Lecture 2: Uninformed search

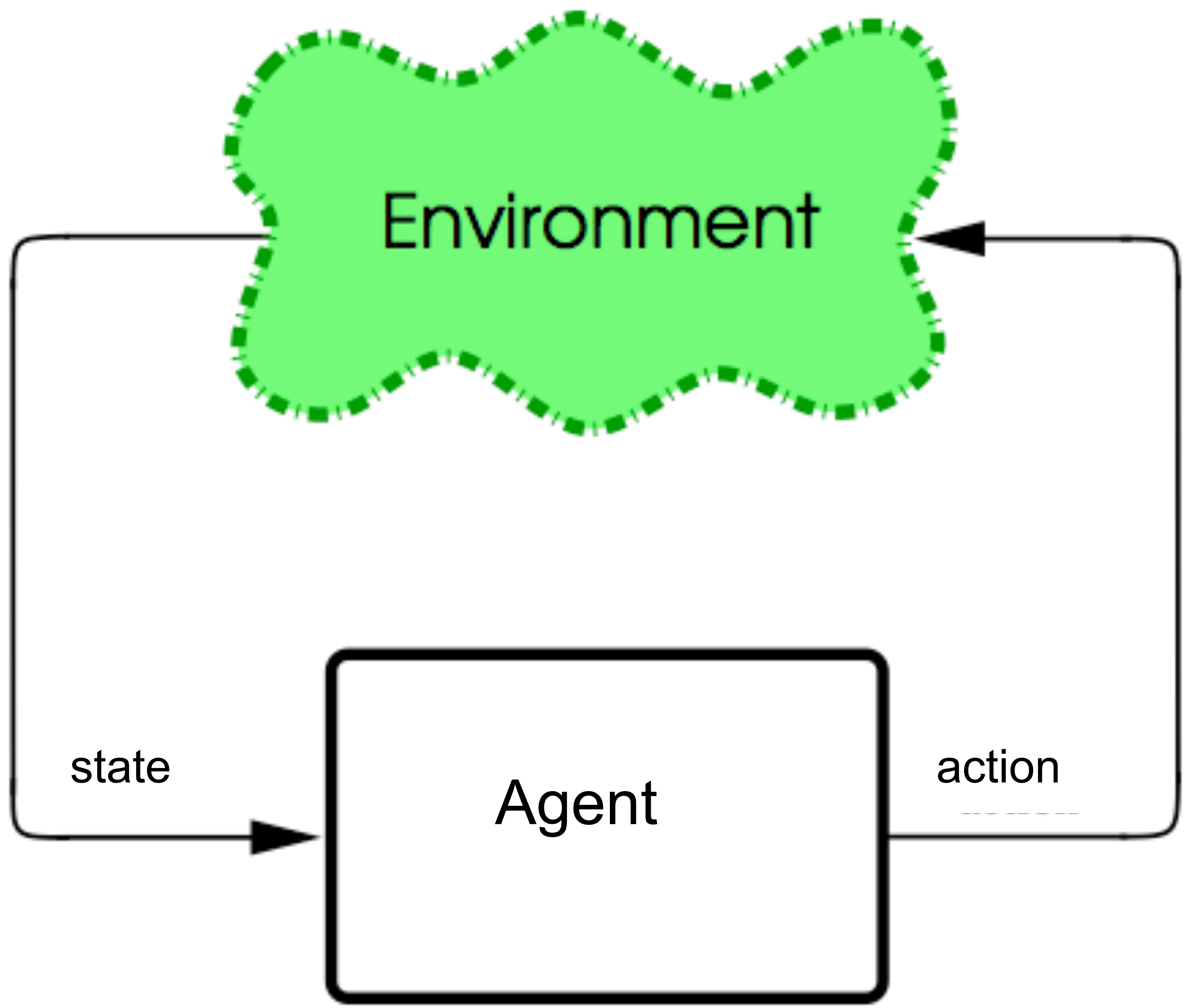
Artificial Intelligence
CS-GY-6613-I

Julian Togelius
julian.togelius@nyu.edu

On the menu

- Problem solving as search
- Uninformed search algorithms
- Assignments

An agent



Problem-solving as search

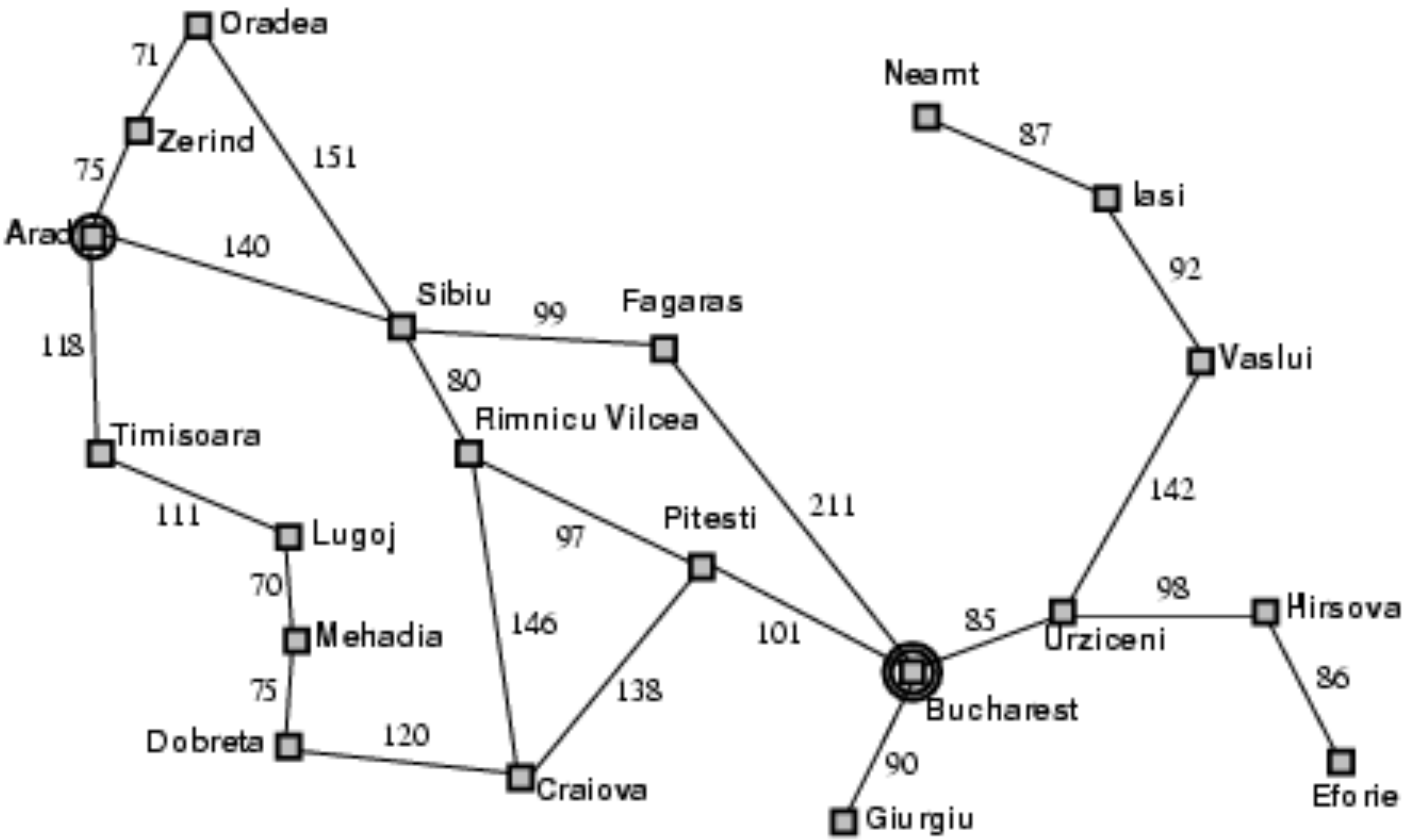
- An agent is in a *state*
- It wants to get to a *goal state*
- It needs to find the sequence of actions that gets it there
  - (cheapest, fastest, safest…)

Problem-solving agent

```
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
```

From Arad to Bucharest



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
  - 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
- Unknown state space: exploration problem

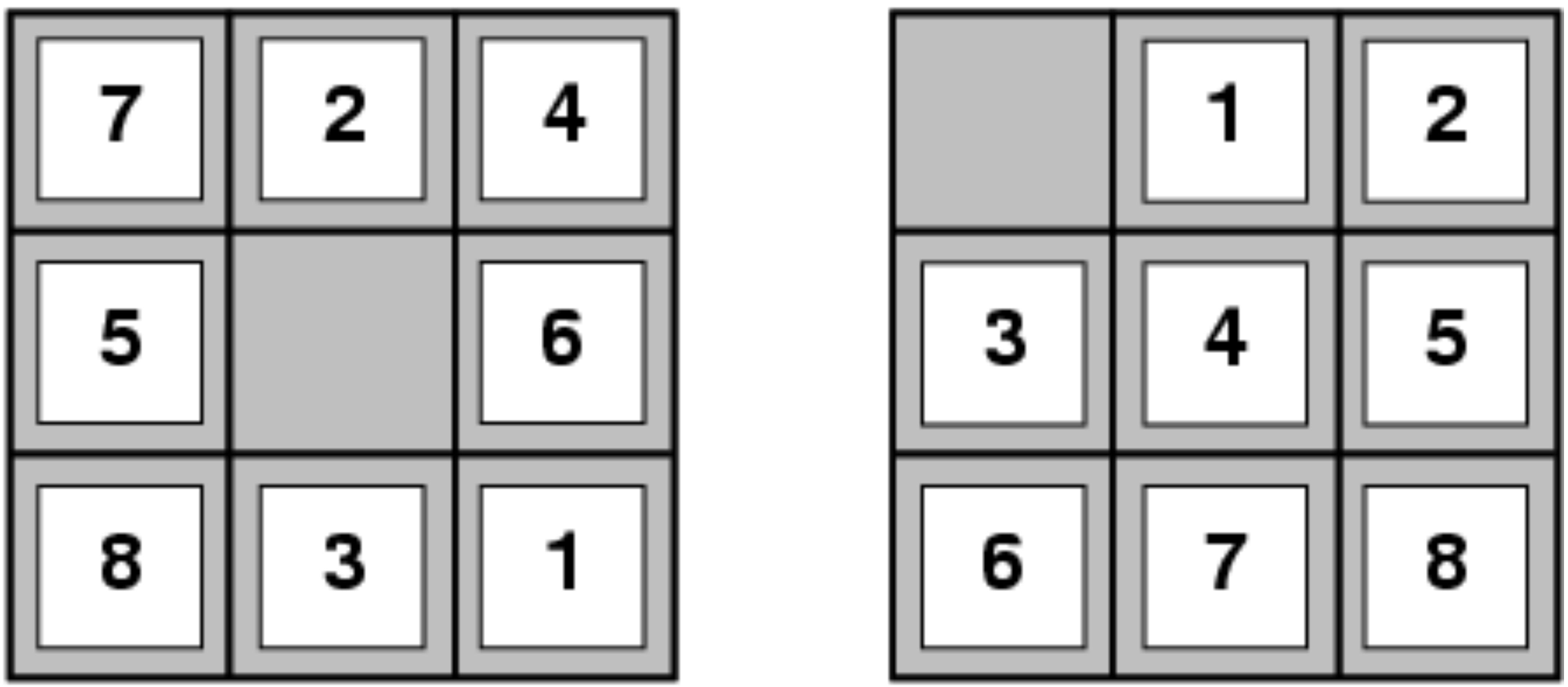
What problem type is…

- Driving to Bucharest?
- Flying to Bucharest?
- Playing Pac-Man?
- Playing StarCraft?
- Playing Poker?
- Living a good life?

