

Artificial Intelligence

CE-417, Group 1

Computer Eng. Department

Sharif University of Technology

Spring 2020

By Mohammad Hossein Rohban, Ph.D.

Courtesy: Most slides are adopted from CSE-573 (Washington U.), original
slides for the textbook, and CS-188 (UC. Berkeley).

Problem Space and Uninformed Search

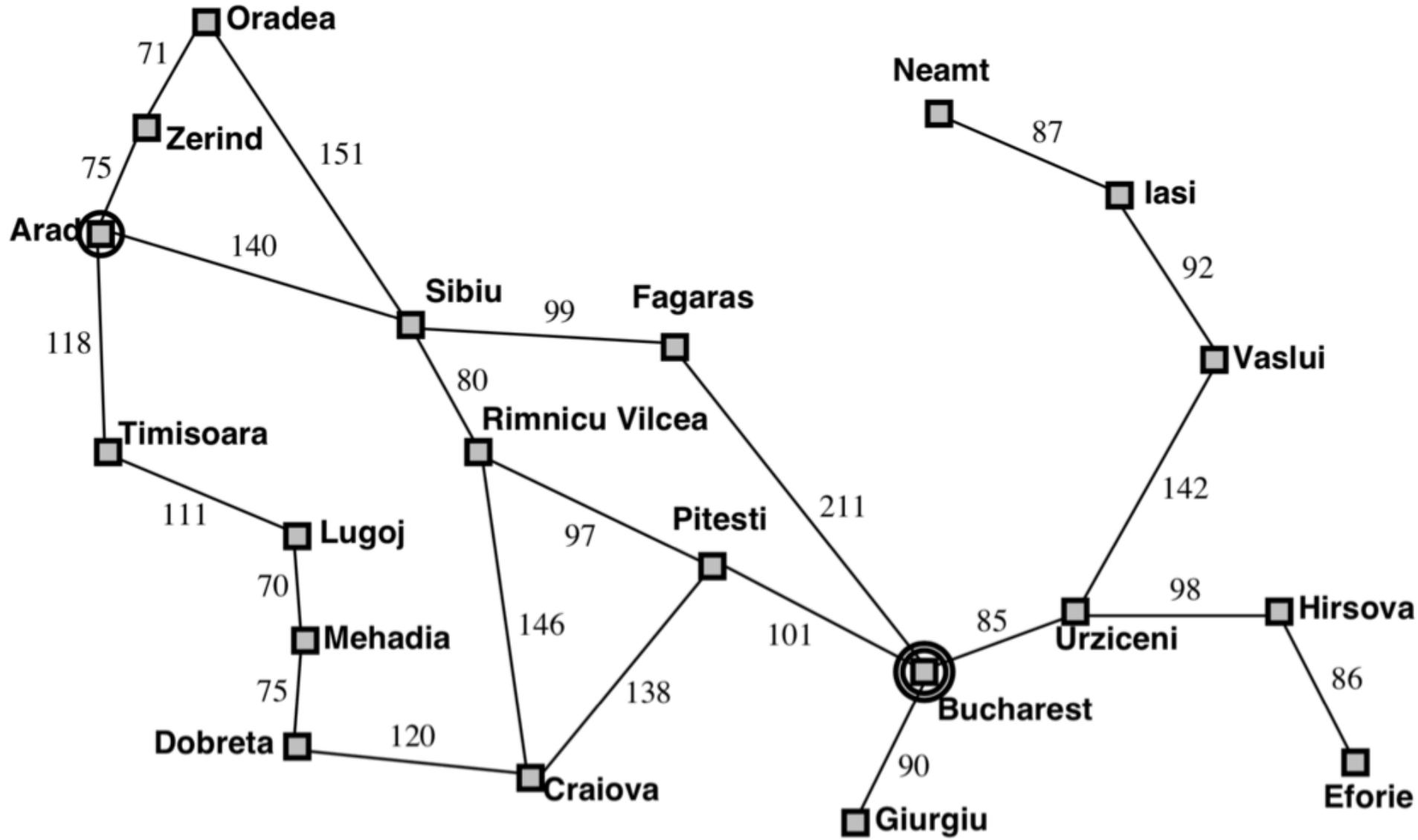
Problem solving agents

- On holiday in Romania; currently in Arad.

