

Last update: September 4, 2008

# PROBLEM SOLVING AND SEARCH

## CMSC 421: CHAPTER 3

# Motivation and Outline

Lots of AI problem-solving requires trial-and-error search  
Chapter 3 describes some algorithms for this

- ◇ Problem-solving agents
- ◇ Problem types
- ◇ Problem formulation
- ◇ Example problems
- ◇ Basic search algorithms

# Problem-solving agents

**Offline** problem solving: develop the entire solution at the start, before you ever start to execute it

```
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

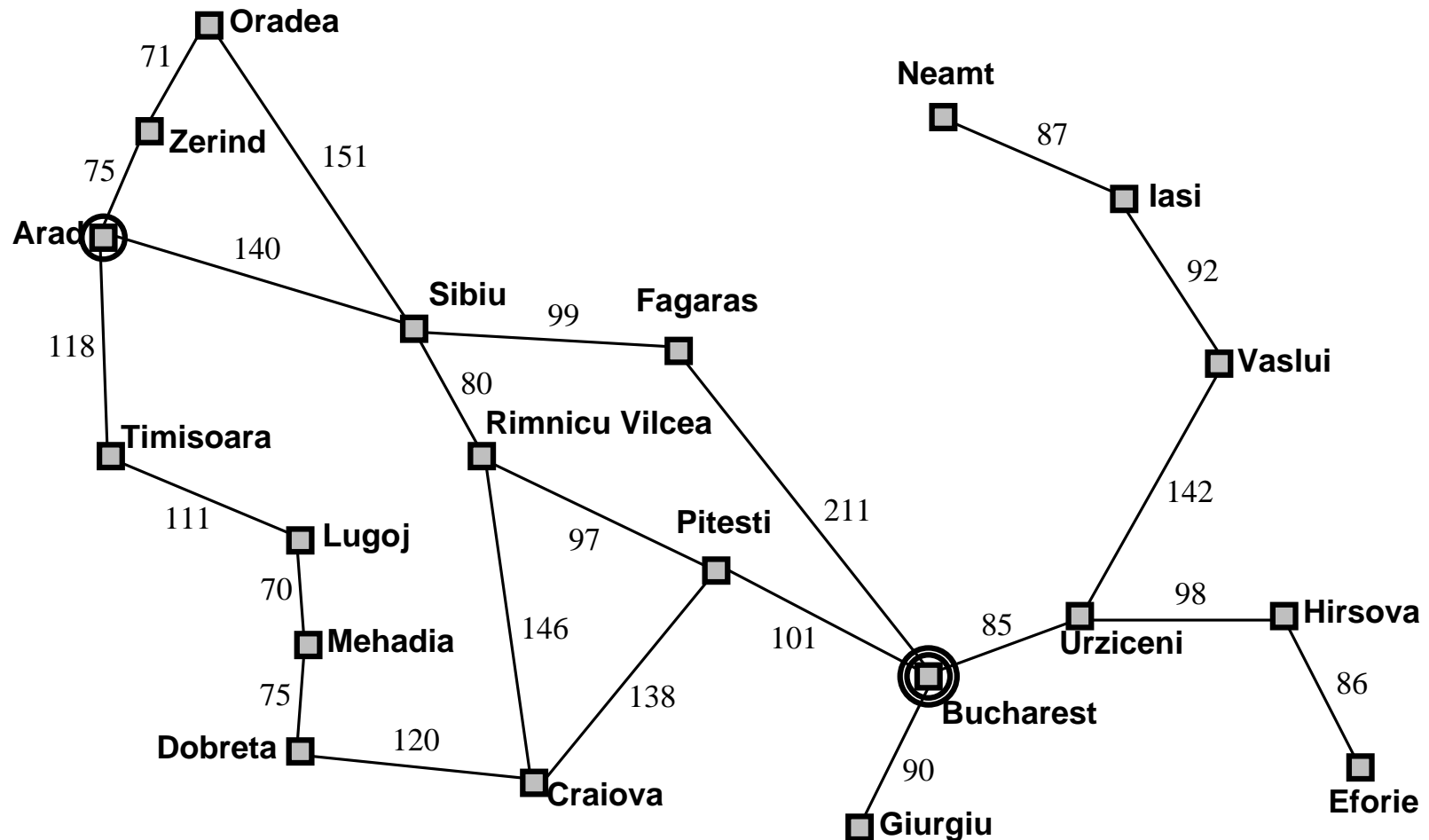
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

**Online** problem solving (not shown here): gather knowledge as you go.

# Example: Romania

Currently in Arad, Romania; flight leaves tomorrow from Bucharest

states = cities; actions = drive between cities; goal = be in Bucharest



# Problem types

**Deterministic, fully observable**  $\implies$  **classical search problem**

- ◇ agent knows exactly which state it starts in, what each action does
- ◇ no exogenous events (or else they're encoded into the actions' effects)
- ◇ solution is a sequence, can predict future states exactly

**Non-observable**

- ◇ Agent may have no idea where it is
- ◇ solution (if any) is a sequence that is **conformant**,  
