Artificial Intelligence

Fall 2017

CSE 440

Solving Problems by Searching
(Chapter 03)

Mirza Mohammad Lutfe Elahi
Department of Electrical and Computer Engineering
North South University

Search

- One of the most basic techniques in AI
  - Underlying sub-module in most AI systems
- Can solve many problems that humans are not good at (achieving super-human performance)
- Very useful as a general algorithmic technique for solving many non-AI problems

Why Search?

- To achieve goals or to maximize our utility we need to predict what the result of our actions in the future will be.

- There are many sequences of actions, each with their own utility.

- We want to find, or search for, the best one.

Limitation of Search

- There are many difficult questions that are not resolved by search. In Particular, the whole question of how does an intelligent system formulate the problem it wants to solve as a search problem is not addressed by search.

- Search only shows how to solve the problem once we have it correctly formulated.

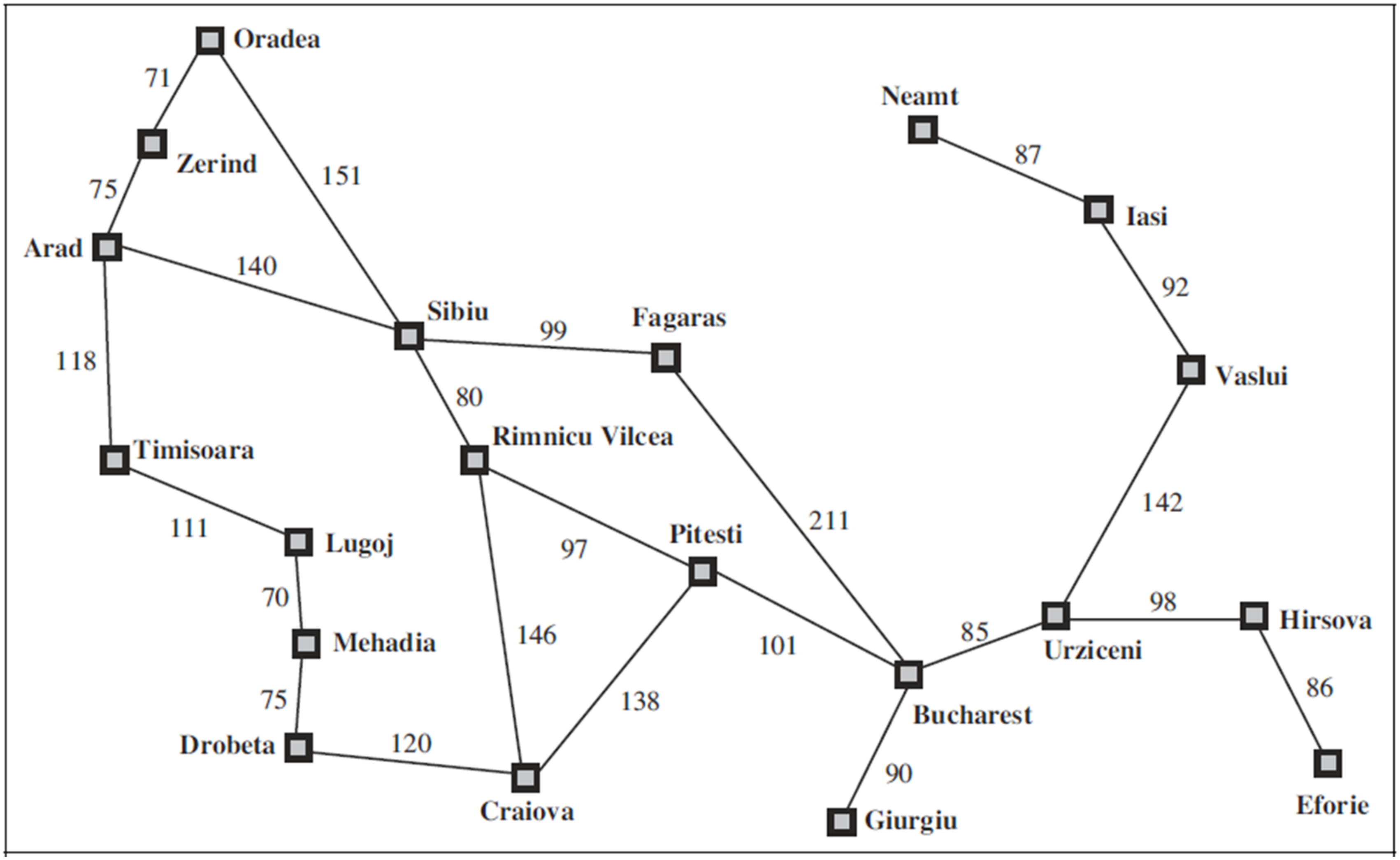
Problem Solving Agents

```
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
```

A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Example: Romania



Example: Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
  – be in Bucharest
- Formulate problem:
  – states: various cities
  – actions: drive between cities or choose next city
- Find solution:
  – sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem Types

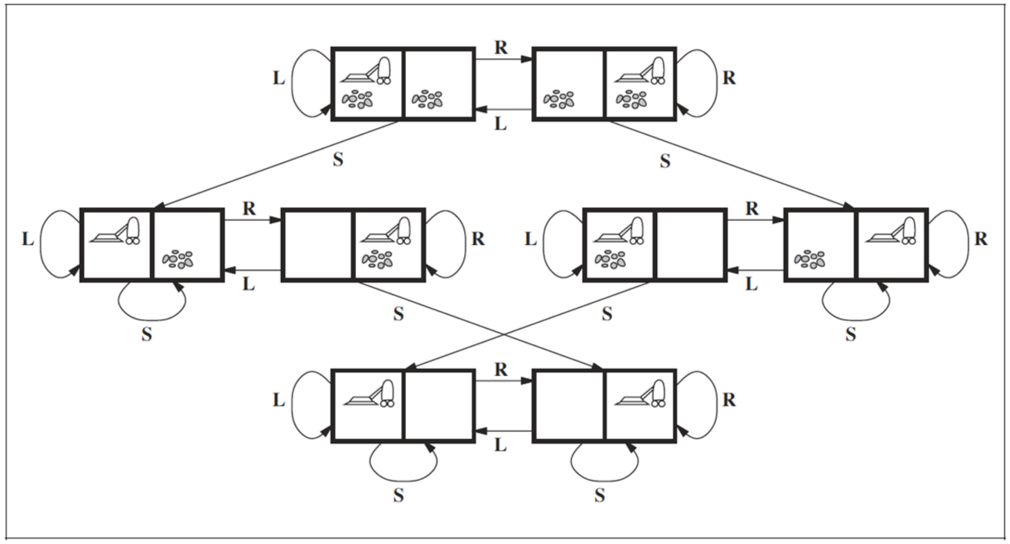
- **Static / Dynamic**
  - Previous problem was static: no attention to changes in environment
- **Observable / Partially Observable / Unobservable**
  - Previous problem was observable: it knew initial state.
- **Deterministic / Stochastic**
  - Previous problem was deterministic: no new percepts were necessary, we can predict the future perfectly given our actions
- **Discrete / continuous**
  - Previous problem was discrete: we can enumerate all possibilities

Representing a Problem: The Formalism

To formulate a problem as search problem we need the following components:

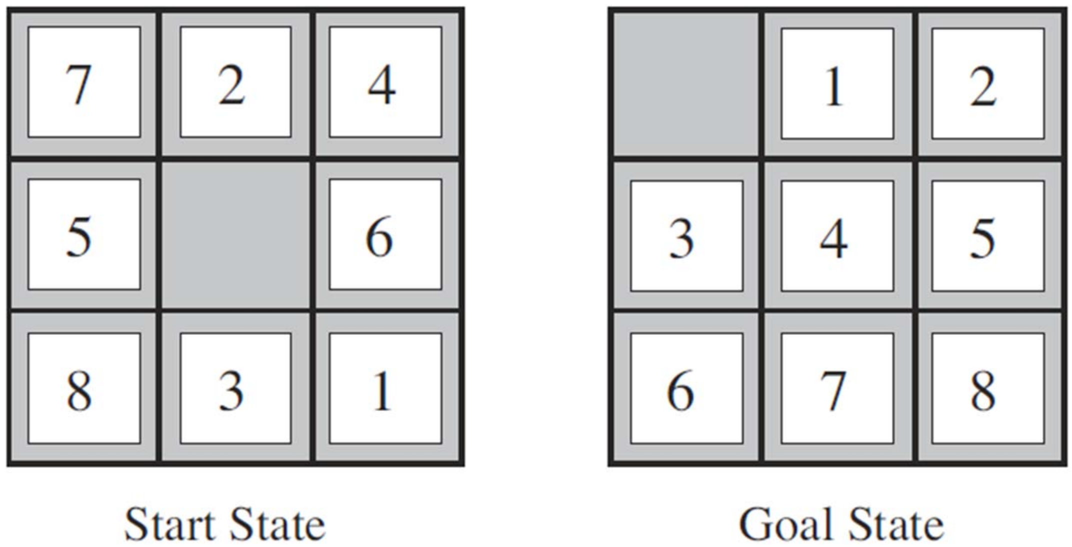
- **States:** Set of all states reachable from the initial state by any sequence of actions.
- **Initial State:** State that the agent starts in.
- **Actions:** A description of the possible actions available to the agent.
- **Goal Test:** Determines whether a given state is a goal state.
- **Path Cost:** Assigns a numeric cost to each path.

Example: Vacuum World



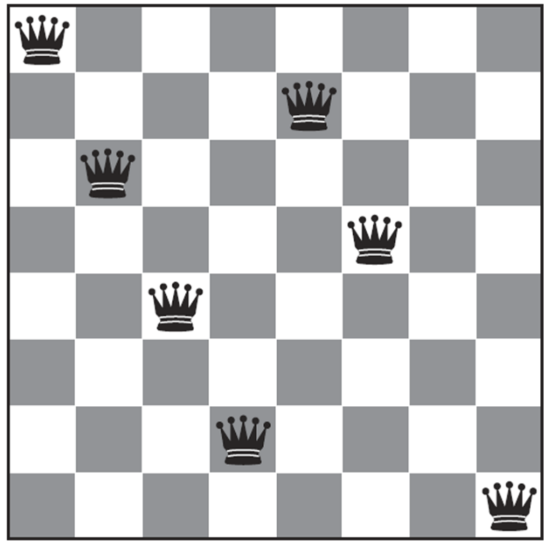
- **States:** agent location and the dirt locations.
- **Initial State:** Any state.
- **Actions:** *Left, Right,* and *Suck.*
- **Goal Test:** No dirt at all location.
- **Path Cost:** Each step costs 1.

Example: 8 - puzzle



- **States:** Locations of tiles.
- **Initial State:** Any state.
- **Actions:** *Left, Right, Up,* or *Down.*
- **Goal Test:** Goal configuration.
- **Path Cost:** Each step costs 1.

Example: 8 queens



- **States:** Arrangements of n <= 8 queens, one per columns, with no queen attacks any other.
- **Initial State:** No queens on the board.
- **Actions:** Add queen to leftmost empty square such that it is not attacked by other queens.
- **Goal Test:** 8 queens on the board, none attacked.
- **Path Cost:** Each move costs 1.

Implementation for Search Algorithms

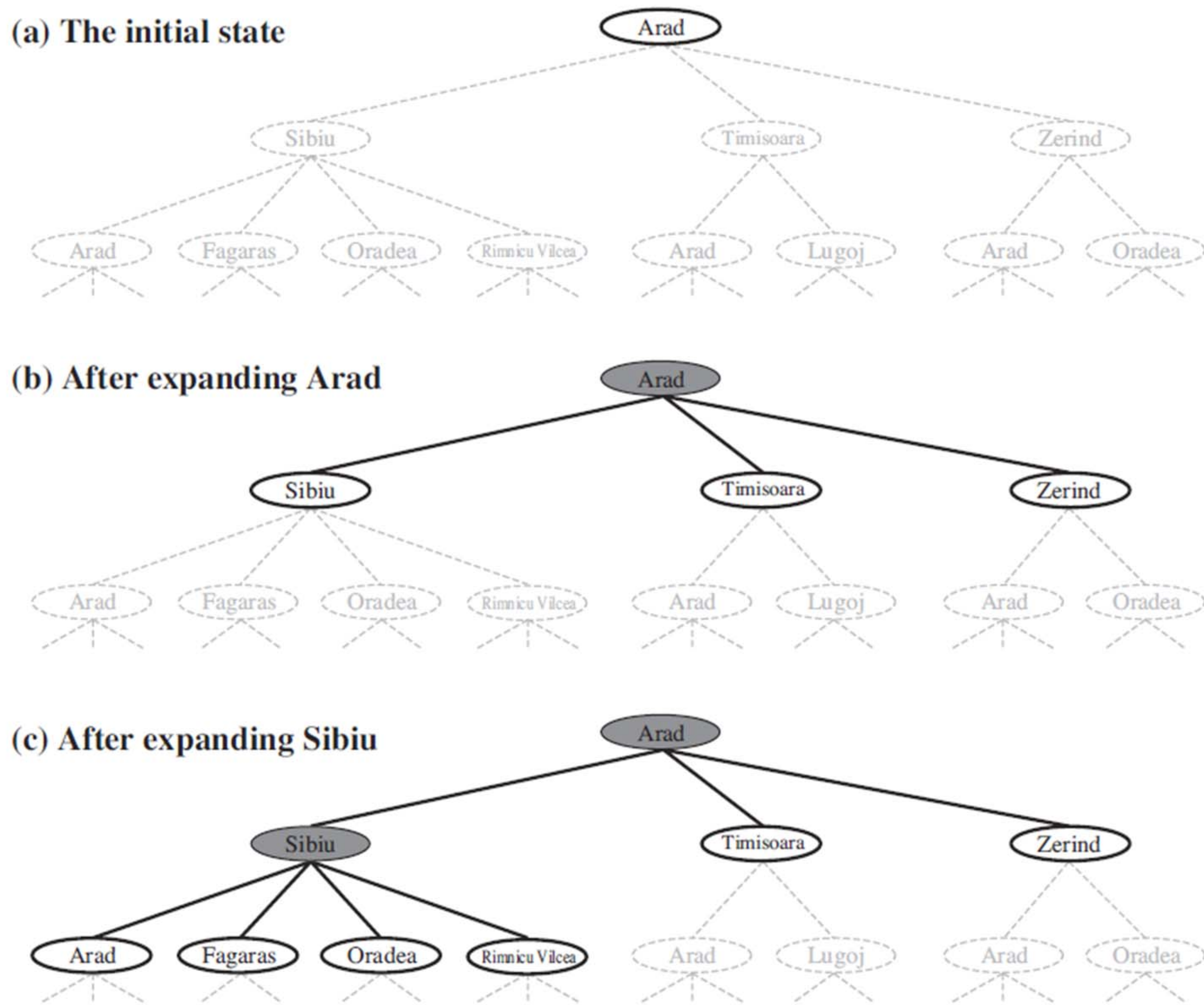
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return**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

An informal description of the general tree-search and graph-search algorithms.