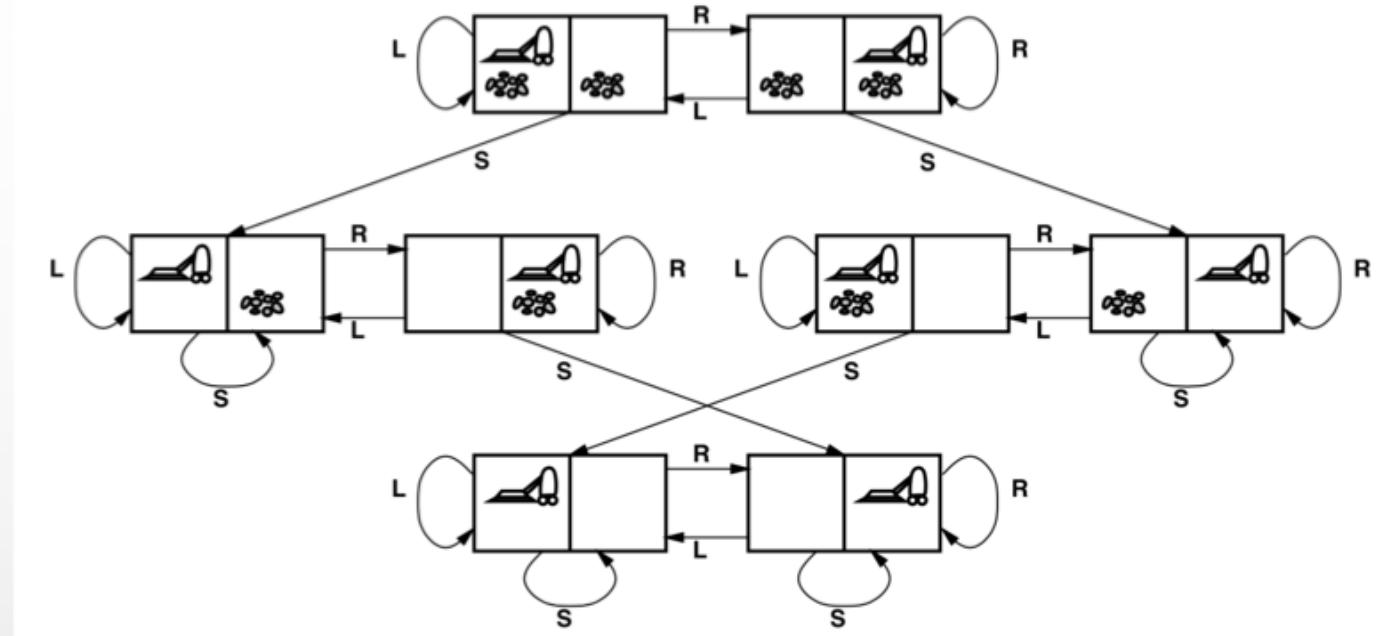
Flight leaves tomorrow from Bucharest

- 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

Problem solving agents (cont.)

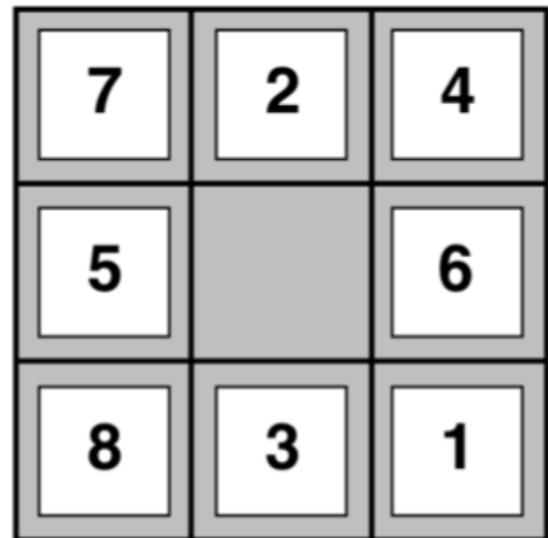


Another example: vacuum world

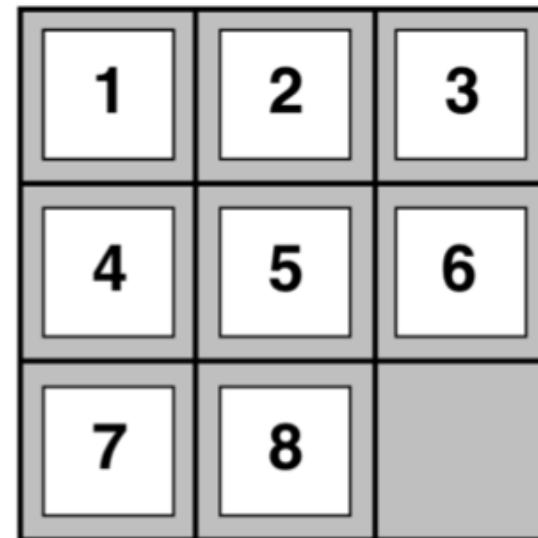


- States: **integer dirt and robot locations** (ignore dirt amounts etc.)
- Actions: **Left, Right, Suck, NoOp**
- Goal test: **no dirt**
- Path cost: **1 per action (0 for NoOp)**

Another example: The 8-puzzle



Start State



Goal State

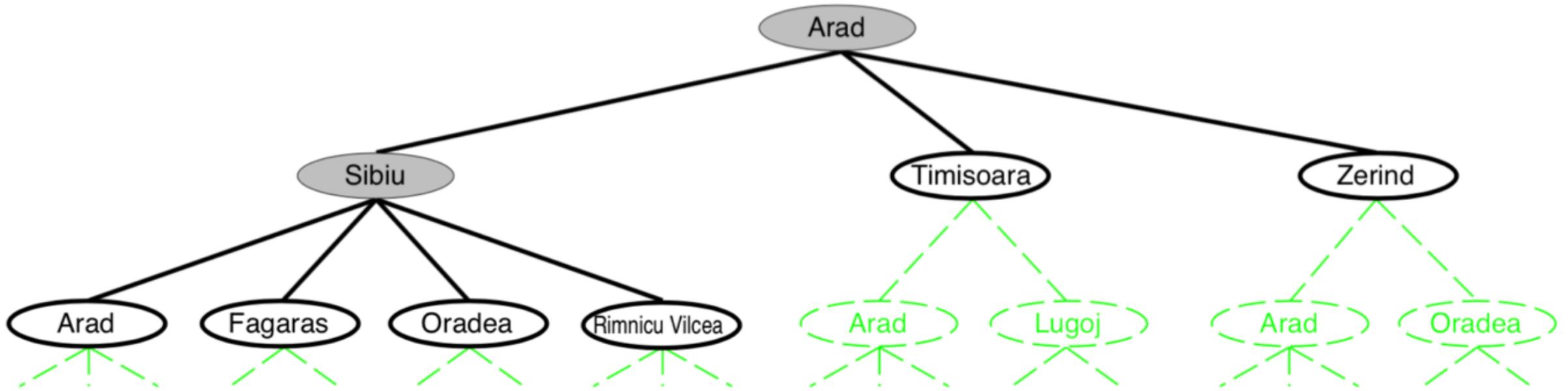
- States: **integer locations of tiles** (ignore intermediate positions)
- Actions: **move blank left, right, up, down** (ignore unjamming etc.)
- Goal test: **= goal state** (given)
- Path cost: **1 per move**
- [Note: optimal solution of n-Puzzle family is NP-hard]

