# Solving Problems by Searching

Berlin Chen 2004

Reference:

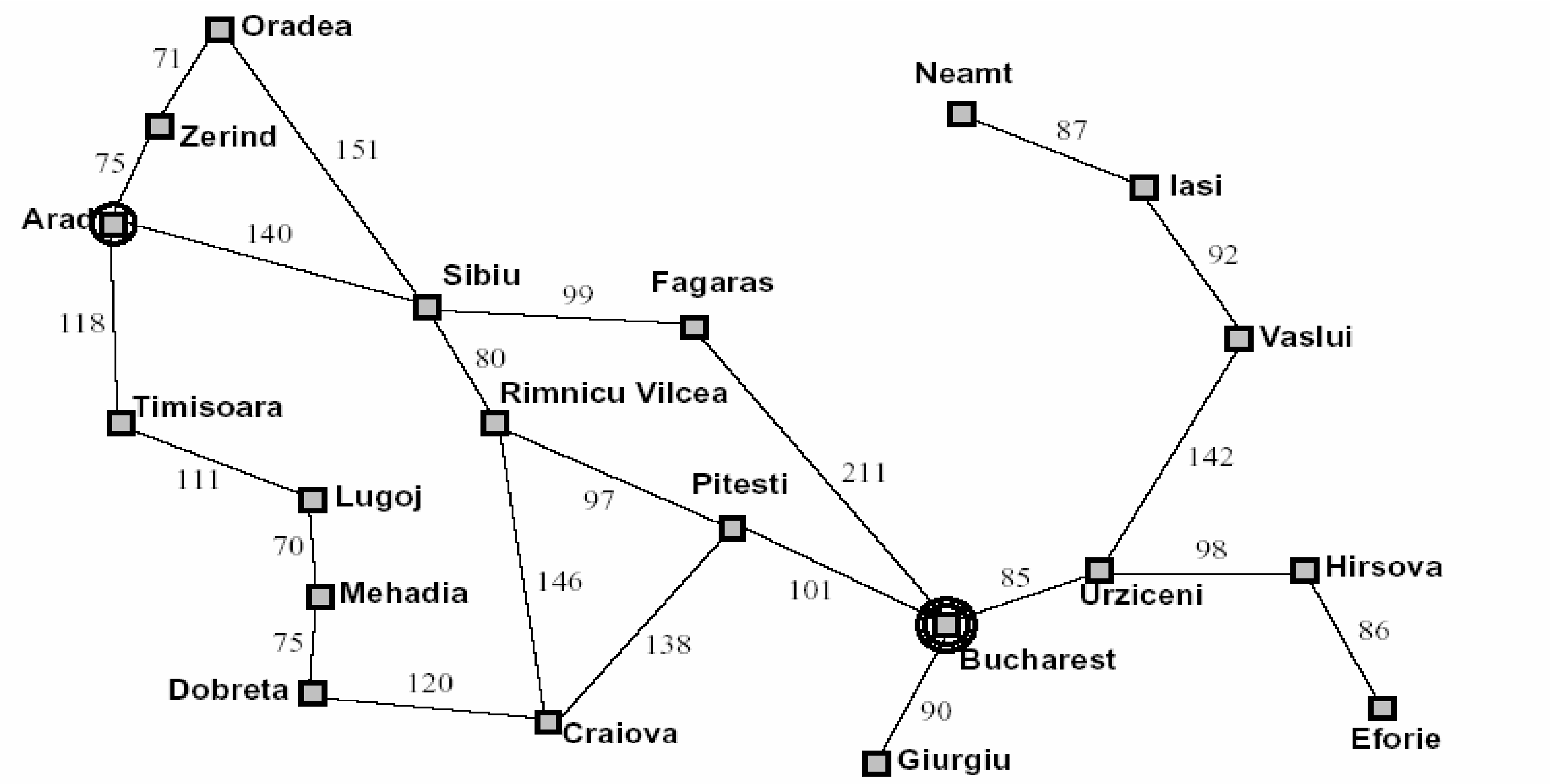
1. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Chapter 3

# Introduction

- Problem-Solving Agents vs. Reflex Agents
  - Problem-solving agents : a kind of goal-based agents
    - Decide what to do by finding sequences of actions that lead to desired solutions
  - Reflex agents
    - The actions are governed by a direct mapping from states to actions

- Problem and Goal Formulation
  - Performance measure
  - Appropriate Level of Abstraction/Granularity
    - Remove details from a representation
    - To what level of description of the states and actions should be considered ?

# Map of Part of Romania



- Find a path from Arad to Bucharest
  - With fewest cities visited
  - Or with a shortest path cost
  - ….

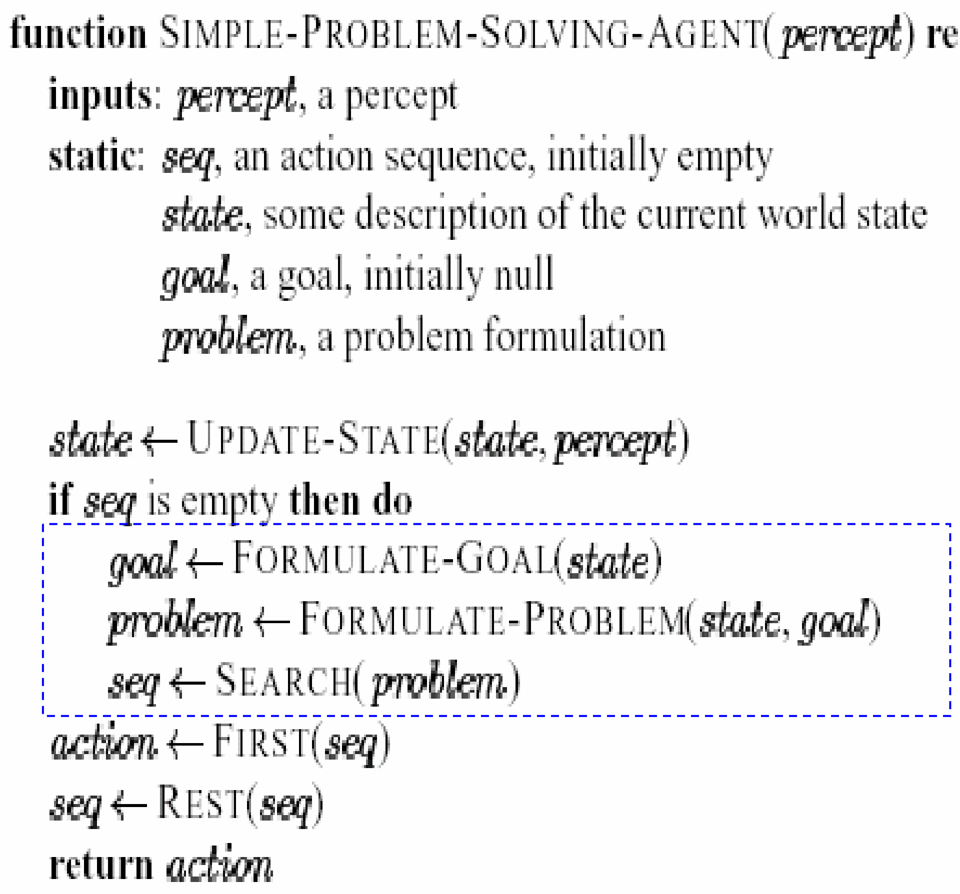
# Search Algorithms

- Take a problem as input and return a solution in the form of an action sequence
  - Formulate → Search → Execution

- Search Algorithms introduced here
  - General-purpose
  - Uninformed: have no idea of where to look for solutions, just have the problem definition
  - Offline searching

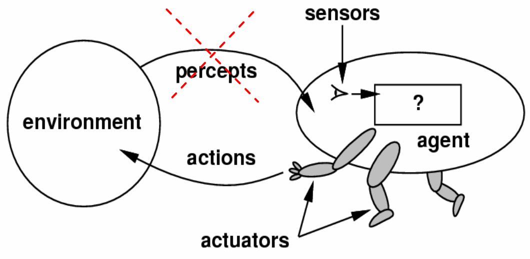
- Offline searching vs. online searching ?

# A Simple-Problem Solving Agent



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    inputs: percept, a percept
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

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

Done once?



- Formulate → Search → Execute

# A Simple-Problem Solving Agent (cont.)

- The task environment is
  - Static
    - The environment will not change when formulating and solving the problem
  - Observable
    - The initial state and goal state are known
  - Discrete
    - The environment is discrete when enumerating alternative courses of action
  - Deterministic
    - Solution(s) are single sequences of actions
    - Solution(s) are executed without paying attention to the percepts

# A Simple-Problem Solving Agent (cont.)

- Problem formulation
  - The process of deciding what actions and states to consider, given a goal
  - Granularity: Agent only consider actions at the level of driving from one major city (state) to another

- World states vs. problem-solving states
  - World states
    - The towns in the map of Romania
  - Problem-solving states
    - The different paths that connecting the initial state (town) to a sequence of other states constructed by a sequence of actions

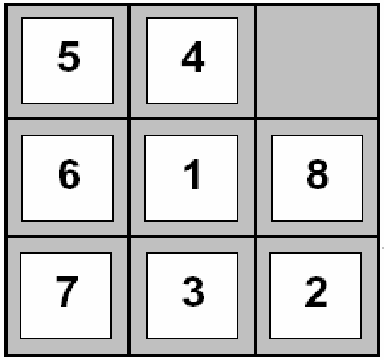
# Problem Formulation

- A problem is characterized with 4 parts
  - The initial state(s)
    - E.g., *In*(*Arad*)
  - A set of actions/operators
    - functions that map states to other states
    - A set of <*action*, *successor*> pairs generated by the successor function
    - E.g.,{<*Go*(*Sibiu*), *In*(*Sibiu*)>, <*Go*(*Zerind*), *In*(*Zerind*)>, …}
  - A goal test function
    - Check an explicit set of possible goal states
      - E.g.,{<*In*(*Bucharest*)>}
    - Or, could not be implicitly defined
      - E.g., Chess game → "checkmate"!
  - A path cost function (optional)
    - Assign a numeric cost to each path
    - E.g., c(*x, a, y*)
    - For some problems, it is of no interest!

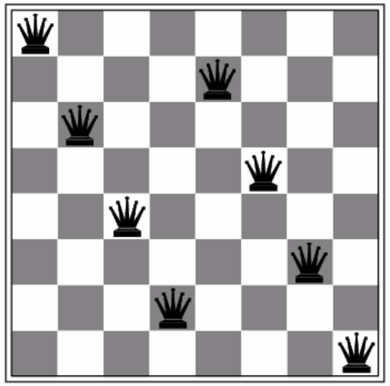
# What is a Solution?

