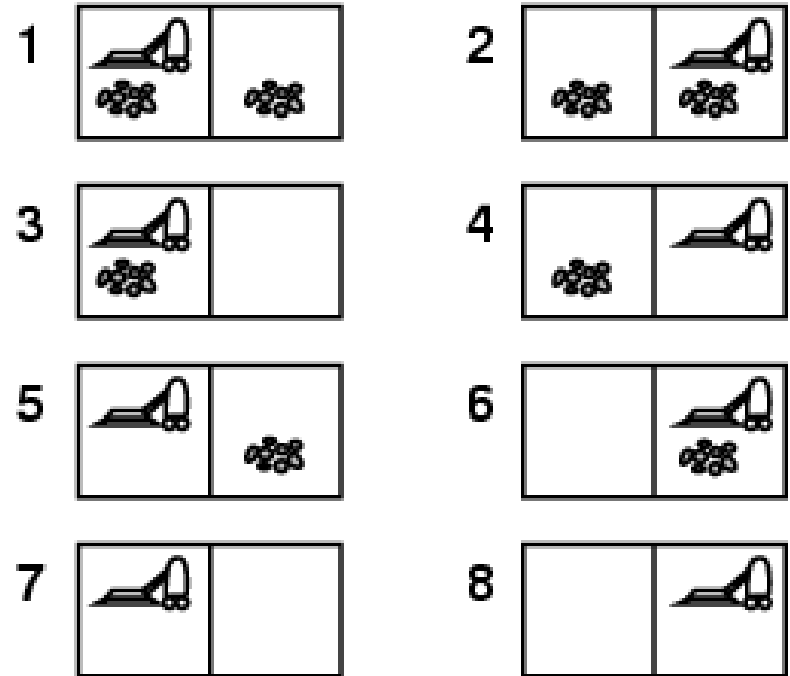
Solution?



# Example: vacuum world

- 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?



# Example: vacuum world

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

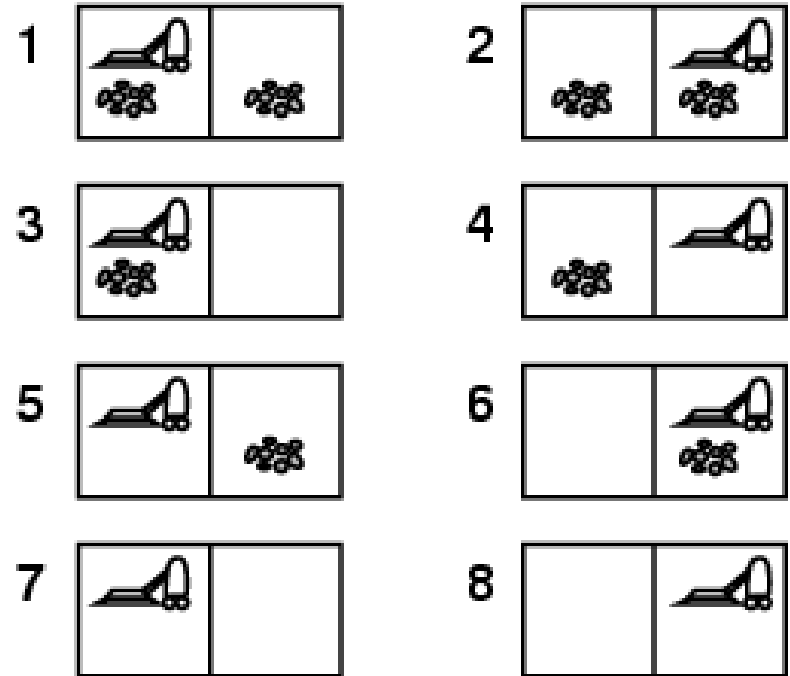
Solution?

*[Right, Suck, Left, Suck]*

- Contingency

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable: location, dirt at current location.
- Percept:  $[L, \text{Clean}]$ , i.e., start in #5 or #7

Solution?



# Example: vacuum world

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

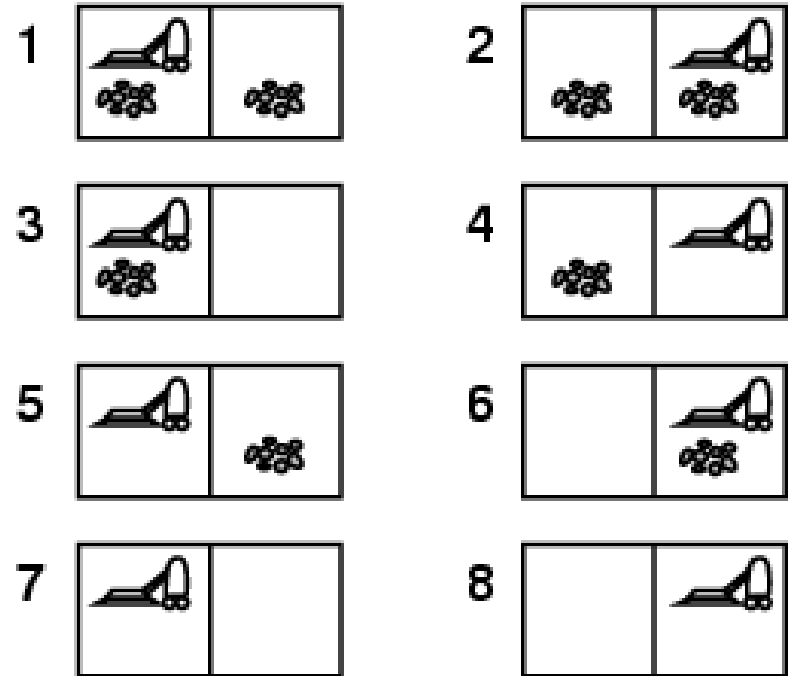
Solution?

*[Right, Suck, Left, Suck]*

- Contingency

- Nondeterministic: *Suck* may dirty a clean carpet
- Partially observable: location, dirt at current location.
- Percept:  $[L, \text{Clean}]$ , i.e., start in #5 or #7

Solution? *[Right, if dirt then Suck]*





# Single-state problem formulation

A **problem** is defined by four items:

1. **initial state** e.g., "at Arad"
  2. **actions** or **successor function**  $S(x)$  = set of action–state pairs
    - e.g.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind} \rangle, \dots \}$
  3. **goal test**, can be
    - **explicit**, e.g.,  $x = \text{"at Bucharest"}$
    - **implicit**, e.g.,  $\text{Checkmate}(x)$
  4. **path cost** (additive)
    - e.g., sum of distances, number of actions executed, etc.
    - $c(x, a, y)$  is the **step cost**, assumed to be  $\geq 0$
- A **solution** is a sequence of actions leading from the initial state to a goal state

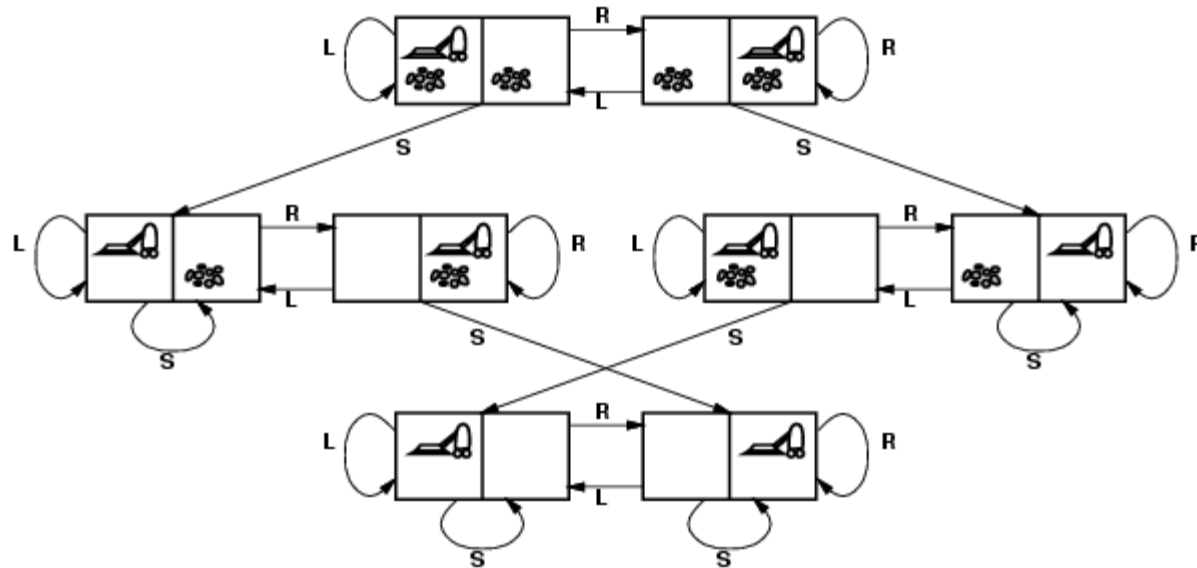


# Selecting a state space

- Real world is absurdly complex
  - state space must be **abstracted** for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, **any** real state "in Arad" must get to **some** real state "in Zerind"
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

