Artificial Intelligence

Chapter 3: Solving Problems by Searching

Slides from Russell and Norvig's *Artificial Intelligence: A Modern Approach, Second Edition*

Overview

- Problem-Solving Agents
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
- Avoiding Repeated States

Problem-Solving Agents

- Reach goals through sequences of actions
- Formulate the goal(s) and the problem
  - Abstraction: Should be easier than original problem
- Search for a sequence of actions to reach a goal state
  - A <u>solution</u> is a sequence of actions from initial state to a goal state
- Execute the sequence of actions

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 ← 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
```

Formulating the Problem

- Initial state = starting state for the agent
- Given a state and a set of actions, the successor function gives the possible next states
- Initial state + successor function yields the state space
  - State space = the set of all states reachable from the initial state

Formulating the Problem, Continued

- The goal test determines if a state is a goal state
- A path is a particular sequence of states, connected by particular actions
- The path cost function assigns a cost to each path
  - Optimal solution: a solution with minimal path cost

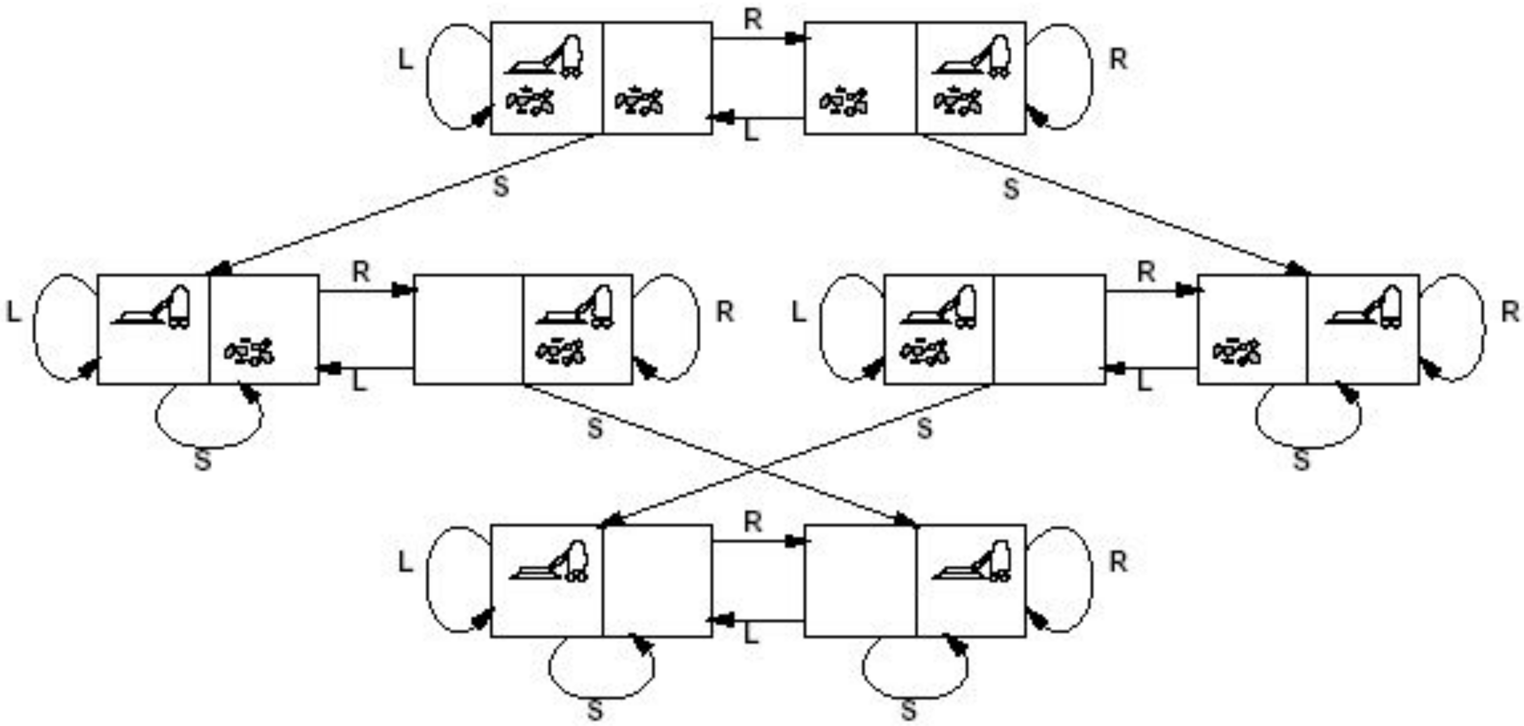
Example Problems

- <u>“Toy” Problems</u>: Simple, but useful for theory and analysis
  - Vacuum world, Sliding-block puzzles, N-Queens
- <u>Real Problems</u>: The solutions are more useful, but not as helpful to theory
  - Route-finding, touring, assembly sequencing, Internet searching

Example: The Vacuum World