- A sequence of actions that will transform the initial state(s) into the goal state(s), e.g.:
  - A path from one of the initial states to one of the goal states
  - Optimal solution: e.g., the path with lowest path cost

- Or sometimes just the goal state itself, when getting there is trivial



| | | |
|---|---|---|
| 5 | 4 | |
| 6 | 1 | 8 |
| 7 | 3 | 2 |



# Example: Romania

- Current town/state
  - Arad
- Formulated Goal
  - Bucharest
- Formulated Problem
  - World states: various cites
  - Actions: drive between cities
- Formulated Solution
  - Sequences of cities,
    e.g., Arad → Sibiu → Rimnicu Vilcea → Pitesti →Bucharest

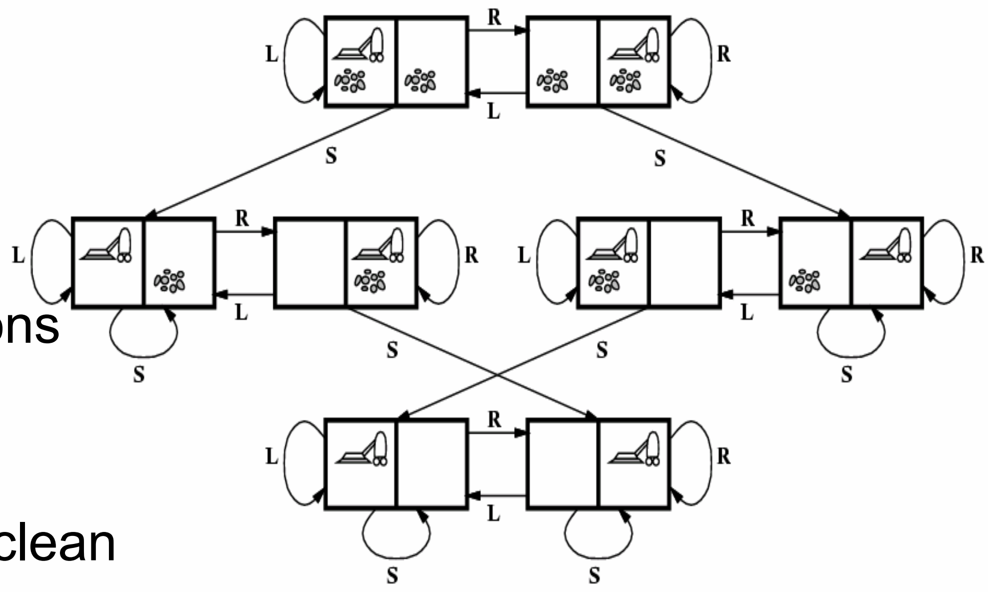
# Abstractions

- States and actions in the search space are abstractions of the agents actions and world states
  - State description
    - All irrelevant considerations are left out of the state descriptions
    - E.g., scenery, weather, …
  - Action description
    - Only consider the change in location
    - E.g., time & fuel consumption, degrees of steering, …

- So, actions carried out in the solution is easier than the original problem
  - Or the agent would be swamped by the real world

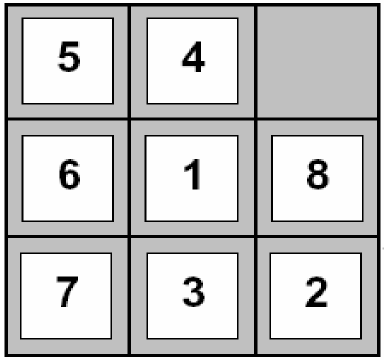
# Example Toy Problems

- The Vacuum World
  - States
    square num
    - $2x2^{2}=8$
    agent loc. dirty or not
  - Initial states
    - Any state can be
  - Successor function
    - Resulted from three actions (*Left, Right, Suck*)
  - Goal test
    - Whether all squares are clean
  - Path cost
    - Each step costs 1
    - The path cost is the number of steps in the path



# Example Toy Problems (cont.)

- The 8-puzzle
  - States
    - 9!=362,880 states
    - Half of them can reach the goal state (?)
  - Initial states
    - Any state can be
  - Successor function
    - Resulted from four actions, blank moves (*Left, Right, Up, Down*)
  - Goal test
    - Whether state matches the goal configuration
  - Path cost
    - Each step costs 1
    - The path cost is the number of steps in the path

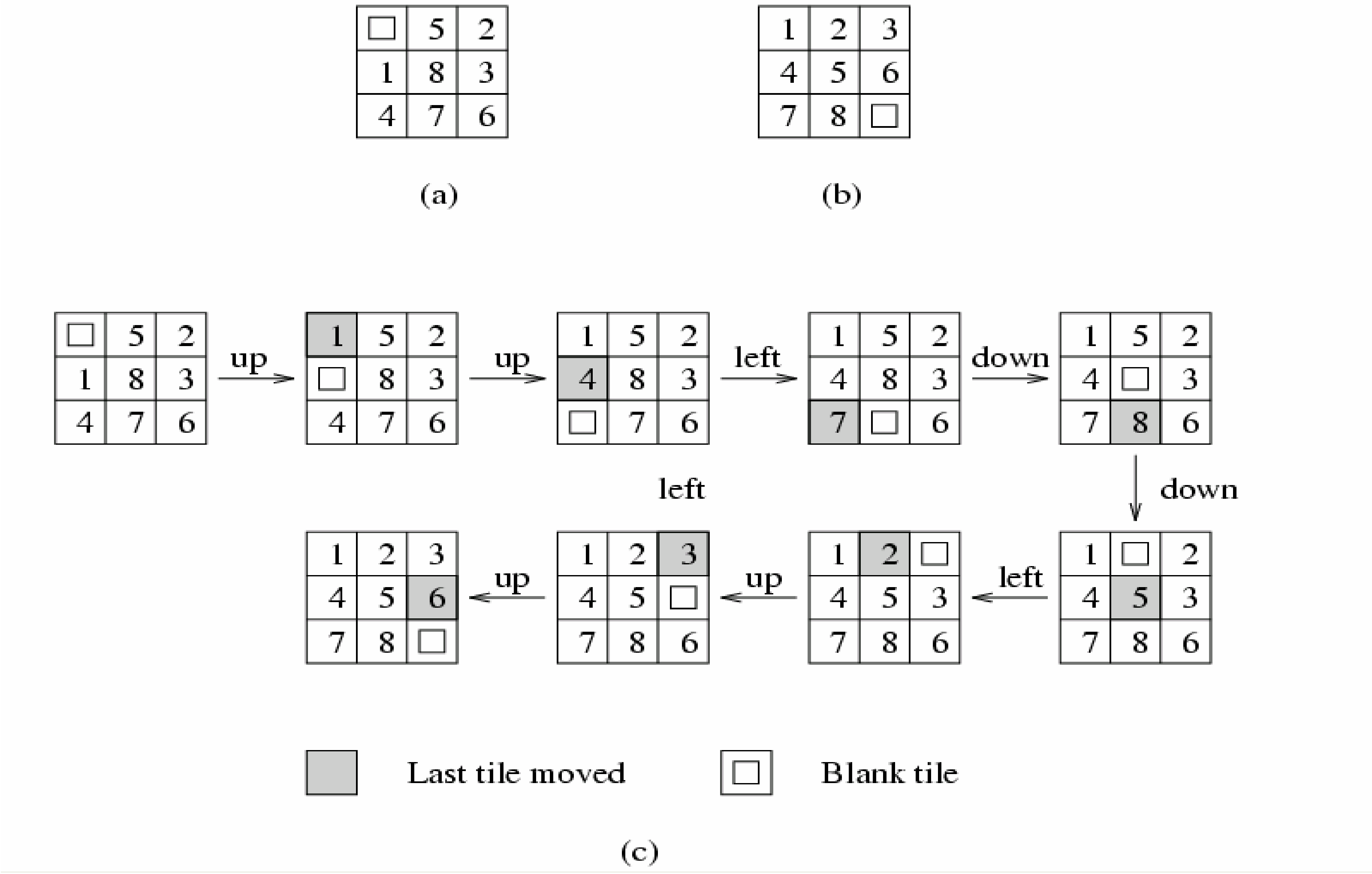


# Example Toy Problems (cont.)

- The 8-puzzle

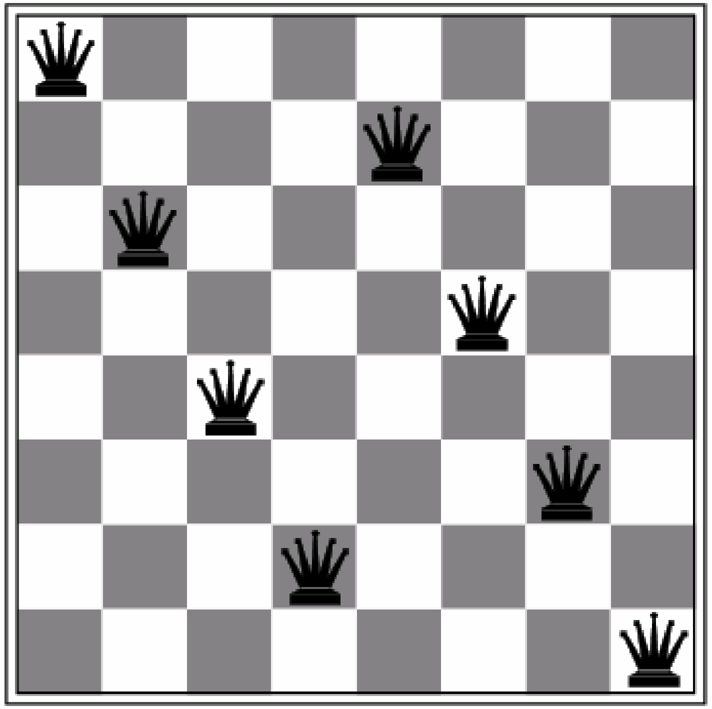
Start State

Goal State



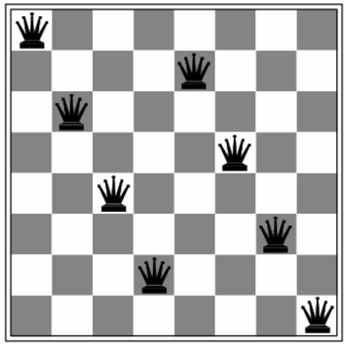
# Example Toy Problems (cont.)

- The 8-queens problem
  - Place 8 queens on a chessboard such that no queen attacks any other (no queen at the same row, column or diagonal)
  - Two kinds of formulation
    - Incremental or complete-state formulation



# Example Toy Problems (cont.)

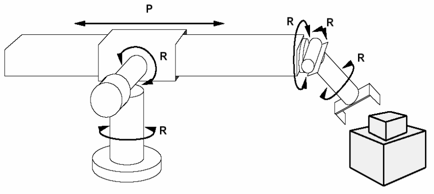
- Incremental formulation for the 8-queens problem
  - States
    - Any arrangement of 0~8 queens on the board is a state
    - Make 64x63x62….x57 possible sequences investigated
  - Initial states
    - No queens on the board
  - Successor function
    - Add a queen to any empty square
  - Goal test
    - 8 queens on the board, non attacked

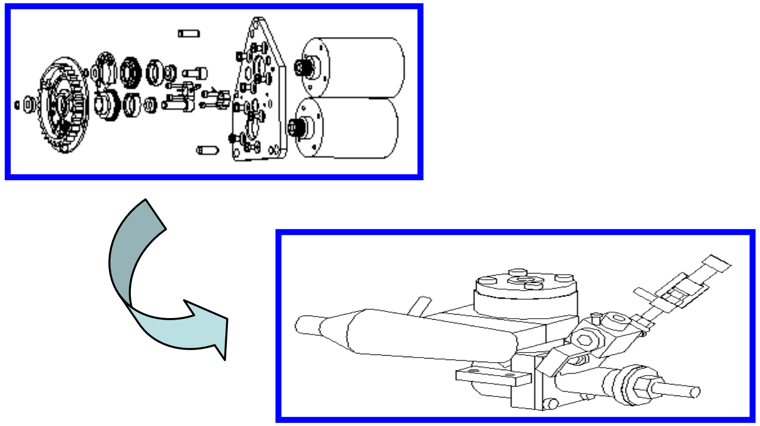


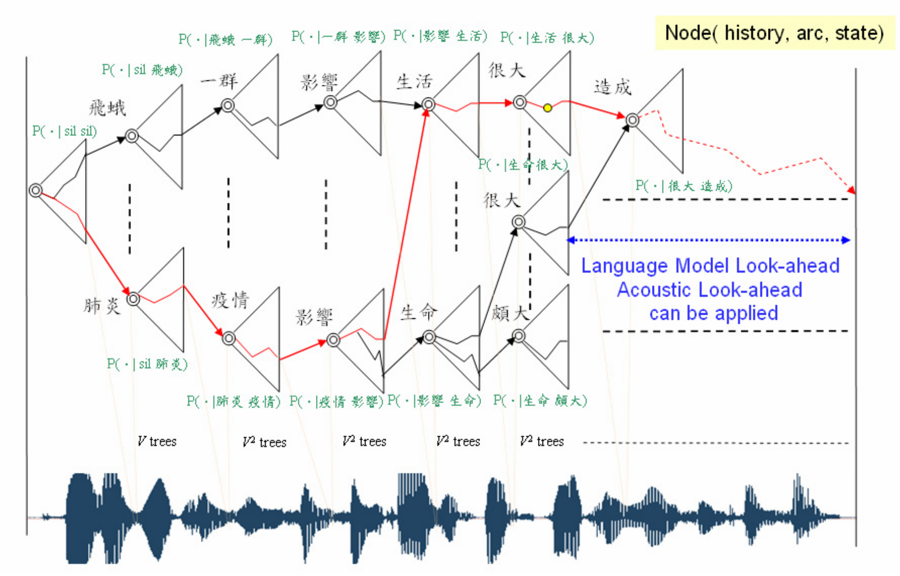
  - States
    - Arrangements of n queens, one per column in the leftmost n columns, non attacked
  - Successor function
    - Add a queen to any square in the leftmost empty column such that non queens attacked

# Example Problems

- Real-world Problems
  - Route-finding problem/touring problem
  - Traveling salesperson problem
  - VLSI layout
  - Robot navigation
  - Automatic assembly sequencing
  - Speech recognition
  - …..