Tree Search Algorithms

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

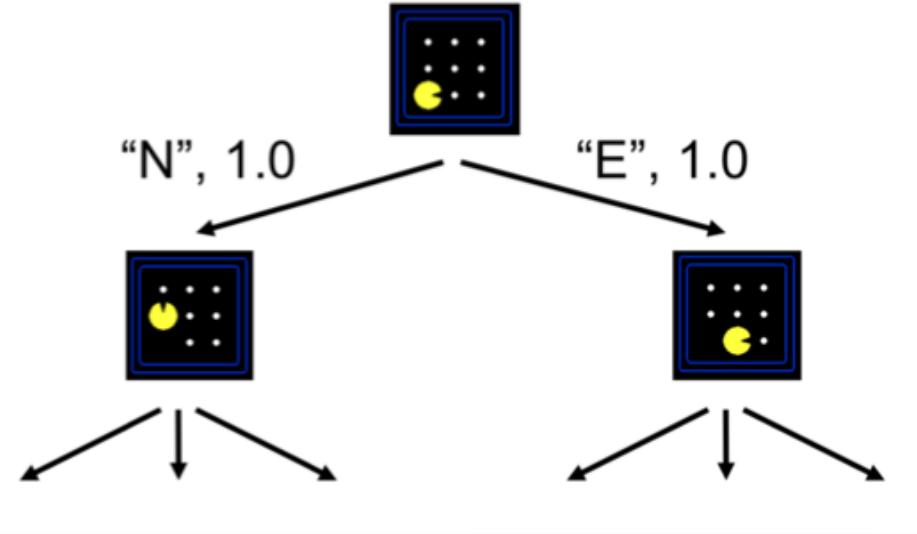
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node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

Search Tree Example



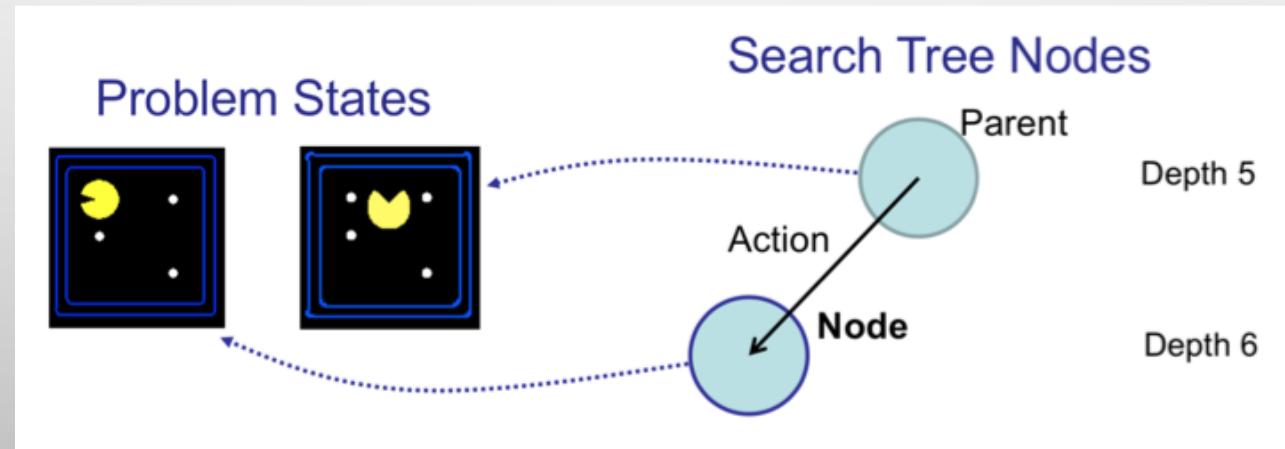
Search Tree

- A search tree:
 - Start state at the root node
 - Children correspond to successors
 - Nodes **contain** states, correspond to **PLANS** to those states
 - Edges are labeled with actions and costs
 - For most problems, we can never actually build the whole tree



States vs. Nodes

- Vertices in state space graphs are problem states
- Represent an abstracted state of the world
- Have successors, can be goal / non-goal, have multiple predecessors
- Vertices in search trees (“Nodes”) are plans
- Contain a **problem state** and one parent, a path length, a depth, and a cost
- Represent a plan (sequence of actions) which results in the node’s state
- **The same problem state may be achieved by multiple search tree nodes**



Search Strategies

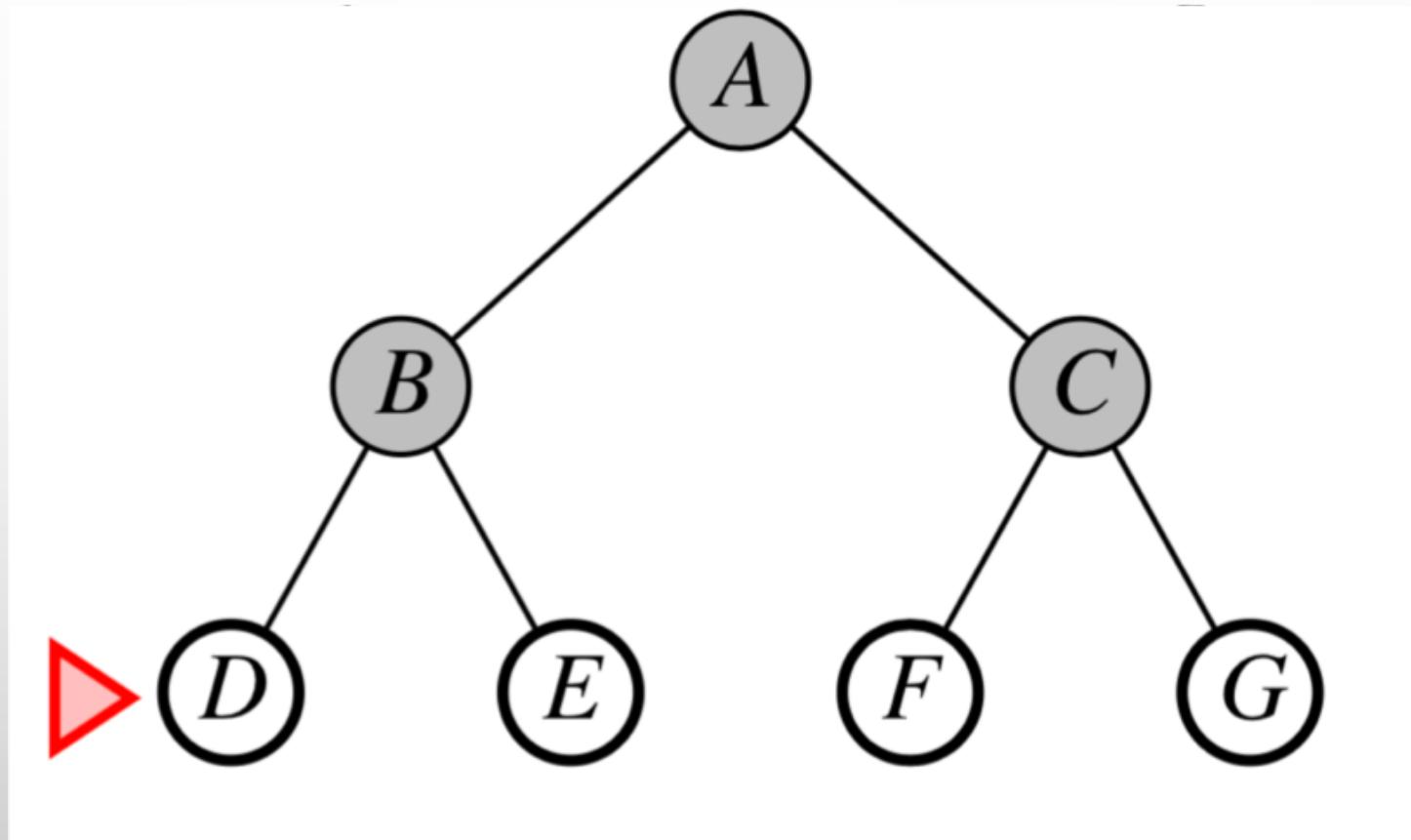
- A strategy is defined by picking **the order of node expansion**
- Strategies are evaluated along the following dimensions:
 - **completeness:** does it always find a solution if one exists?
 - **time complexity:** number of nodes generated/expanded
 - **space complexity:** maximum number of nodes 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
 - **d :** depth of the least-cost solution
 - **m :** maximum depth of the state space (may be ∞)

