


# Chapter 3

## Solving Problems by Searching



Jane Hsu  
National Taiwan University

Acknowledgements: This presentation is created by Jane hsu based on the lecture slides from *The Artificial Intelligence: A Modern Approach* by Russell & Norvig, a PowerPoint version by Min-Yen Kan, as well as various materials from the web.

# The Physical Symbol System Hypothesis

---

- *A physical symbol system has the necessary and sufficient means for intelligent action.*
  - The class of computational models defined by a Turing Machine
  - **Designation.** An expression designates an object if, given the expression, the system can either affect the object itself or behave in ways dependent on the object
  - **Interpretation.** The system can interpret an expression if the expression designates a process and if, given the expression, the system can carry out the process.
  - "Computer Science as Empirical Inquiry: Symbols and Search" by Allen Newell & Herbert A. Simon, 1975 ACM Turing Award Lecture

# General Search Problem

---

- Search is a *universal problem solving mechanism* that
  - Systematically explores the alternatives.
  - Finds the sequence of steps toward a solution.
- Problem Space Hypothesis (SOAR [Newell])
  - All goal-oriented symbolic activities occur in a problem space.
  - Search in a problem space is claimed to be a completely general model of intelligence.

# Outline

---

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

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

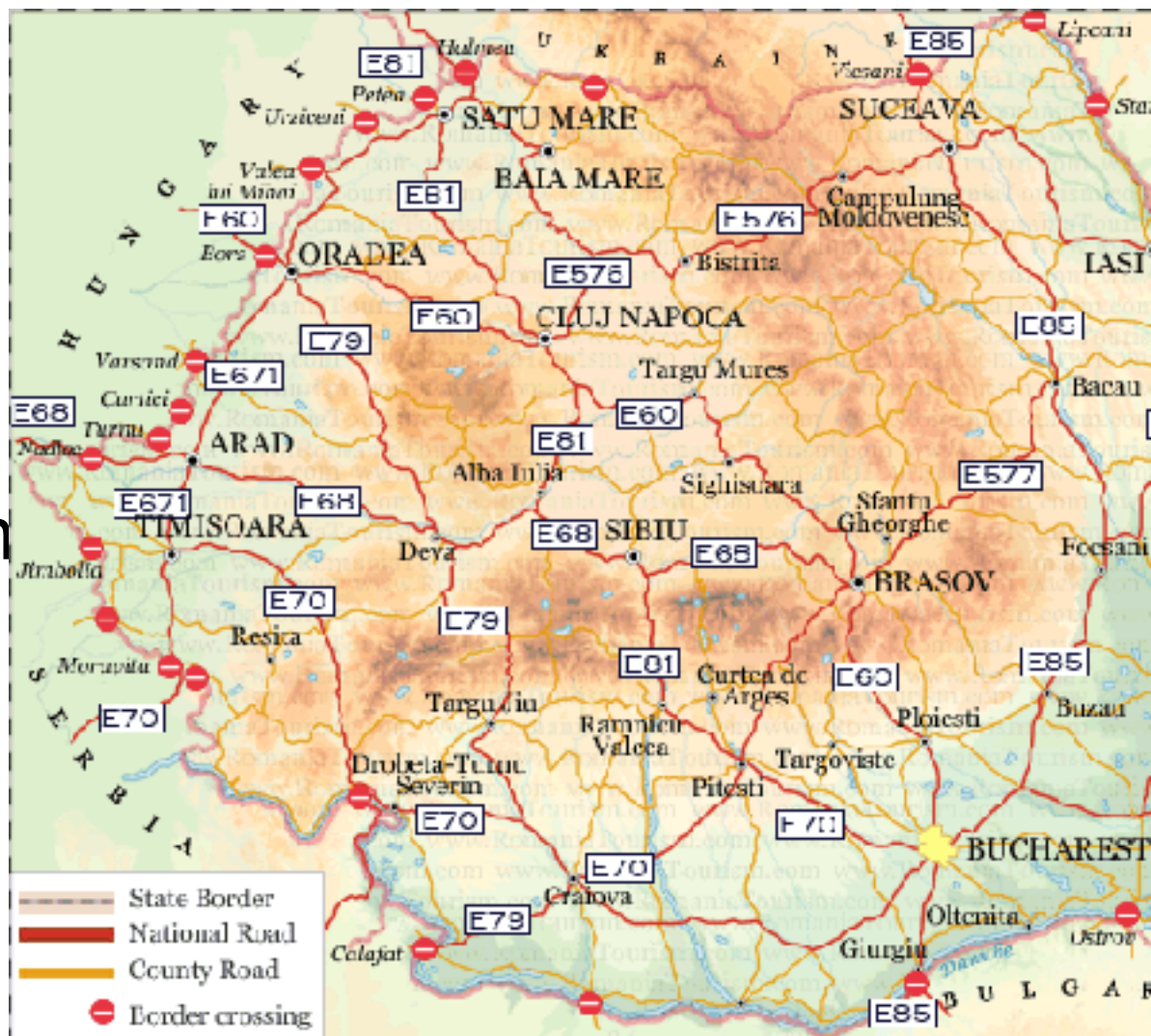
# Problem Types

---

- **Deterministic, fully observable → single-state problem**
  - Agent knows exactly which state it will be in; solution is a sequence
  
- Non-observable → sensorless problem (conformant 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
  - often interleave} search, execution
  