Problem components

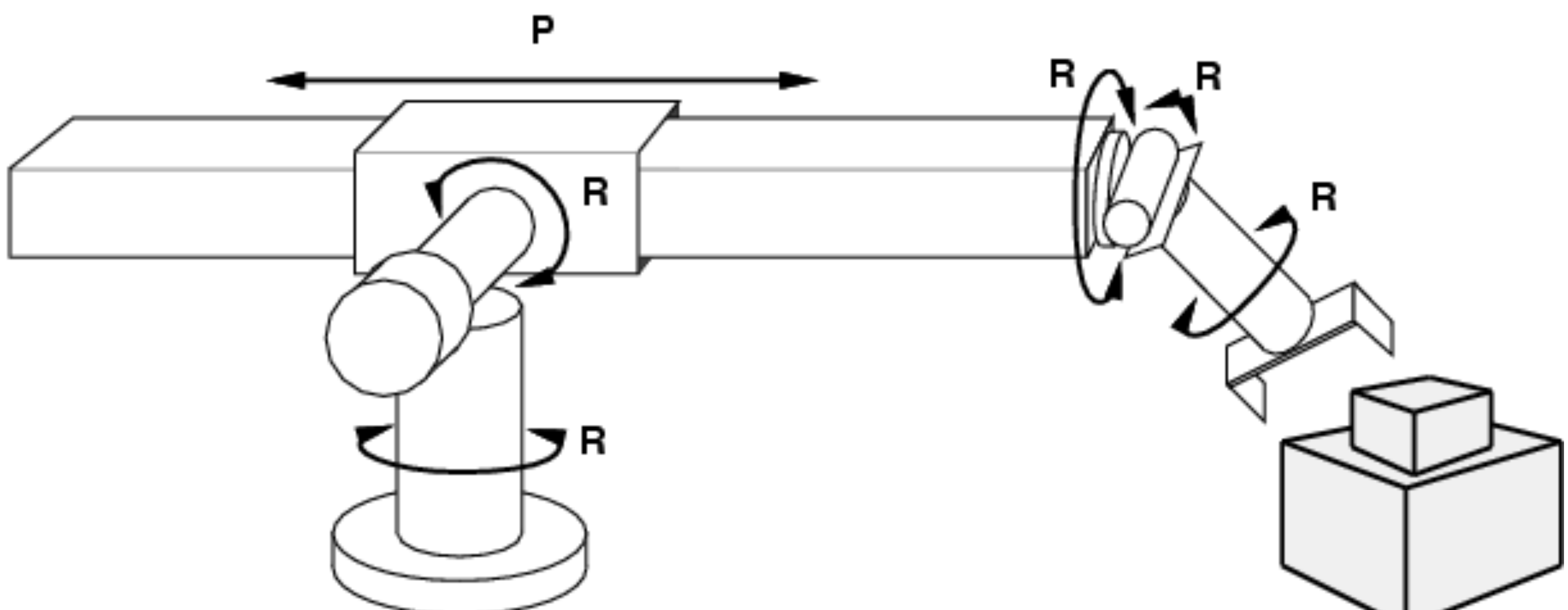
- Initial state
- Actions and successor function
  S(state, action) -> state’
- Goal test (are we there yet?)
- Path cost
- A solution is a sequence of actions leading from the initial state to a goal state

Problem components?



Start State Goal State

Problem components?



Problem components?

Problem components?

Problem components?

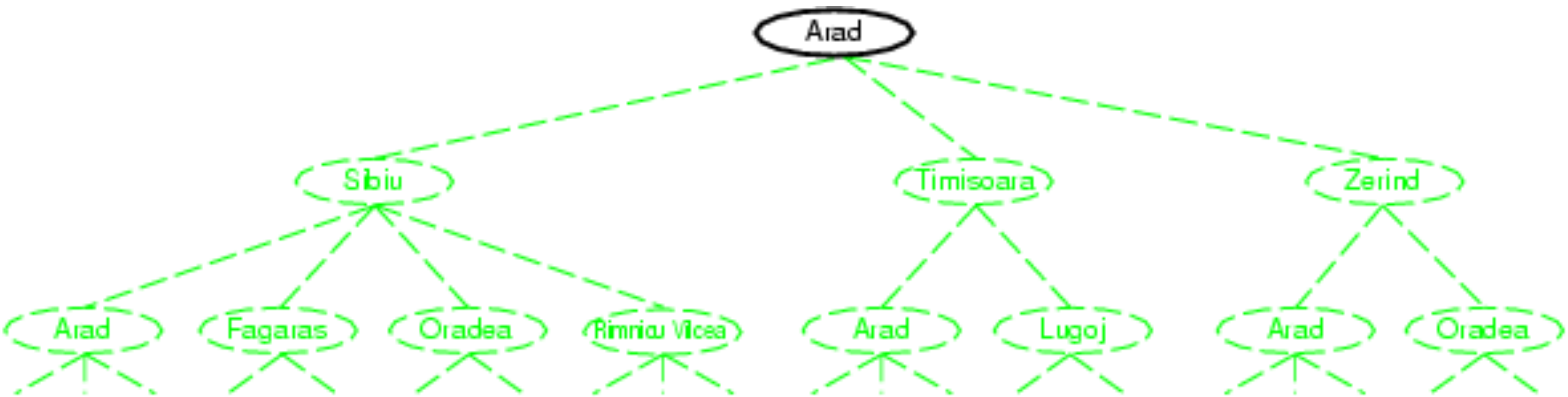
Problem components?

Tree search

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

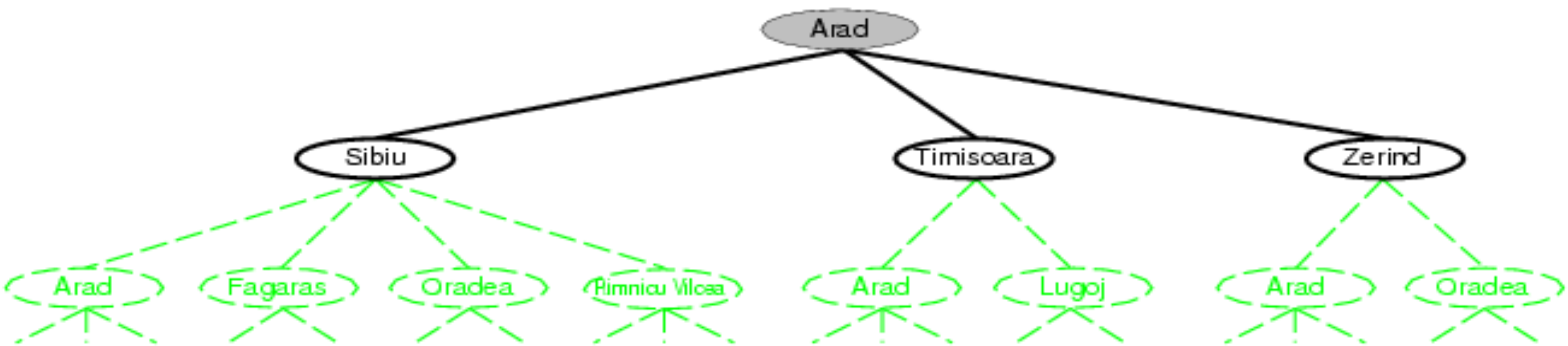
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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



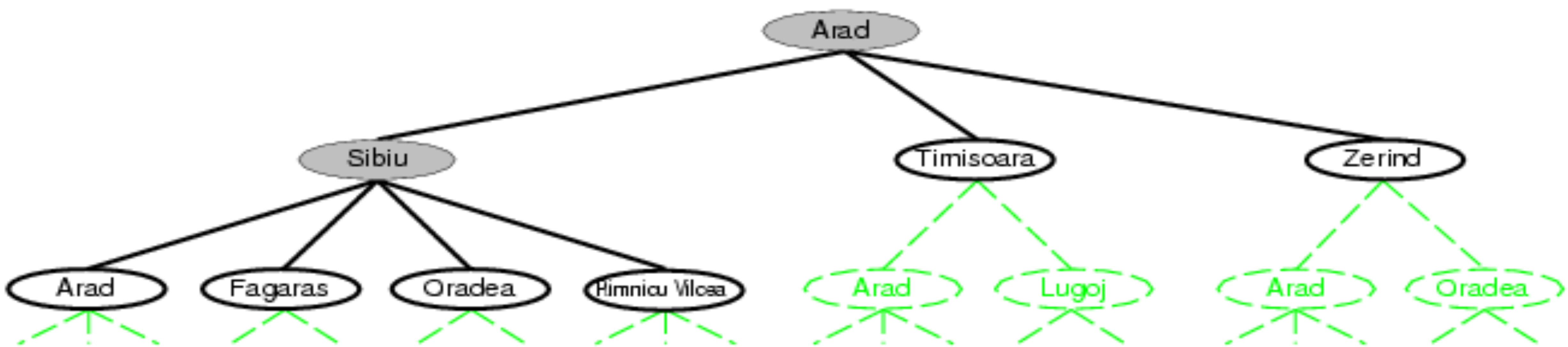
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Tree search



Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Tree search



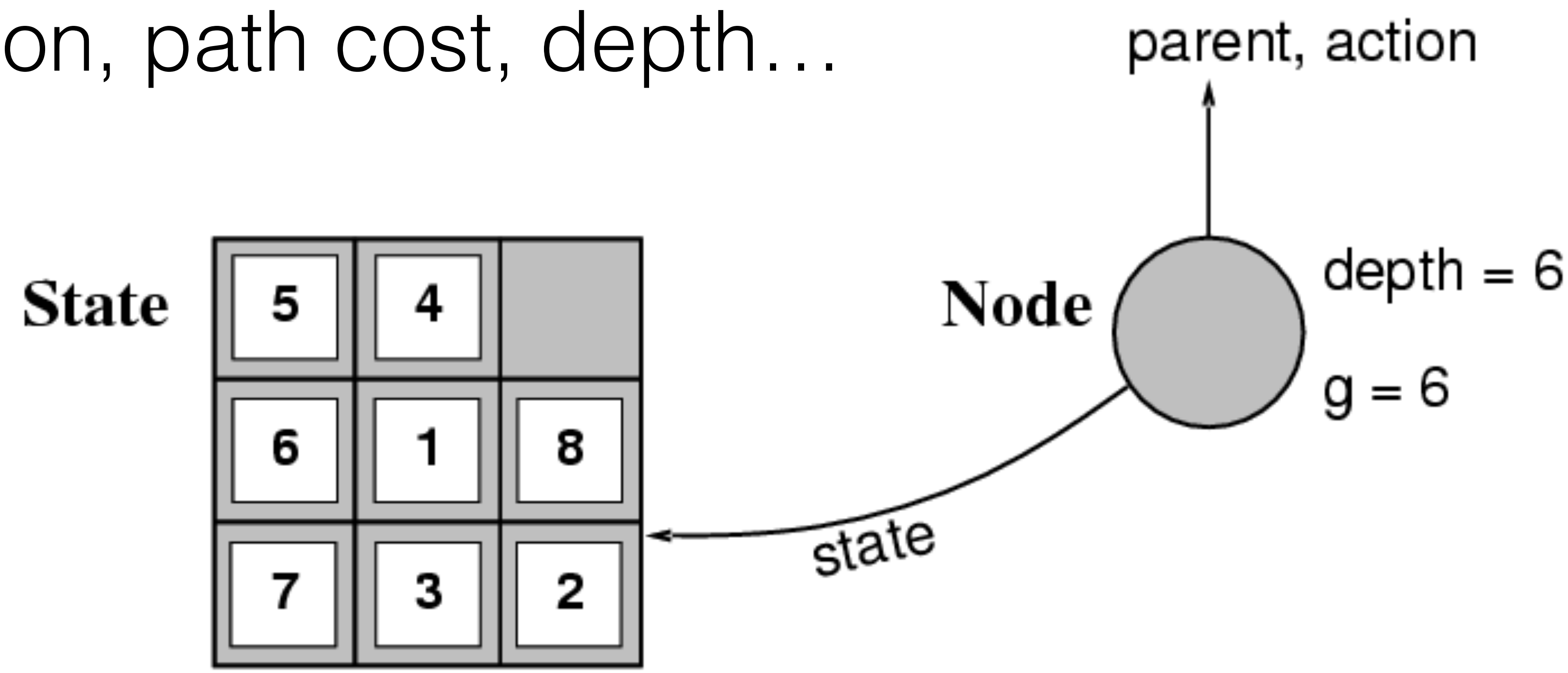
Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

```
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
```

Nodes versus states

- A state is a configuration of an environment/agent
- A node is a data structure constituting part of a search tree, might include state, parent node, action, path cost, depth…

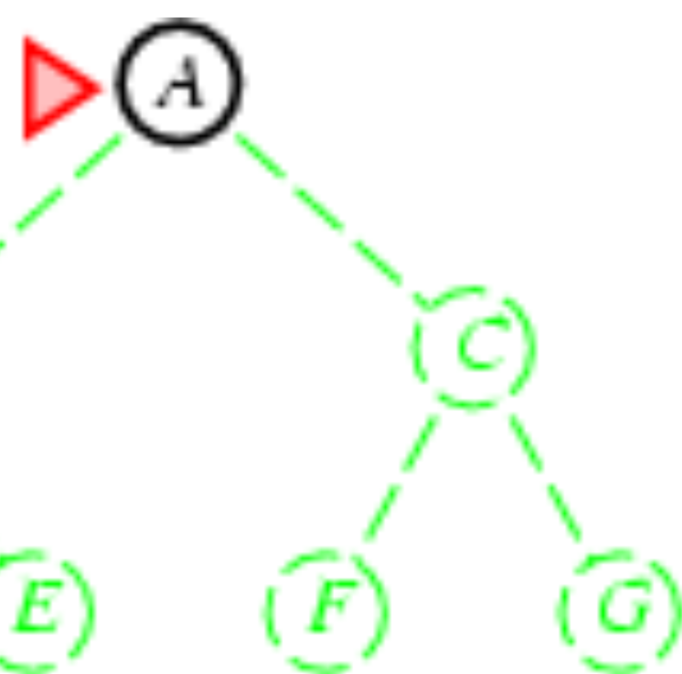


Uninformed search

- 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

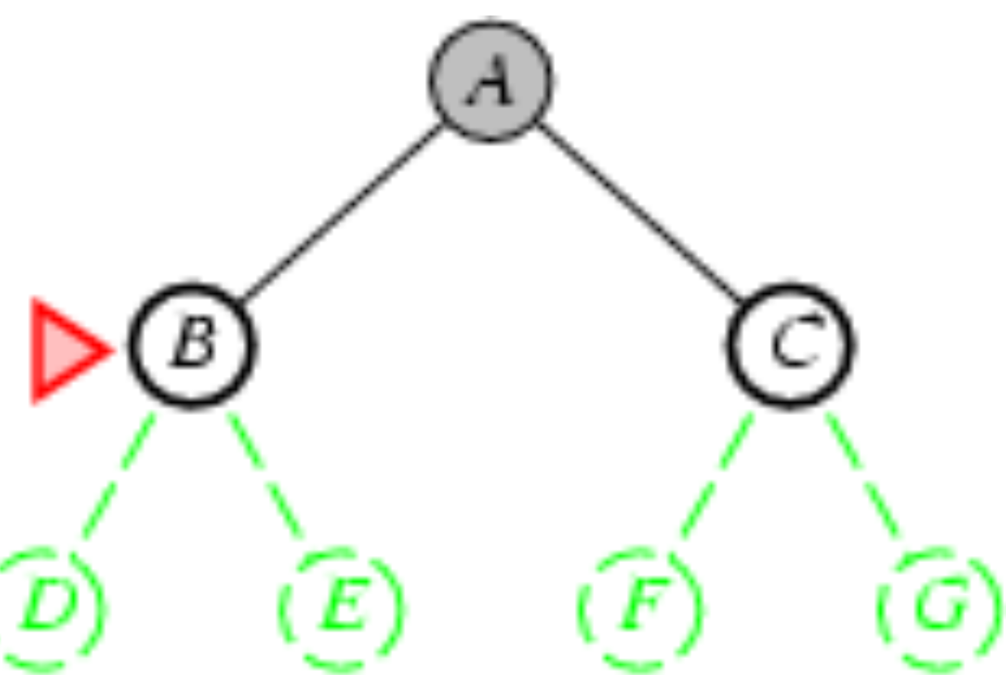
Breadth-first search

- Expand shallowest unexpanded node
- Implementation: new nodes added to a FIFO queue



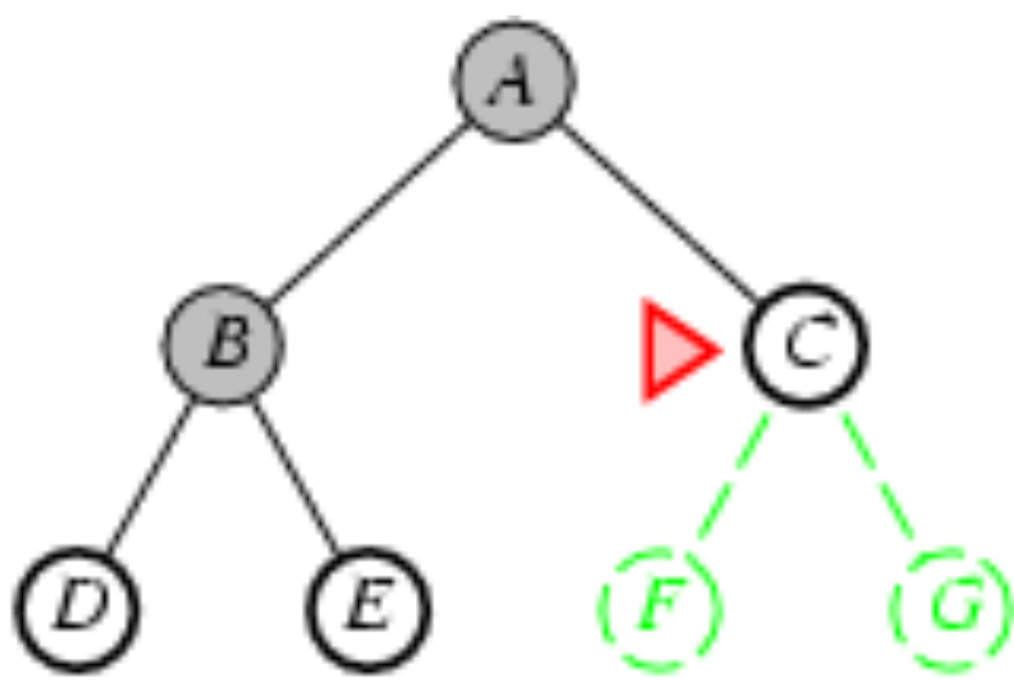
Breadth-first search

- Expand shallowest unexpanded node
- Implementation: new nodes added to a FIFO queue



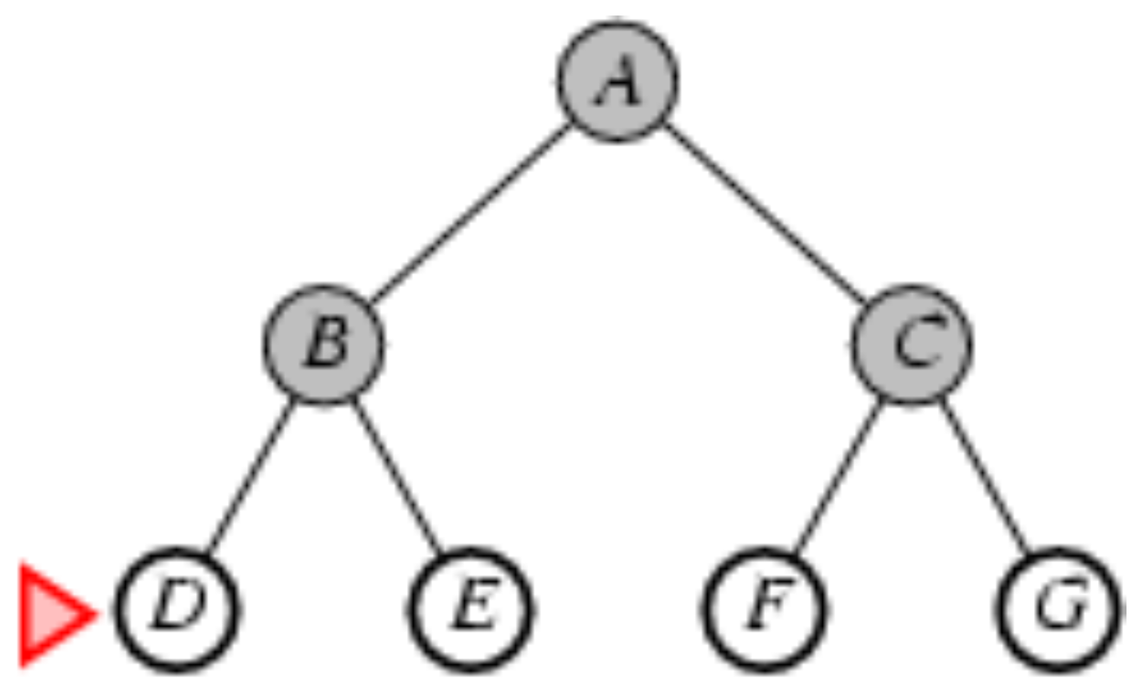
Breadth-first search

- Expand shallowest unexpanded node
- Implementation: new nodes added to a FIFO queue