Search Strategies

- **Uninformed** 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

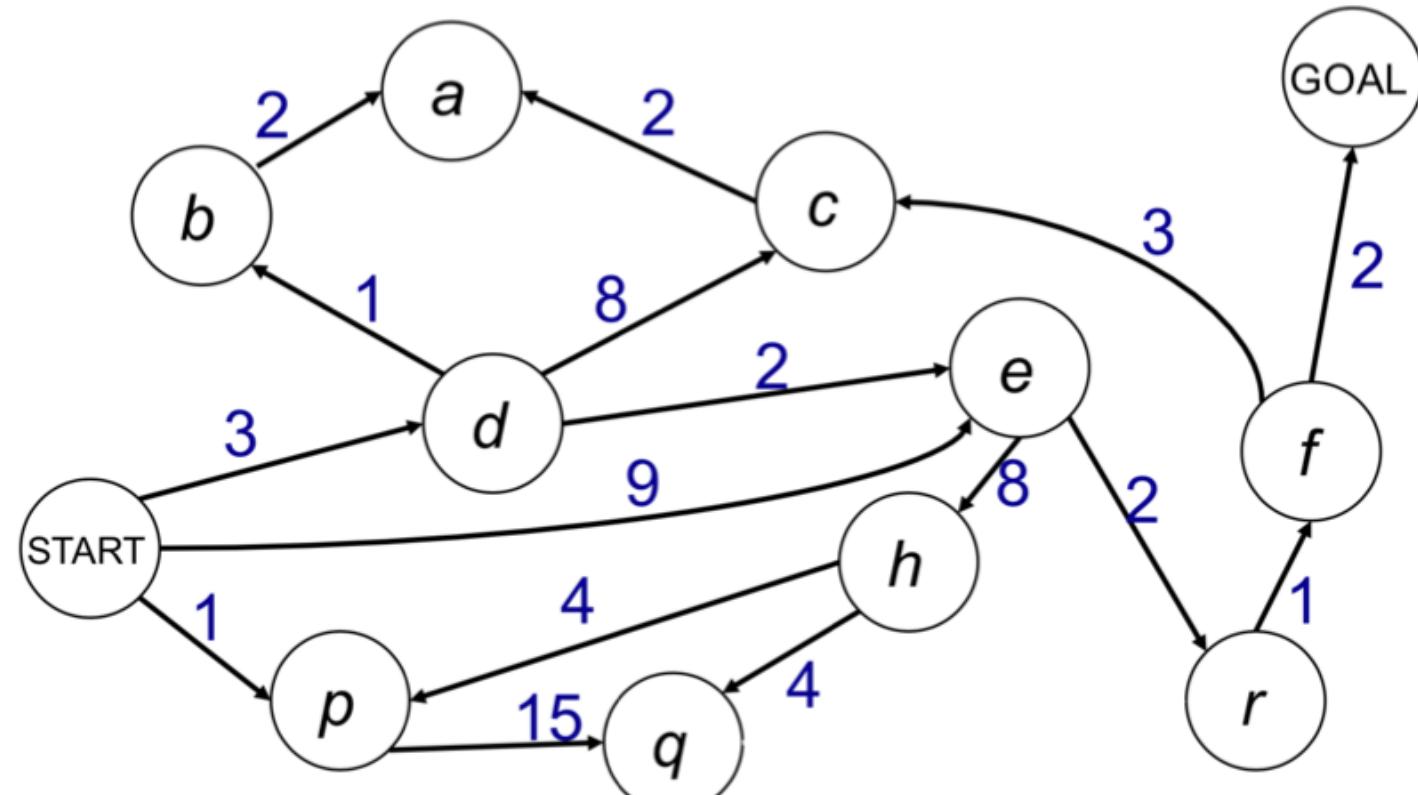
- Expand shallowest unexpanded node



Properties of breadth-first search

- **Complete:**
 - Yes (if b is finite)
- **Time:**
 - $1+b+b^2+b^3+\dots+b^d+b(b^d-1)=O(b^{d+1})$, i.e. exp. in d
- **Space:**
 - $O(b^{d+1})$ (keeps every node in memory)
- **Optimal:**
 - Yes (if cost = 1 per step); not optimal in general
 - Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Costs on Actions



- Objective: Path with smallest overall cost
- BFS will return shortest path in terms of number of transitions
 - It doesn't find the least cost path.