- Unknown state space → exploration problem

# Example: Route-finding in Romania

- On vacation in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest



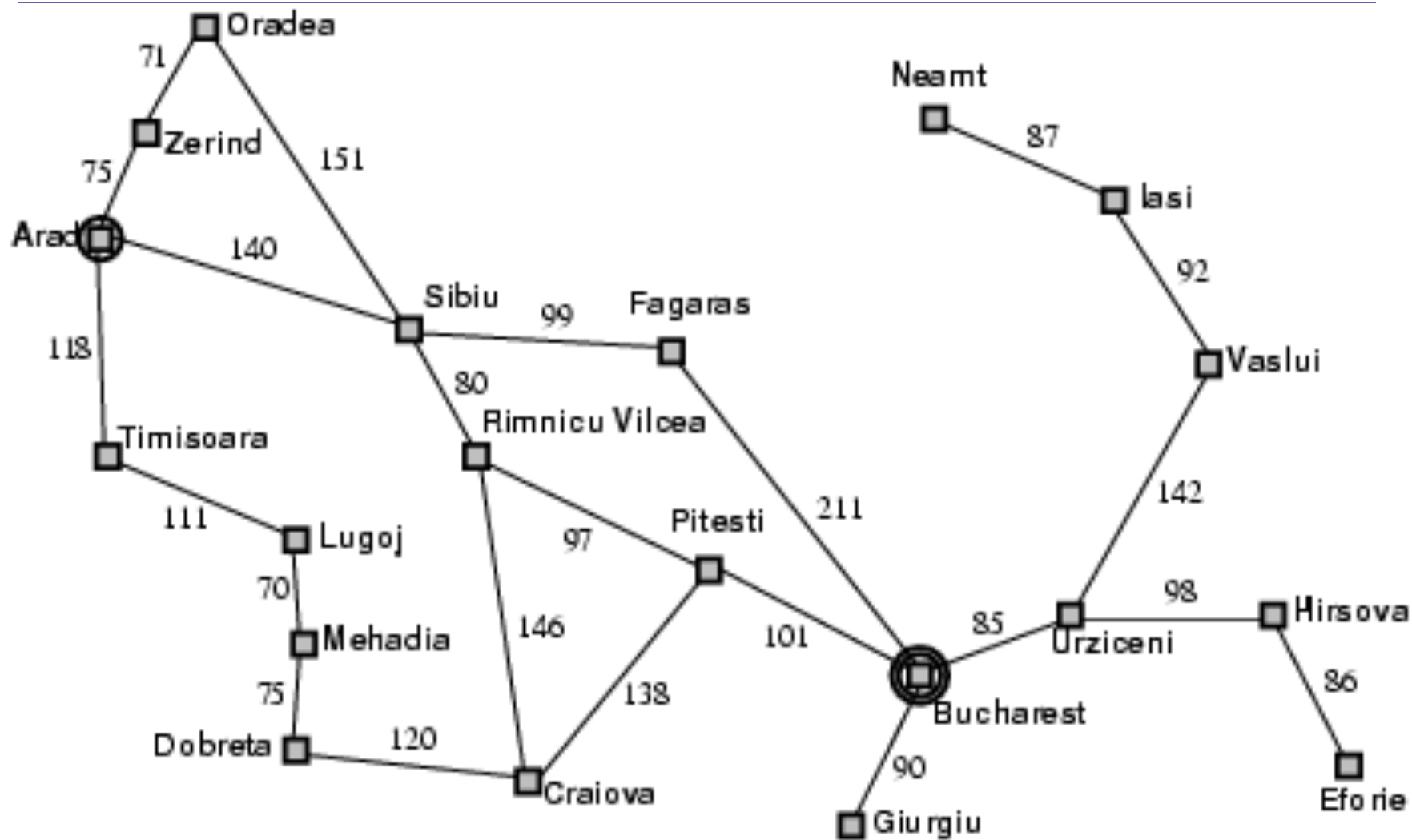
# Problem Solving as Search

---

- On vacation 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



# Romanian Road Map



# Single-State Problem Formulation

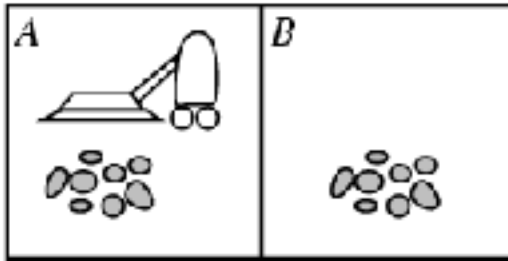
---

A **problem** is defined by four items:

1. **initial state** e.g., "at Arad"
  - ▣ **actions** or **successor function**  $S(x)$  = set of action–state pairs
    1. e.g.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
  - ▣ **goal test**, can be
    - **explicit**, e.g.,  $x = \text{"at Bucharest"}$
    - 2. **implicit**, e.g.,  $\text{Checkmate}(x)$
  - ▣ **path cost** (additive)
    - e.g., sum of distances, number of actions executed, etc. $c(x, a, y)$  is the **step cost**, assumed to be  $\geq 0$
- ▣ A **solution** is a sequence of actions leading from the initial state to a goal state

# Vacuum-Cleaner World

---



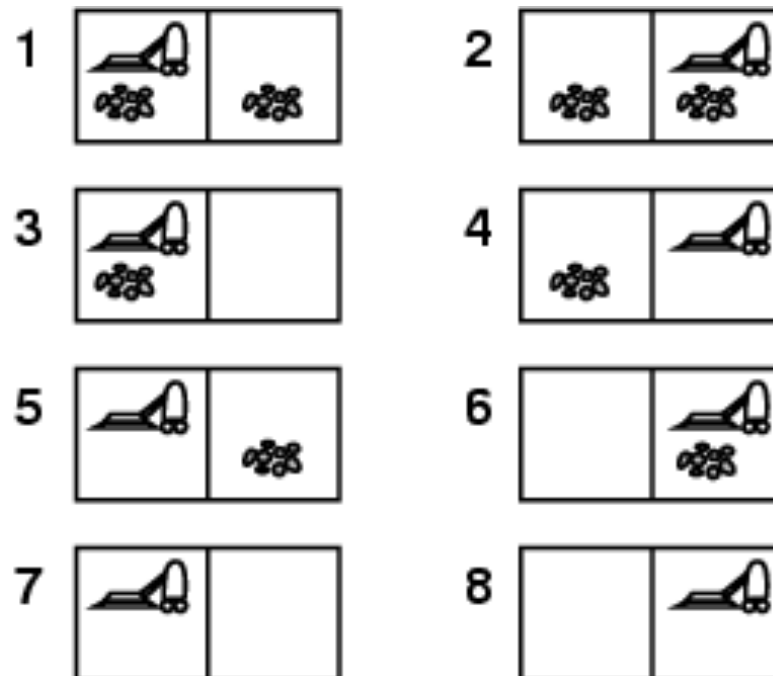
- Percepts: location and contents, e.g.,  $[A, \textit{Dirty}]$
- Actions: *Left*, *Right*, *Suck*, *NoOp*

# How many different states are there?



# Example: Vacuum World (1/4)

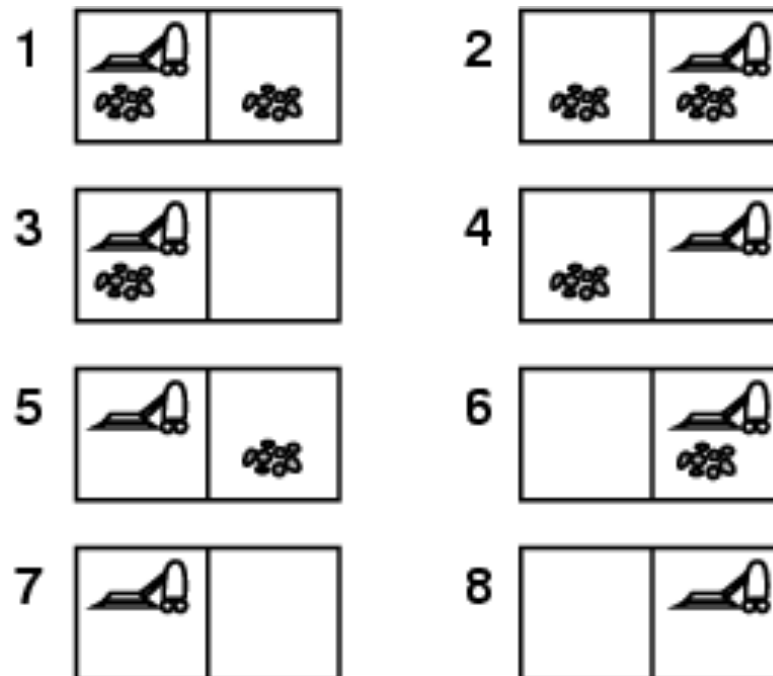
- Single-state, start in #5.  
Solution?