Breadth-first search

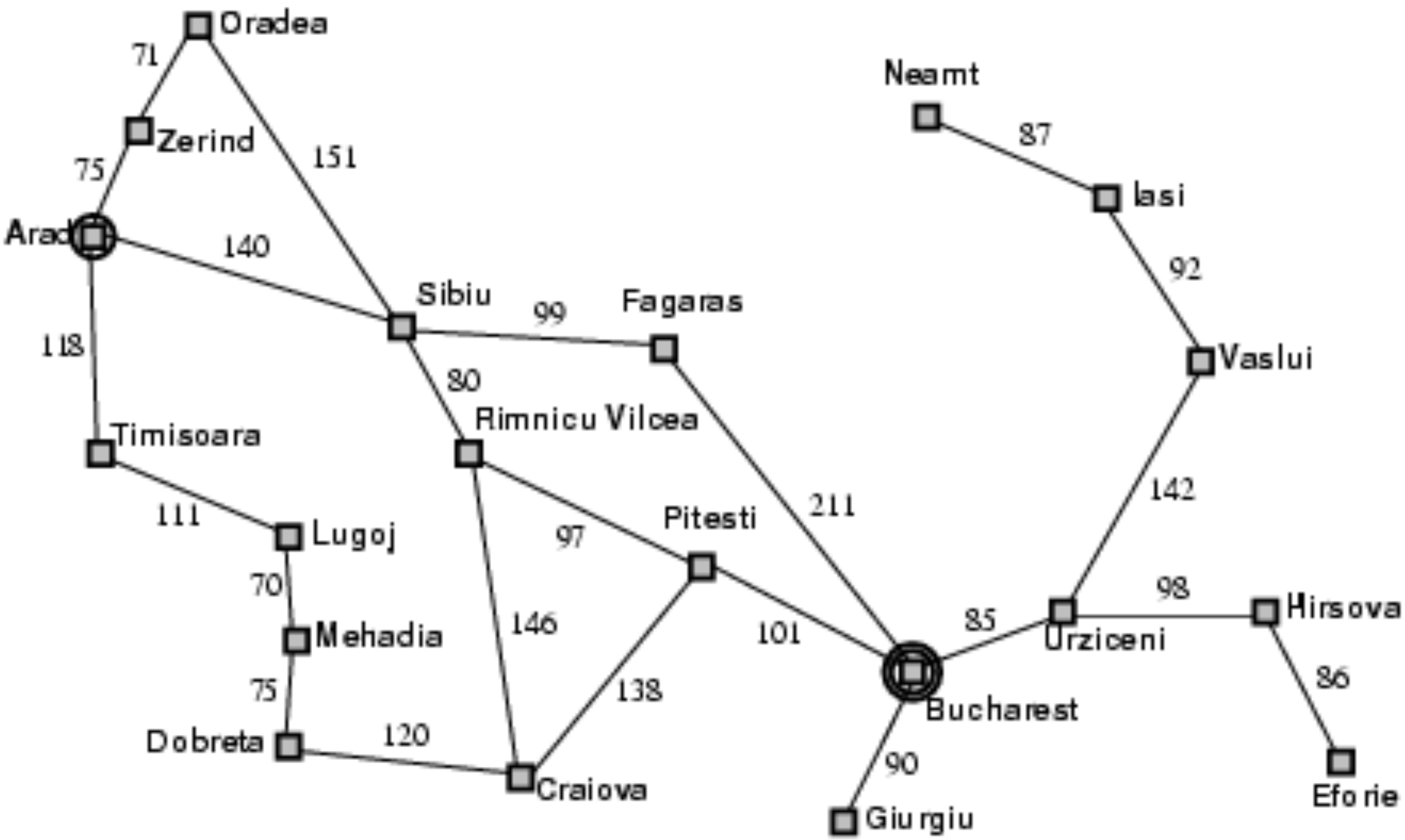
- Expand shallowest unexpanded node
- Implementation: new nodes added to a FIFO queue



Breadth-first search

- 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)

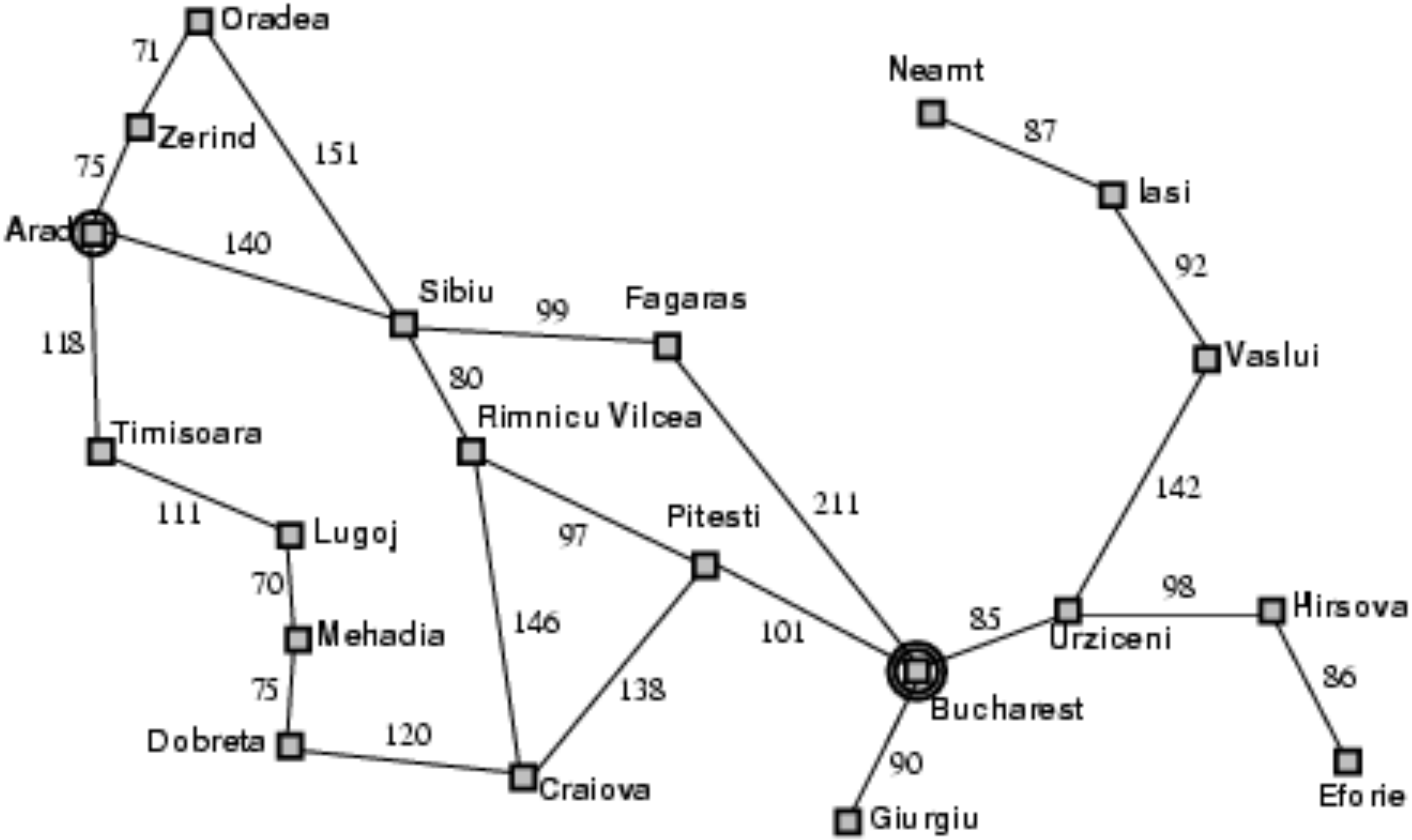
From Arad to Bucharest



Uniform-cost search

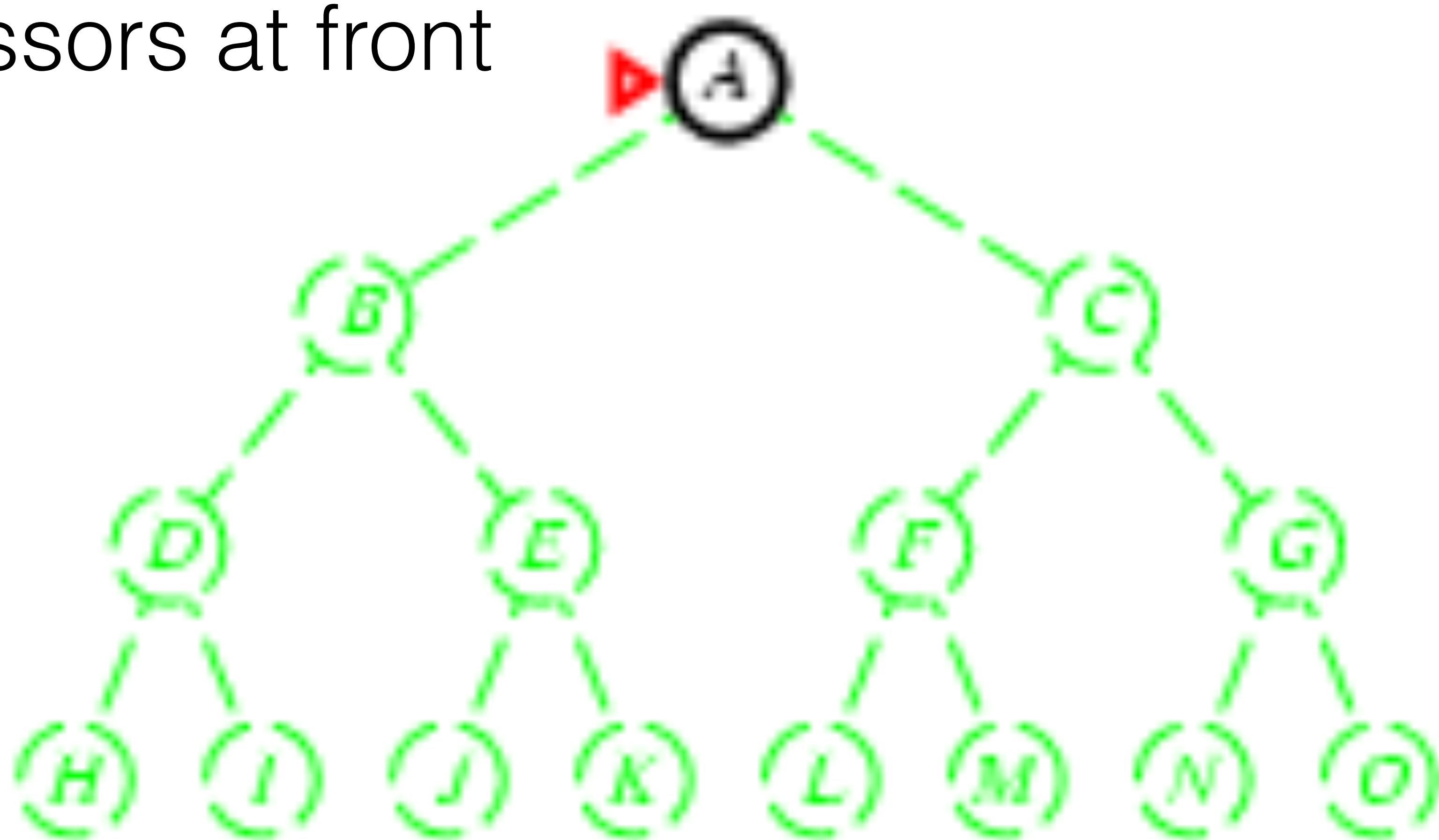
- Expand least-cost unexpanded node
- Equivalent to breadth-first if step costs all equal
- Complete and optimal