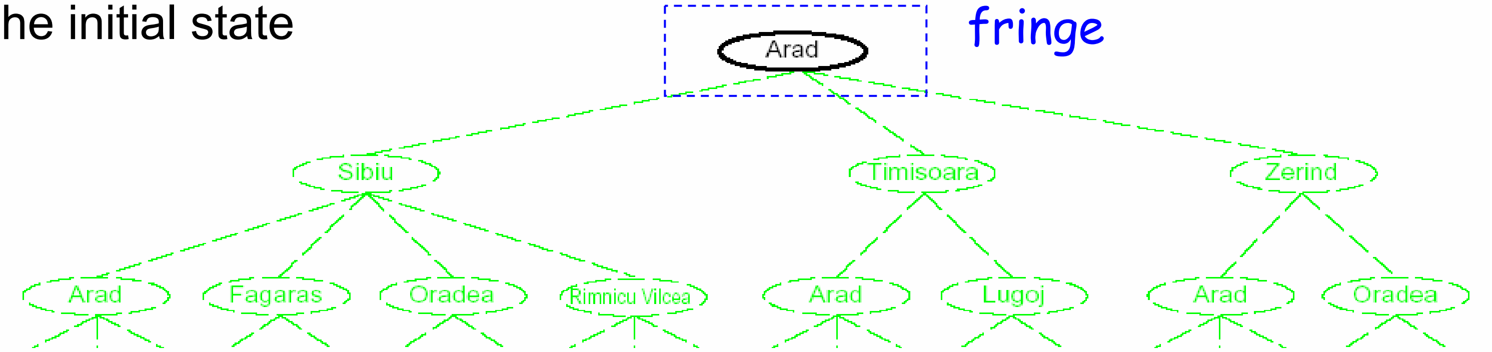
# State Space

- The representation of initial state(s) combined with the successor functions (actions) allowed to generate states which define the state space
  - The search tree
    - A state can be reached just from one path in the search tree
  - The search graph
    - A state can be reached from multiple paths in the search graph

- Nodes vs. States
  - Nodes are in the search tree/graph
  - States are in the *physical* state space
  - Many-to-one mapping
  - E.g., 20 states in the state space of the Romania map, but infinite number of nodes in the search tree

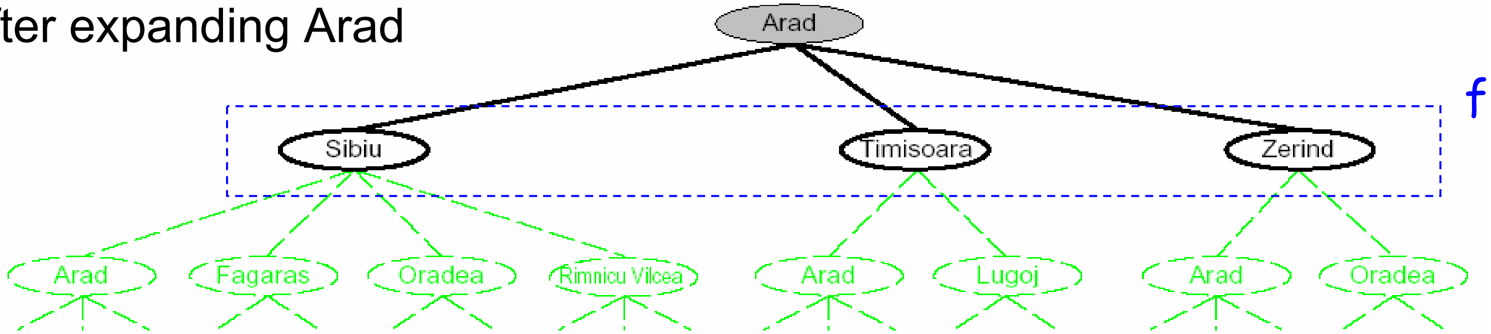
# State Space (cont.)

(a) The initial state



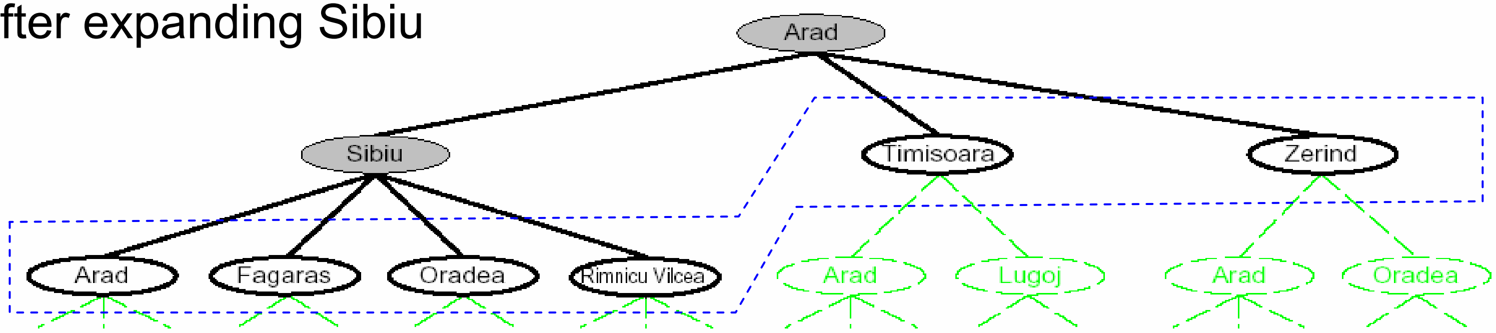
fringe

(b) After expanding Arad



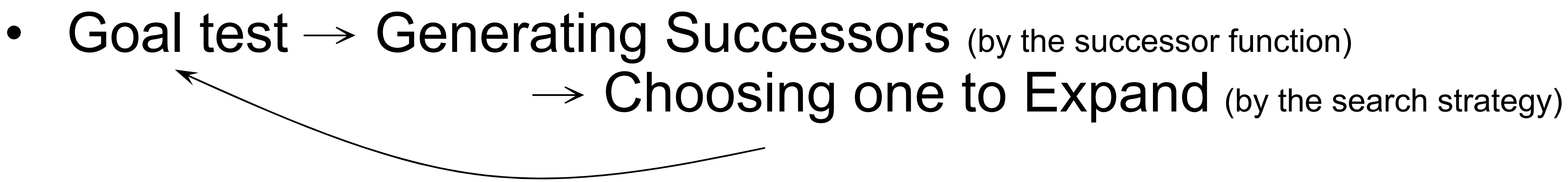
fringe

(b) After expanding Sibiu

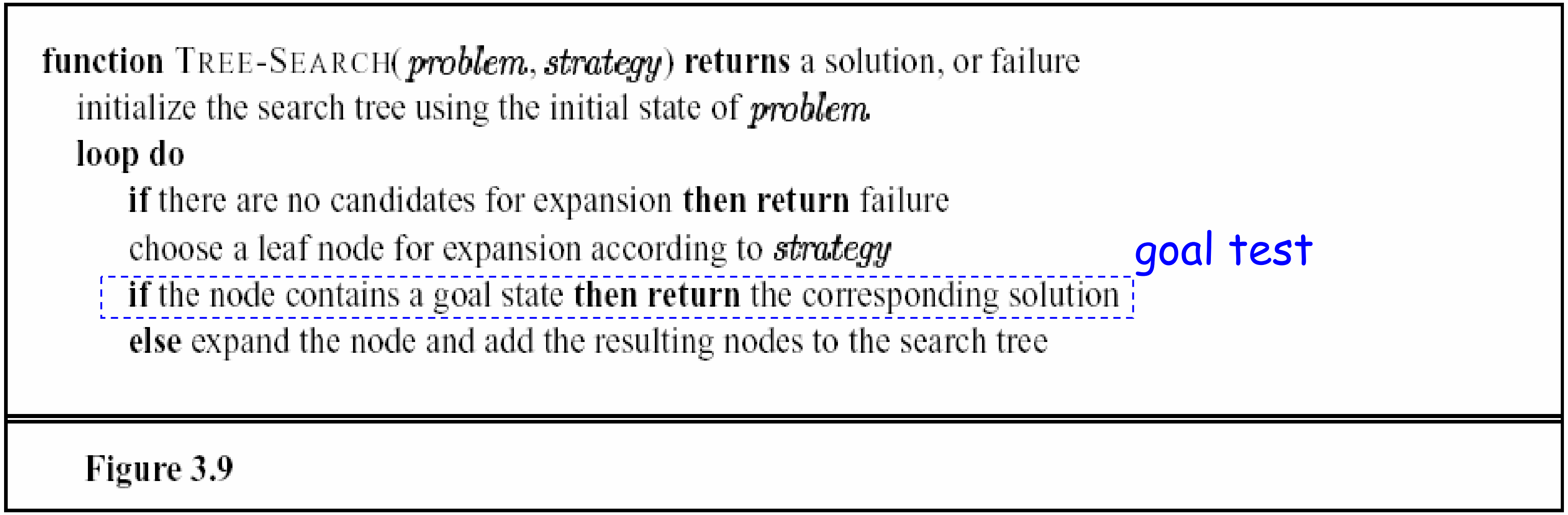


fringe

# State Space (cont.)



- Goal test → Generating Successors (by the successor function)
  → Choosing one to Expand (by the search strategy)
- Search strategy
  - Determine the choice of which state to be expanded next



```
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
```