# Example: Vacuum World (2/4)

□ **Single-state**, start in #5.  
Solution? [*Right, Suck*]

□ **Sensorless**, start in  
 $\{1,2,3,4,5,6,7,8\}$  e.g.,  
*Right* goes to  $\{2,4,6,8\}$   
Solution?

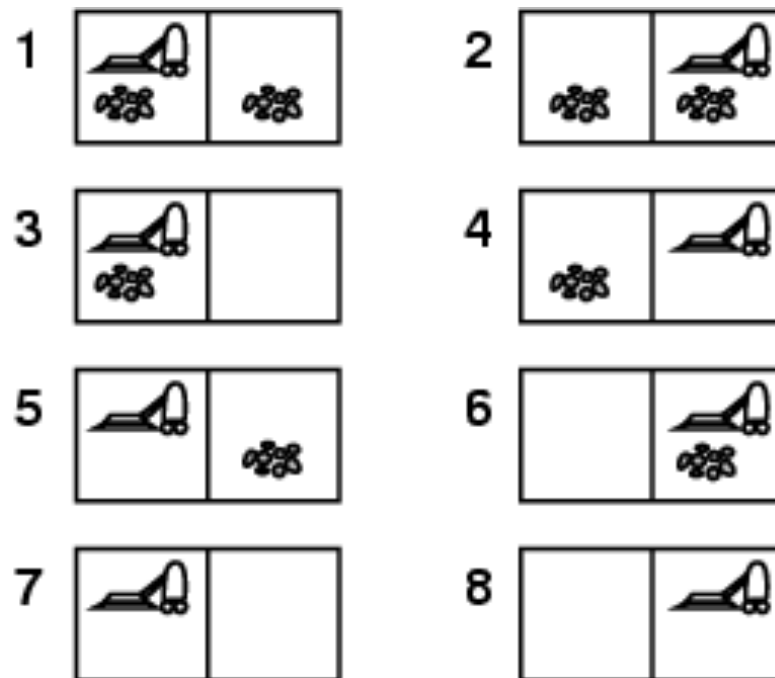


# Example: Vacuum World (3/4)

- **Sensorless**, start in  $\{1,2,3,4,5,6,7,8\}$  e.g., *Right* goes to  $\{2,4,6,8\}$   
Solution?  
*[Right, Suck, Left, Suck]*

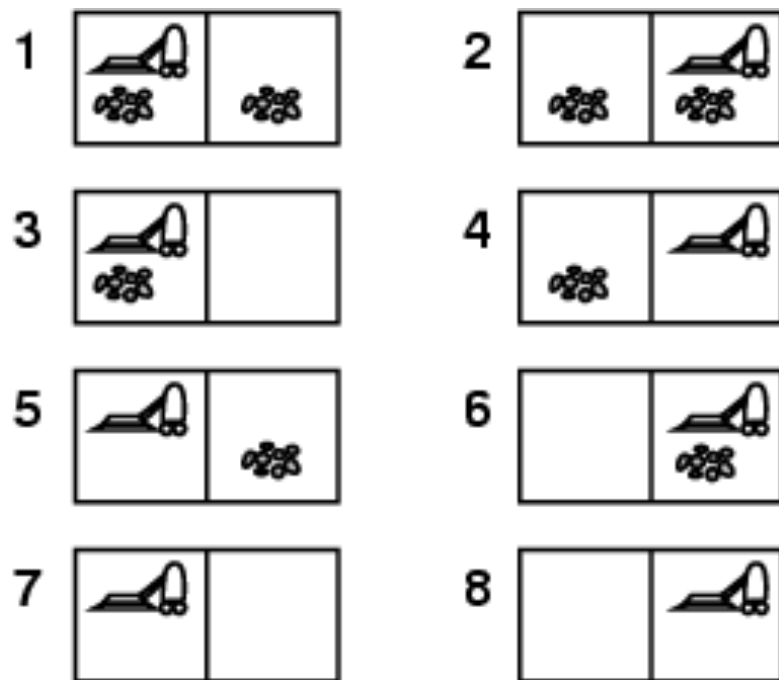
偶然事件

- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept:  $[L, \text{Clean}]$ , i.e., start in #5 or #7Solution?



# Example: Vacuum World (4/4)

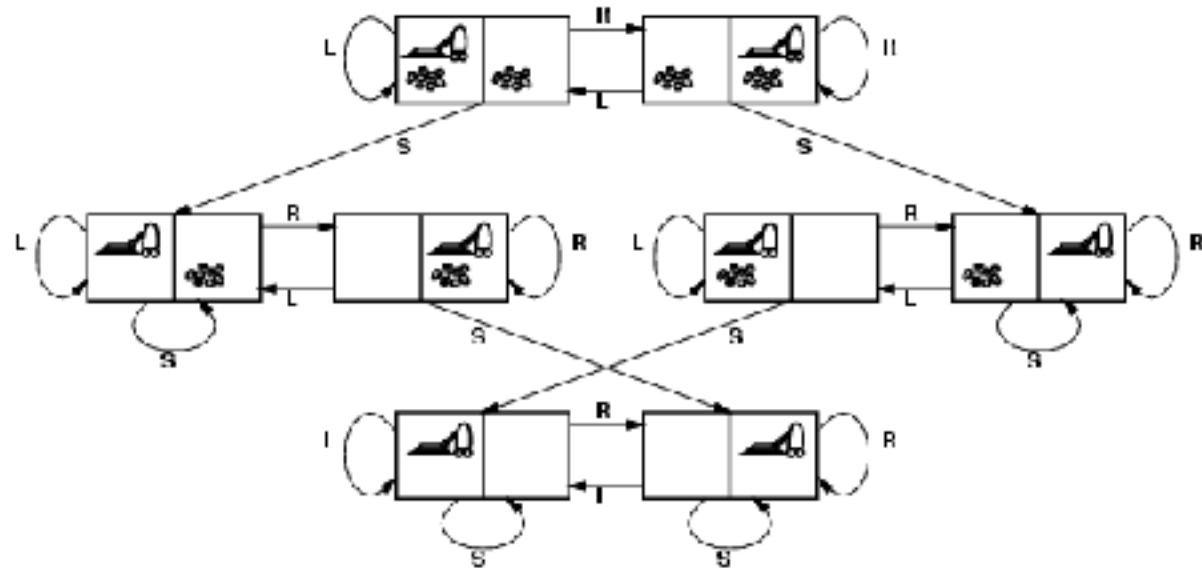
- **Sensorless**, start in  $\{1,2,3,4,5,6,7,8\}$  e.g.,  
*Right* goes to  $\{2,4,6,8\}$   
Solution?  
*[Right, Suck, Left, Suck]*



- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept:  $[L, \text{Clean}]$ , i.e., start in #5 or #7  
Solution? *[Right, **if** dirt **then** Suck]*

