Artificial Intelligence

Solving Problems by searching

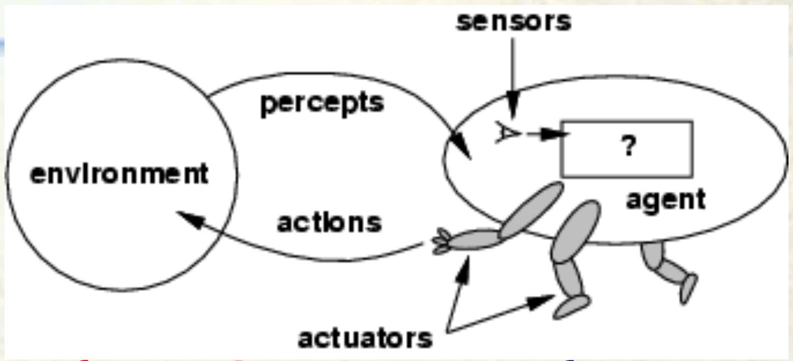
Outline

- Agent & Environment
- Agent types
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Uninformed search algorithms

Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators
- Human agent: eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators
- Robotic agent: cameras and infrared range finders for sensors; various motors for actuators

Agents and environments

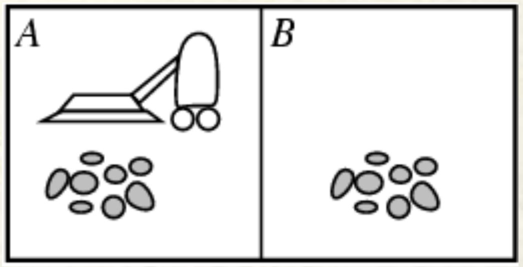


- The agent function maps from percept histories to actions:

$$[f: \mathcal{P}^* \rightarrow \mathcal{A}]$$

- The agent program runs on the physical architecture to produce *f*
- agent = architecture + program

Vacuum-cleaner world



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

A vacuum-cleaner agent

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean]$, $[A, Clean]$	*Right*
$[A, Clean]$, $[A, Dirty]$	*Suck*
⋮	⋮

```
function REFLEX-VACUUM-AGENT([location, status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Rational agents

- An agent should strive to "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful
- Performance measure: An objective criterion for success of an agent's behavior
- E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

Rational agents

- Rational Agent: For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

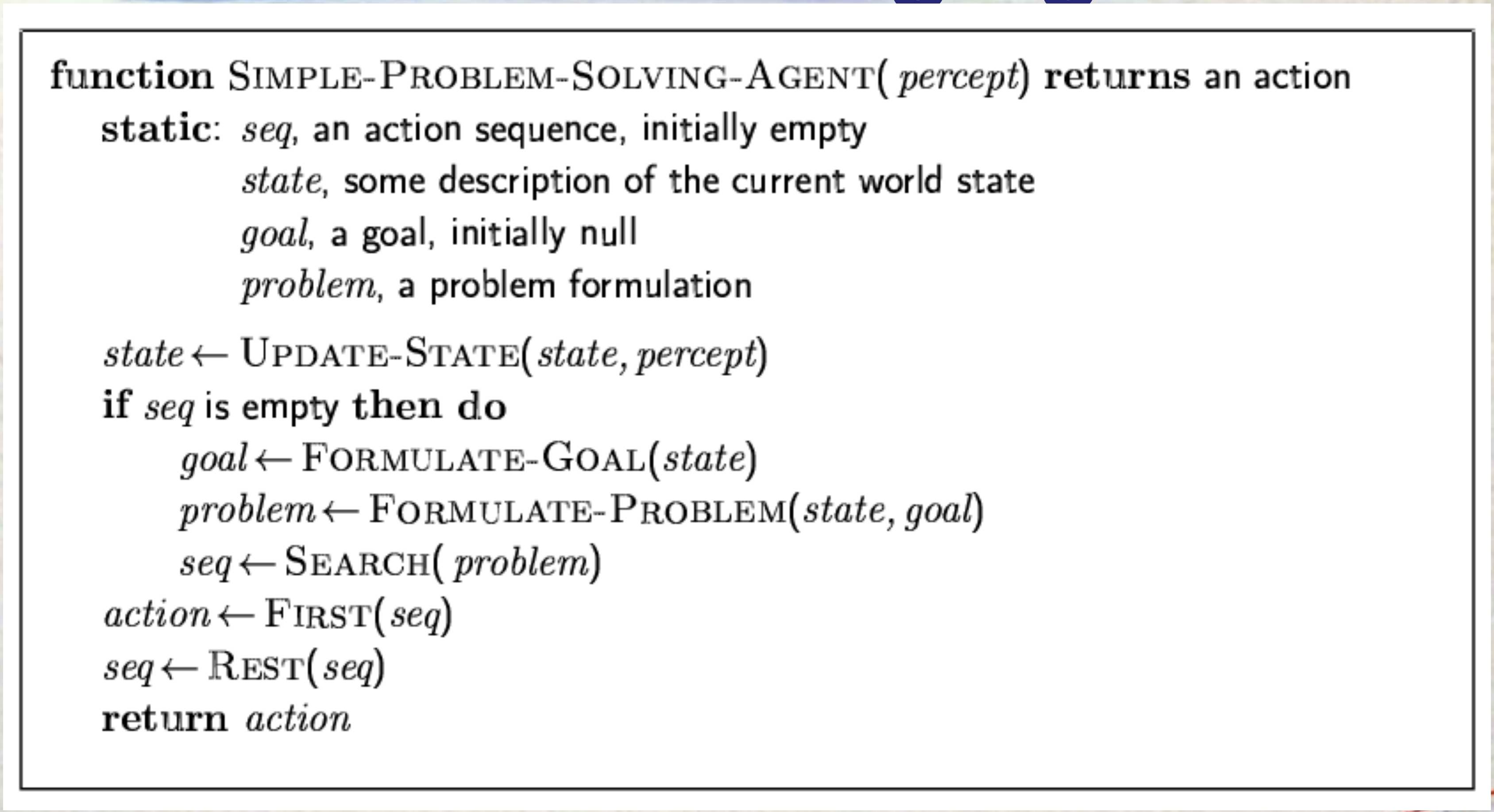
Reflex Agents

- Choose action based on current percept (and maybe memory)
- May have memory or a model of the world's current state
- Do not consider the future consequences of their actions
- Consider how the world currently is!
- Can a reflex agent be rational?

Planning Agents

- Ask "what if"
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Must formulate a goal
- Consider how the world would be!
- Planning agents search

Problem-solving agents

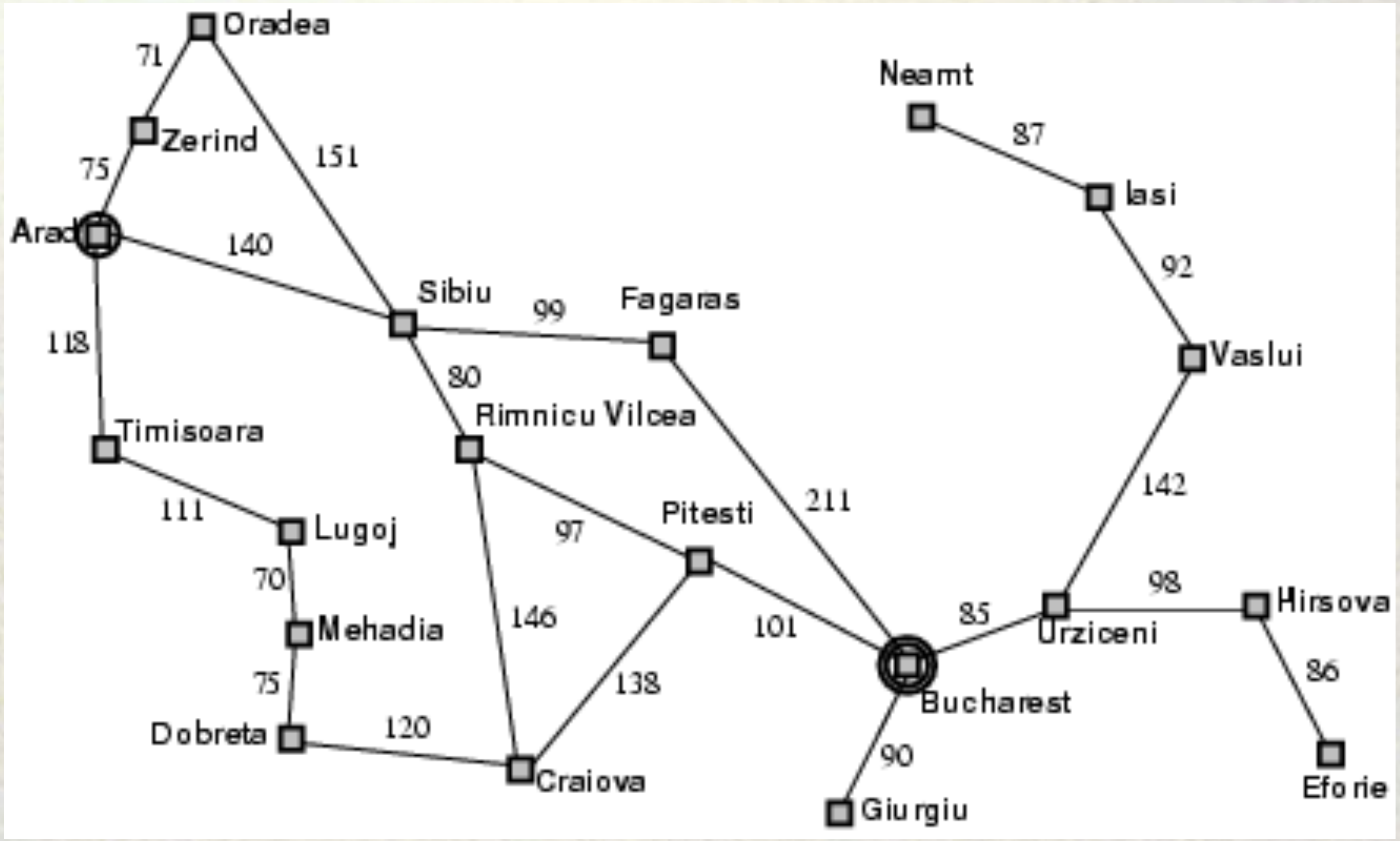


```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

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

Search problem

- A state space: an abstraction of the world, it encodes how the world is at a certain point
- Successor function: models how the world works
- A start state and a goal test

Example: Romania



Example: Romania

- On touring holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest; non-refundable ticket.