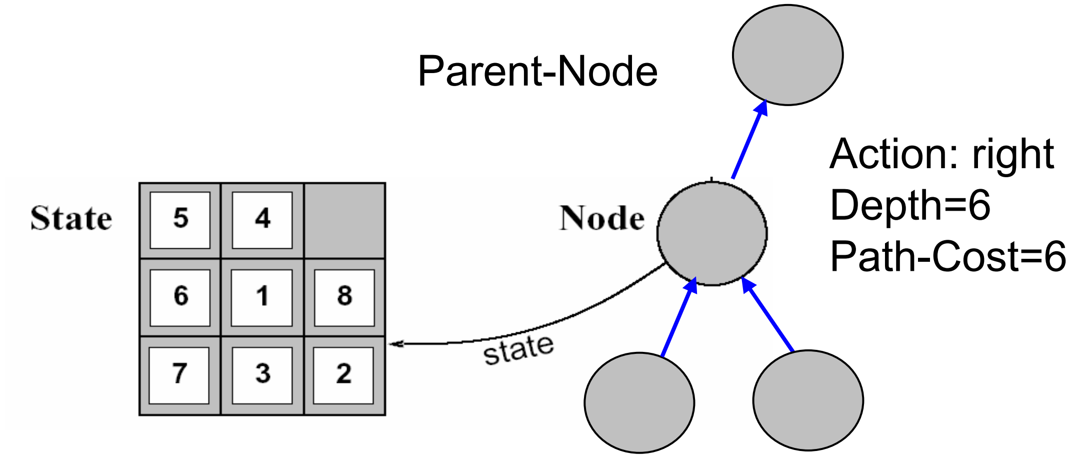
goal test

Figure 3.9

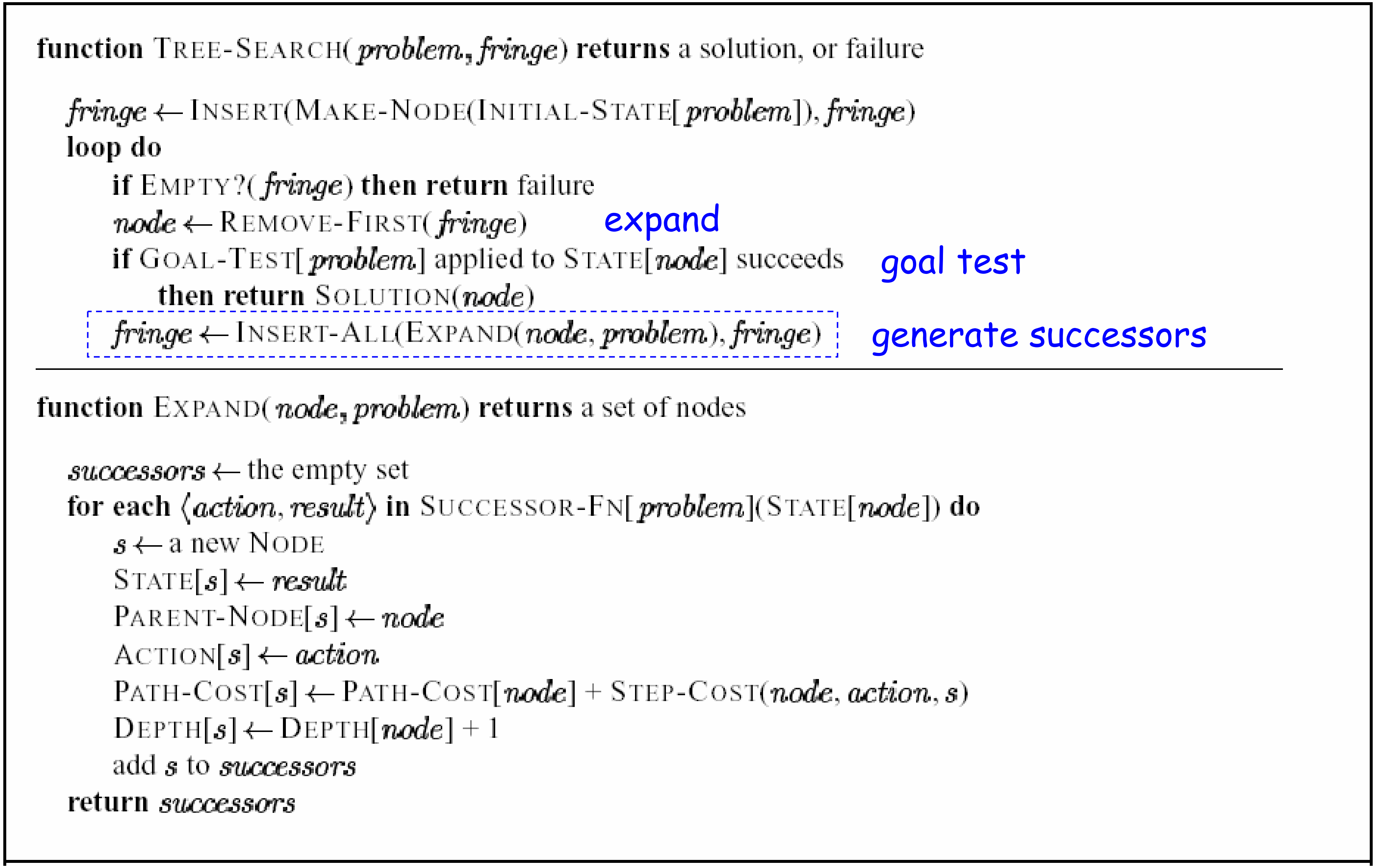
- Fringe
  - A set of (leaf) nodes generated but not expanded

# Representation of Nodes

- Represented by a data structure with 5 components
  - **State**: the state in the state space corresponded
  - **Parent-node**: the node in the search tree that generates it
  - **Action**: the action applied to the parent node to generate it
  - **Path-cost**: *g*(*n*), the cost of the path from the initial state to it
  - **Depth**: the number of steps from the initial state to it



# General Tree Search Algorithm



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

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem] applied to STATE[node] succeeds
          then return SOLUTION(node)
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```

expand

goal test

generate successors

# Judgment of Search Algorithms/Strategies

- Completeness
  - Is the algorithm guaranteed to find a solution when there is one ?
- Optimality
  - Does the strategy find the optimal solution ?
  - E.g., the path with lowest path cost
- Time complexity
  - How long does it take to find a solution ?
  - Number of nodes generated during the search
- Space complexity
  - How much memory is need to perform the search ?
  - Maximum number of nodes stored in memory

Measure of problem difficulty

# Judgment of Search Algorithms/Strategies (cont.)

- Time and space complexity are measured in terms of
  - *b* : maximum branching factors (or number of successors)
  - *d* : depth of the least-cost (shallowest) goal/solution node
  - *m*: Maximum depth of the any path in the state pace (may be $\infty$)

# Uninformed Search

- Also called blinded search
- No knowledge about whether one non-goal state is "more promising" than another

- Six search strategies to be covered
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limit search
  - Iterative deepening search
  - Bidirectional search

# Breadth-First Search (BFS)

- Select the *shallowest* unexpended node in the search tree for expansion
- Implementation
  - Fringe is a FIFO queue, i.e., new successors go at end

- Complete (if *b* is finite)

- Optimal (if unit step costs were adopted)
  - The *shallowest goal is not always the optimal one ?*

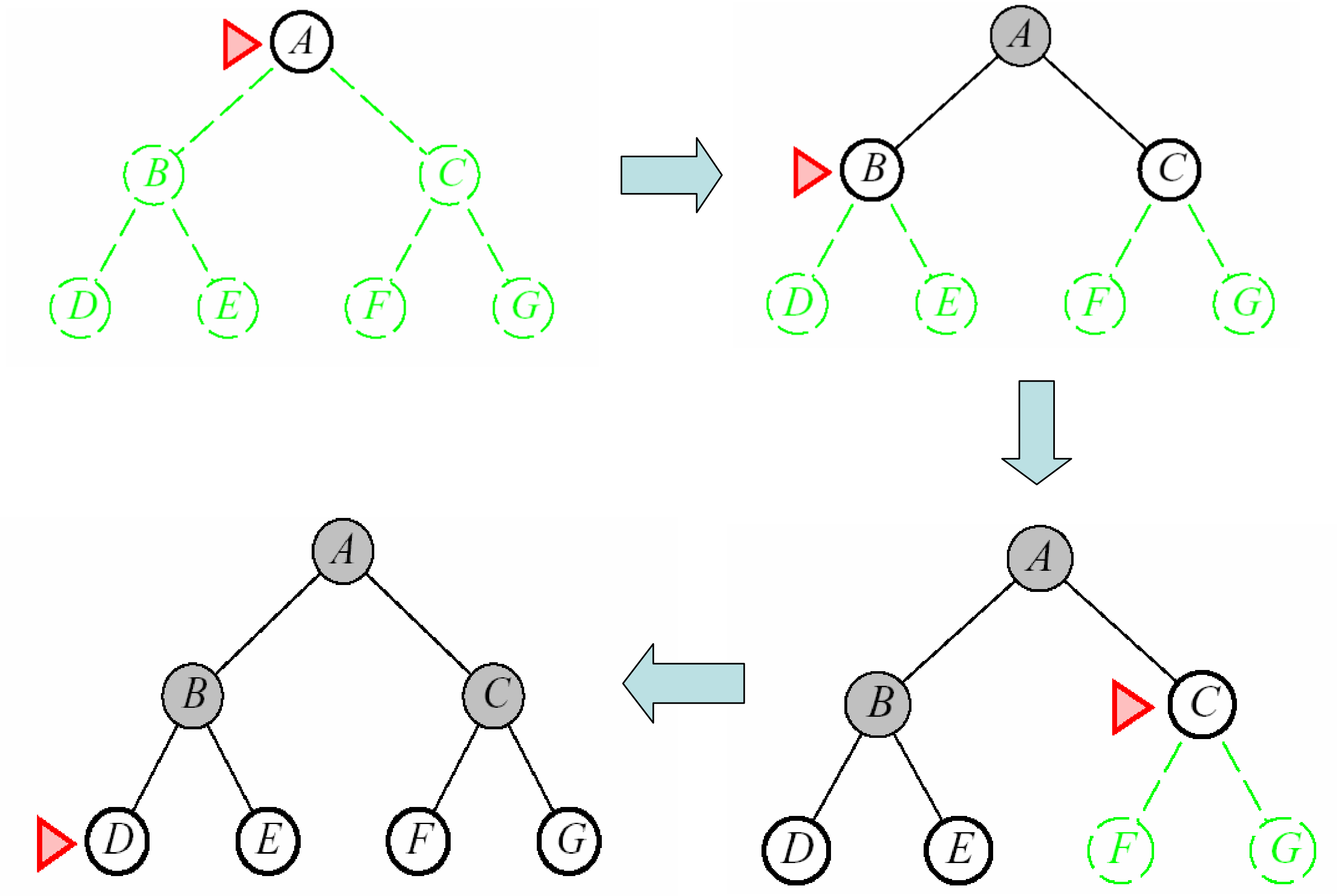
- Time complexity: $O(b^{d+1})$
  - $1+b+b^2+b^3+\ldots. +b^d+b(b^d-1)= O(b^{d+1})$

suppose that the solution is the right most one at depth *d*

Number of nodes generated

- Space complexity: $O(b^{d+1})$
  - Keep every node in memory

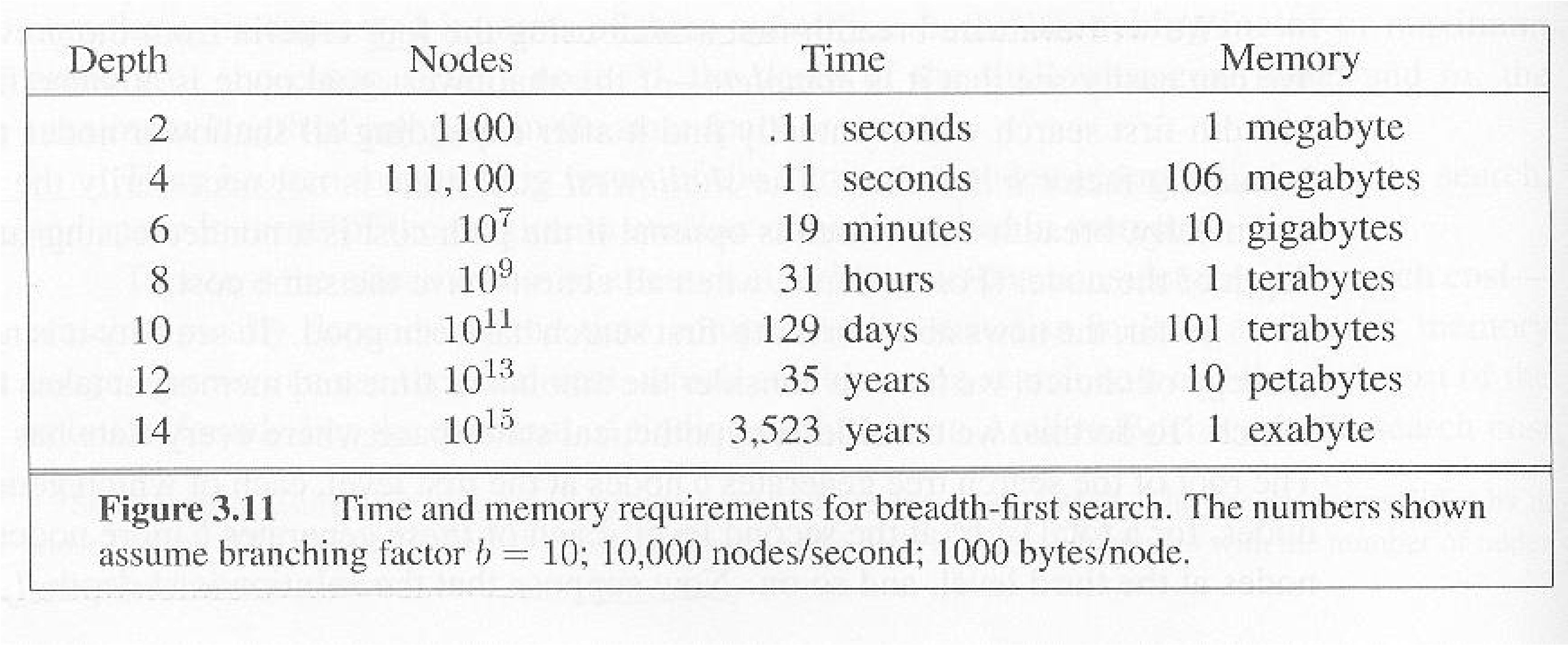
# Breadth-First Search (cont.)



For the same level/depth, nodes are expanded in a left-to-right manner.

