Chapter 3:
Solving Problems by Searching

Problem Solving Agent

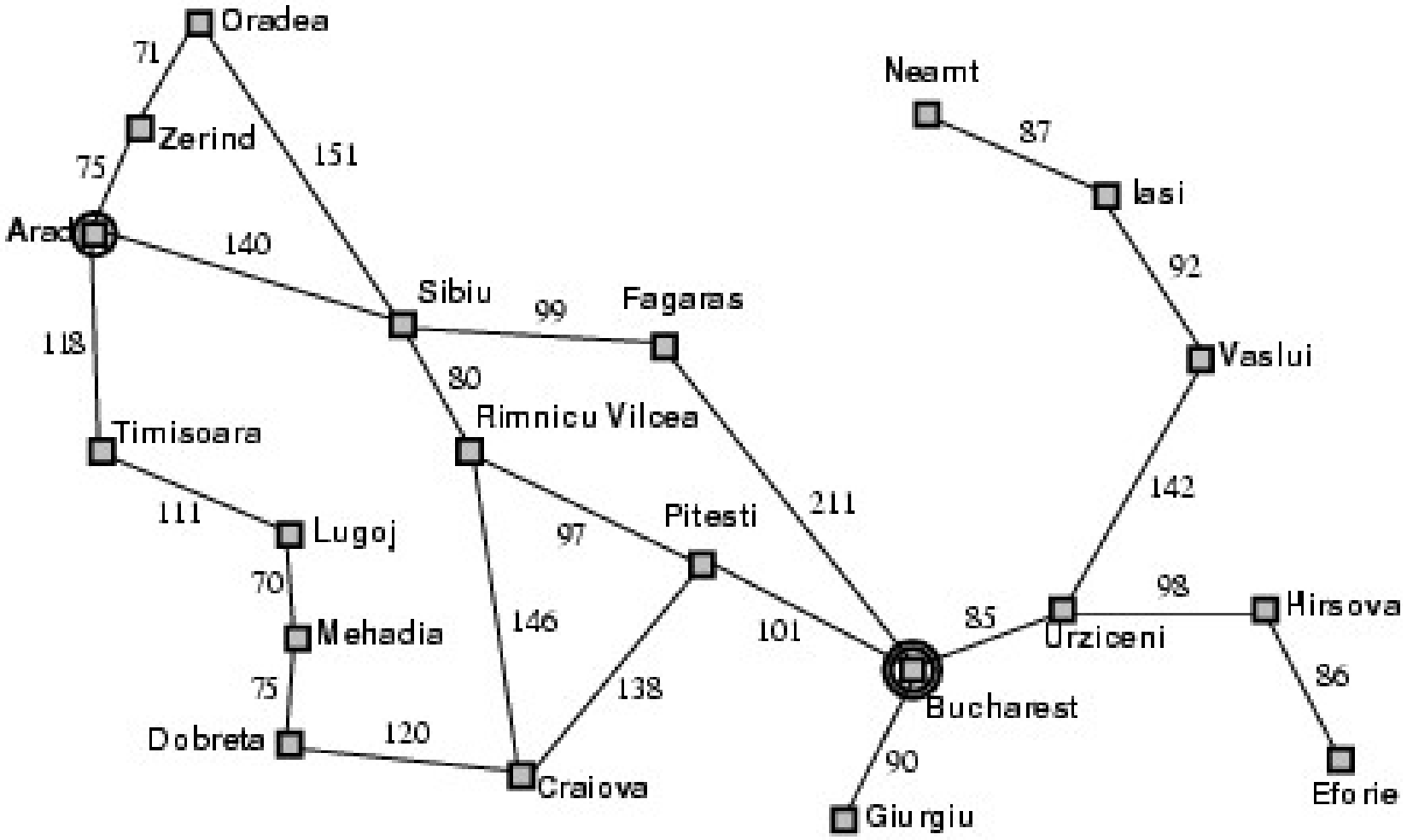
- Problem-solving agent: a type of goal-based agent
  - Decide what to do by finding sequences of actions that lead to desirable states
- Goal formulation: based on current situation and agent's performance measure
- Problem formulation: deciding what actions and states to consider, given a goal
- The process of looking for such a sequence of actions is called search


Satu Mare
MARAMUREŞ
SATU MARE
MARAMUREŞ
Câmpulung Moldovenesc
SUCEAVA
BOTOŞANI
Suceava
BUCOVINA
Fălticeni
SĂLAJ
Oradea
BISTRIŢA-NĂSĂUD
Pașcani
IAŞI
Roman
Cherla
BIHOR
Reghin
NEAMŢ
Gheorgheni
Vaslui
CLUJ
CRIŞANA
Câmpia Turzii
Bacău
MOLDAVIA
MUREŞ
TRANSYLVANIA
HARGHITA
BACĂU
VASLUI
ARAD
Arad
Blaj
Mediaş
Adjud
Alba Iulia
COVASNA
Deva
ALBA
SIBIU
Sibiu
GALAŢI
Timişoara
Odobeşti
Orăştie
Făgăraş
BANAT
BRAŞOV
VRANCEA
Lugoj
TIMIŞ
HUNEDOARA
Bocşa
Câmpulung
Tulcea
CARAŞ-SEVERIN
VÎLCEA
BUZĂU
Târgu Jiu
Brăila
PRAHOVA
Rîmnicu Vîlcea
BRĂILA
TULCEA
GORJ
ARGEŞ
Drobeta-Turnu Severin
DÎMBOVIŢA
Ploieşti
DOBROGEA
Drăgăşani
MUNTENIA
Slobozia
OLTENIA
Slatina
Feteşti
MEHEDINŢI
Cernavodă
Craiova
Bucharest
CĂLĂRAŞI
GIURGIU
CONSTANŢA
DOLJ
OLT
Black Sea
Mangalia
TELEORMAN

Example: Romania Touring

- On holiday in Romania; currently in Arad
- Non-refundable ticket to fly out of Bucharest tomorrow
- Formulate goal (perf. evaluation):
  - be in Bucharest before the flight
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Search:
  - sequence of cities

Road Map of Romania



Problem-Solving Agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Aspects of the Simple Problem Solver

- Where does it fit into the agents and environments discussion?
  - Static environment
  - Observable
  - Discrete
  - Deterministic
  - Open-loop system: percepts are ignored, thus break the loop between agent and environment

Well-Defined Problems

- A problem can be defined formally by five components:
  - Initial state
  - Actions
  - Transition model: description of what each action does (successor)
  - Goal test
  - Path cost

Problem Formulation – 5 Components

- Initial state: In(*Arad*)
- Actions, if current state is In(Arad), actions = {Go{Sibiu), Go(Timisoara), Go(Zerind)}
- Transition model:
  - e.g., Results(In(Arad), Go(*Sibiu*)) = In(*Sibiu*)
- Goal test determines whether a given state is a goal state
  - explicit, e.g. In(*Bucharest*)
  - implicit, e.g. checkmate
- Path cost function that assigns a numeric cost to each path
  - e.g., distance traveled
  - step cost: $c(x, a, y)$

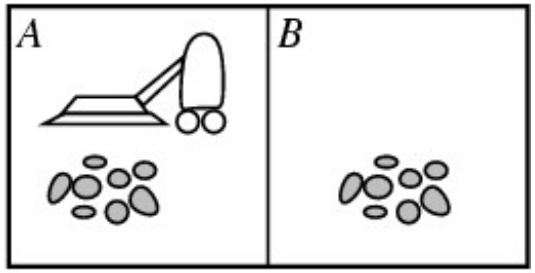
- Solution: a path from the initial state to a goal state
- Optimal solution: the path that has the lowest path cost among all solutions; measured by the path cost function

Problem Abstraction

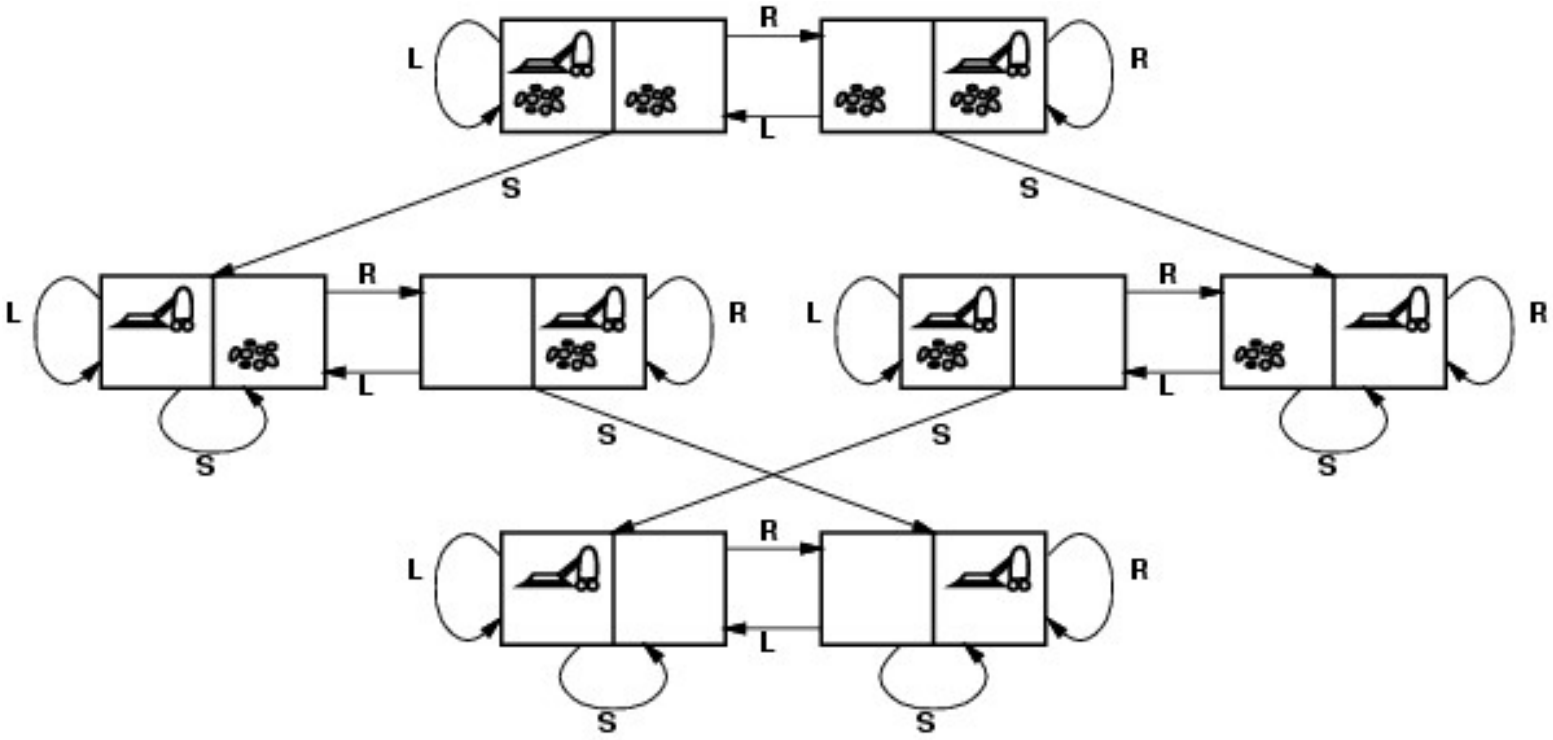
- Real world is complex and has more details
- Irrelevant details should be removed from state space and actions, which is called abstraction
- What's the appropriate level of abstraction?
  - the abstraction is valid, if we can expand it into a solution in the more detailed world
  - the abstraction is useful, if carrying out the actions in the solution is easier than the original problem
  - remove as much detail as possible while retaining validity and usefulness