From Arad to Bucharest



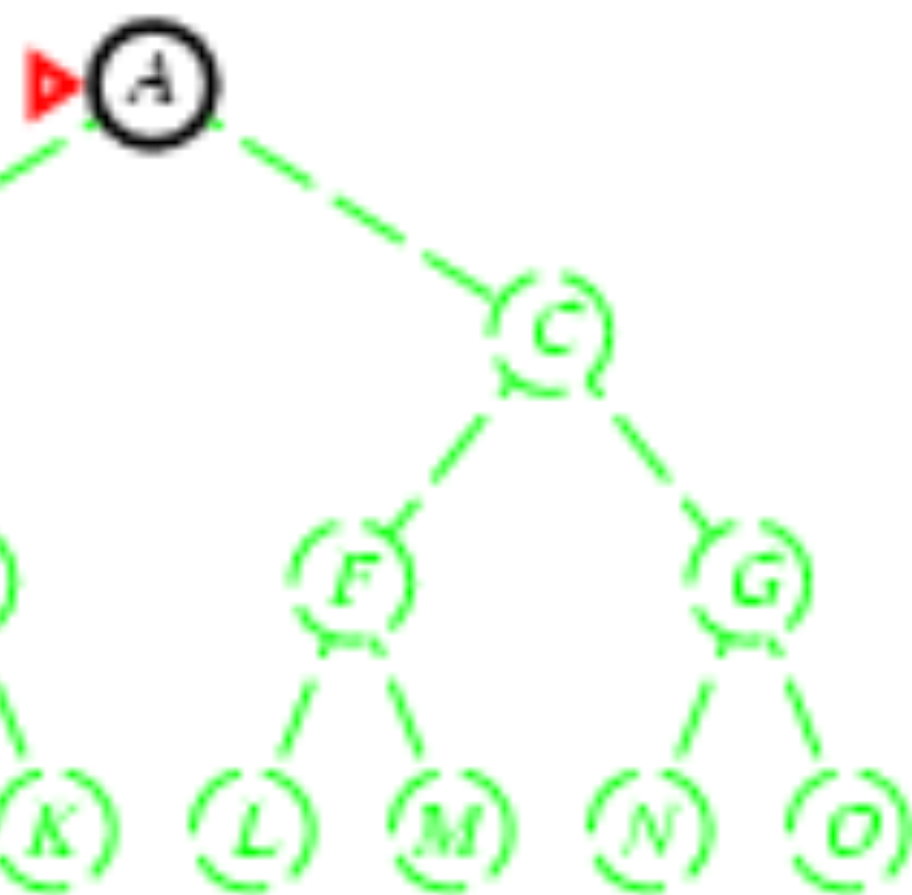
Depth-first search

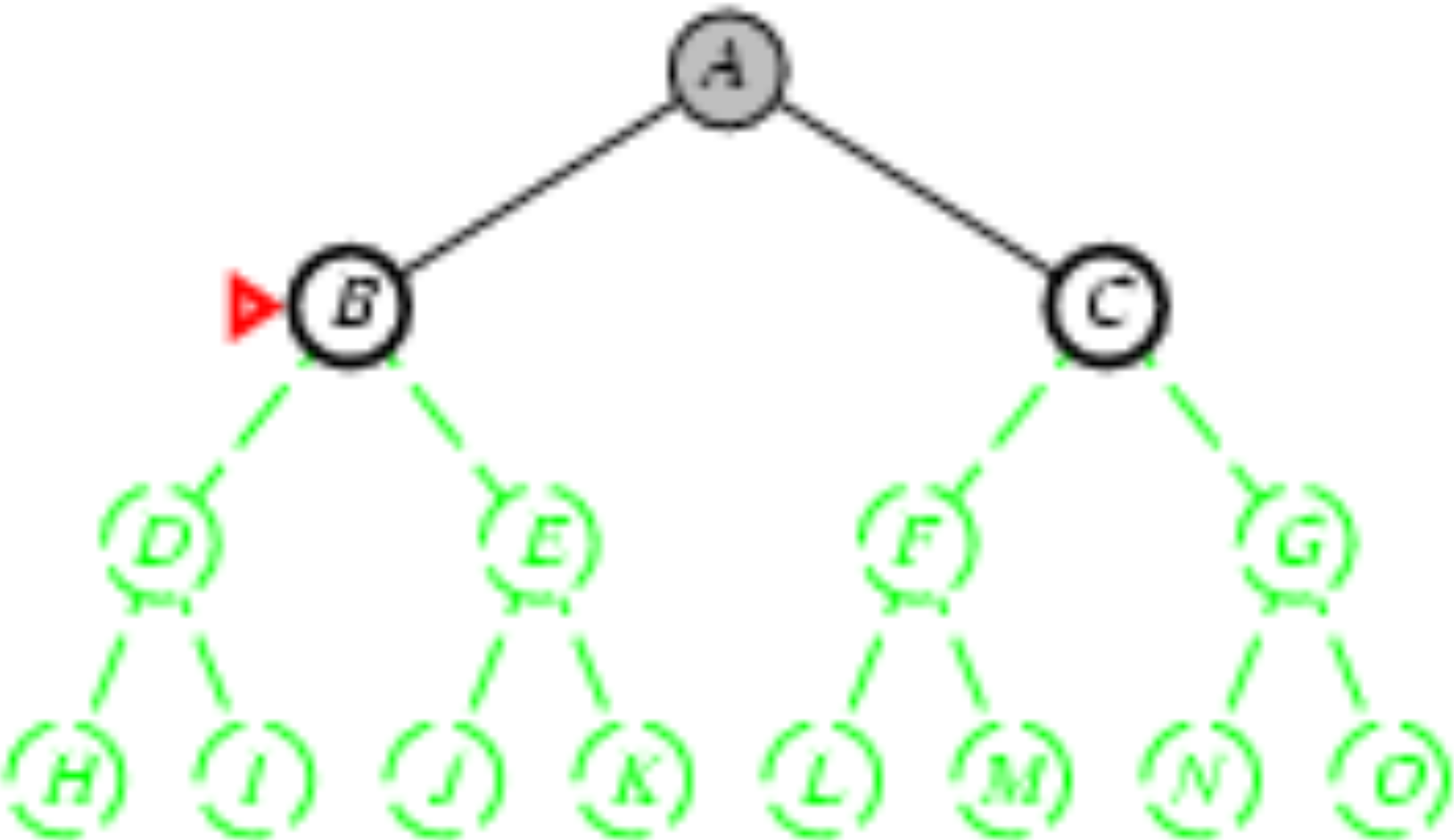
- Expand deepest unexpanded node
- Implementation: fringe = LIFO queue, i.e., put successors at front