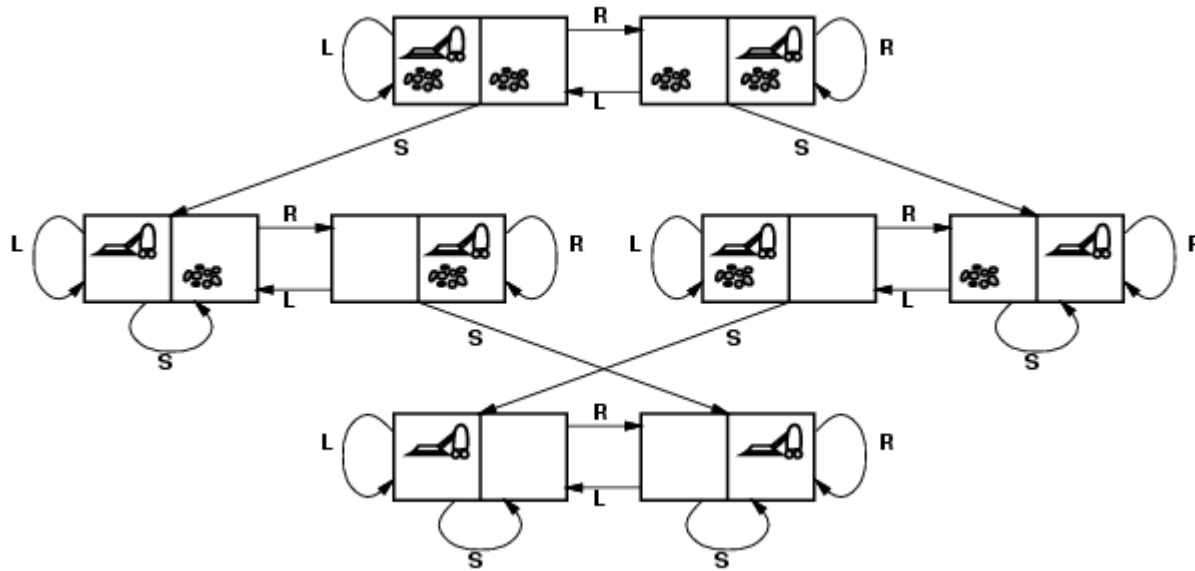


# Vacuum world state space graph



- states?
- actions?
- goal test?
- path cost?

# Vacuum world state space graph



- states? integer dirt and robot location
- actions? *Left, Right, Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

# Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

- states?
- actions?
- goal test?
- path cost?

# Example: The 8-puzzle

7	2	4
5		6
8	3	1

Start State

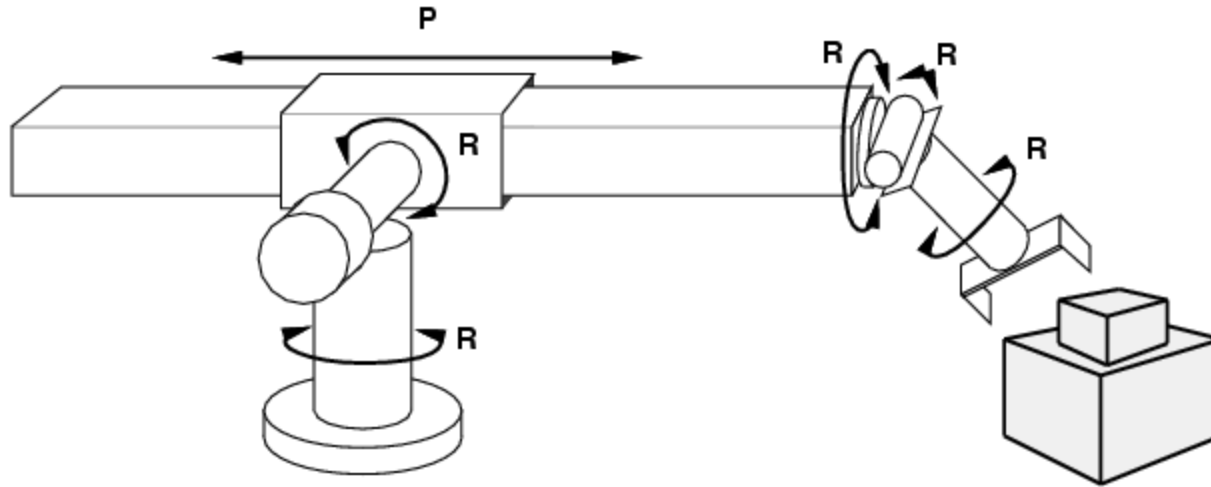
	1	2
3	4	5
6	7	8

Goal State

- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of  $n$ -Puzzle family is NP-hard]

# Example: robotic assembly



- 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
- path cost?: time to execute

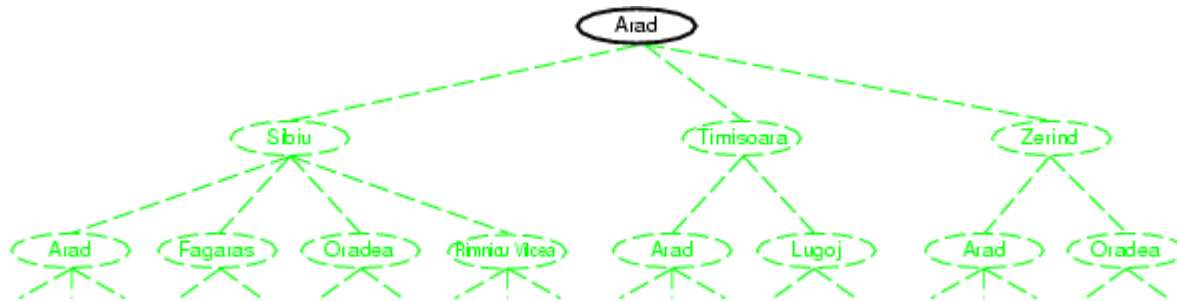
# Tree search algorithms

## ○ Basic idea:

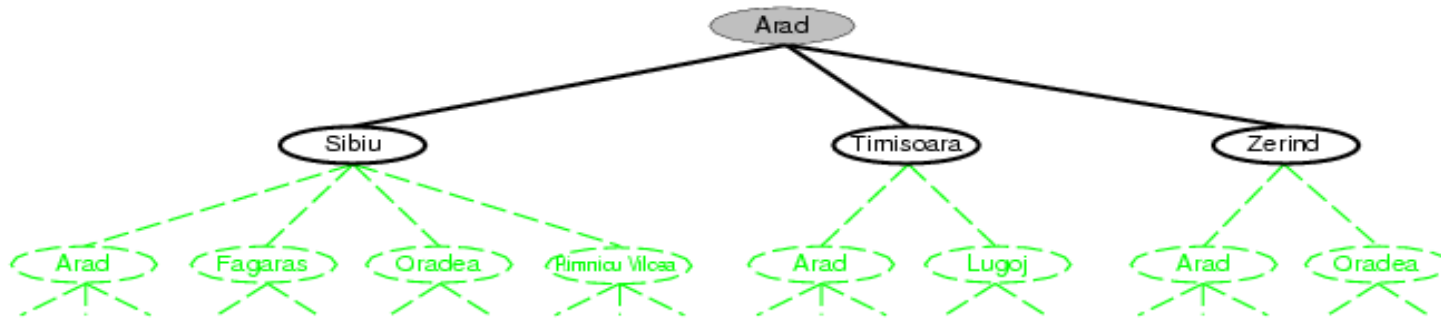
- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. ~**expanding** states)

```
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
```