Implementation for Search Algorithms



Partial search trees for finding a route from Arad to Bucharest.

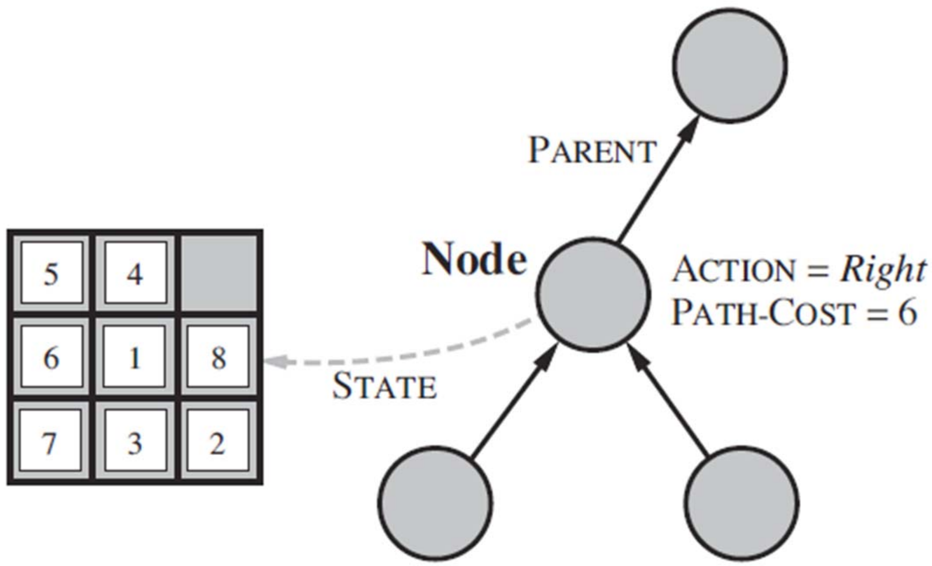
Infrastructure for Search Algorithms

Data Structure:

- *n.*STATE: the state in the state space to which the node corresponds;
- *n.*PARENT: the node *in* the search tree that generated this node;
- *n.*ACTION: the action that was applied to the parent to generate the node;
- *n*.PATH-COST: the cost, traditionally denoted by *g*(n), of the path from the initial state to the node, as indicated by the parent pointers.

```
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)
```

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
- A ***node*** includes: parent, children, depth, path cost g(x).
- States do not have parents, children, depth, or path cost!

Measuring Performance

- A search strategy is defined by 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: maximum branching factor of the search tree
  - d: depth of the least-cost solution
  - m: maximum depth of the state space (may be ∞)

Uninformed Search Strategies

Uninformed 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

Breadth-First Search

- Expand shallowest unexpanded node
- *Frontier* (or fringe): nodes in queue to be explored
- *Frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- *Goal-Test* when inserted.

Breadth-First Search Algorithm

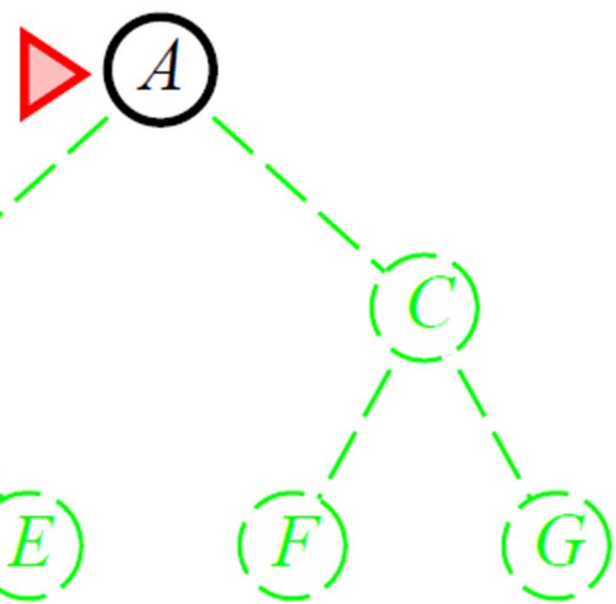
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if 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

Initial state = A
Is A a goal state?

Put A at end of queue.
frontier = [A]

Future = green dotted circles
Frontier = white nodes
Expanded/active = gray nodes
Forgotten/reclaimed = black nodes

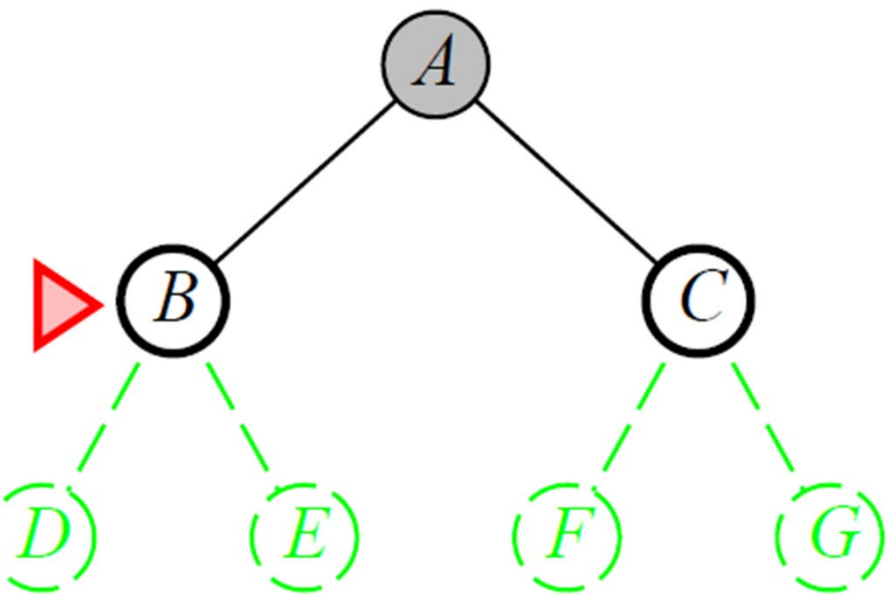


Breadth-First Search

Expand A to B, C.
Is B or C a goal state?

Put B, C at end of queue.
frontier = [B, C]

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

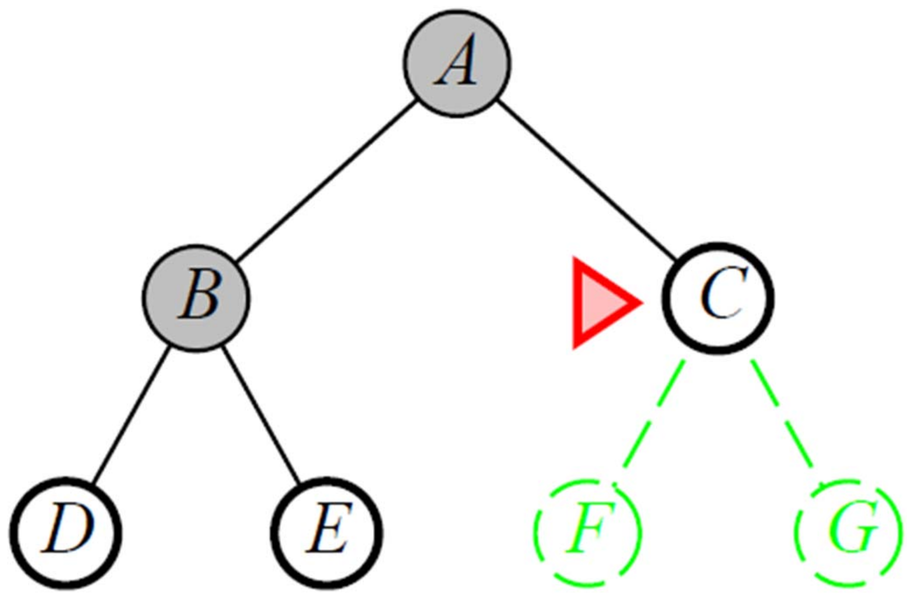


Breadth-First Search

Expand B to D, E
Is D or E a goal state?

