Artificial Intelligence
(IT3160E)

Than Quang Khoat

khoattq@soict.hust.edu.vn

School of Information and Communication Technology
Hanoi University of Science and Technology

2022

Content:

- Introduction of Artificial Intelligence
- Intelligent agent
- **Problem solving: Search**, Constraint satisfaction
  - **Uninformed search**
- Logic and reasoning
- Knowledge representation
- Machine learning

Problem solving by search

- Problem solving by search
  - Finds the sequence of actions that allow the desired state(s) to be reached
- Main steps:
  - **Goal** formulation
    - A set of final (target) states
  - **Problem** formulation
    - Given a goal, identify *actions* and *states* to consider
  - **Search** process
    - Consider possible sequences of actions
    - Choose the best sequence of actions
- Search algorithm
  - Input: A problem (to be solved)
  - Output: A solution, in the form of a sequence of actions to perform

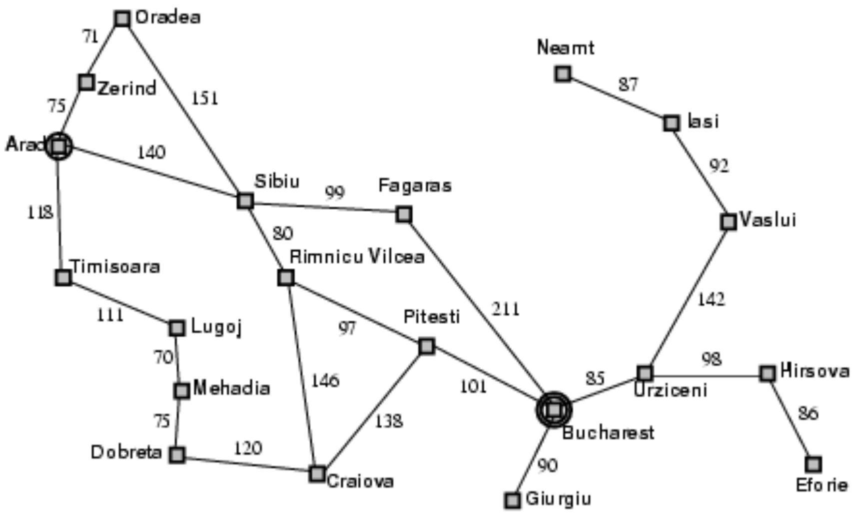
Problem-solving agents

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

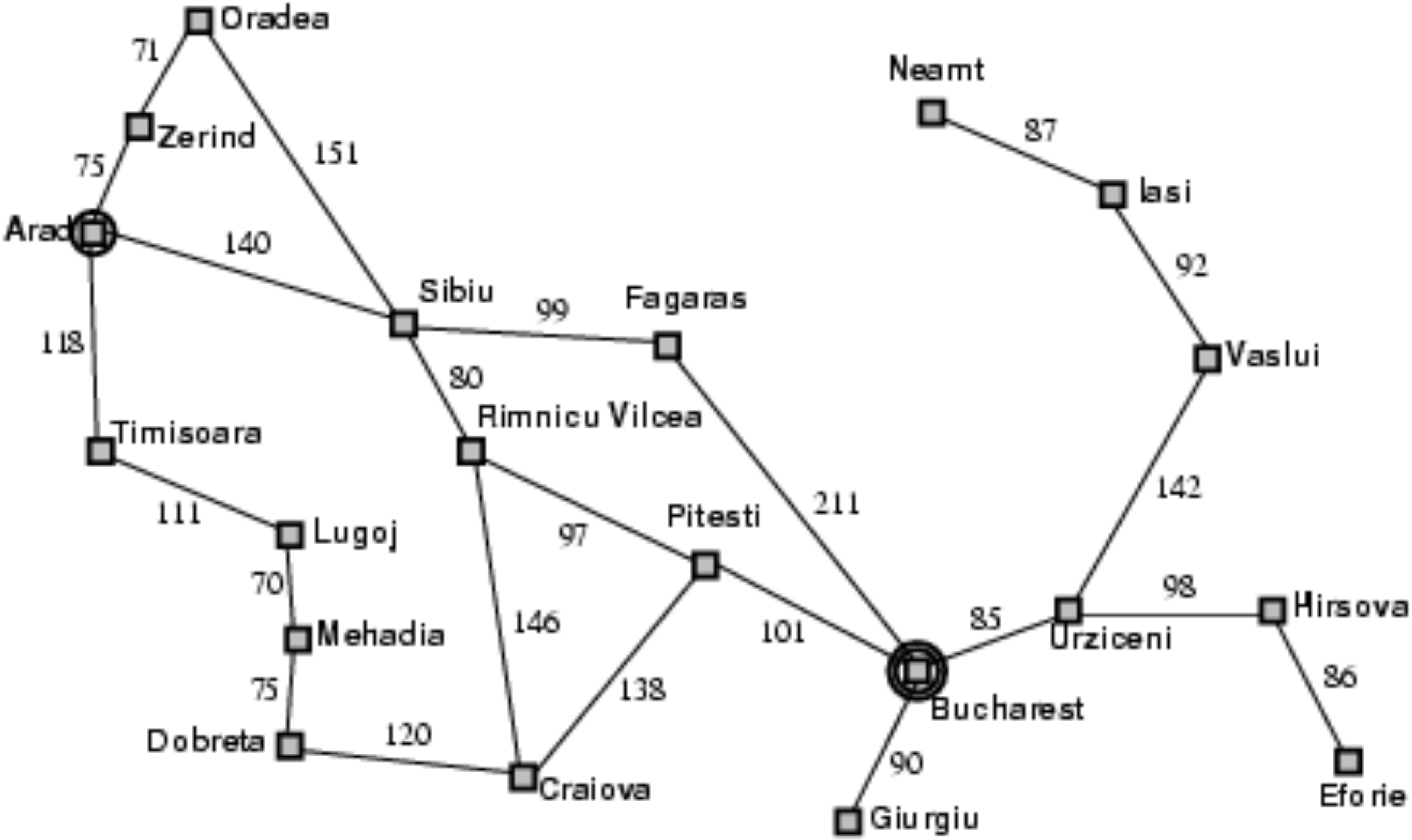
Problem solving by search: Example

- A tourist is on a tour in Romania
  - He is currently in Arad
  - Tomorrow, he has a flight departing from Bucharest
  - Now, he needs to move (i.e., drive) from Arad to Bucharest
- Formulate **goal**:
  - Must be in Bucharest
- Formulate **problem**:
  - *States*: cities (passing through)
  - *Actions*: driving between cities



- Find **solution**:
  - The sequence of cities to pass through, for example: Arad, Sibiu, Fagaras, Bucharest

Problem solving by search: Example

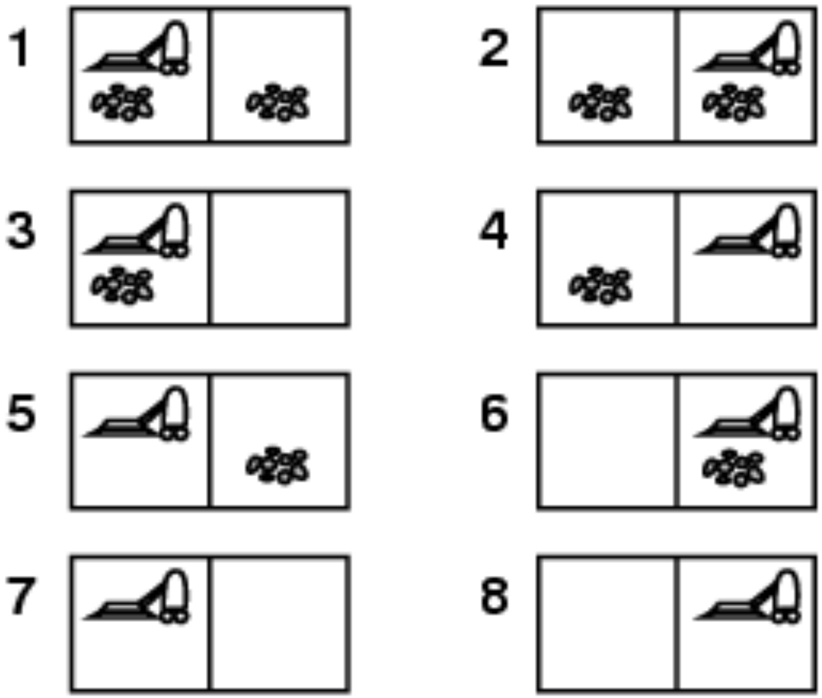


Search problem types

- **Deterministic, fully observable → Single-state problem**
  - The agent knows exactly which (next) state it will be in
  - Solution: A sequence of actions
- **Non-observable → Sensorless problem**
  - The agent may not know what state it is currently in
  - Solution: A sequence of actions
- **Non-deterministic** and/or **partially observable → Contingency problem**
  - Percepts provide new information about the current state
  - Solution: A contingent plan or a policy
  - Often interleave search and execution
- **Unknown state space → Exploration problem**

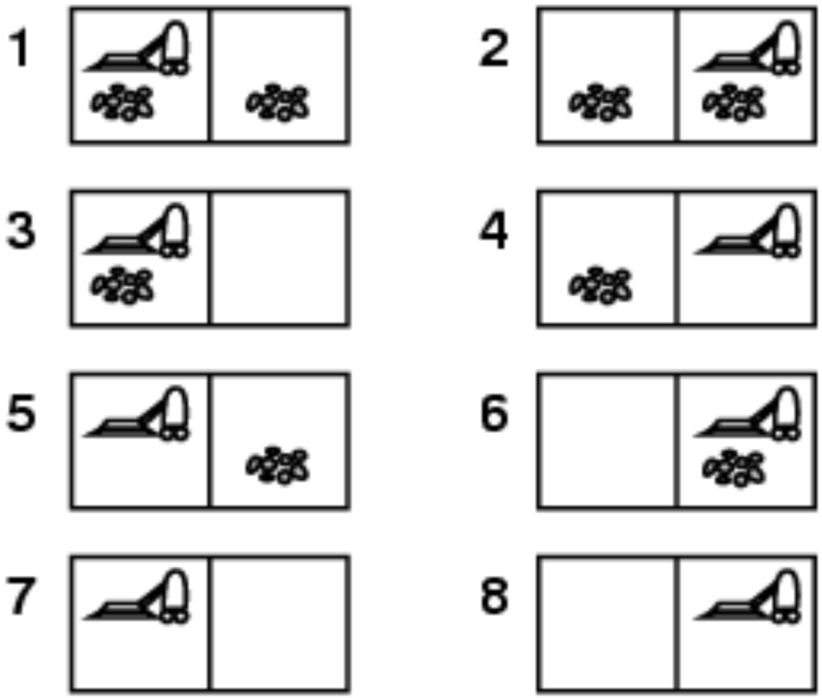
Example: Vacuum cleaner (1)

- **Single-state problem**
  - Start in state #5
- Solution?