# Tree search example

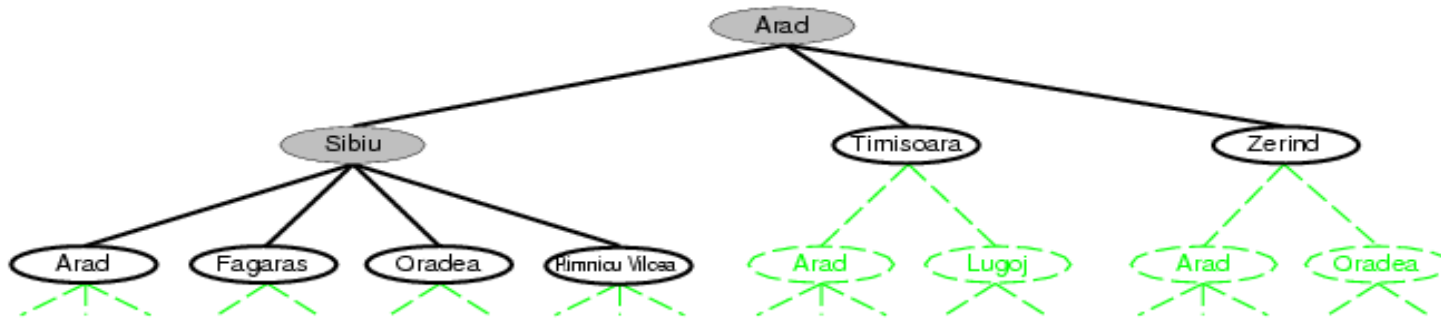


# Tree search example





# Tree search example





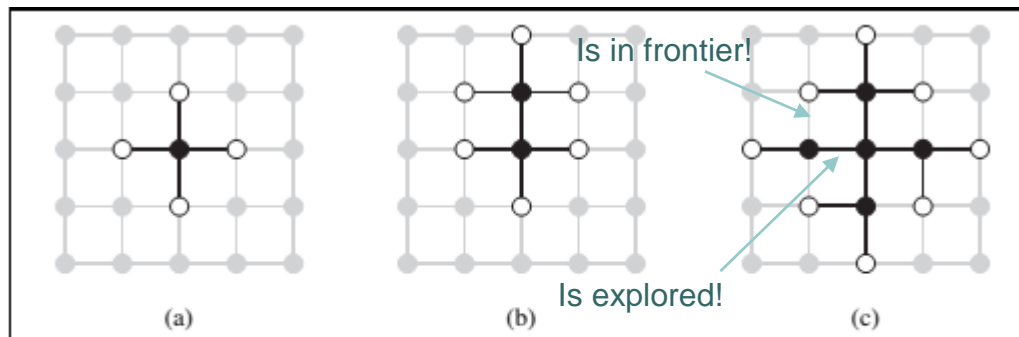
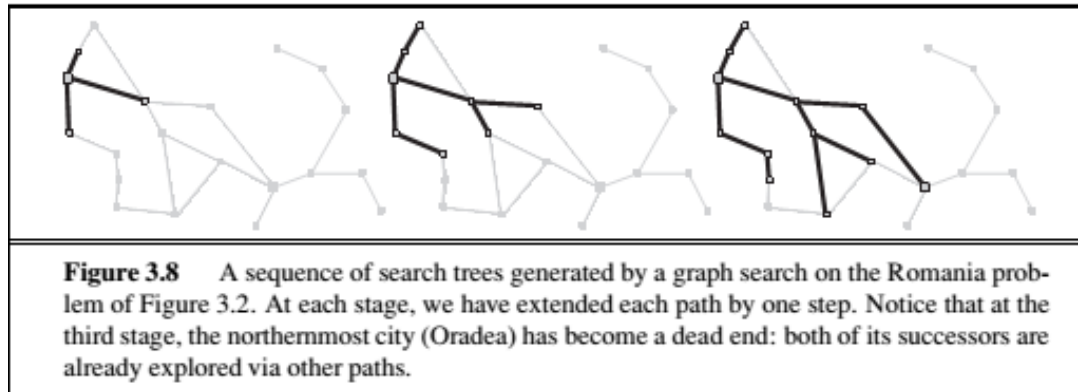
# Implementation: general tree search

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

---

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

# Examples



**Figure 3.9** The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been explored. In (b), one leaf node has been explored. In (c), the remaining successors of the root have been explored in clockwise order.

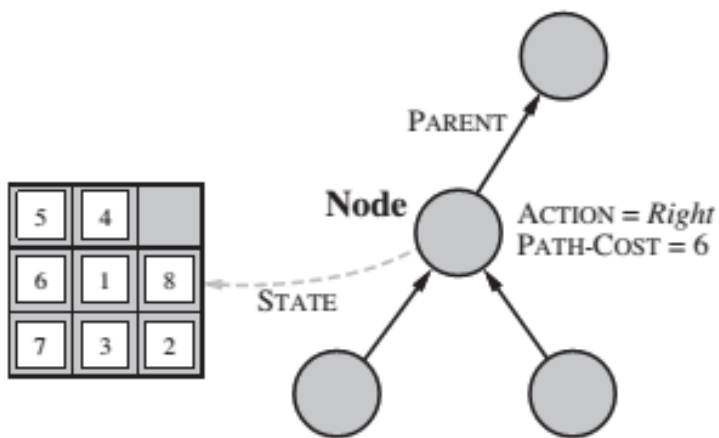


# Infrastructure of search algorithms

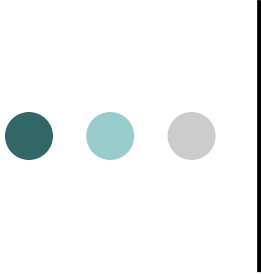
- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- For each node  $n$  of the tree, we have a structure that contains four components:
  - **$n.STATE$** : the state in the state space to which the node corresponds;
  - **$n.PARENT$** : the node in the search tree that generated this node;
  - **$n.ACTION$** : the action that was applied to the parent to generate the node;
  - **$n.PATH-COST$** : the cost, traditionally denoted by  $g(n)$ , of the path from the initial state to the node, as indicated by the parent pointers.

# Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree includes **state**, **parent node**, **action**, **path cost**  $g(x)$



```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```



# Storing the frontier and explored set

- Now that we have nodes, we need somewhere to put them.
- The **frontier** needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**.
- The **explored set** can be implemented with a **hash table** to allow efficient checking for repeated states.
- The operations on a queue are as follows:
  - **EMPTY?(queue)** returns true only if there are no more elements in the queue.
  - **POP(queue)** removes the first element of the queue and returns it (LIFO or FIFO)
  - **INSERT(element, queue)** inserts an element and returns the resulting queue.



# Search strategies

- A search strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - $b$ : maximum branching factor of the search tree (***maximum number of successors of any node***)
  - $d$ : depth of the least-cost solution
  - $m$ : maximum depth of the state space



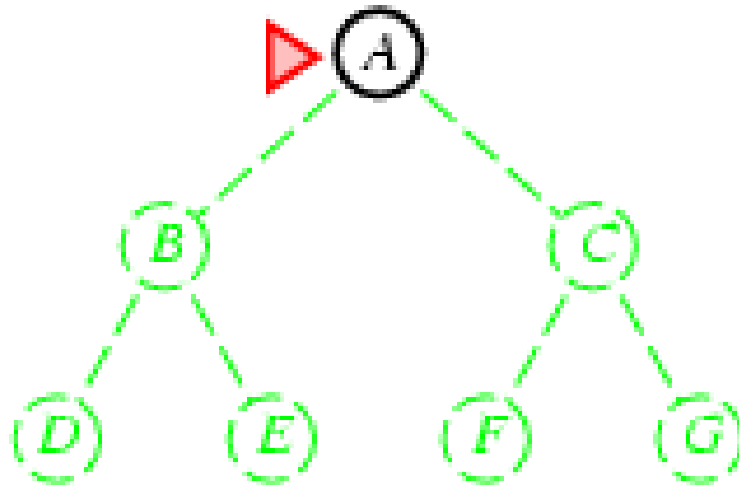
# Uninformed search strategies

- **Uninformed** search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search



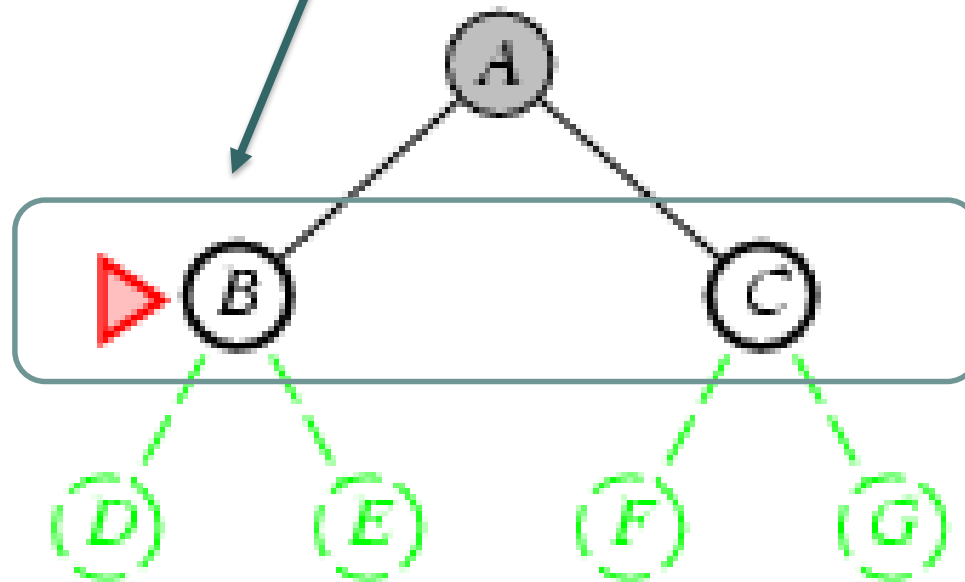
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - frontier* is a FIFO queue, i.e., new successors go at end



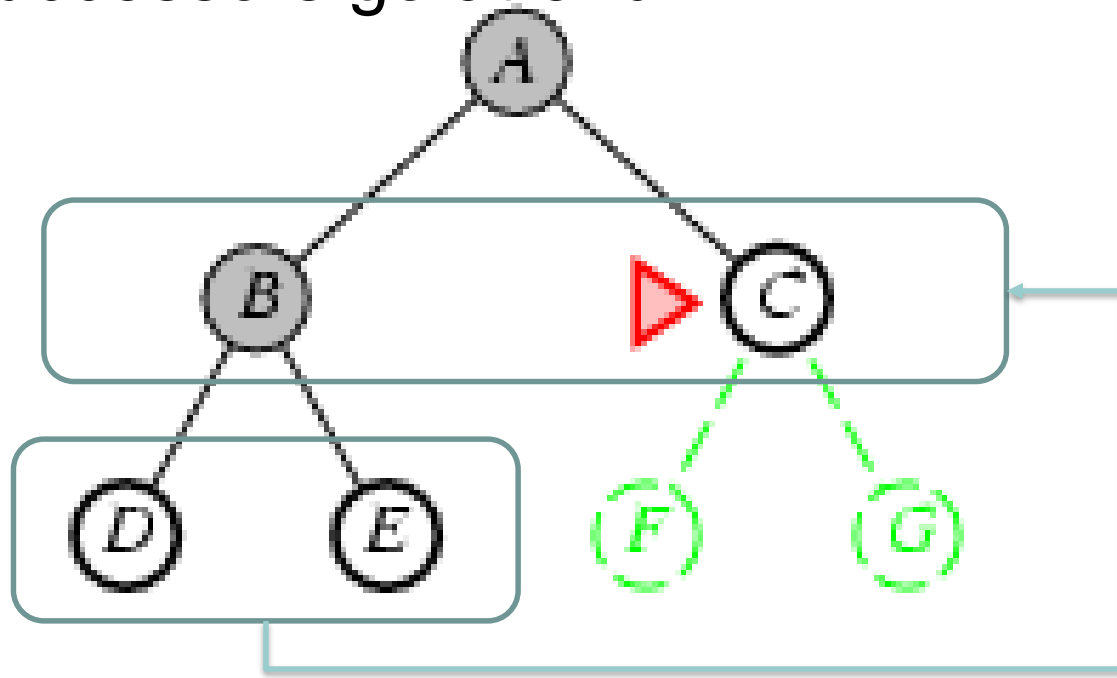
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



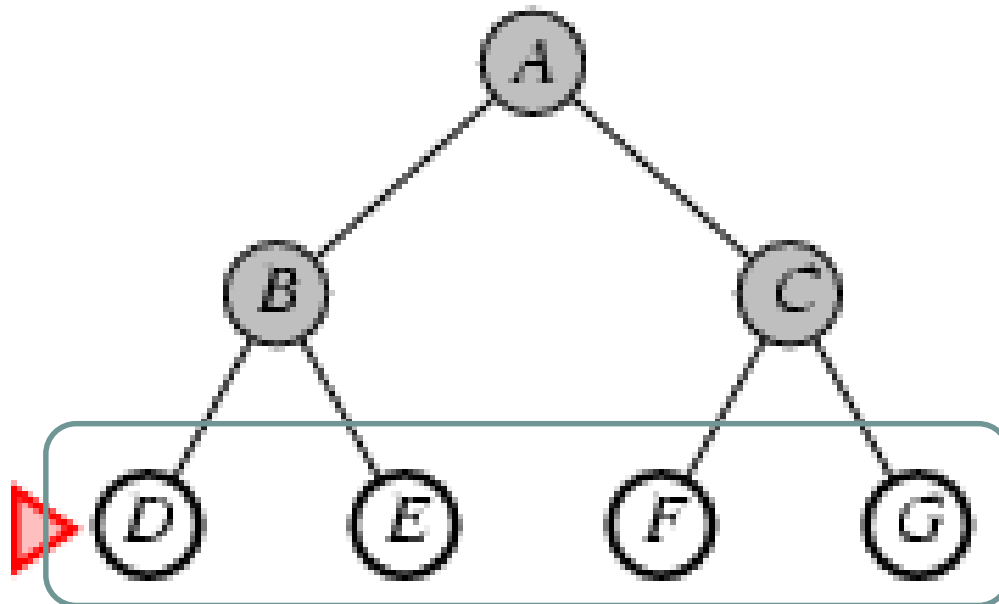
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - frontier* is a FIFO queue, i.e., new successors go at end



# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - frontier* is a FIFO queue, i.e., new successors go at end





# Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure  
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  frontier  $\leftarrow$  a FIFO queue with node as the only element  
  explored  $\leftarrow$  an empty set  
  loop do  
    if EMPTY?(frontier) then return failure  
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */  
    add node.STATE to explored  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      if child.STATE is not in explored or frontier then  
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)  
        frontier  $\leftarrow$  INSERT(child, frontier)
```