Example: Vacuum-Cleaner

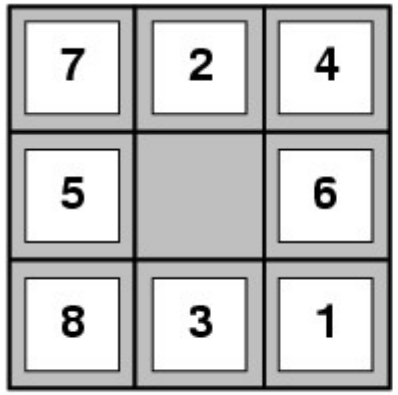
- States
  - 8 states
- Initial state
  - any state
- Actions
  - Left, Right, and Suck
- Transition model
  - complete state space, see next page
- Goal test
  - whether both squares are clean
- Path cost
  - each step costs 1



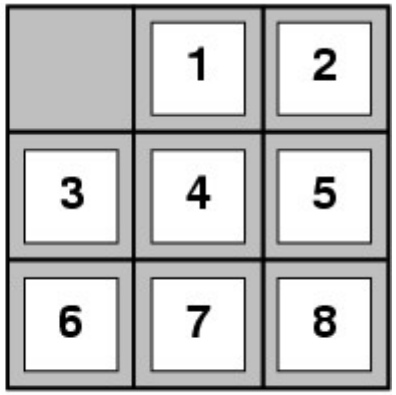
Complete State Space



Example: 8-puzzle



Start State

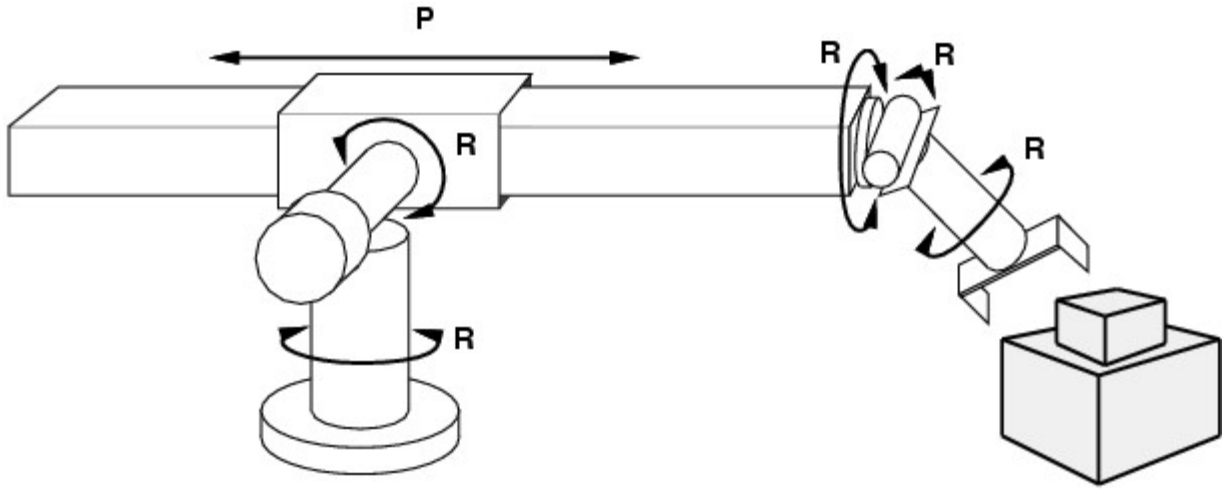


Goal State

NP-Complete

- States:
  - location of each tile and the blank
- Initial state: any, 9!/2
- Actions:
  - blank moves Left, Right, Up or Down
- Transition model:
  - Given a state and action, returns the resulting state
- Goal test: Goal configuration
- Path cost: Each step costs 1

Example: Robotic Assembly



- States
  - real-valued coordinates of robot joint angles; parts of the object to be assembled
- Actions
  - continuous motions of robot joints
- Transition model
  - States of robot joints after each action
- Goal test
  - complete assembly
- Path cost: time to execute

Missionaries & Cannibals

- 3 missionaries and 3 cannibals need to cross a river
- 1 boat that can carry 1 or 2 people
- Find a way to get everyone to the other side, without ever leaving the group of missionaries in one place outnumbered by cannibals in that place
- Check on this link:
  - http://www.learn4good.com/games/puzzle/boat.htm

Problem Formulation

- States:
  - <m, c, b> representing the # of missionaries and the # of cannibals, and the position of the boat
- Initial state:
  - <3, 3, 1>
- Actions:
  - take 1 missionary, 1 cannibal, 2 missionaries, 2 cannibals, or 1 missionary and 1 cannibal across the river
- Transition model:
  - state after an action
- Goal test:
  - <0, 0, 0>
- Path cost:
  - number of crossing

Real-World Problems

- Touring problem: visit every city at least once, starting and ending in Bucharest
- Traveling sales problem: exactly once
- Robot navigation
- Internet searching: software robots

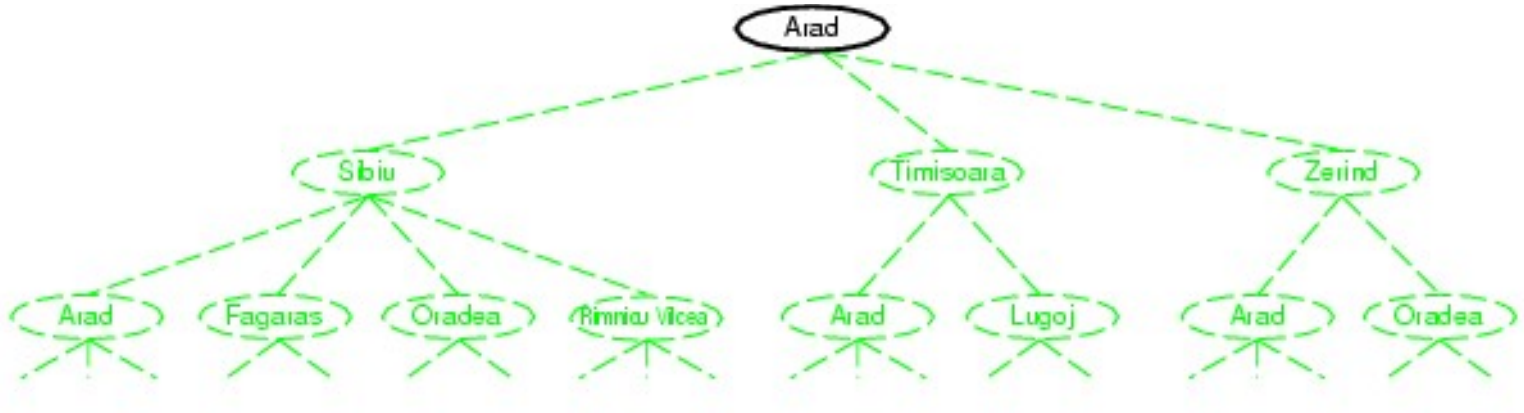
Searching for Solutions

Searching for Solutions

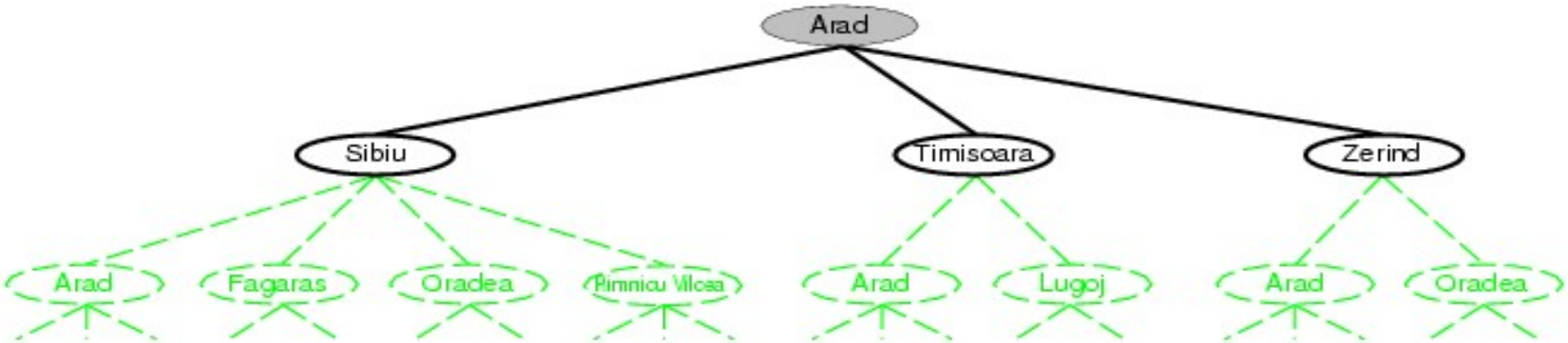
- Search tree: generated by initial state and possible actions