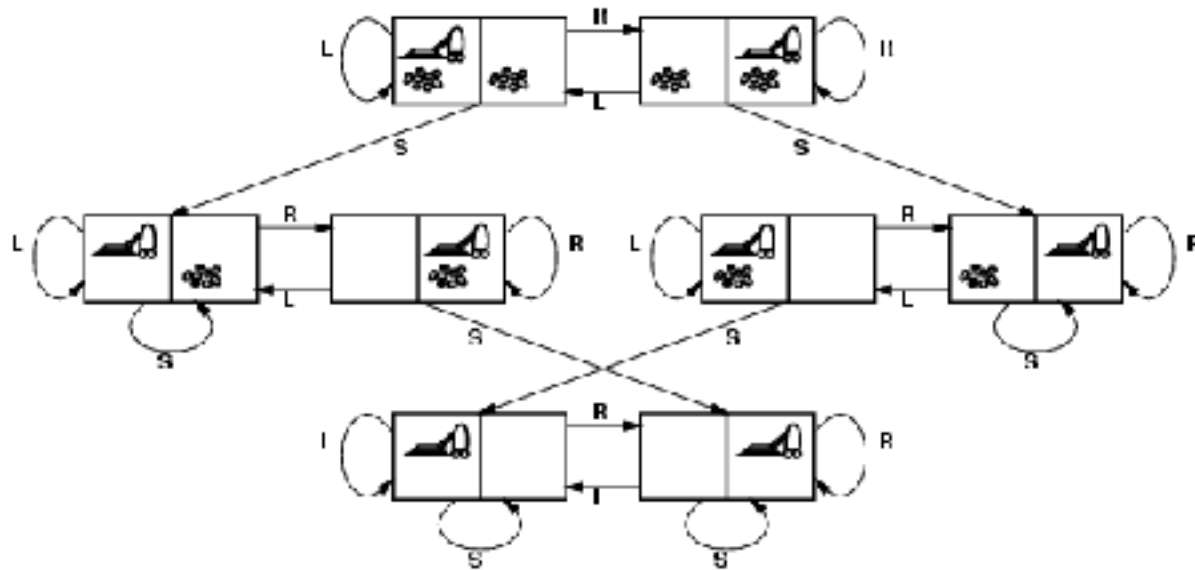


# Vacuum World State Space Graph



- states?
- actions?
- goal test?
- path cost?

# Vacuum World State Space Graph



- states? integer dirt and robot location
- actions? *Left, Right, Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

# Questions

---

- What are the states?
  - What are the legal actions?
  - What is the goal test?
  - What is the cost of the solution path?
- 
- How big is the state space?
  - How big is the search space?

# Sample Problem Domains

---

## **Toy problem domains**

- Tower of Hanoi
- Water jug
- 8-puzzle
- Vacuum world
- Wumpus world
- Missionaries & cannibals
- Monkey and Bananas
- Blocks world
- Cryptarithmic
- Crossword puzzles
- Chess, Bridge, Go

## **Real-world problems**

- Route-finding
- Touring: travelling salesperson problem
- VLSI layout
- Robot navigation
- Automatic assembly sequencing
- Scheduling & planning
- Protein design

# Example: Missionaries and Cannibals

---

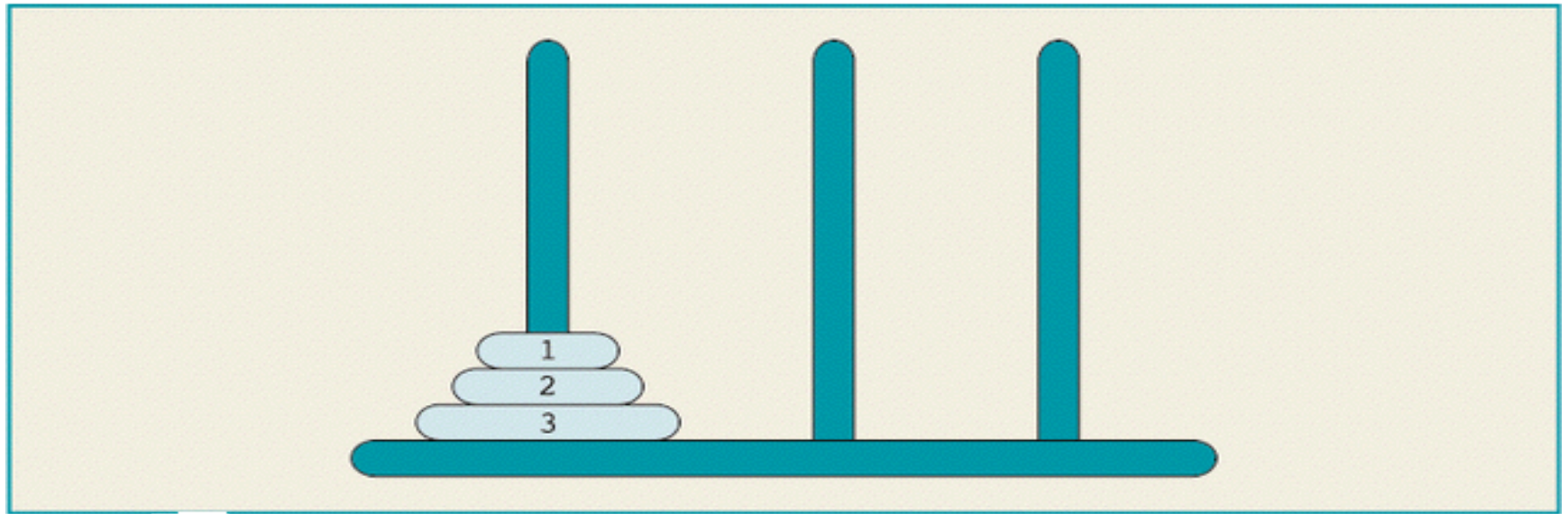
- Three missionaries and three cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten.
- states? (mcb) *the number on the initial bank*
- actions?
- goal test?
- path cost?

# Example: Water Jug Problem

---

- You are given two jugs with no measuring marks, a 4-gallon one and a 3-gallon one.
- There is a pump to fill the jugs with water.
- How can you get exactly 2 gallons of water into the 4-gallon jug?