- States: Location, dirt or not
- Initial State: Given by problem
- Goal: No dirt in any location
- Actions: Left, Right, Vacuum, NoOp
- Path Cost: 1 per action (0 for NoOp)

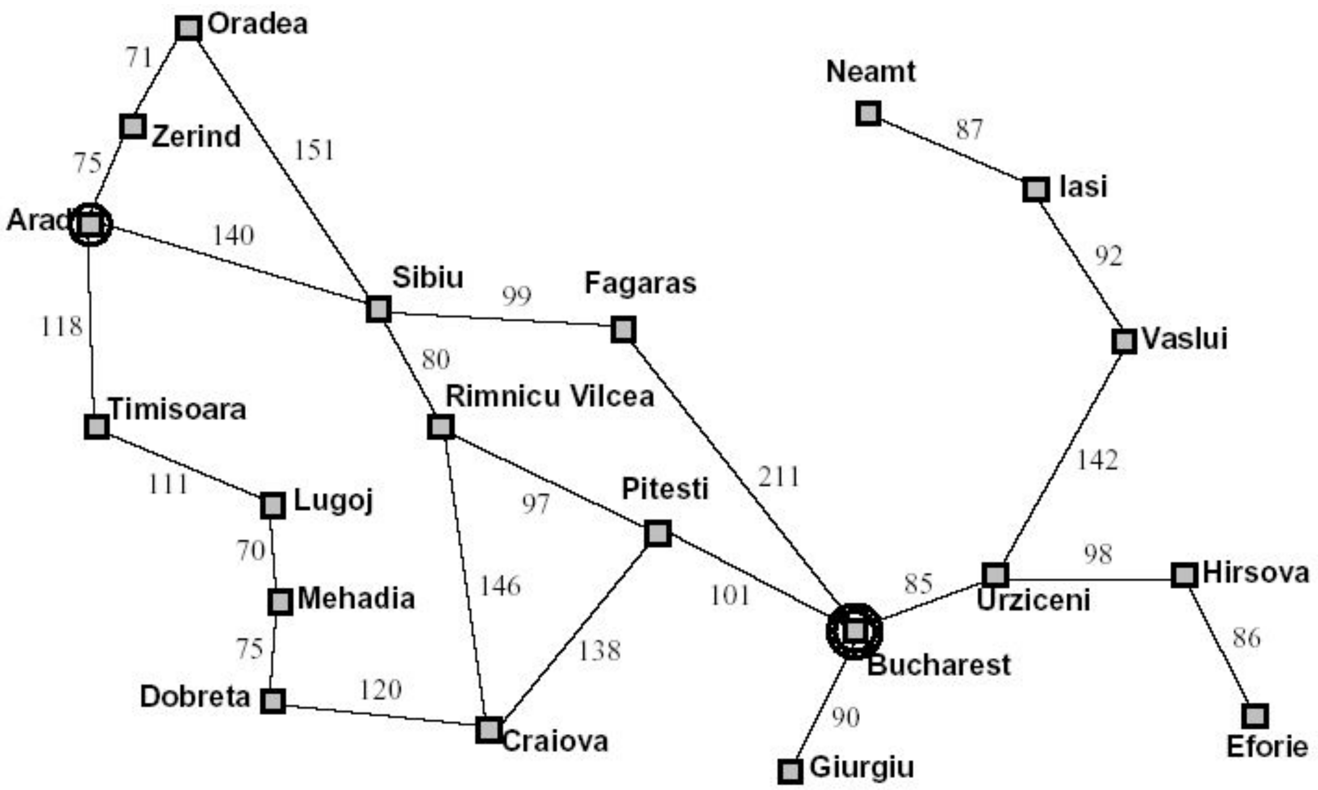
State Space for the Vacuum World



Example: Driving in Romania

- States: In a city
- Goal: Arrive in Bucharest
- Initial State: in Arad
- Actions: Drive from one city to another

Example: Driving in Romania



Searching for Solutions

- State space can be represented as a <u>search tree</u> (or graph) of search nodes
- Each node represents a state
  - Many nodes can represent the same state
- Each arc represents a valid action from one state to another
- A solution is a path from initial to goal nodes

Search Nodes

- A search node is a data structure that contains
  - The <u>real-world state</u> represented by the node
  - The <u>parent</u> node that generated it
  - The <u>action</u> that lead to it
  - The path cost to get to it (from the initial state)
  - Its <u>depth</u> (the number of steps to reach it from the initial state)

Searching for Solutions

- The search tree starts with just the <u>root node</u>, which represents the initial state
- First, use the goal test to see if the root is a goal
- If not, <u>expand</u> this node, which <u>generates</u> new nodes
  - Use successor function to apply valid actions. The results of these actions lead to new states, represented by new nodes
- Nodes that are generated, but not yet expanded are in the <u>fringe</u>, (physically a queue data structure)

Search Strategies

- Given several states, which to consider (i.e., goal test, expand, etc.) first?