# Properties of breadth-first search

- Complete? Yes (if  $b$  is finite)
- Time?  $1+b+b^2+b^3+\dots +b^d + b(b^d-1) = O(b^{d+1})$
- Space?  $O(b^{d+1})$  (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
  
- **Space** is the bigger problem (more than time)

$b$ : maximum branching factor of the search tree  
(*maximum number of successors of any node*)

$d$ : depth of the least-cost solution

$m$ : maximum depth of the state space



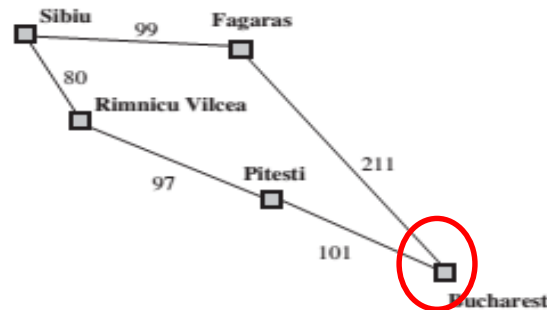
# Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *frontier* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost  $\geq \epsilon$
- Time? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\text{ceiling}(C^*/\epsilon)})$  where  $C^*$  is the cost of the optimal solution
- Space? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\text{ceiling}(C^*/\epsilon)})$
- Optimal? Yes – nodes expanded in increasing order of  $g(n)$

# Uniform-cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

**Figure 3.14** Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.



**Figure 3.15** Part of the Romania state space, selected to illustrate uniform-cost search.



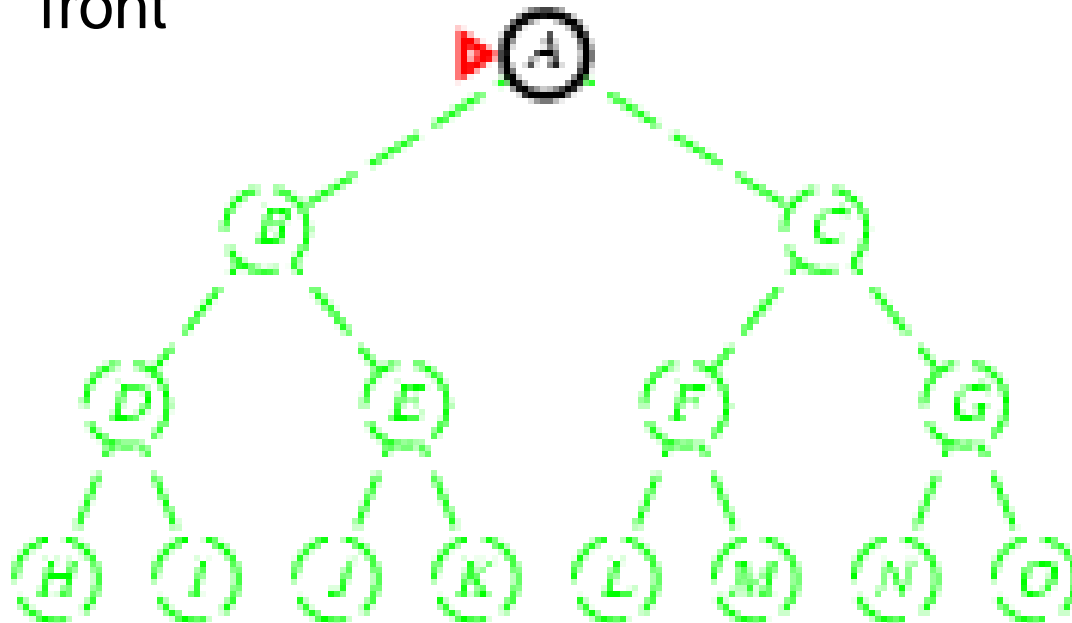


# Uniform-cost Search

- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The **least-cost node**, Rimnicu Vilcea, is expanded next, adding Pitesti with cost  $80 + 97 = 177$ .
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost  $99 + 211 = 310$ . Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost  $80 + 97 + 101 = 278$ .
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded.
- Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

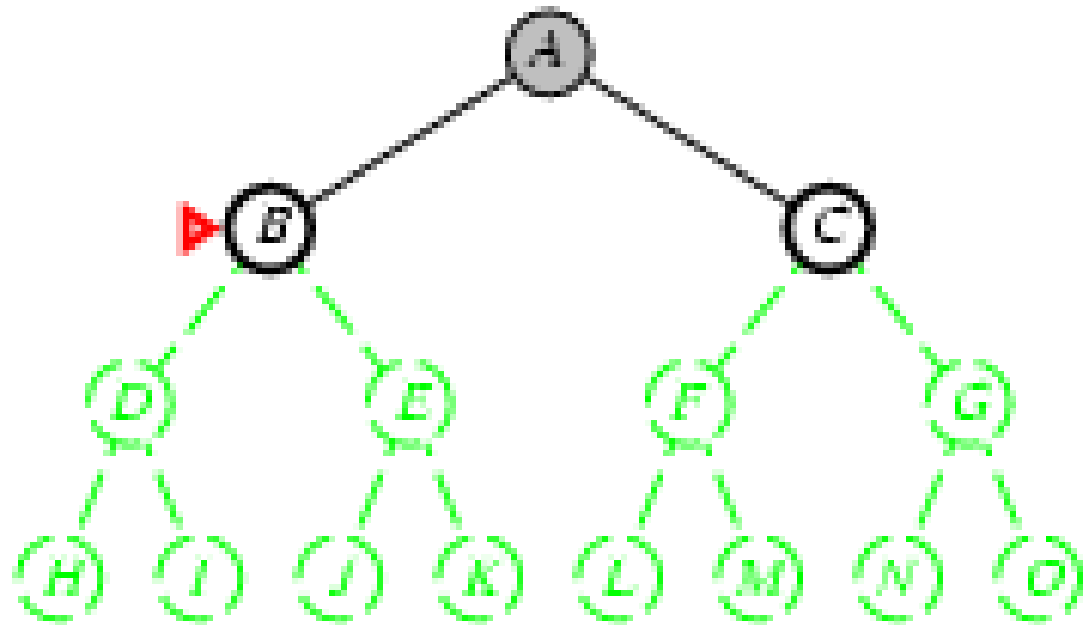
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