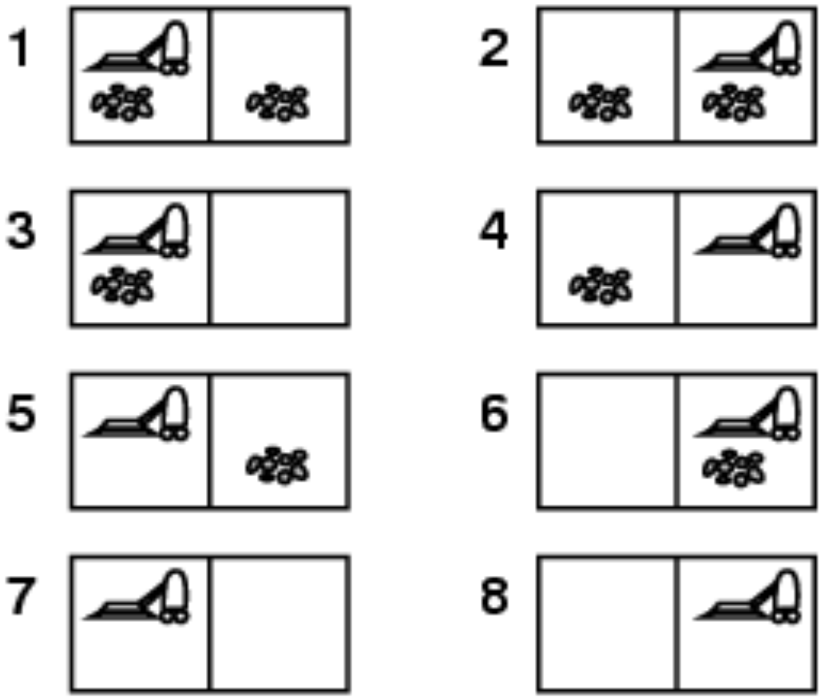
Example: Vacuum cleaner (2)

- **Single-state problem**
  - Start in state #5
- Solution?
  - *[Right, Suck]*



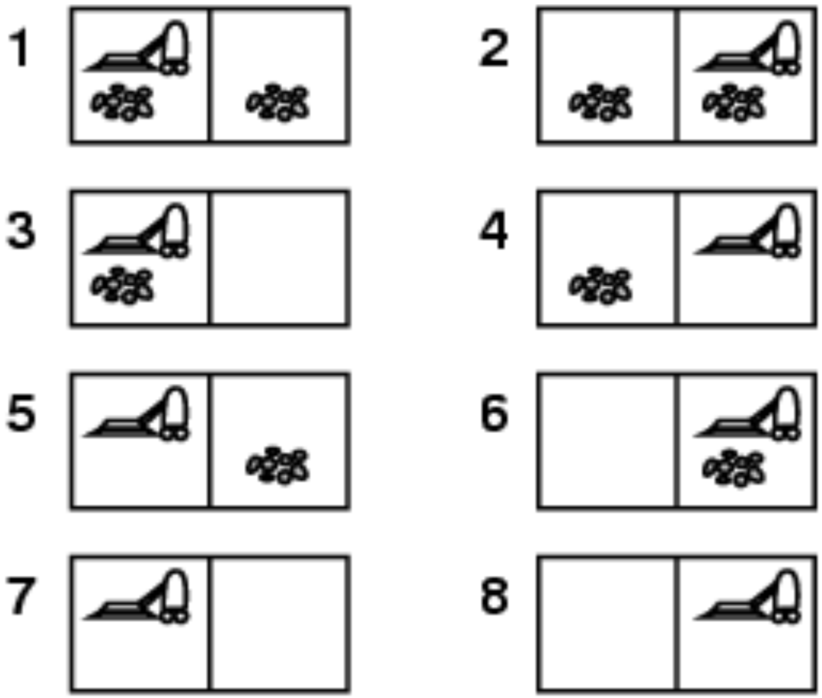
Example: Vacuum cleaner (3)

- **Sensorless problem**
  - Start in a state of {#1,#2,#3,#4,#5,#6,#7,#8}
  - Always start with moving right
- Solution?



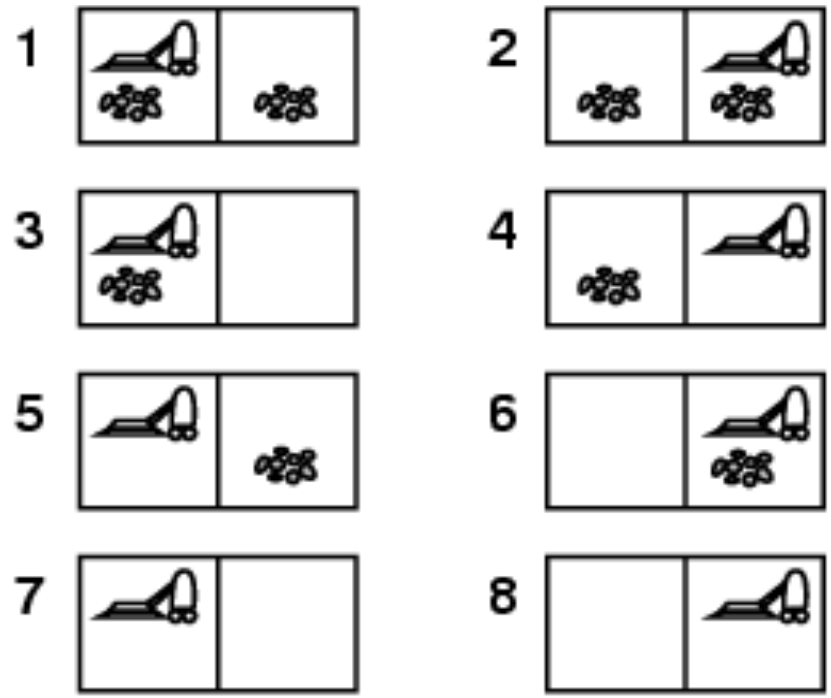
Example: Vacuum cleaner (4)

- **Sensorless problem**
  - Start in a state of {#1,#2,#3,#4,#5,#6,#7,#8}
  - Always start with moving right
- Solution?
  - *[Right, Suck, Left, Suck]*



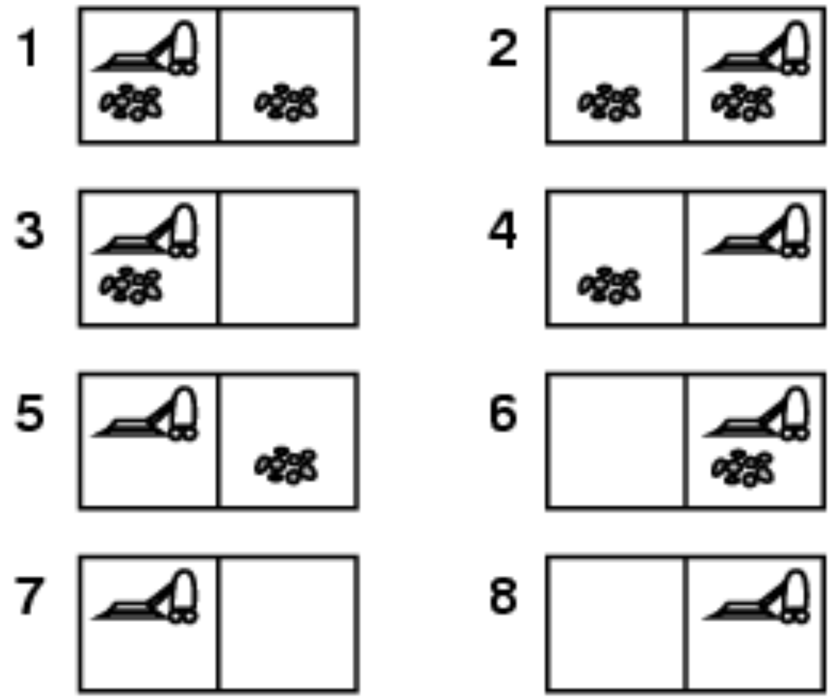
Example: Vacuum cleaner (5)

- **Contingency problem**
  - Start in state #5
  - Non-deterministic: Suck may dirty a clean carpet!
  - Partially observable: location, dirt at current location
- Solution?



Example: Vacuum cleaner (6)

- **Contingency problem**
  - Start in state #5
  - Non-deterministic: Suck may dirty a clean carpet!
  - Partially observable: location, dirt at current location
- Solution?
  - *[Right,* ***if*** *Dirt* ***then*** *Suck]*



Single-state problem formulation

A problem is defined by four items:

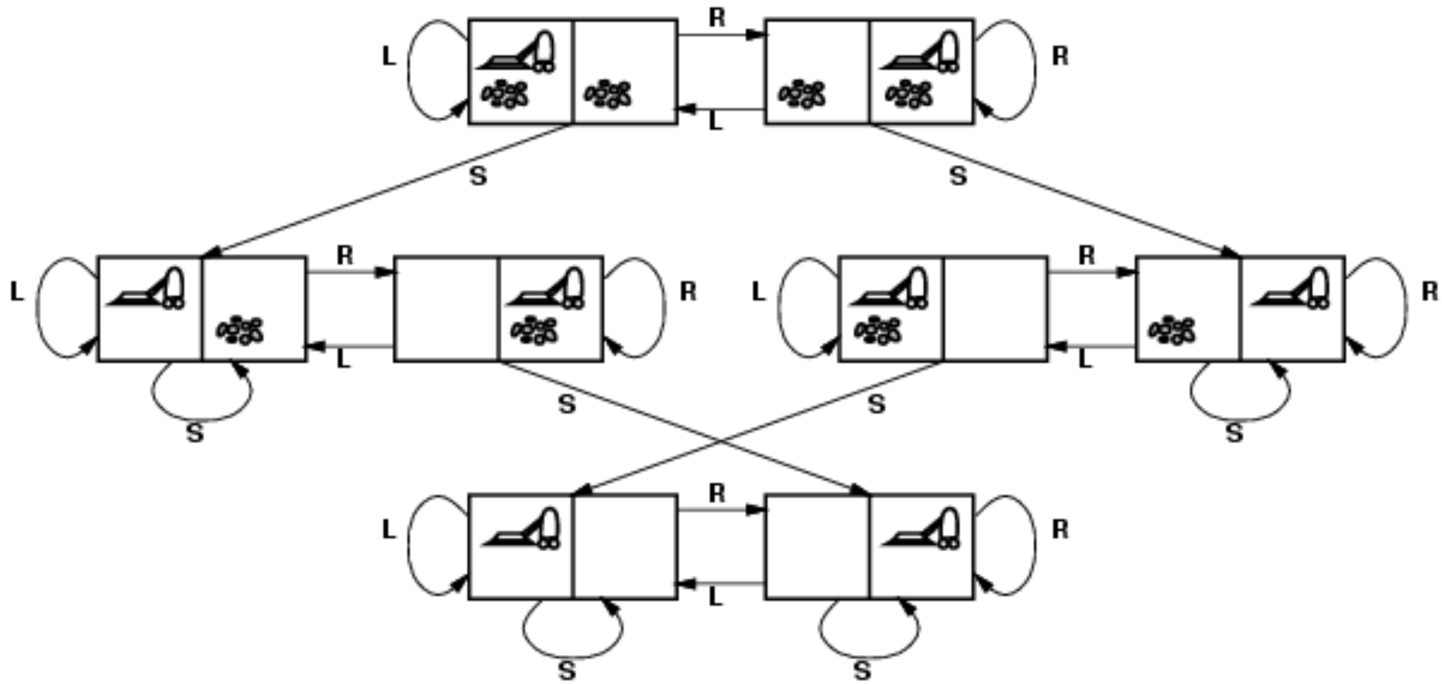
- Initial state
  - Example: "at Arad"
- Actions – Defined by the state-transition function: *S(trạng_thái_hiện_thời)* = tập các cặp <hành_động, trạng_thái_tiếp_theo>
  - Example: *S(Arad)* = {<*Arad → Zerind*, *Zerind*>, … }
- Goal test
  - Direct – Example: Current state *x* = "at Bucharest"
  - Indirect – Example: *Check-mate(x)*, *Cleanliness*(x), etc.
- Path cost (additive)
  - Example: sum of distances, number of actions executed, etc.
  - *c(x,a,y)* ≥ 0 is the step cost, assumed to be ≥ 0 – the cost for applying action *a* to transition from state *x* to state *y*
- A *solution* is a sequence of actions leading from the initial state to a goal state

Selecting a state space

- Real world is often complex
  → The state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - Example: Action "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any actual state must be reachable from other one
- (Abstract) solution = A set of real paths that are solutions in the real world

State space graph (1)