1. Formulate goal: be in Bucharest on time.
2. Specify task:
   1. States: various cities
   2. Operations or actions (= transitions between states): drive between cities
3. Find solution (= action sequences): sequence of cities: e.g. Arad, Sibiu, Fagaras, Bucharest.
4. Execute: drive through all the cities given by the solution.

Single-State Task Specification

A task is specified by states and actions:

- Initial state e.g. “at Arad”
- State space e.g. other cities
- Actions or operators (or successor function) e.g. Arad -> Zerind Arad-> Sibiu
- Goal test, check if a state is goal state
  - Explicit x = “at Bucharest”
  - Implicit x = NoDirt(x)
- Path cost e.g. sum of distances, number of actions
- Total cost = search cost + path cost
- A solution is a state-action sequence (initial to goal state)

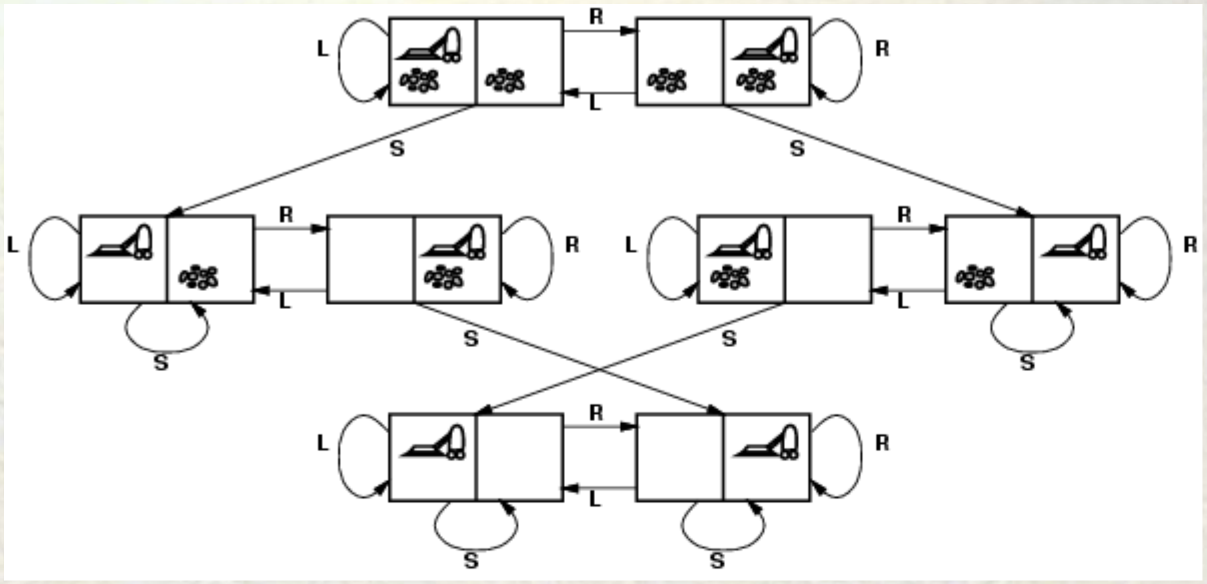
Choosing States and Actions

- Real world is absurdly complex: state space must be abstracted for problem solving.
- (abstract) state = set of real states
- (abstract) action = complex combination of real actions:
  - E.g. Arad -> Zerind represents a complex set of possible routes, detours, rest stops etc.
  - For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (abstract) solution = set of real paths that are solutions in the real world.

Example Problem

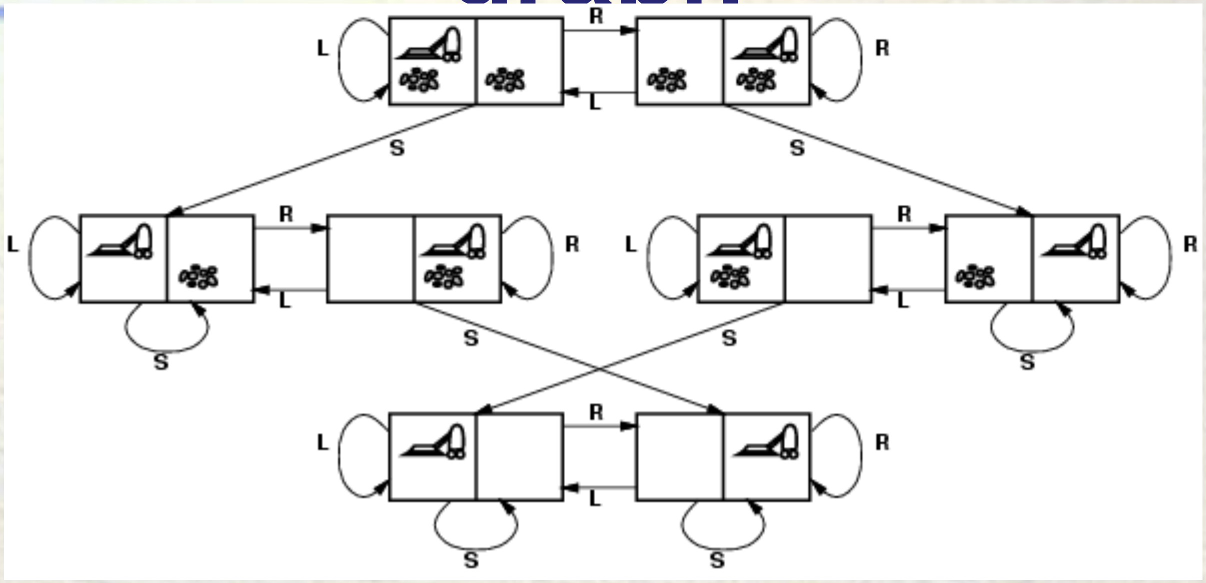
- Toy problems: concise exact description
- Real world problems: don’t have a single agreed description.

Vacuum world state space graph



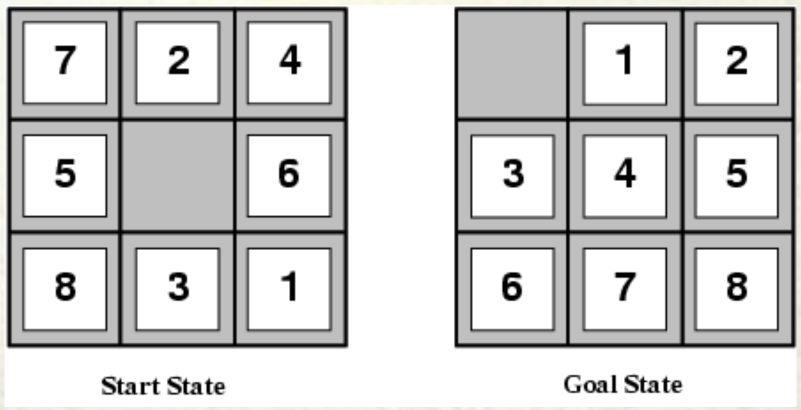
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



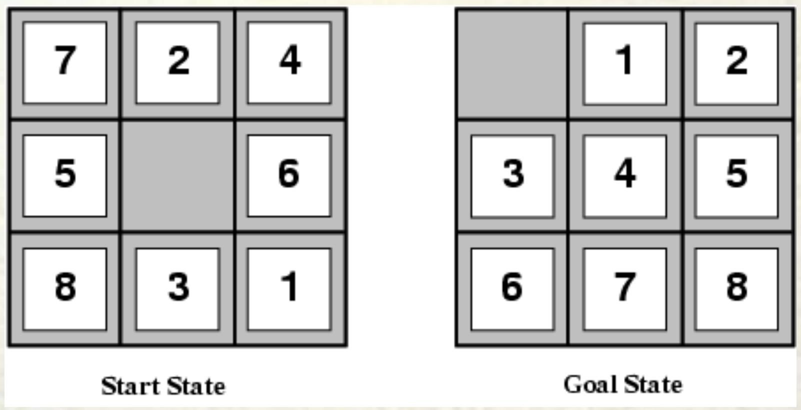
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

The 8-Puzzle



- States:?
- Operators: ?
- Goal test: ?
- Path cost: ?

The 8-Puzzle



- States: integer locations of tiles
- Operators: move blank left, right, up, down.
- Goal test: goal state (given)
- Path cost: 1 per move

Rubik’s Cube

- States: ?
- Operators: ?
- Goal test: ?
- Path cost: ?

Path Search Algorithm

- Search: Finding state-action sequences that lead to desirable states. Search is a function

  solution search(task)

- Basic idea:

Simulated explorations of state space by generating successors of already-explored states (i.e. “expanding” them).

Path search vs. Constraint Satisfaction

Important difference between Path Search Problems and CSP:

- Constraint Satisfaction Problems (e.g. n-Queens)
  - Difficult part is knowing the final state
  - How to get there is easy
- Path search problem (e.g. Rubik's cube):
  - Knowing the final state is easy
  - Difficult part is how to get there.

Generating action sequences

- Start with the initial state.
- Test if it is a goal state
- Expand one of the states
- If there are multiple possibilities, you must make a choice.
- Procedure: choosing, testing and expanding until a solution is found or there are no more states to expand.
- Think of it as building up a search tree.