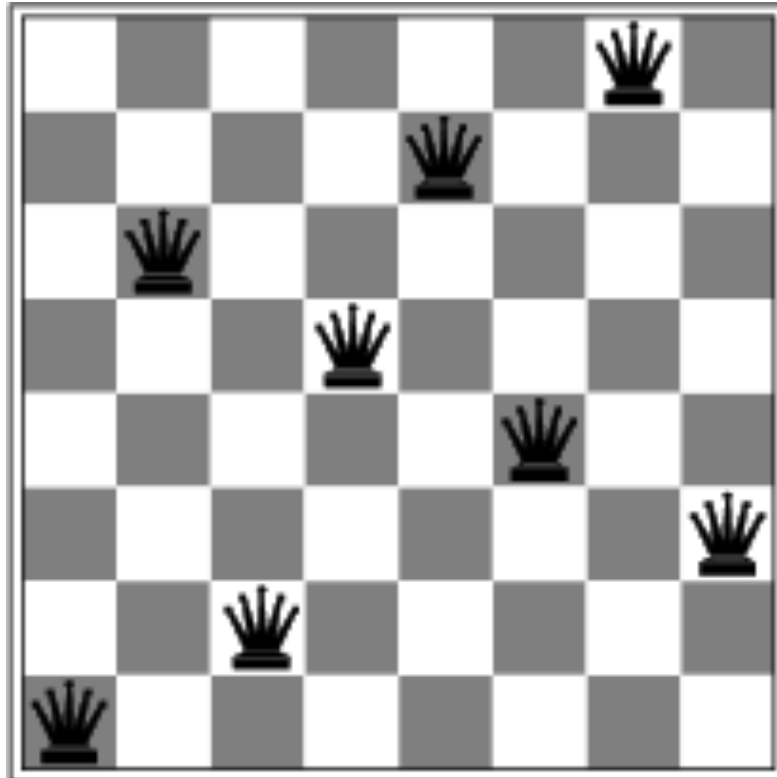
# Example: Tower of Hanoi



Tower of Hanoi problem with three disks

# Example: 8-Queens Problem

---





# Example: The 8-puzzle

---

7	2	4
5		6
8	3	1

**Start State**

	1	2
3	4	5
6	7	8

**Goal State**

# 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?
- actions?
- goal test?
- path cost?

# 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? locations of tiles
  - actions? move blank left, right, up, down
  - goal test? = goal state (given)
  - path cost? 1 per move
- [Note: optimal solution of n-Puzzle family is NP-hard]

# General Search Problem

---

## □ Given:

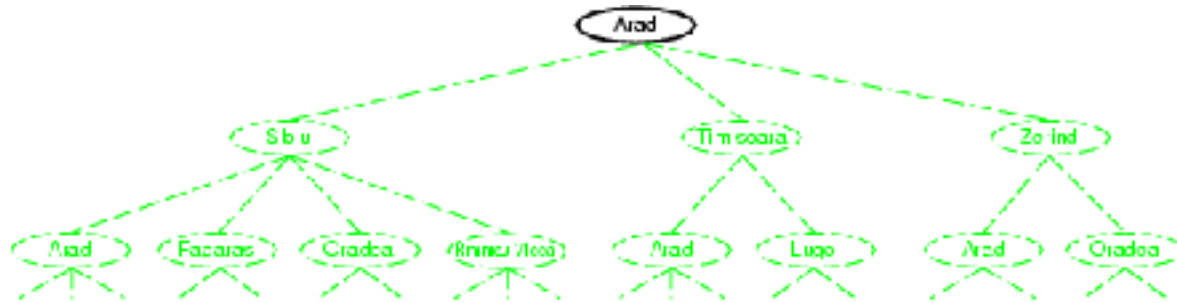
- Problem space (or *state space*)
  - A set of nodes  $N$  (each representing a problem state)
  - A function  $Next(n)$  defining the next states
- Start node
- Goal: a subset of  $N$

## □ To find

- One path from the start node to a goal node
- All paths from the start node to any goal node
- The best path from the start to the best goal

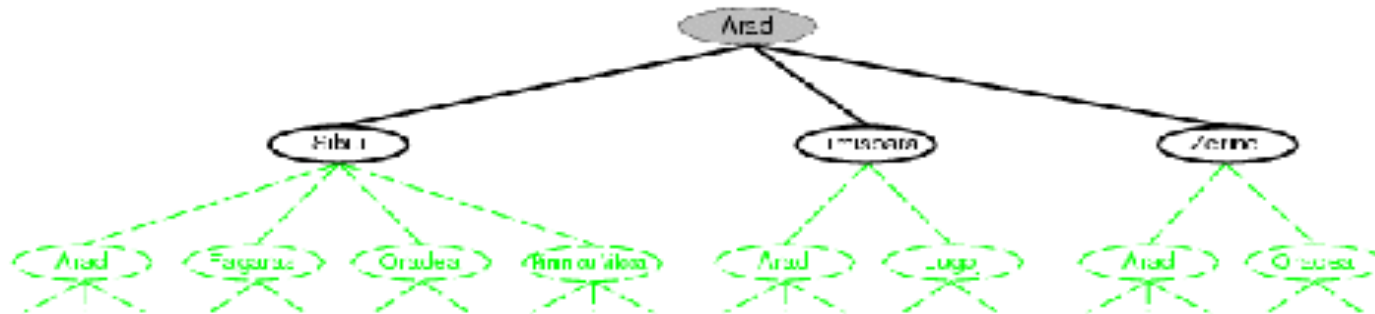
# Tree Search Example (1/3)

---



# Tree Search Example (2/3)

---



# Tree Search Example (3/3)

---



# Tree Search

---

## □ Basic idea:

- An offline, simulated exploration of state space by generating successors of already-explored states
- The *frontier* of the tree are the nodes yet to be explored.
- A node  $n$  is expanded using  $Next(n)$ .