Put D, E at end of queue
frontier=[C, D, E]

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

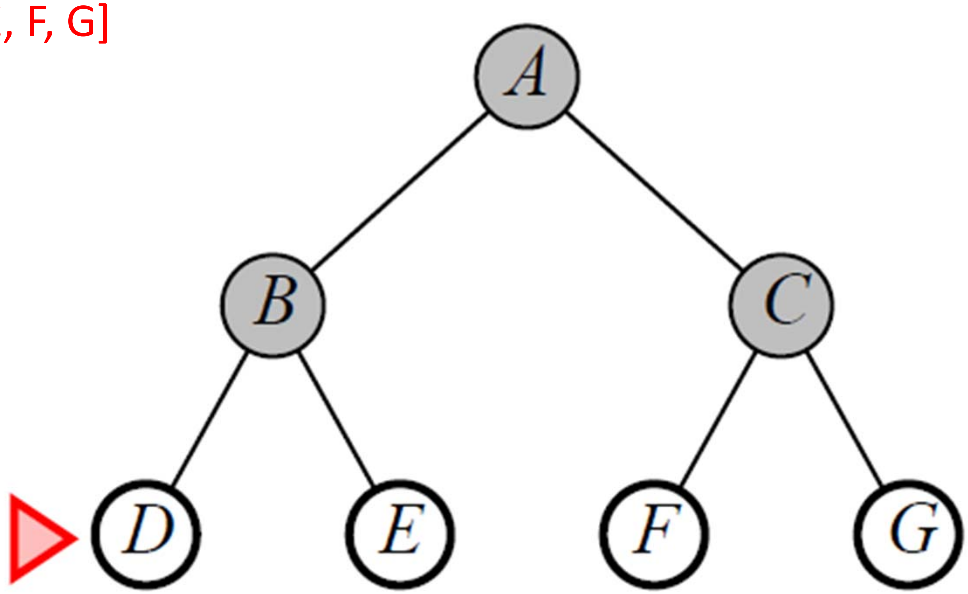


Breadth-First Search

Expand C to F, G.
Is F or G a goal state?

Put F, G at end of queue.
frontier = [D, E, F, G]

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



Properties of Breadth-First Search

- Complete? Yes, it always reaches a goal (if *b* is finite)
- Optimal? No, for general cost functions.

  Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goal nodes are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal
- Time? $1+b+b^2+b^3+\ldots + b^d = O(b^d)$

  (this is the number of nodes when generated; $O(b^{d+1})$ when expanded.)
- Space? $O(b^{d-1})$ in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$

  It is the bigger problem (more than time). can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

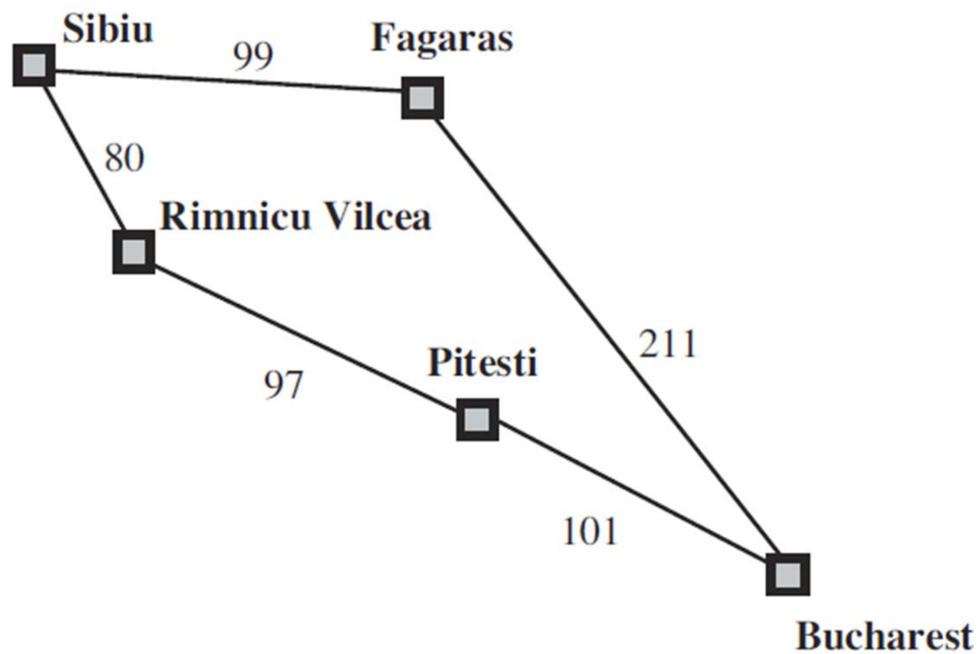
Uniform-Cost Search

- Expand the least-cost unexpanded node
- **Implementation**: frontier is a queue ordered by path cost (priority queue)
- **Equivalent** to breadth-first if step costs are all equal
- **Difference** with breadth-first search
  - Goal test is applied to a node when it is selected for expansion
  - Test is added in case a better path is found to a node currently on frontier

Uniform-Cost Search Algorithm

```
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
```

Uniform-Cost Search Illustration



- Successor of **Sibiu** are **Rimnicu Vilcea** and **Fagaras**
- **Rimnicu Vilcea** is expanded adding **Pitesti**
- **Fagaras** is expanded and adding **Bucharest**
- **Pitesti** is expanded and adding 2nd path to **Bucharest**

Properties of Uniform-Cost Search

- Complete? Yes, if b is finite and step cost ≥ $\varepsilon > 0$.
  (otherwise it can get stuck in infinite loops)

- Optimal? Yes, for any step cost ≥ $\varepsilon > 0$.

- Time? # of nodes with *path cost* ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

- Space? # of nodes with path cost ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

Depth-First Search

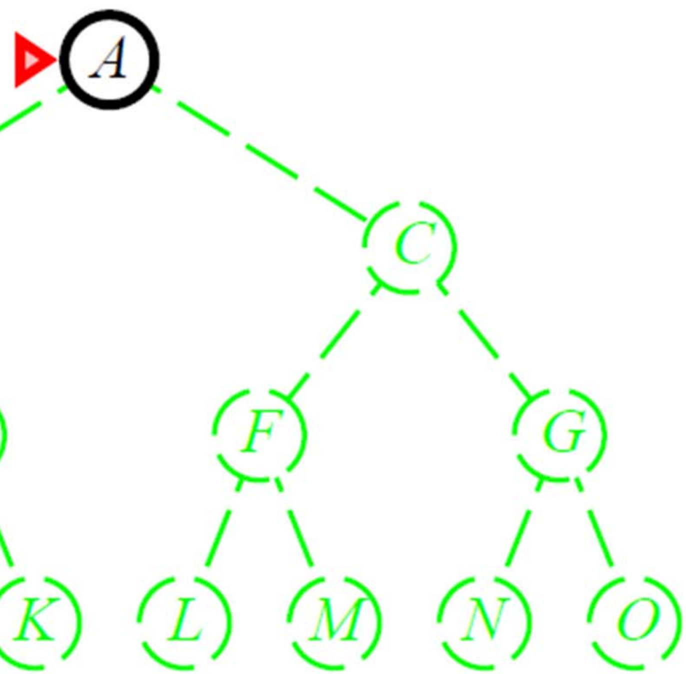
- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* when inserted.