i.e., guaranteed to work under all conditions

**Nondeterministic** and/or **partially observable**

- ◇ percepts provide new information about current state
- ◇ solution is a **contingent plan** or a **policy**
- ◇ often **interleave** search, execution

**Unknown state space**  $\implies$  **exploration problem**

# Example: vacuum world

Assume no exogenous events

e.g., rooms won't spontaneously get dirty again

**Deterministic, fully observable**, start in #5

Solution: [*Right, Suck*]

**Non-observable**, start in {1, 2, 3, 4, 5, 6, 7, 8}

Assume hitting the wall causes no harm

*Left* goes to {1, 3, 5, 7}

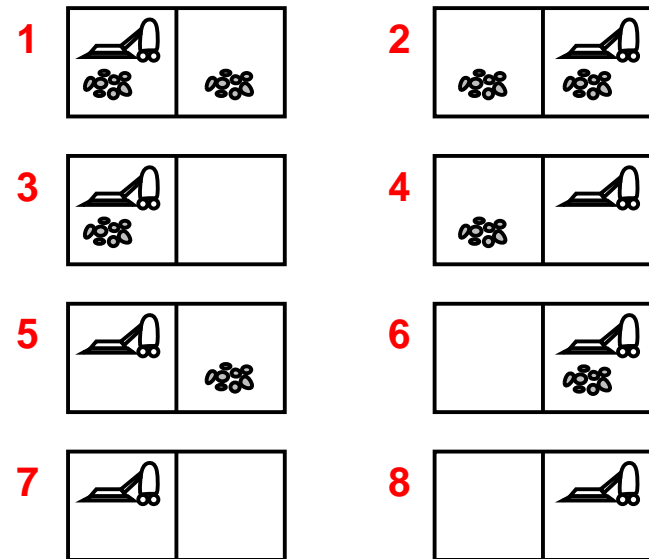
*Right* goes to {2, 4, 6, 8}

Solution: [*Right, Suck, Left, Suck*]

**Partially observable**, start in #5

Local sensing: dirt, location only.

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

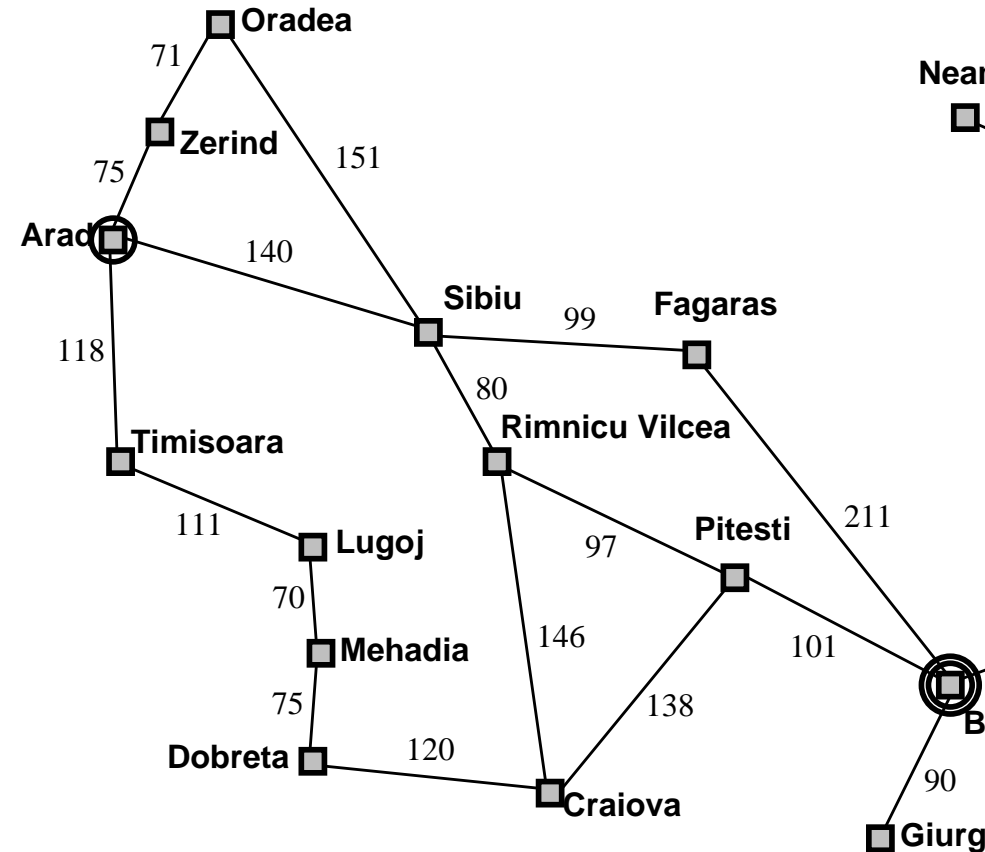


# Formulation of classical search problems

A **problem** consists of:

- ◇ **initial state**  $s_0$ , e.g., **at-Arad**
- ◇ **set of actions**, e.g.,  
 $A = \{\text{goto-Zerind}, \dots\}$
- ◇ **state-transition function**  $\gamma(s, a)$ , e.g.,  
 $\gamma(\text{at-Arad}, \text{goto-Zerind}) = \text{at-Zerind}, \dots$
- ◇ **goal test** can be **explicit**, e.g.,  
 $s = \text{"at Bucharest"}$   