# General Tree-Search and Graph-Search

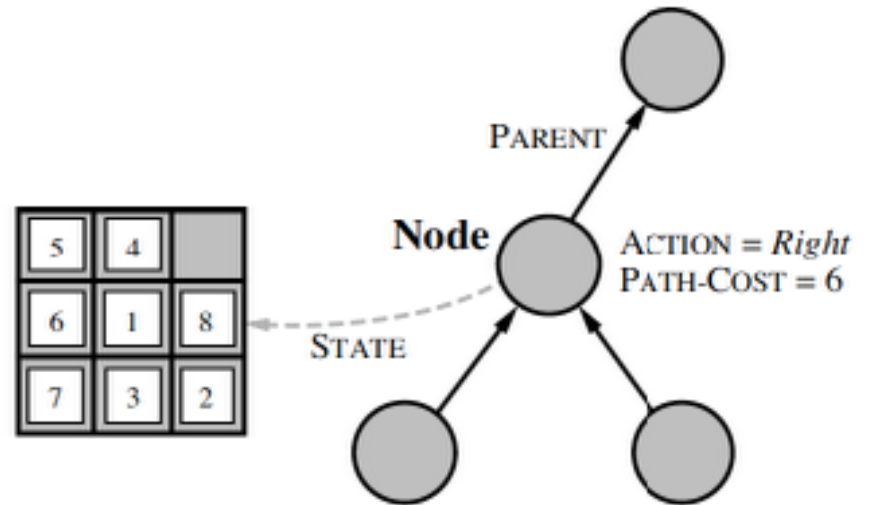
```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  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
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

**Figure 3.7** An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

# Infrastructure of Search Algorithms

- $n$ .STATE
- $n$ .PARENT
- $n$ .ACTION
- $n$ .PATH-COST



# 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

# Properties of Search Strategies

---

- A search 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
  - **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
  - $b$ : the maximum number of successors of any node
  - $d$ : the depth of the shallowest goal node
  - $m$ : the maximum length of any path in the state space (may be  $\infty$ )

# Breadth-First Search

---

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

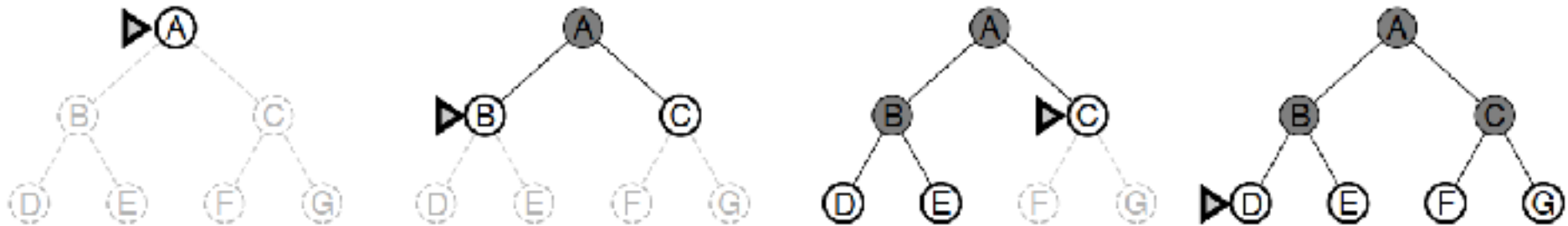
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

# Breadth-First Search

□ Expand shallowest unexpanded node

□ **Implementation:**

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



**b=10**, 10,000 nodes/sec, 1KB/node

---

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	$10^7$	19 minutes	10 gigabytes
8	$10^9$	31 hours	1 terabytes
10	$10^{11}$	129 days	101 terabytes
12	$10^{13}$	35 years	10 petabytes
14	$10^{15}$	3,523 years	1 exabye

# Properties of Breadth-First Search

---

- Complete? Yes (if  $b$  is finite)
- Time?  $1 + b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$
- Space?  $O(b^{d+1})$  (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
  
- **Space** is the bigger problem (more than time)

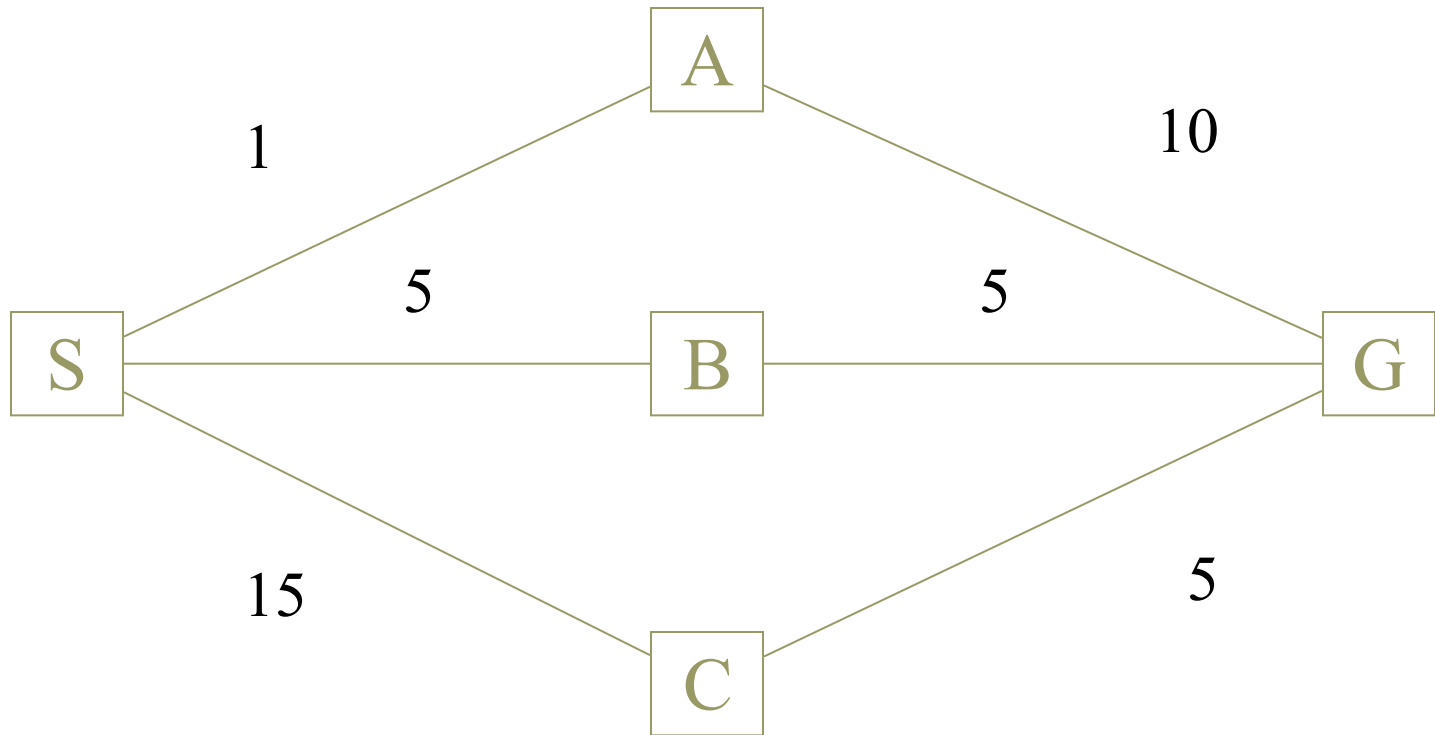


# Uniform-Cost Search

---