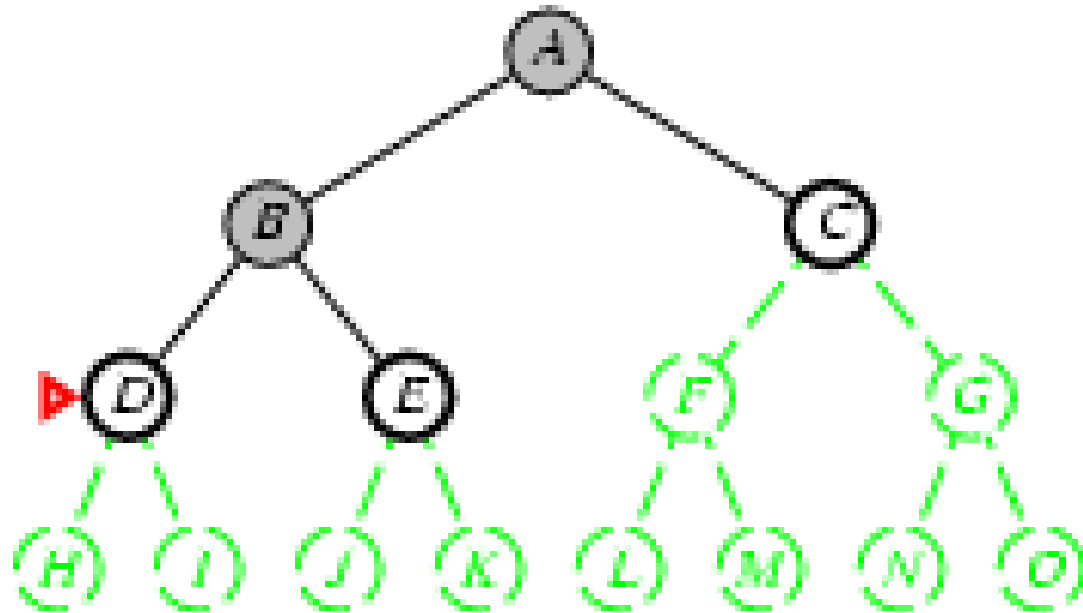
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - frontier* = LIFO queue, i.e., put successors at front



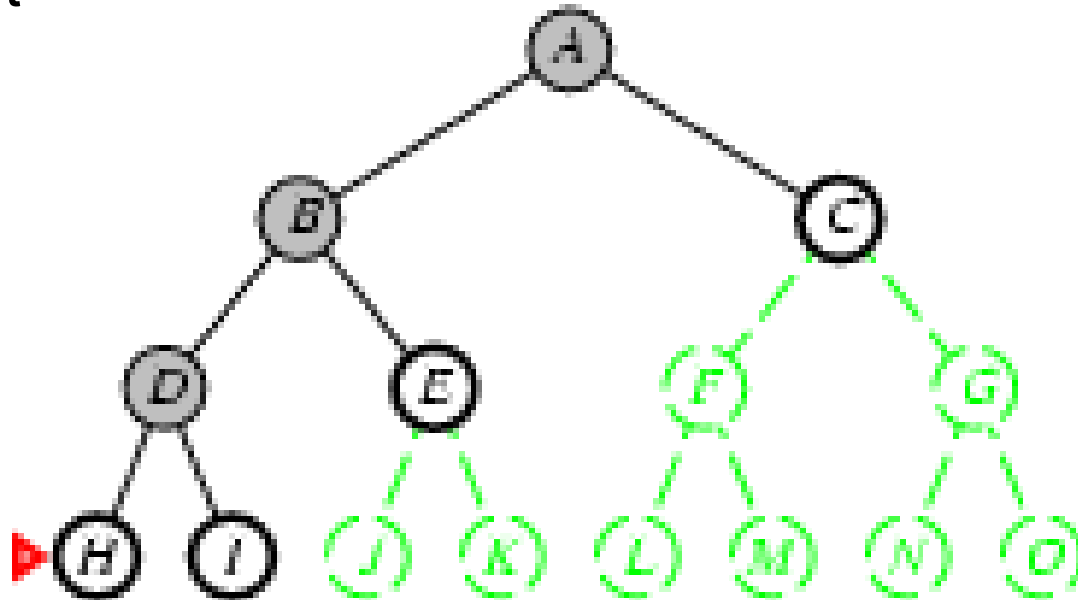
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



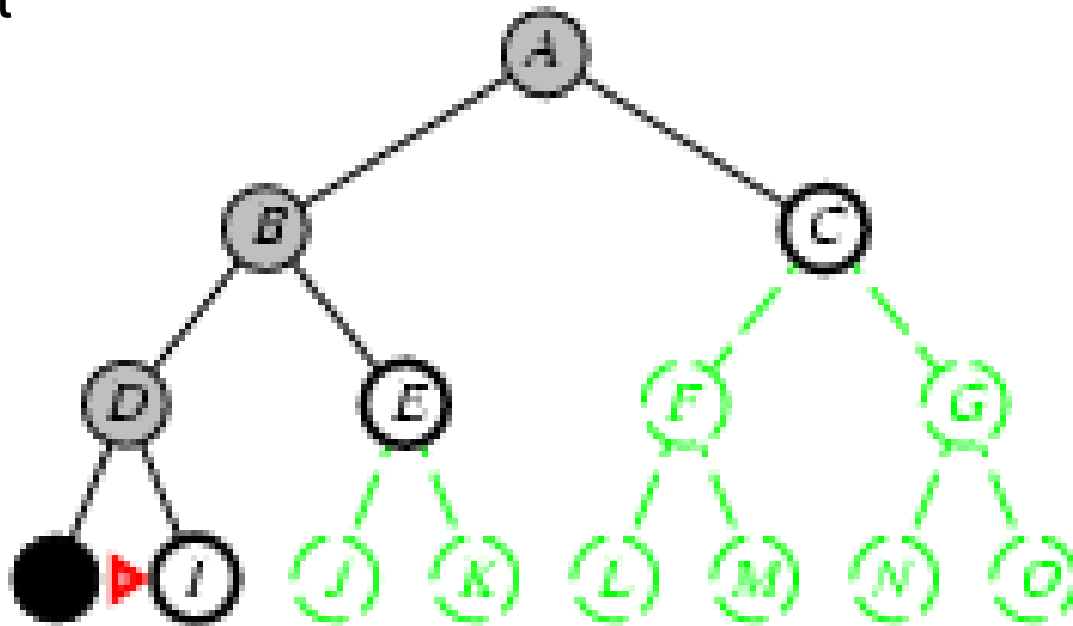
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



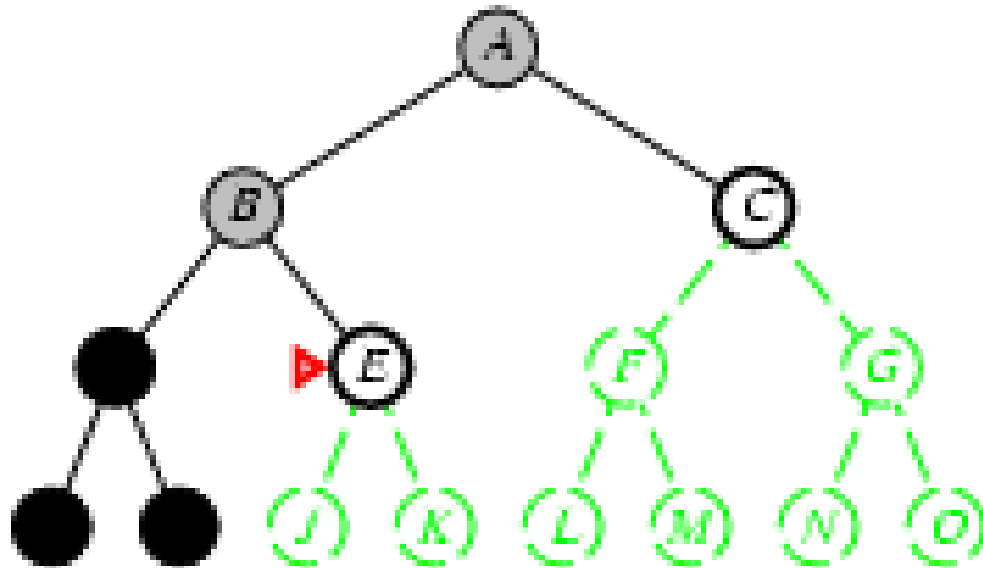
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



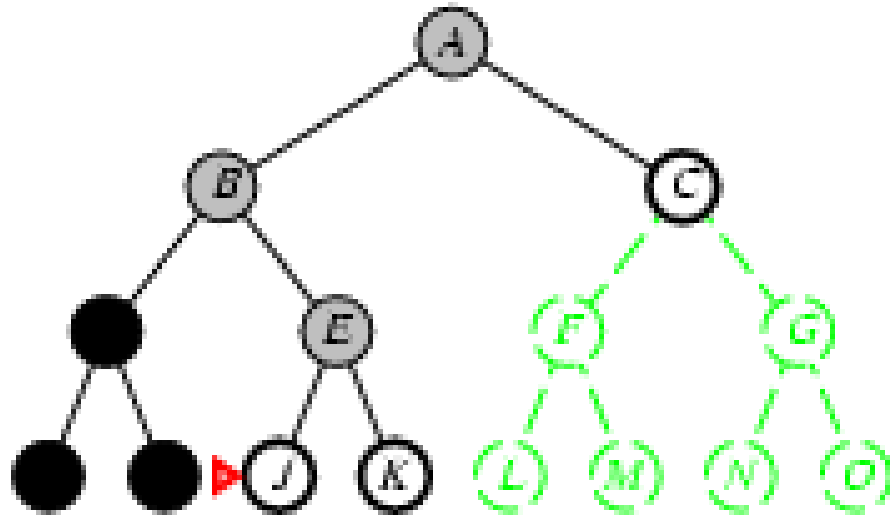
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