  - This is determined by the particular <u>search strategy</u> being used in the algorithm

General Idea

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure
  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*] applied to STATE(*node*) succeeds **return** *node*
    fringe ← INSERTALL(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
    PARENT-NODE[*s*] ← *node*; ACTION[*s*] ← *action*; STATE[*s*] ← *result*
    PATH-COST[*s*] ← PATH-COST[*node*] + STEP-COST(*node*, *action*, *s*)
    DEPTH[*s*] ← DEPTH[*node*] + 1
    add *s* to *successors*
  return *successors*

Search Algorithm Performance

- Performance measured along four dimensions:
  - **Completeness**: Will the algorithm always return a solution, if one exists?
  - **Optimality**: Will it always return the optimal solution, if one exists?
  - **Time complexity**: How long does it take?
  - **Space complexity**: How much memory does it take?
- Branching factor (b), depth of shallowest goal node (d), maximum length of any path in the state space (m)
- Search cost vs. total cost
  - Tradeoffs

Uninformed Search

- Uninformed (or “blind”) search algorithms can only generate successor nodes and do the goal test
  - No other “problem insight”, common sense, etc. is used in finding a solution
- Search strategies differ in how they choose which node to check/expand next

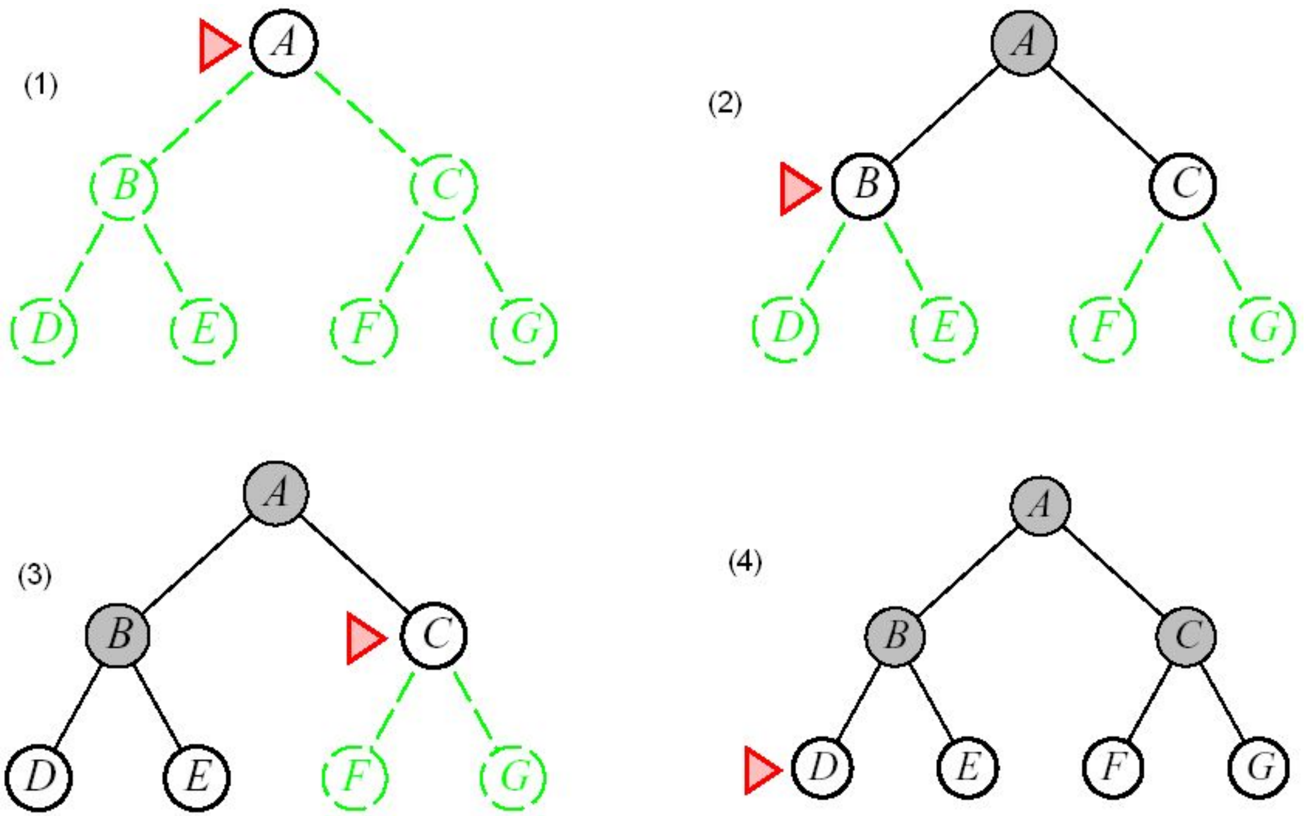
Some Uninformed Search Algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-First Search (BFS)

- BFS considers all nodes at a given depth before moving on to the next depth
  - Always expands the shallowest node first
- Implemented in TREE-SEARCH using a first-in first-out (FIFO) queue
  - State to consider is pulled from front of queue