Search tree

- Search tree: superimposed over a state space
- Root: search node corresponding to the initial state
- Leaf node: correspond to states that have no successors in the tree because they were not expanded or generated no new nodes
- State space is not the same as search tree:
  - There are 20 states = 20 cities in the route finding example
  - But there are infinitely many paths.

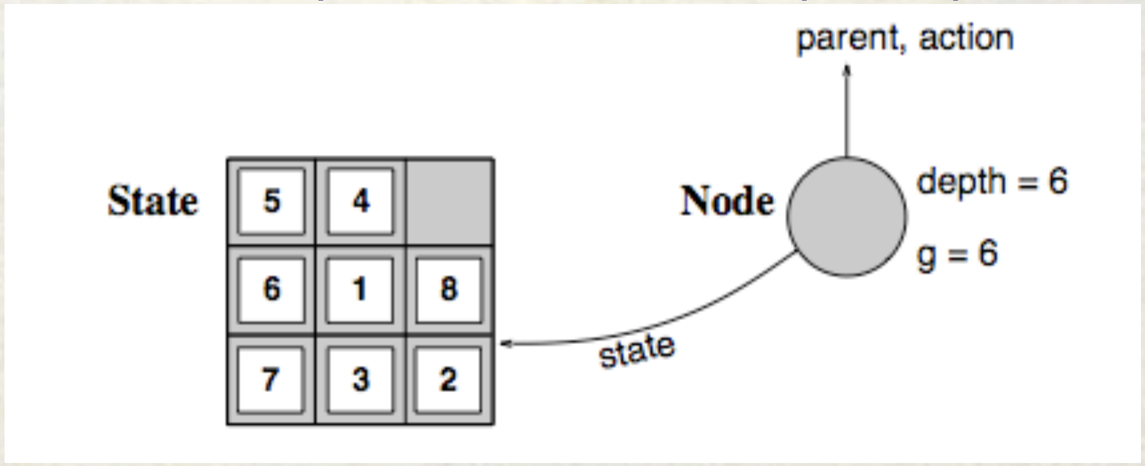
Data structures for a node

One possibility is to have a node structure with five component:

- Corresponding state
- Parent node: the node which generated the current node
- Operator: that was applied to generate the current node
- Depth: number of nodes from the root to the current node
- Path cost

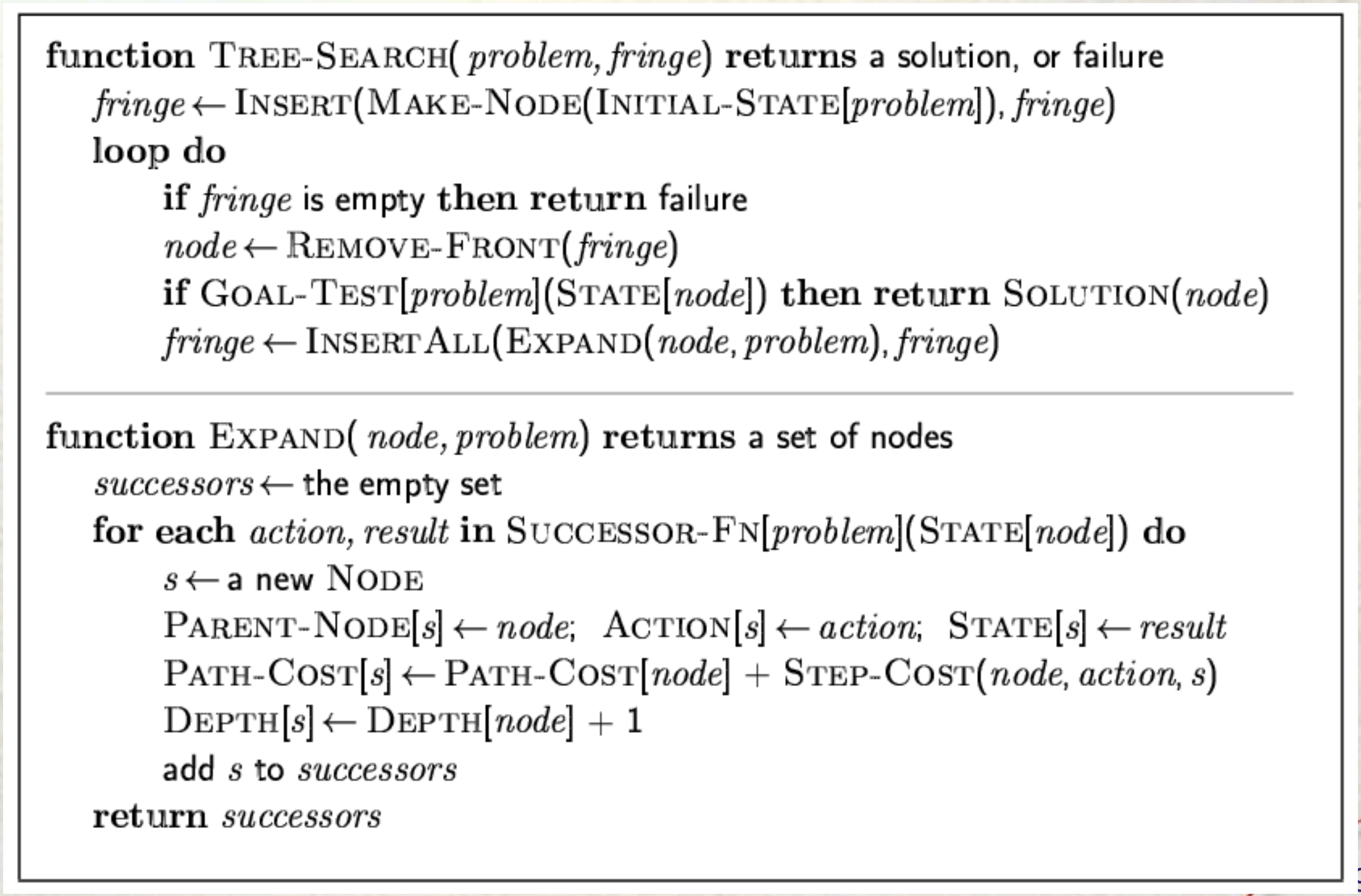
States vs. Nodes

- A state is a representation of a physical configuration
- A node is a data structure constituting part of a search tree including parent, children, depth, path cost
- States do not have parents, children, depth or path cost g(x)



- Note: Two different nodes can contain the same state.