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (expanding states)
  - the choice of which state to expand is determined by search strategy

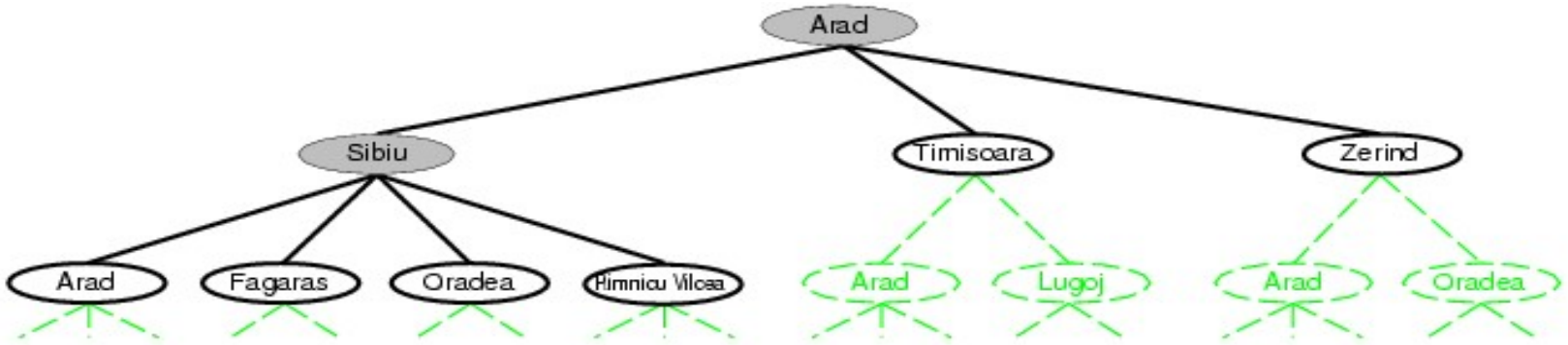
Tree Search Example



Tree Search Example



Tree Search Example



Terminologies

- Frontier: set of all leaf nodes available for expansion at any given point
- Repeated state
- Loopy path: Arad to Sibiu to Arad
- Redundant path: more than one way to get from one state to another

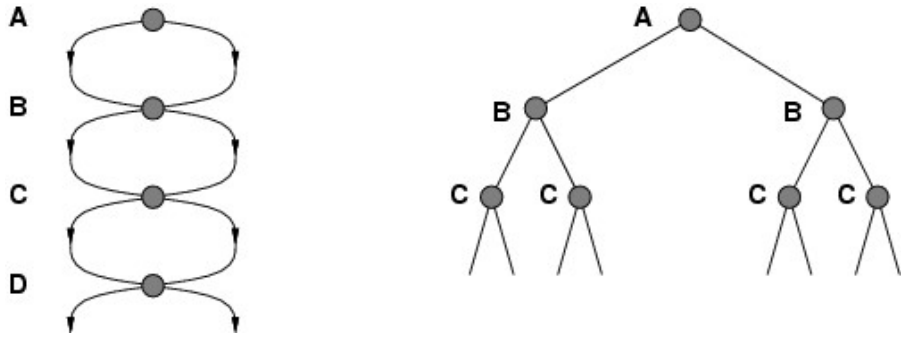
- Sometimes, redundant paths are unavoidable
  - Sliding-block puzzle

General Tree Search Algorithm

```
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
```

Avoiding Repeated States

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Algorithms that forget their history are doomed to repeat it



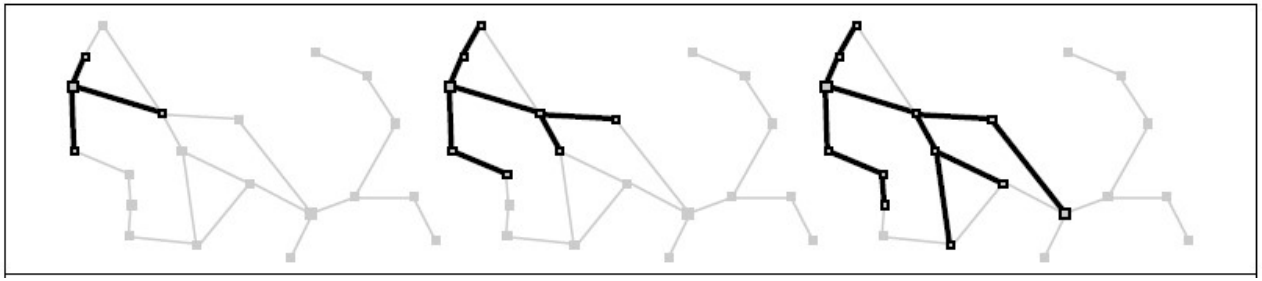
state space size: d + 1 → search tree leaves: 2^d

General Graph Search Algorithm

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

We augment Tree-Search with a explored set, which remembers every expanded node

Graph Search Examples



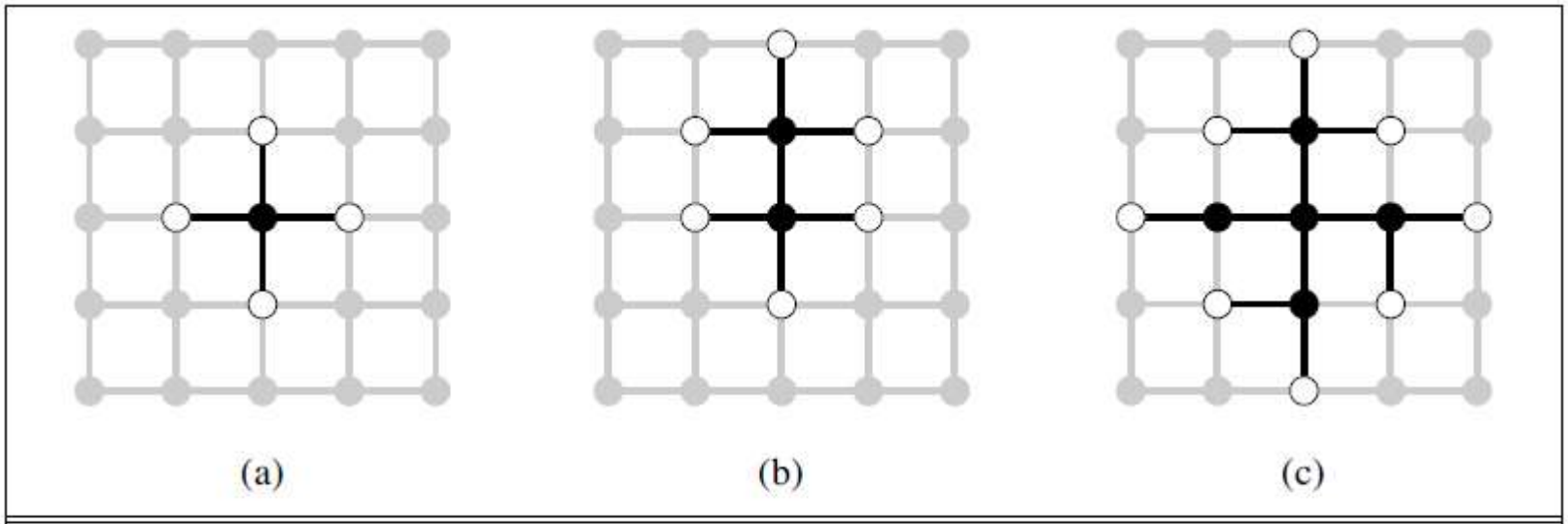
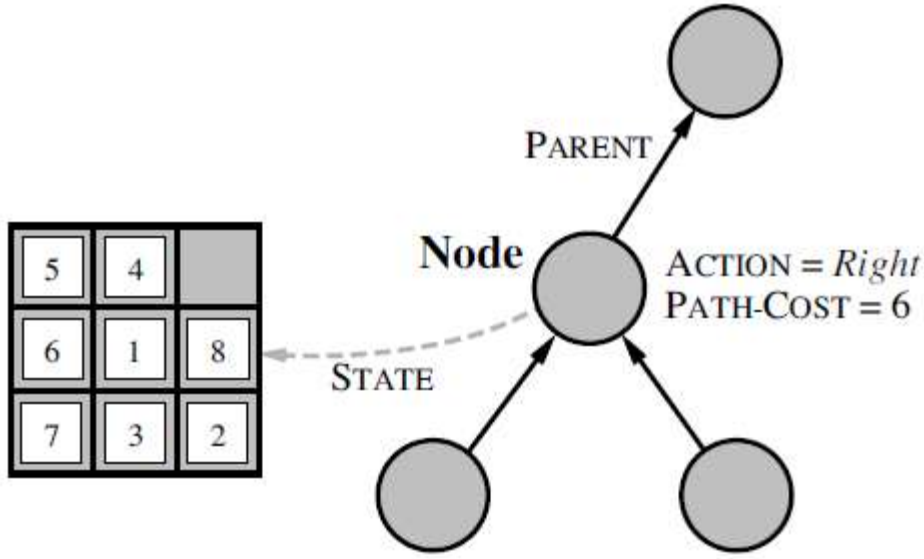