  - Generated states are added at the back of the queue
  - Check all successors of the root node. Then check all of *their* successors, etc.

Breadth-First Search (BFS)



Performance of Breadth-First Search

- Good news
  - It is complete
  - It is optimal, as long as the path cost function is non-decreasing (e.g., if all actions have the same cost)
- Bad news
  - Time and (especially) space complexity can be prohibitively high
    - Has to keep all nodes in memory: queue gets large
    - If d is large, takes a long time to reach a goal

Uniform-Cost Search (UCS)

- UCS expands the closest (in terms of path cost) node first
  - If all step costs are equal, this is equivalent to BFS
  - “Closest” means total path cost so far, which is not necessarily the same as the number of steps
- Implemented in TREE-SEARCH using a first-in first-out (FIFO) queue
  - State to consider is pulled from front of queue
  - Generated states are added to the queue, then contents of queue are ordered according to path cost

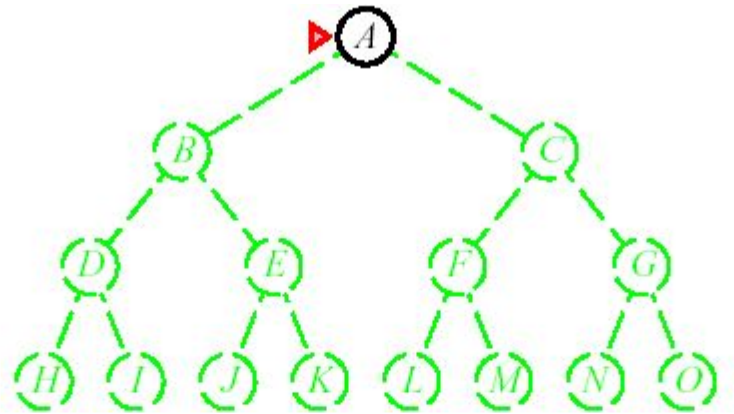
Performance of Uniform-Cost Search

- Good news
  - It is complete and optimal, as long as step costs are positive
- Bad news
  - Time and (especially) space complexity can be prohibitively high, just as with BFS
  - Can get stuck in an infinite loop: zero-cost steps, repeated states

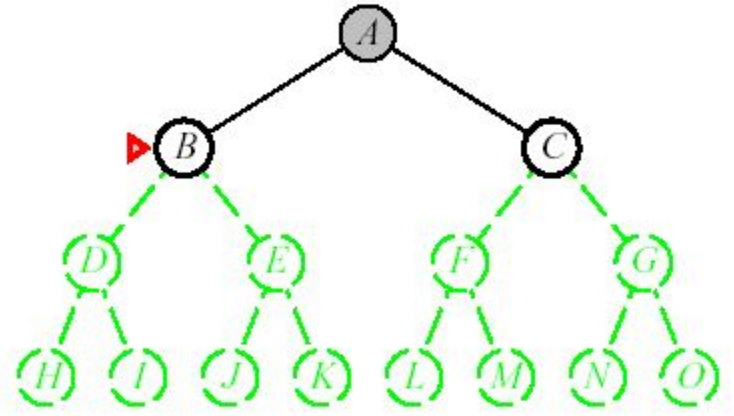
Depth-First Search (DFS)

- DFS expands the <u>deepest</u> node first
- Implemented in <u>TREE-SEARCH</u> using a last-in first-out (LIFO) queue
  - State to consider is pulled from front of queue
  - Generated states are added at the <u>front</u> of the queue
  - Check first successor of the root node. Then check *its* successor, etc.