# Depth-first search

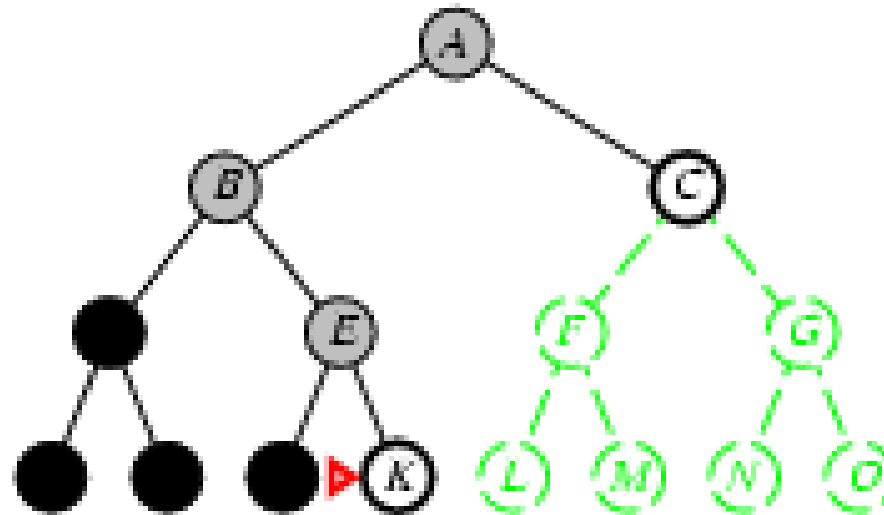
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front





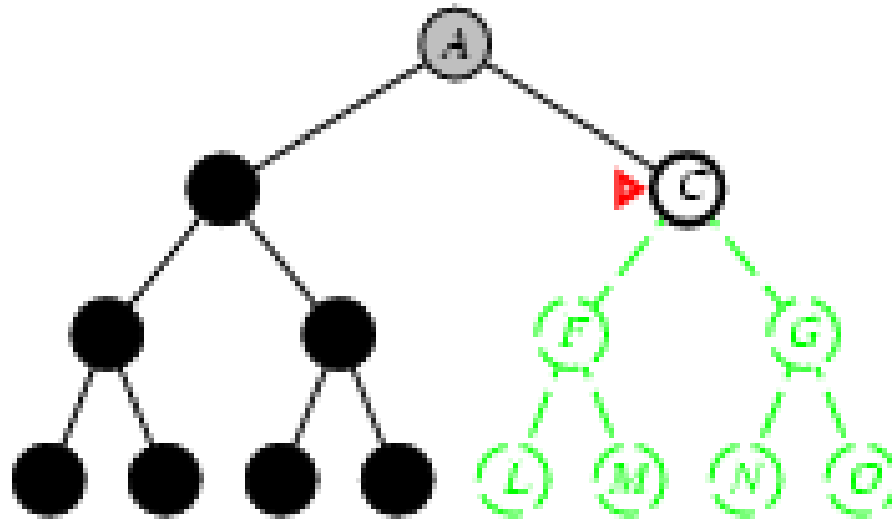
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



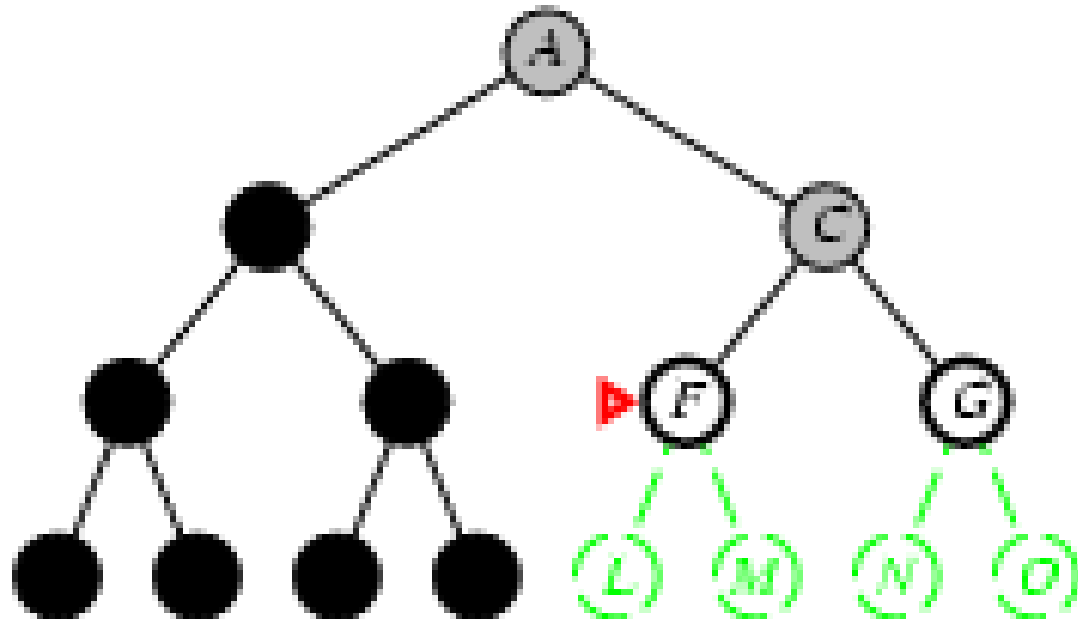
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



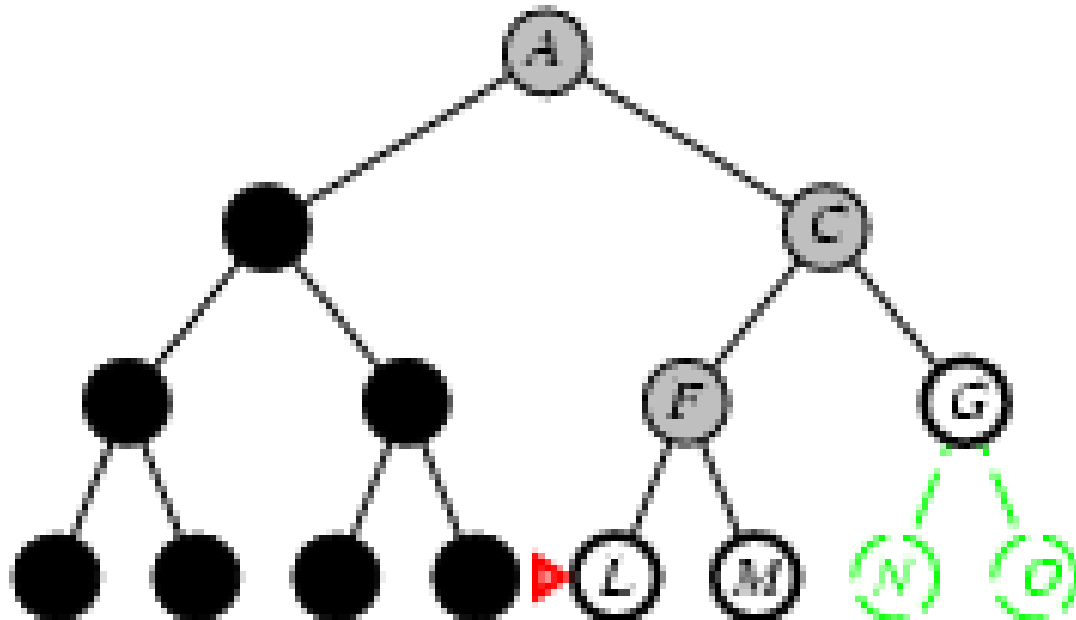
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



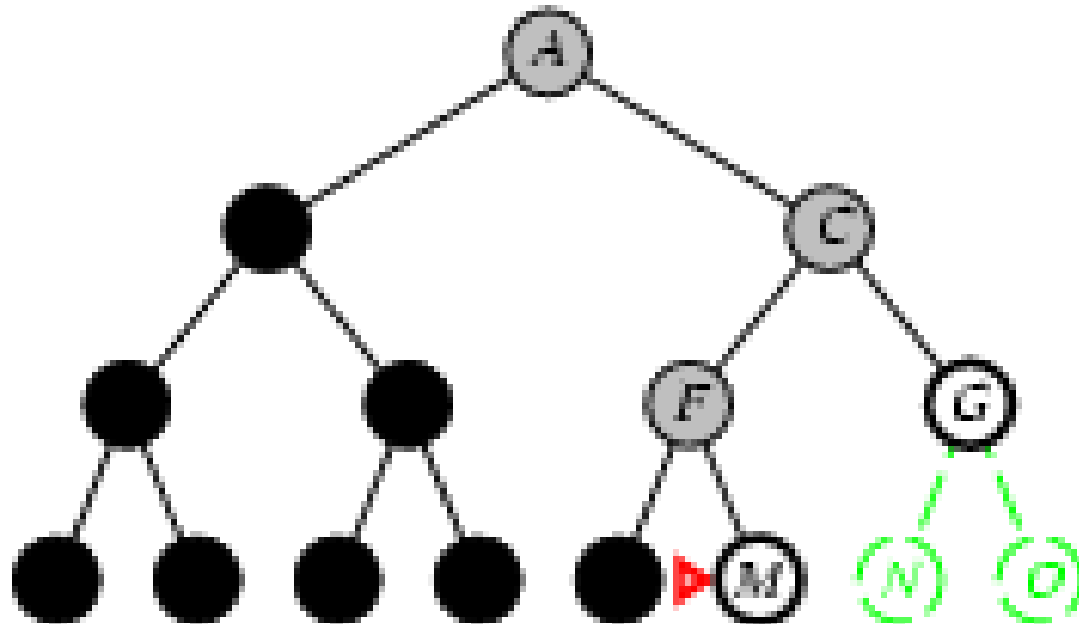
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front





# Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - complete in finite spaces
- Time?  $O(b^m)$ : terrible if  $m$  is much larger than  $d$ 
  - but if solutions are dense, may be much faster than breadth-first
- Space?  $O(bm)$ , i.e., linear space!
- Optimal? No

$b$ : maximum branching factor of the search tree (***maximum number of successors of any node***)

$d$ : depth of the least-cost solution

$m$ : maximum depth of the state space



# Depth-limited search

= depth-first search with depth limit  $l$ ,  
i.e., nodes at depth  $l$  have no successors

- Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff  
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)  
  
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff  
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)  
  else if limit = 0 then return cutoff  
  else  
    cutoff_occurred?  $\leftarrow$  false  
    for each action in problem.ACTIONS(node.STATE) do  
      child  $\leftarrow$  CHILD-NODE(problem, node, action)  
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)  
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true  
      else if result  $\neq$  failure then return result  
  if cutoff_occurred? then return cutoff else return failure
```



# Iterative deepening search

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



# Iterative deepening search / =0

Limit = 0



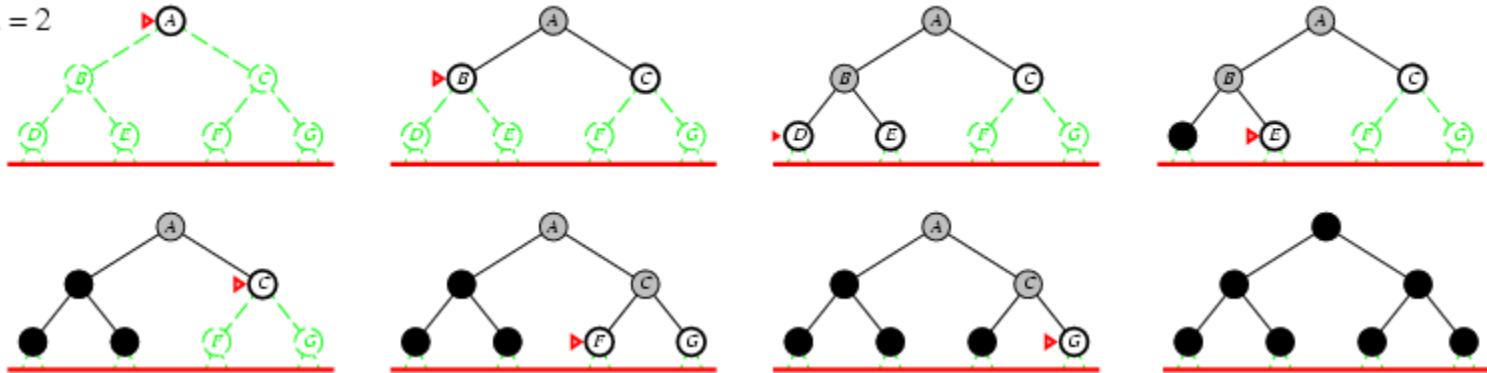
# Iterative deepening search / =1

Limit = 1



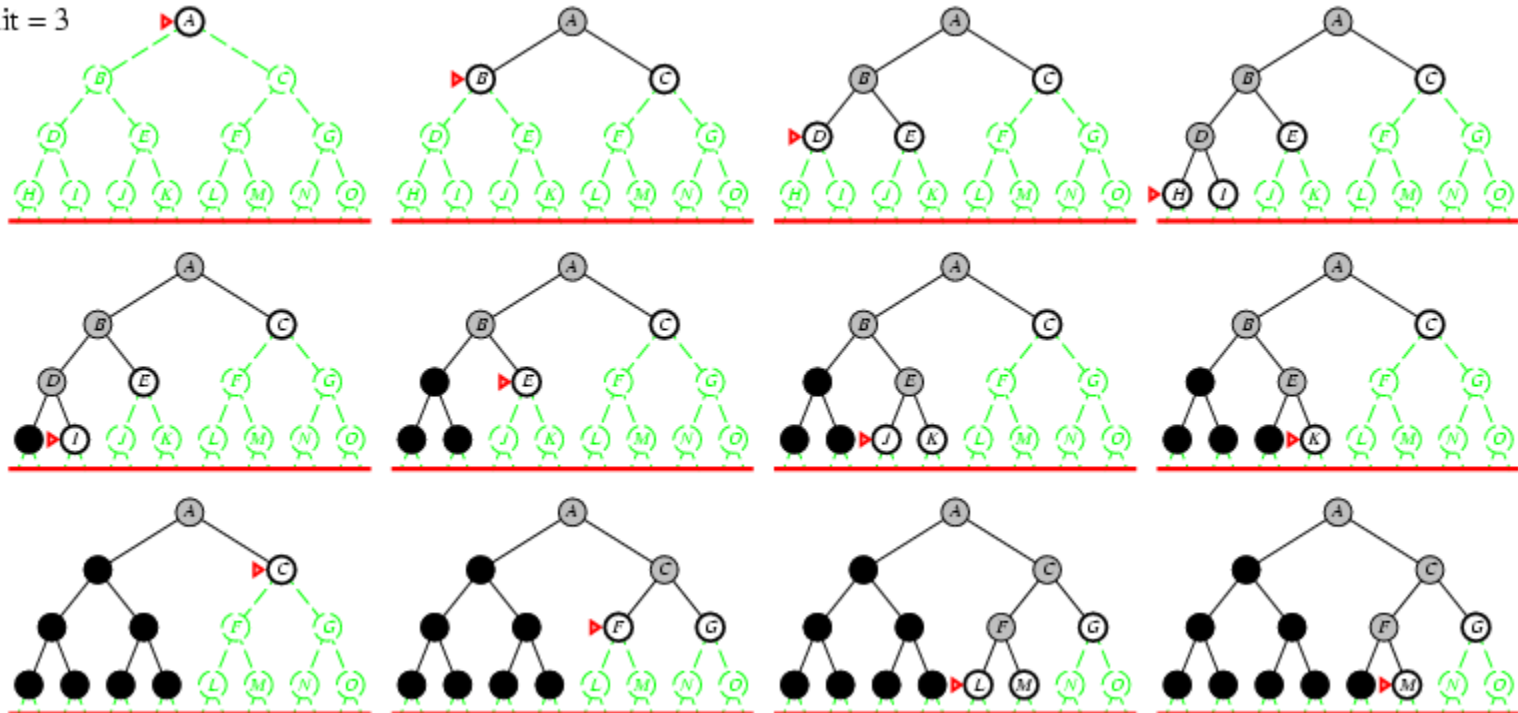
# Iterative deepening search / =2

Limit = 2



# Iterative deepening search / =3

Limit = 3





# Iterative deepening search

- Number of nodes generated in a depth-limited search to depth  $d$  with branching factor  $b$ :

$$N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth  $d$  with branching factor  $b$ :

$$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For  $b = 10$ ,  $d = 5$ ,
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$
- Overhead =  $(123,456 - 111,111)/111,111 = 11\%$



# Properties of iterative deepening search

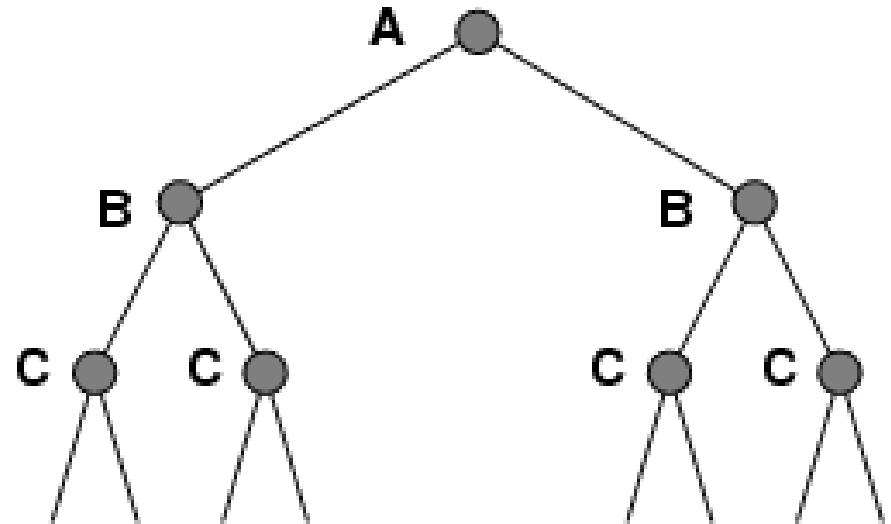
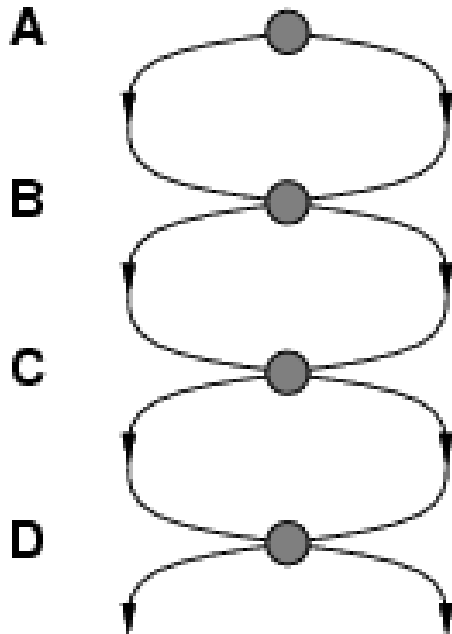
- Complete? Yes
- Time?  $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space?  $O(bd)$
- Optimal? Yes, if step cost = 1

# Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

# Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!