Figure 3.9 **FILES: figures/graph-separation.eps.** The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

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 state, parent node, action, path cost $g(n)$, depth
- A solution path can be easily extracted



CHILD-NODE Function

- The CHILD-NODE function takes a parent node and an action and returns the resulting child node

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action)
        PARENT = parent
        ACTION = action
        PATH-COST = parent.PATH-COST +
                    problem.STEP-COST(parent.STATE, action)
```

Frontier and Explored Set

- Goal of frontier: the next node to expand can be easily located in the frontier
- Possible data structures?

- Goal of explored set: efficient checking for repeated states
- Possible data structures?

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?
  - optimality: does it always find a least-cost solution?
  - time complexity: number of nodes generated
  - 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 ∞)
- Search cost (time), total cost (time+space)

Uninformed Search Strategies

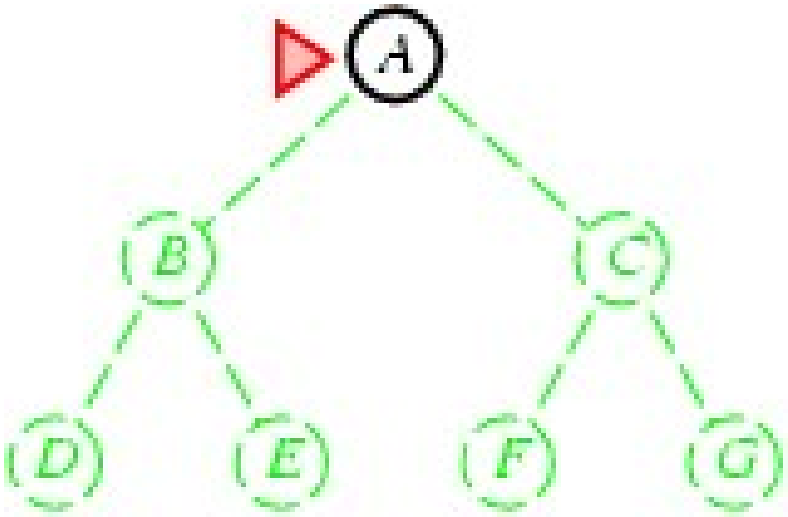
- Uninformed search (blind search) strategies use only the information available in the problem definition

- Strategies that know whether one non-goal state is better than another are called informed search or heuristic search

- General uninformed search strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

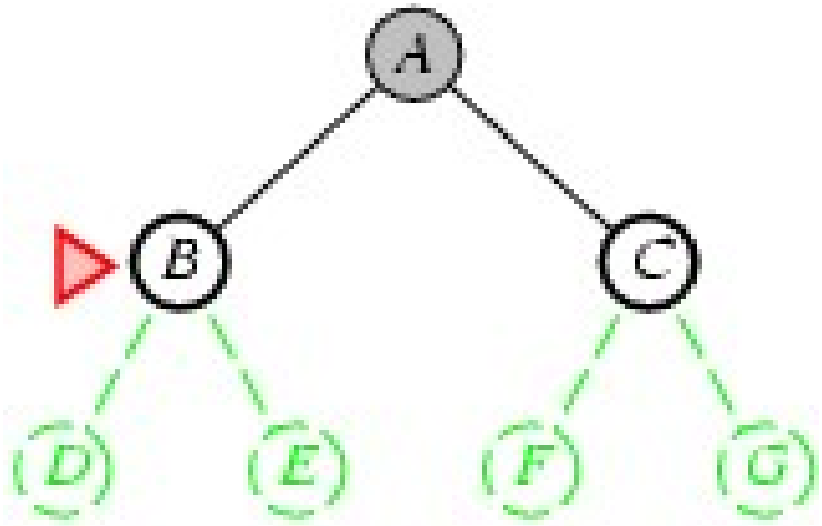
Breadth-First Search

- Expand shallowest unexpanded node
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



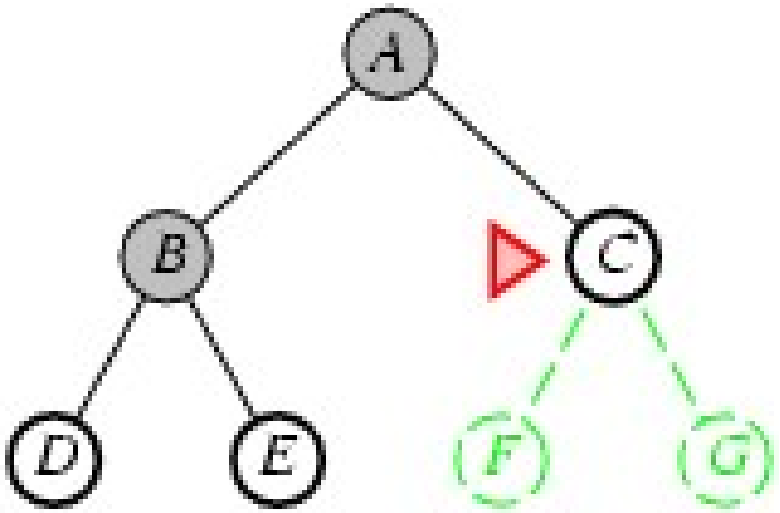
Breadth-First Search

- Expand shallowest unexpanded node
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



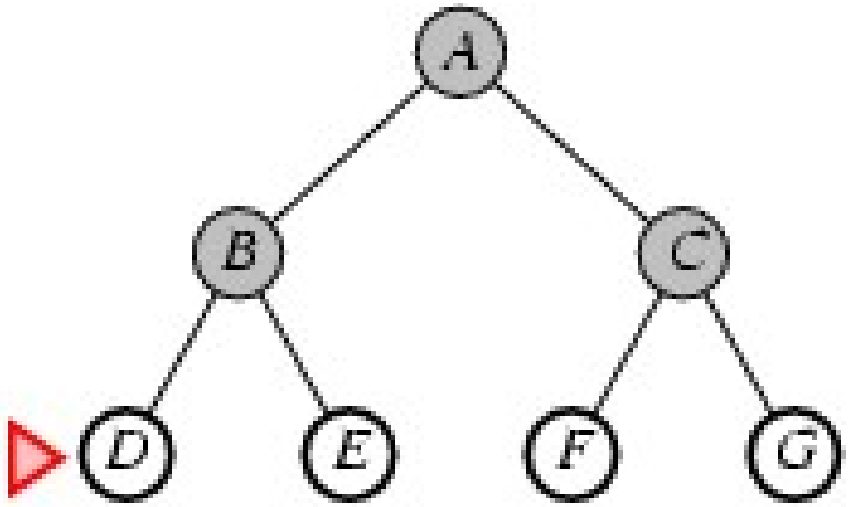
Breadth-First Search

- Expand shallowest unexpanded node
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



Breadth-First Search

- Expand shallowest unexpanded node
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



BFS on a Graph