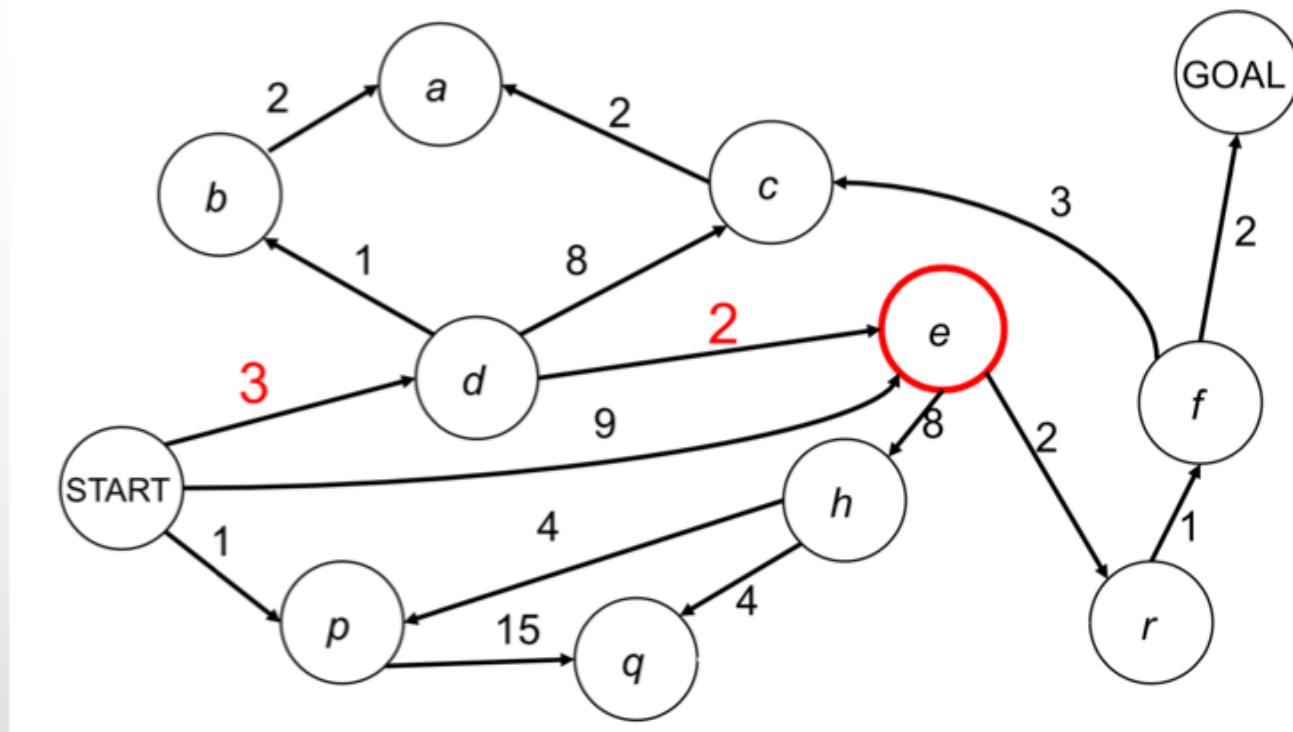
Best-first search

- Generalization of breadth-first search
- Cost function $f(n)$ applied to each node
 - Breadth-first search : $f(n) = \text{depth}(n)$
 - Dijkstra's Algorithm (Uniform cost) : $f(n) = \text{the sum of edge costs from start to } n$

Uniform Cost Search

- Best first, where

$f(n)$ = “cost from **start** to **n**”

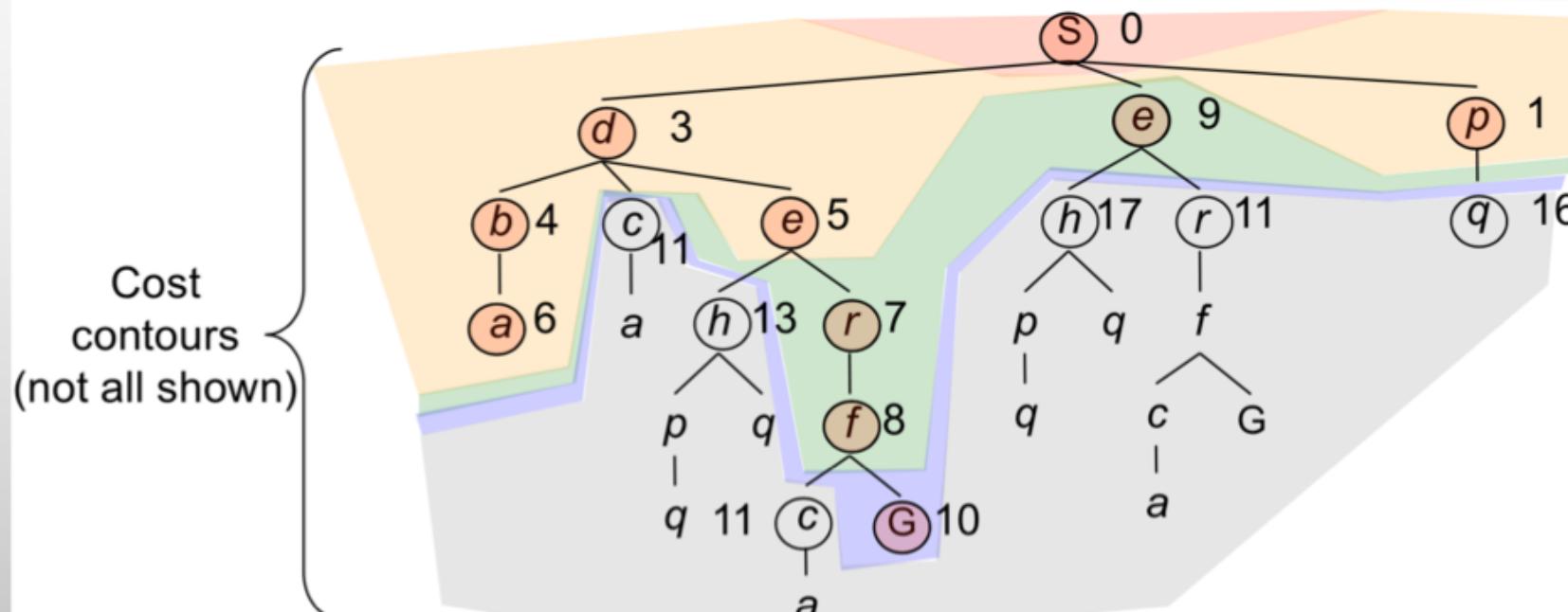
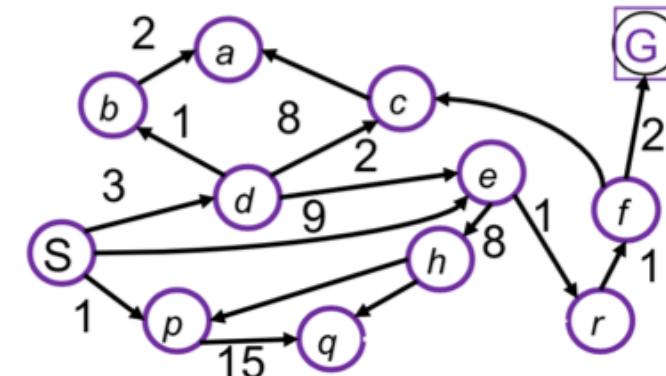