Implementation: general tree search



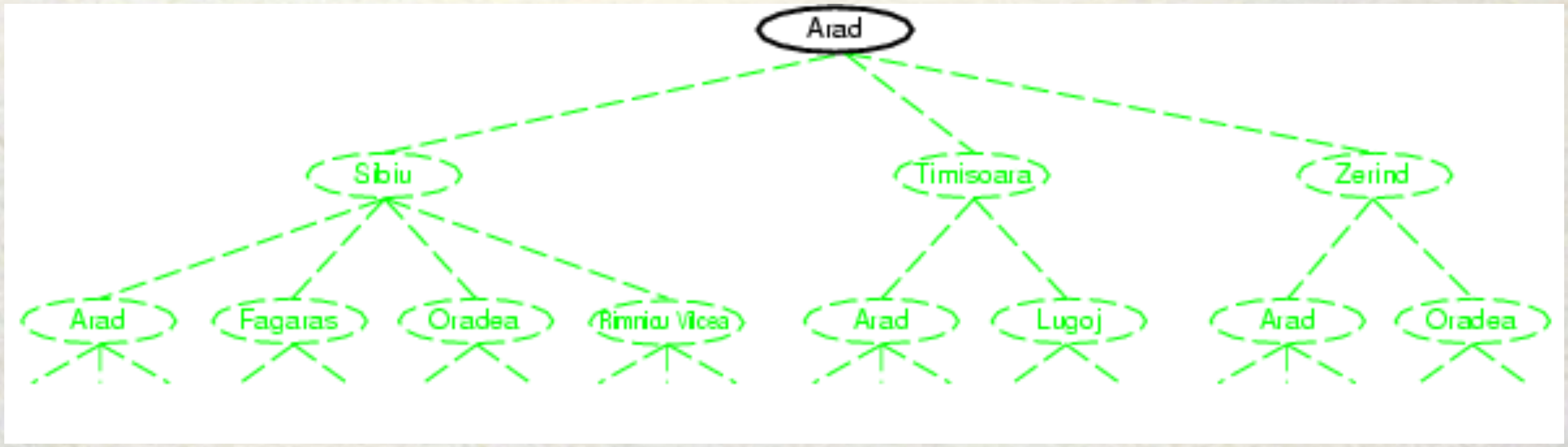
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        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
```

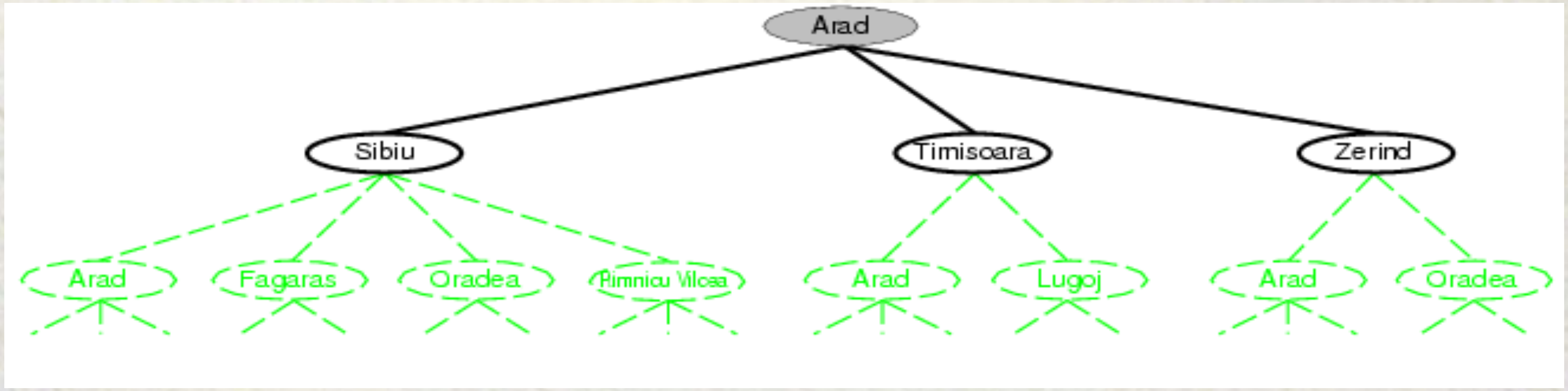
General Tree Search

- Important ideas:
  - Fringe: partial plans under consideration
  - Expansion
  - Exploration strategy
- Main questions: which fringe nodes to explore?

Search Tree

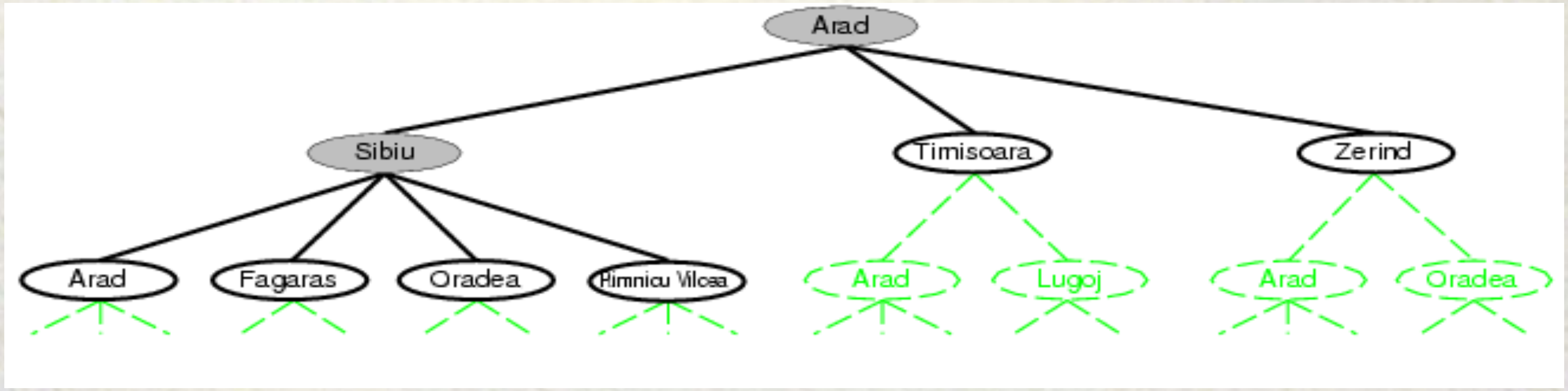

Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Search Tree



Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Search Tree



Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* 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

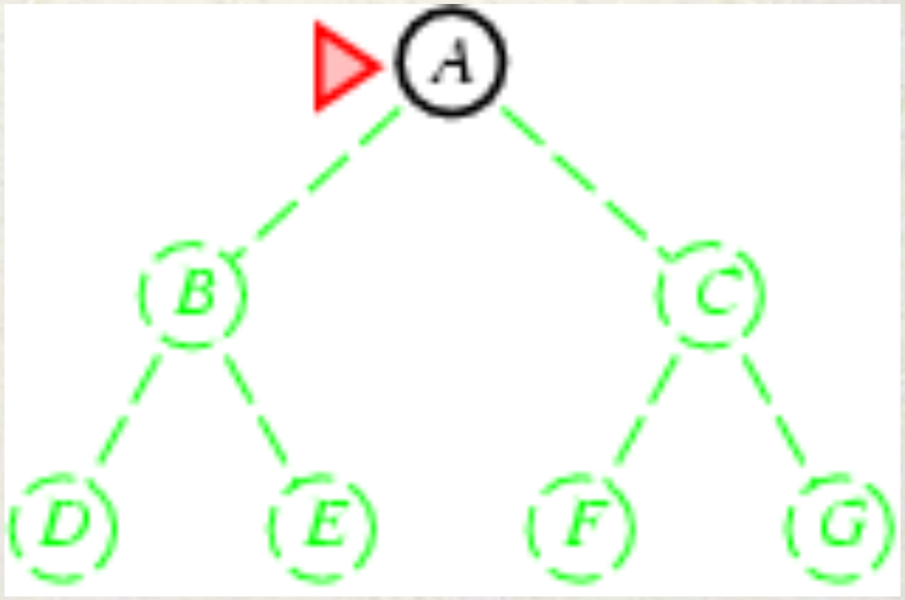
- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

- All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded.
- Expand root first, then all nodes generated by root, then all nodes generated by those nodes etc.
- Expand shallowest unexpanded node
- Implementation: put newly generated successors at the end of the queue