Depth-First Search

Initial state = A
Is A a goal state?

Put A at front of queue.
frontier = [A]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

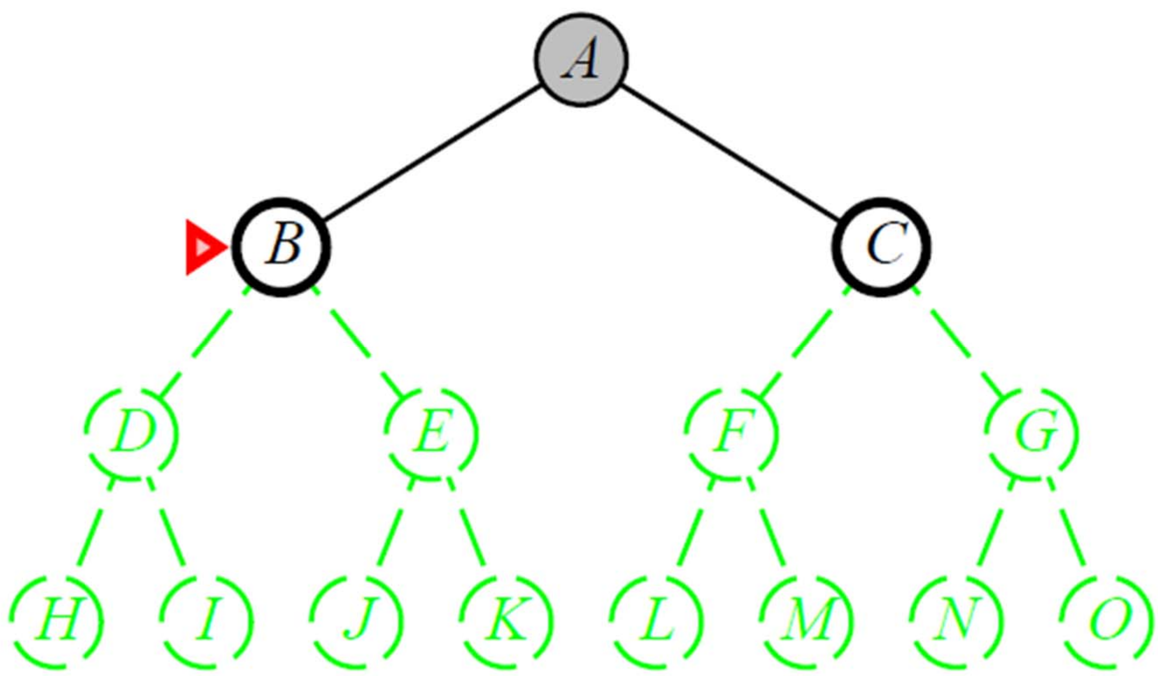


Depth-First Search

Expand A to B, C.
Is B or C a goal state?

Put B, C at front of queue.
frontier = [B, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

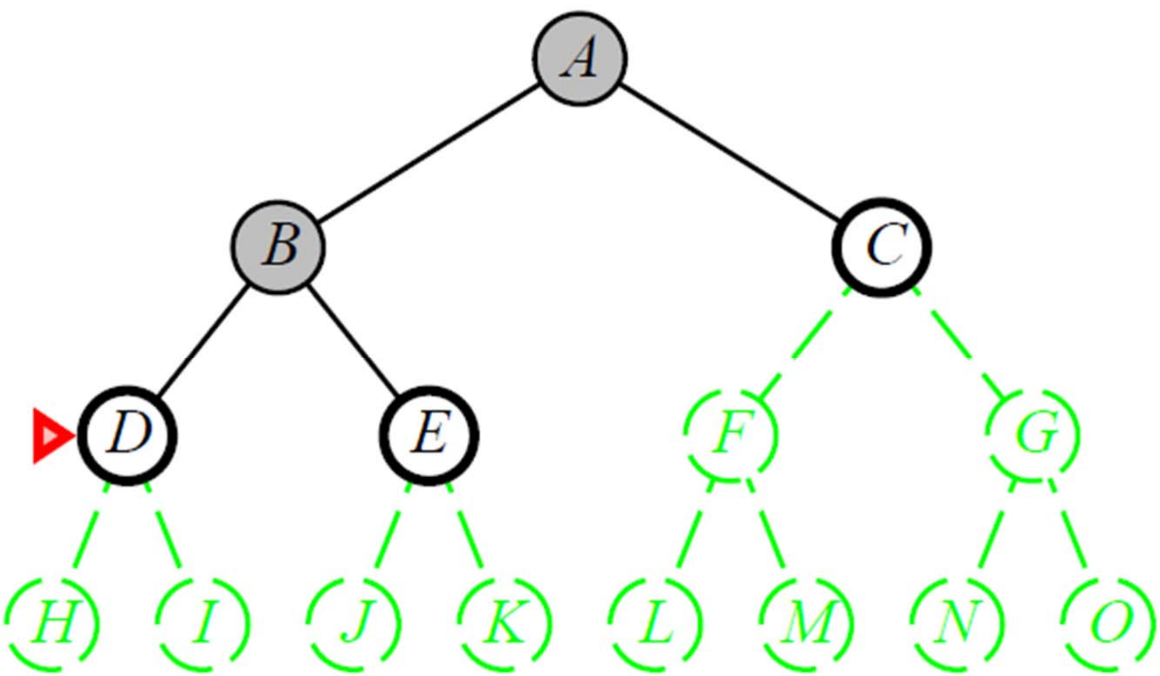


Depth-First Search

Expand B to D, E.
Is D or E a goal state?

Put D, E at front of queue.
frontier = [D, E, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

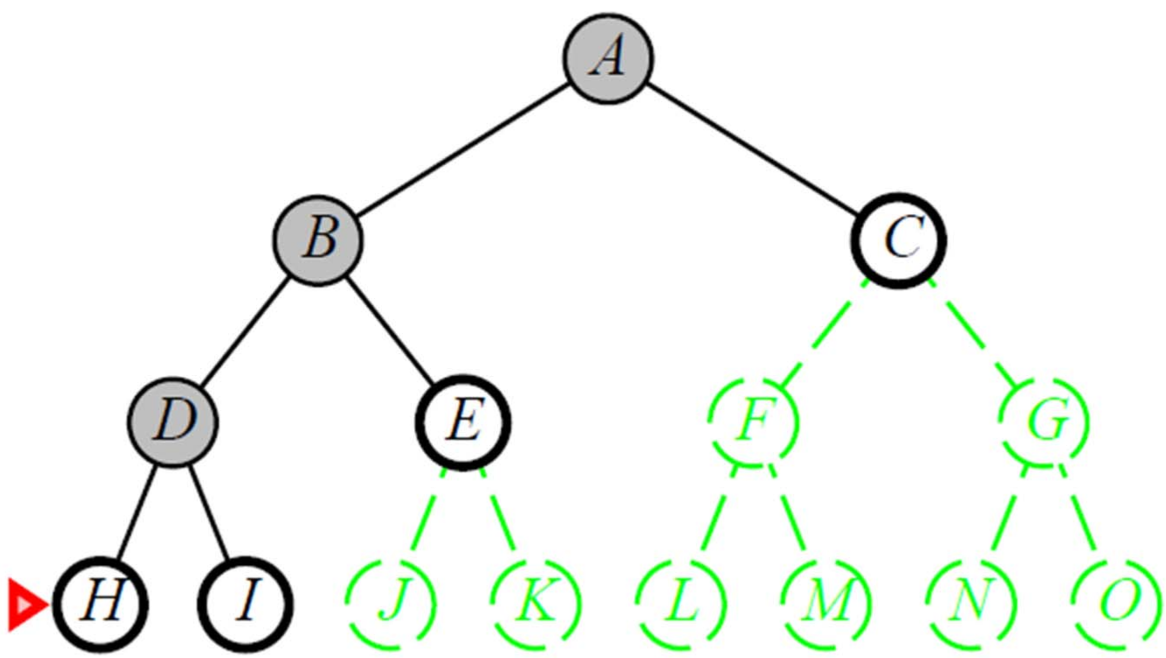


Depth-First Search

Expand D to H, I.
Is H or I a goal state?