aka “Dijkstra’s Algorithm”

Uniform Cost Search (cont.)

Expansion order:

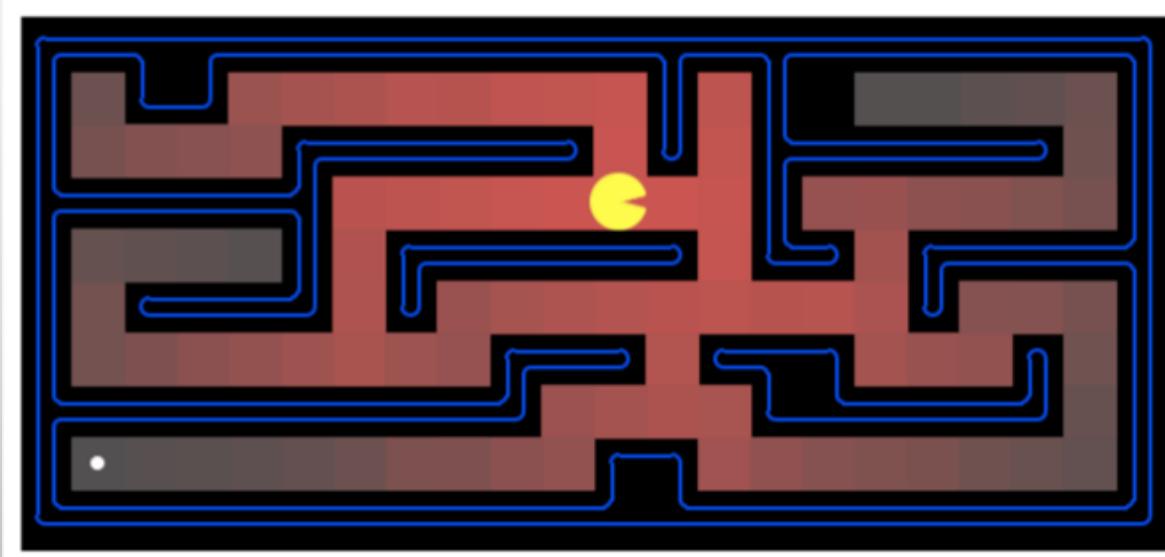
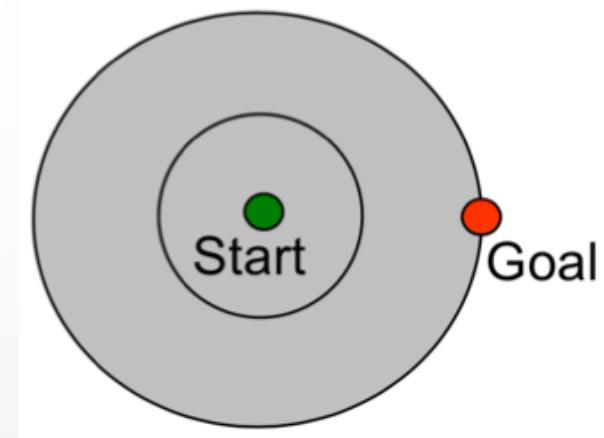
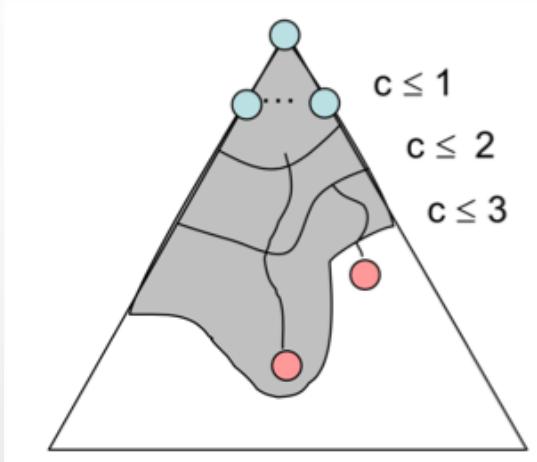
S, p, d, b, e, a, r, f, e, G