- Very systematic
- Finds the shallowest goal first

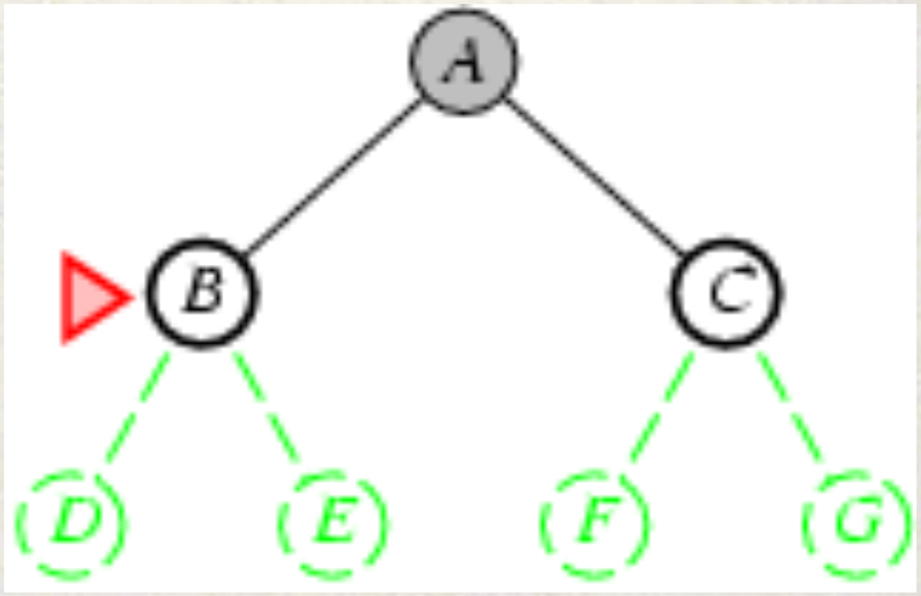
Breadth-first search

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



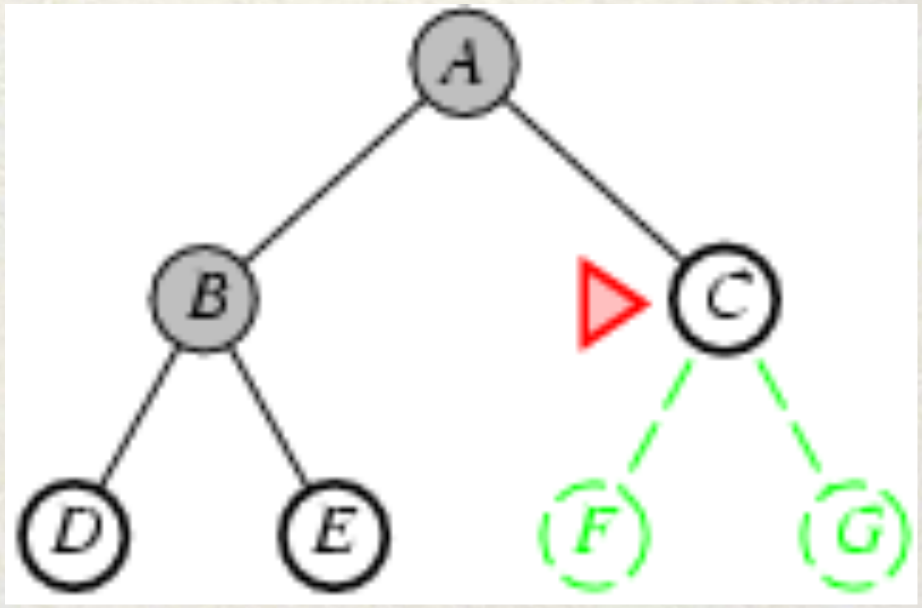
Breadth-first search

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



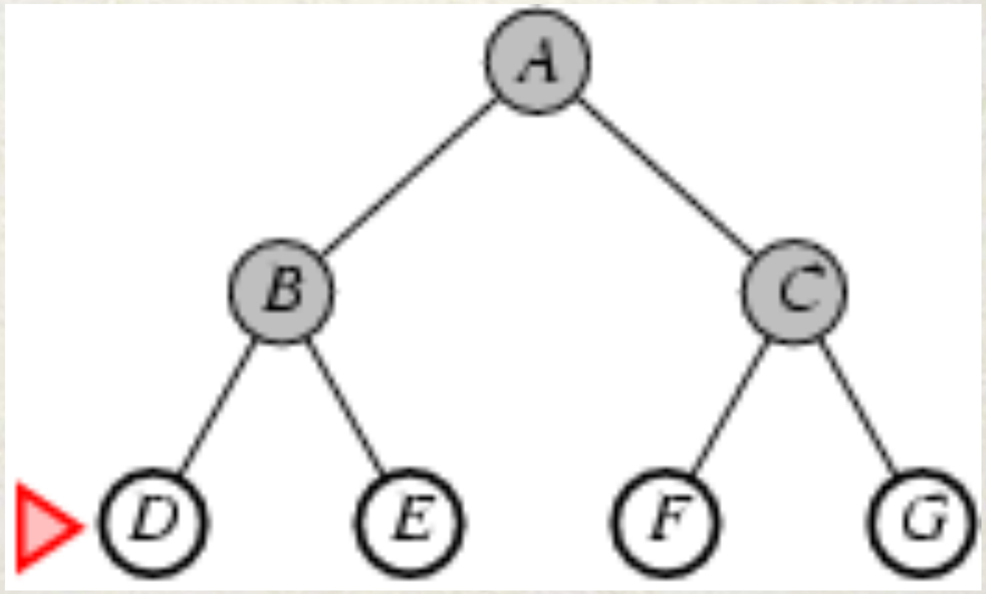
Breadth-first search

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



Breadth-first search

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



Properties of BFS

- Complete? Yes (if b is finite, the shallowest goal is at a fixed depth d and expanded)
- Time? $b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space? $b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$

(keeps every node in memory; expand all but the last node (goal) at level d -> $(b^{d+1} - b)$ nodes at level d+1)

- Optimal? Yes (if all actions have same cost)

- Space is the bigger problem (more than time) ; it grows exponentially with depth.

Properties of BFS