```
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)
```

Analysis of Breadth-First Search

- Complete?
  - Yes (if b is finite), the shallowest solution is returned
- Time?
  - $b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space?
  - $O(b^d)$ (keeps every node in memory)
- Optimal?
  - Yes if step costs are all identical or path cost is a nondecreasing function of the depth of the node

- Space is the bigger problem (more than time)
- Time requirement is still a major factor

How Bad is BFS?

- With b = 10; 1 million nodes/sec; 1k bytes/node
- It takes 13 days for the solution to a problem with search depth 12, nodes 10^{12}
- 350 years at depth 16

- Memory is more of a problem than time
  - Requires 103G when d = 8
- Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

Uniform-Cost Search

- Expand least-cost unexpanded node
- Implementation:
  - *Frontier* = priority queue ordered by path cost g(n)
- Example, shown on board, from Sibiu to Bucharest
- breadth-first = uniform-cost search when?

Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← 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 ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Analysis

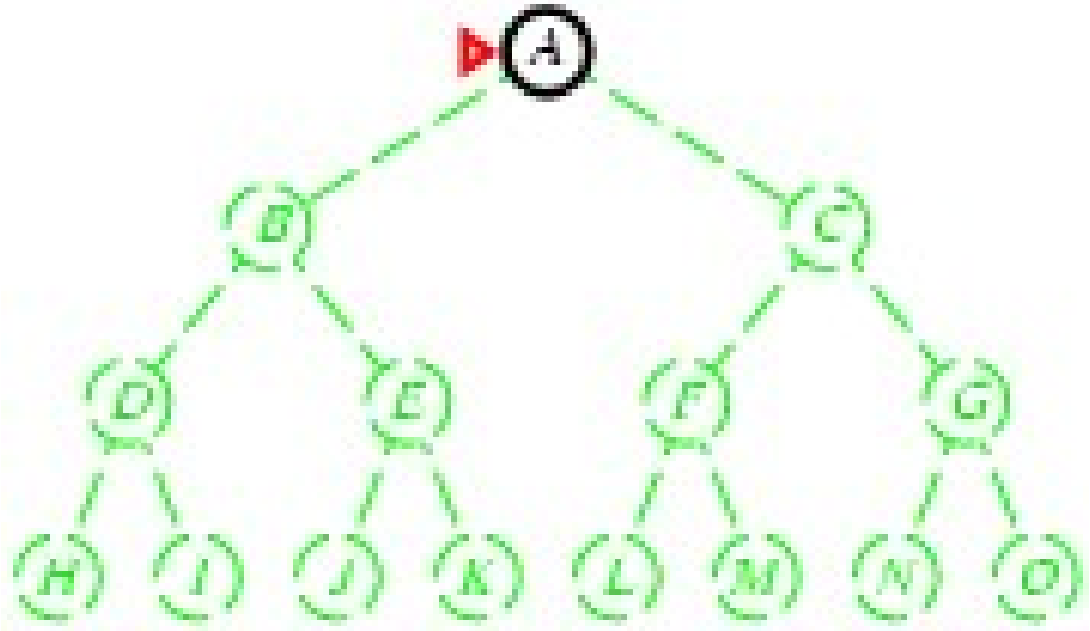
- Complete?
  - Yes, if step cost ≥ ε
- Time?
  - $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
  - # of nodes with $g \leq$ cost of optimal solution
- Space?
  - $O(b^{ceiling(C^*/\varepsilon)})$
  - # of nodes with $g \leq$ cost of optimal solution
- Optimal?
  - Yes – nodes expanded in increasing order of $g(n)$

Uniform-Cost Search is Optimal

- Uniform-cost search expands nodes in order of their optimal path cost
- Hence, the first goal node selected for expansion must be the optimal solution

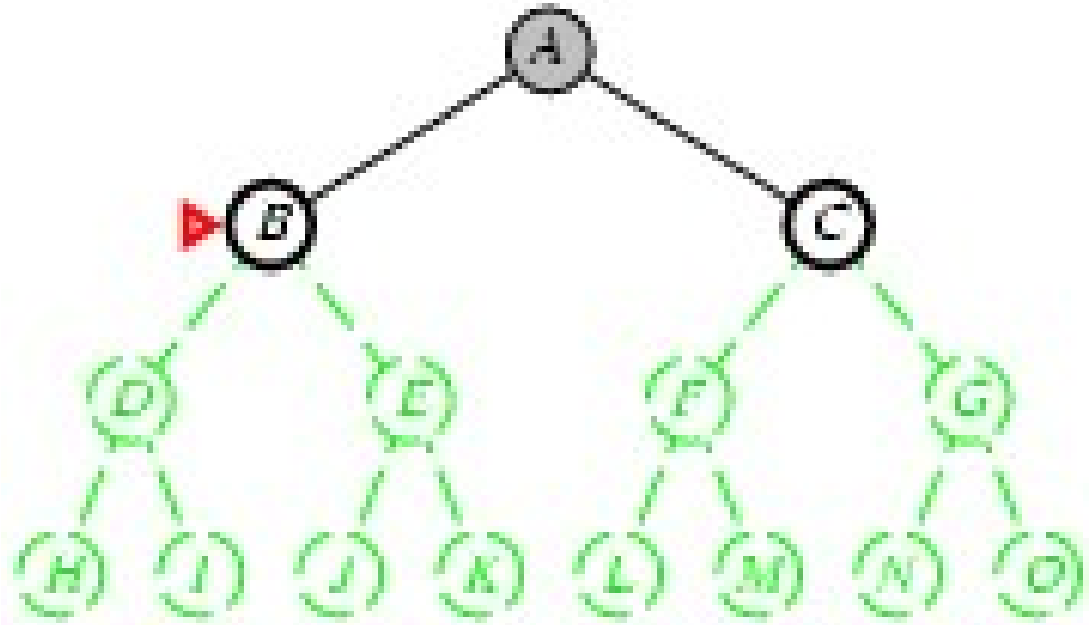
Depth-First Search