Put H, I at front of queue.
frontier = [H, I, E, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

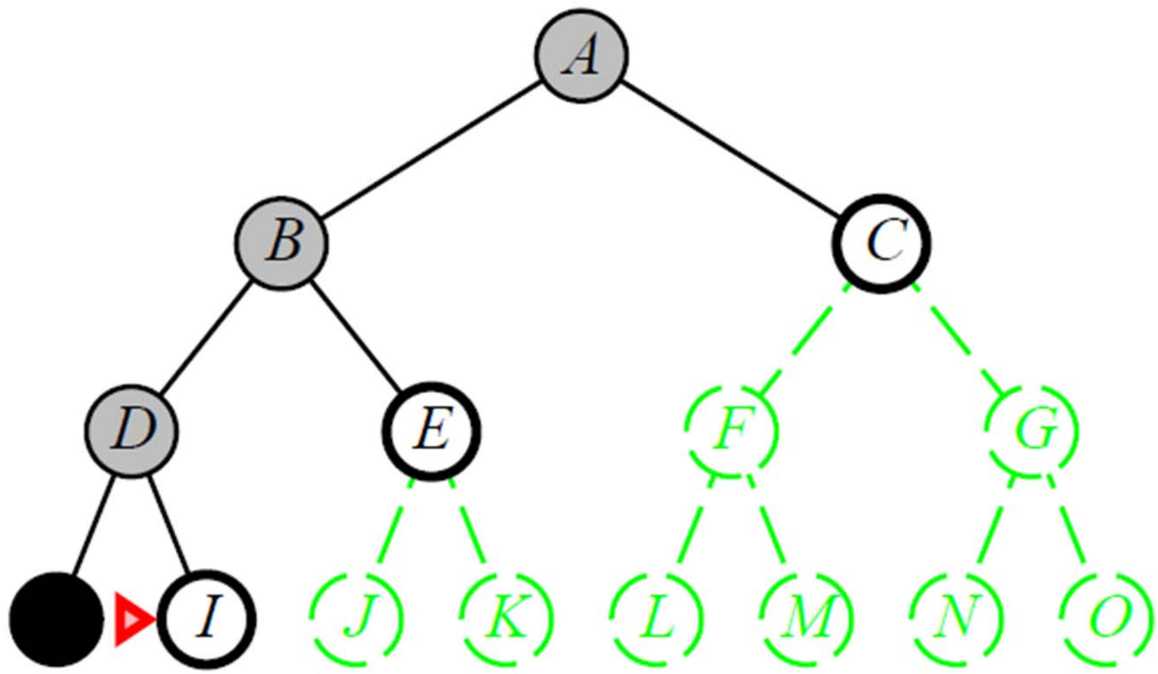


Depth-First Search

Expand H to no children.
Forget H.

frontier = [I, E, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

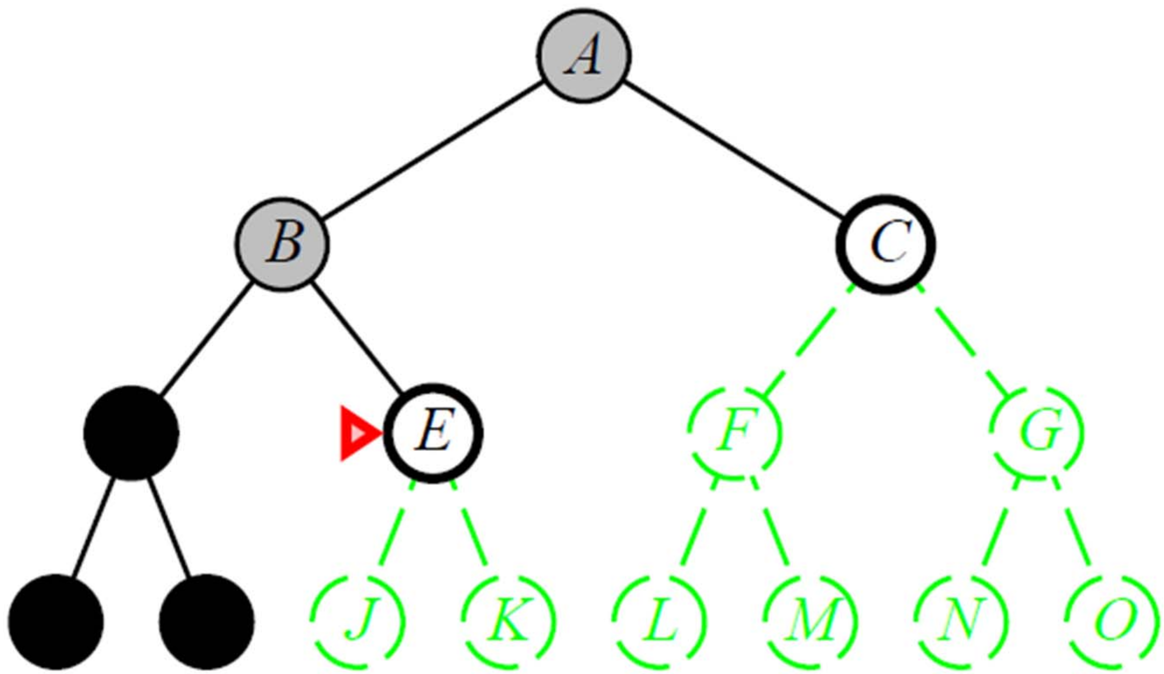


Depth-First Search

Expand I to no children.
Forget D, I.