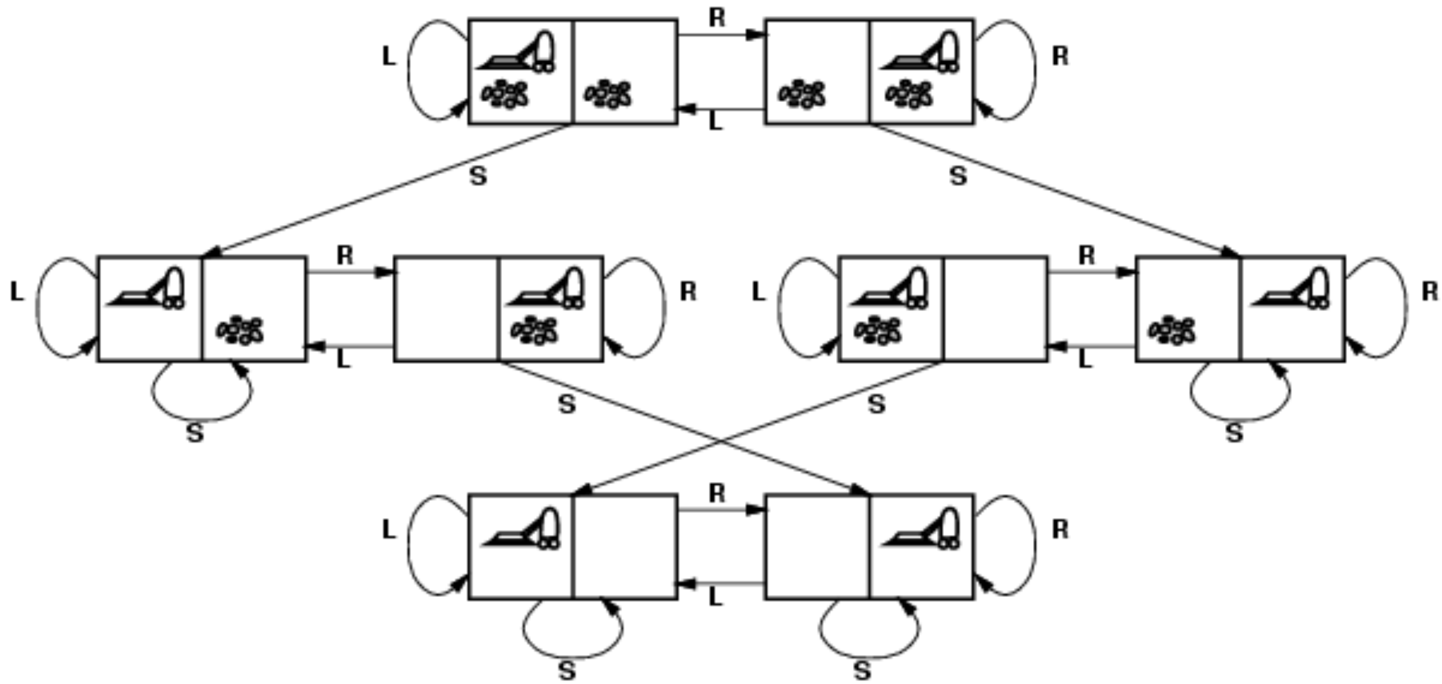
Vacuum cleaner problem



- States?
- Actions?
- Goal test?
- Path cost?

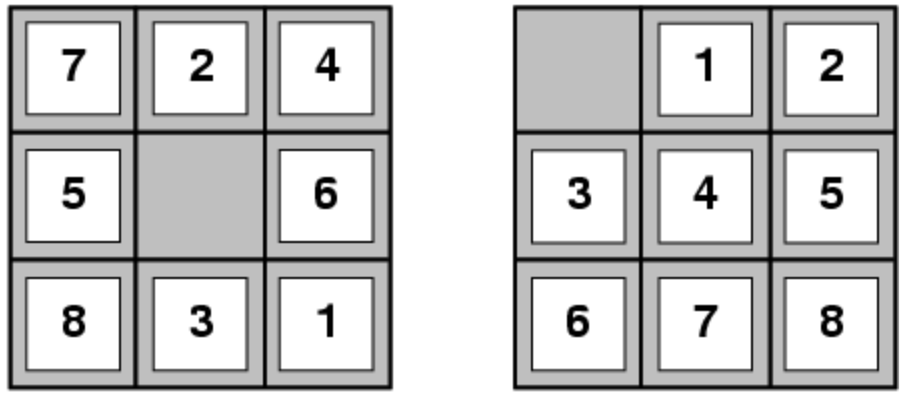
State space graph (2)

Vacuum cleaner problem



- States? *Dirt and robot location*
- Actions? *Left, Right, Suck*
- Goal test? *No dirt at all locations*
- Path cost? *1 per action*

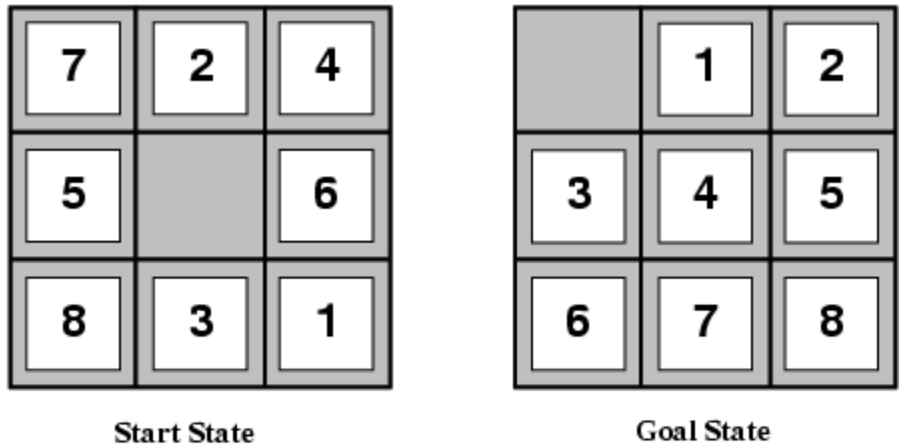
Example: The 8-puzzle (1)



Start State Goal State

- States?
- Actions?
- Goal test?
- Path cost?

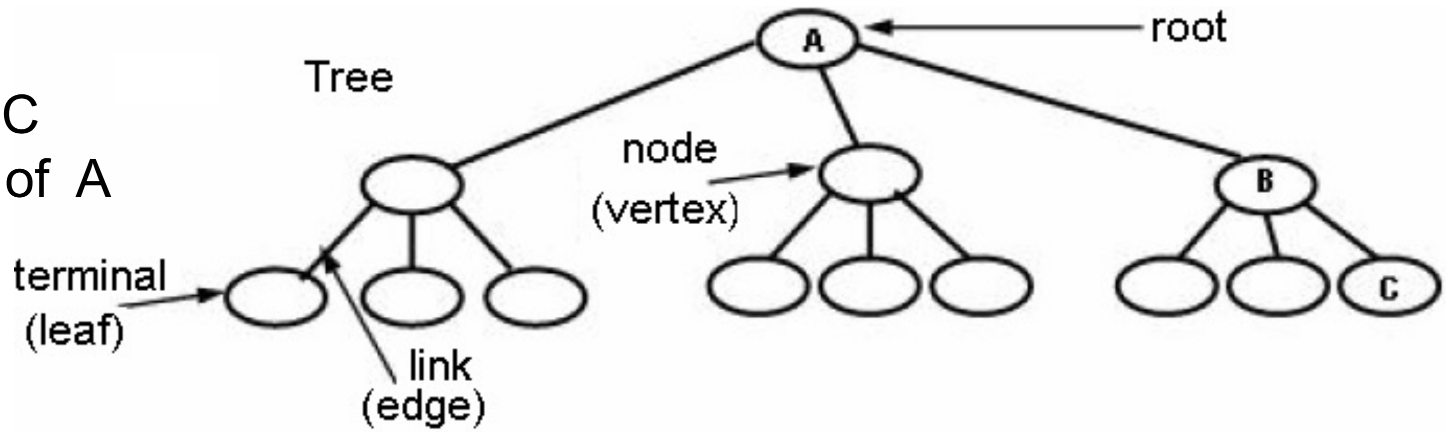
Example: The 8-puzzle (2)

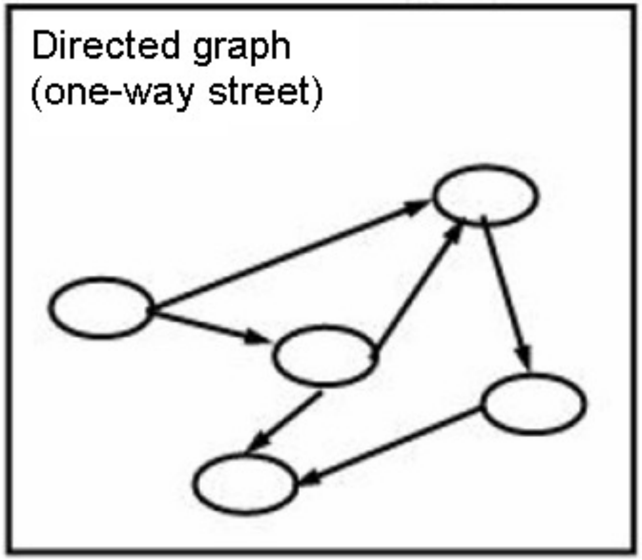


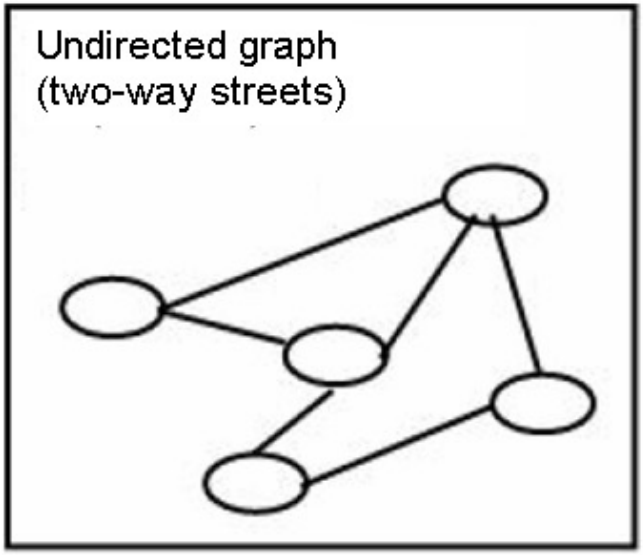
- States? *Locations of tiles*
- Actions? *Move blank left, right, up, down*
- Goal test? = *Goal state*
- Path cost? *1 per move*

Representation by tree and graph

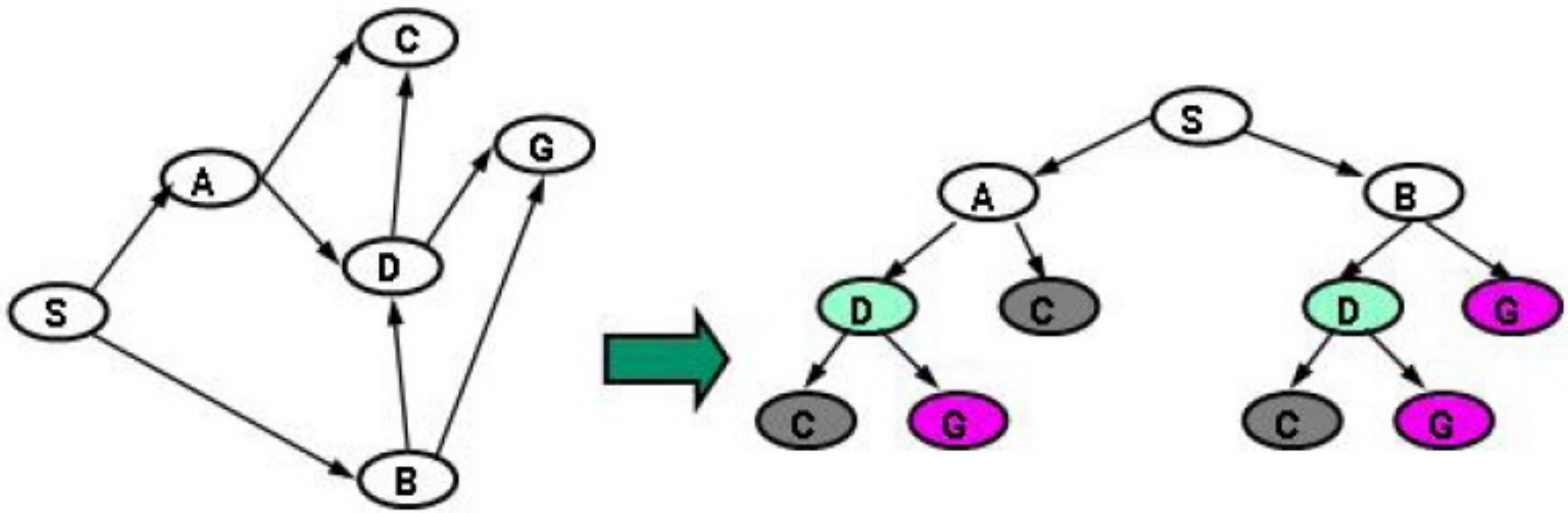
B is parent of C
C is child of B
A is ancestor of C
C is descendant of A