# Informed (Heuristic) Search Strategies

- **best-first search.** Best-first search is an instance of the general **TREE-SEARCH** or **GRAPH-SEARCH** algorithm in which a node is selected for expansion based on an **evaluation function**,  $f(n)$ .
- The evaluation function is constructed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of  $f$  instead of  $g$  to order the priority queue.
- Most best-first algorithms include as a component of  $f$  a **heuristic function**, denoted  $h(n)$ :
- $h(n)$  = estimated cost of the cheapest path from the state at node  $n$  to a goal state.

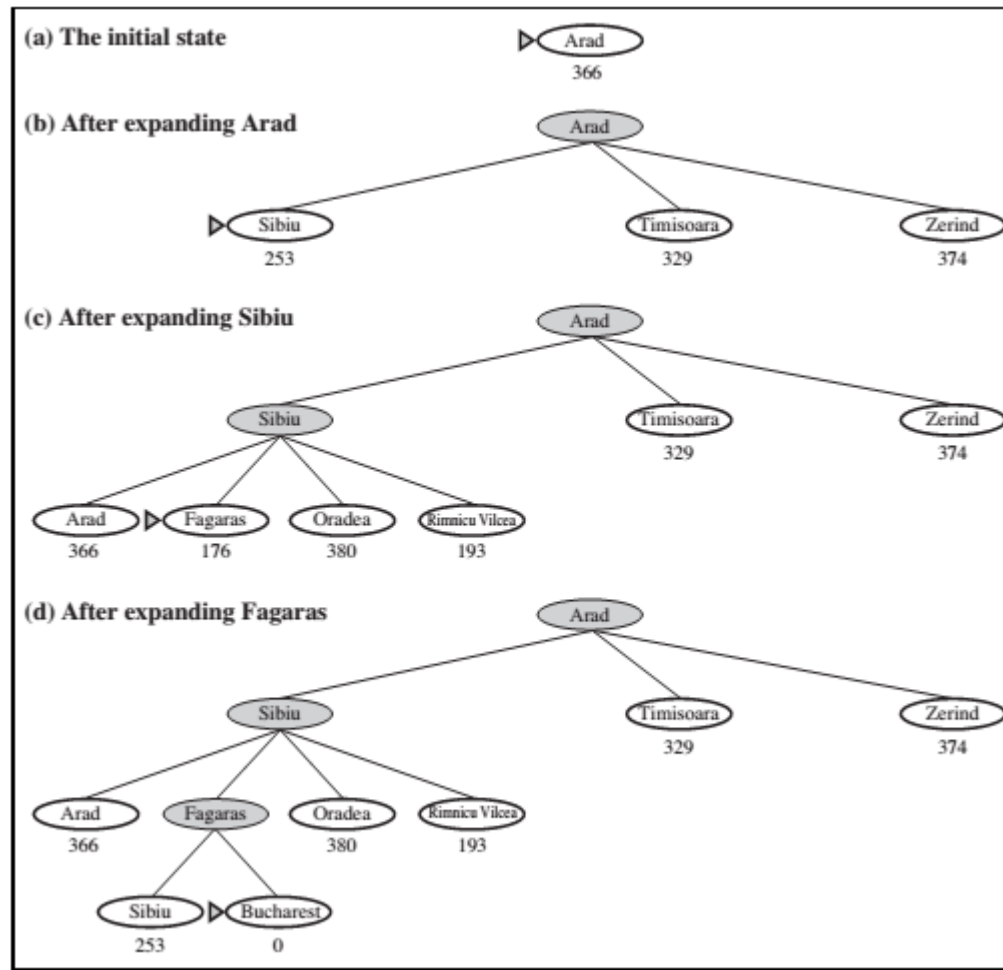
# Greedy best-first search

- Tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is,  $f(n) = h(n)$ .
- **straight line distance** heuristic (hSLD).
- If the goal is Bucharest, we need to know the straight-line distances to Bucharest
- For example,  $hSLD(\text{In}(\text{Arad})) = 366$ .

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

**Figure 3.22** Values of  $h_{SLD}$ —straight-line distances to Bucharest.

# Greedy best-first search





- It evaluates nodes by combining  $g(n)$ , the cost to reach the node, and  $h(n)$ , the cost to get from the node to the goal:  $f(n) = g(n) + h(n)$
- Since  $g(n)$  gives the path cost from the start node to node  $n$ , and  $h(n)$  is the estimated cost of the cheapest path from  $n$  to the goal, we have  $f(n)$  = estimated cost of the cheapest solution through  $n$  .

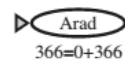


$A^*$

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of  $g(n) + h(n)$ .
- It turns out that this strategy is more than just reasonable: provided that the heuristic function  $h(n)$  satisfies certain conditions,  $A^*$  search is both ***complete and optimal***.

A\*

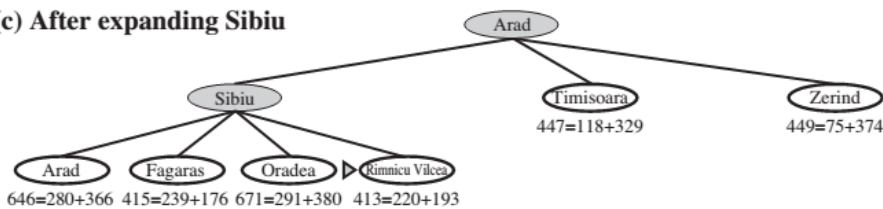
(a) The initial state



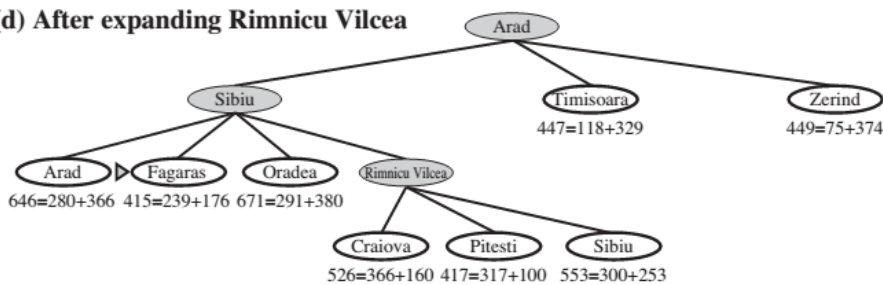
(b) After expanding Arad



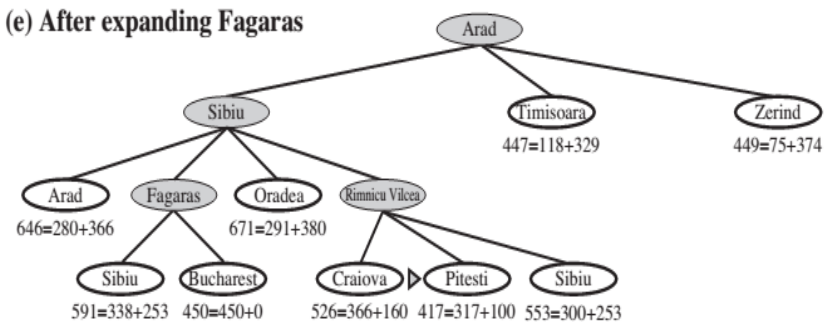
(c) After expanding Sibiu



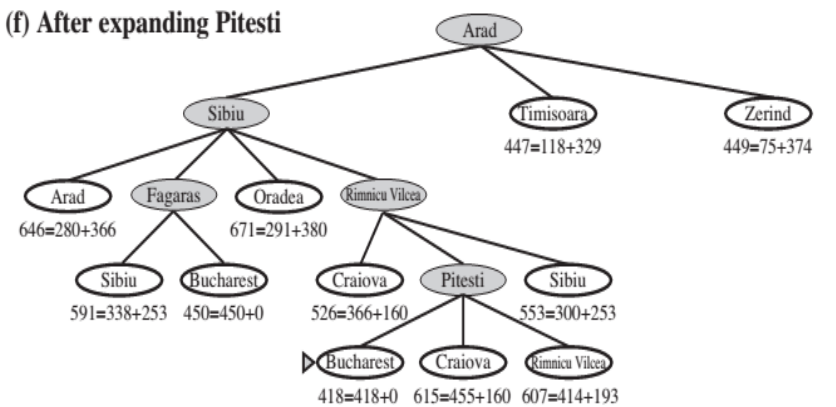
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti





# Summary

- Problem formulation usually requires **abstracting away real-world details** to define a state space that can feasibly be explored
- Variety of uninformed and informed search strategies
- **Iterative deepening search** uses only linear space and not much more time than other uninformed algorithms
- $A^*$  obtains cheapest solution to the goal