- A breadth-first search finds the shallowest goal state and will therefore be the cheapest solution provided the path cost is a function of the depth of the solution. But, if this is not the case, then breadth-first search is not guaranteed to find the best (i.e. cheapest solution).
- Uniform cost search remedies this by expanding the lowest cost node on the fringe, where cost is the path cost,  $g(n)$ .
- In the following slides those values that are attached to paths are the cost of using that path.

Consider the following problem...



We wish to find the shortest route from node S to node G; that is, node S is the initial state and node G is the goal state. In terms of path cost, we can clearly see that the route *SBG* is the cheapest route. However, if we let breadth-first search loose on the problem it will find the non-optimal path *SAG*, assuming that A is the first node to be expanded at level 1. Press space to see a Uniform-Cost Search of the same node set.

# Uniform Cost Search

---

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

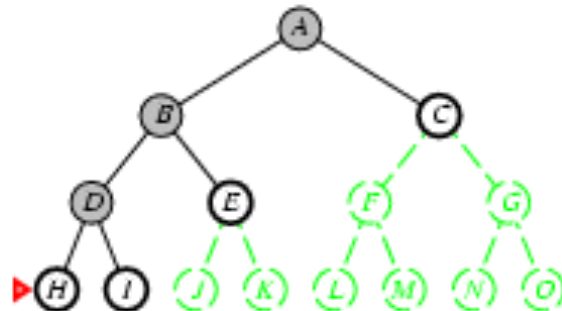
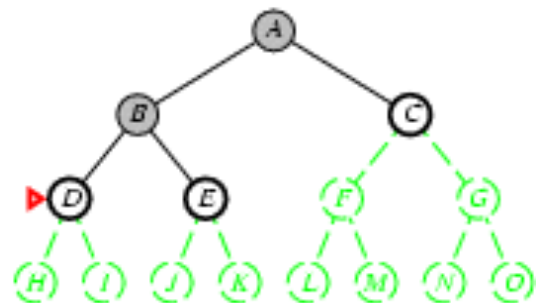
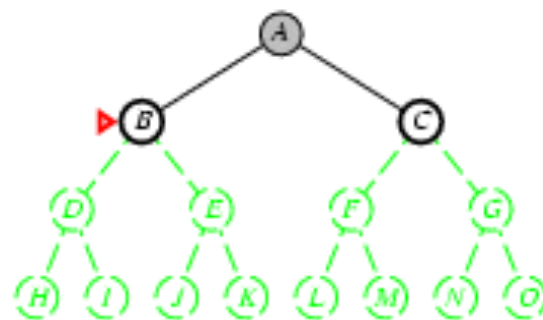
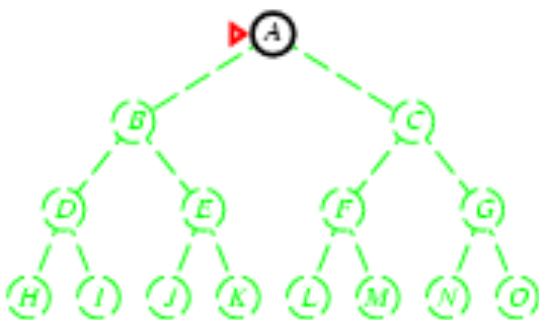
# Uniform-Cost Search

---

- Expand least-cost unexpanded node
- **Implementation:**
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
  
- Complete? Yes, if step cost  $\geq \epsilon$
- Time? # of nodes with  $g \leq$  cost of optimal solution,  
 $O(b^{\text{ceiling}(C^*/\epsilon)})$ 
  - where  $C^*$  is the cost of the optimal solution
  - How does it compare with  $b^d$  ?
- Space? # of nodes with  $g \leq$  cost of optimal solution,  
 $O(b^{\text{ceiling}(C^*/\epsilon)})$
- Optimal? Yes – nodes expanded in increasing order of  $g(n)$