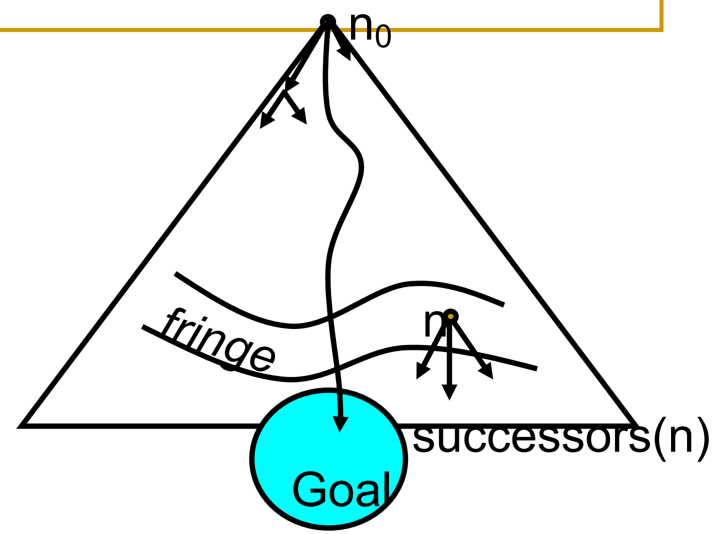
Search graph → Search tree



- Graph-based search problems can be transformed into tree-based search ones
  - Replace each undirected link (edge) with 2 oriented links (edges)
  - Eliminate loops that exist in the graph (to avoid not considering multiple times for a node in any path)

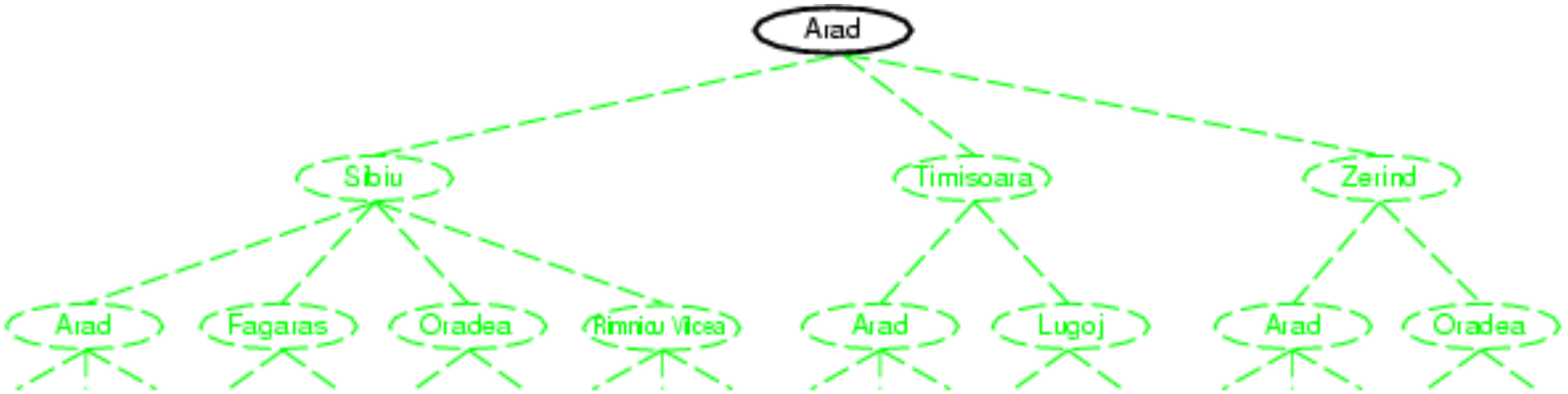
Tree-based search algorithms

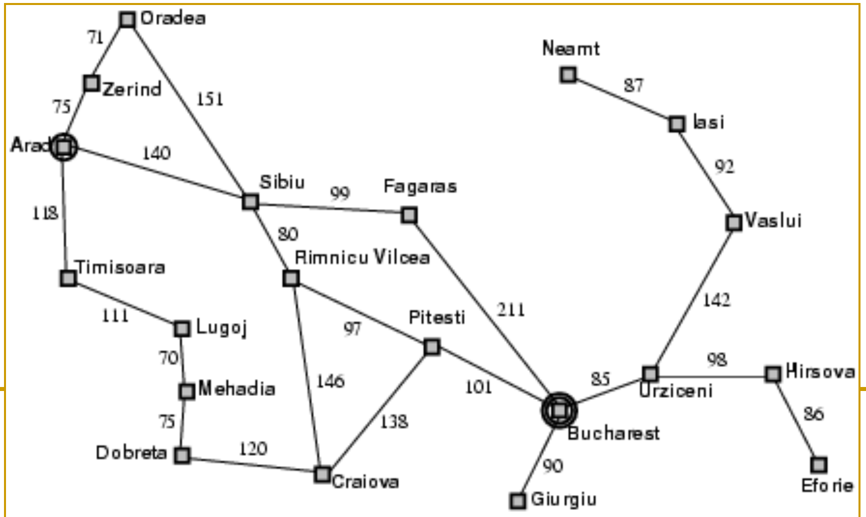
- Intuitive idea:
  - Explore (i.e., consider) the state space by generating successive states of the discovered (i.e., considered) ones
  - Also known as the method of expanding states