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



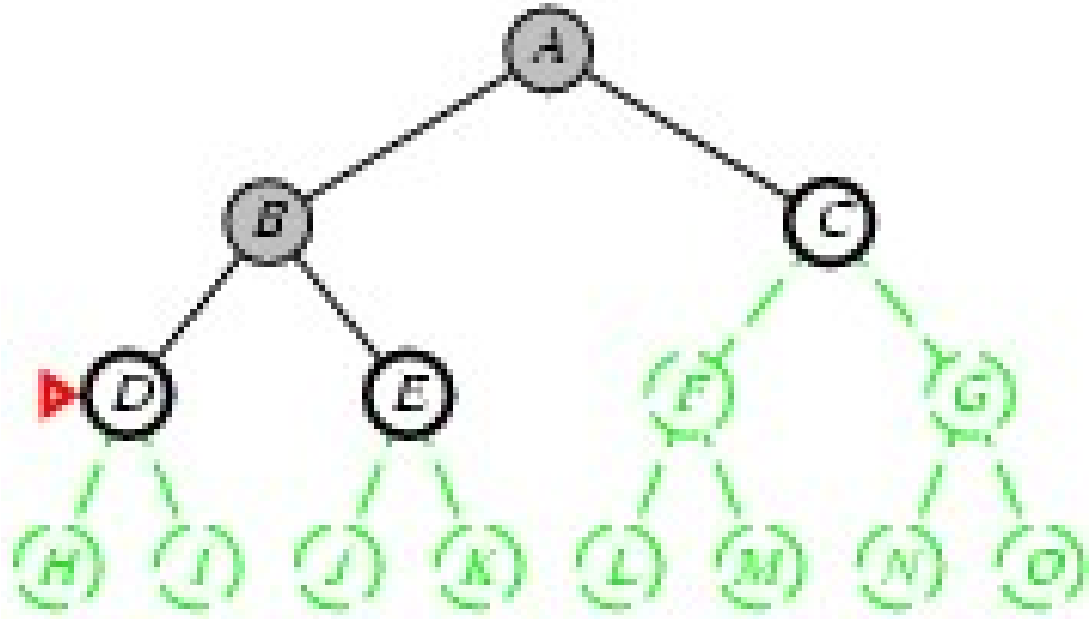
Depth-First Search

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



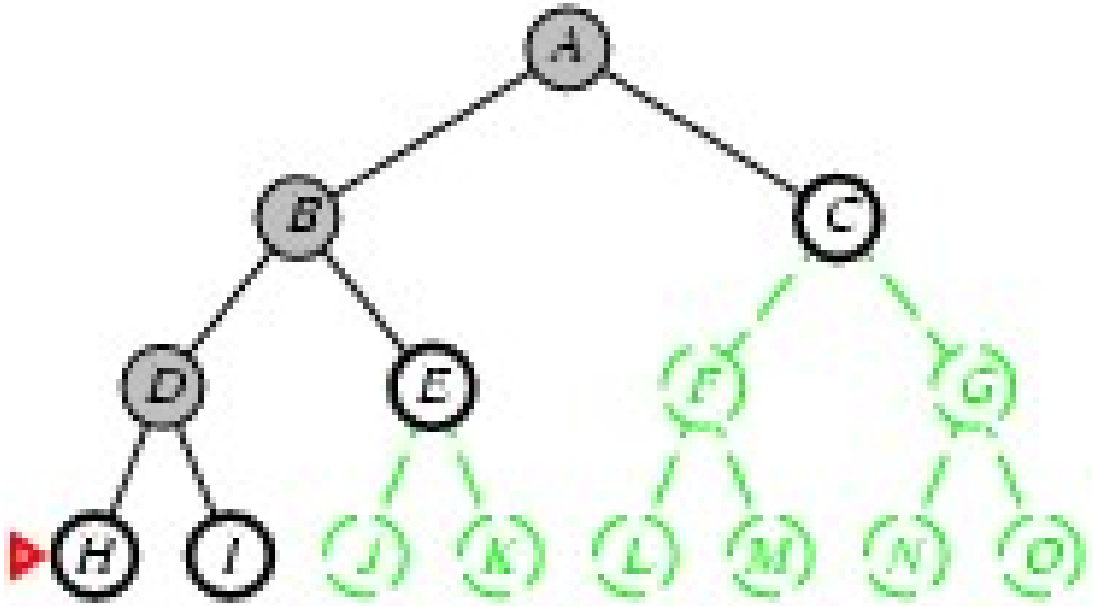
Depth-First Search

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



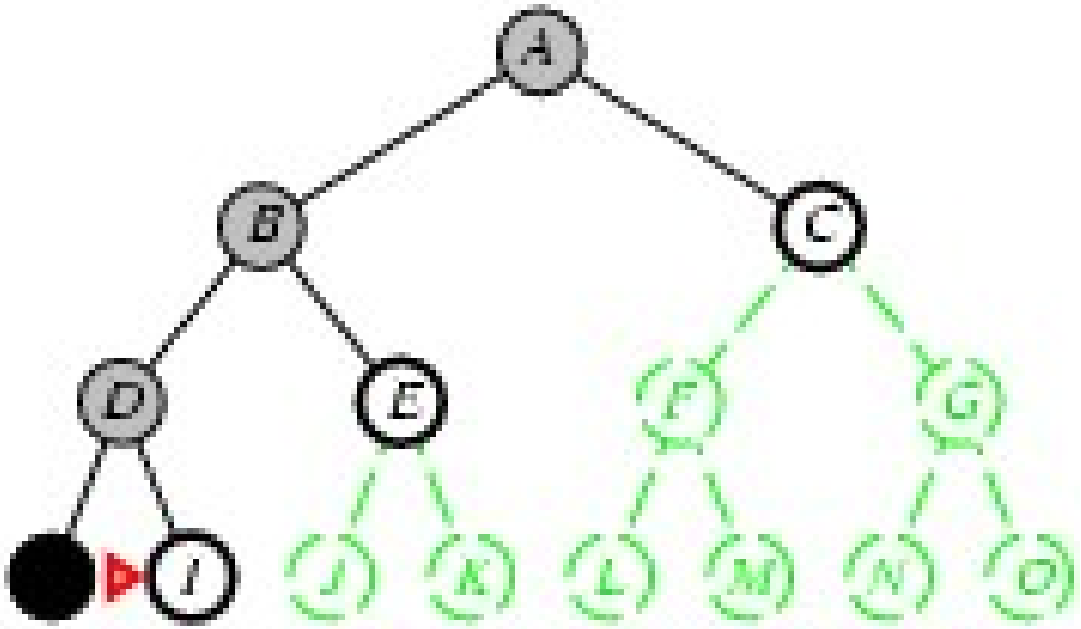
Depth-First Search

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



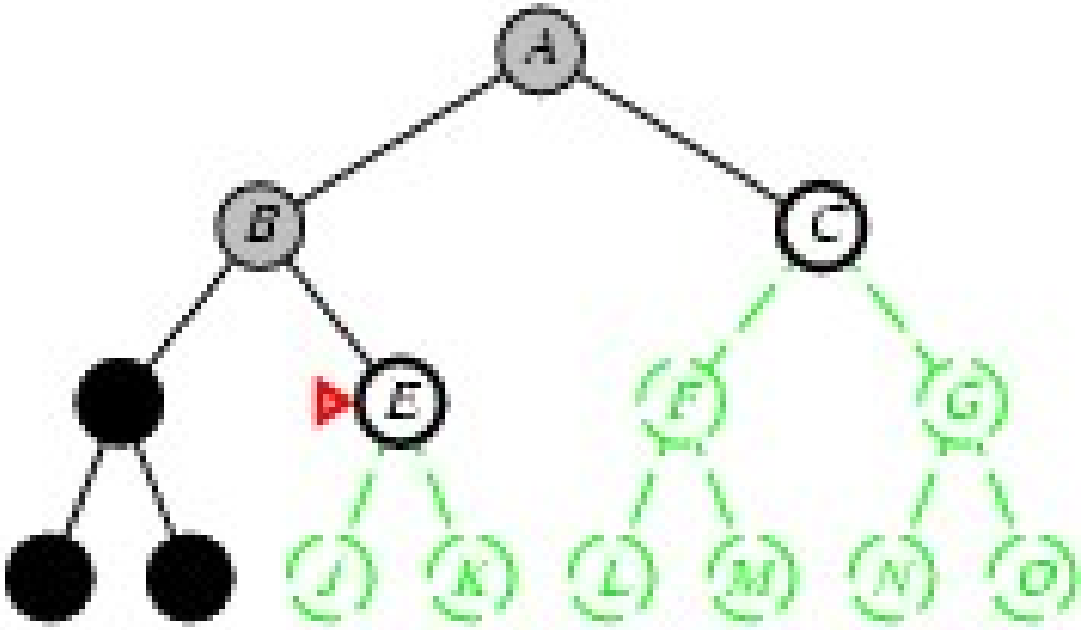
Depth-First Search

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



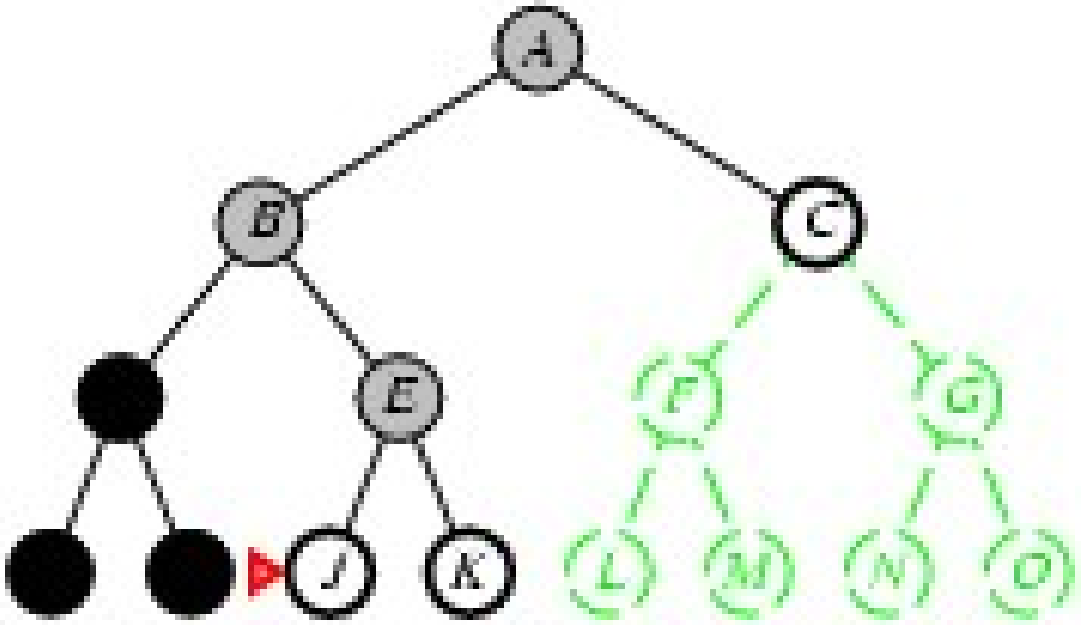
Depth-First Search

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



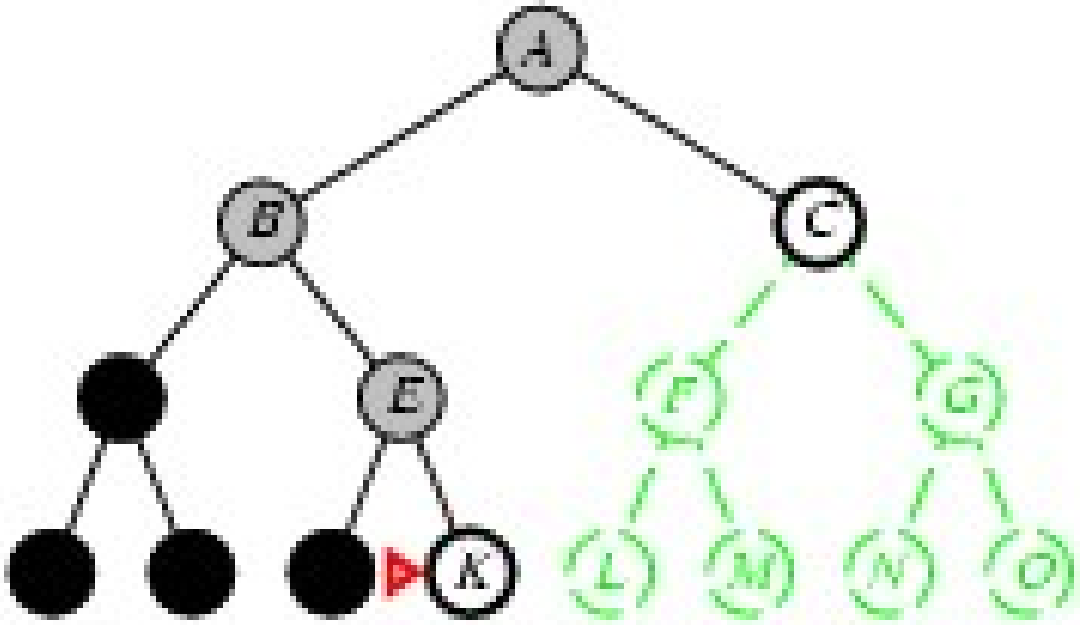
Depth-First Search

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



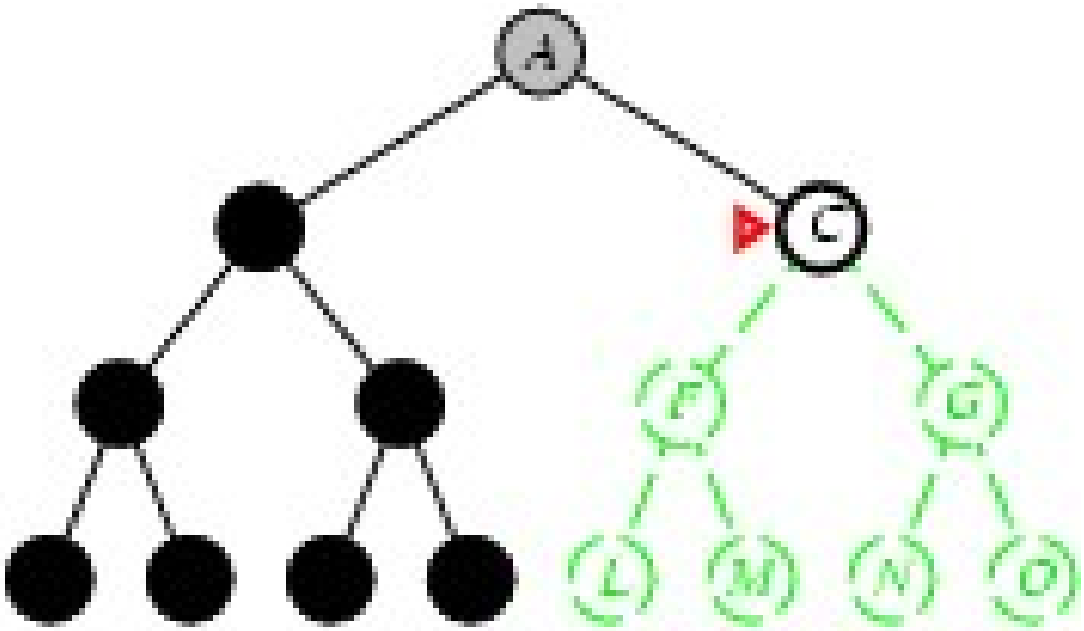
Depth-First Search

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



Depth-First Search

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



Properties of DFS

- Properties of DFS depend strongly on whether the graph-search or tree-search version is used

Analysis of DFS + Tree Search

- Complete?
  - No: fails in infinite-depth spaces, or 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

Analysis of DFS + Graph Search

- Complete?
  - No: also fails in infinite-depth spaces
  - Yes: for finite state 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?
  - Not linear any more, because of explored set
- Optimal?
  - No

Backtracking Search

- Backtracking search is a variant of DFS
  - Only one successor is generated at a time rather than all successors
  - Each partially expanded node remembers which successor to generate next
  - Memory requirement: O(m) vs. O(bm)

Depth-Limited Search

- Is the same as depth-first search with depth limit l, nodes at depth l are treated as if they have no successors