# Breadth-First Search (cont.)

- Impractical for most cases
- Can be implemented with beam pruning
  - Completeness and Optimality will not be kept



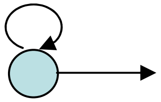
| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 1100 | .11 seconds | 1 megabyte |
| 4 | 111,100 | 11 seconds | 106 megabytes |
| 6 | $10^7$ | 19 minutes | 10 gigabytes |
| 8 | $10^9$ | 31 hours | 1 terabytes |
| 10 | $10^{11}$ | 129 days | 101 terabytes |
| 12 | $10^{13}$ | 35 years | 10 petabytes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

**Figure 3.11** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.

  - Memory is a bigger problem than execution time

# Uniform-Cost Search

- Similar to breadth first search but the node with lowest path cost expanded instead
- Implementation
  - Fringe is a queue ordered by path cost
- Complete and optimal if the path cost of each step was positive (and greater than a small positive constant $\varepsilon$ )
  - Or it will get suck in an infinite loop (e.g. *NonOp* action) with zero-cost action leading back to the same state



- Time and space complexity: $O(b^{\lceil C^* / \varepsilon \rceil})$
  - $C^*$ is the cost of the optimal solution

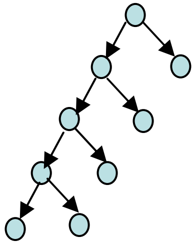
# Depth-First Search (DFS)

- Select the **deepest** unexpended node in the current fringe of the search tree for expansion

- Implementation
  - Fringe is a LIFO queue, i.e., new successors go at front

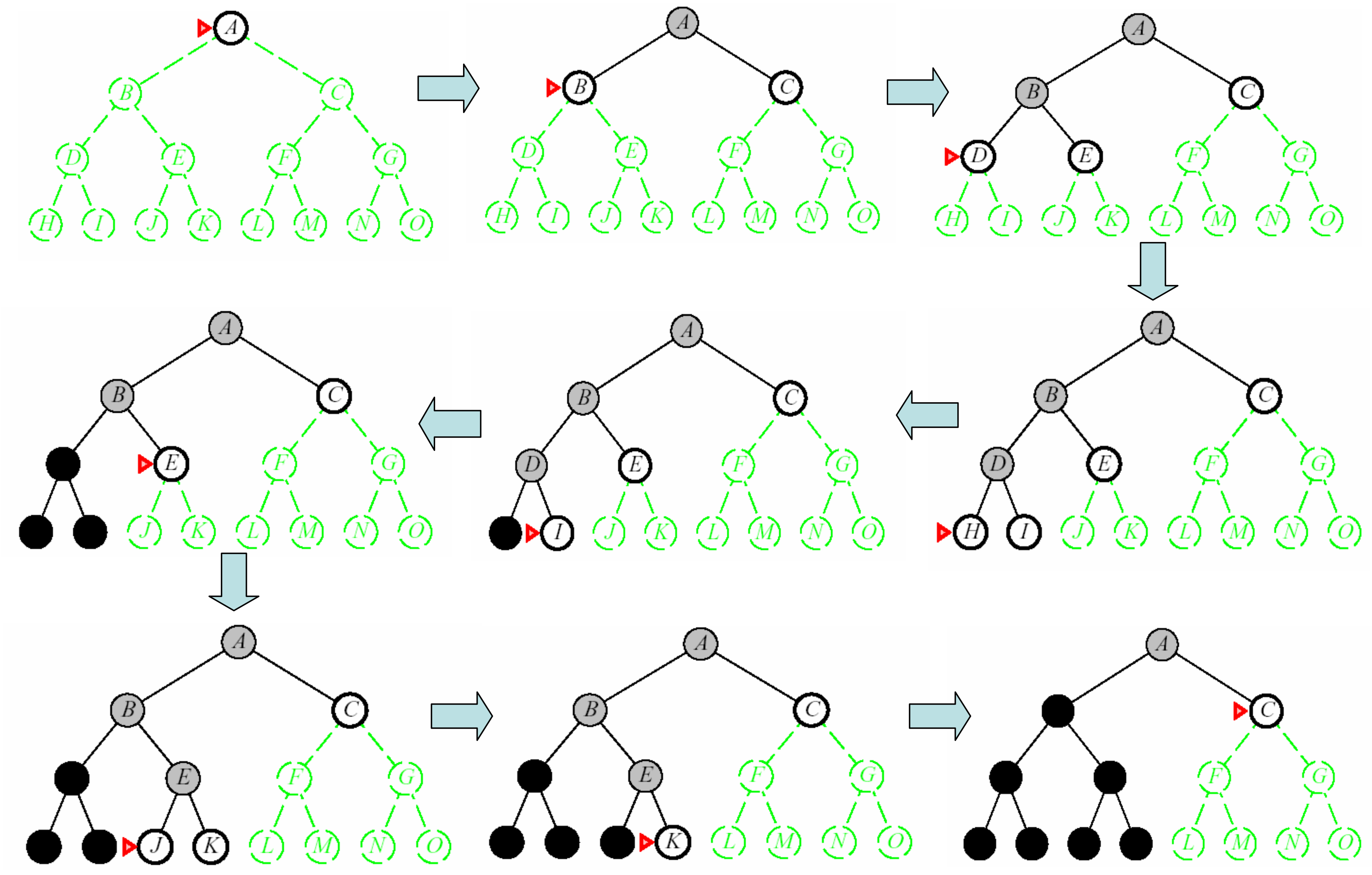
- Neither complete nor optimal

- Time complexity is $O(b^m)$
  - *m* is the maximal depth of any path in the state space

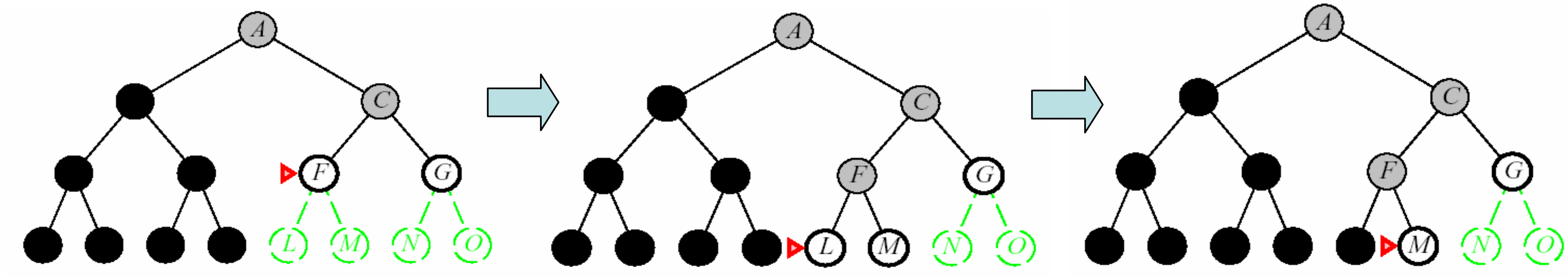


- Space complexity is $O(bm) \rightarrow bm+1$
  - Linear space !

# Depth-First Search (cont.)



# Depth-First Search (cont.)



- Would make a wrong choice and get suck going down infinitely

# Depth-First Search (cont.)

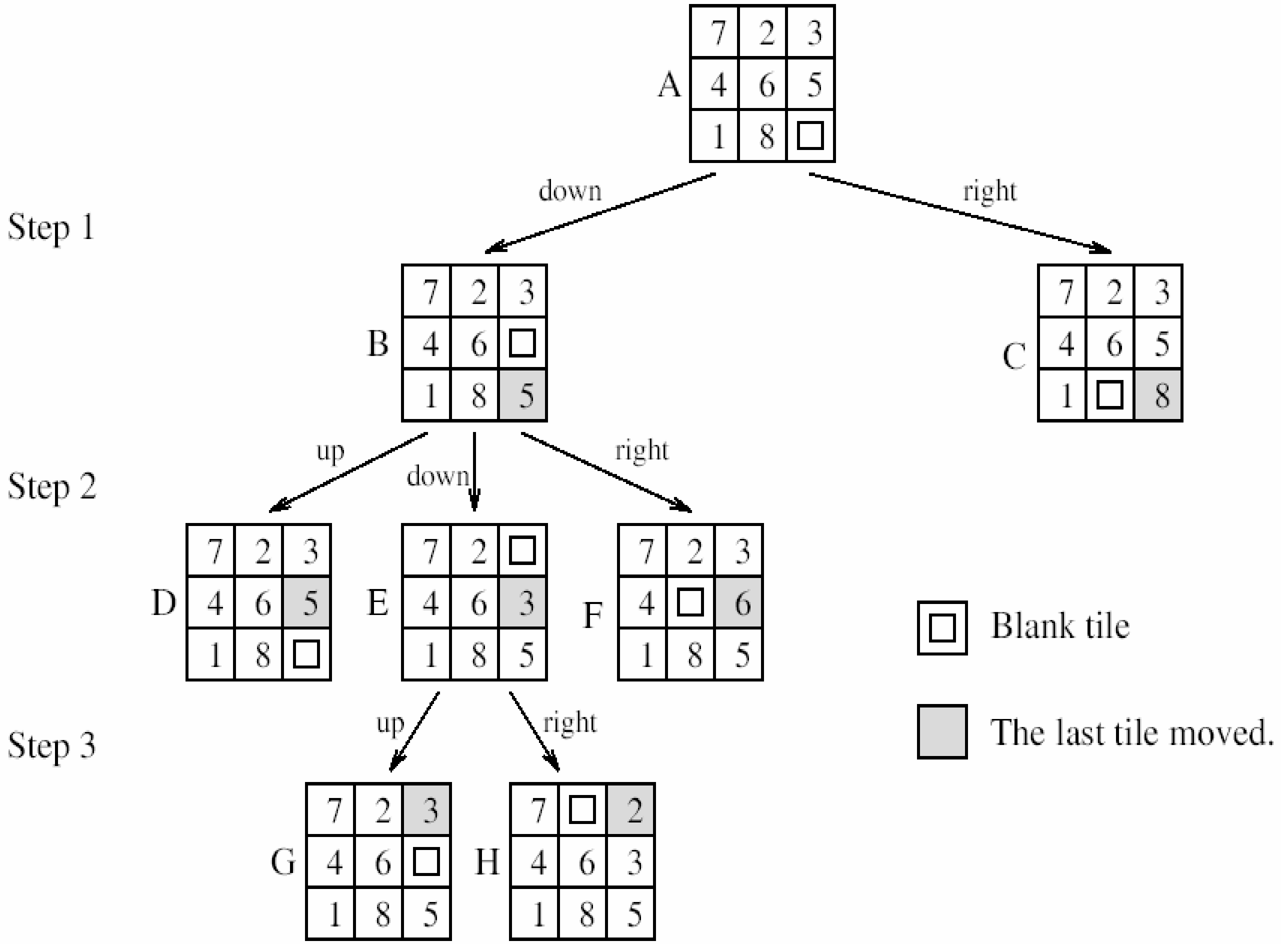
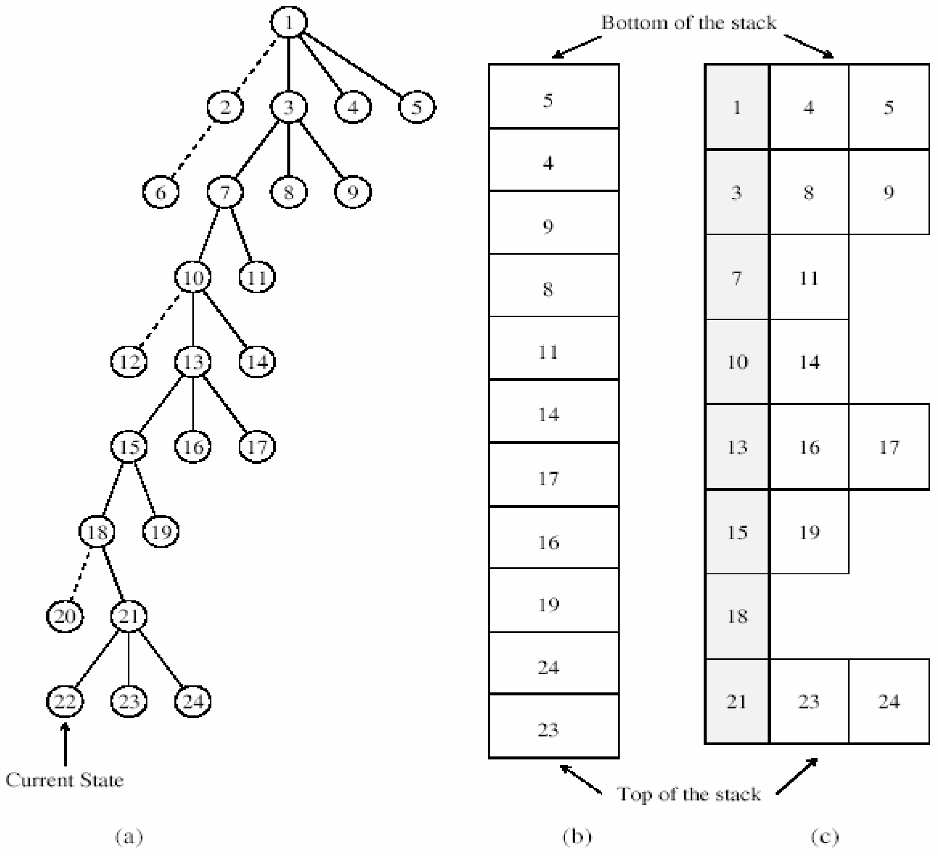


**Figure 11.4** States resulting from the first three steps of depth-first search applied to an instance of the 8-puzzle.