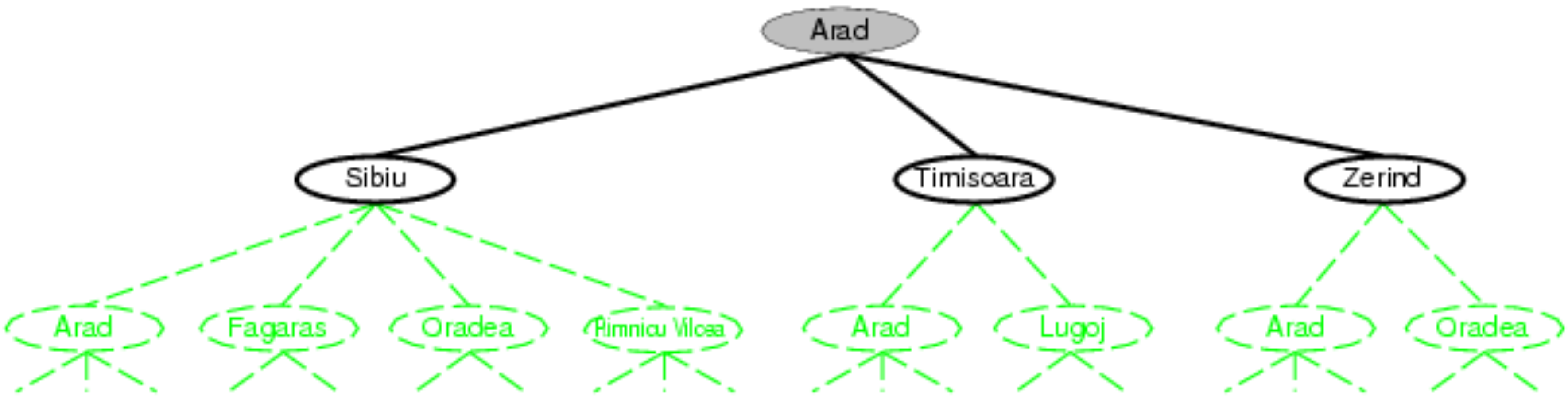

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
```

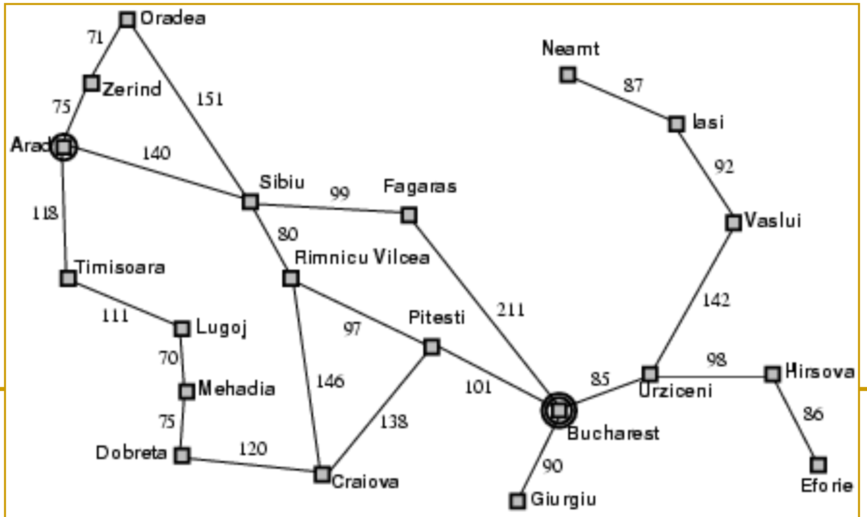
Tree-based search: Example (1)



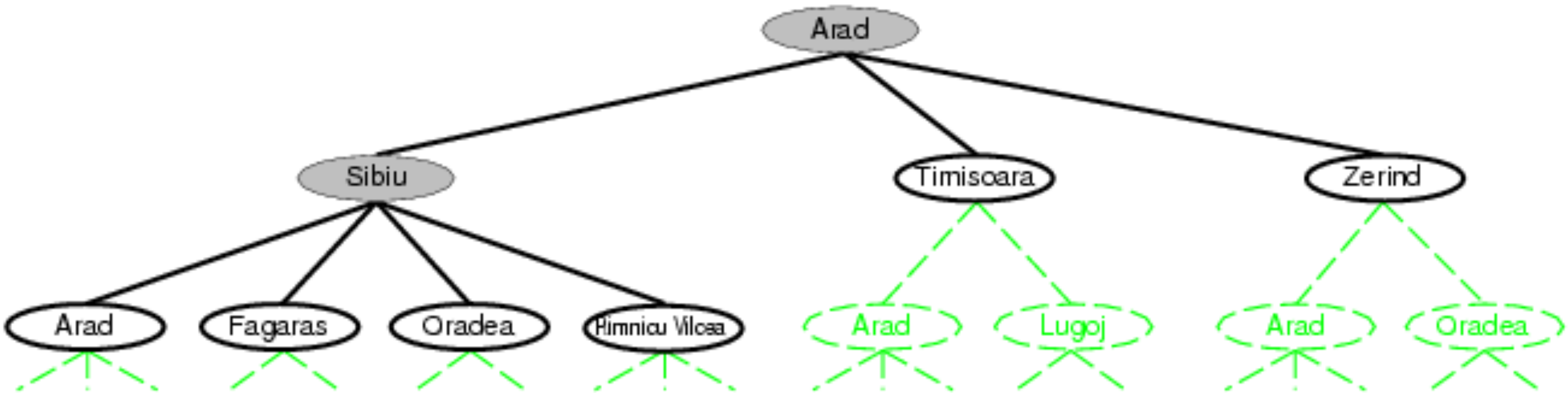


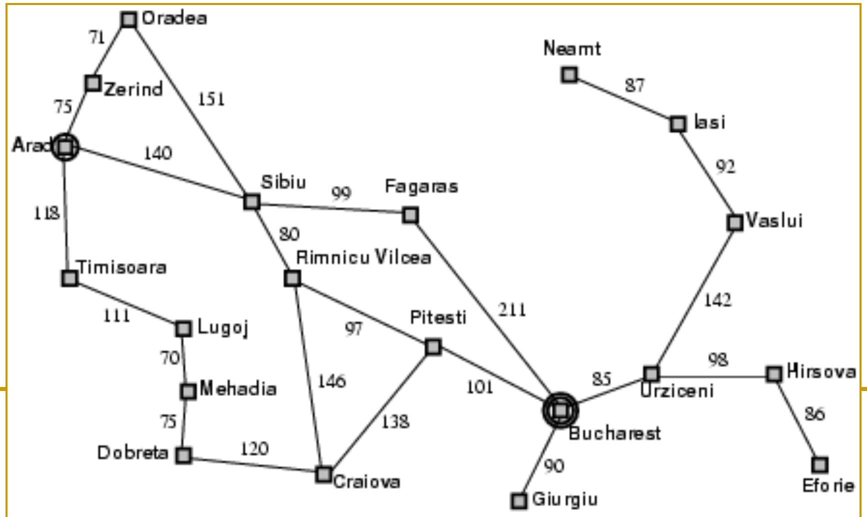
Tree-based search: Example (2)



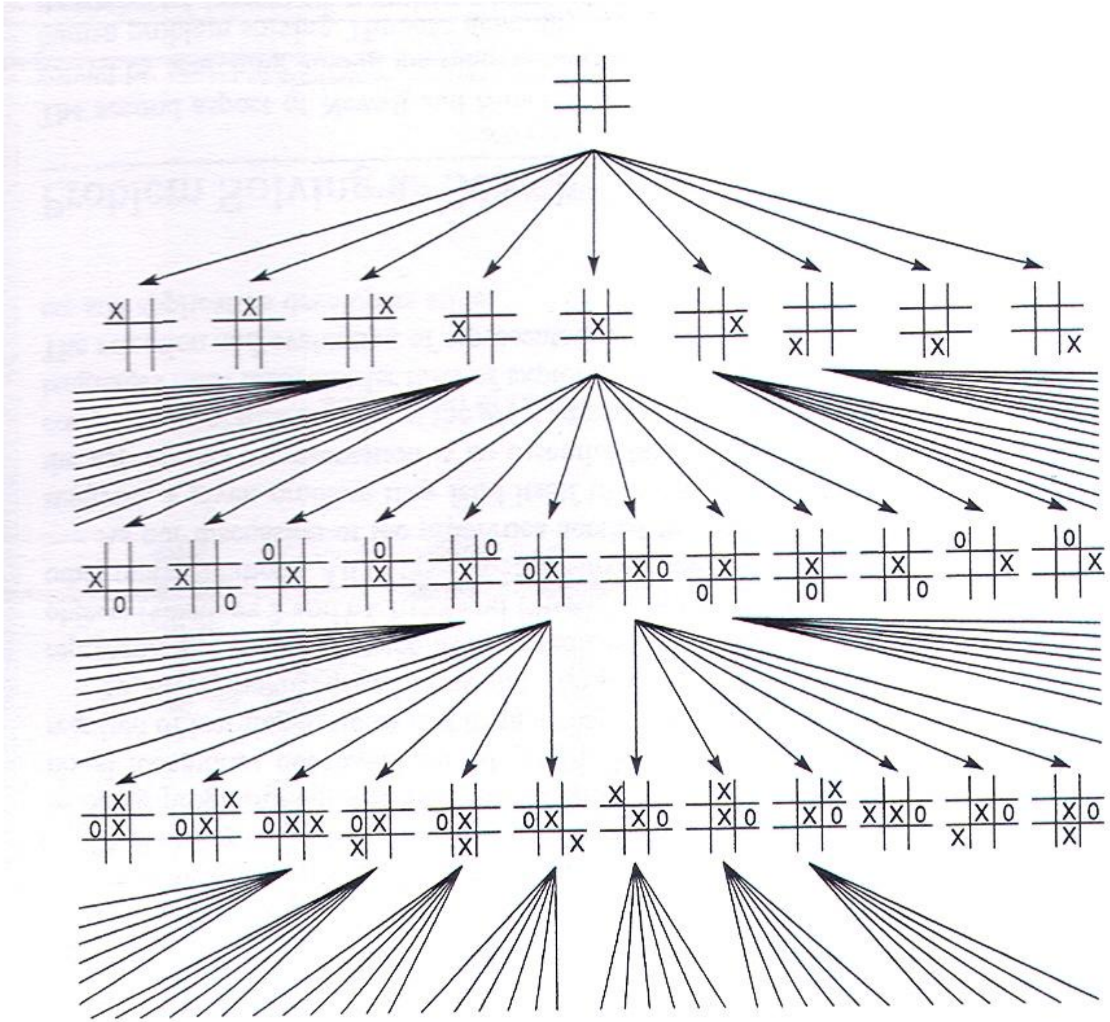


Tree-based search: Example (3)





Tic-Tac-Toe (i.e., Noughts and Crosses)



Tree-based search: General algorithm

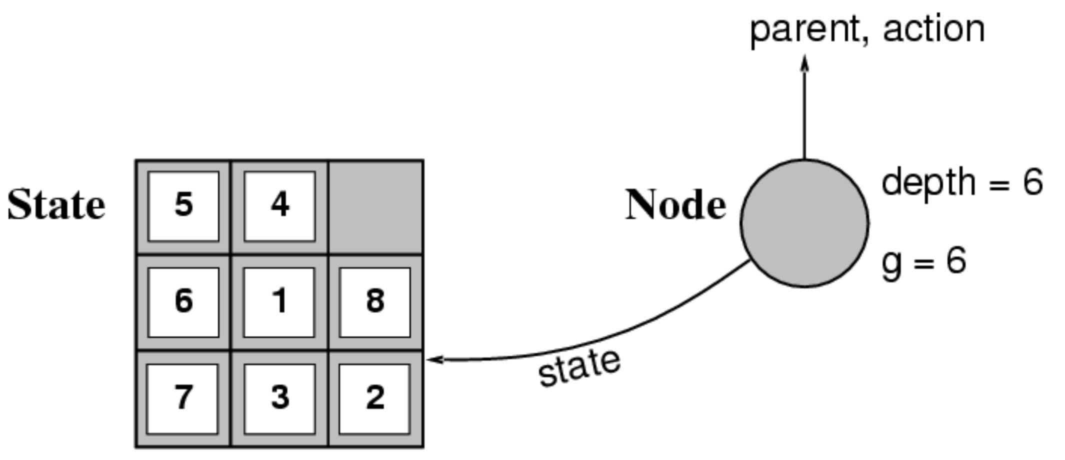
function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if *fringe* is empty **then return** failure
        node ← REMOVE-FRONT(*fringe*)
        if GOAL-TEST[*problem*](STATE[*node*]) **then return** SOLUTION(*node*)
        fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

---

function EXPAND(*node*, *problem*) **returns** a set of nodes
    successors ← the empty set
    for each *action*, *result* **in** SUCCESSOR-FN[*problem*](STATE[*node*]) **do**
        s ← a new NODE
        PARENT-NODE[*s*] ← *node*; ACTION[*s*] ← *action*; STATE[*s*] ← *result*
        PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*, *s*)
        DEPTH[*s*] ← DEPTH[*node*] + 1
        add *s* to *successors*
    return *successors*

Search tree representation

- 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*, *depth*, *path cost g(x)*



- The `Expand` function creates new nodes:
  - Assign the attribute values of the new node,
  - Use the `Successor-Fn` function to create the corresponding states

Search strategies

- A search strategy is defined by picking the **order of node expansion**
- Search strategies are evaluated along the following dimensions:
  - *Completeness*: Does it always find a solution if one exists?
  - *Time complexity*: The number of nodes generated
  - *Space complexity*: The 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:* The maximum branching factor of the search tree
  - *d:* The depth of the least-cost solution
  - *m*: The maximum depth of the state space (i.e., the depth of the search tree) – may be $+\infty$

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 (BFS)

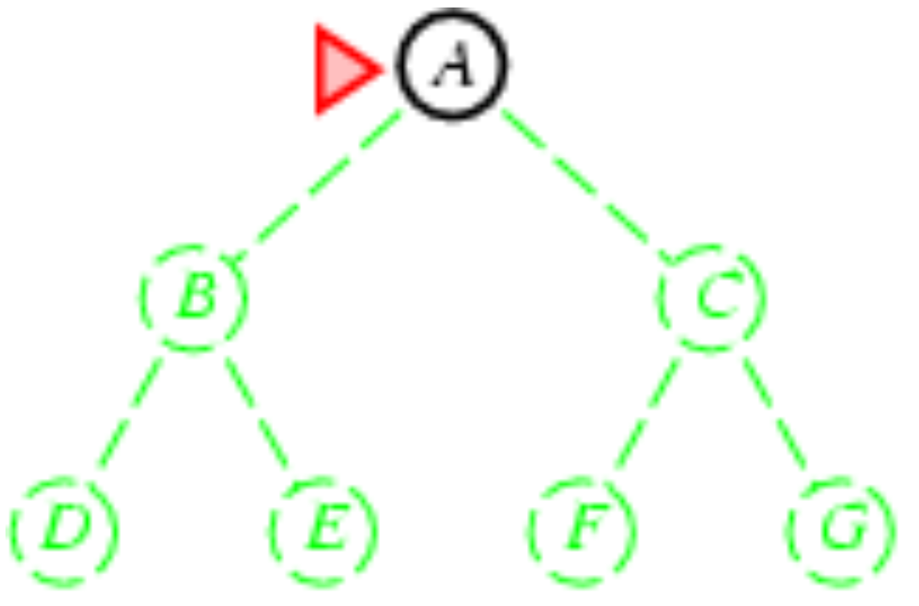
- Expand shallowest unexpanded node – Nodes are considered in increasing order of depth
- Implementation of the BFS algorithm
  - *fringe* is a FIFO queue – new successors go at end
- The symbols are used in the BFS algorithm
  - *fringe*: The queue structure holds the nodes (i.e., states) that **will be considered**
  - *closed*: The queue structure holds the nodes (i.e., states) that **have been considered**
  - *G=(N,A)*: The tree representation of the problem's state space
  - n_0: The initial state (i.e., the root node of the search tree)
  - *GOAL*: The set of the goal states
  - $\Gamma(n)$: The set of successive nodes (i.e., states) of the current one n

BFS: Algorithm