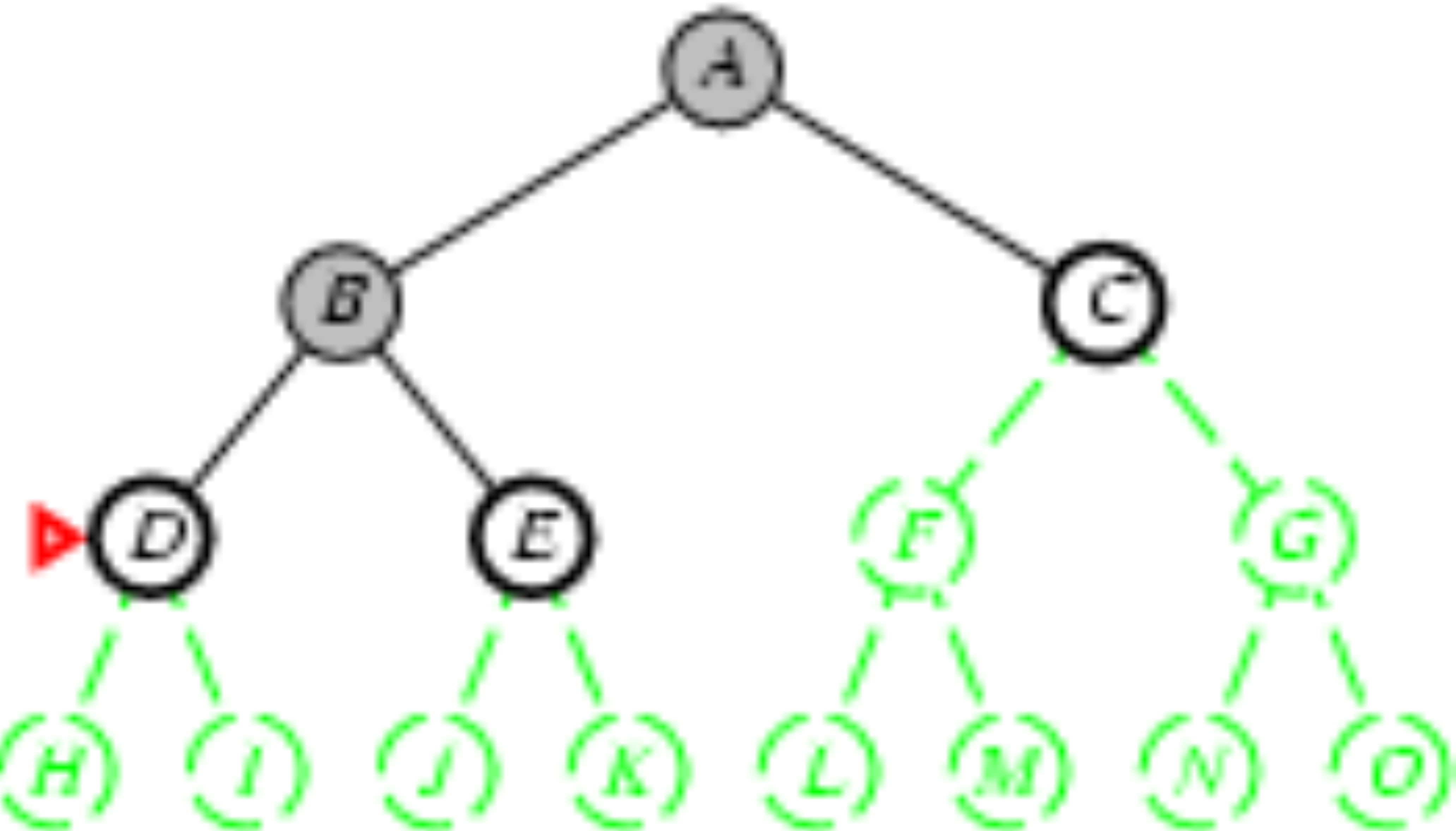


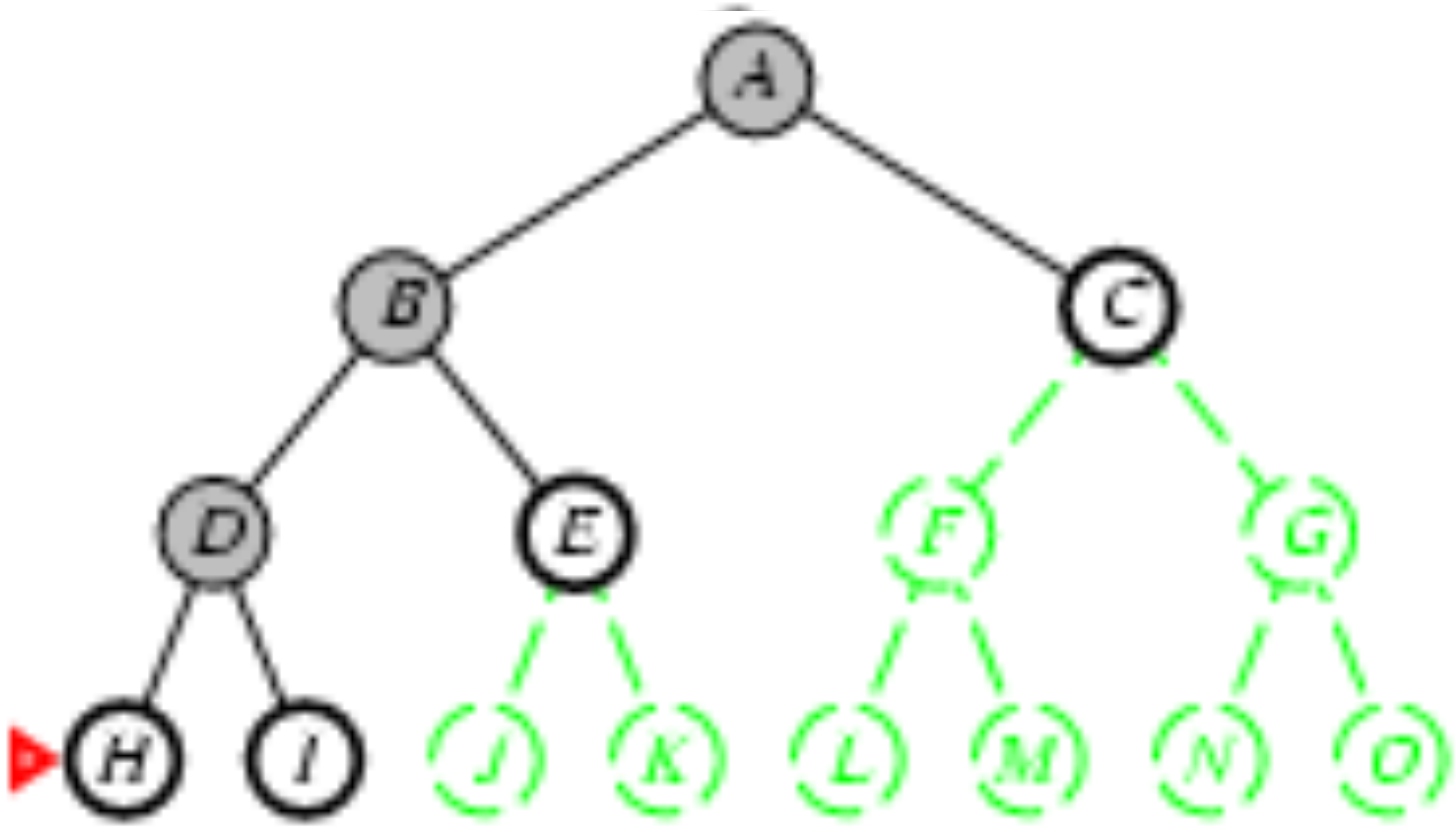
Depth-first search

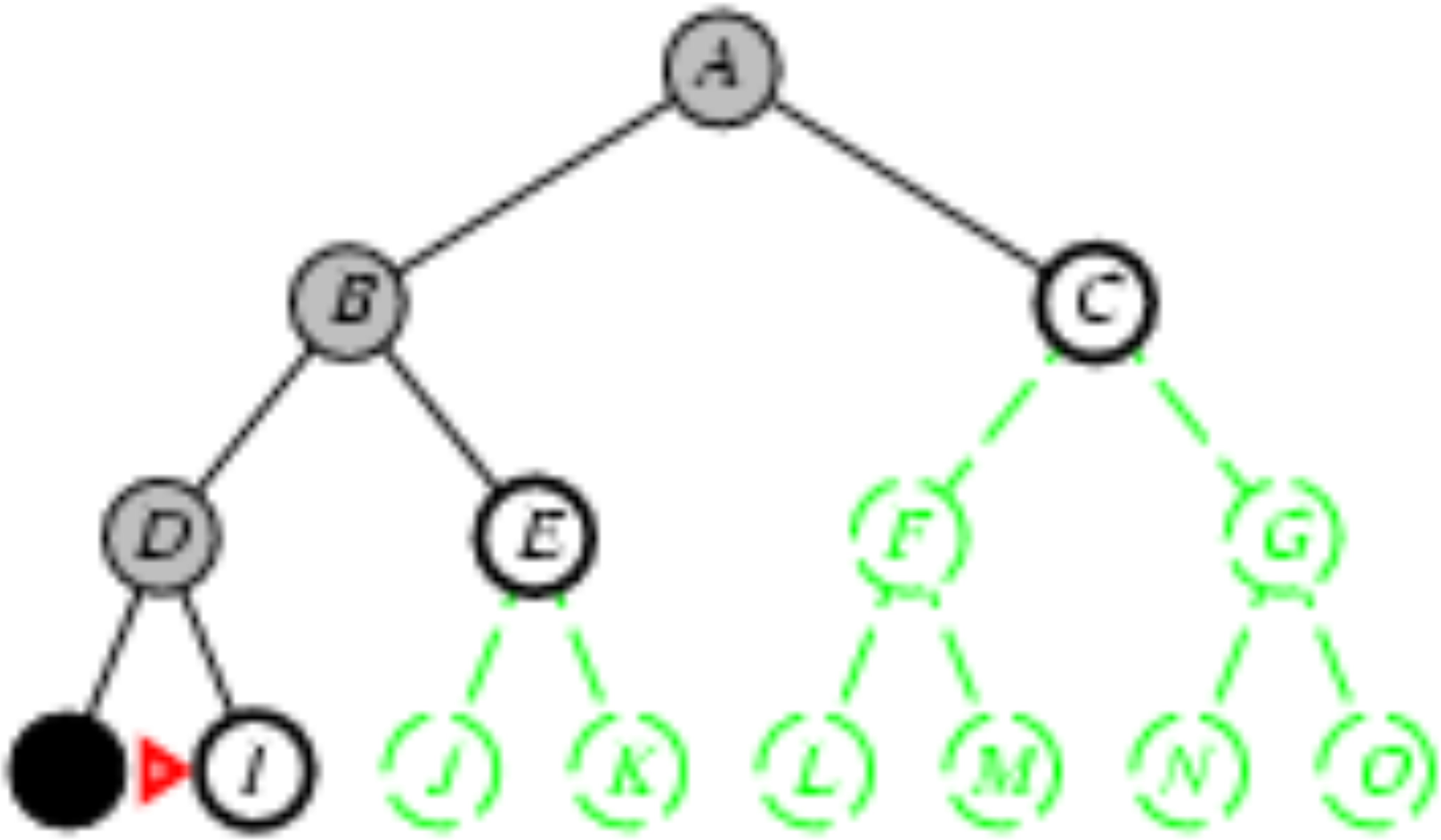
- Expand deepest unexpanded node
- Implementation: fringe = LIFO queue, i.e., put successors at front