frontier = [E, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

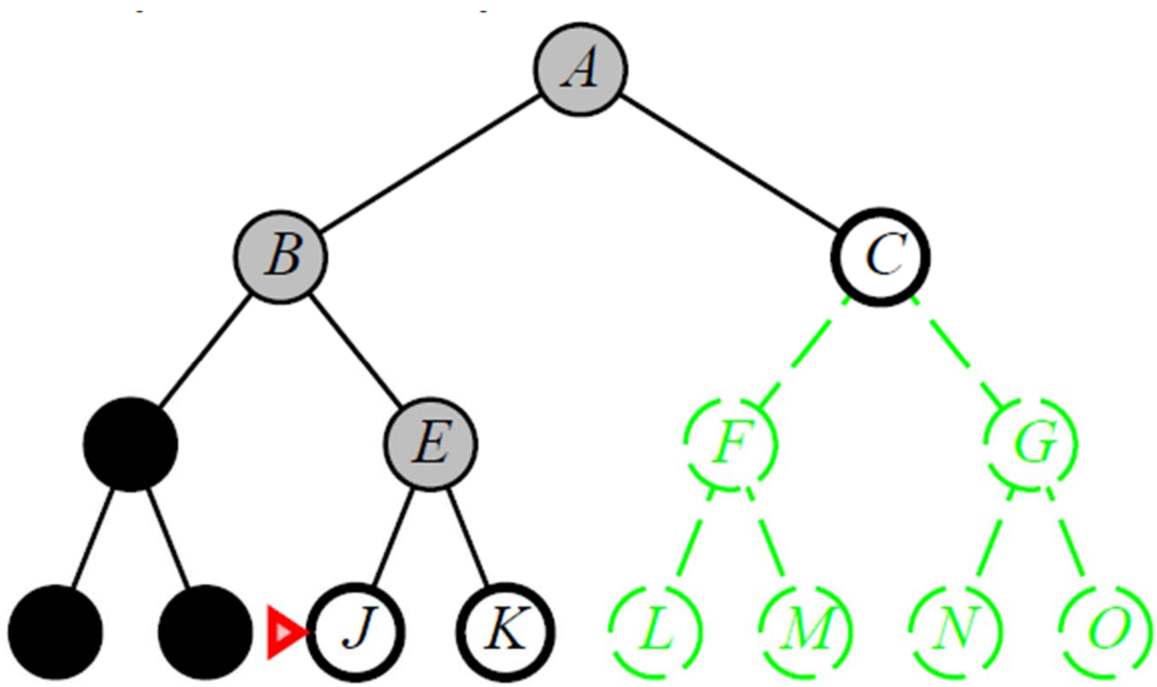


Depth-First Search

Expand E to J, K.
Is J or K a goal state?

Put J, K at front of queue.
frontier = [J, K, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

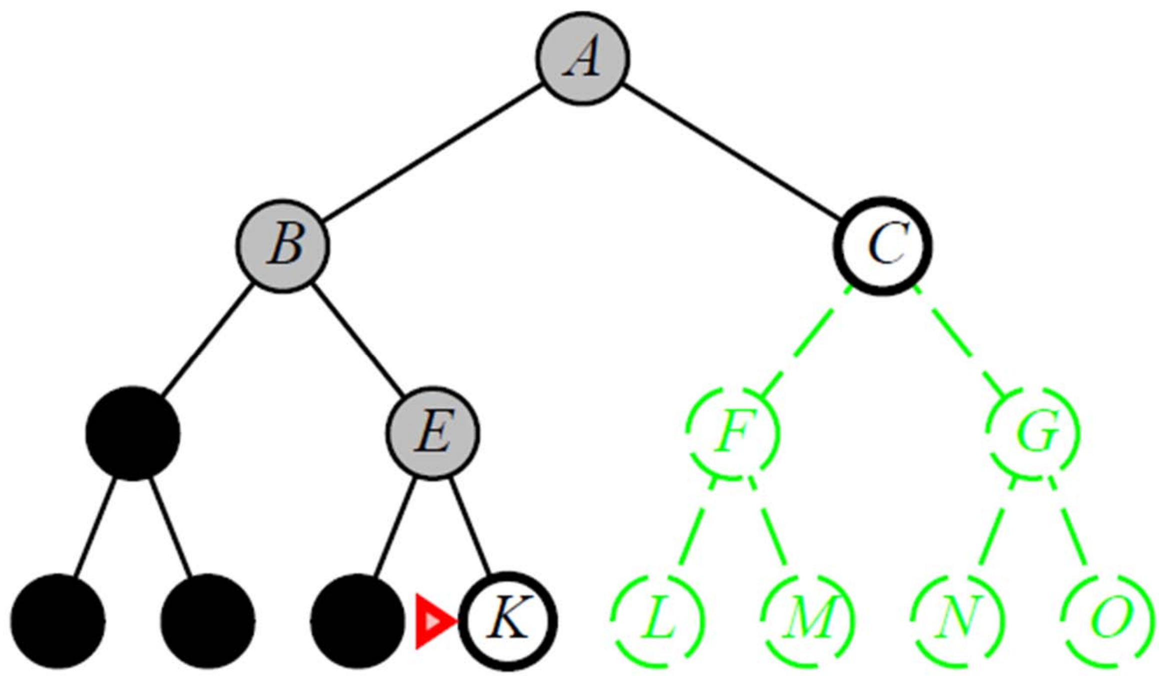


Depth-First Search

Expand J to no children.
Forget J.

frontier = [K, C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

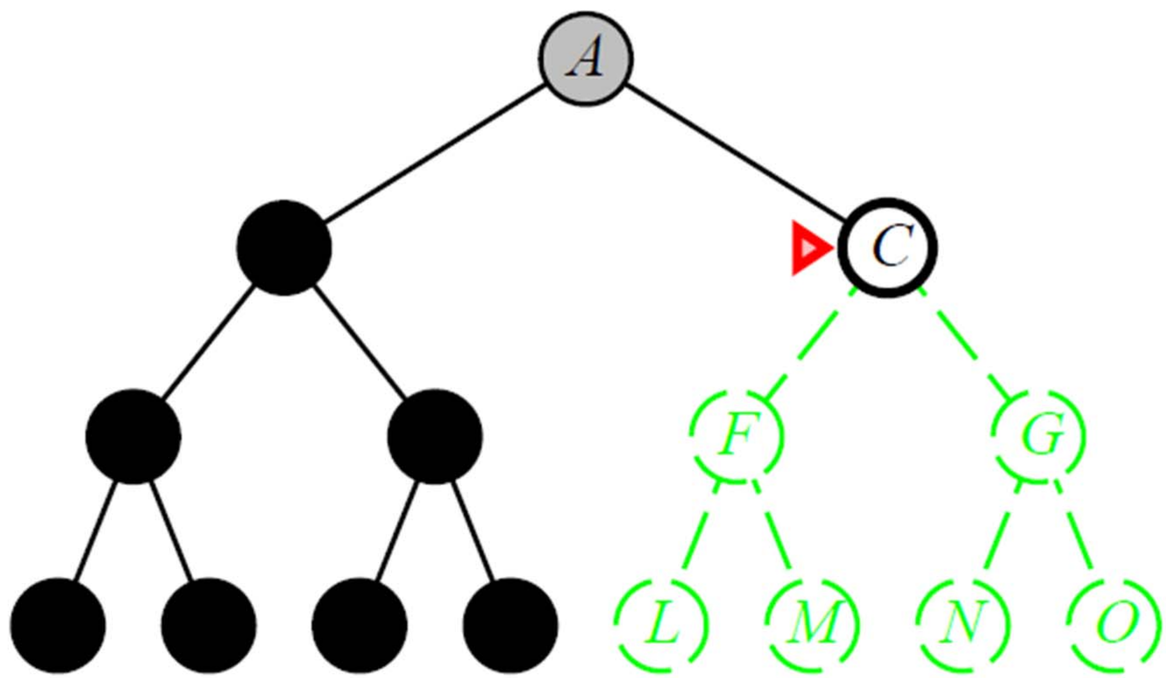


Depth-First Search

Expand K to no children.
Forget B, E, K.