Uniform-cost search

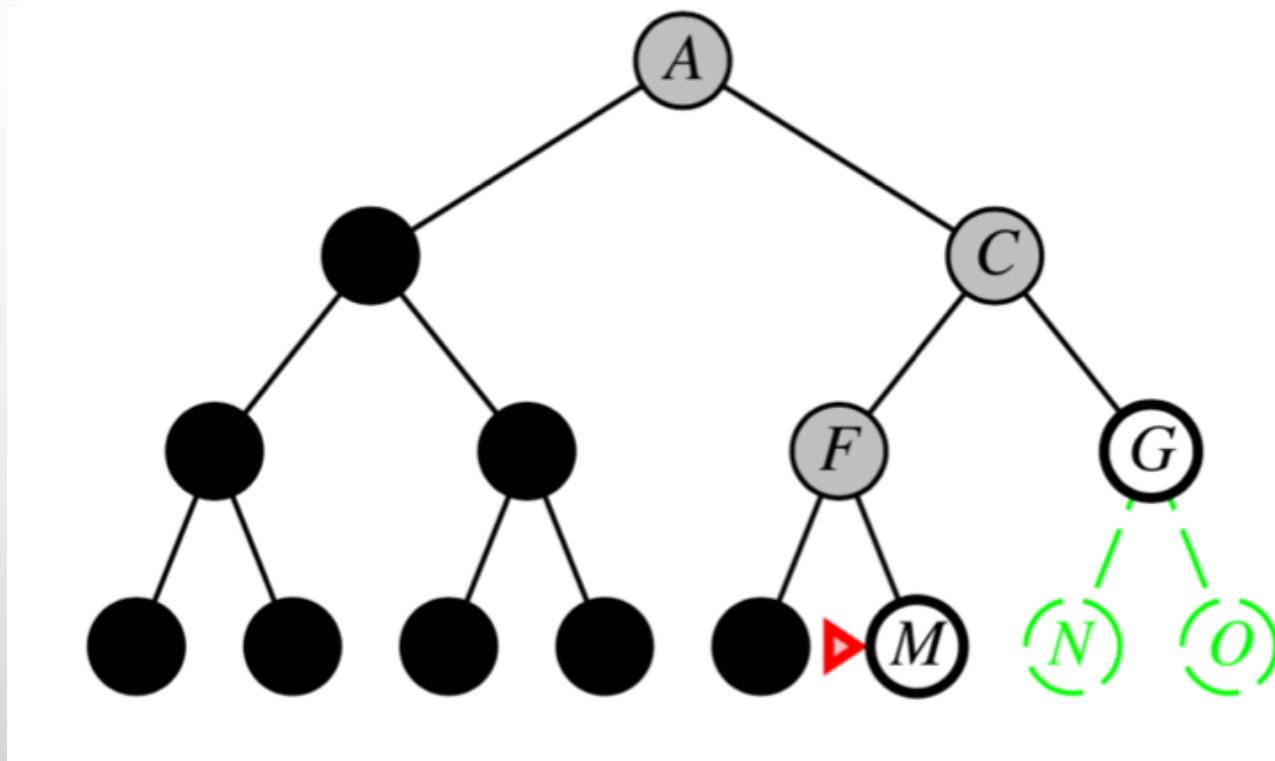
- **Complete:**
 - Yes, if step cost $\geq \varepsilon$
- **Time:**
 - # of nodes with $f \leq$ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$ where C^* is the cost of the optimal solution
- **Space:**
 - # of nodes with $f \leq$ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$
- **Optimal:**
 - Yes—nodes expanded in increasing order of $f(n)$
- **Caveat:** Explores options in every “direction” (No information about goal location)

Uniform-cost search (cont.)



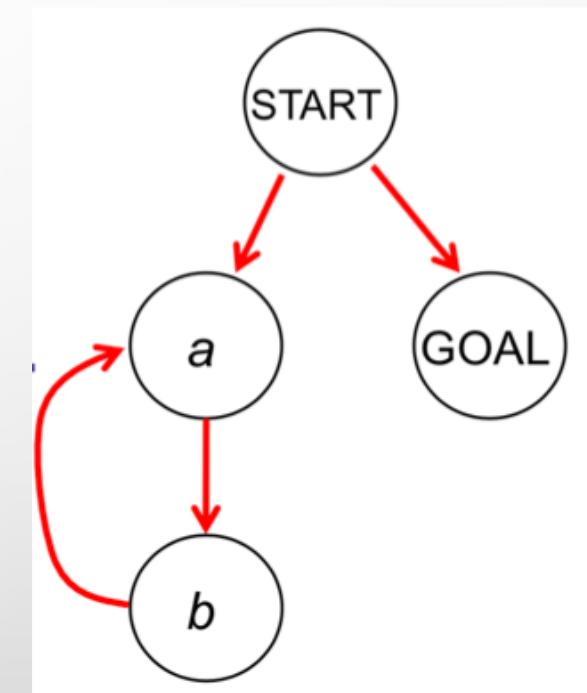
Depth-first search

- Expand deepest unexpanded node



Properties of depth-first search

- **Complete:**
 - No: fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along path
 - \Rightarrow 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



Combining BFS and DFS?

- DFS is efficient in space complexity
- BFS is better in time complexity
- How can we combine strength of both in a method?