  - If a non-goal node is reached with no successors, back up just until an unexpanded node is reached, then continue.

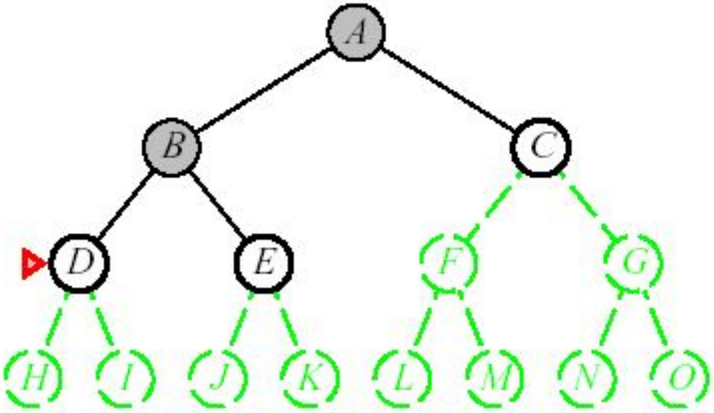
Depth-First Search (DFS)



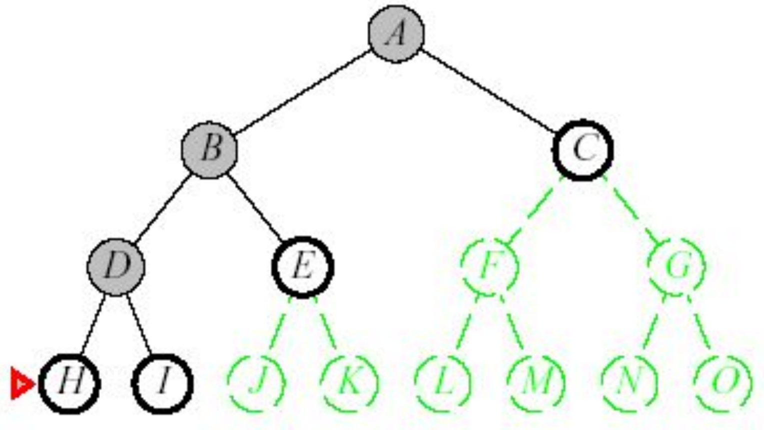
Depth-First Search (DFS)



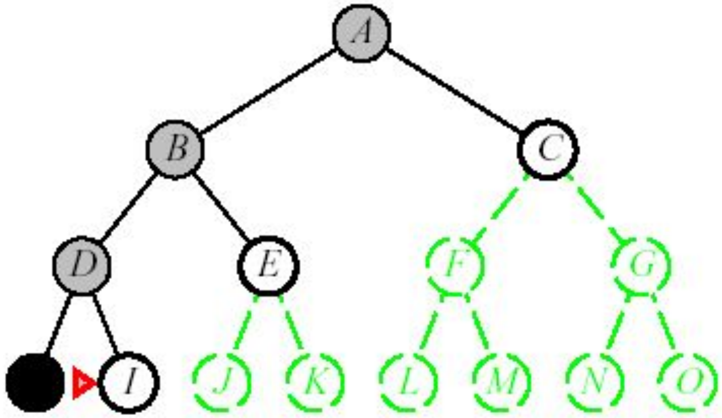
Depth-First Search (DFS)



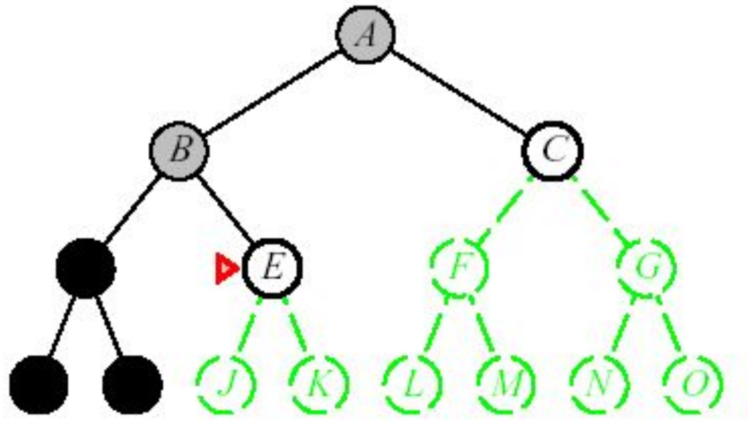
Depth-First Search (DFS)



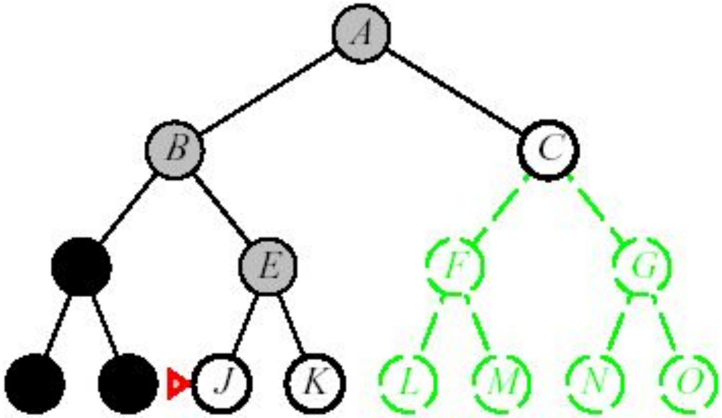
Depth-First Search (DFS)



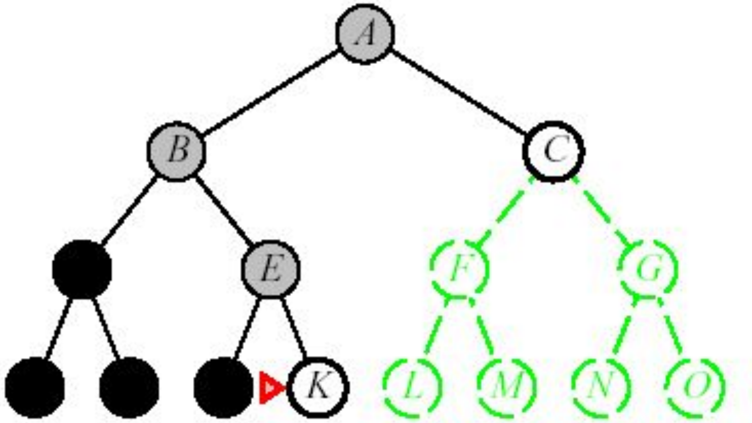
Depth-First Search (DFS)



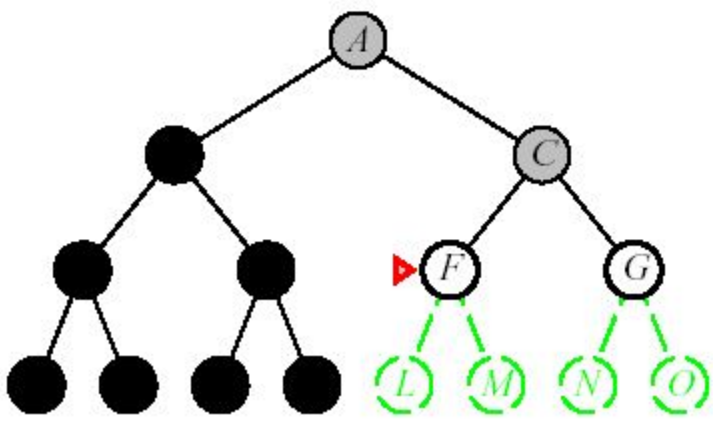
Depth-First Search (DFS)



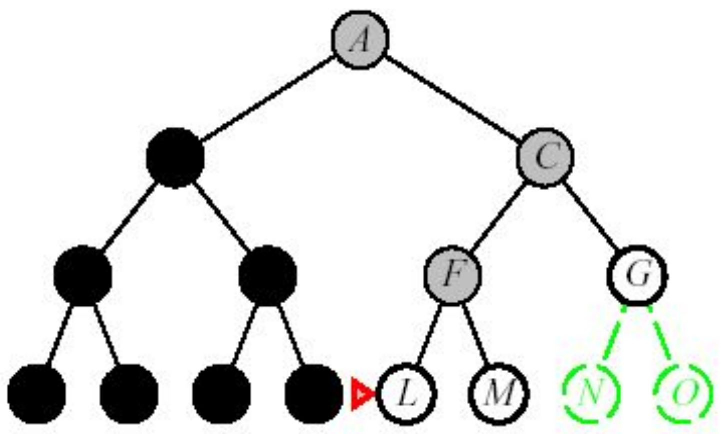
Depth-First Search (DFS)



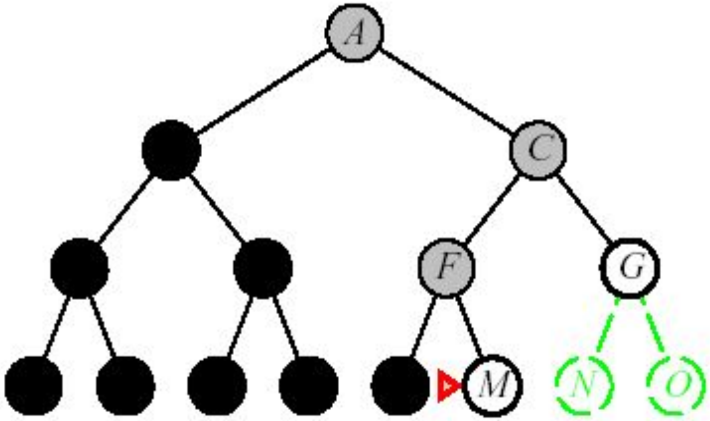
Depth-First Search (DFS)



Depth-First Search (DFS)



Depth-First Search (DFS)



Performance of Depth-First Search

- Good news
  - Space complexity is small