or **implicit**, e.g.,  $\text{NoDirt}(s)$
- ◇ **path cost** (additive)  
e.g., sum of distances, number of actions executed, etc.  
 $c(s, a)$  is the **step cost**, assumed to be  $\geq 0$

**solution**: 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

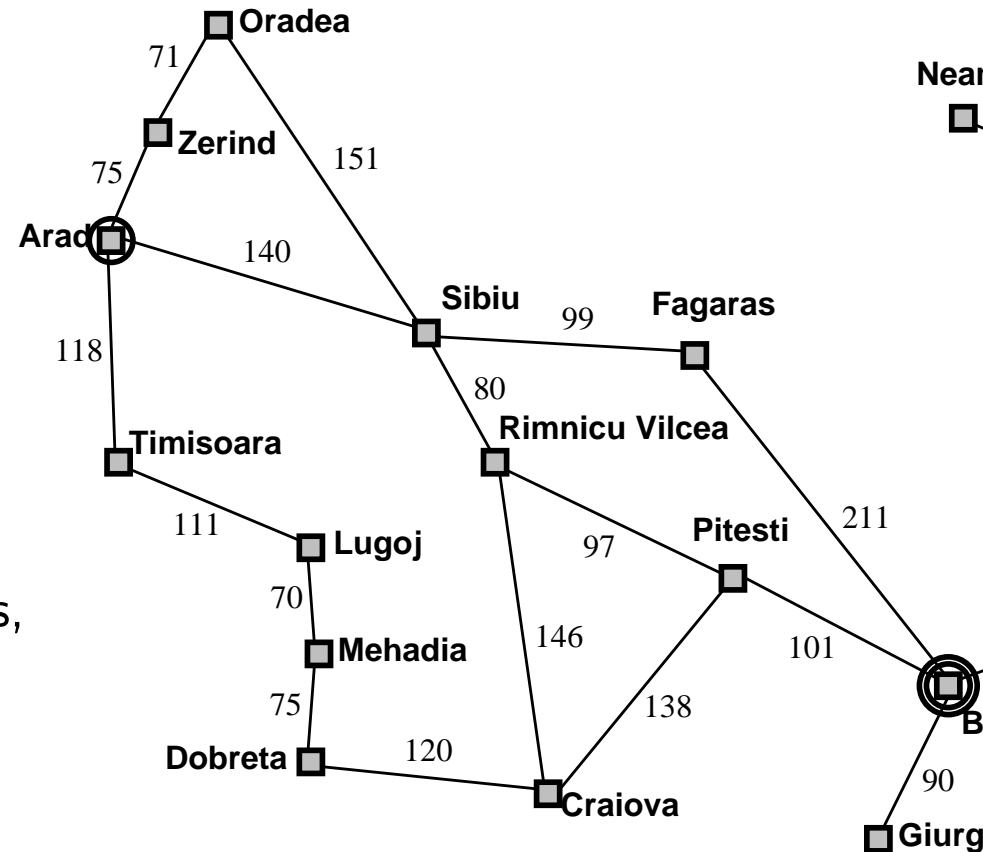
◇ **Abstract state** = set of real states

◇ **Abstract action** = complex combination of real actions

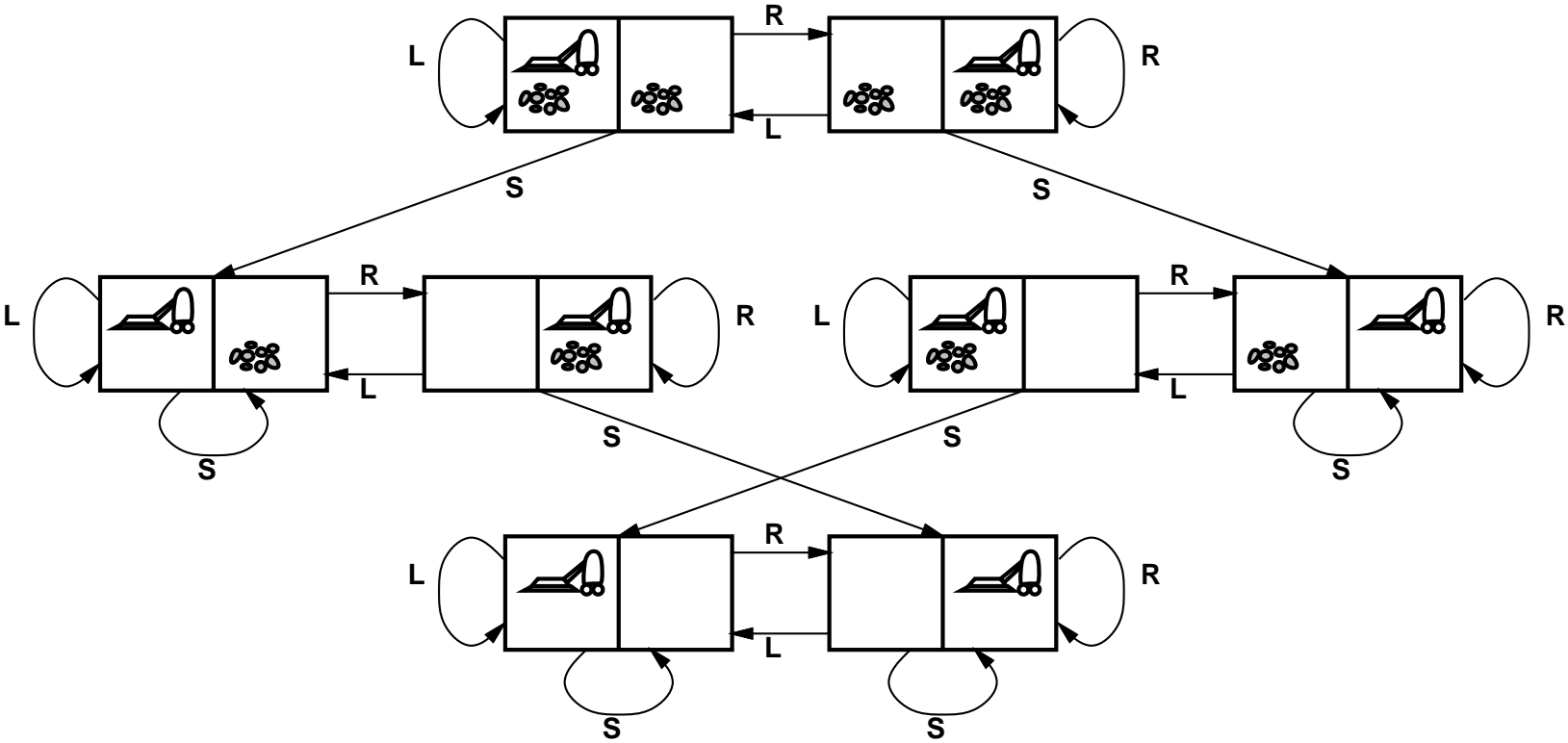
e.g., **goto-Zerind** may include possible routes, detours, rest stops, etc. For guaranteed realizability, it must get you to Zerind no matter where you are in Arad

◇ **Abstract solution** = sequence of abstract actions

represents a set of real paths that are solutions in the real world



# Example: vacuum world state space graph



- states: 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*)

## 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:** integer locations of tiles (ignore intermediate positions)

**actions:** move blank left, right, up, down (ignore unjamming etc.)

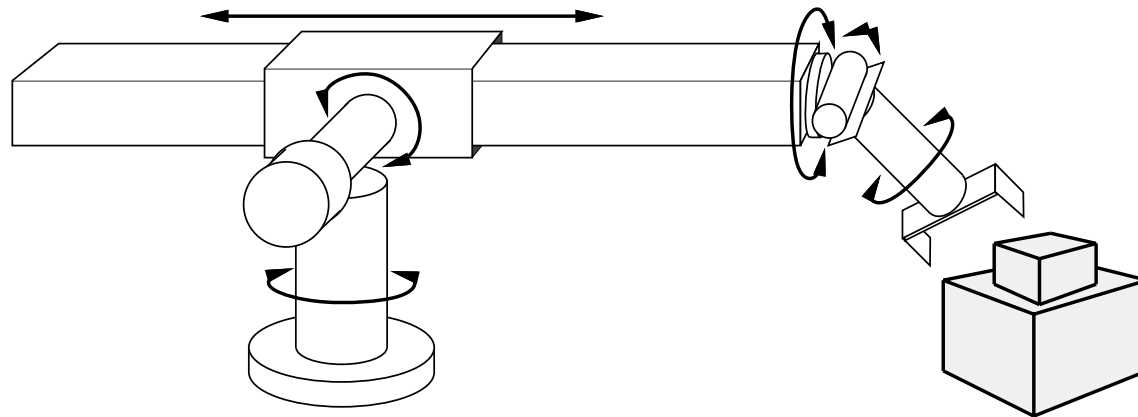
**goal test:** = goal state (given)

**path cost:** 1 per move

Finding optimal solution of  $n$ -Puzzle family is NP-hard.

Easier if we don't care whether the solution is optimal.

## 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

```
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
```

*node*: includes *state*  $s$ , *parent*, *children*, *depth*, *path cost*  $g(s)$

*expanding* a node: generating all of its children

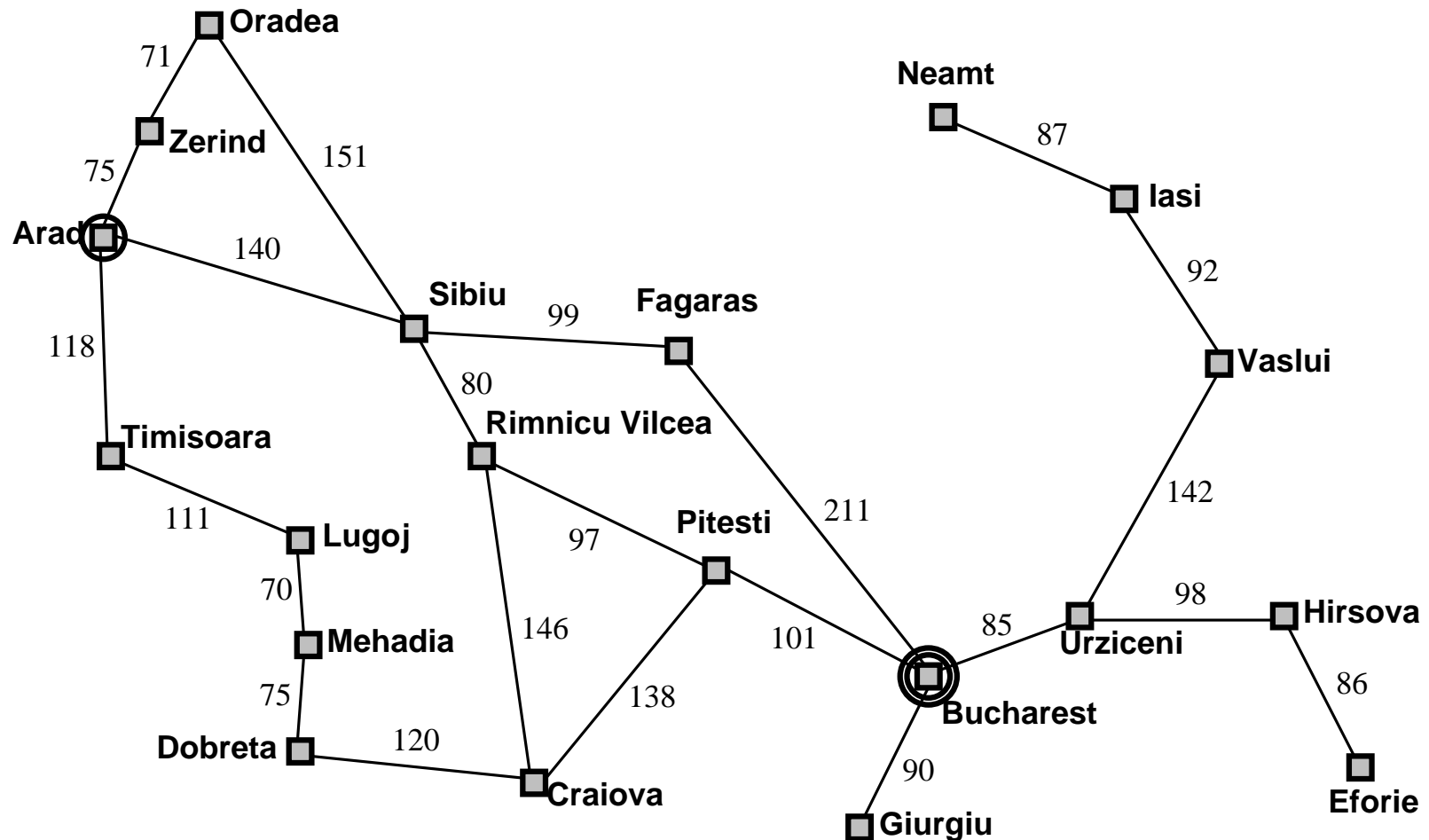
*fringe* or *frontier* = {all candidates for expansion}

= {all nodes that have been generated but not expanded}

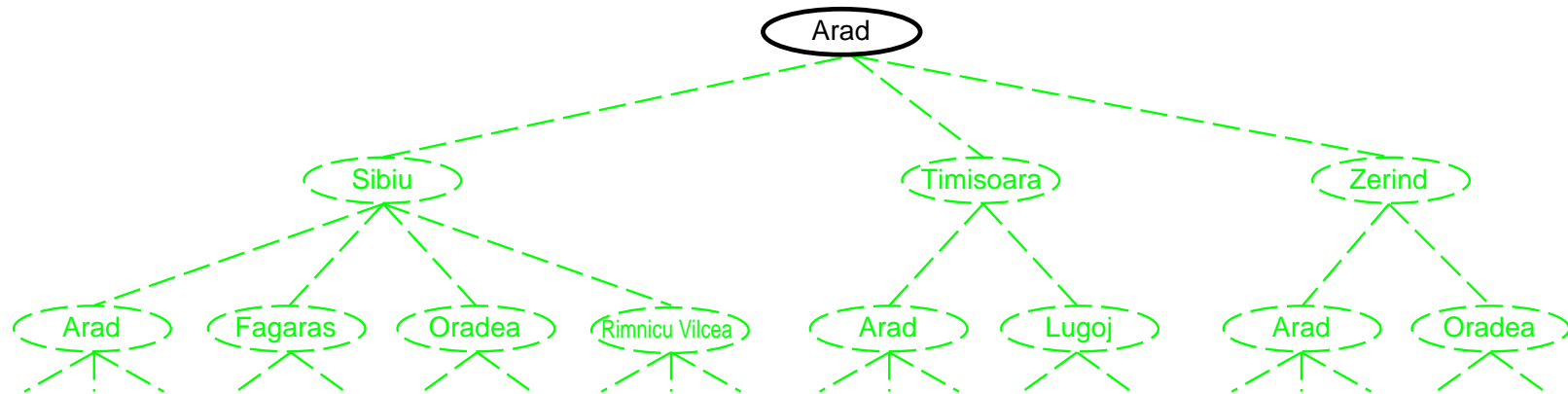
# Tree search example

Currently in Arad, Romania; flight leaves tomorrow from Bucharest

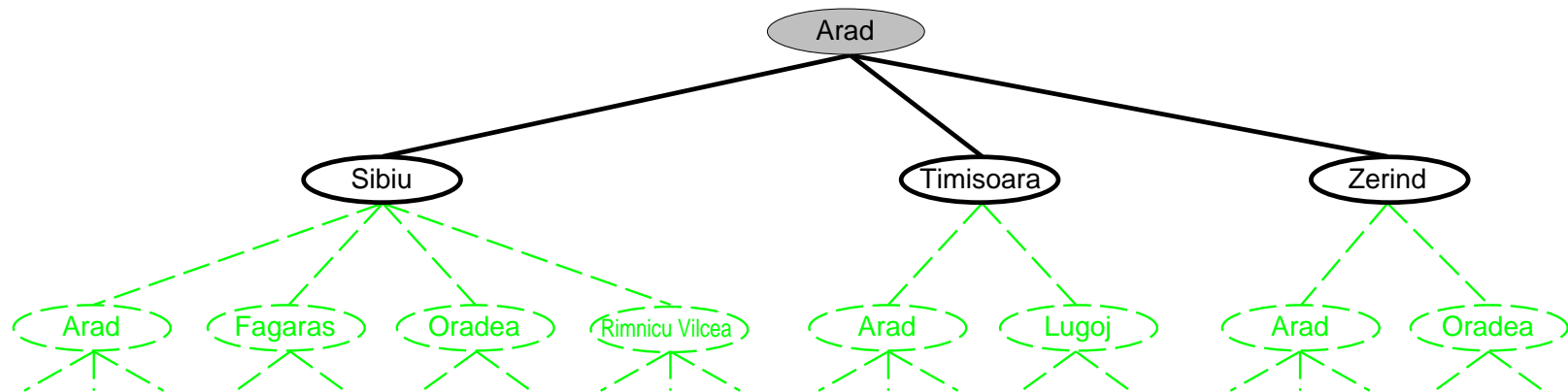
states = cities; actions = drive between cities; goal = be in Bucharest



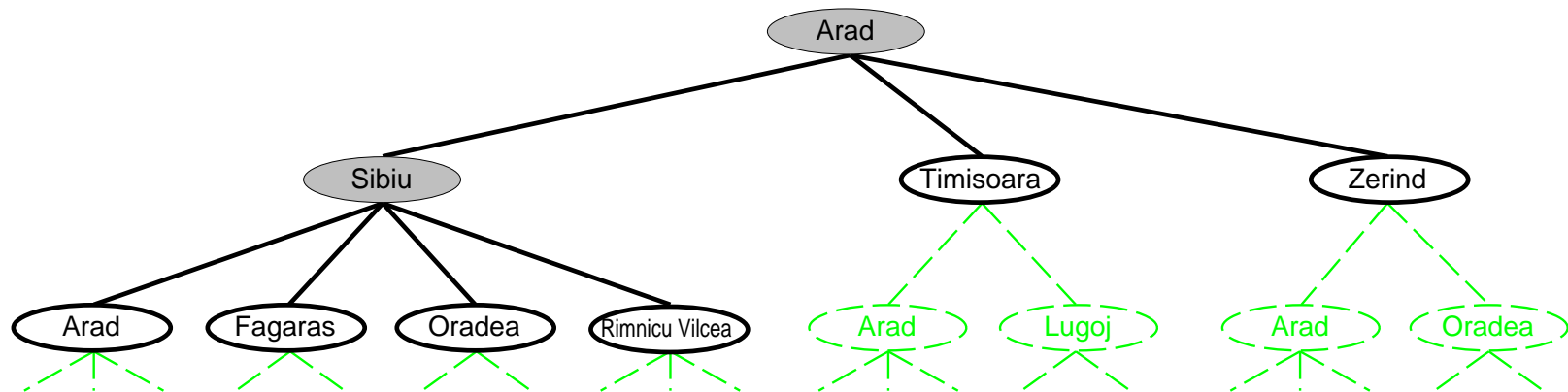
# Tree search example



# Tree search example

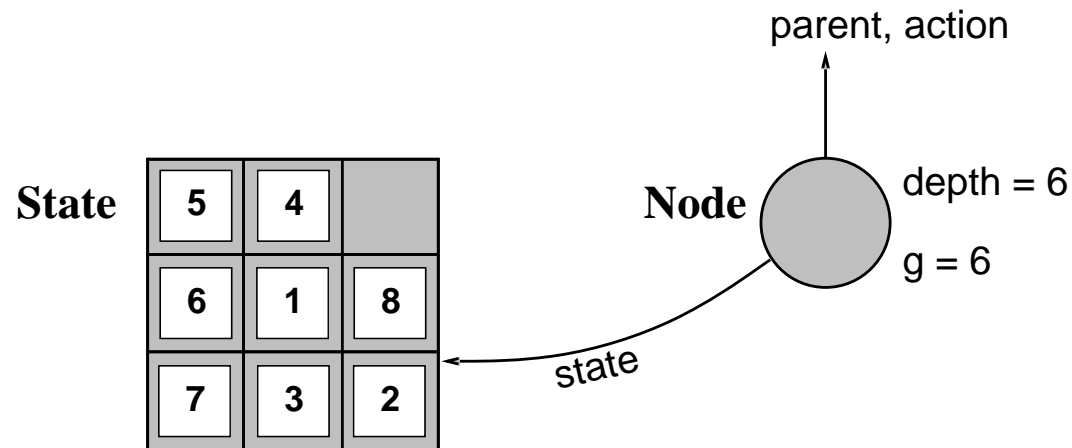


# Tree search example



## Implementation: states vs. nodes

- ◇ A **state** is a (representation of) a physical configuration
- ◇ A **node**  $x$  is a data structure that's part of a search tree. It includes **state**  $s$ , **parent**, **children** (if  $s$  has been expanded), **depth**, **path cost**  $g(x)$
- ◇ The states themselves don't have parents, children, depth, or path cost



- ◇ The EXPAND function creates new nodes:
  - uses the state-transition function  $\gamma$  to generate the states for  $x$ 's children:  $\{\gamma(s, a) : a \text{ is applicable to } s\}$
  - fills in the various fields

# Search strategies

A strategy is defined by picking the **order of node expansion**

Ways to evaluate a strategy:

**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 solutions**—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; we'll assume it's finite

$d$  = depth of the least-cost solution (or  $\infty$  if there's no solution)

$m$  = maximum depth of the state space (may be  $\infty$ )

# Uninformed search strategies

**Uninformed** strategies use only the information available in the problem definition

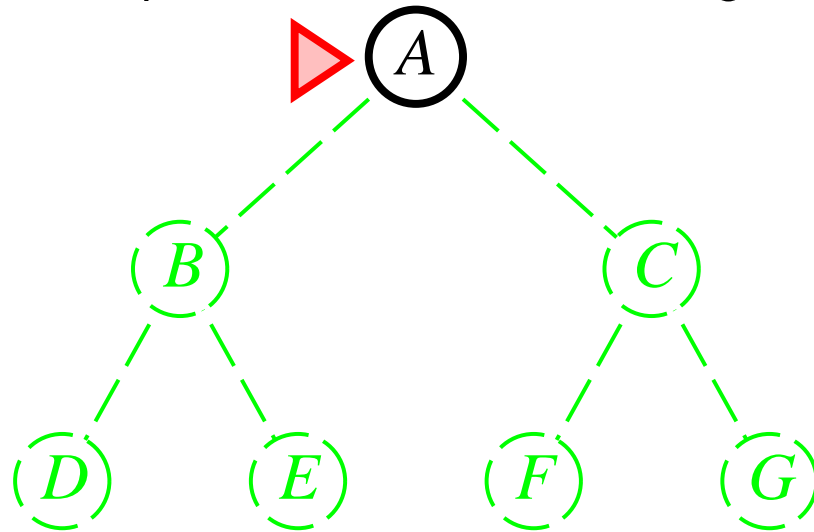
- ◇ Breadth-first search
- ◇ Depth-first search
- ◇ Depth-limited search
- ◇ Uniform-cost search
- ◇ Iterative deepening search

# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

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

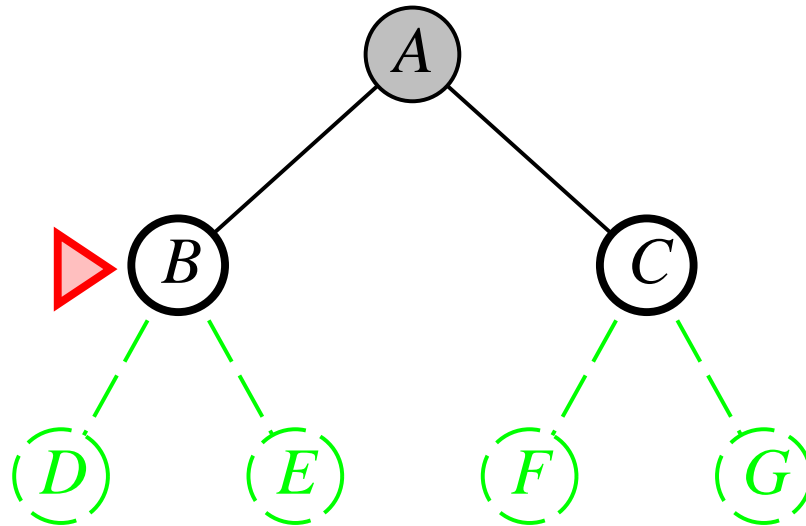


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

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

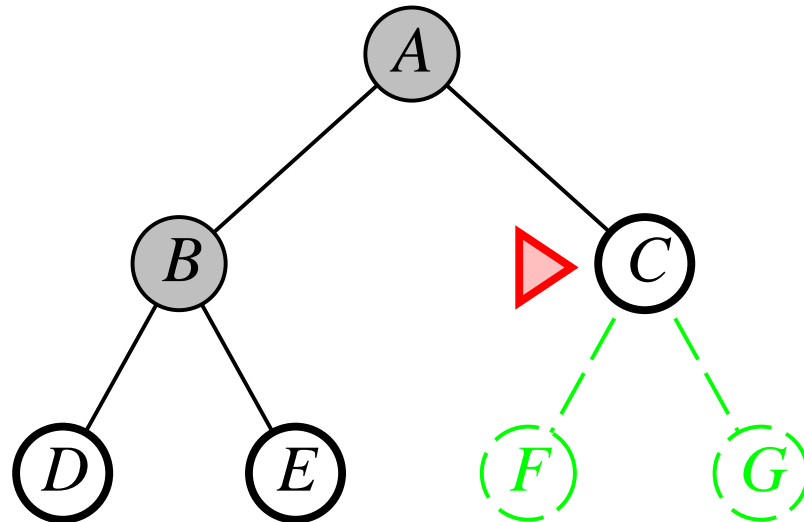


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

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

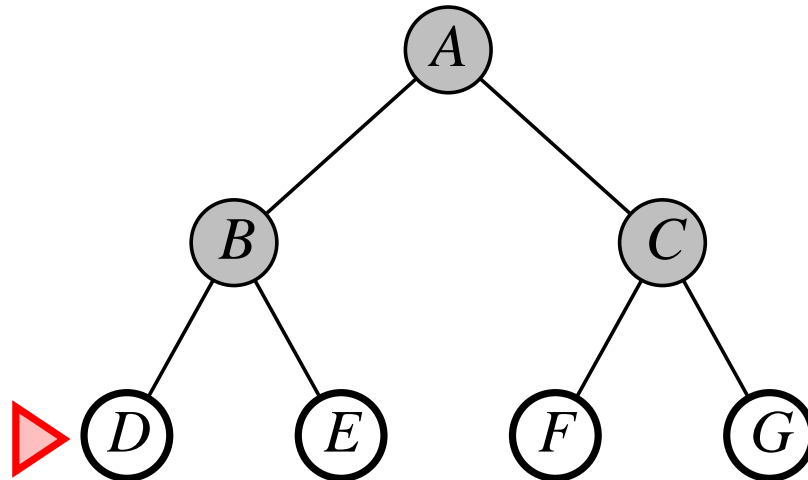


# Breadth-first search

Expand shallowest unexpanded node

## Implementation:

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



# Properties of breadth-first search

Complete?

---

$b$  = maximum branching factor of the search tree  
 $d$  = depth of the least-cost solution  
 $m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of breadth-first search

Complete? Yes

Time?

---

$b$  = maximum branching factor of the search tree  
 $d$  = depth of the least-cost solution  
 $m$  = maximum depth of the state space (may be  $\infty$ )

## Properties of breadth-first search

Complete? Yes

Time?  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^d)$ , i.e., exp. in  $d$

Space?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of breadth-first search

Complete? Yes

Time?  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^d)$ , i.e., exp. in  $d$

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

This is a big problem. If we run for 24 hours and generate nodes at 100MB/sec, the space requirement is 8.64 TB

Optimal solutions?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

## Properties of breadth-first search

Complete? Yes

Time?  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^d)$ , i.e., exp. in  $d$

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

This is a big problem. If we run for 24 hours and generate nodes at 100MB/sec, the space requirement is 8.64 TB

Optimal solutions? Yes if cost = 1 per step, but not in general

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Uniform-cost search

Expand least-cost unexpanded node

**Implementation:** *fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Uniform-cost search

Expand least-cost unexpanded node

**Implementation:** *fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

## Uniform-cost search

Expand least-cost unexpanded node

**Implementation:** *fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Uniform-cost search

Expand least-cost unexpanded node

**Implementation:** *fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal solutions?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Uniform-cost search

Expand least-cost unexpanded node

**Implementation:** *fringe* = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$   
where  $C^*$  is the cost of the optimal solution

Space? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal solutions? Yes

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

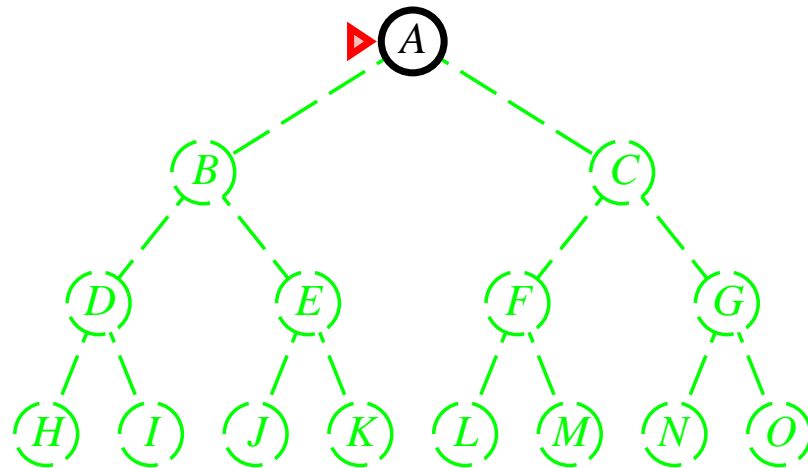
$m$  = maximum depth of the state space (may be  $\infty$ )

# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

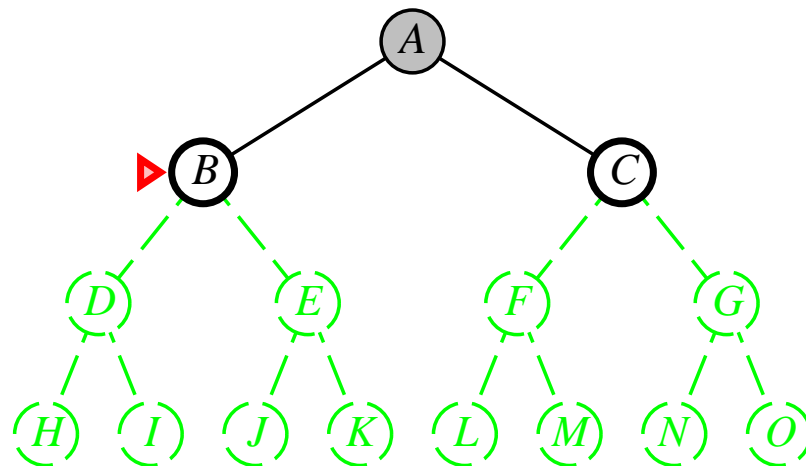


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front



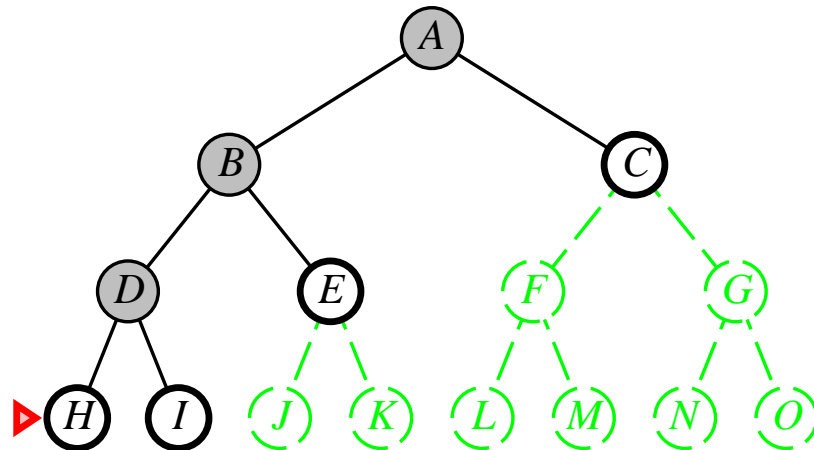


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

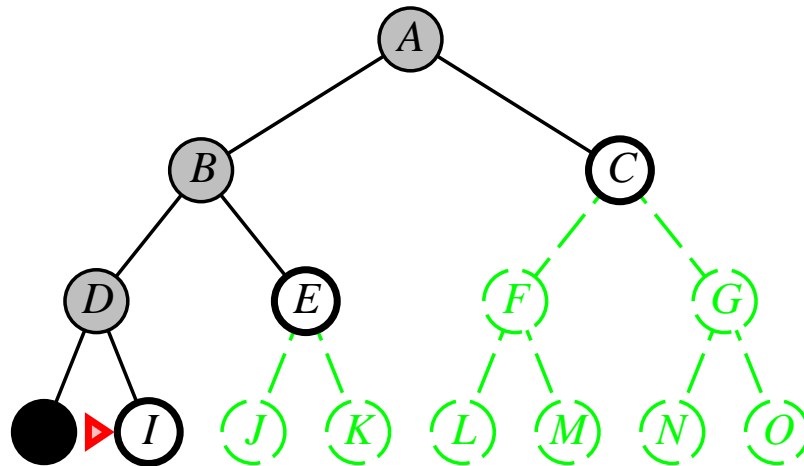


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

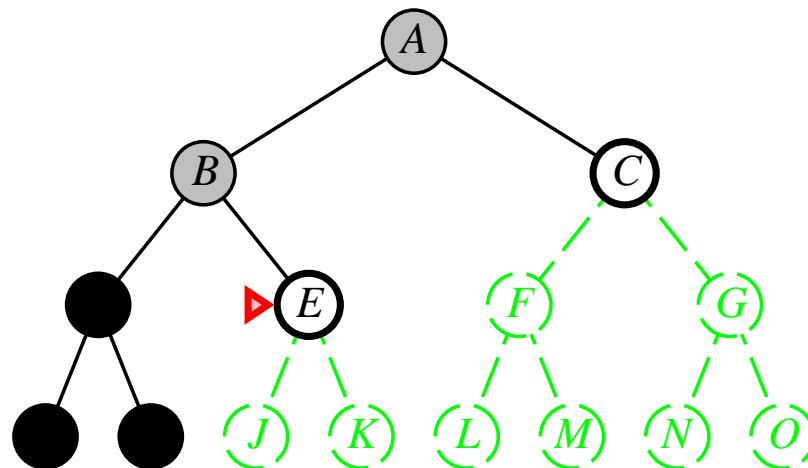


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

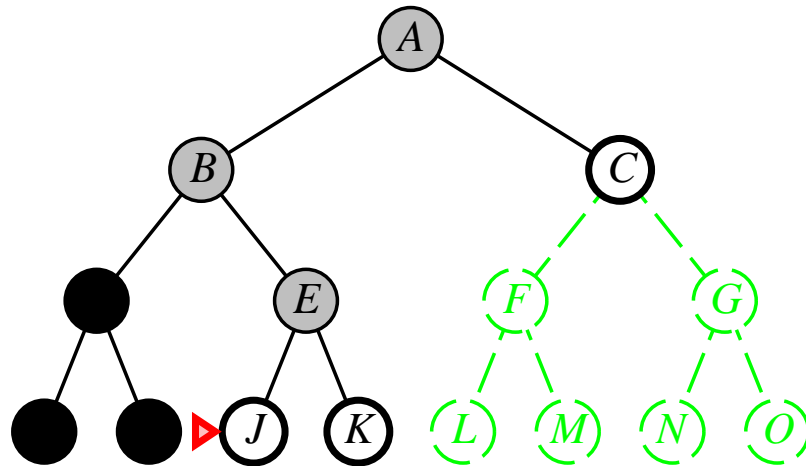


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

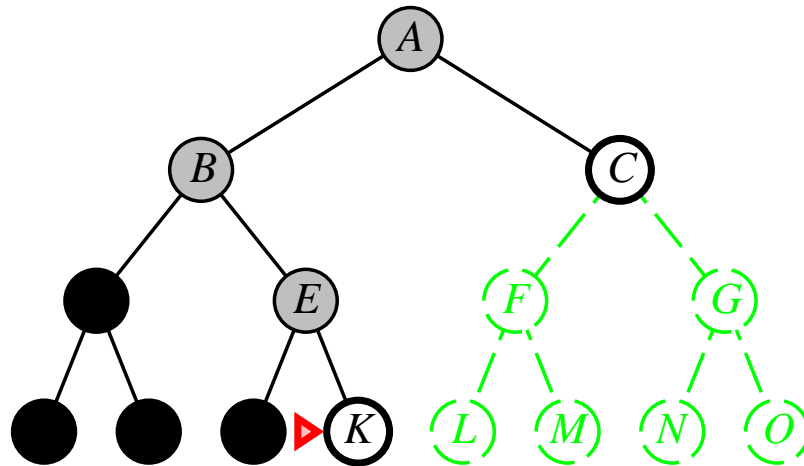


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

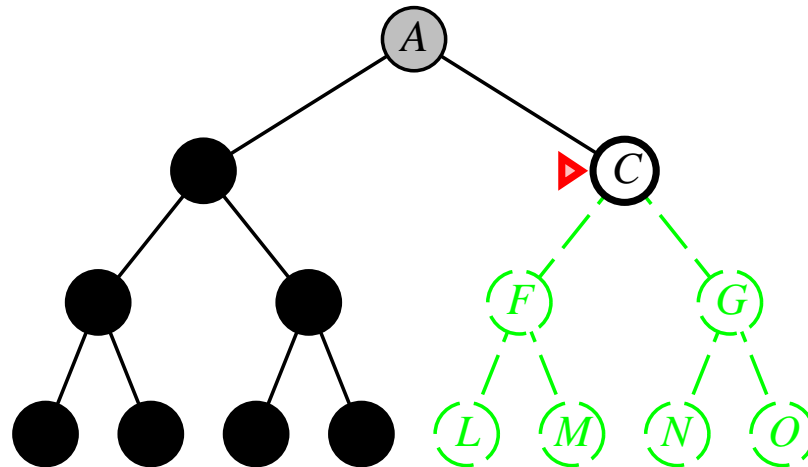


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

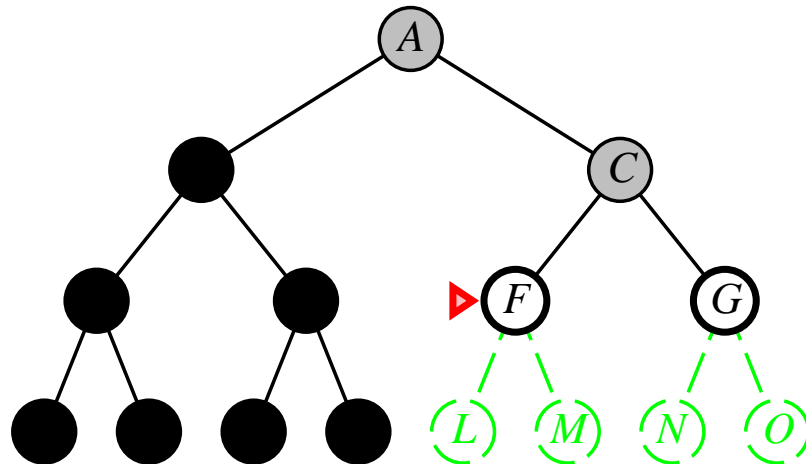


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

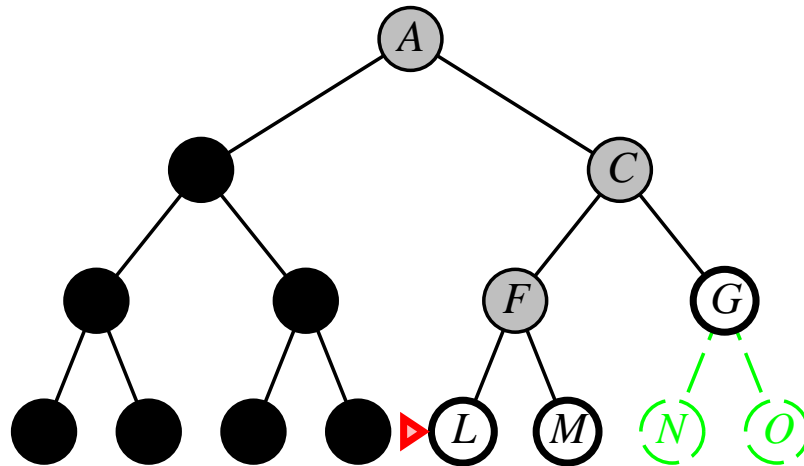


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front

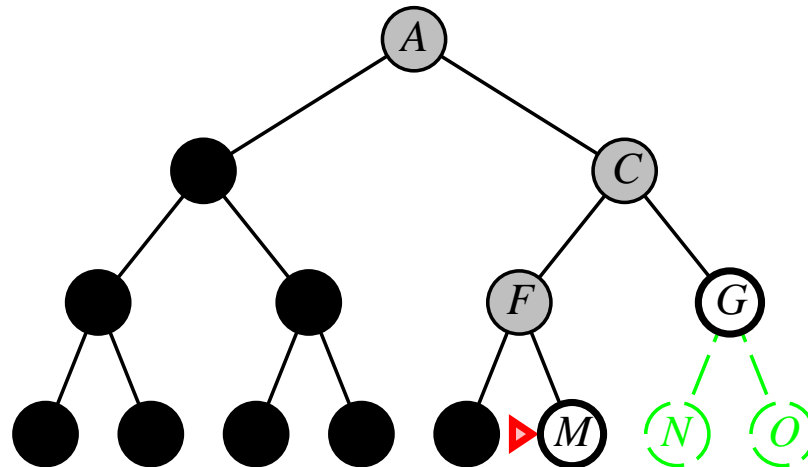


# Depth-first search

Expand deepest unexpanded node

## Implementation:

*fringe* = LIFO queue, i.e., put successors at front



# Properties of depth-first search

Complete?

---

$b$  = maximum branching factor of the search tree  
 $d$  = depth of the least-cost solution  
 $m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of depth-first search

## Complete?

No in infinite-depth spaces

Yes in finite spaces, if we modify to avoid loops:

Backtrack if you reach a state you've already seen on the current path

## Time?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of depth-first search

## Complete?

No in infinite-depth spaces

Yes in finite spaces, if we modify to avoid loops:

Backtrack if you reach a state you've already seen on the current path

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?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of depth-first search

## Complete?

No in infinite-depth spaces

Yes in finite spaces, if we modify to avoid loops:

Backtrack if you reach a state you've already seen on the current path

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 solutions?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of depth-first search

## Complete?

No in infinite-depth spaces

Yes in finite spaces, if we modify to avoid loops:

Backtrack if you reach a state you've already seen on the current path

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 solutions? Not unless it's lucky

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Depth-limited search

Depth-first search, backtrack at each node of depth  $l$  unless it's a solution

## Recursive implementation:

```
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
        /* tells what to return if we don't find a solution */
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

# Iterative deepening search

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

# Iterative deepening search

Limit = 0



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

# Iterative deepening search

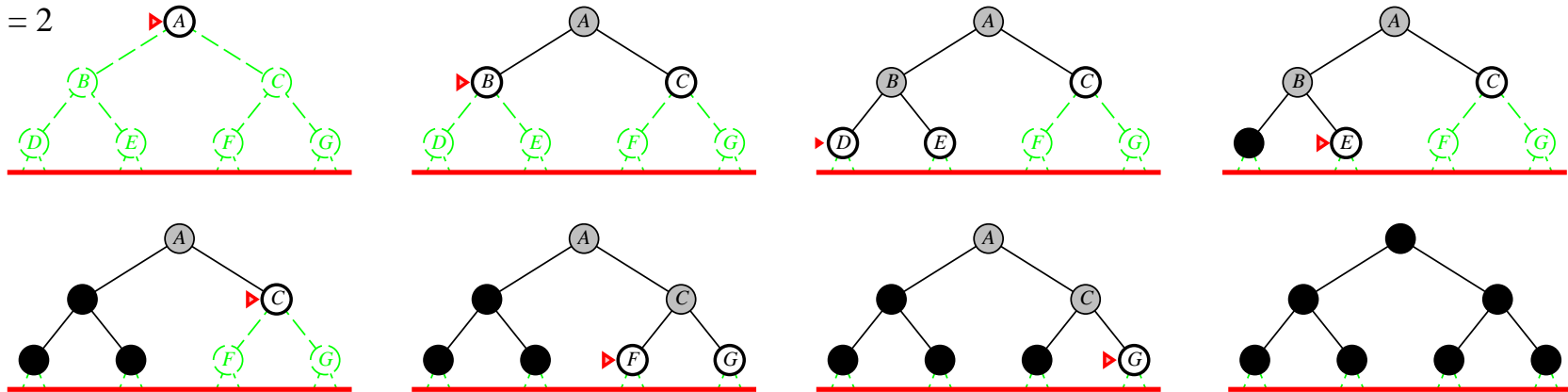
Limit = 1



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

# Iterative deepening search

Limit = 2

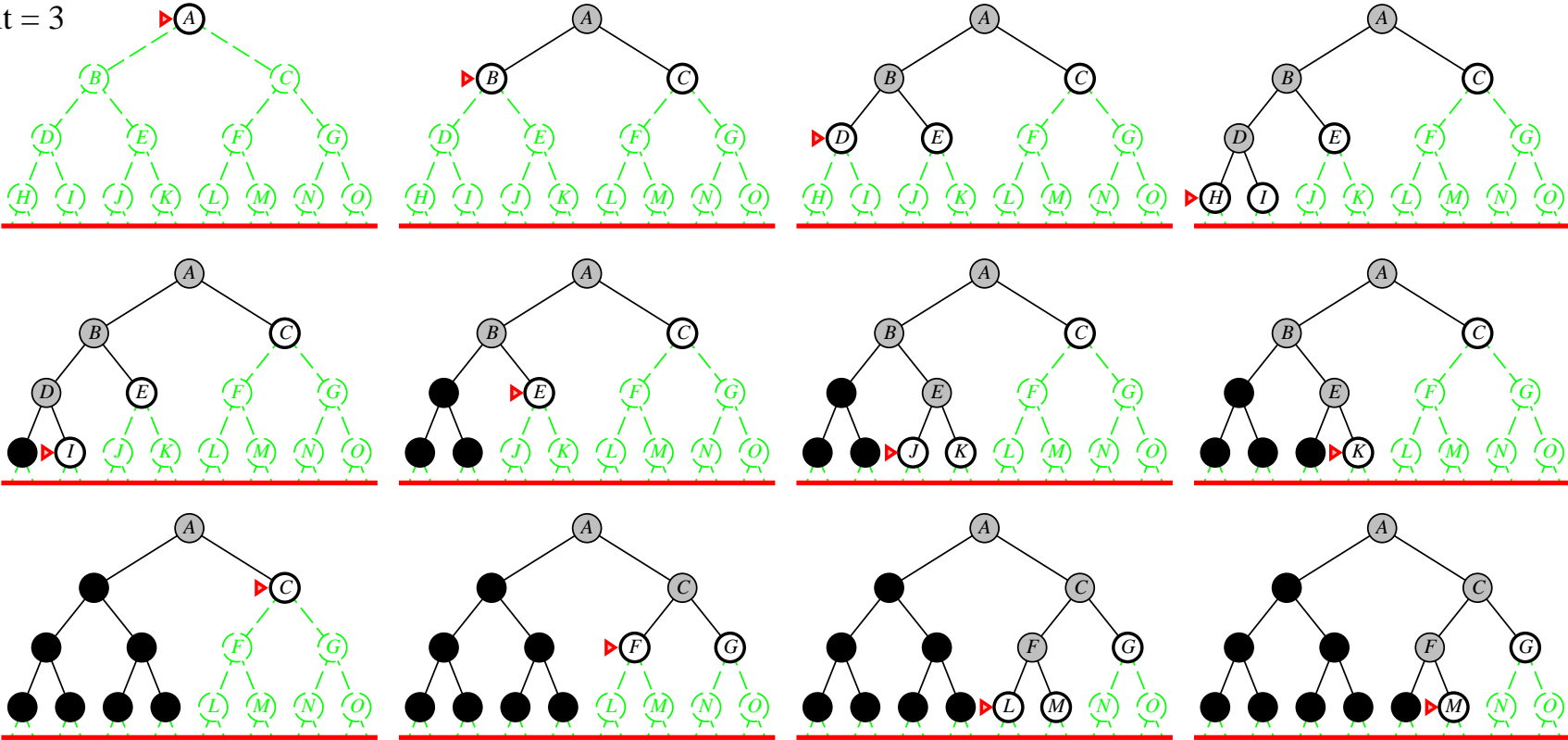


```

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

# Iterative deepening search

Limit = 3



# Properties of iterative deepening search

Complete?

---

$b$  = maximum branching factor of the search tree  
 $d$  = depth of the least-cost solution  
 $m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of iterative deepening search

Complete? Yes

Time?

---

$b$  = maximum branching factor of the search tree  
 $d$  = depth of the least-cost solution  
 $m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of iterative deepening search

Complete? Yes

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

Space?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of iterative deepening search

Complete? Yes

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

Space?  $O(bd)$

Optimal solutions?

---

$b$  = maximum branching factor of the search tree

$d$  = depth of the least-cost solution

$m$  = maximum depth of the state space (may be  $\infty$ )

# Properties of iterative deepening search

Complete? Yes

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

Space?  $O(bd)$

Optimal solutions? Yes, if step cost = 1

Can be modified to behave like uniform-cost search

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

Number of node generations:

IDS:  $50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

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 do the same:

apply the goal test when a node is **generated**

# Tree search algorithms

```
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
```

*If any of them is a solution, return it immediately*

Number of node generations:

$$\text{IDS: } 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$\text{BFS: } 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$$

Highest number of nodes stored:

$$\text{IDS: } 10 \times 5 + 1 = 51$$

$$\text{BFS: } 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$$

## Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes <sup>(2)</sup>	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 <sup>(1)</sup>	Yes	No	No	Yes <sup>(1)</sup>

where

$b$  = branching factor

$C^*$  = cost of optimal solution, or  $\infty$  if there's no solution

$d$  = depth of shallowest solution, or  $\infty$  if there's no solution

$\epsilon$  = smallest cost of each edge

$l$  = cutoff depth for depth-limited search

$m$  = depth of deepest node (may be  $\infty$ )

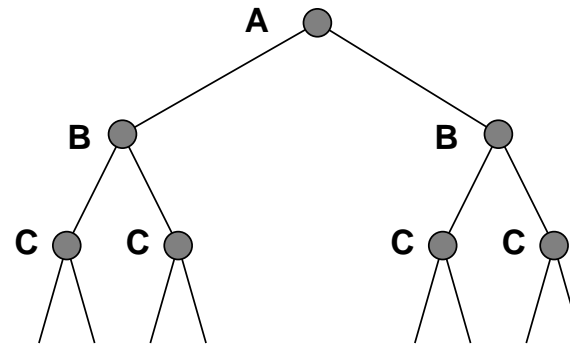
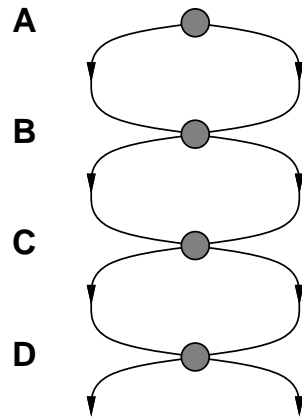
---

<sup>1</sup> if step cost is 1

<sup>2</sup> if  $\epsilon > 0$

# 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 ← an empty set
  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 node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

Can do breadth-first graph search, uniform-cost graph search

Can also do depth-first graph search, but there's a tradeoff:

- ◇ Sometimes get exponentially less time than depth-first tree search
- ◇ Usually need exponentially more memory than depth-first tree search

## Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

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

**Homework assignment** (due in one week)  
five problems, 10 points each – total 50 points

2.9, 3.7(a,b), 3.8, 3.9(a,c), 3.13