```
BFS (N, A, n0, GOAL)
{
    fringe ← n0;
    closed ← ∅;
    while  (fringe ≠ ∅)  do
    {     n ← GET_FIRST(fringe);          // get the first element of  fringe
           closed ← closed ⊕ n;
          if  (n ∈ GOAL)  then  return  SOLUTION(n);
          if  (Γ(n) ≠ ∅) then  fringe ← fringe ⊕ Γ(n);
    }
    return  ("No solution");
}
```

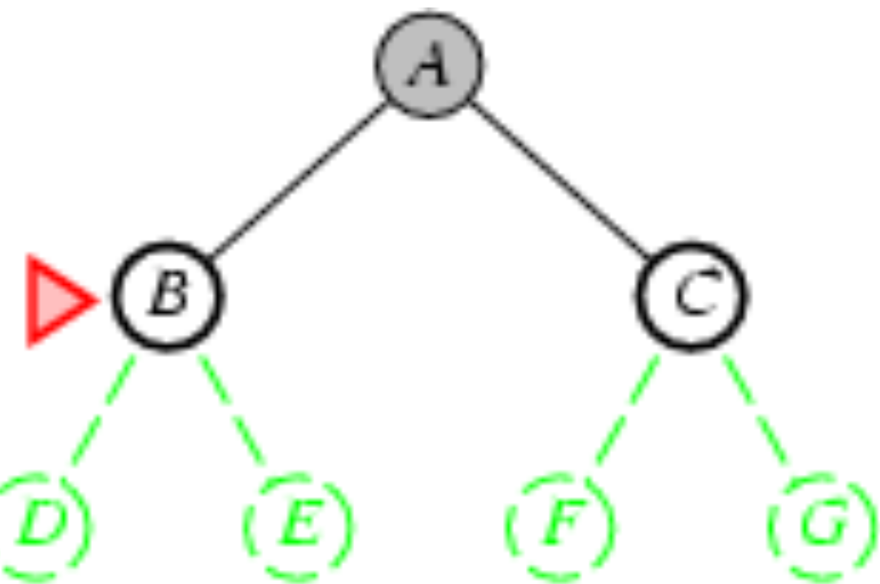
BFS: Example (1)

- Expand shallowest unexpanded node – Nodes are considered in increasing order of depth



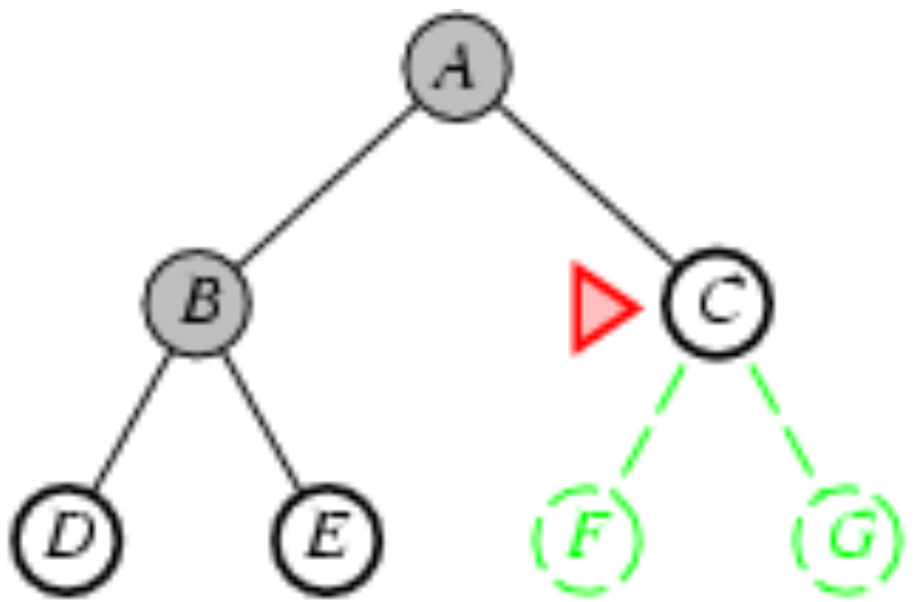
BFS: Example (2)

- Expand shallowest unexpanded node – Nodes are considered in increasing order of depth



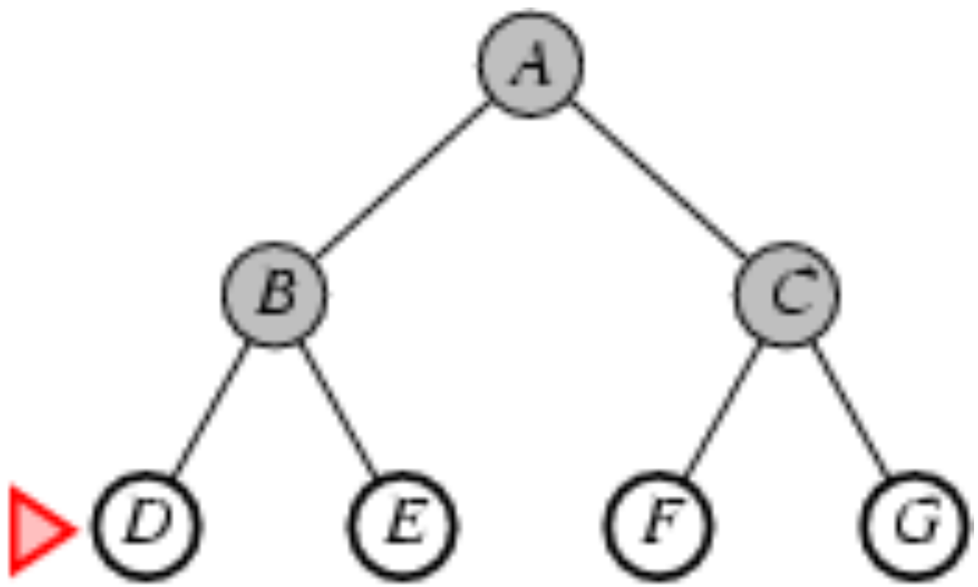
BFS: Example (3)

- Expand shallowest unexpanded node – Nodes are considered in increasing order of depth



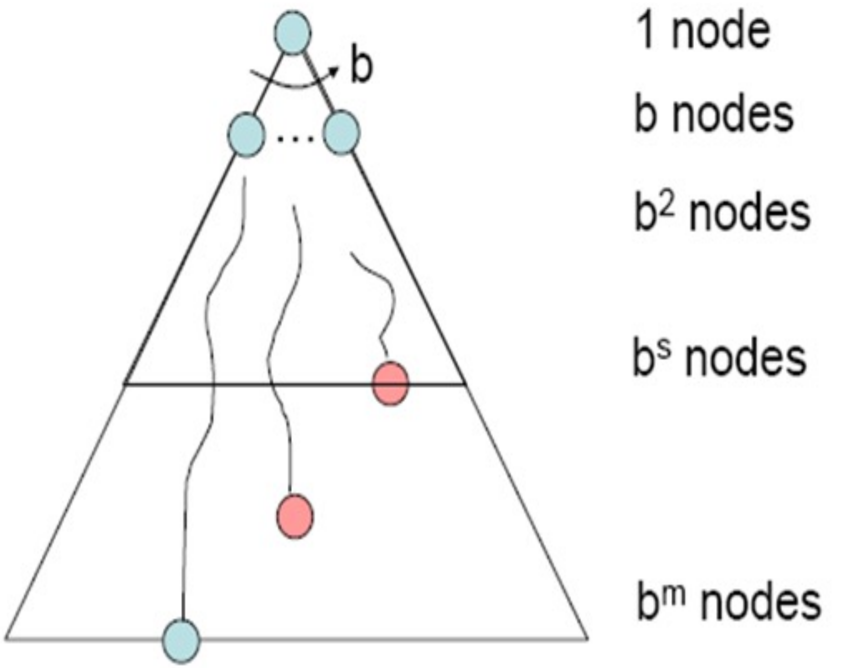
BFS: Example (4)

- Expand shallowest unexpanded node – Nodes are considered in increasing order of depth



Properties of BFS

- Complete?
  - Yes (if b is finite)
- Time?
  - $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- Space?
  - $O(b^{d+1})$ – Keeps every node in memory
- Optimal?
  - Yes (if cost =1 per step)



Uniform-cost search (UCS)

- Expand least-cost unexpanded node – Nodes are considered in order of increasing cost (from the root node to the current one)
- Implementation:
  - *fringe* is a queue ordered by path cost
- Equivalent to breadth-first search (BFS) if the costs of all the steps (i.e., the edges of the search tree) are equal

UCS: Algorithm

```
UCS (N, A, n0, GOAL, c)
{
    fringe ← n0;
    closed ← ∅;
    while  (fringe ≠ ∅)  do
    {      n ← GET_LOWEST_COST(fringe);        // get the element of
                                               // lowest path cost c(n)
           closed ← closed ⊕ n;
           if  (n ∈ GOAL)  then  return  SOLUTION(n);
           if  (Γ(n) ≠ ∅) then  fringe ← fringe ⊕ Γ(n);
    }
    return  ("No solution");
}
```

Properties of UCS

- Complete?
  - Yes (if step cost at least ε, for some constant $\varepsilon > 0$)
- Time?
  - Depends on the number of nodes that have the path cost ≤ the path cost of the optimal solution: $O(b^{\lceil C^*/\varepsilon \rceil})$, where C^* is the path cost of the optimal solution
- Space?
  - Depends on the number of nodes that have the path cost ≤ the path cost of the optimal solution: $O(b^{\lceil C^*/\varepsilon \rceil})$
- Optimal?
  - Yes (if nodes are expanded in increasing order of $g(n)$)

Depth-first search (DFS)

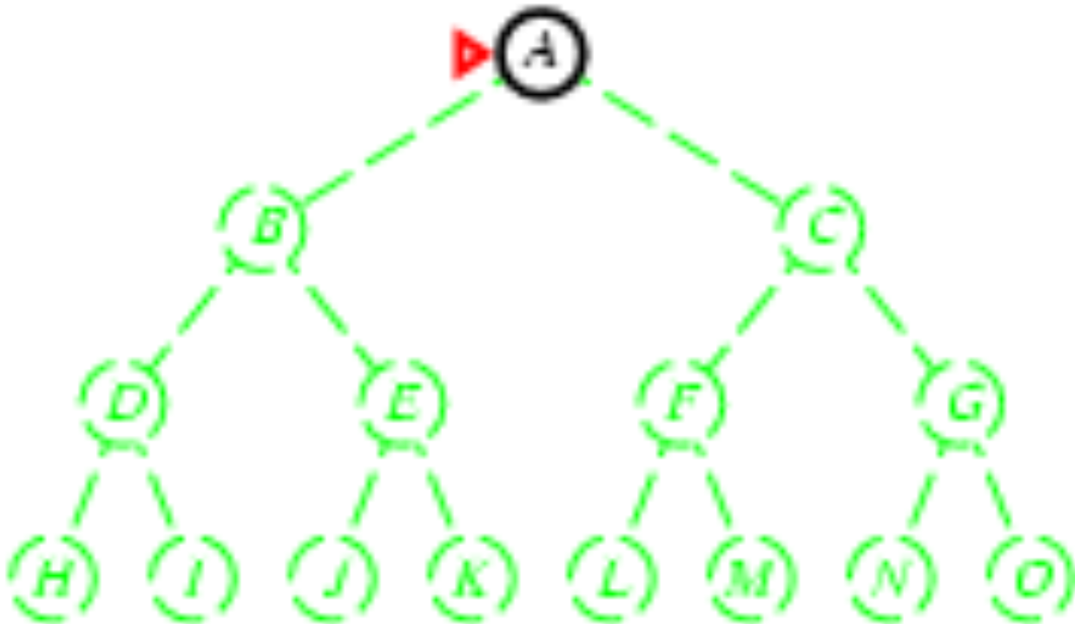
- Expand deepest unexpanded node
- Implementation:
  - *fringe* is a stack (i.e., LIFO) structure – New nodes are added to the top of *fringe*

DFS: Algorithm