- Giả sử b=10, kiểm tra 1000 trạng thái cần 1 giây, lưu một trạng thái cần 100 byte

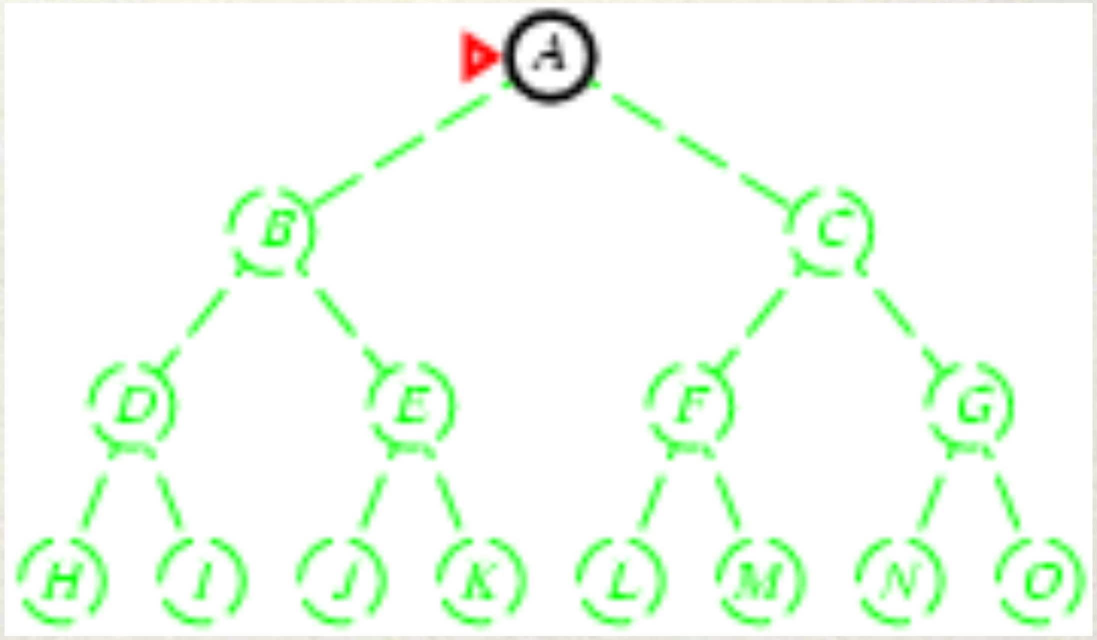
Độ sâu d	Thời gian	Không gian
4	11 giây	1 megabyte
6	18 giây	111 megabyte
8	31 giờ	11 gigabyte
10	128 ngày	1terabyte
12	35 năm	111 terabyte
14	3500 năm	11 111 terabyte

Depth-first search

- Expands one of the nodes at the deepest level of the tree
- Implementation:
  - Insert newly generated states at the front of the queue
  - Can alternatively be implemented by recursive function calls.

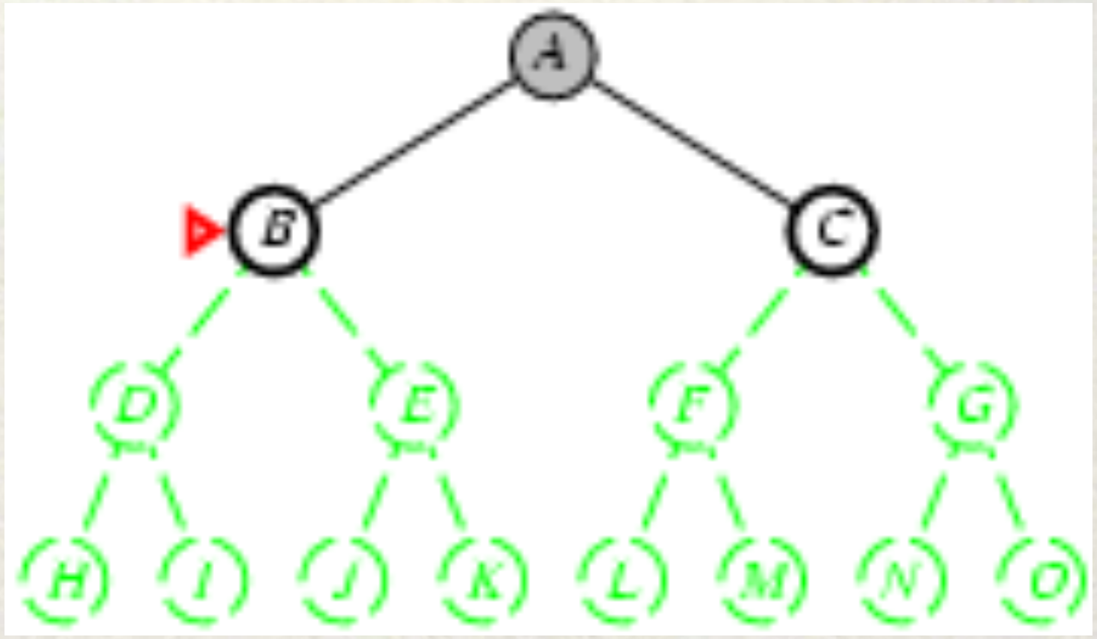
Depth-first search

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



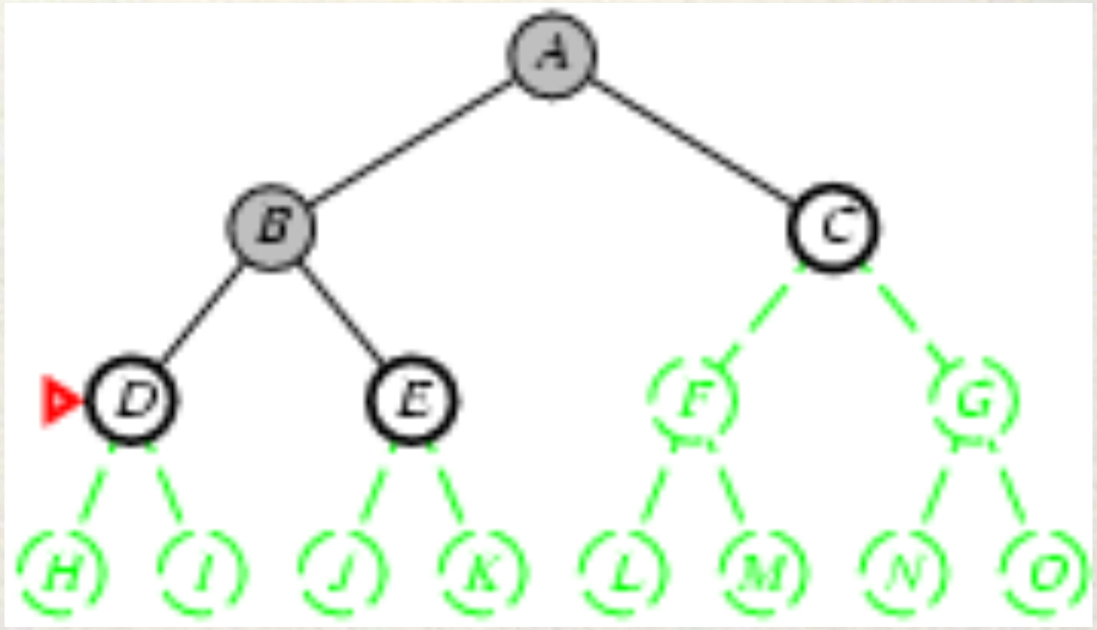
Depth-first search

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



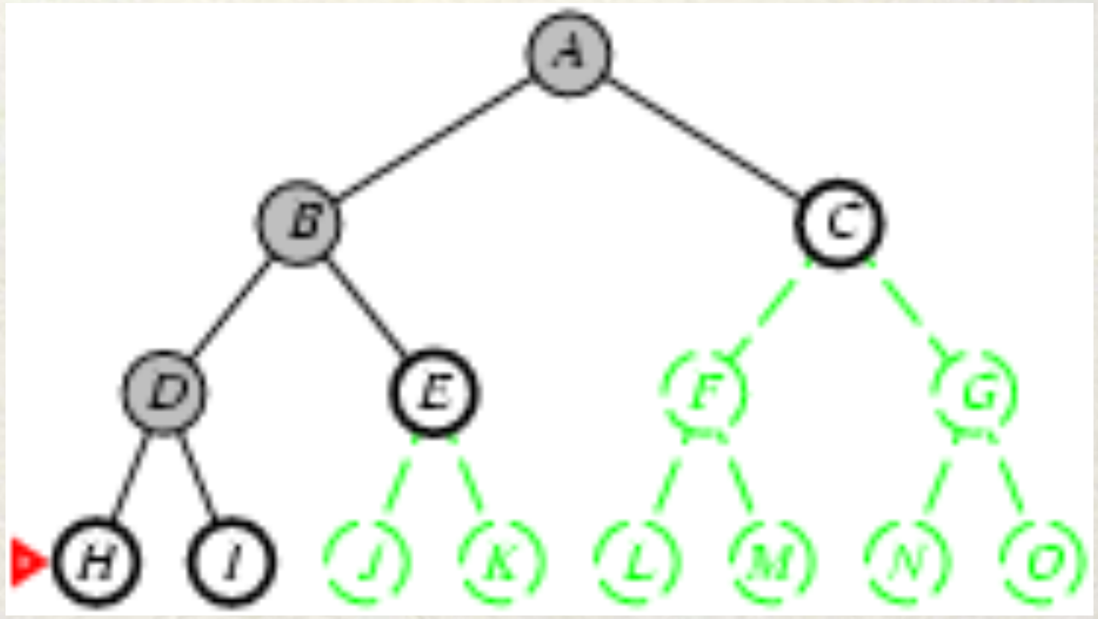
Depth-first search

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



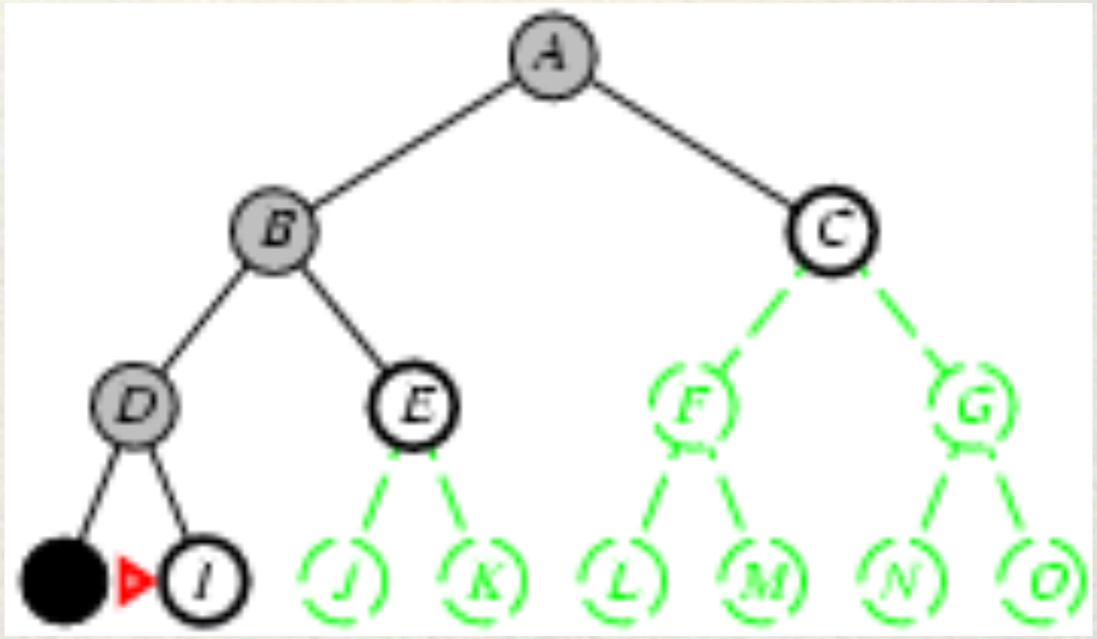
Depth-first search

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



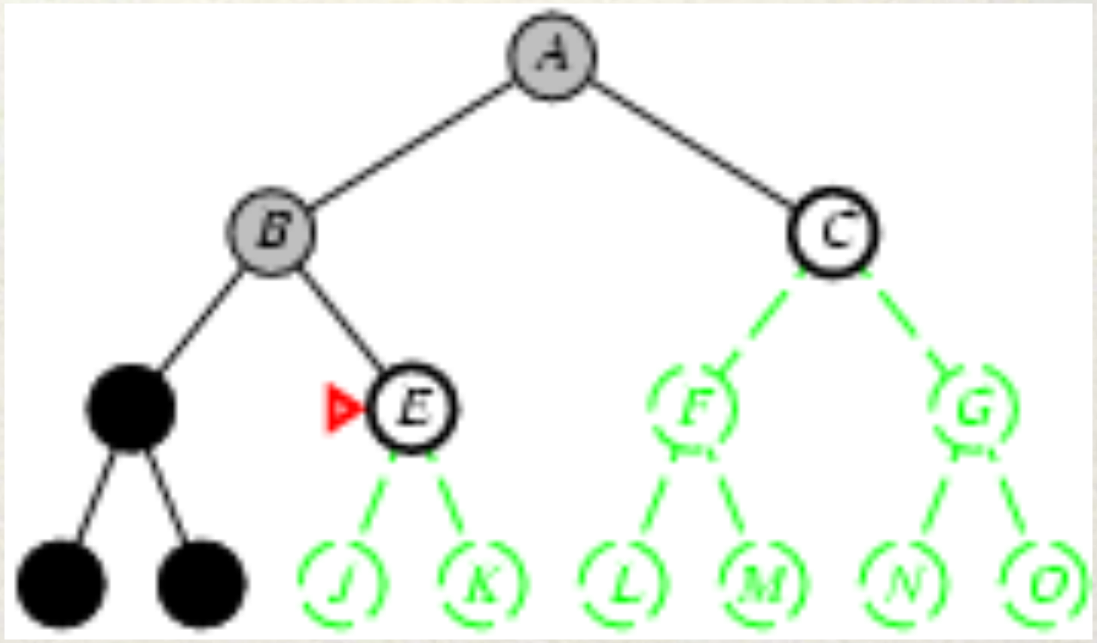
Depth-first search

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