```
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? ← false
      for each action in problem.ACTIONS(node.STATE) 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
```

- Complete? Time? Space? Optimal?

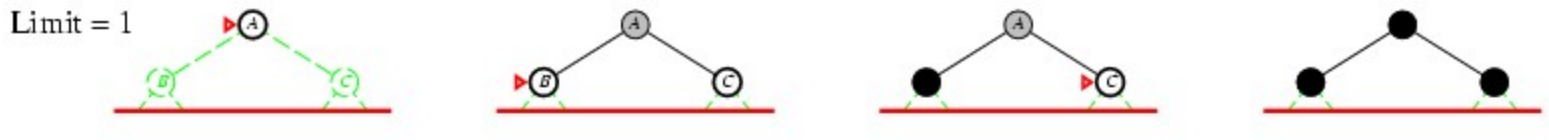
Iterative Deepening DF-Search

- Gradually increase the depth limit until a goal is found
- Combines the benefits of depth-first and breadth-first search

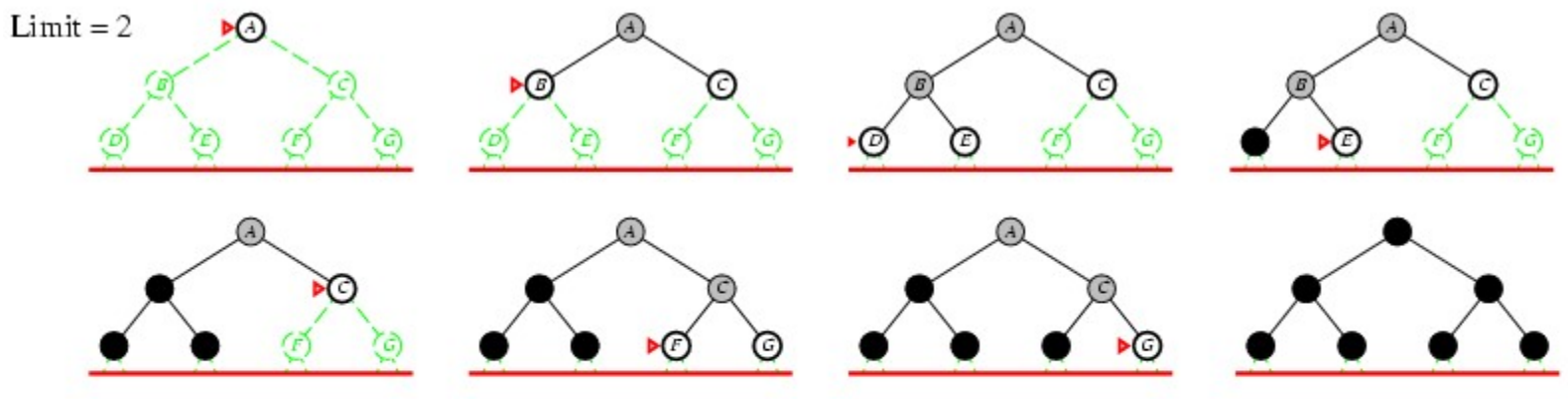
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Depth Limit = 0

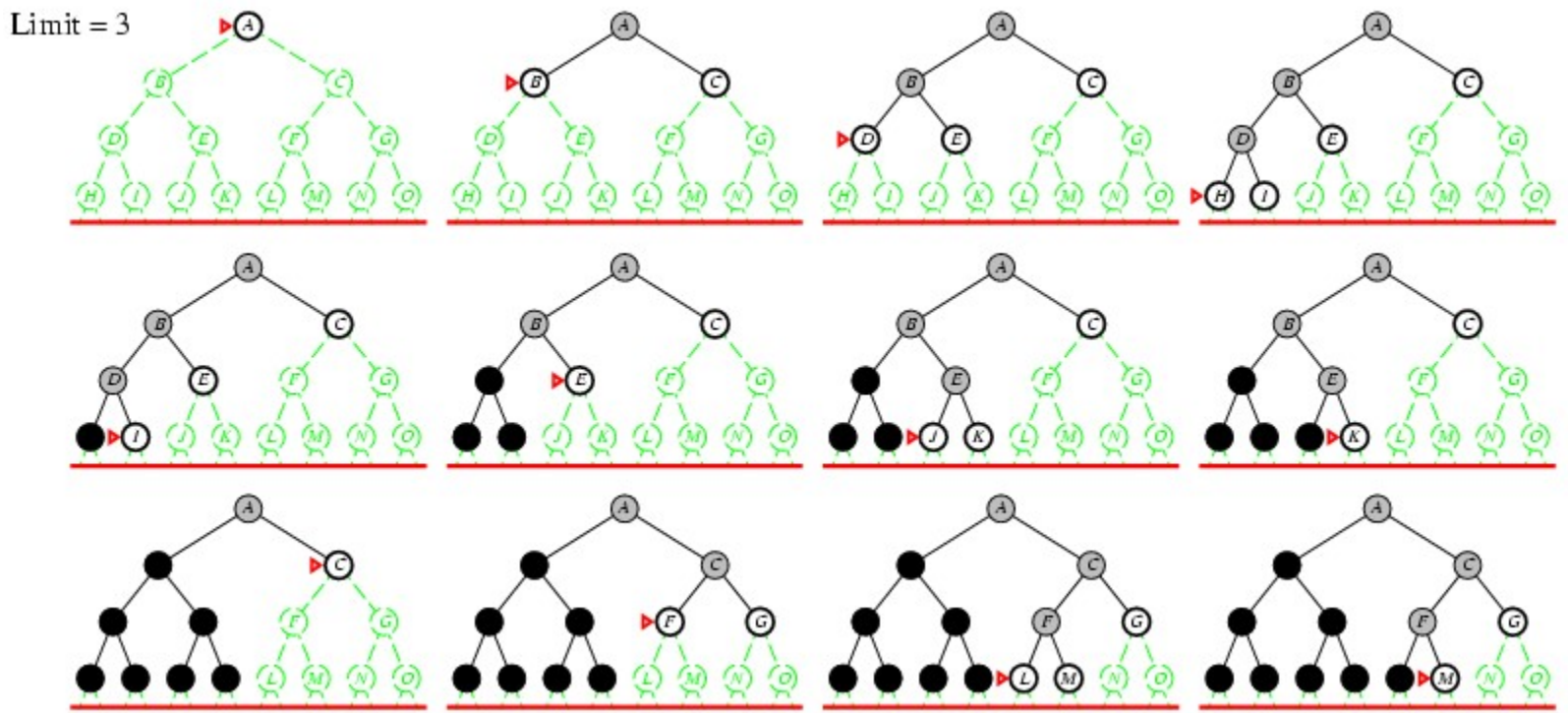
Depth Limit = 1



Depth Limit = 2



Depth Limit = 3



Analysis of Iterative Deepening Search

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

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

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

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

- 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%

- IDS is the preferred uninformed search method when search space is large and depth of solution is unknown

Analysis, Continue

- Complete?
  - Yes
- Time?
  - $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?
  - $O(bd)$ (tree search version)
- Optimal?
  - Yes, if step costs are identical or path cost is a nondecreasing function of the depth of the node

Summary of Uninformed Tree Search Strategies

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

- Complete and optimal under certain conditions
- Discussion on bidirectional search

Analysis of Graph Search

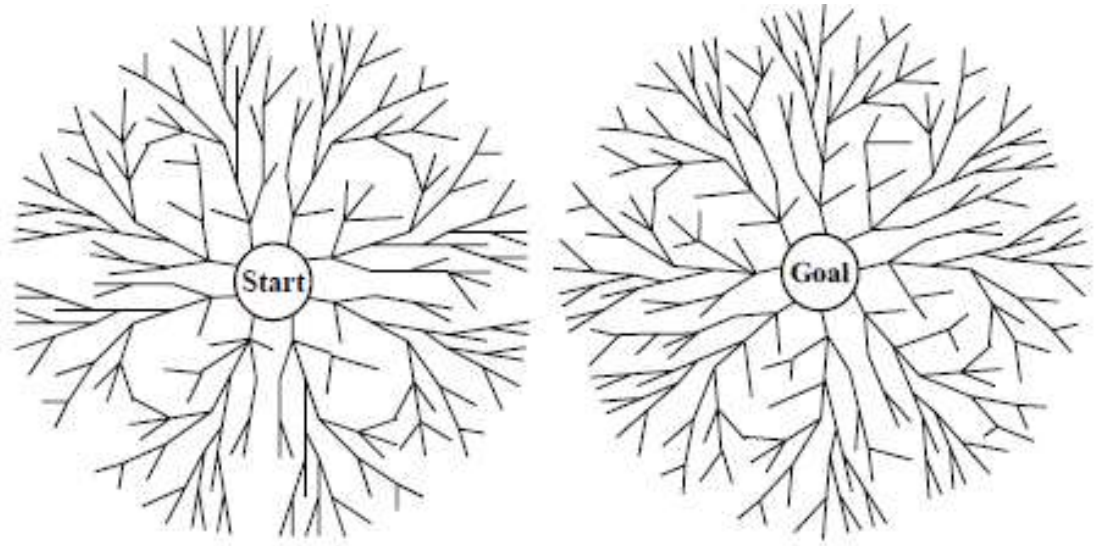
- Much more efficient than Tree-Search
- Time and space are proportional to the size of the state space

- Optimality:
  - uniform-cost search or breadth-first search with identical step costs are still optimal even if it returns the first path found
  - iterative-deepening, identical step cost or non-decreasing function of depth of a node

- Tradeoff: depth-first or iterative deepening are not linear anymore

Bidirectional Search

- Runs two simultaneous searches
  - Forward from initial state
  - Backward from goal state

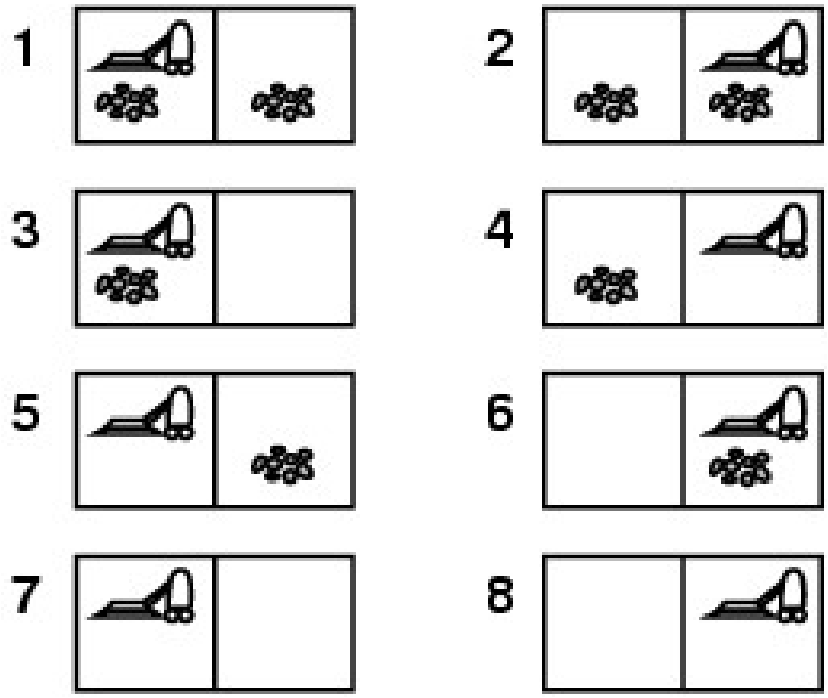


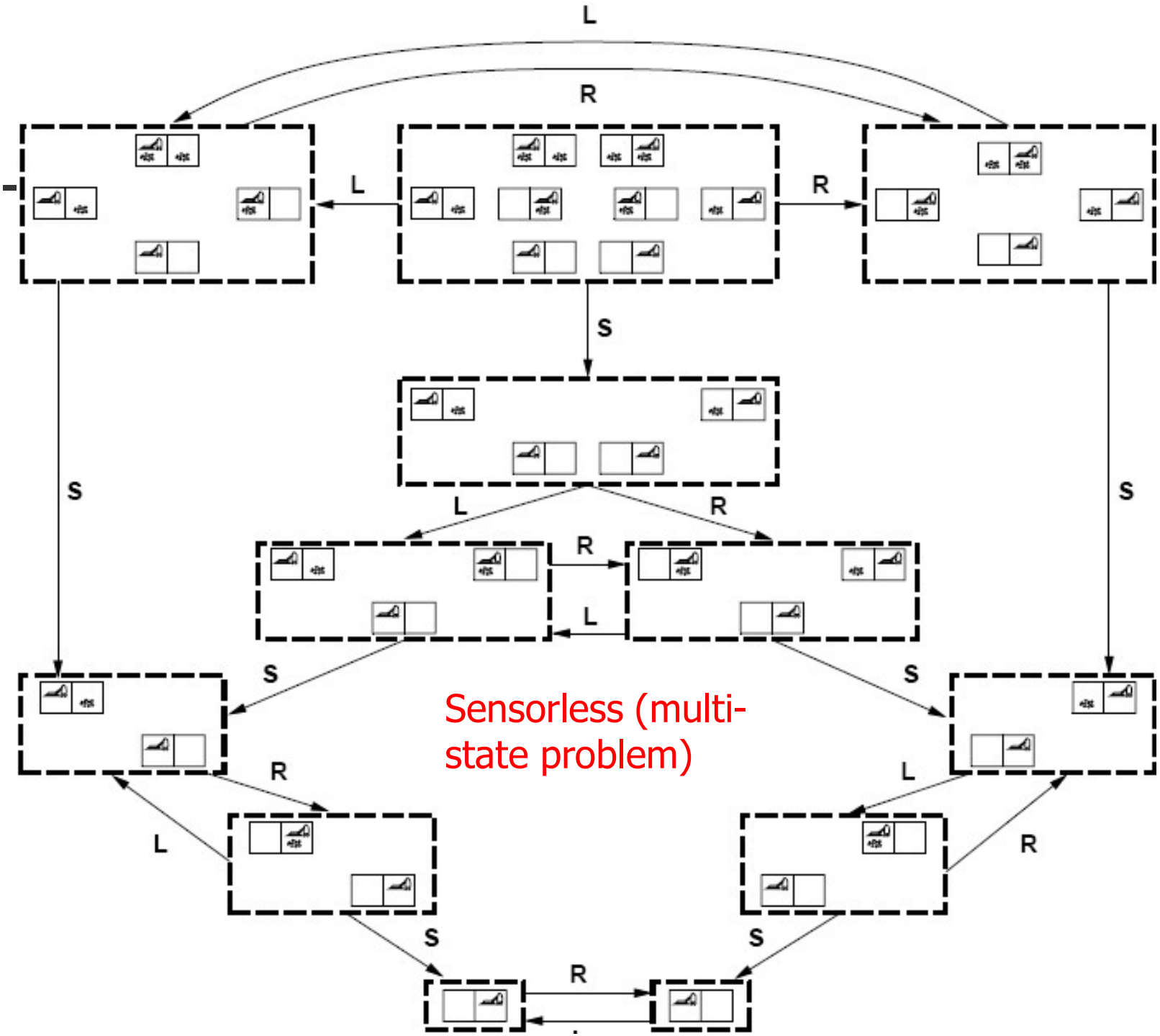
Searching With Partial Information

- We have covered: Deterministic, fully observable → single-state problem
  - agent knows exactly which state it will be in
  - solution is a sequence
- Deterministic, non-observable → multi-state problem