```
DFS (N, A, n0, GOAL)
{
    fringe ← n0;
    closed ← ∅;
    while  (fringe ≠ ∅)  do
    {      n ← GET_FIRST(fringe);          // get the first element of fringe
           closed ← closed ⊕ n;
           if  (n ∈ GOAL)  then  return  SOLUTION(n);
           if  (Γ(n) ≠ ∅) then  fringe ← Γ(n) ⊕ fringe;
    }
    return  ("No solution");
}
```

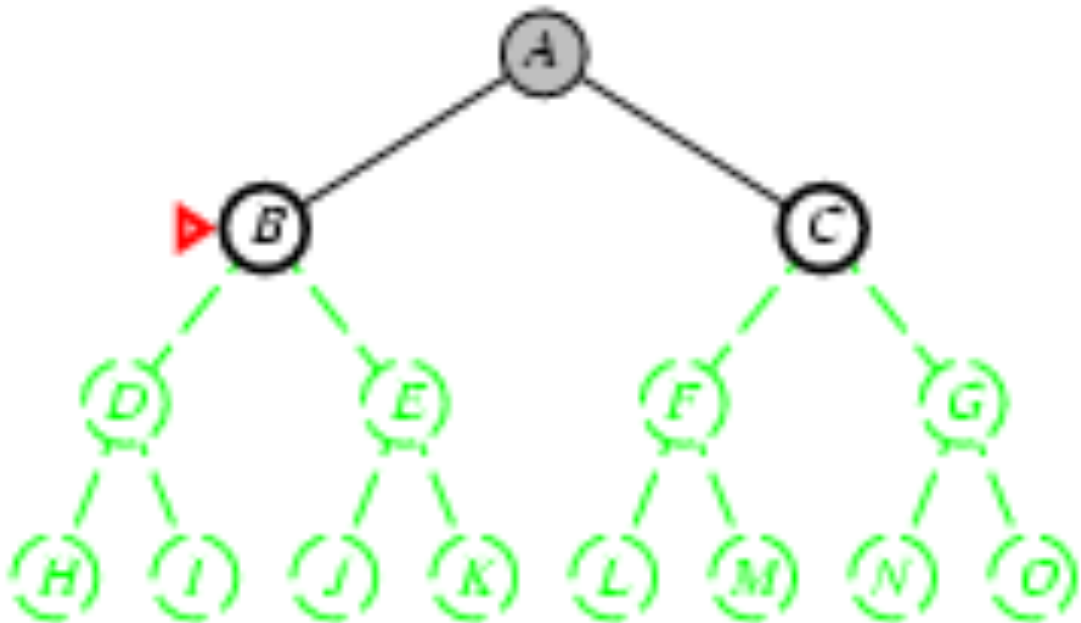
DFS: Example (1)

- Expand deepest unexpanded node



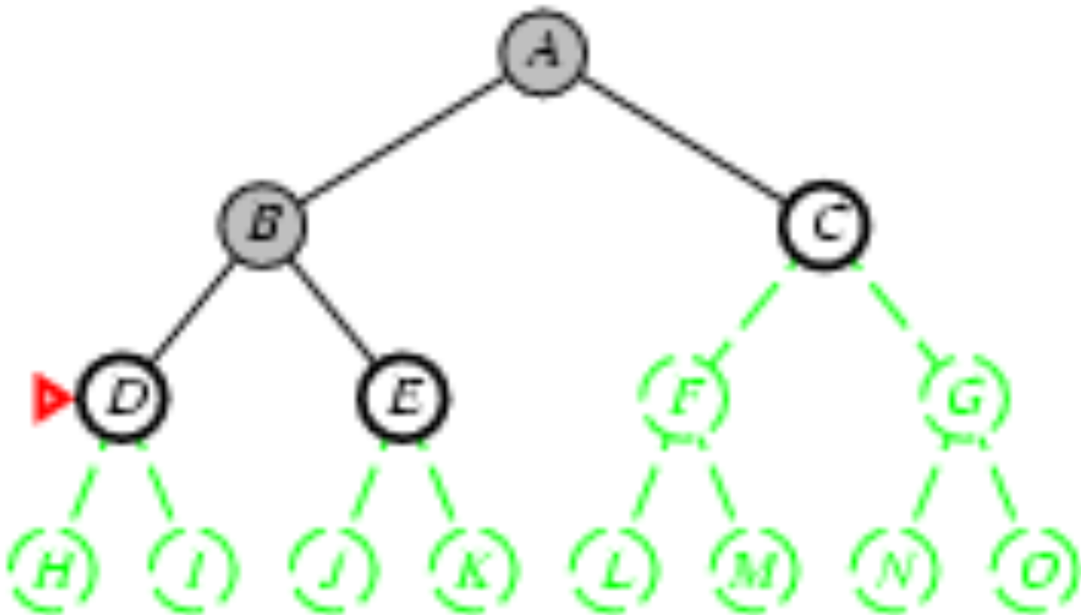
DFS: Example (2)

- Expand deepest unexpanded node



DFS: Example (3)

- Expand deepest unexpanded node

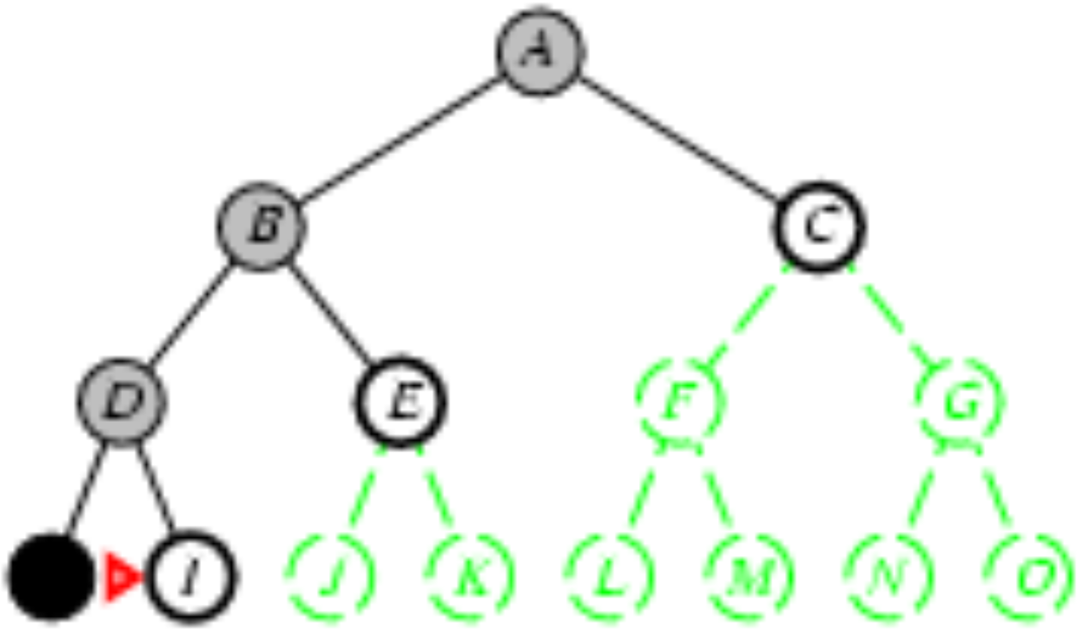


DFS: Example (4)

- Expand deepest unexpanded node

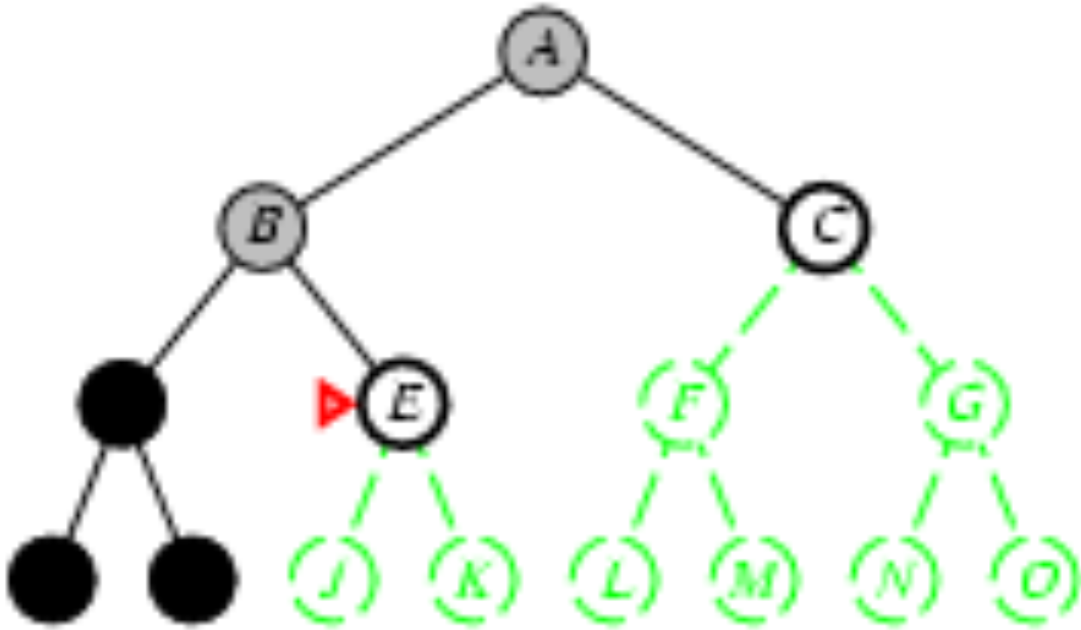
DFS: Example (5)

- Expand deepest unexpanded node



DFS: Example (6)

- Expand deepest unexpanded node



Properties of DFS

- Complete?
  - No – Fails in infinite-depth spaces, spaces with loops
  - Proposal: Modify to avoid repeated states along path

    → Complete in finite spaces
- Time?
  - $O(b^m)$: Very large, if m is much larger than d
- Space?
  - $O(bm)$ – Linear space
- Optimal?
  - No

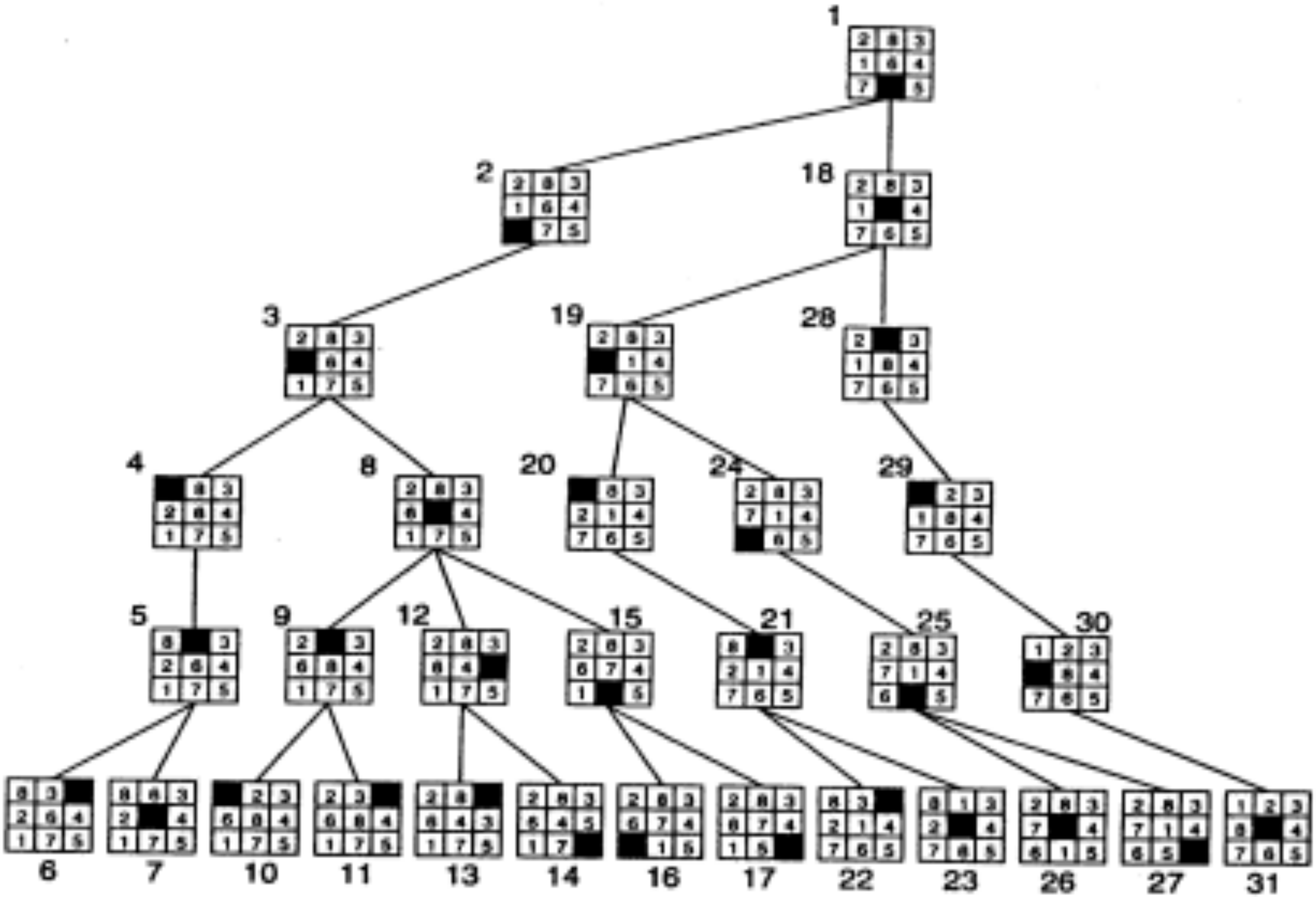
Depth-limited search (DLS)

Is depth-first search (DFS) with depth limit *l*

$\rightarrow$ nodes at depth *l* have no successors

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```



(The 8-puzzle - The DLS algorithm using the depth limit *l*=5)

Goal

Iterative deepening search (IDS)

- Problem of the depth-limited search (DLS) algorithm:
  - If all the solutions (i.e., the target nodes) are at a depth greater than the depth limit *l*, then the DLS algorithm fails (i.e., can't find a solution)

- IDS algorithm:
  - Apply the DFS algorithm for the paths of length <=1
  - If it fails (can't find the solution), then continue to apply the DFS algorithm for the paths of length <=2
  - If it fails (can't find the solution), then continue to apply the DFS algorithm for the paths of length <=3
  - …(continue as above, until: 1)find a solution, or 2)the entire tree has been examined but no solution is found)

IDS: Algorithm

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

  inputs: *problem*, a problem

  for $depth \leftarrow 0$ **to** ∞ **do**

    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

    if *result* $\neq$ cutoff **then return** *result*

IDS: Example (1)

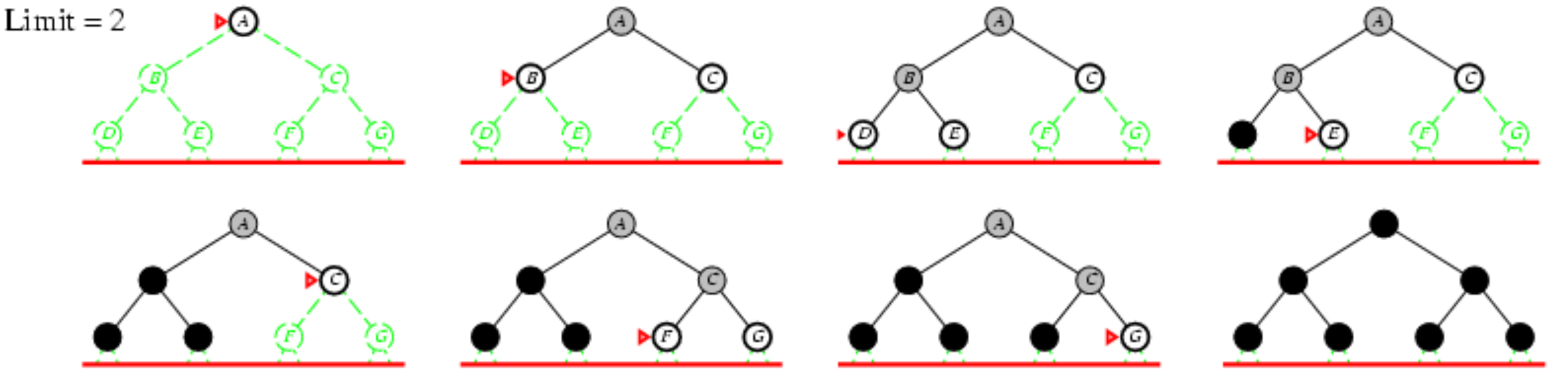
Depth limit *l*=0

IDS: Example (2)

Depth limit *l*=1

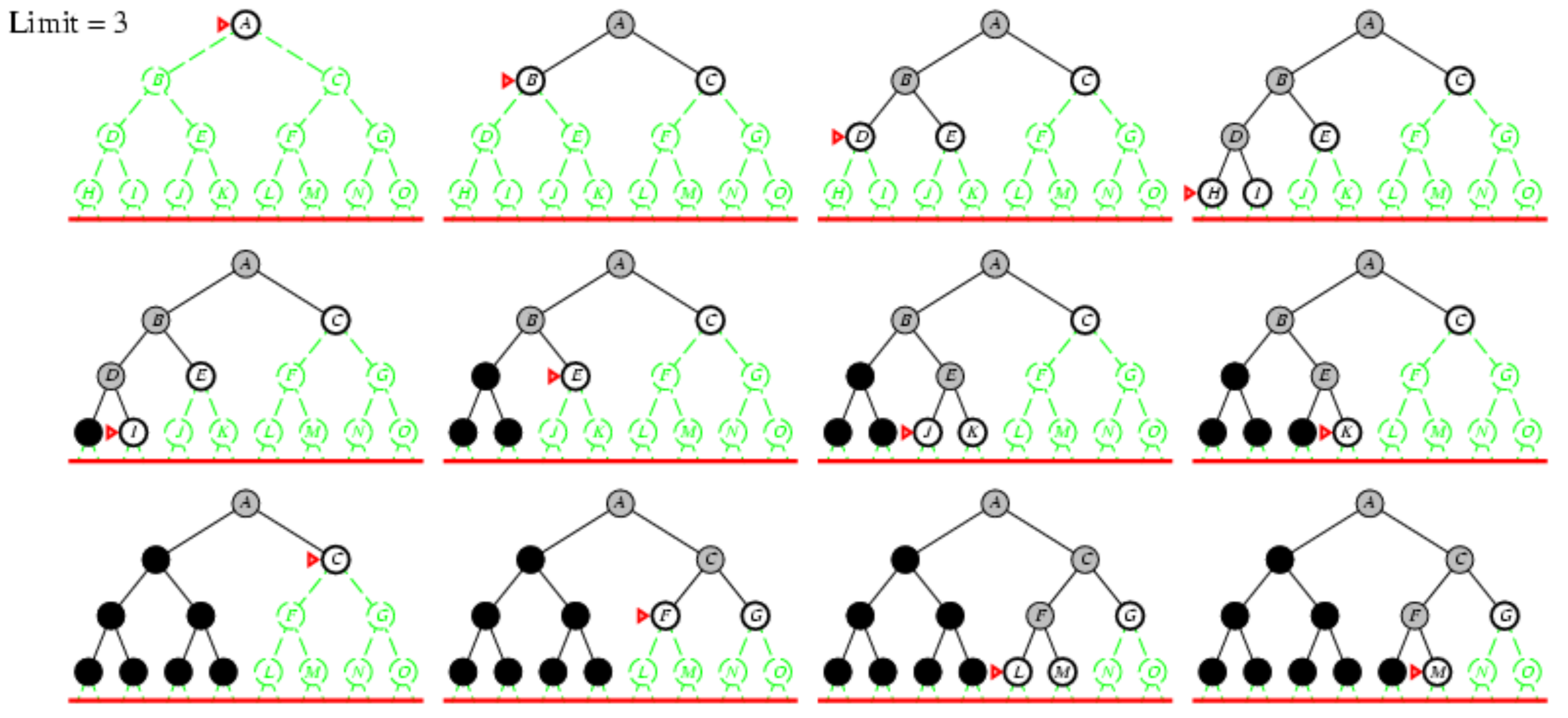
IDS: Example (3)

Depth limit *l*=2



IDS: Example (4)

Depth limit *l*=3



IDS: Another algorithm