# Depth-First Search (cont.)

Two variants of stack implementation

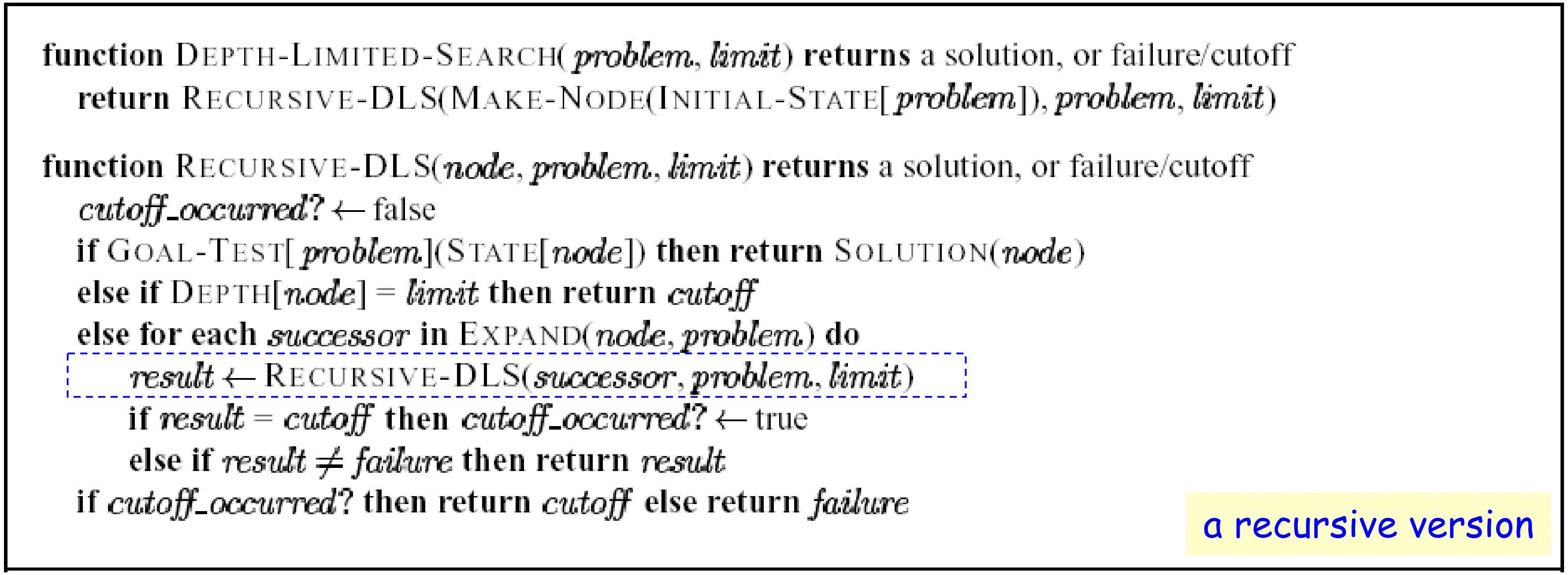


Termed as **backtracking search** in textbook

**Figure 11.5** Representing a DFS tree: (a) the DFS tree; Successor nodes shown with dashed lines have already been explored; (b) the stack storing untried alternatives only; and (c) the stack storing untried alternatives along with their parent. The shaded blocks represent the parent state and the block to the right represents successor states that have not been explored.

# Depth-limited Search (cont.)

- Depth-first search with a predetermined depth limit *l*
  - Nodes at depth *l* are treated as if they have no successors
- Neither complete nor optimal
- Time complexity is $O(b^l)$
- Space complexity is $O(bl)$

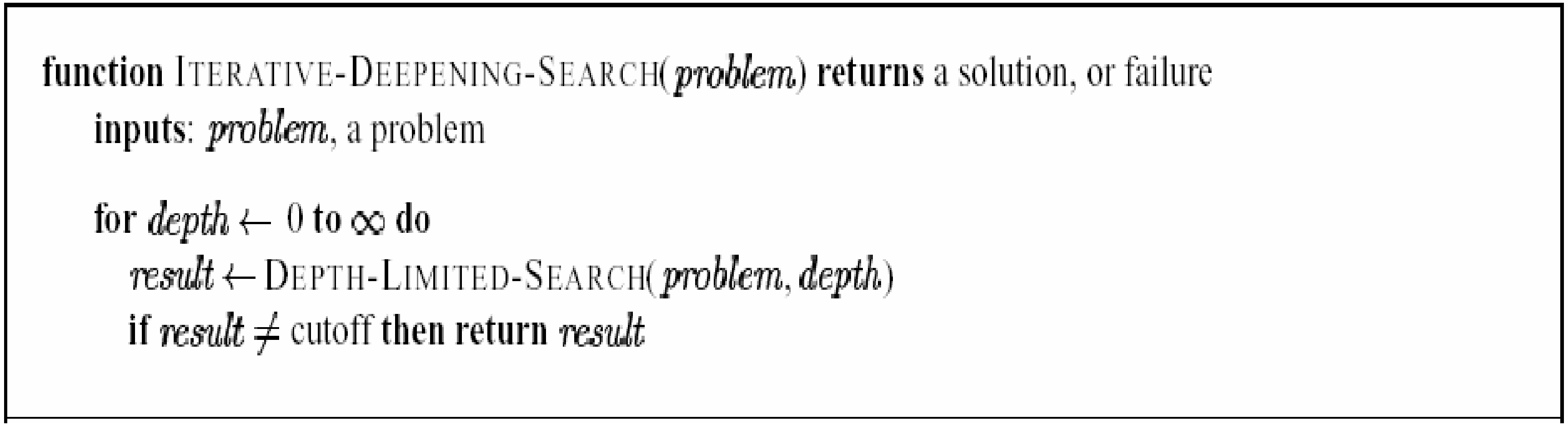


```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   cutoff_occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
       result ← RECURSIVE-DLS(successor, problem, limit)
       if result = cutoff then cutoff_occurred? ← true
       else if result ≠ failure then return result
   if cutoff_occurred? then return cutoff else return failure
```

a recursive version

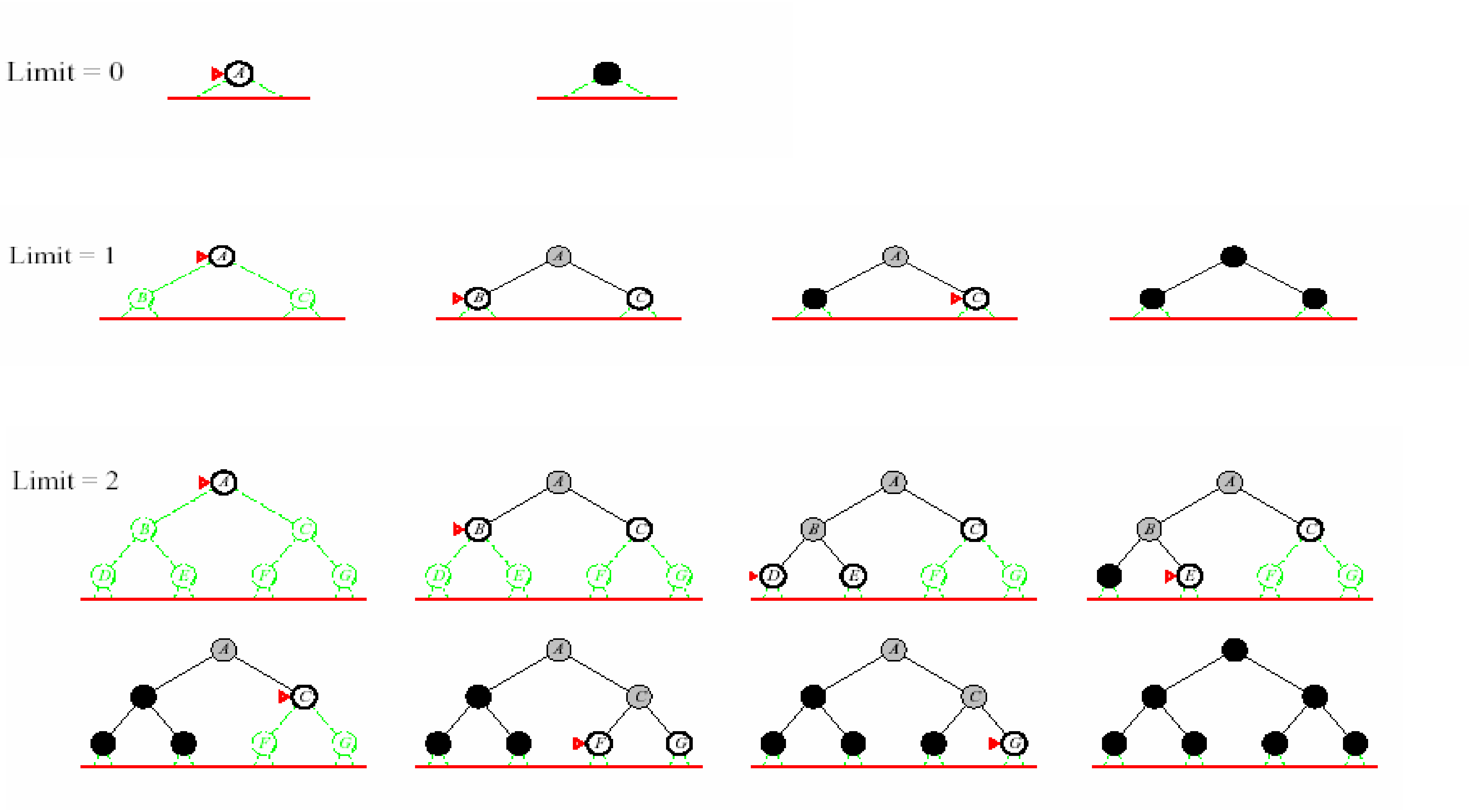
# Iterative Deepening Depth-First Search

- Also called Iterative Deepening Search (IDS)
- Iteratively call depth-first search by gradually increasing the depth limit $l$ ($l$ = 0, 1, 2, ..)
  - Go until a shallowest goal node is found at a specific depth $d$
- Nodes would be generated multiple times
  - The number of nodes generated : $N(\text{IDS})=(d)b+(d-1)b^2+\ldots+(1)\,b^d$
  - Compared with BFS: $N(\text{BFS})=b+b^2+\ldots+b^d+(b^{d+1}-b)$