  - Also called sensorless problems (conformant problems)
  - 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
  - solution is a tree or policy
- Unknown state space → exploration problem ("online")
  - states and actions of the environment are unknown

Example: Vacuum World

- Single-state, start in #5. Solution?
  - [Right, Suck]
- Multi-state, start in #[1, 2, …, 8]. Solution?
  - [Right, Suck, Left, Suck]

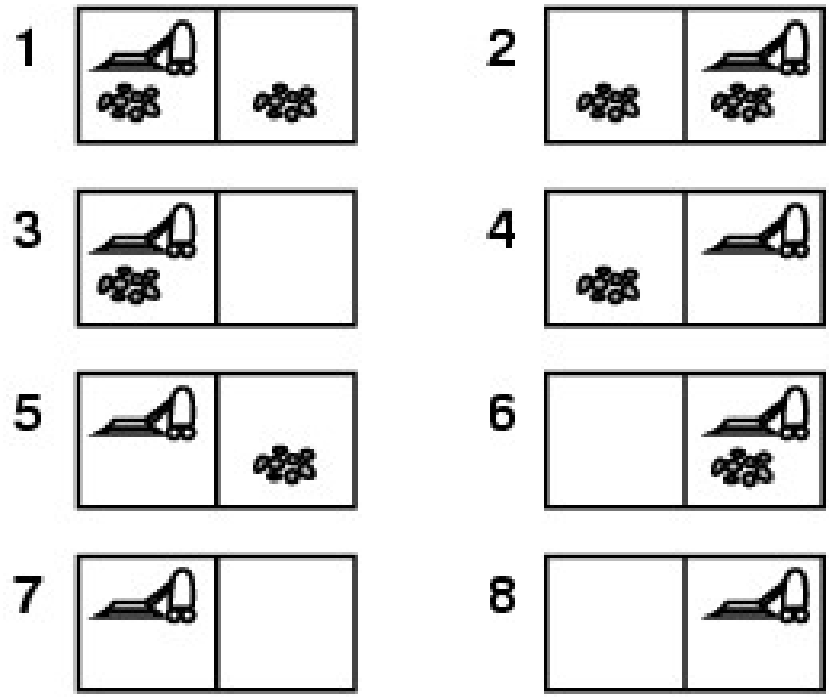




Sensorless (multi-state problem)

Contingency Problem

- Contingency, start in #5 & 7.
  - *Nondeterministic*: suck may dirty a clean carpet
  - *local sensing*: dirt, location only at current location
  - Solution?
  - Percept: [Left, Clean] → [Right, **if** dirty **then** Suck]



Summary

- We have covered methods for selecting actions in environments that are deterministic, observable, static, and completely known
- Problem formulation requires abstraction
- Uninformed search strategies

Announcement

- Now, in-class exercises

Question 1

- Define the following items:
  - State, state space, search tree
- Does a finite state space always lead to a finite search tree?
- How about a finite state space that is a tree or a finite directed acyclic graph?

Question 2

- Give a complete problem formulation for each of the following.
  - You have to color a planar map using only four colors, in such a way that no two adjacent regions have the same color.
  - A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable, climable 3-foot-high crates.

Question 3

- Consider a state space where the start state is number 1 and the successor function for state n returns two states, numbers 2n and 2n+1
  - Draw the portion of the state space from 1 to 15
  - Suppose the goal state is 11. List the order in which nodes will be visited for BFS, DLS with limit 3, and IDS.
  - How well would bidirectional search work on this problem?

Question 4

- Prove that uniform-cost search and BFS with constant step costs are optimal when used with the Graph-Search algorithm.

- Show a state space with varying step costs in which Graph-Search using iterative deepening finds a suboptimal solution.

Question 5

- Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $O(n^2)$ vs. $O(n)$).

Question 6

- Use uniform-cost search implemented with the graph search algorithm to find a route from Arad to Bucharest.