```
IDS (N, A, n0, GOAL, l)                                   // l: depth limit
{
   fringe ← n0;          closed ← ∅;           depth ← l;
   while  (fringe ≠ ∅)  do
   {   n ← GET_FIRST(fringe);                             // get the first element of fringe
       closed ← closed ⊕ n;
       if  (n ∈ GOAL)  then  return  SOLUTION(n);
       if  (Γ(n) ≠ ∅) then
       {      case  d(n)  do                              // d(n): depth of node n
              [0..(depth-1)]:  fringe ← Γ(n) ⊕ fringe;
              depth:           fringe ← fringe ⊕ Γ(n);
              (depth+1):       {    depth ← depth + l;
                                    if  (l=1)  then  fringe ← fringe ⊕ Γ(n)
                                               else  fringe ← Γ(n) ⊕ fringe;
                               }
        }
   }
   return  ("No solution");
}
```

DLS vs. IDS

- Given depth *d* and branching factor *b*, the number of nodes generated in the DLS algorithm is:

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

- Given depth *d* and branching factor *b*, the number of nodes generated in the IDS algorithm is:

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

- Example: Given *b=10* and *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 IDS

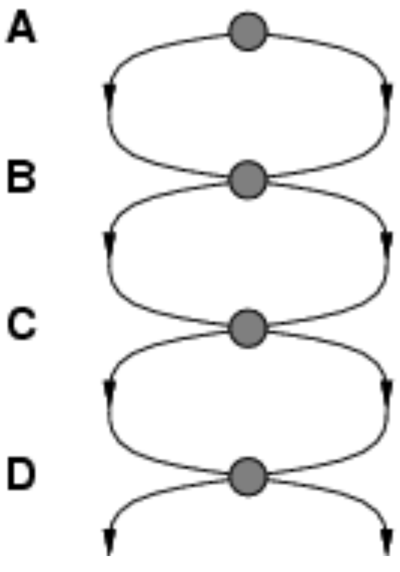
- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$
- Optimal?
  - Yes, if step cost =1

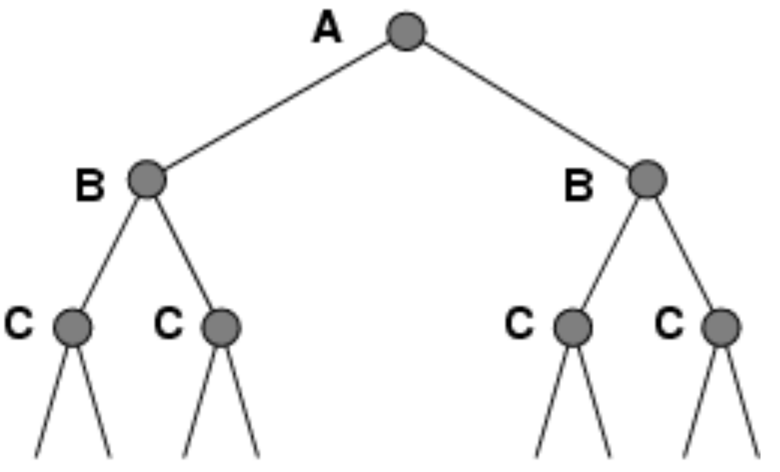
Summary of uninformed search strategies

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

Repeated states

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





- Solution: Never consider a node more than once!

Graph search: Algorithm

```
function Graph-Search(problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State(problem)), fringe);
closed ← an empty set
while (fringe not empty)
   node ← RemoveFirst(fringe);
   if (Goal-Test(problem, State(node))) then return Solution(node);
   if (State(node) is not in closed then
      add State(node) to closed
      fringe ← InsertAll(Expand(node, problem), fringe);
   end if
end
return failure;
```

- Never consider a node more than once!

Uninformed search: Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Uninformed search strategies:
  - Breath-first search (BFS)
  - Depth-first search (DFS)
  - Uniform-cost search (UCS)
  - Depth-limited (DLS)
  - Iterative deepening search (IDS)
- Iterative deepening search (IDS):
  - Memory space complexity is linear
  - Time complexity is higher just a little than the other uninformed search algorithms