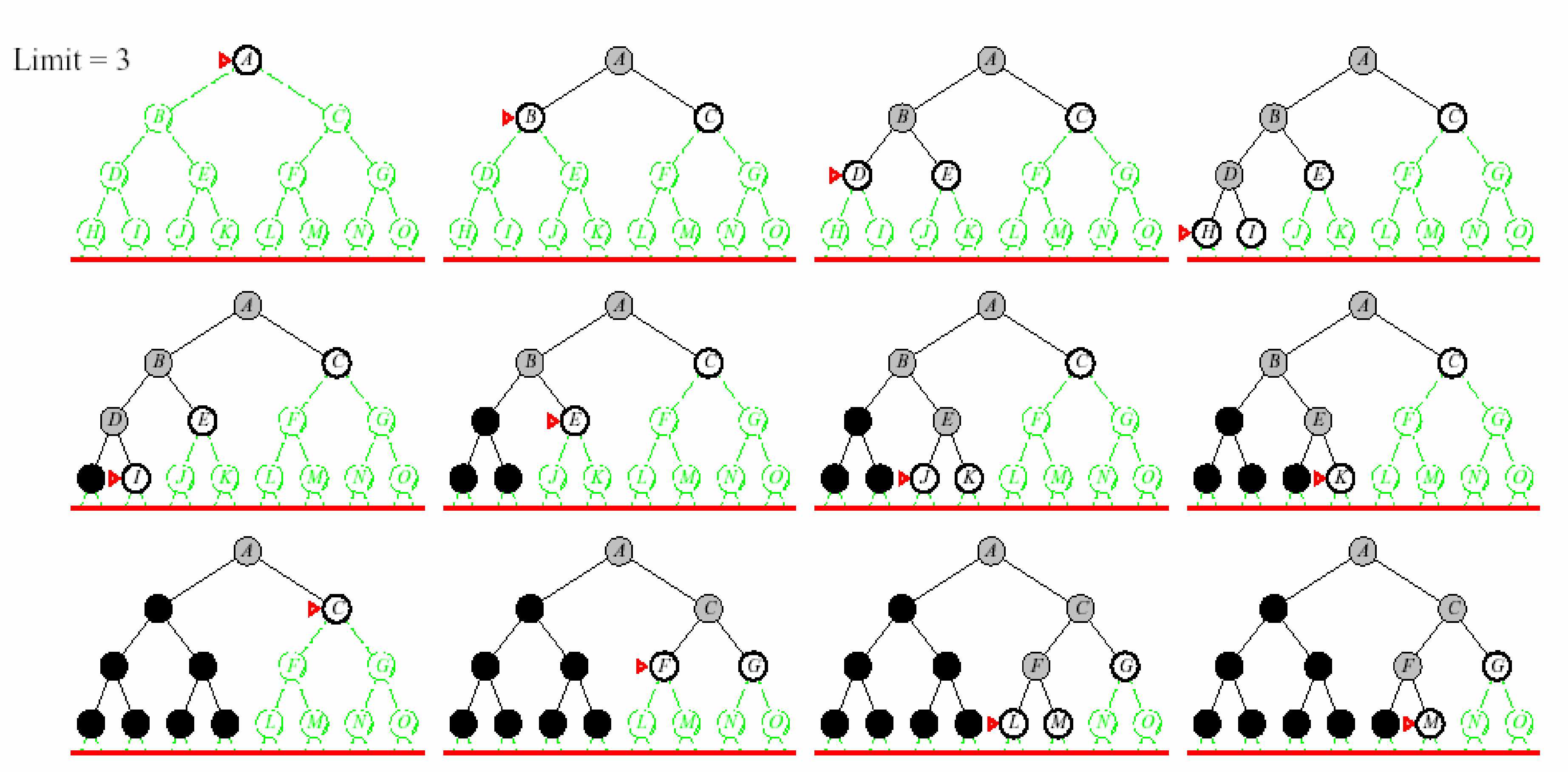


```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

# Iterative Deepening Depth-First Search (cont.)



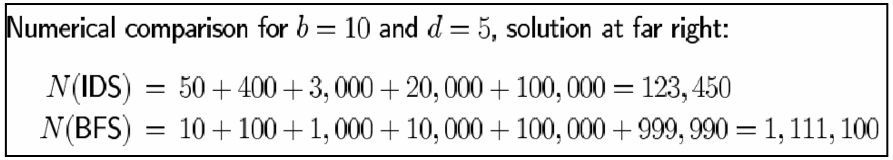
# Iterative Deepening Depth-First Search (cont.)



– Explore a complete layer if nodes at each iteration before going on next layer (analogous to BFS)

# Iterative Deepening Depth-First Search (cont.)

- Complete (if *b* is finite)
- Optimal (if unit step costs are adopted)
- Time complexity is $O(b^d)$
- Space complexity is $O(bd)$

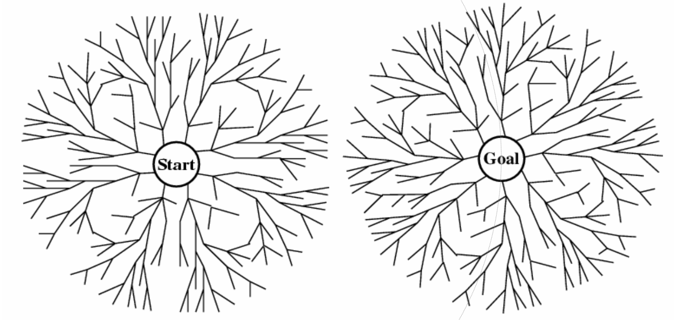


Numerical comparison for $b = 10$ and $d = 5$, solution at far right:

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

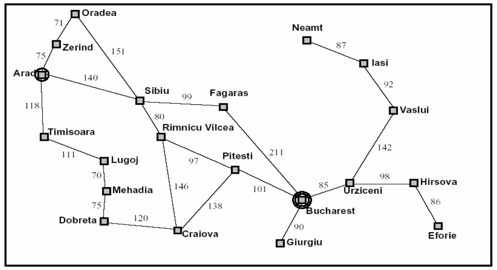
IDS is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

# Bidirectional Search



- Run two simultaneous search
  - One BFS forward from the initial state
  - The other BFS backward from the goal
  - Stop when two searches meet in the middle
    - Both searches check each node before expansion to see if it is in the fringe of the other search tree
    - How to find the predecessors?

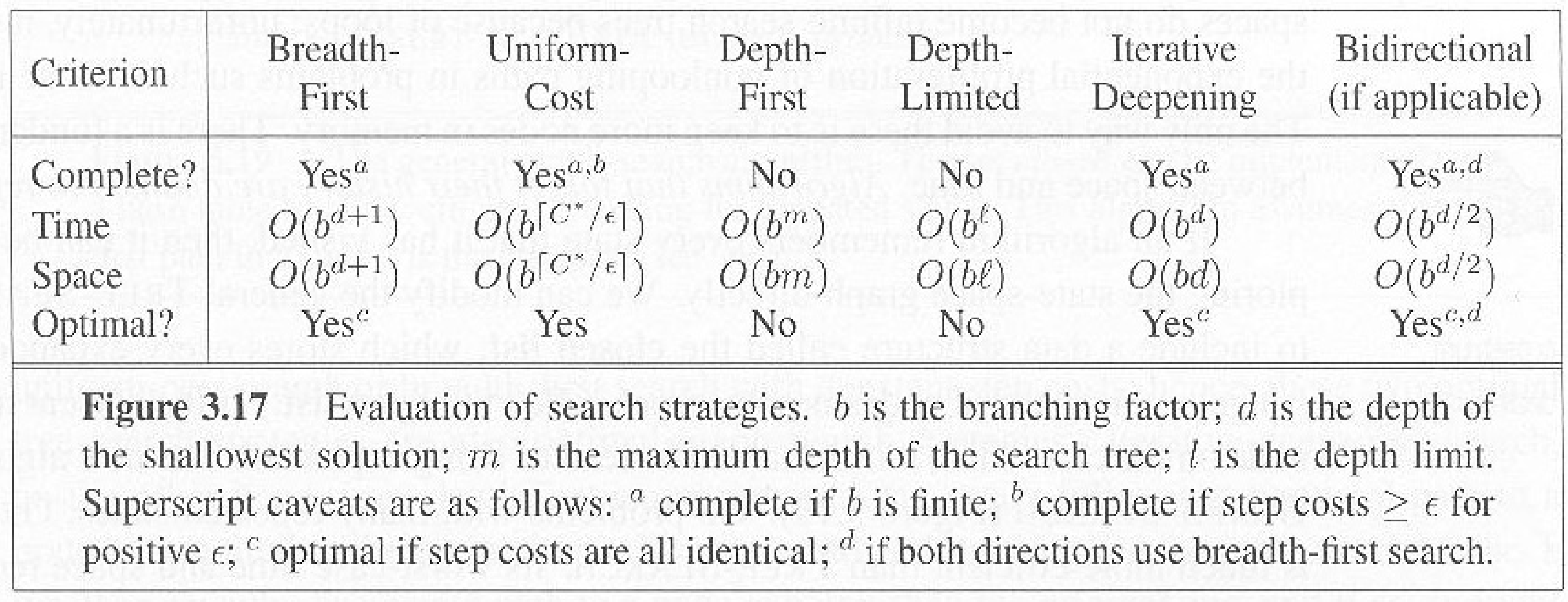
- Can enormously reduce time complexity: $O(b^{d/2})$



- But requires too much space: $O(b^{d/2})$

- How to efficiently compute the predecessors of a node in the backward pass

# Comparison of Uniformed Search Strategies



| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes$^{a}$ | Yes$^{a,b}$ | No | No | Yes$^{a}$ | Yes$^{a,d}$ |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^{m})$ | $O(b^{\ell})$ | $O(b^{d})$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes$^{c}$ | Yes | No | No | Yes$^{c}$ | Yes$^{c,d}$ |

**Figure 3.17** Evaluation of search strategies. $b$ is the branching factor; $d$ is the depth of the shallowest solution; $m$ is the maximum depth of the search tree; $l$ is the depth limit. Superscript caveats are as follows: $^{a}$ complete if $b$ is finite; $^{b}$ complete if step costs $\geq \epsilon$ for positive $\epsilon$; $^{c}$ optimal if step costs are all identical; $^{d}$ if both directions use breadth-first search.

# Avoiding Repeated States

- Repeatedly visited a state during search
  - Never come up in some problems if their search space is just a tree (where each state can only by reached through one path)
  - Unavoidable in some problems

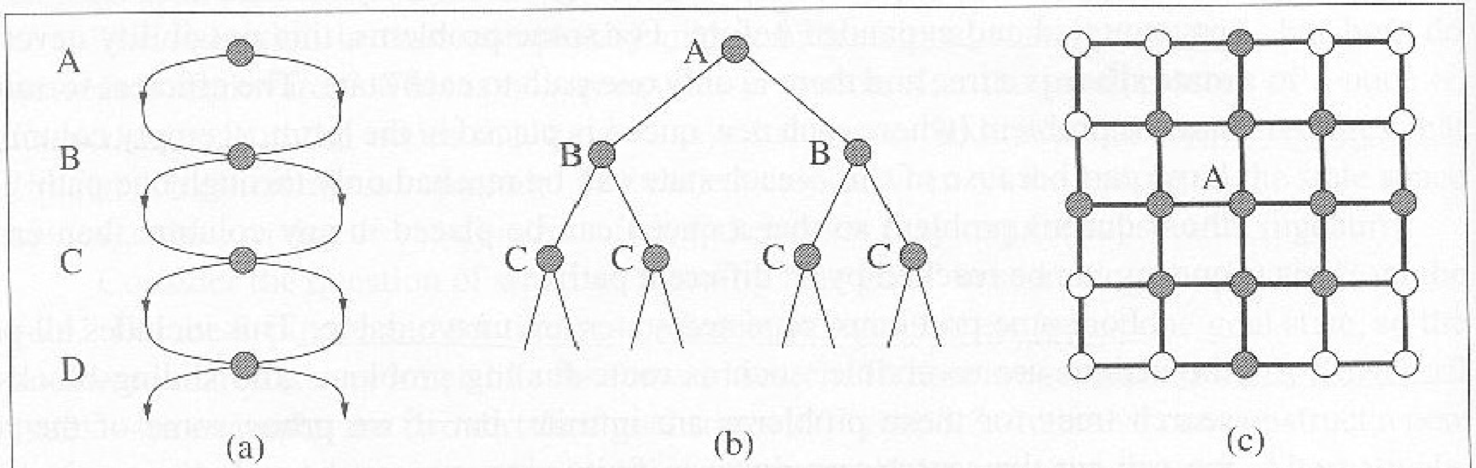
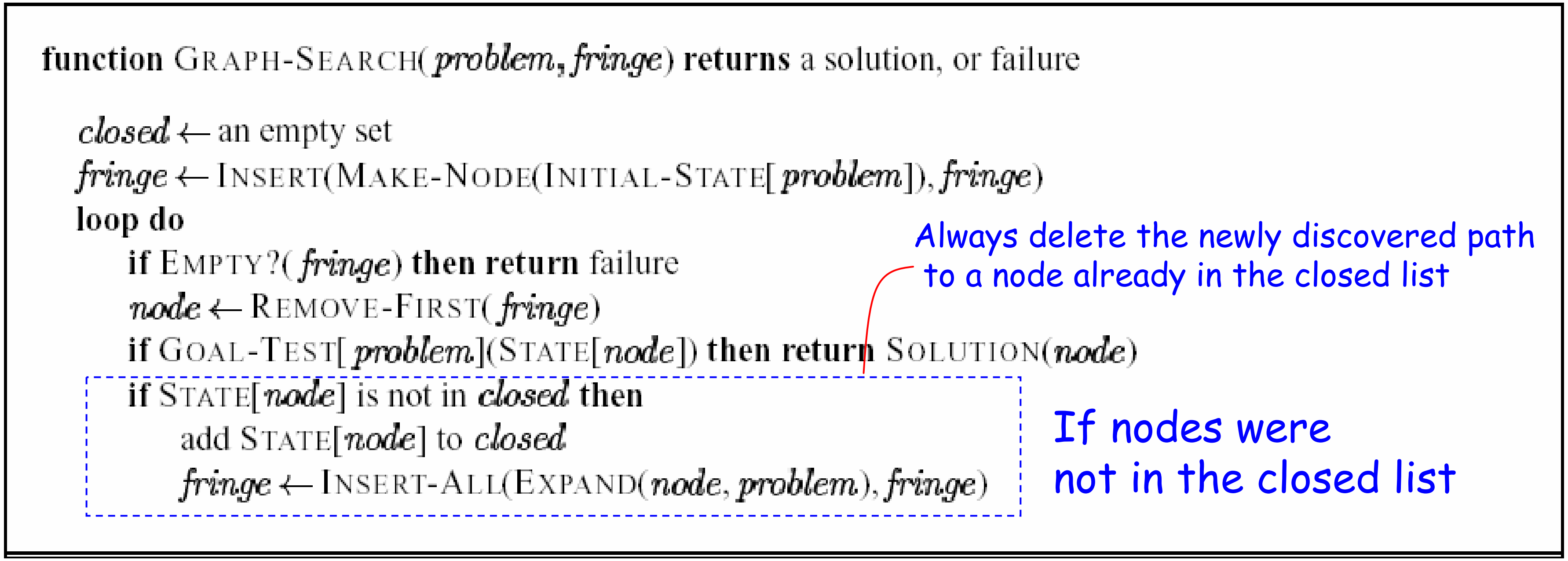


**Figure 3.18** State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where $d$ is the maximum depth. (b) The corresponding search tree, which has $2^d$ branches corresponding to the $2^d$ paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

# Avoiding Repeated States (cont.)

- Remedies
  - Delete looping paths
  - Remember every states that have been visited
    - The closed list (for expanded nodes) and open list (for unexpanded nodes)
    - If the current node matches a node on the closed list, discarded instead of being expanded (missing an optimal solution ?)



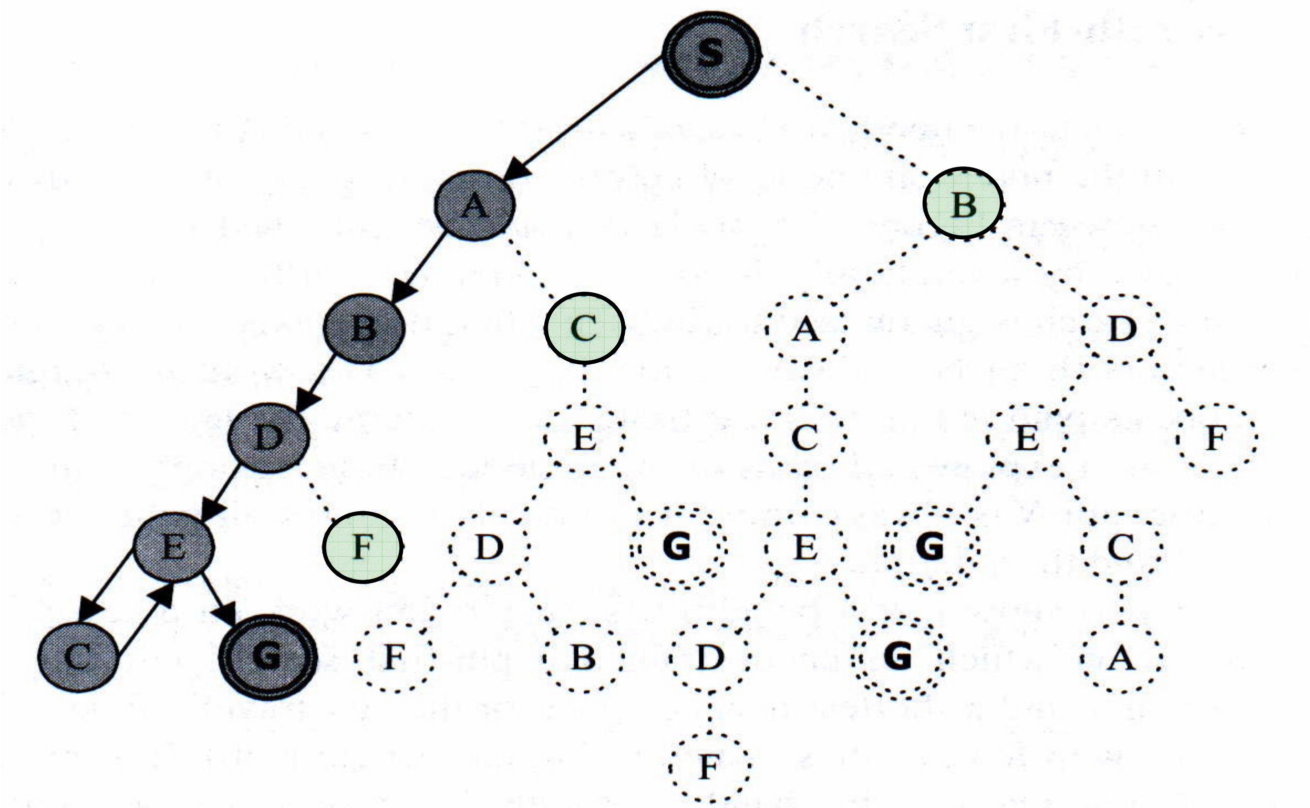
```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Always delete the newly discovered path to a node already in the closed list