- Bad news
  - Neither complete nor optimal
  - Time complexity can be large
  - Can get stuck in an infinite loop: repeated states

Depth-Limited Search (DLS)

- DLS is DFS with a depth limit, L
  - Nodes at depth limit are treated as leaves
  - Depth limits can come from domain knowledge
    - E.g., L = diameter of state space
- Implemented same as DFS, but with depth limit
- Performance
  - Incomplete if L < d
  - Nonoptimal if L > d
  - Time and space complexity <= DFS

Iterative Deepening Depth-First Search (IDDFS)

- IDDFS performs DLS one or more times, each time with a larger depth limit, l
  - Start with l = 0, perform DLS
  - If goal not found, repeat DLS with l = 1, and so on
- Performance
  - Combines benefits of BFS (completeness, optimality) with benefits of DFS (relatively low space complexity)
- Not as inefficient as it looks
  - Repeating low-depth levels: these levels do not have many nodes

Bidirectional Search

- Perform two searches simultaneously
  - One search from the initial state to a goal state
  - Another search from a goal state to the initial state
  - Stop searching when either search reaches a state that is in the fringe of the other state (i.e., when they "meet in the middle")
- Performance
  - Complete and optimal if both searches are BFS
  - Not easy to generate <u>predecessors</u> for some goals (e.g., checkmate)

Avoiding Repeated States

- Reaching a node with a repeated state means that there are at least two paths to the same state
  - Can lead to infinite loops
- Keep a list of all nodes expanded so far (called the <u>closed list</u>; the fringe is the <u>open list</u>)
  - If a generated node's state matches a state in the closed list, discard it rather than adding it to the fringe