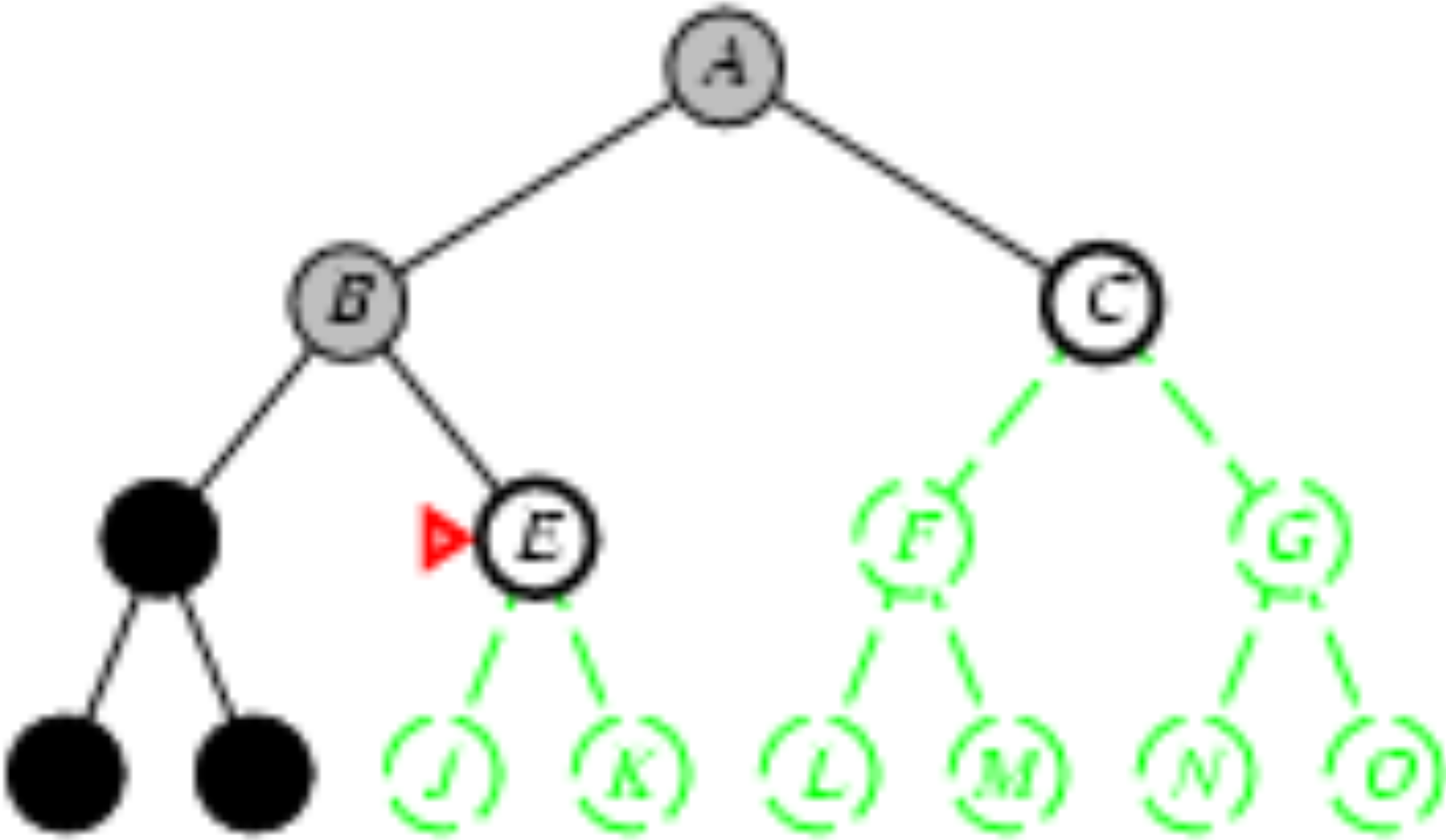


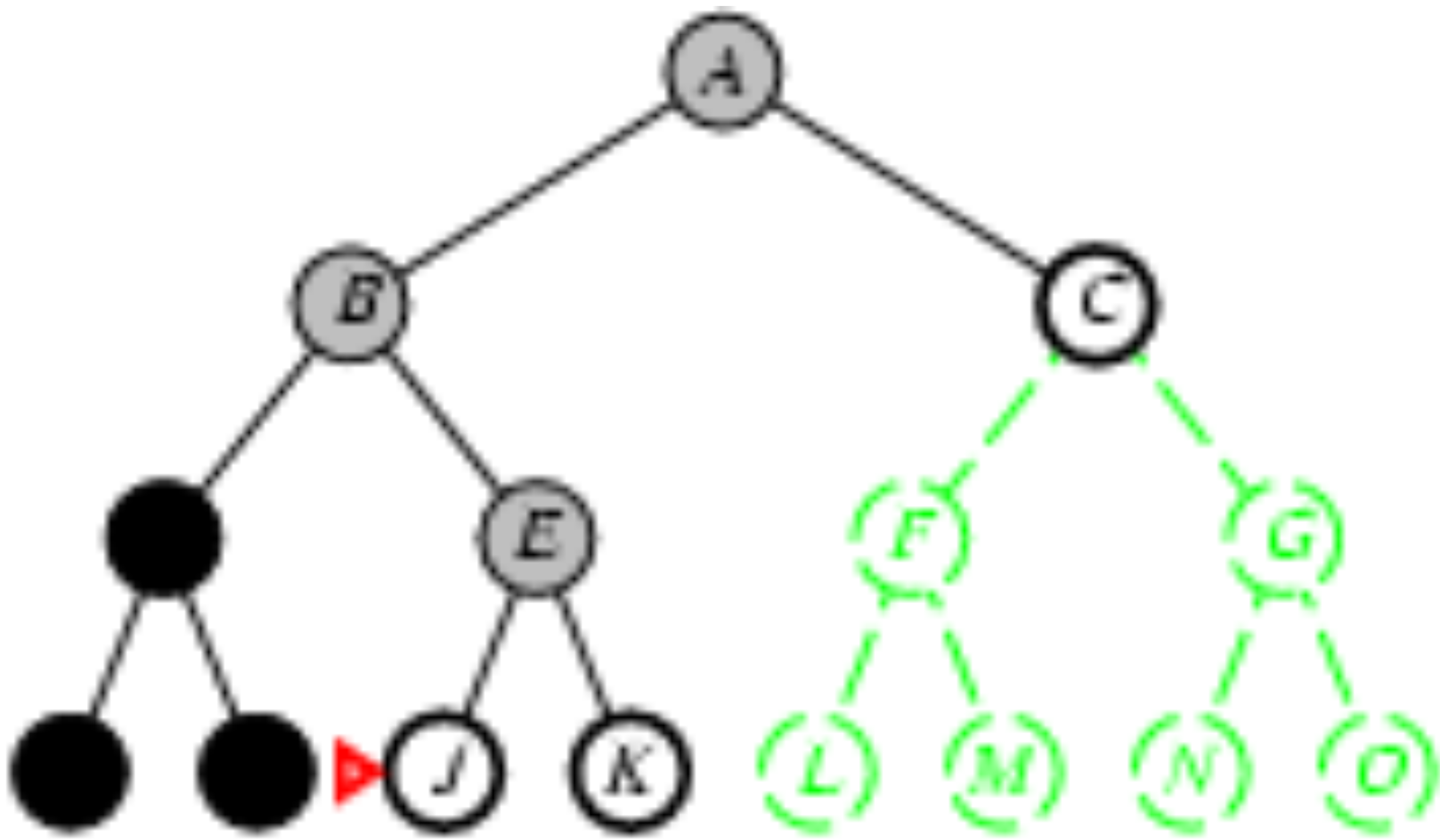

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

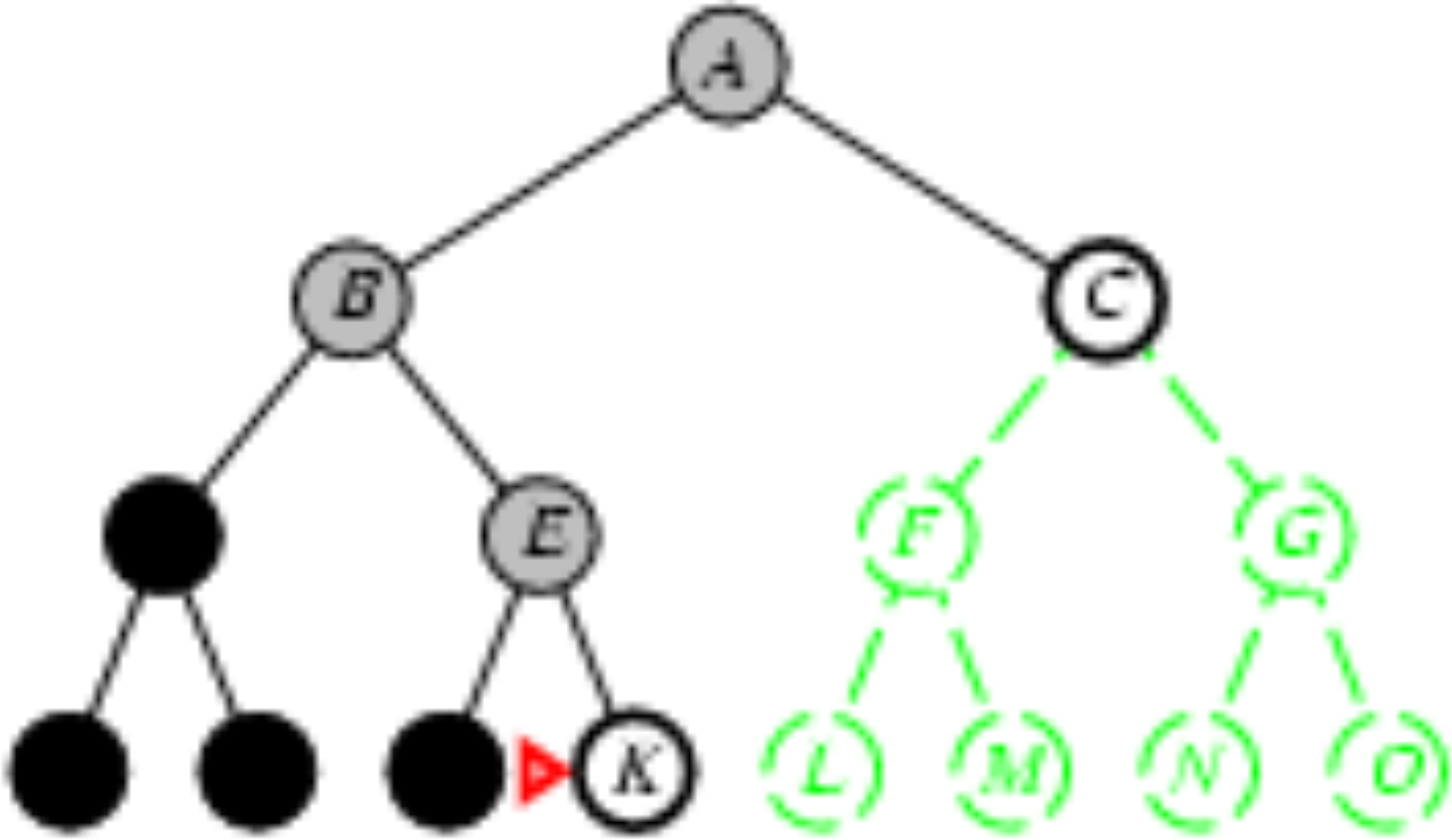

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

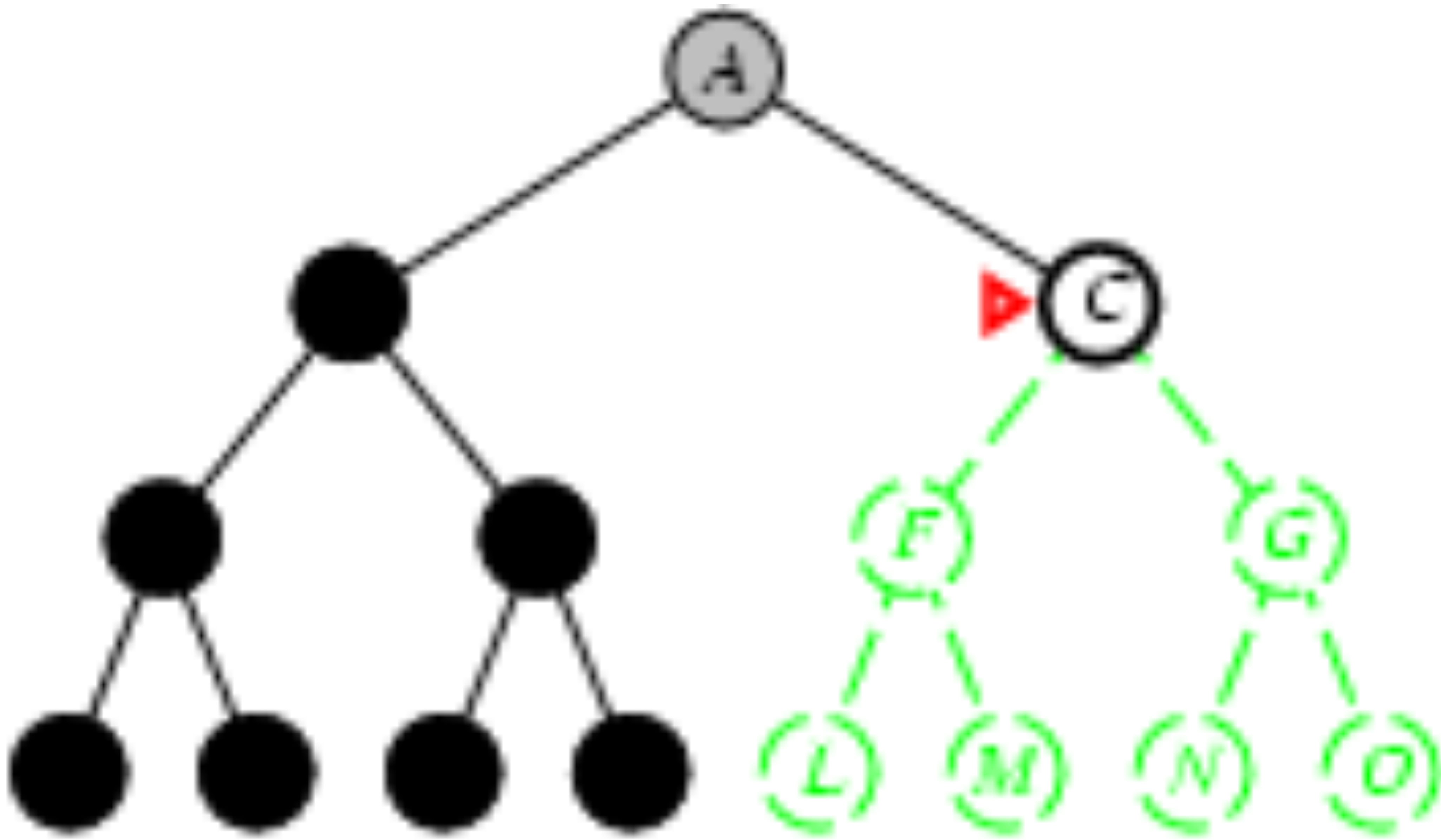

A
B
C
D
E
F
G
I
J
K
L
M
N
O

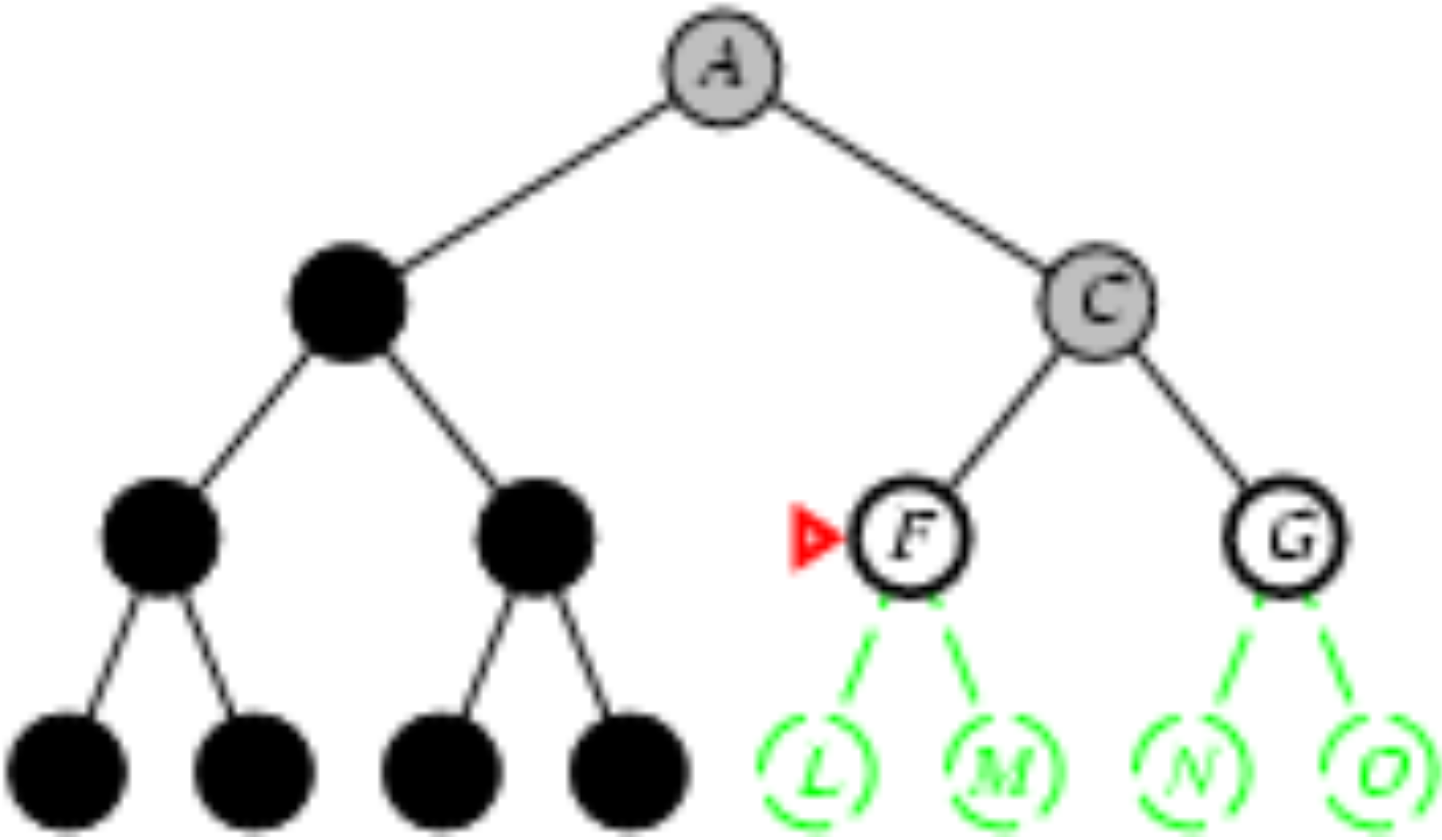

A
B
C
E
F
G
J
K
L
M
N
O



A
B
C
E
F
G
J
K
L
M
N
O


A
B
C
E
F
G
K
L
M
N
O

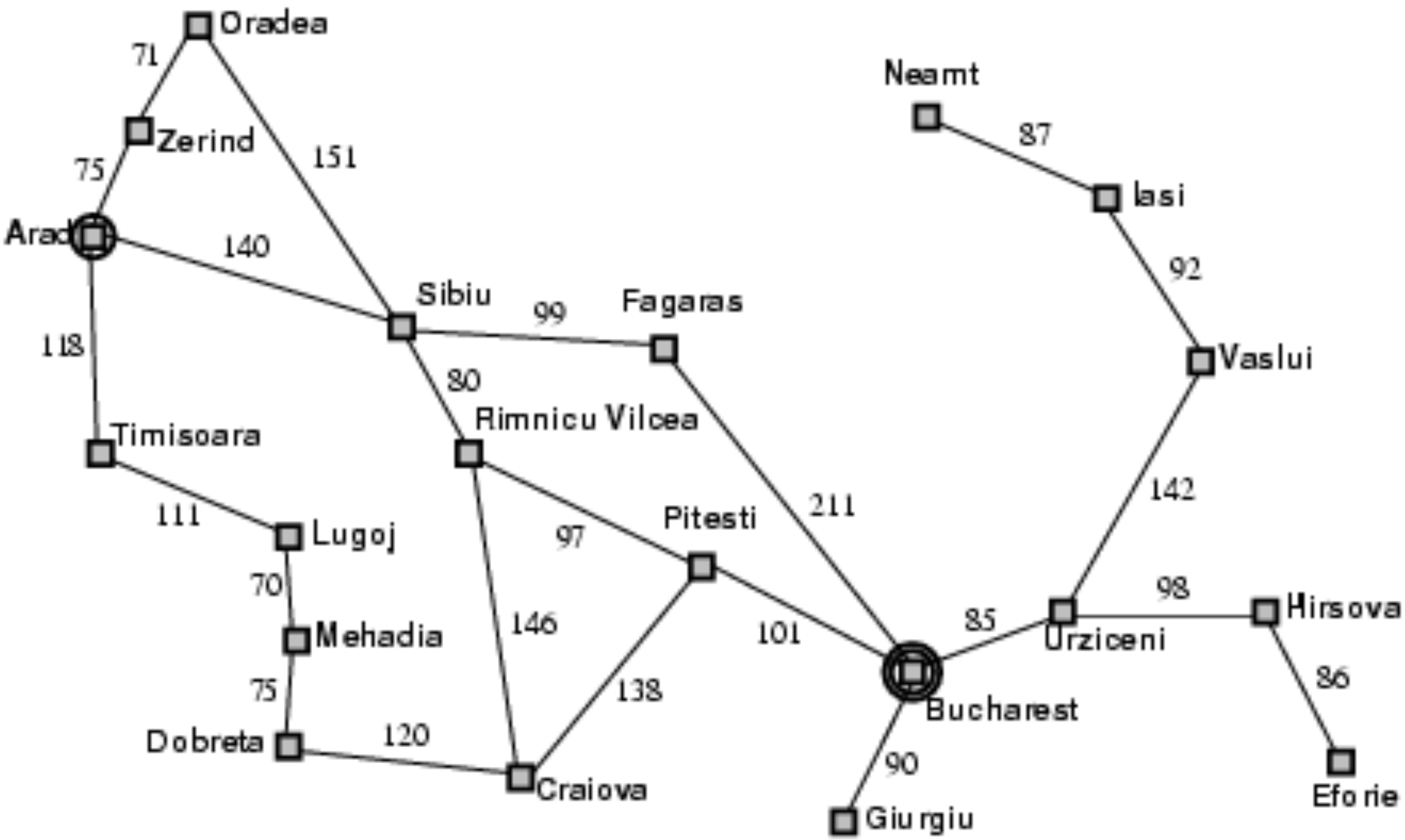

A
C
F
G
L
M
N
O


A
C
F
G
L
M
N
O

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? $O(bm)$: linear space!
- Optimal? No

From Arad to Bucharest



Exercise time

- Bring up your laptop, and go here:
- https://qiao.github.io/PathFinding.js/visual/
- Try creating a few mazes that can be solved with breadth first search and see how it works

Depth-limited search

- Depth-first search with depth limit l; 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
```

Iterative deepening

- Do depth-limited search at increasing depths

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*

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

Bring up your laptop again

- Create a maze that you think will be better solved by depth-first search than breadth-first search
- Take a screenshot of the maze you created (with breadth-first search run on it), and upload this image as exercise 1
- No, it doesn't have to be anything special
- Yes, each one of you has to upload an individual image

First assignment

- Implement a number of search algorithms in Sokoban
- Compare the performance of these algorithms
- Hand in the source code

Important

- Search in state space, not in physical space
- For expanding a node, copy the game state and take actions in it - this is the successor function which returns the next state

TA and office hours

- TA hours held by the TAs (zoom)
  - Tuesday, noon
  - Friday, noon
- Office hours held by Julian (zoom)
  - Tuesday, 1 pm

TA or office hours?

- Go to the TA hours with all implementation questions and questions concerning how algorithms work, reports are written etc
    - If the TAs can’t explain it to you, go to Julian
- Go to Julian (office hours) with administrative questions and everything not listed here
    - Don’t go to Julian with technical problems without going to the TAs first