frontier = [C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

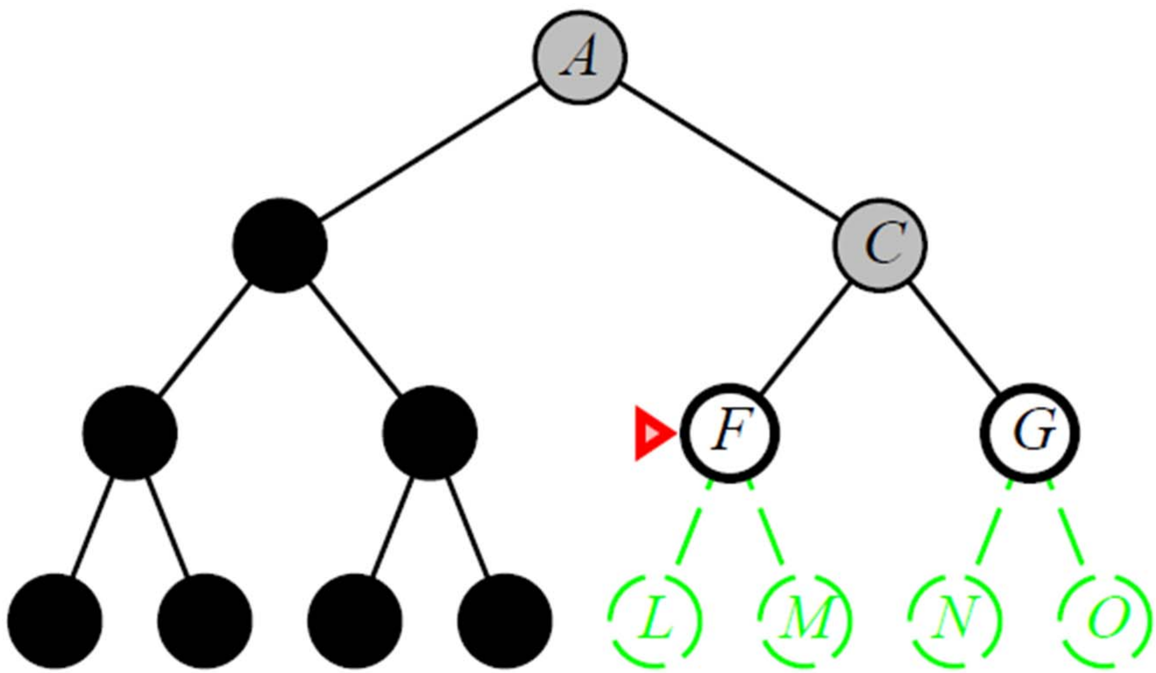


Depth-First Search

Expand C to F, G.
Is F or G a goal state?

Put F, G at front of queue.
frontier = [F, G]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

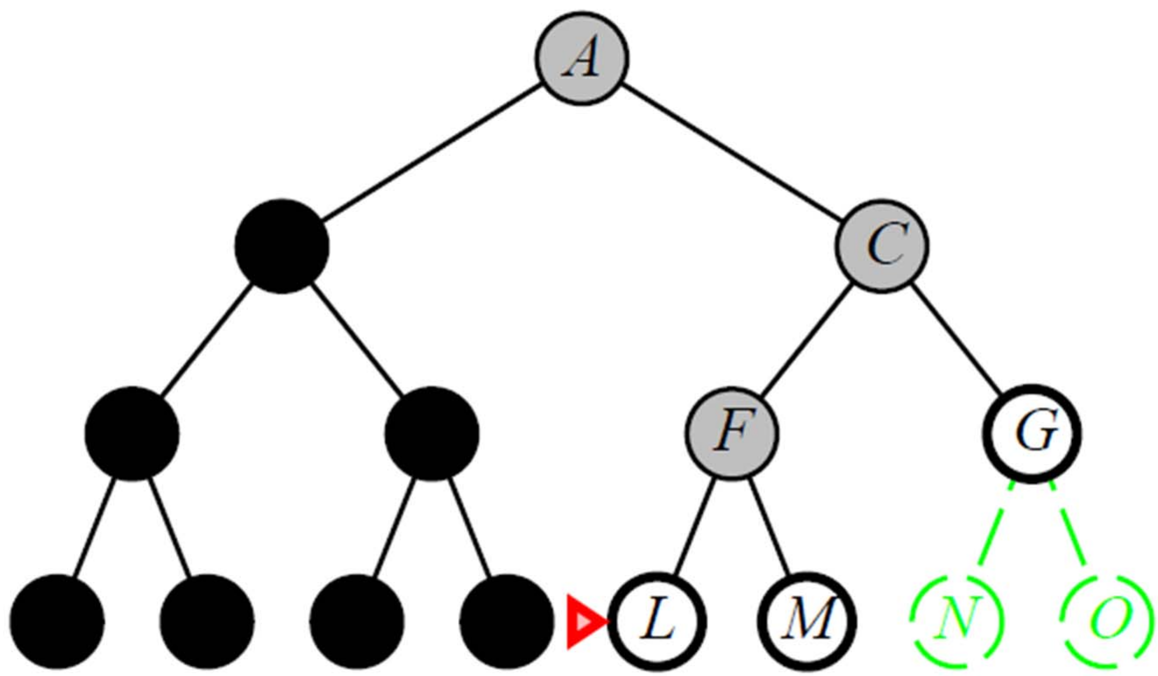


Depth-First Search

Expand F to L, M.
Is L or M a goal state?

Put F, G at front of queue.
frontier = [L, M, G]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

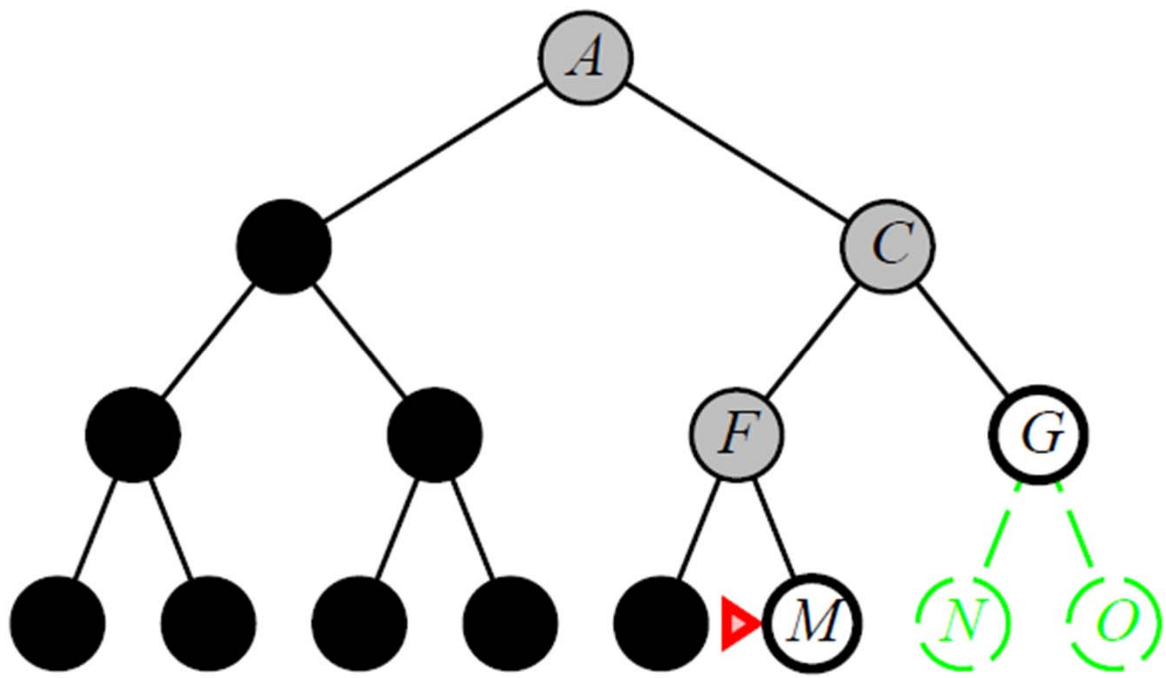


Depth-First Search

Expand L to no children
Forget L

Put F, G at front of queue.
frontier = [M, G]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes



Depth-First Search

- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Complete in finite spaces
- Optimal? No: It may find a non-optimal goal first
- Time? $O(b^m)$ with m = maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes