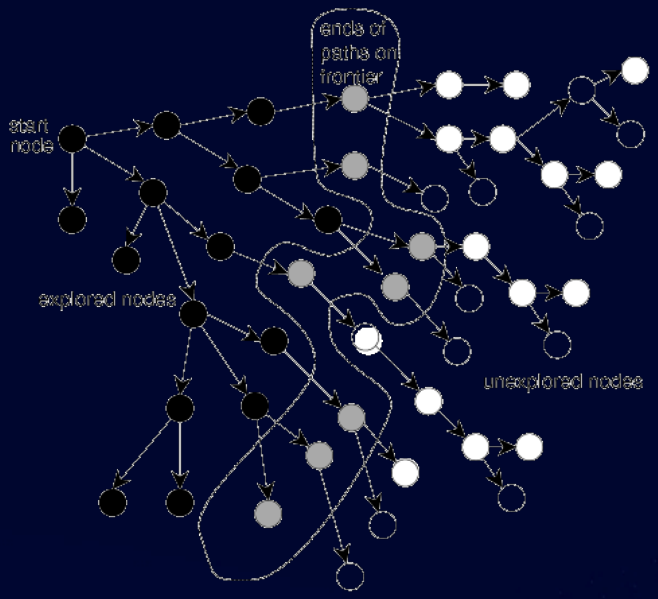
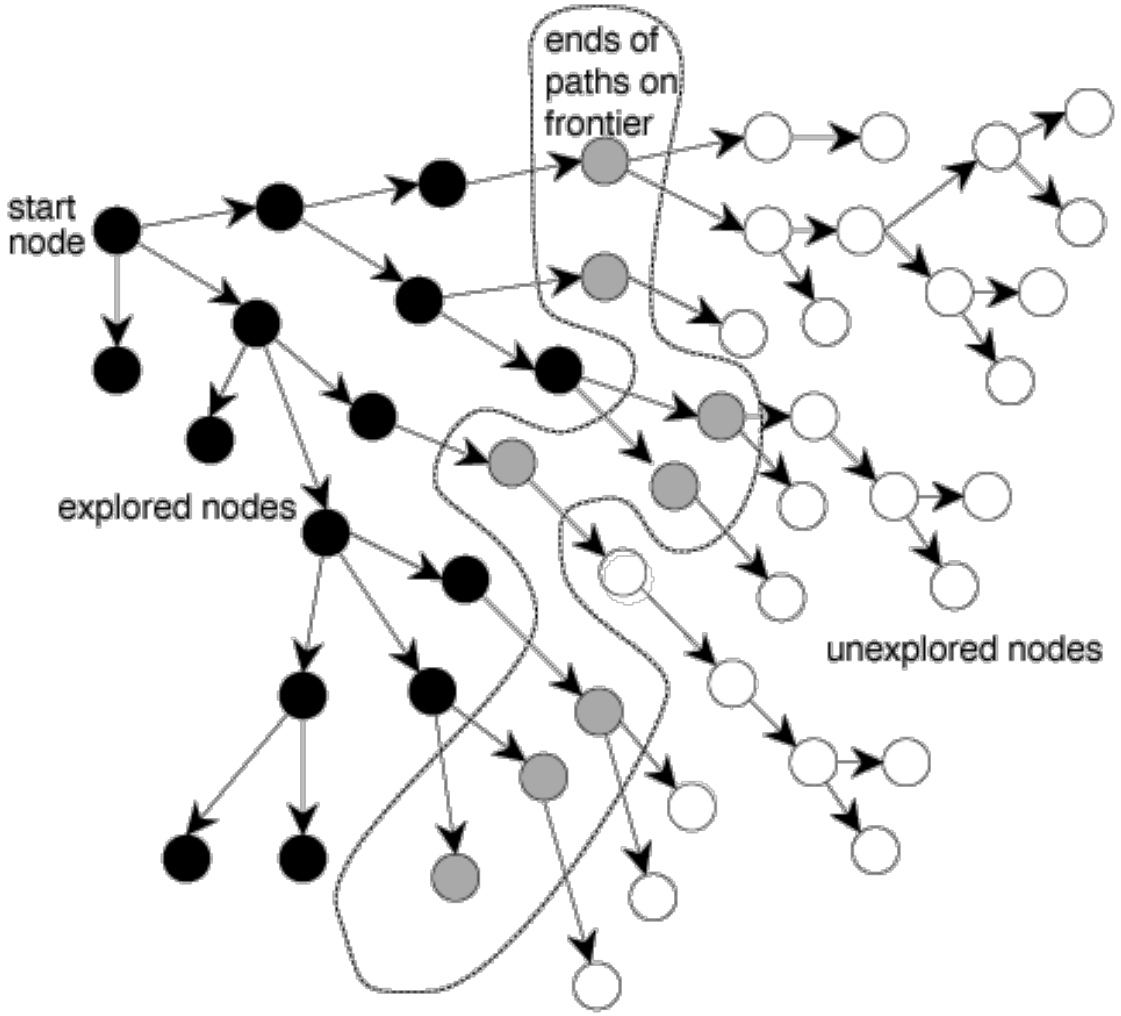


Uninformed Search

Last modified on 2022/03/07 by f.maire@qut.edu.au

Based on material by Stuart Russel, Peter Norvig, Dan Klein and Pieter Abbeel

Search Tree



Make the connection with Dijkstra Algorithm introduced last week

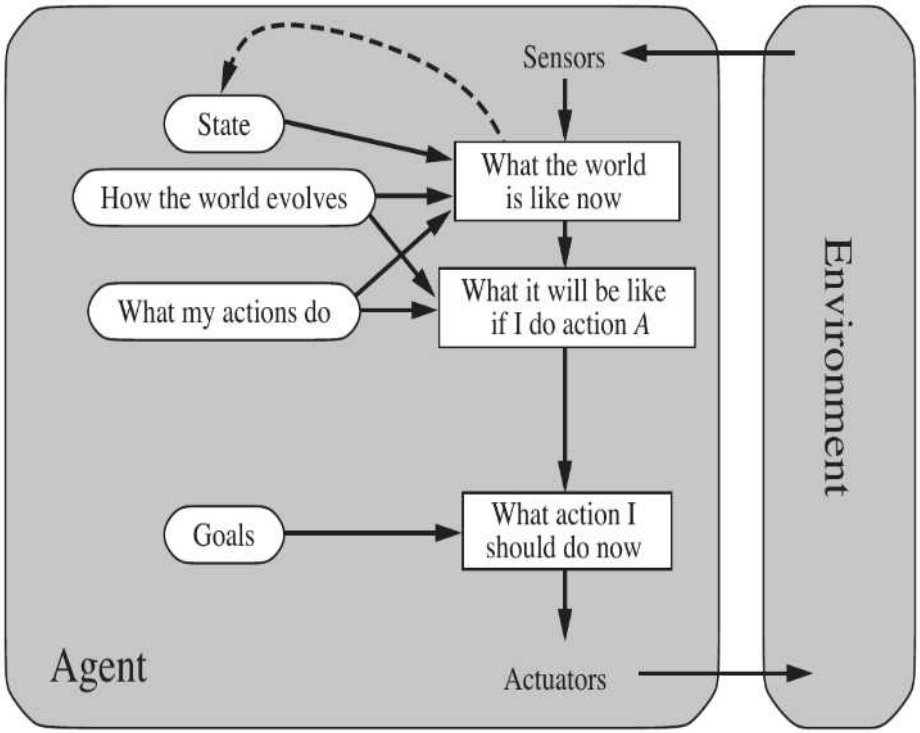
Reading for this week

- Chapter 3, Sections 3.1 to 3.4 of RN
  - Russell and Norvig Textbook:
    Artificial Intelligence, a modern approach
    4th edition

Outline

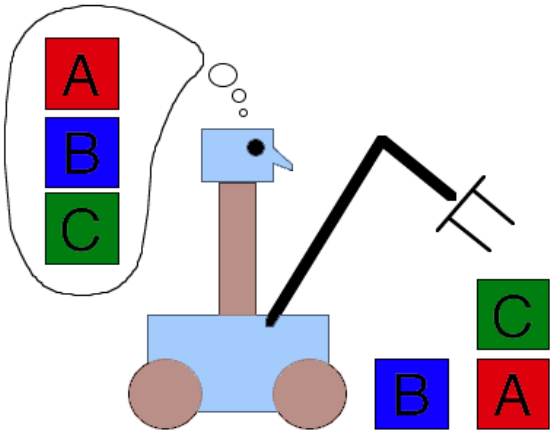
- Planning ahead
- Search problems
- Uninformed search methods
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth limited search
  - Iterative deepening search

A model-based, goal-based agent

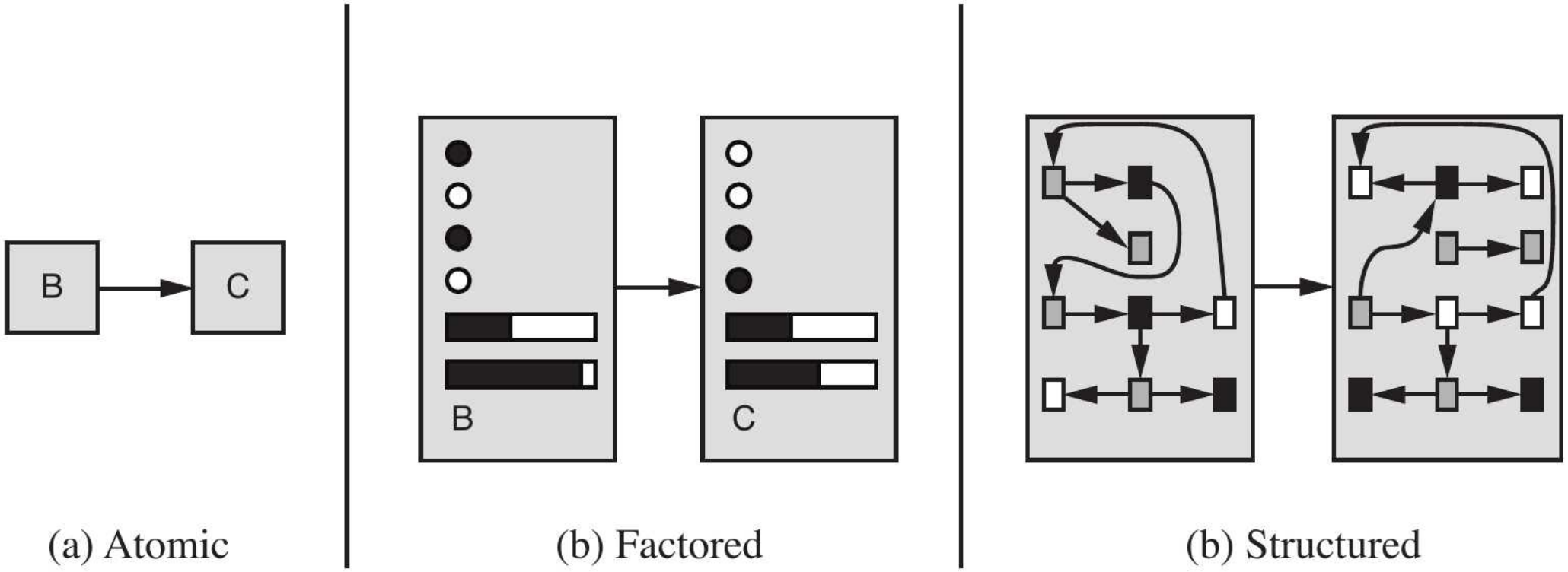


Planning agents

- Planning agents
  - Ask “what if”
  - Decisions based on (hypothesised) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning
- Planning vs. replanning



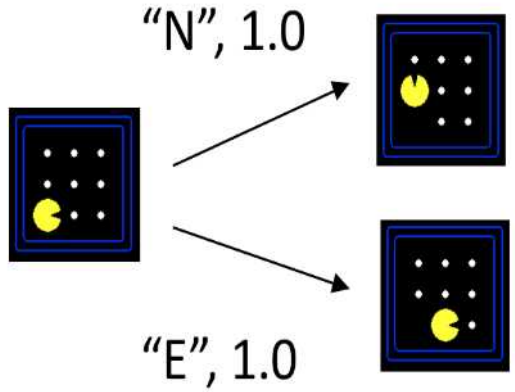
State types

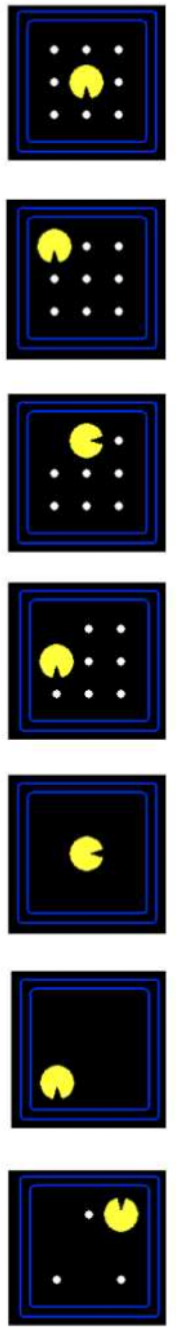


Search problems

- A search problem consists of:
  - A state space
  - A successor function (with actions, costs)
  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state





Example: Romania

We are on holiday in Romania; currently in Arad
Our flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

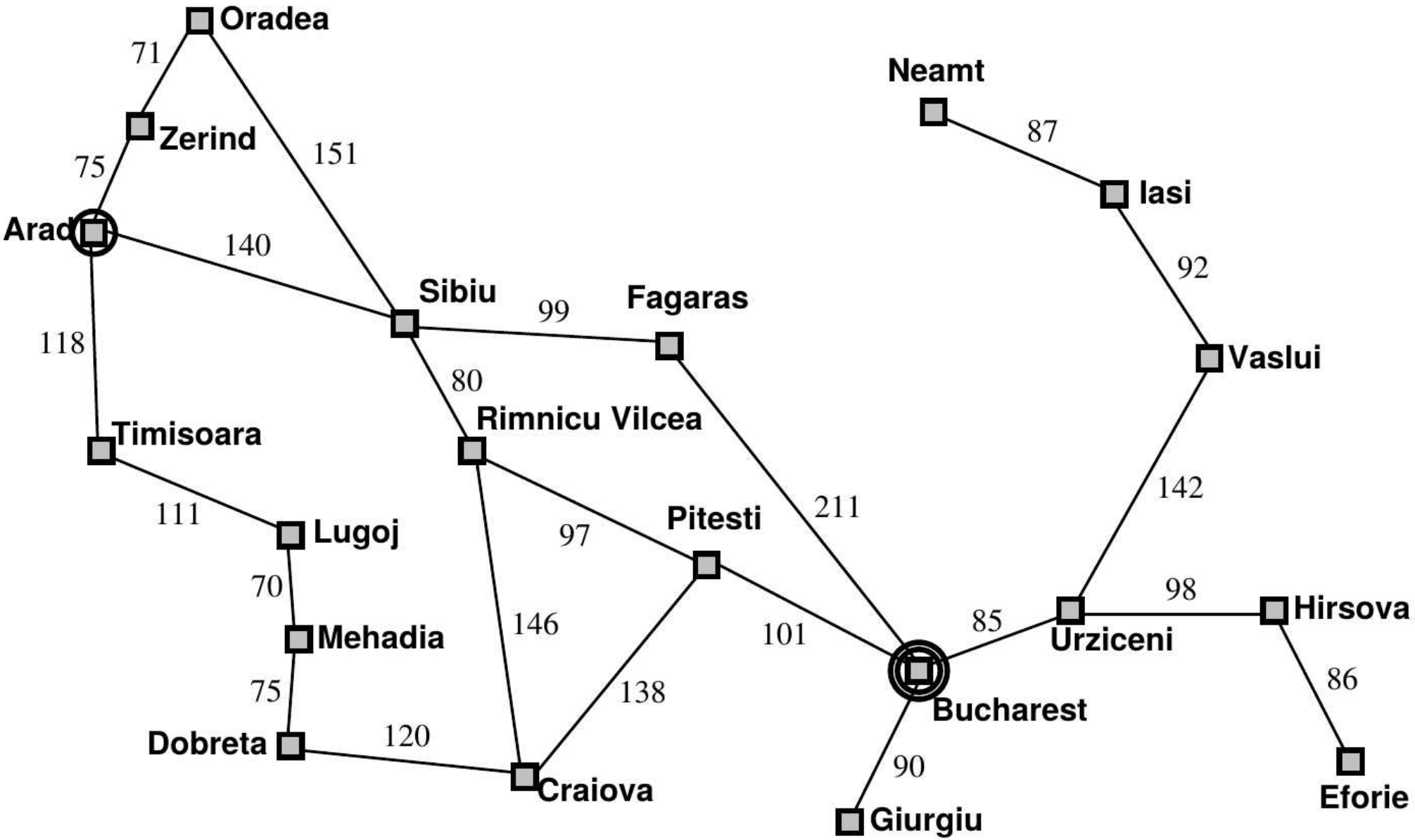
Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem type:

deterministic, fully observable $\Longrightarrow$ single-state problem

Example: Romania



Single-state problem formulation

A problem is defined by five components:

initial state, e.g., $In(Arad)$

actions $\textsc{Actions}(s)$ = set of actions for state s
e.g., $\textsc{Actions}(In(Arad)) = \{Go(Sibiu), Go(Timisoara), Go(Zerind)\}$

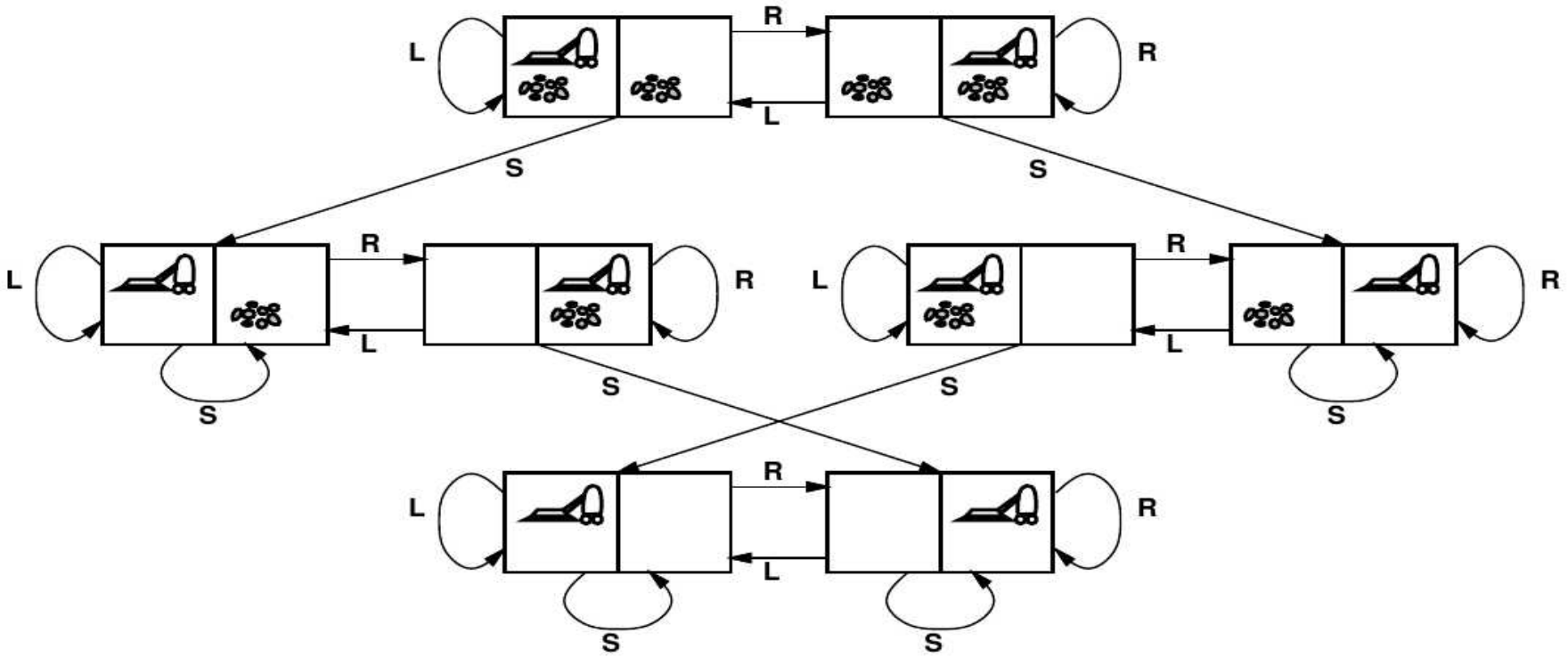
transitions $\textsc{Result}(s, a)$ = the successor state
e.g., $\textsc{Result}(In(Arad), Go(Zerind)) = In(Zerind)$

goal test, can be an explicit set of states, e.g., $\{In(Bucharest)\}$
or an implicit property, e.g., **checkmate** in chess

path cost is the sum of the step costs $c(s, a, s')$
e.g., sum of distances, number of actions executed, etc.
in this chapter we assume c to be positive

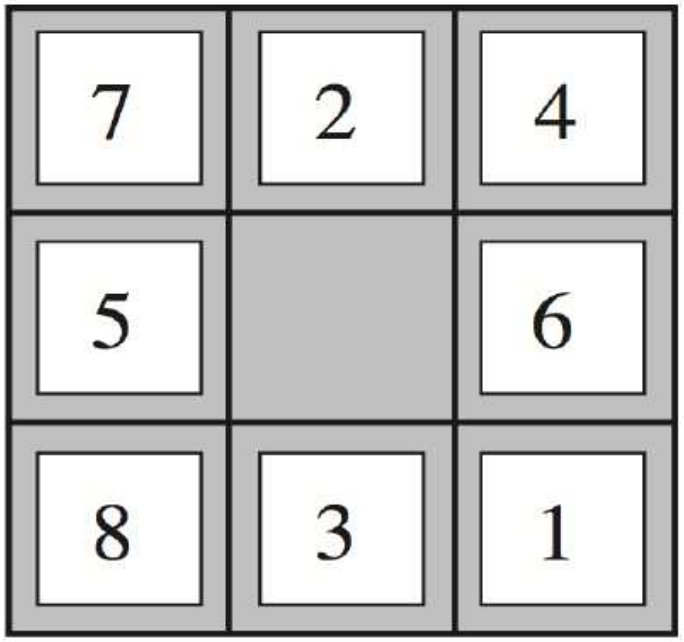
A solution is a sequence of actions
leading from the initial state to a goal state

Example: vacuum world state space graph

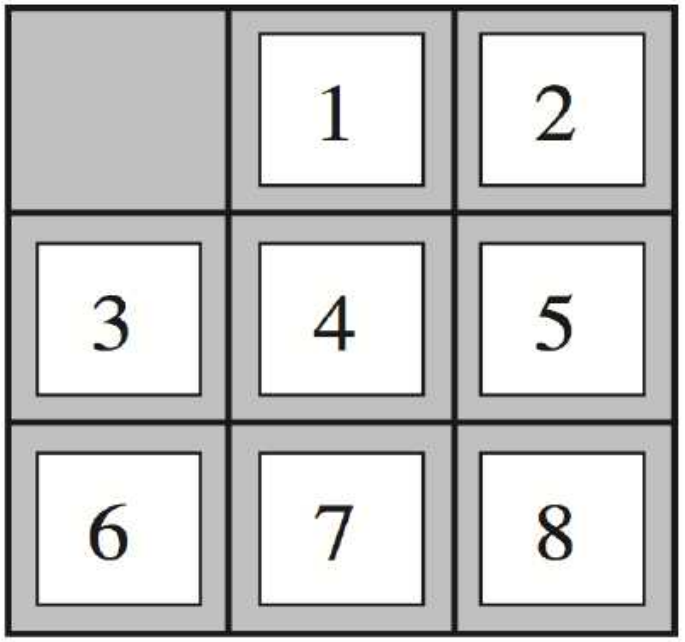


states??: integer dirt and robot locations (ignore dirt amounts etc.)
initial state??: any state
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt in any location
path cost??: 1 per action (0 for $NoOp$)

Example: The 8-puzzle



Start State



Goal State

states??: a 3×3 matrix of integers $0 \leq n \leq 8$

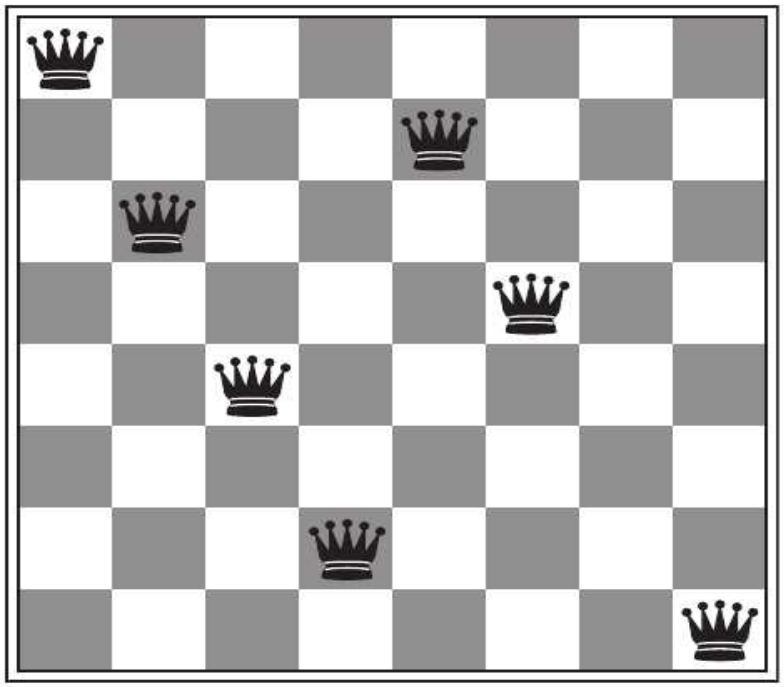
initial state??: any state

actions??: move the blank space: left, right, up, down

goal test??: equal to goal state (given above)

path cost??: 1 per move

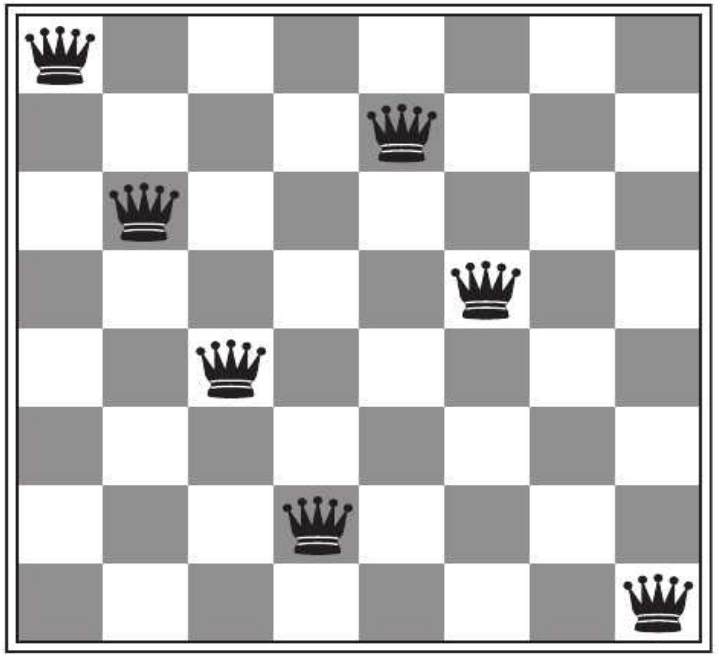
Example: The 8-queens problem



states??: any arrangement of 0 to 8 queens on the board
initial state??: no queens on the board
actions??: add a queen to any empty square
goal test??: 8 queens on the board, none attacked
path cost??: 1 per move

Using this formulation, there are $64 \cdot 63 \cdot \cdots \cdot 57 \approx 1.8 \times 10^{14}$ possible sequences to explore!

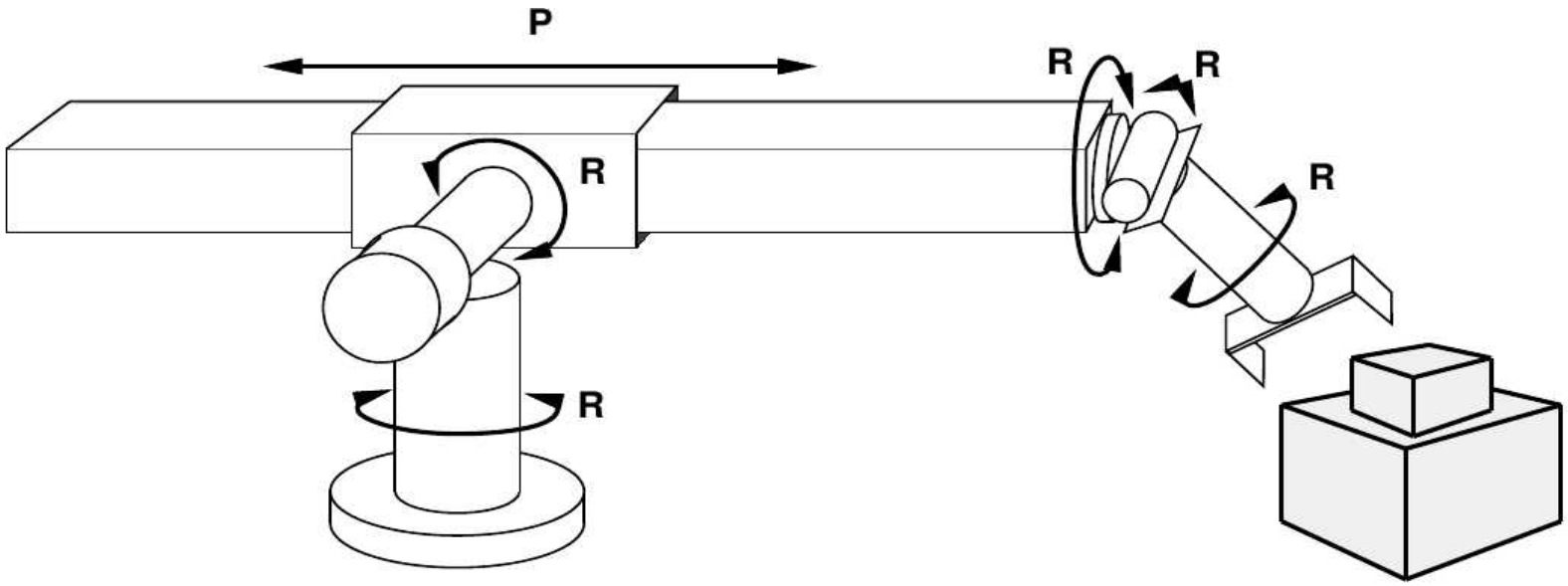
Example: The 8-queens problem



states??: one queen per column in the leftmost columns, none attacked
initial state??: no queens on the board
actions??: add a queen to any square in the leftmost empty column,
making sure that no queen is attacked
goal test??: 8 queens on the board, none attacked
path cost??: 1 per move

Using this formulation, we have only 2,057 sequences!

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 of the object

path cost??: time to execute

Tree search algorithms

Basic idea:

offline, simulated exploration of state space

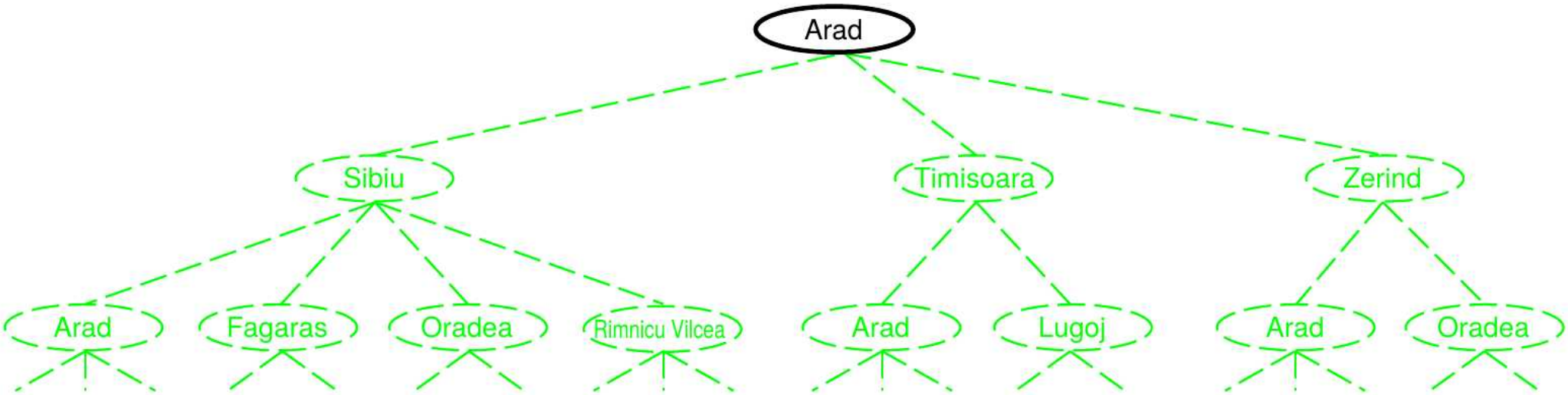
by generating successors of already-explored states

(a.k.a. expanding states)

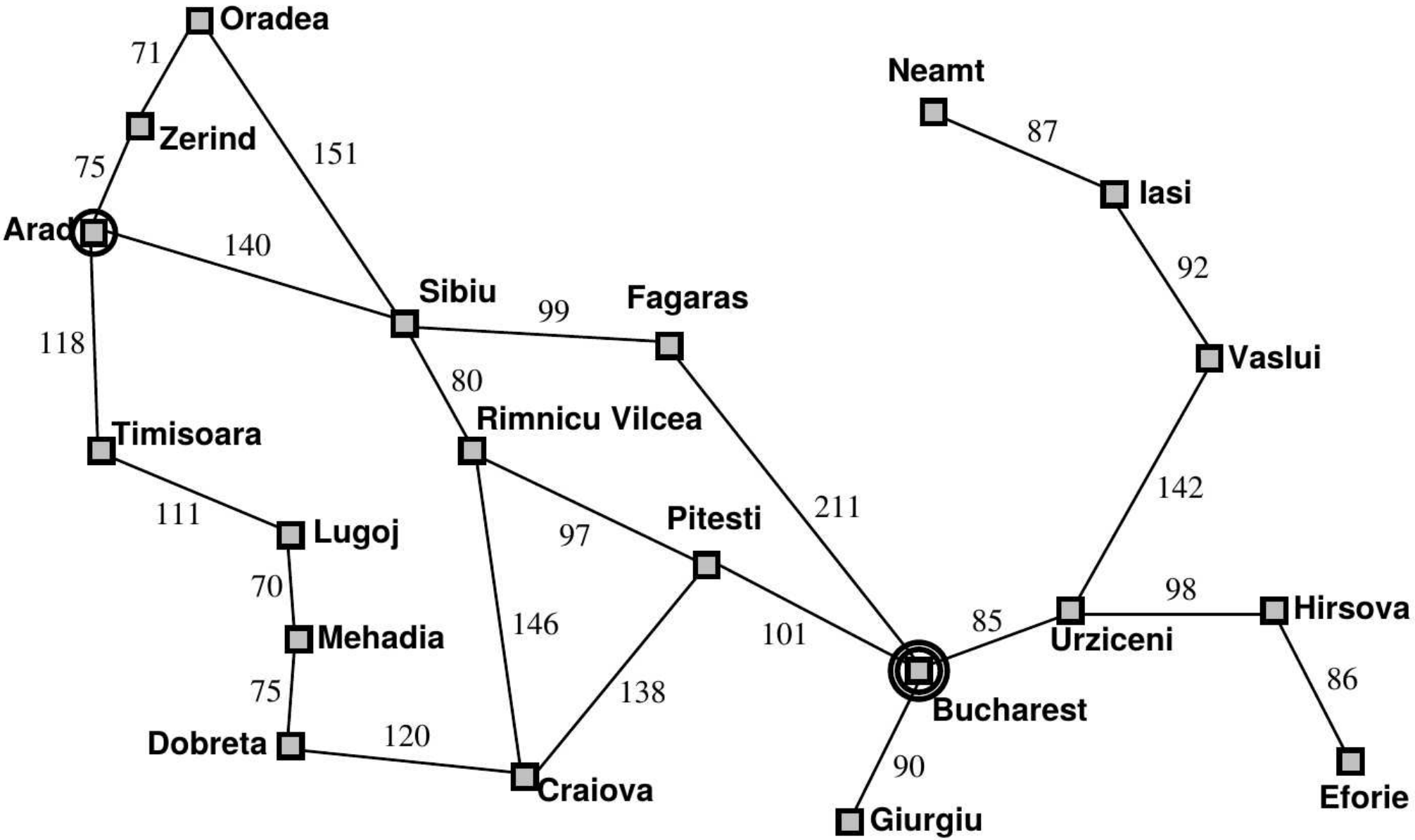
```
function TREE-SEARCH( problem) returns a solution, or failure

    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 and add the resulting nodes to the frontier
    end
```

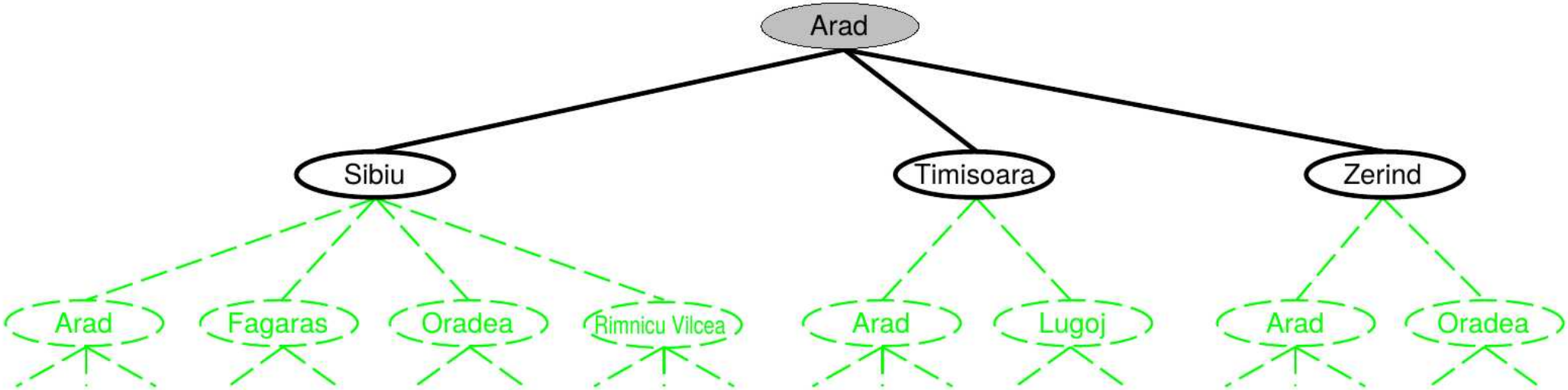
Tree search example



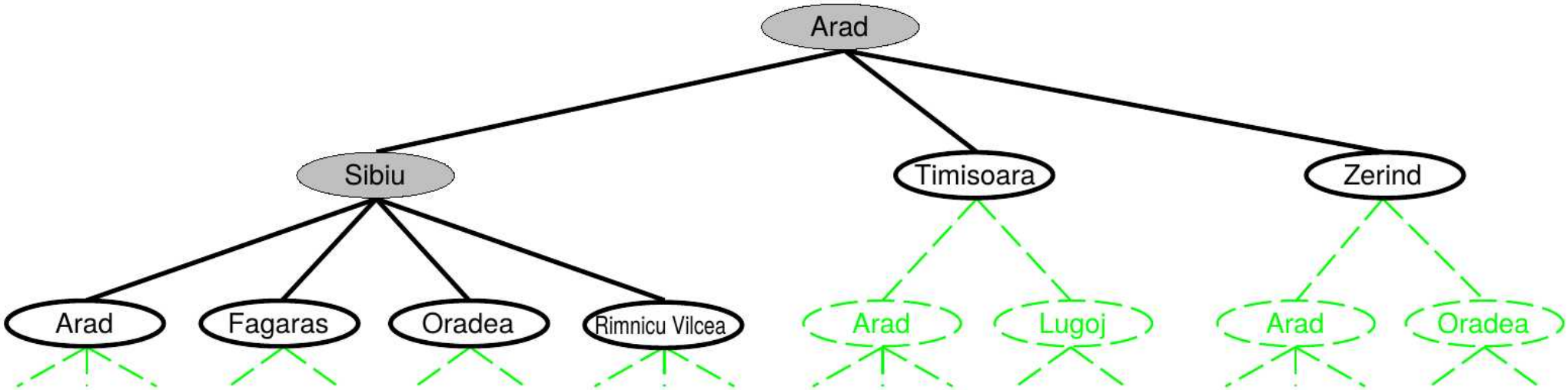
Example: Romania



Tree search example



Tree search example



Note: Arad is one of the expanded nodes!

This corresponds to going to Sibiu and then returning to Arad.

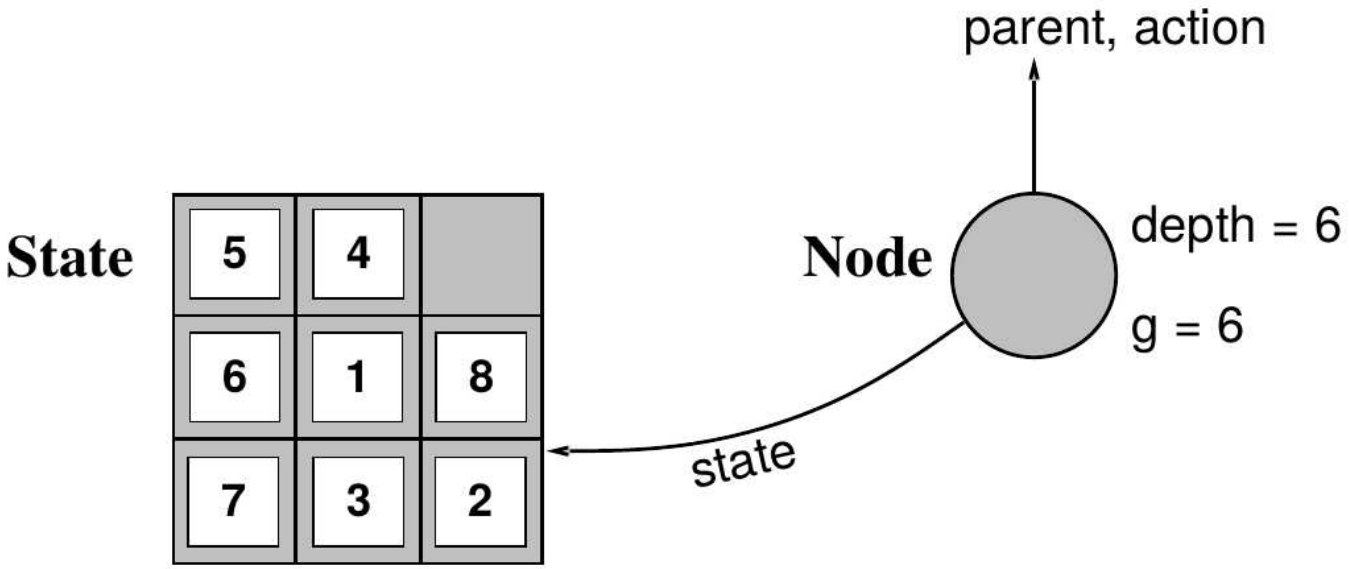
Implementation: states vs. nodes

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree

– includes the state, parent, children, depth, and the path cost $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

Implementation: general tree search

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

---

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(STATE[node], action, result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

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?
- optimality—does it always find a least-cost solution?
- time complexity—number of nodes generated/expanded
- space complexity—maximum number of nodes in memory

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 ∞)

Uninformed 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

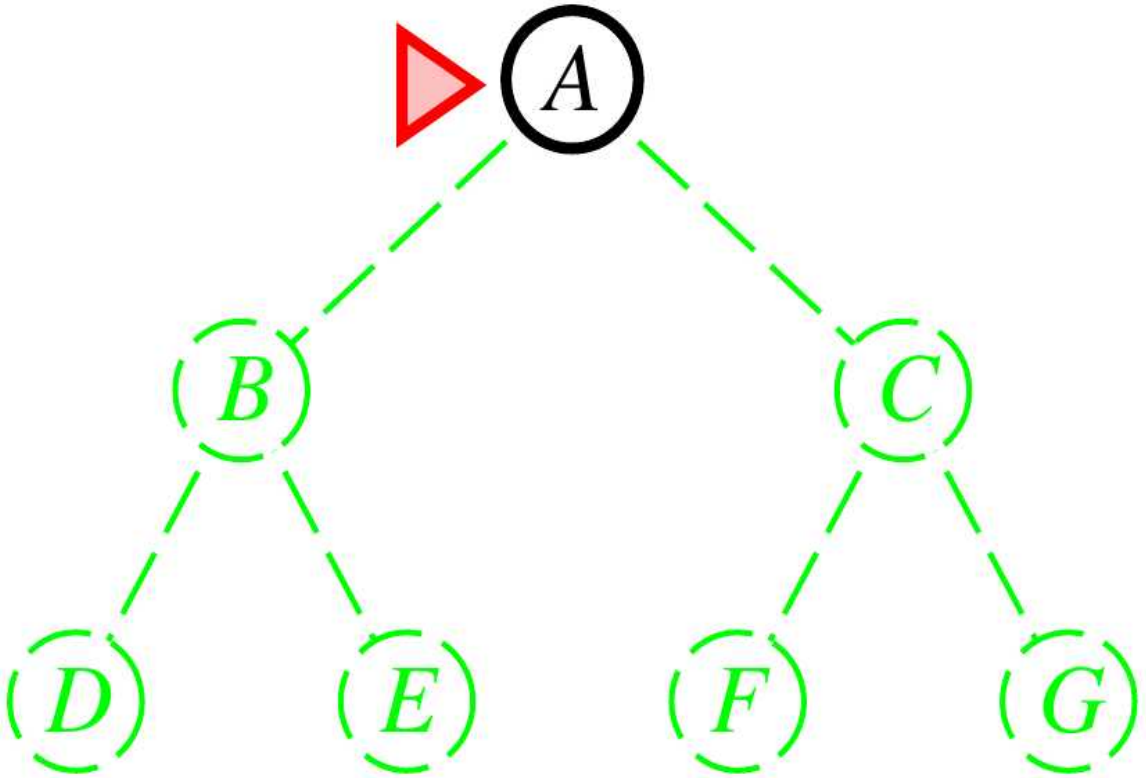
Expand the shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

After initialization

frontier = [A]



Breadth-first search

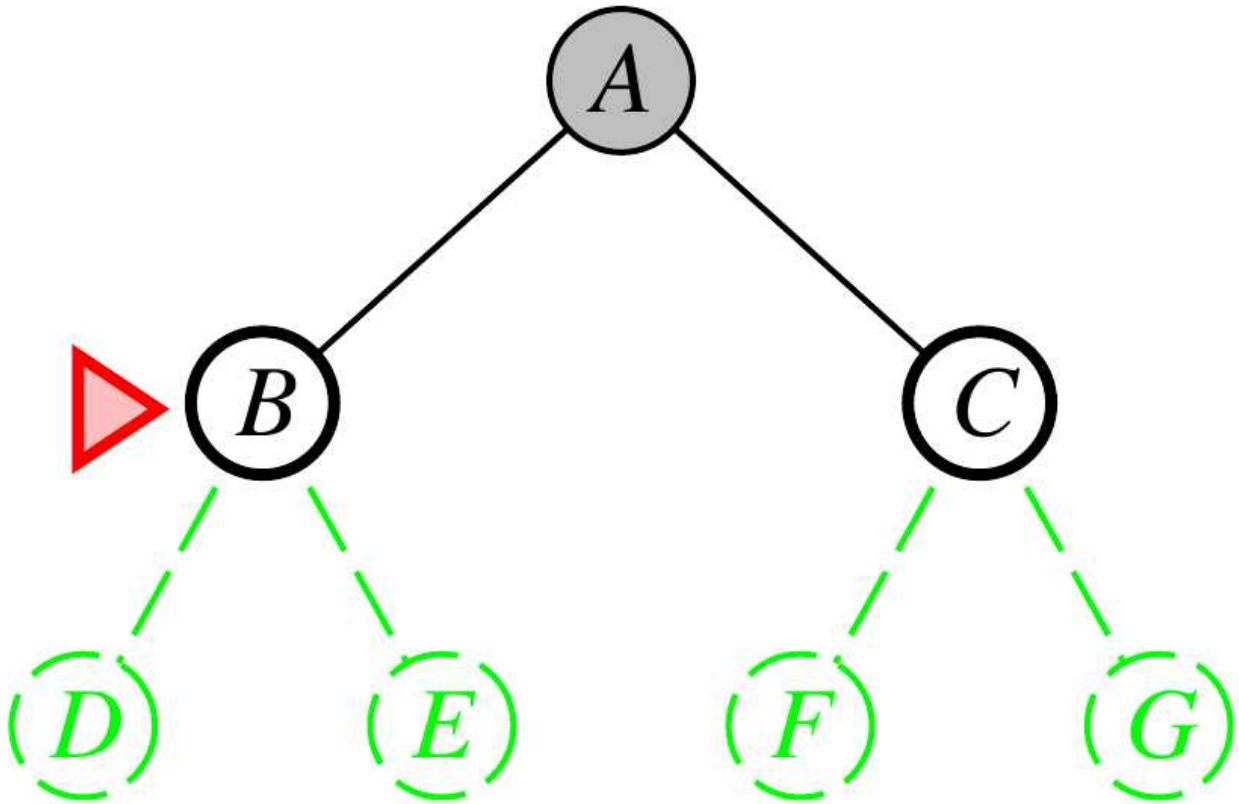
Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

After expanding A,

frontier = [B,C]



Breadth-first search

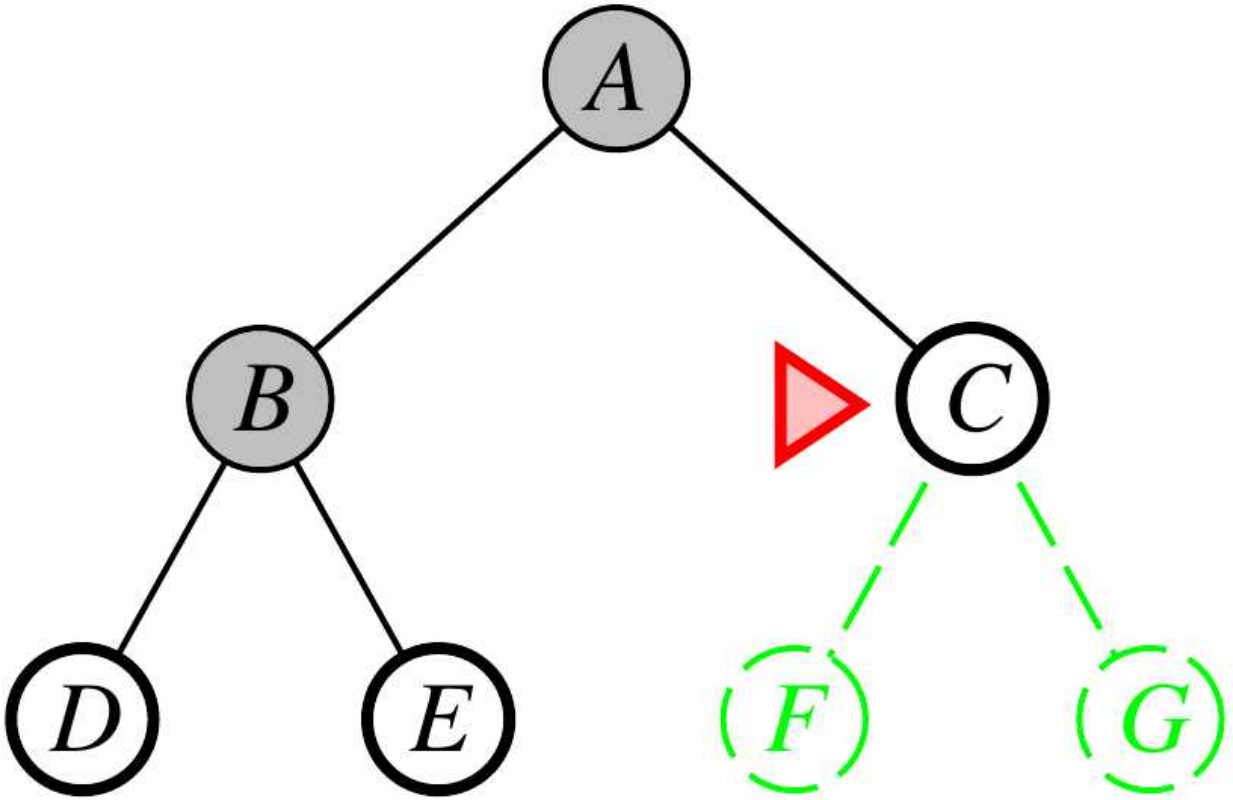
Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

After expanding B,

frontier = [C,D,E]



Breadth-first search

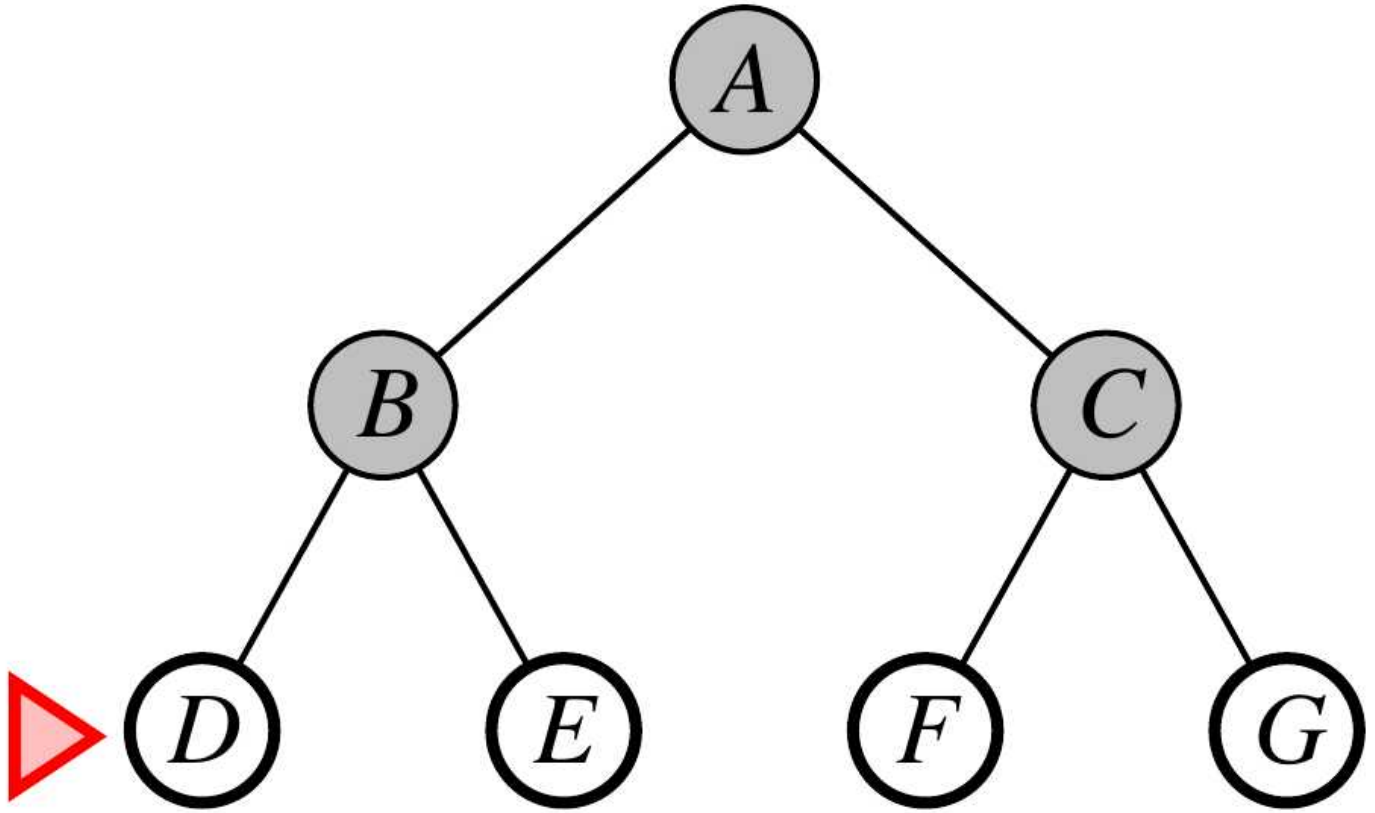
Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

After expanding C,

frontier = [D,E,F,G]



Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d$ $O(b^d)$,
i.e., exponential in d

Space?? $O(b^d)$ (keeps every node in memory)

Optimal?? Yes, if step cost $= 1$
Not optimal in general

Space is the big problem:
it can easily generate 1M nodes/second
so after 24hrs it has used 86,000GB
(and then it has only reached depth 9 in the search tree)

Uniform-cost search

Expand the cheapest unexpanded node

Implementation:

$frontier$ = priority queue ordered by path cost $g(n)$

Equivalent to breadth-first search, if all step costs are equal

Complete?? Yes, if step cost $\geq \epsilon > 0$

Time?? # of nodes with $g(n) \leq C^*$, i.e., $O(b^{\lceil C^*/\epsilon \rceil})$

where C^* is the cost of the optimal solution

and ϵ is the minimal step cost

Space?? Same as time

Optimal?? Yes—nodes are expanded in increasing order of $g(n)$

Depth-first search

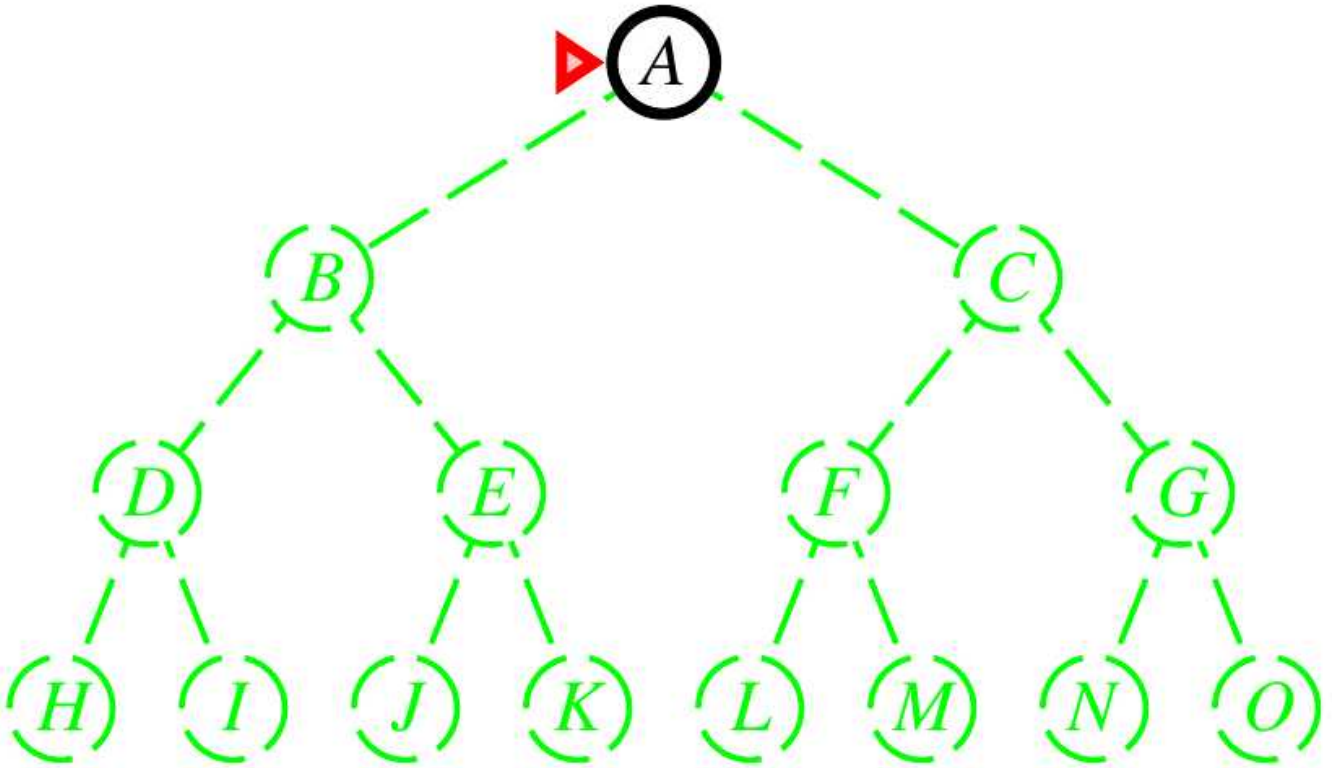
Expand the deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

After initialization,

frontier = [A]



Depth-first search

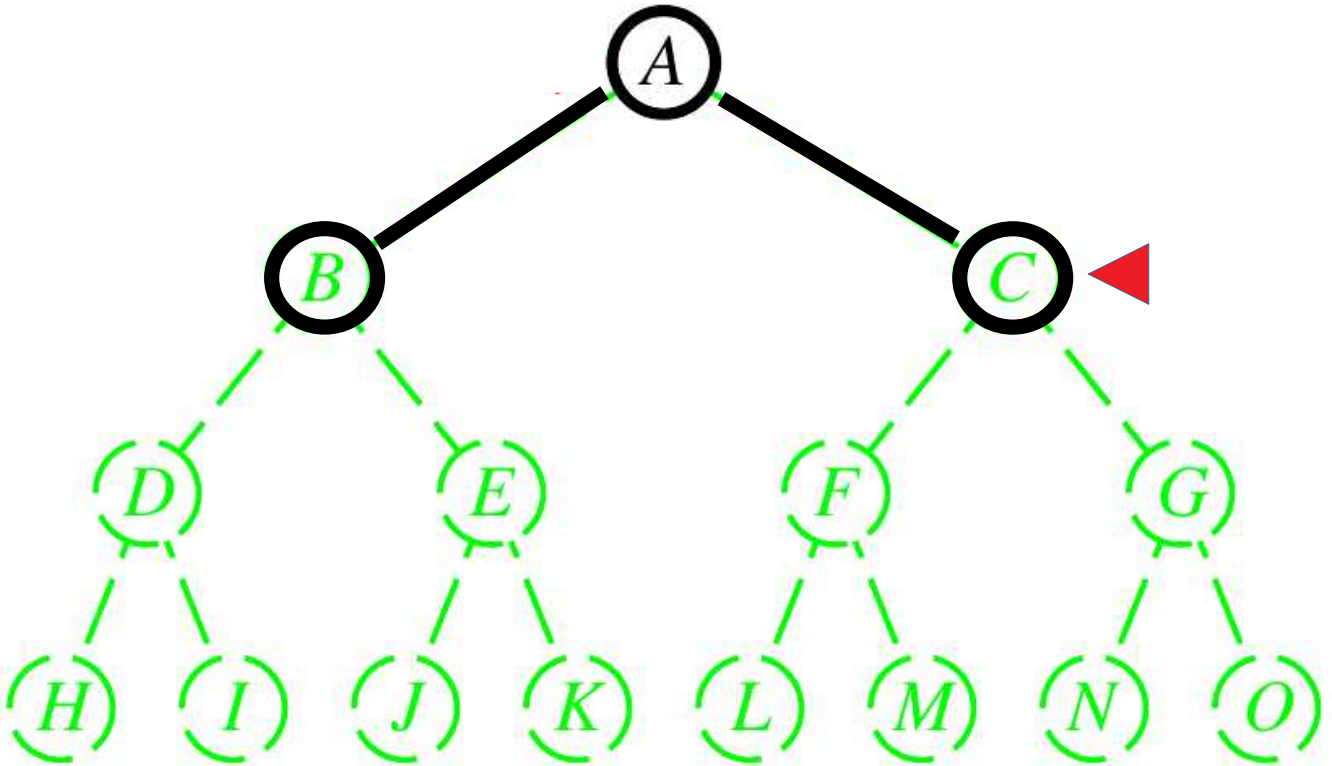
Expand the deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

After expanding A,

frontier = [B,C]



Depth-first search

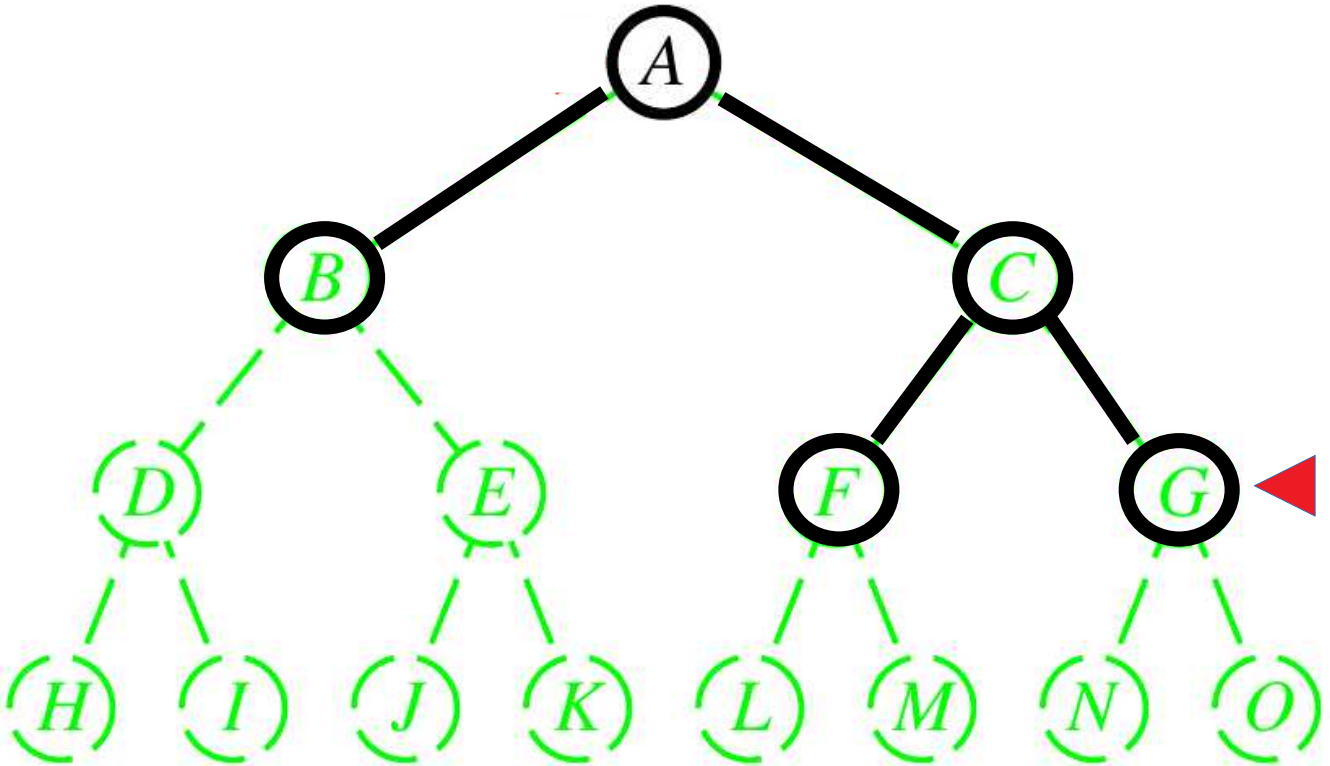
Expand the deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

After expanding C,

frontier = [B,F,G]



Depth-first search

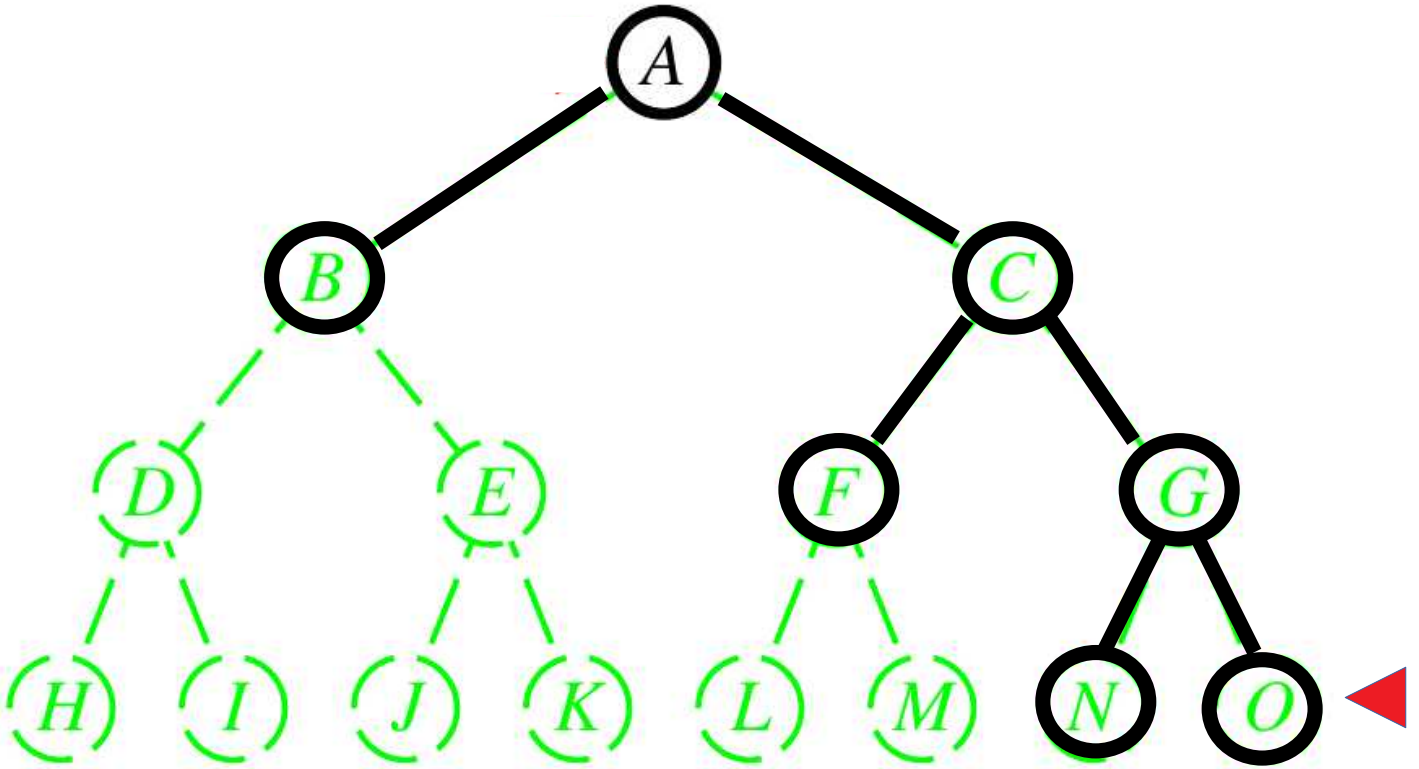
Expand the deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

After expanding G,

frontier = [B,F,N,O]



Depth-first search

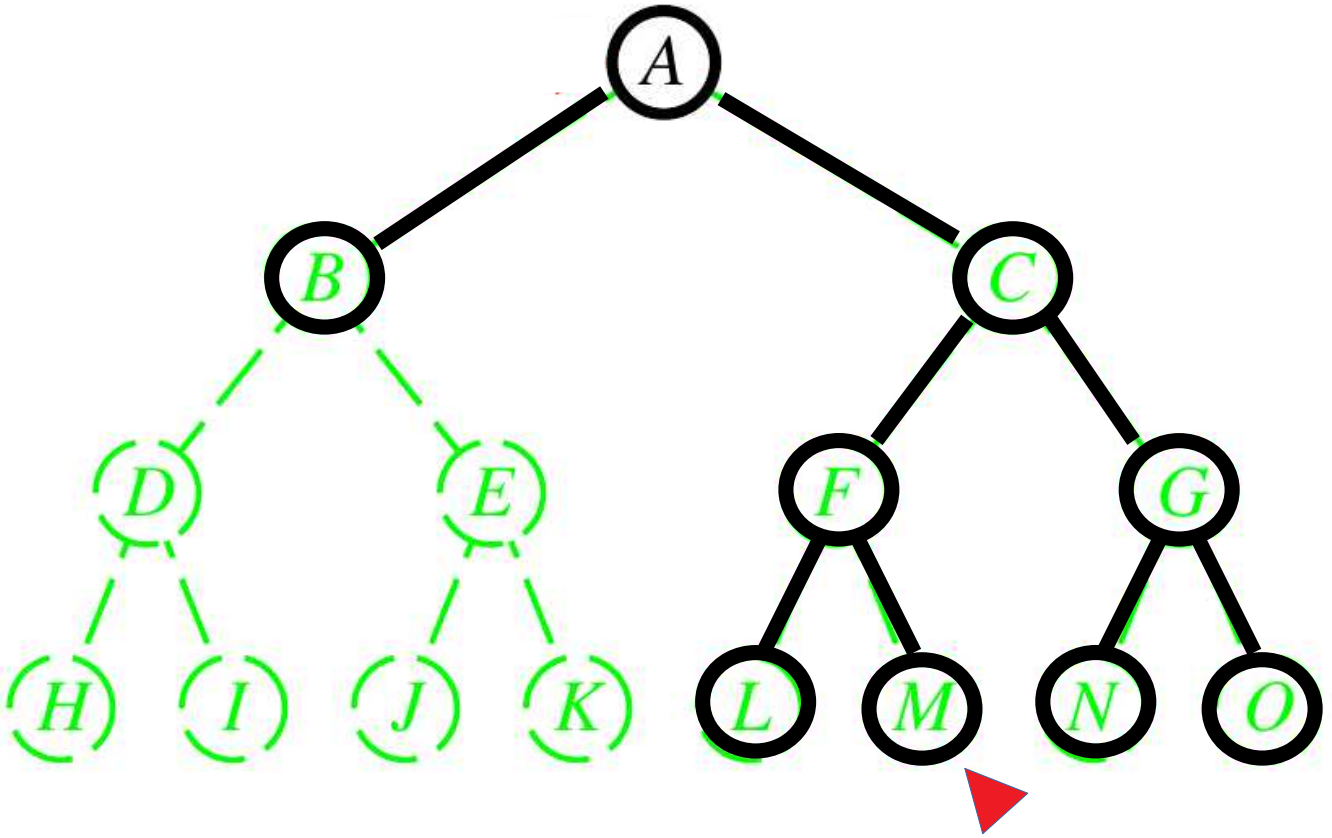
Expand the deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

After processing O, N and expanding F,

frontier = [B,L,M]



Depth-first search

Expand the deepest unexpanded node

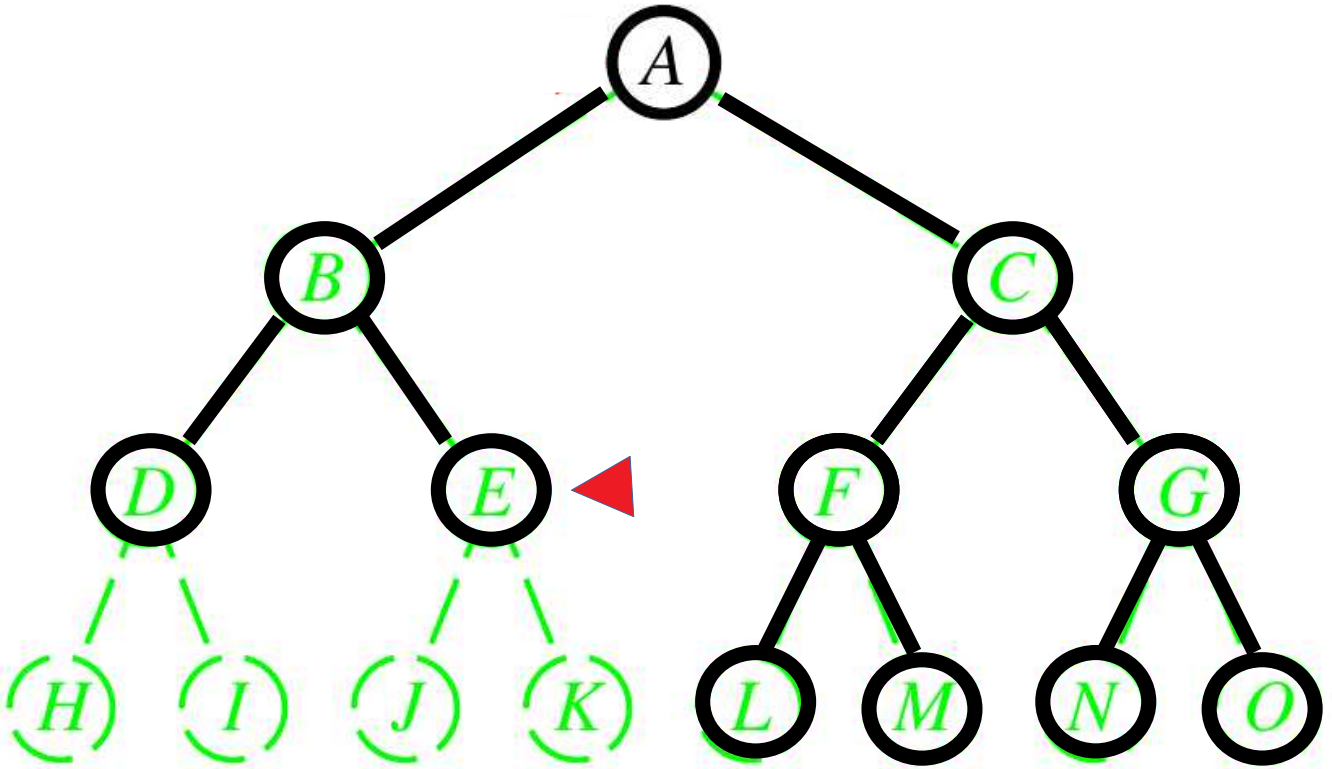
Implementation:

frontier = LIFO queue, i.e., put successors at front

After processing M, L and expanding B,

frontier = [D,E]

The last visited nodes will be K,J,I and H



Properties of depth-first search

Complete?? No: it fails in infinite-depth spaces

it also fails in finite spaces with loops

but if we modify the search to avoid repeated states

$\Rightarrow$ complete in finite spaces (even with loops)

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, it may be much faster than breadth-first

Space?? $O(bm)$: i.e., linear space!

Optimal?? No

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 soln/failure/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/failure/cutoff
    if GOAL-TEST(problem, STATE[node]) then return node
    else if limit = 0 then return cutoff
    else
        cutoff-occurred? ← false
        for each action in ACTIONS(STATE[node], problem) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit − 1)
            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

Successive depth-limited searches, with higher and higher depth limits, until a goal is found.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns solution/failure
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Note: This means that shallow nodes will be recalculated several times!

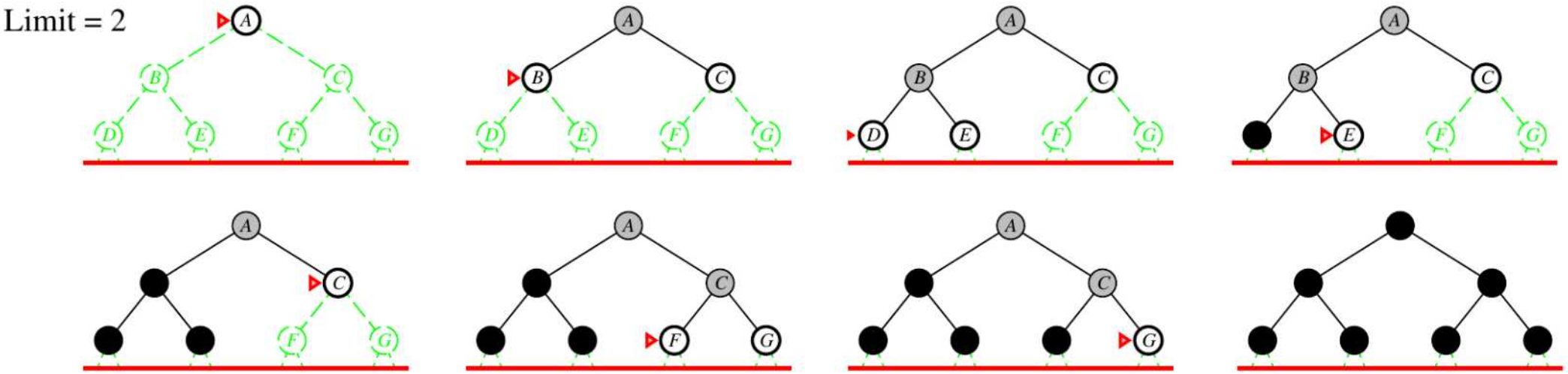
Iterative deepening search *L=0*

Limit = 0

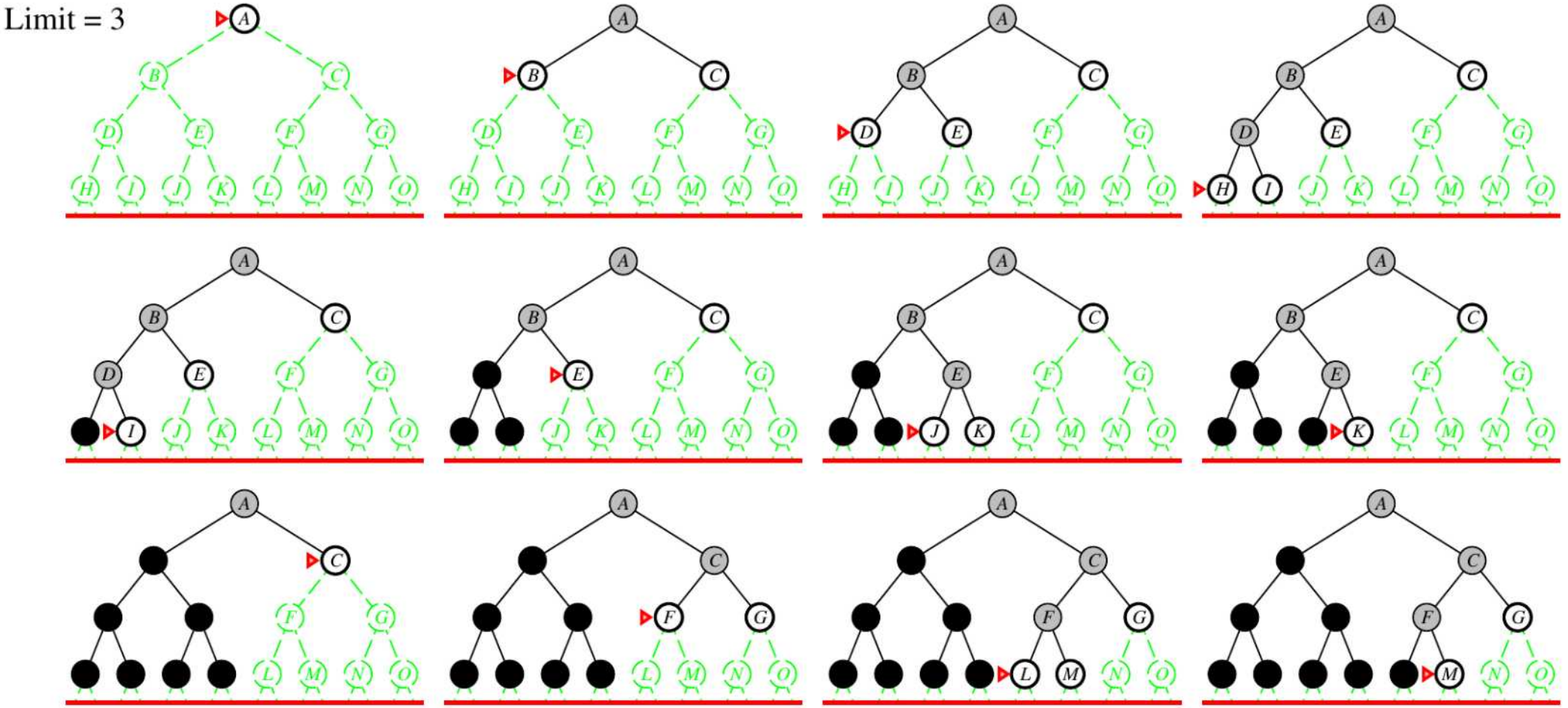
Iterative deepening search *L=1*

Limit = 1

Iterative deepening search *L=2*



Iterative deepening search *L=3*



Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
it can be modified to explore a uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$:

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$
$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

Note: IDS recalculates shallow nodes several times,
but this doesn't have a big effect compared to BFS!

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes, if $\epsilon > 0$	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^d	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

*if all step costs are identical

b = the branching factor

d = the depth of the shallowest solution

m = the maximum depth of the tree

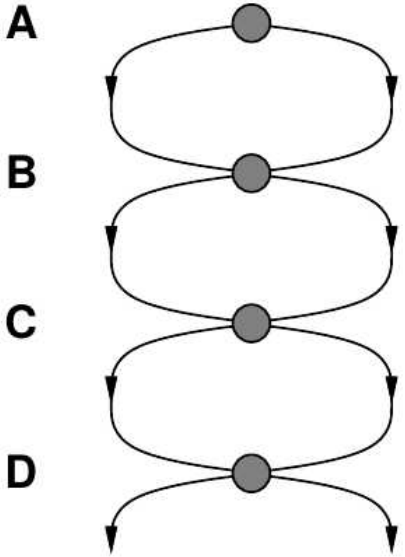
l = the depth limit

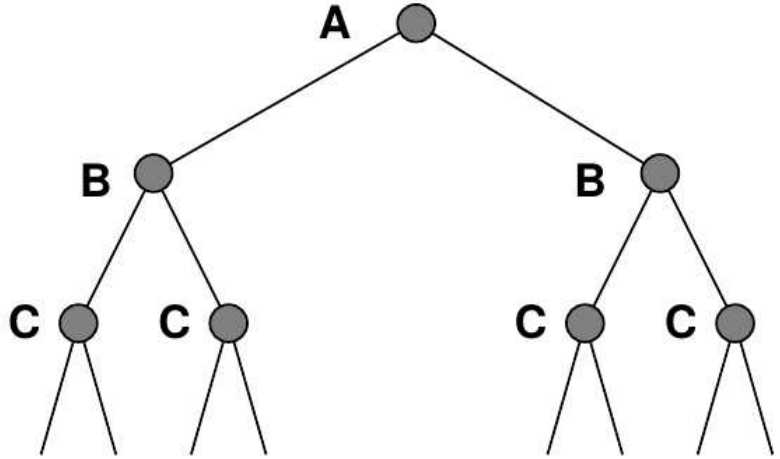
ϵ = the smallest step cost

C^* = the cost of the optimal solution

Repeated states

Failure to detect repeated states can turn a linear problem exponential!





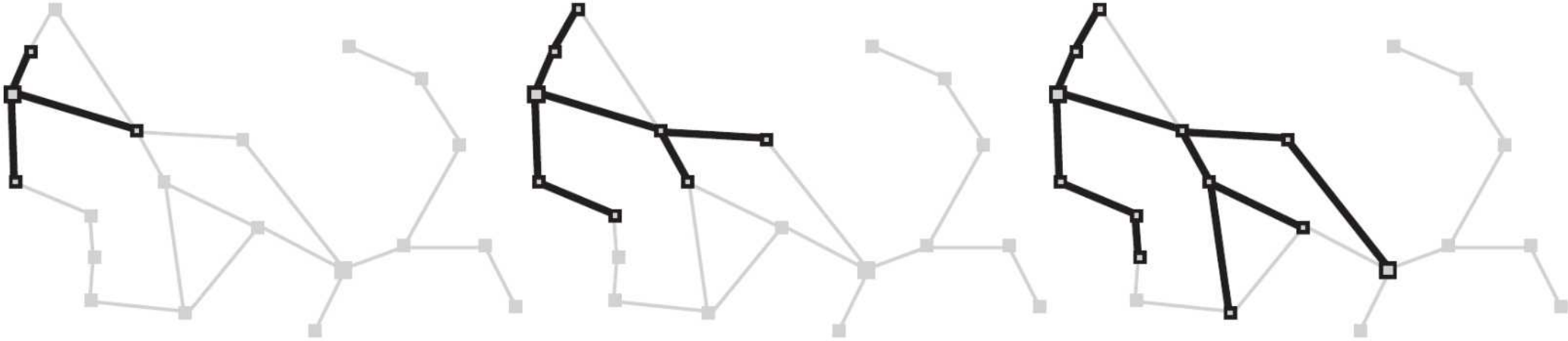
Solution: Use graph search instead of tree search!

Graph search

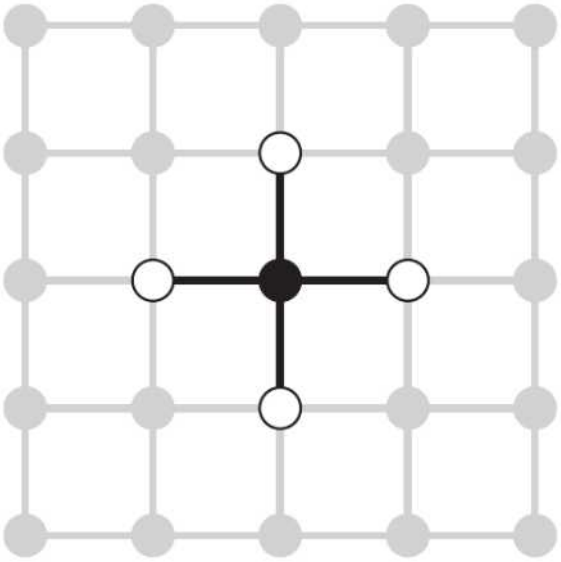
We augment the tree search algorithm with a set *explored*, which remembers every expanded node

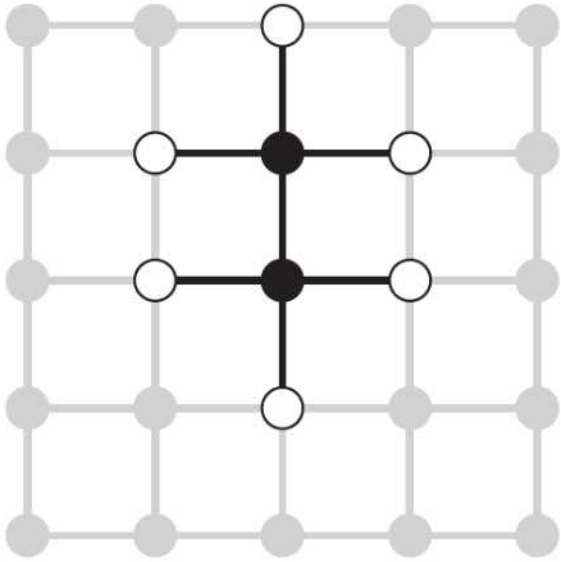
```
294
295 def graph_search(problem, frontier):
296     """
297     Search through the successors of a problem to find a goal.
298     The argument frontier should be an empty queue.
299     If two paths reach a state, only use the first one. [Fig. 3.7]
300     Return
301         the node of the first goal state found
302         or None is no goal state is found
303     """
304     assert isinstance(problem, Problem)
305     frontier.append(Node(problem.initial))
306     explored = set() # initial empty set of explored states
307     while frontier:
308         node = frontier.pop()
309         if problem.goal_test(node.state):
310             return node
311         explored.add(node.state)
312         # Python note: next line uses of a generator
313         frontier.extend(child for child in node.expand(problem)
314                         if child.state not in explored
315                         and child not in frontier)
316     return None
317
```

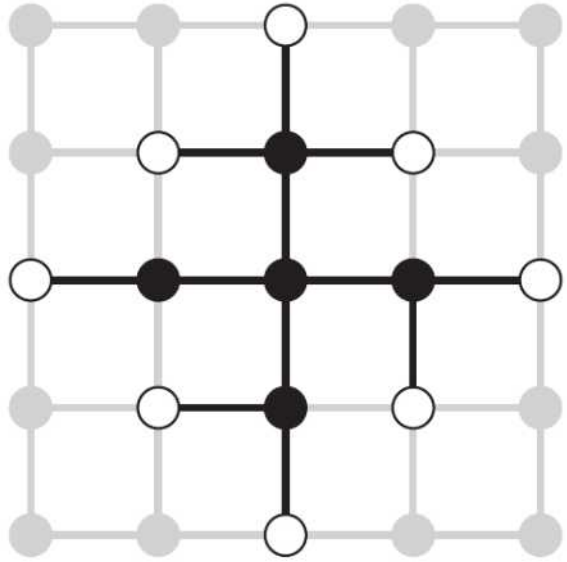
A sequence of search trees generated by a graph search on the Romania problem



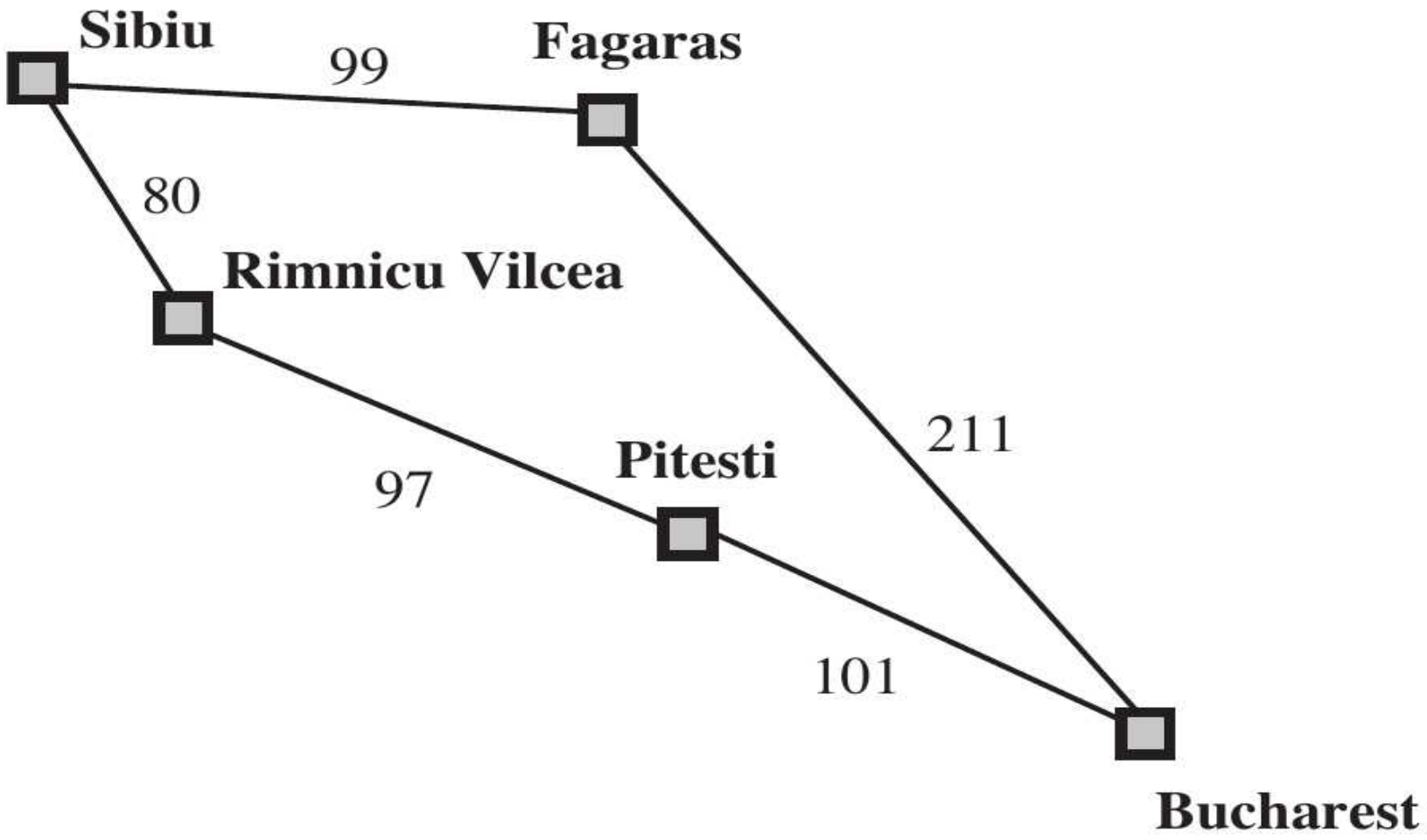
The separation property of GRAPH-SEARCH. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes)



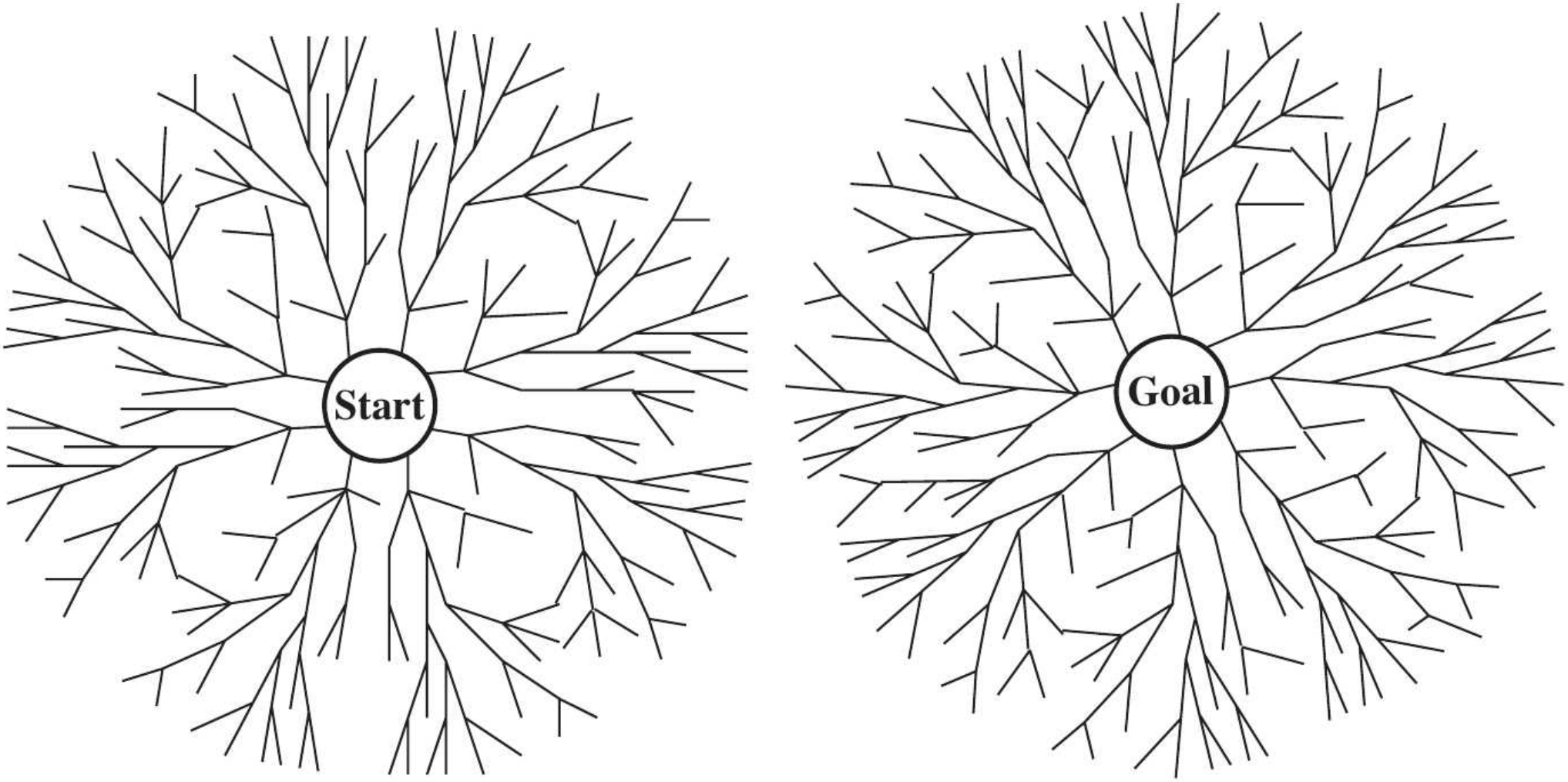




Uniform-cost search on a subgraph of the Romania map



A schematic view of a bidirectional search



Summary

Variety of uninformed search strategies:

- breadth-first search
- uniform-cost search
- depth-first search
- depth-limited search
- iterative deepening search

Iterative deepening search uses only linear space
and not much more time than other uninformed algorithms

Graph search can be exponentially more efficient than tree search