If nodes were not in the closed list

# Avoiding Repeated States (cont.)

- Example: Depth-First Search



  - Detection of repeated nodes along a path can avoid looping
  - Still can’t avoid exponentially proliferation of nonlooping paths

# Searching with Partial Information

- Incompleteness: knowledge of states or actions are incomplete
  - Can't know which state the agent is in (the environment is partially observable)

  - Can't calculate exactly which state results from any sequence of actions (the actions are uncertain)

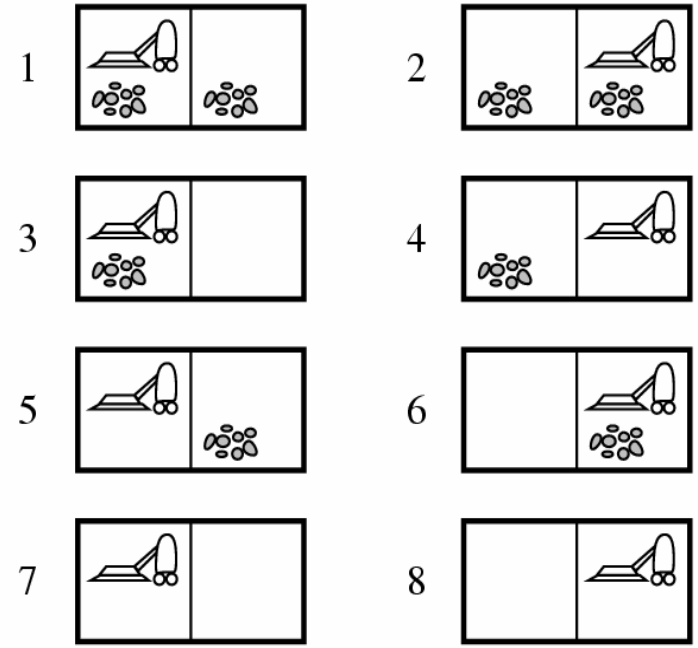
- Kinds of Incompleteness
  - Sensorless problems
  - Contingency problems
  - Exploration problems

# Sensorless Problems

- The agent has no sensors at all
  - It could be in one of several possible initial states
  - Each action could lead to one of several possible states

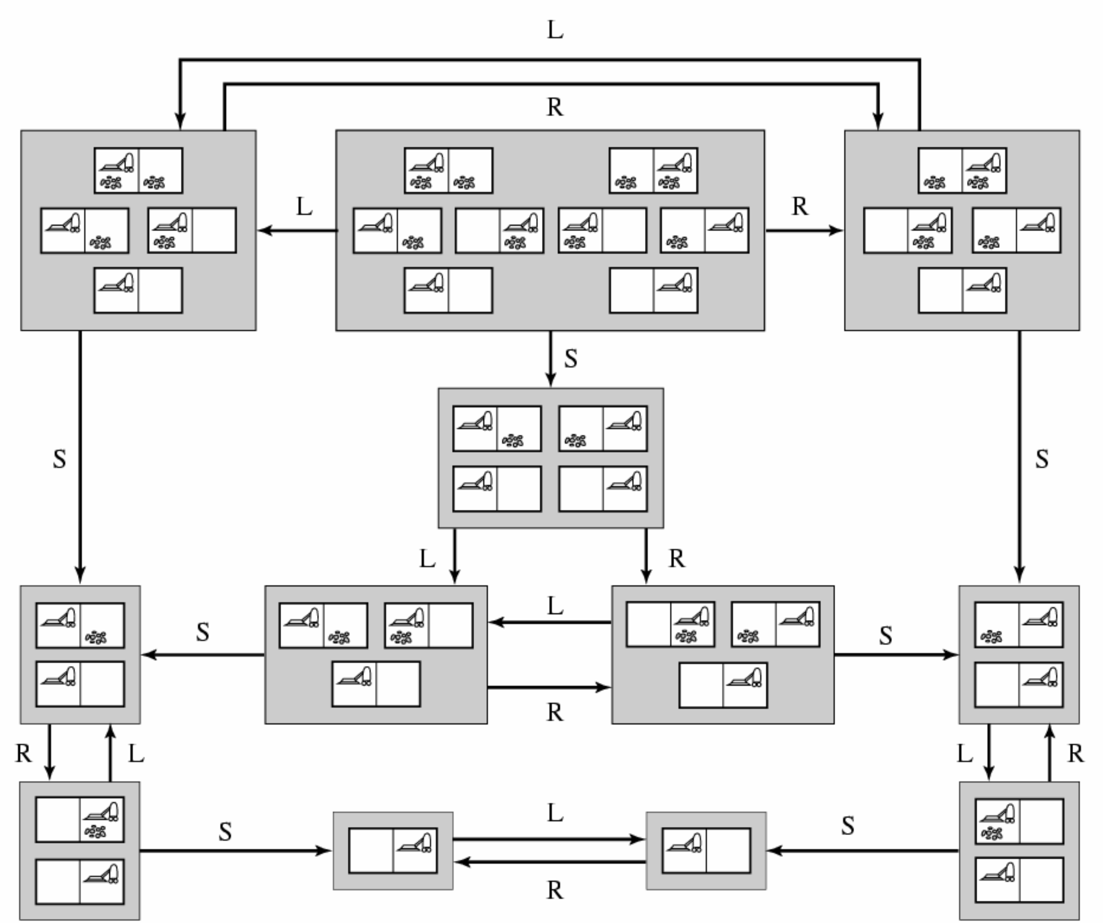
- Example: the vacuum world has 8 states
  - Three actions – *Left, Right, Suck*
  - Goal: clean up all the dirt and result in states 7 and 8
  - Original task environment – *observable, deterministic*

  - What if the agent is partially sensorless
    - Only know the effects of it actions



# Sensorless Problems (cont.)

- Belief State Space
  - A belief state is a set of states that represents the agent's current belief about the possible physical states it might be in



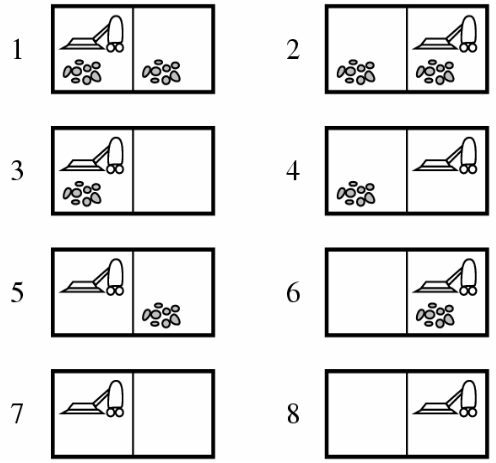
# Sensorless Problems (cont.)

- Actions applied to a belief state are just the unions of the results of applying the action to each physical state in the belief state
- A solution is a path that leads to a belief state all of whose elements are goal states

# Contingency Problems

- If the environment is partially observable or if actions are uncertain, then the agent's percepts provide new information after each action

- Murphy Law: If anything can go wrong, it will!
  - E.g., the suck action sometimes deposits dirt on the carpet but there is no dirt already
    - Agent perform the *Suck* operation in a clean square



# Exploration Problems

- The states and actions of the environment are unknown
- An extreme case of contingency problems