Depth-limited search

= depth-first search **with depth limit l** , i.e., 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 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
```

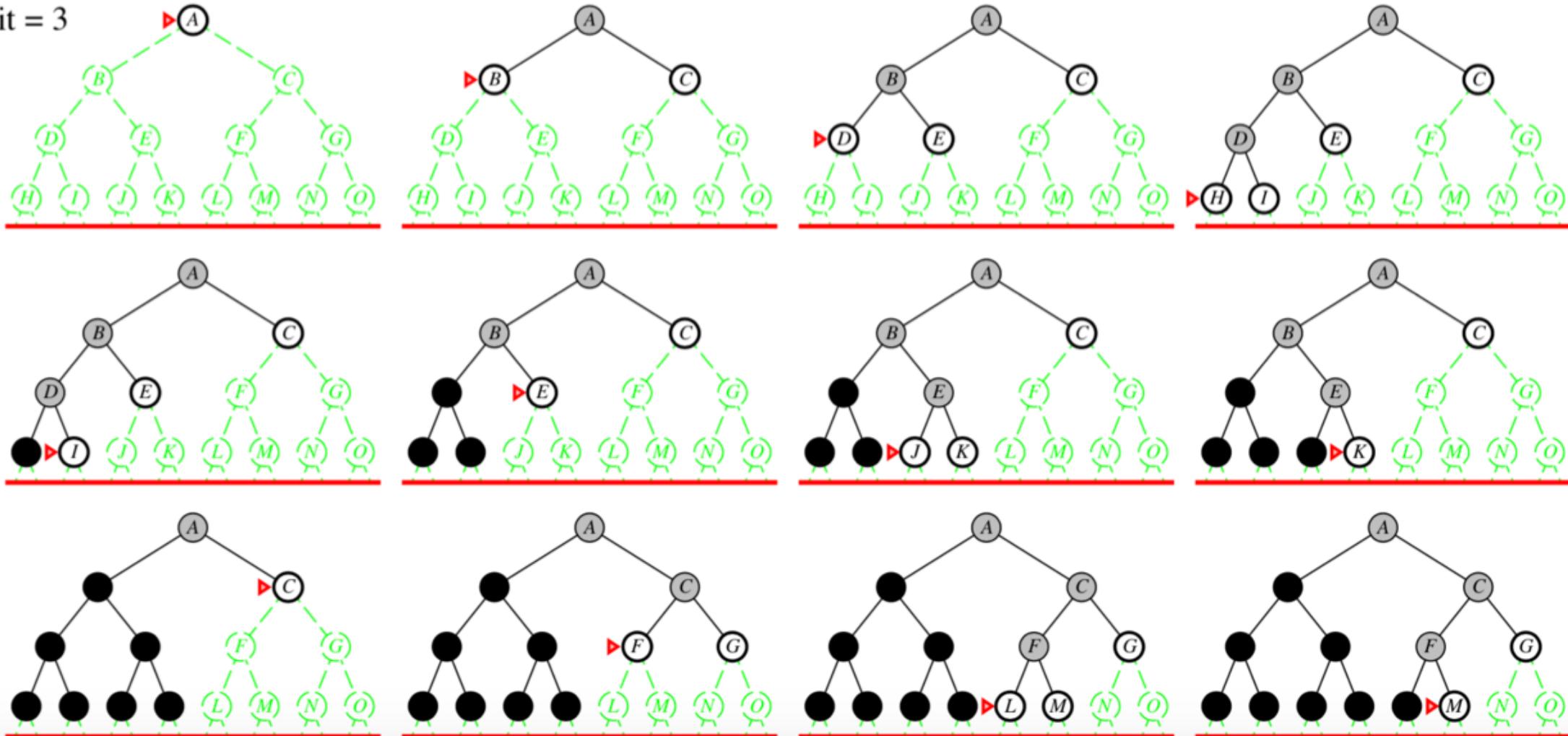
Iterative deepening search (cont.)

- Gradually increasing the limit in depth-limited search, until the solution is found:

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

Iterative deepening search (cont.)

Limit = 3



Properties of iterative deepening search

- **Complete:**
 - Yes
- **Time:**
 - $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
 - or more precisely $O(b^d(1 - 1/b)^{-2})$
- **Space:**
 - $O(bd)$
- **Optimal:**
 - Yes, if step cost = 1
 - Can be modified to explore uniform-cost tree

Properties of iterative deepening search (cont.)

- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(IDS) = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$$

$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is generated

Cost of iterative deepening

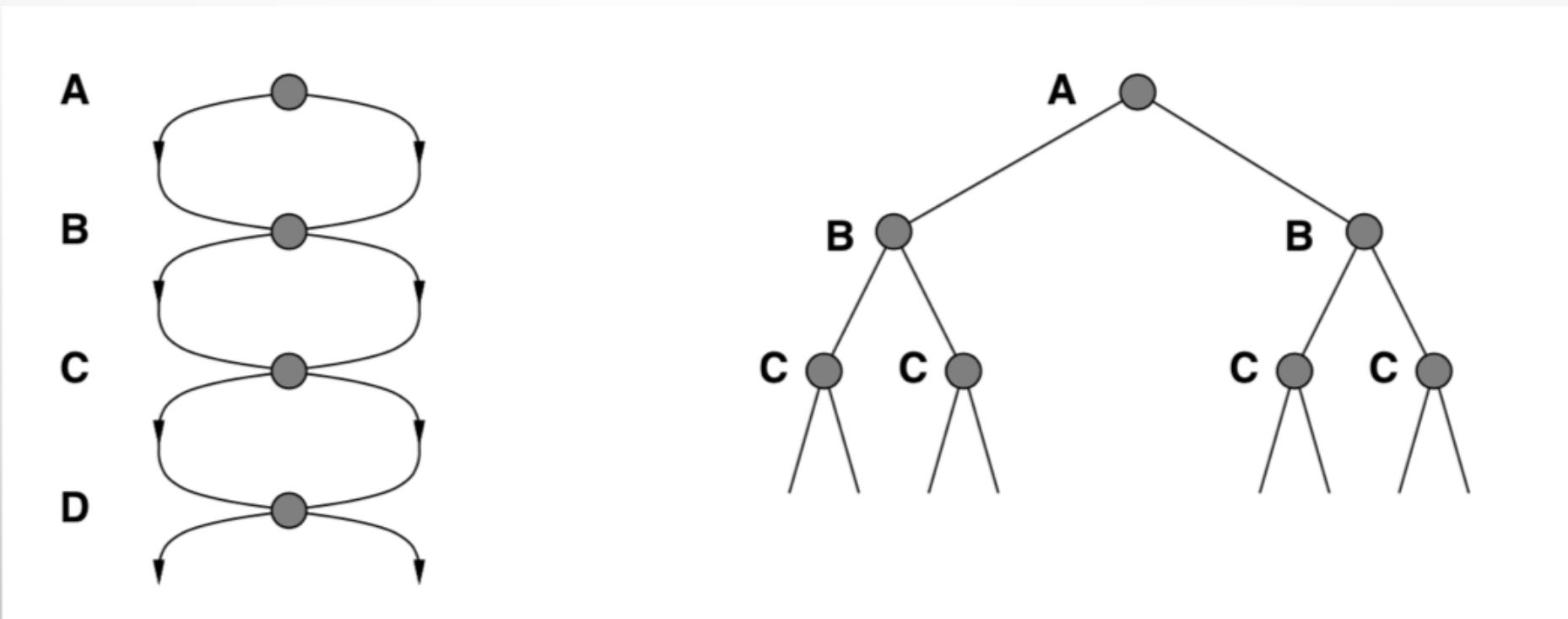
b	ratio ID to DFS
2	3
3	2
5	1.5
10	1.2
25	1.08
100	1.02

Summary of algorithms

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

Repeated states

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



Graph Search

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed  $\leftarrow$  an empty set
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
  end
```

Graph Search (cont.)

- On small problems
 - Graph search almost always better than tree search
- Implement your closed list as a dict. or set!
- On many real problems
 - Storage space is a huge concern.
 - Graph search impractical