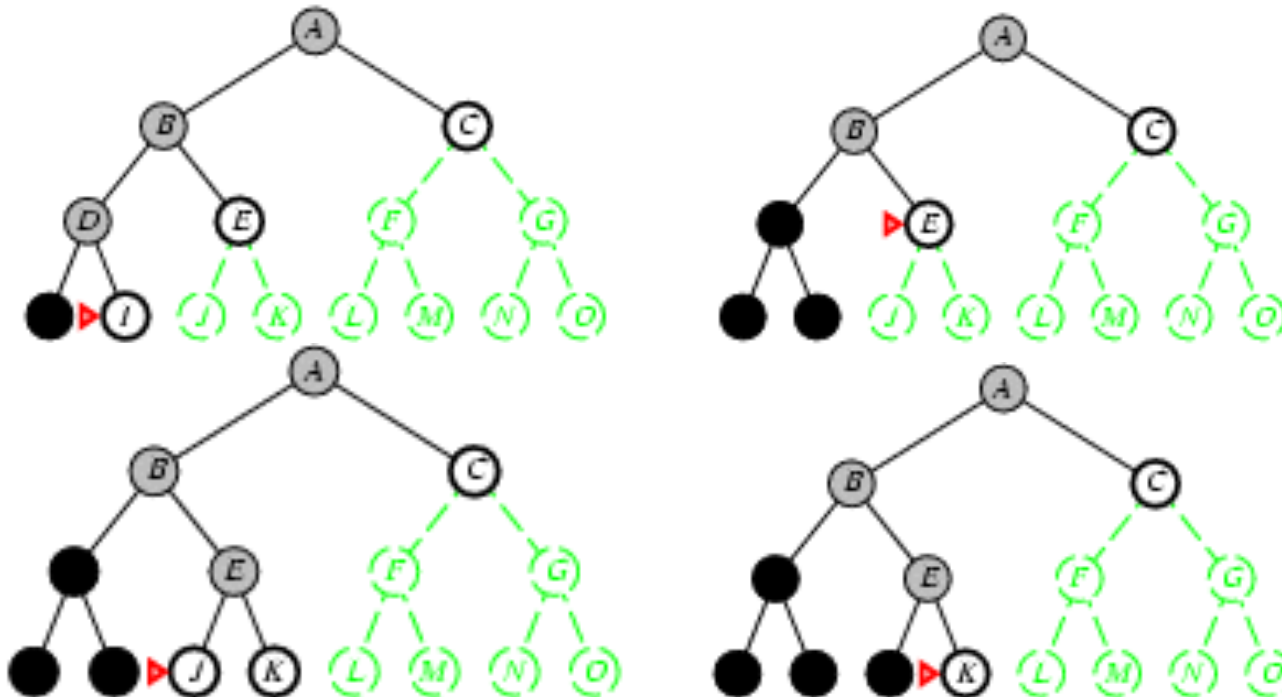
# Depth-First Search (1/3)

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



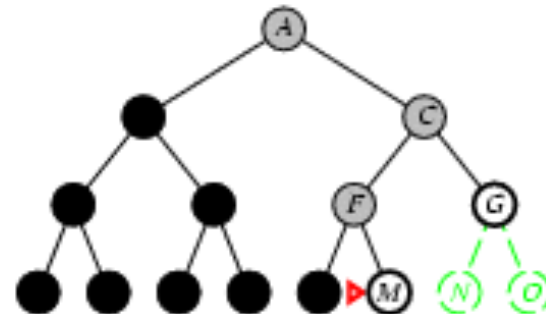
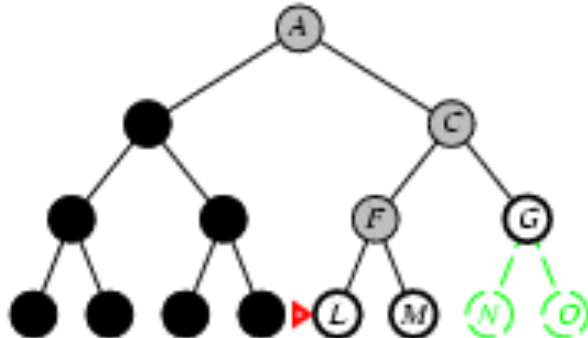
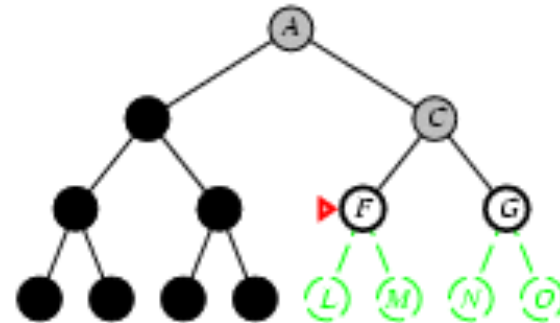
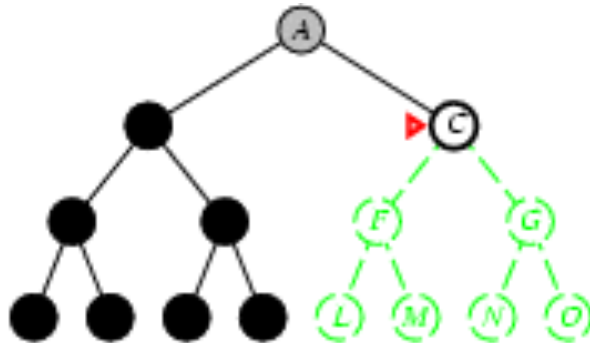
# Depth-First Search (2/3)

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



# Depth-First Search (3/3)

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



# Properties of 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?  $O(bm)$ , 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 a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

# Iterative Deepening Search

---

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

# Iterative Deepening Search $l=0$

---

Limit = 0



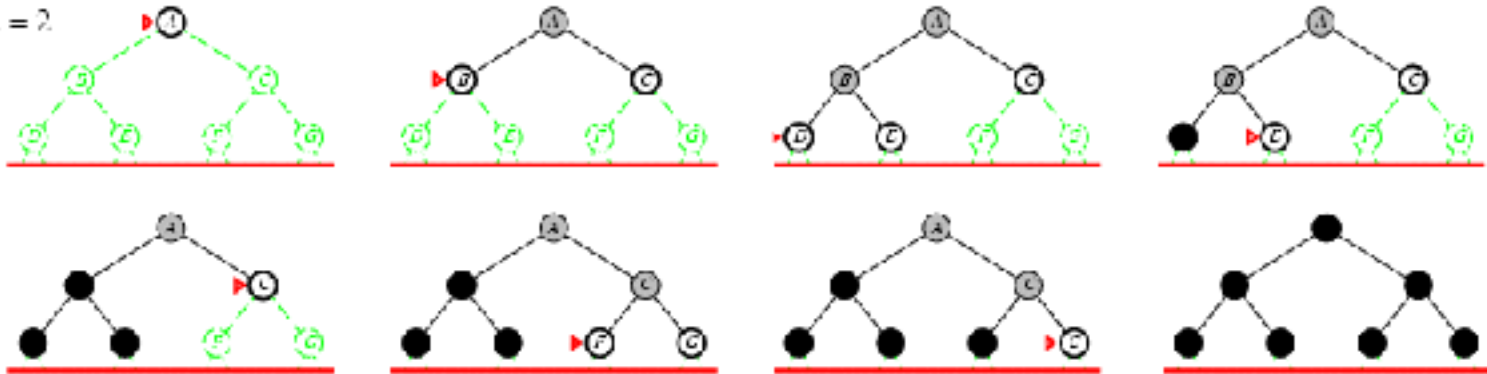
# Iterative Deepening Search $l=1$

Limit = 1

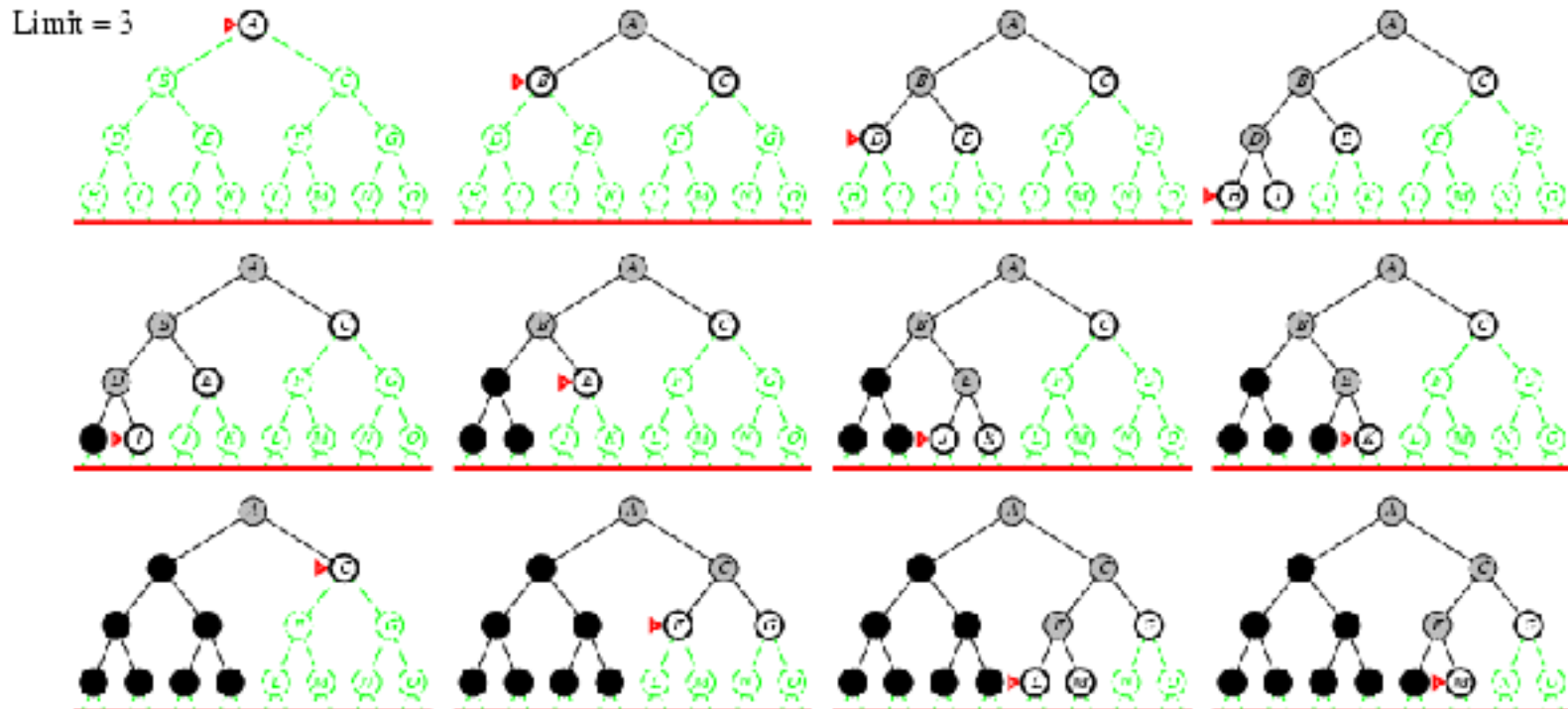


# Iterative Deepening Search $l=2$

Limit = 2



# Iterative Deepening Search $l=3$



# Analysis

---

- Number of nodes generated in a depth-limited search to depth  $d$  with branching factor  $b$ :

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

- Number of nodes generated in an iterative deepening search to depth  $d$  with branching factor  $b$ :

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

- For  $b = 10, d = 5$ ,

- $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
- $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead =  $(123,456 - 111,111)/111,111 = 11\%$

# Properties of Iterative Deepening Search

---

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



# Summary of Uninformed Search

---

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

# 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