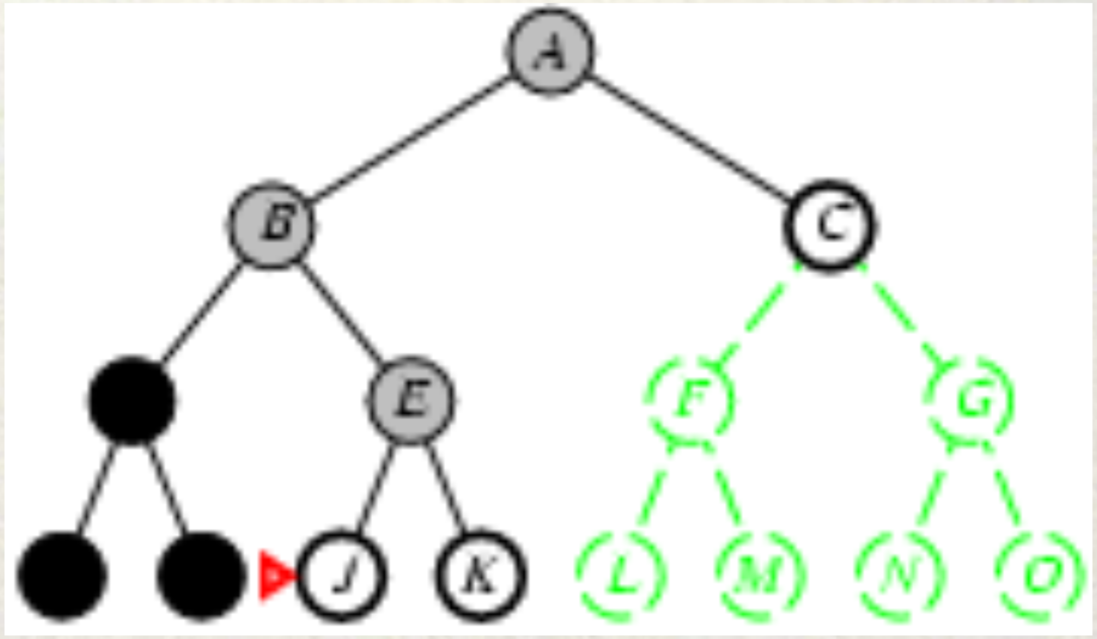
Depth-first search

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



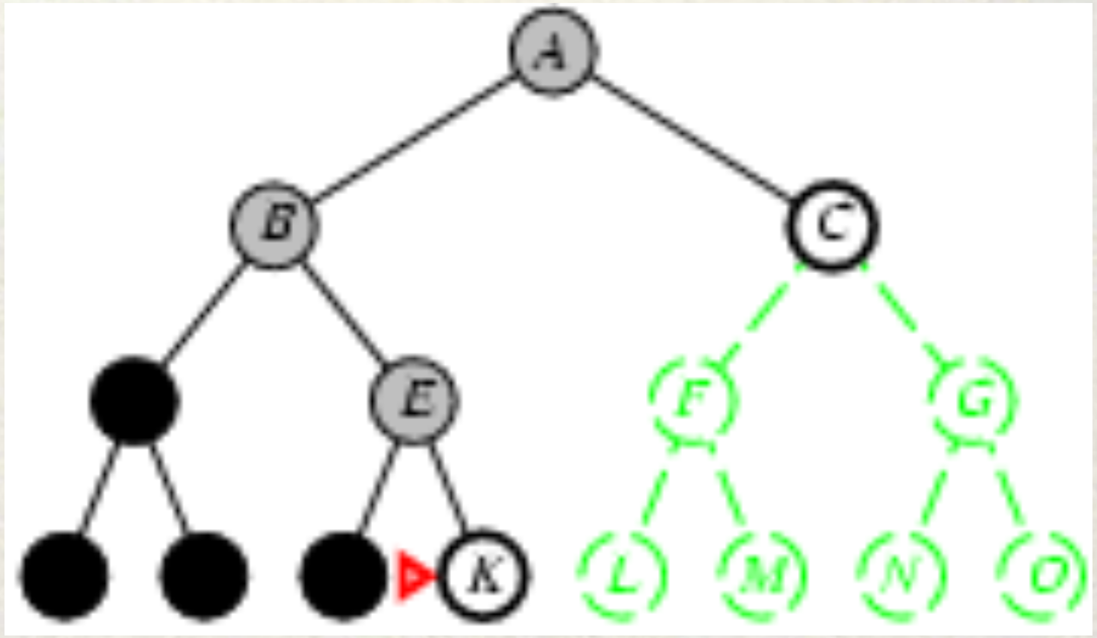
Depth-first search

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



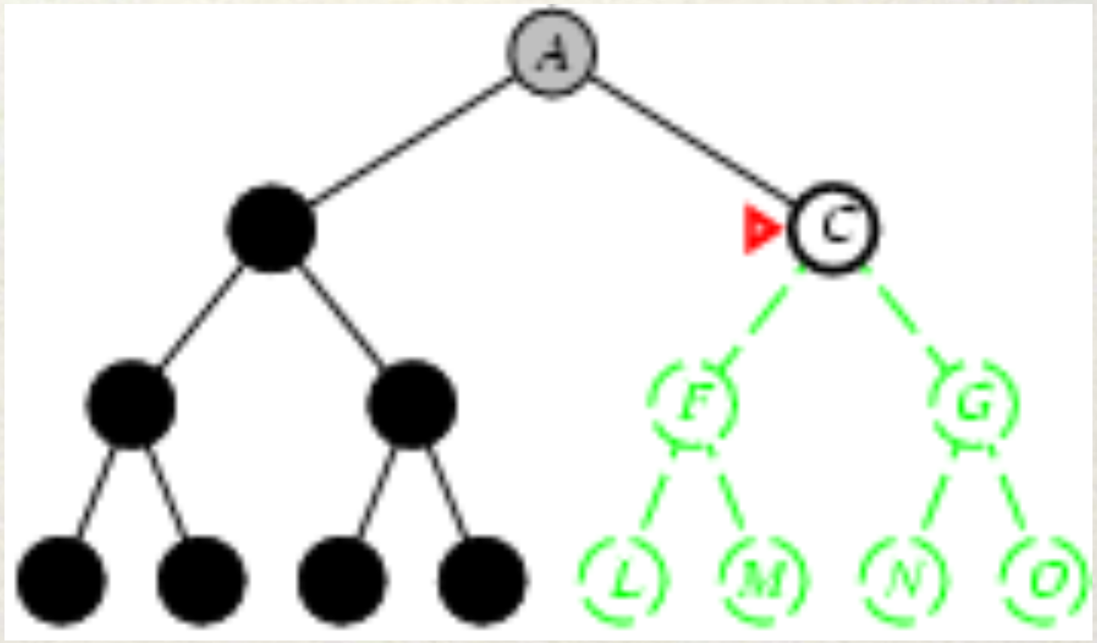
Depth-first search

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



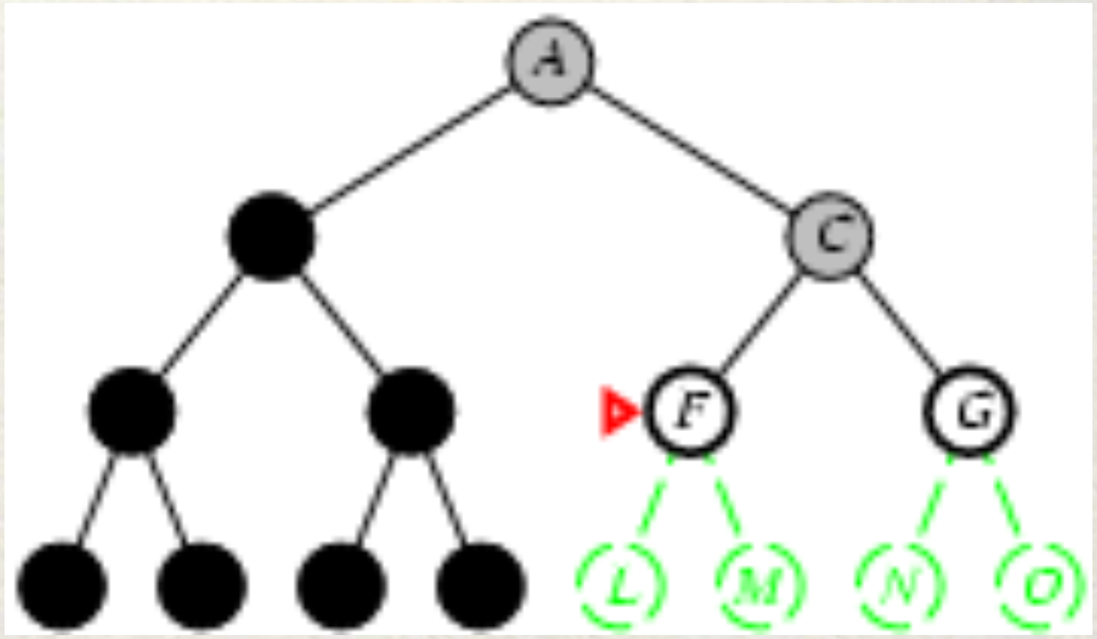
Depth-first search

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



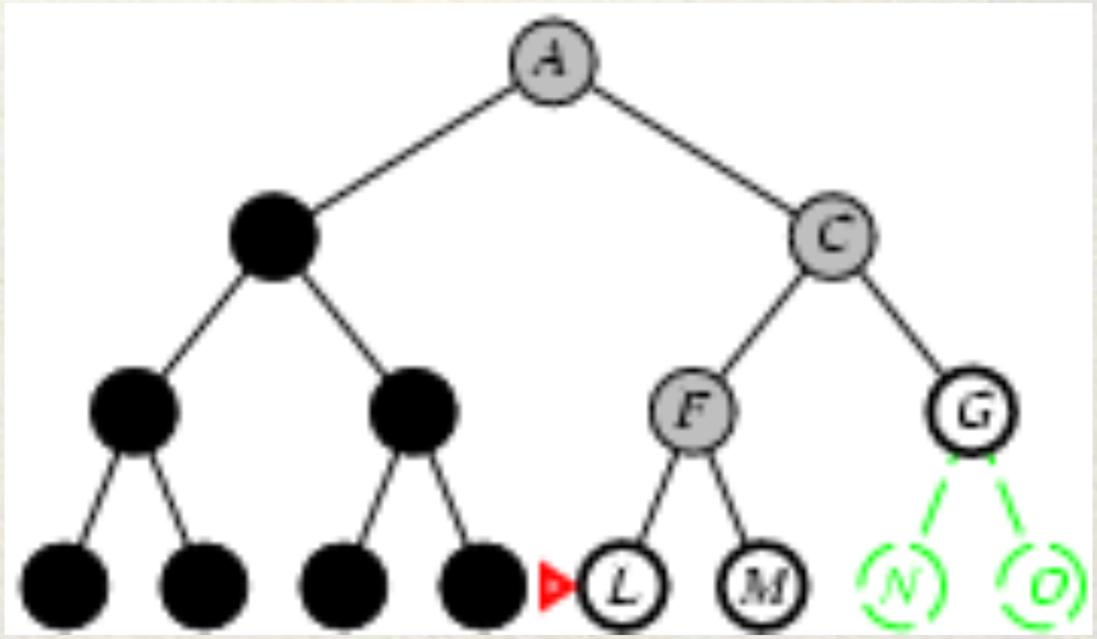
Depth-first search

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



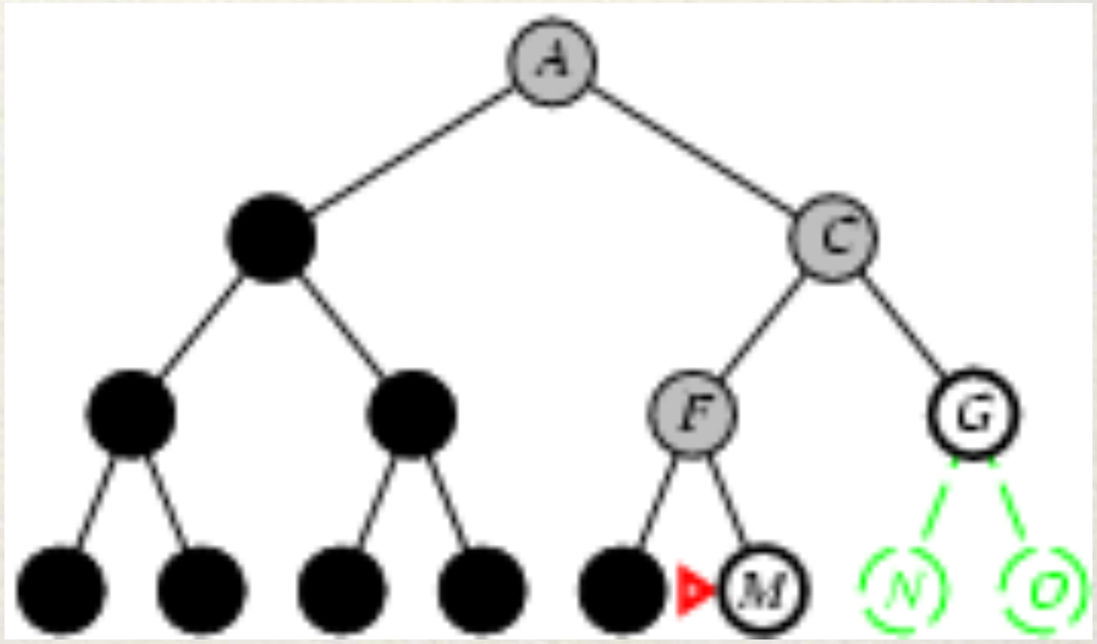
Depth-first search

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



Depth-first search

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



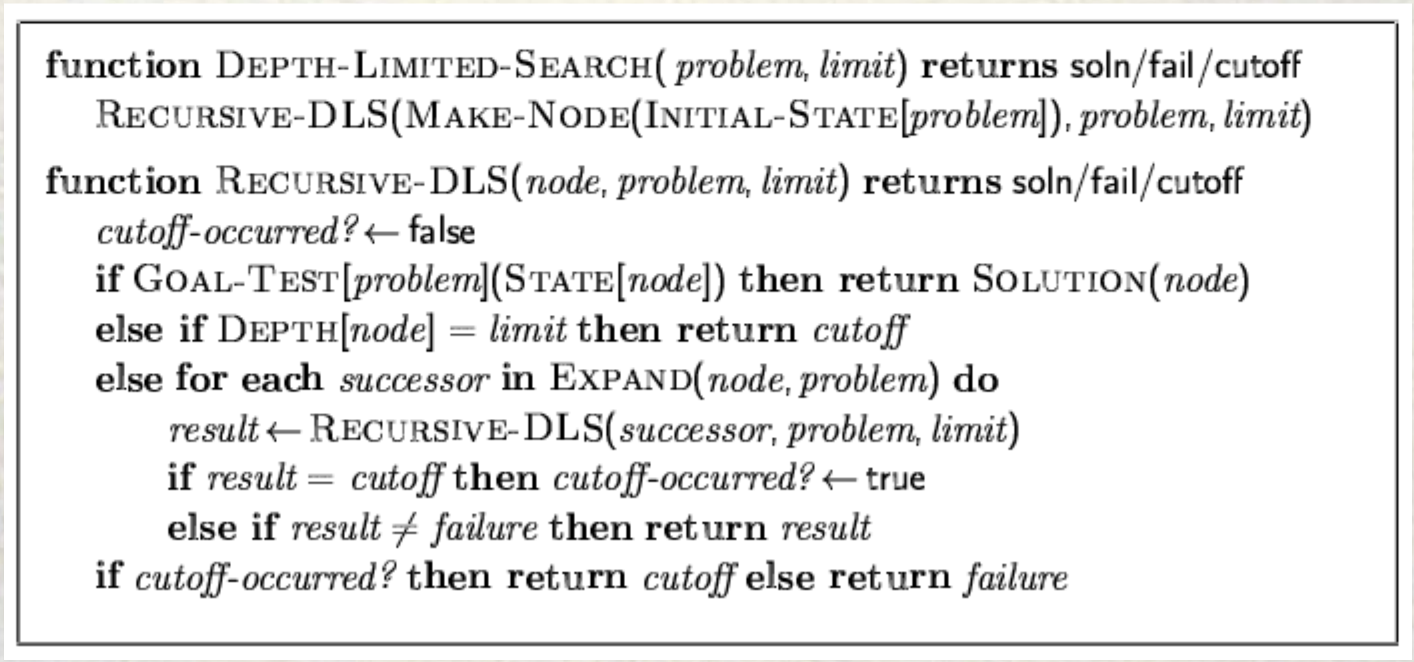
Properties of DFS

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Depth-limited search

= depth-first search with depth limit *l*,
i.e., nodes at depth *l* have no successors

- Recursive implementation:



```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/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
```

Properties Depth-limited search

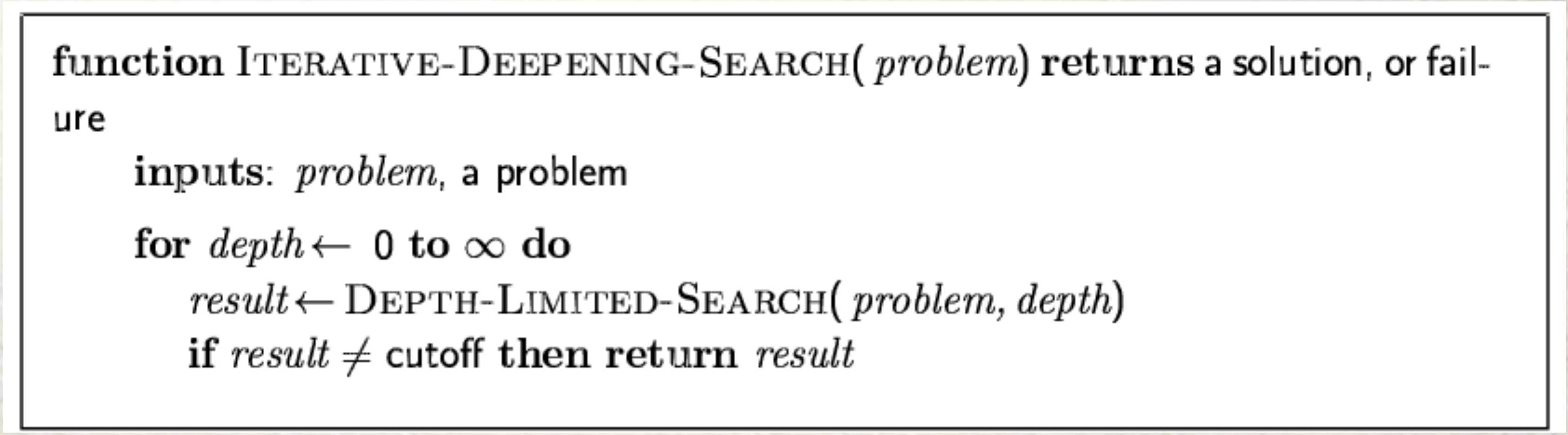
- Complete? (Yes if solution is within the depth limit. No infinite loop)
- Time? *$O(b^l)$:* l is the depth limit
- Space? *$O(bl)$*, i.e., linear space similar to depth first search.
- Optimal? No, can find suboptimal solution first

- Problem: How to pick a good limit?

Iterative deepening search

- Tries to combines the benefits of depth first (low memory) and bread-first (optimal and complete) by doing a series of depth limited searches to depth 1, 2, 3, etc.
- Early states will be expanded multiple times, but that might not matter too much as most of the nodes are near the leaves

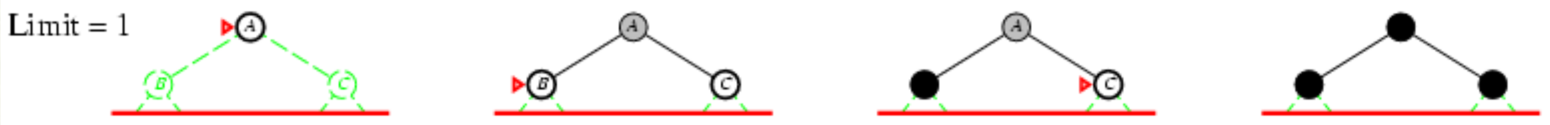
Iterative deepening search



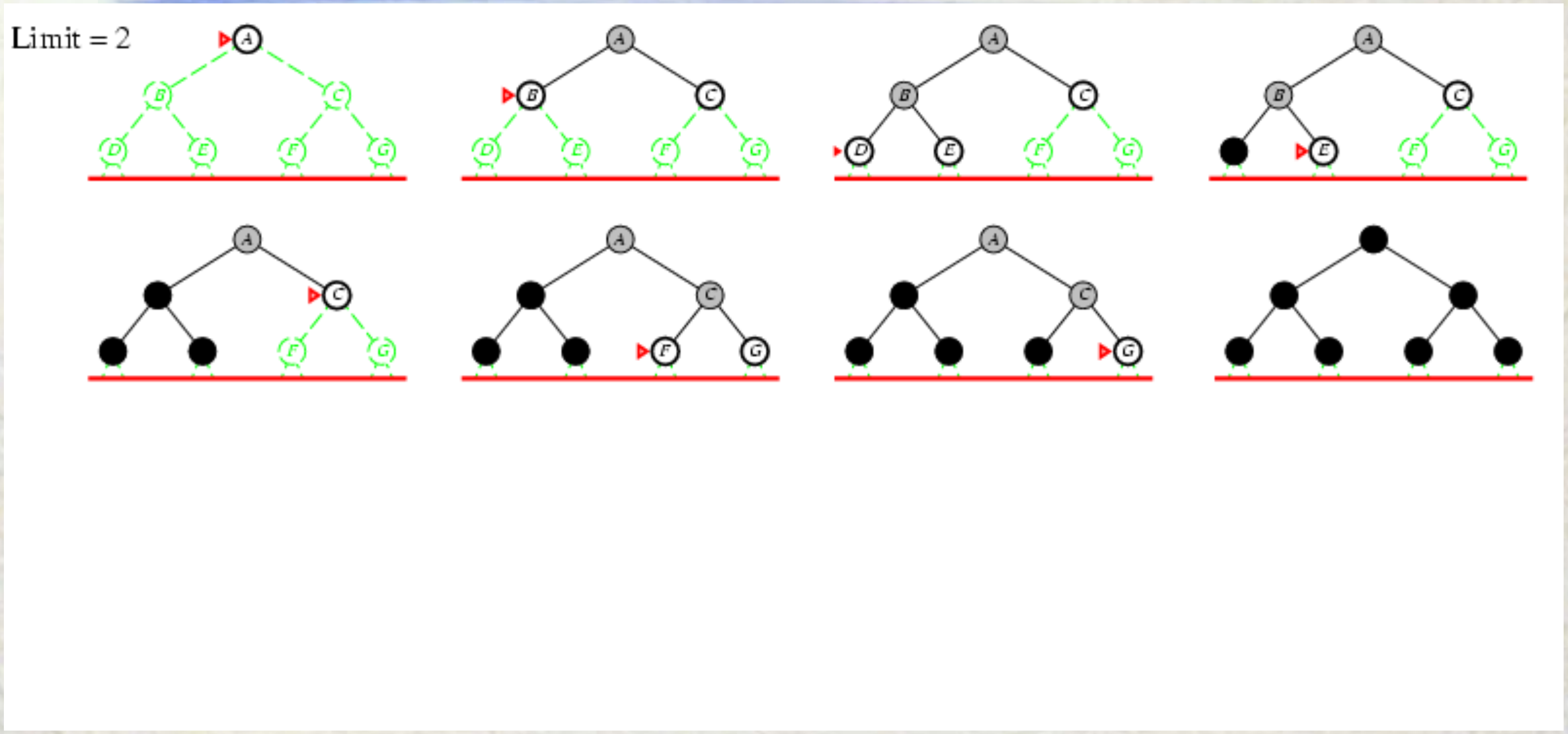
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem

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

IDS *l* =0

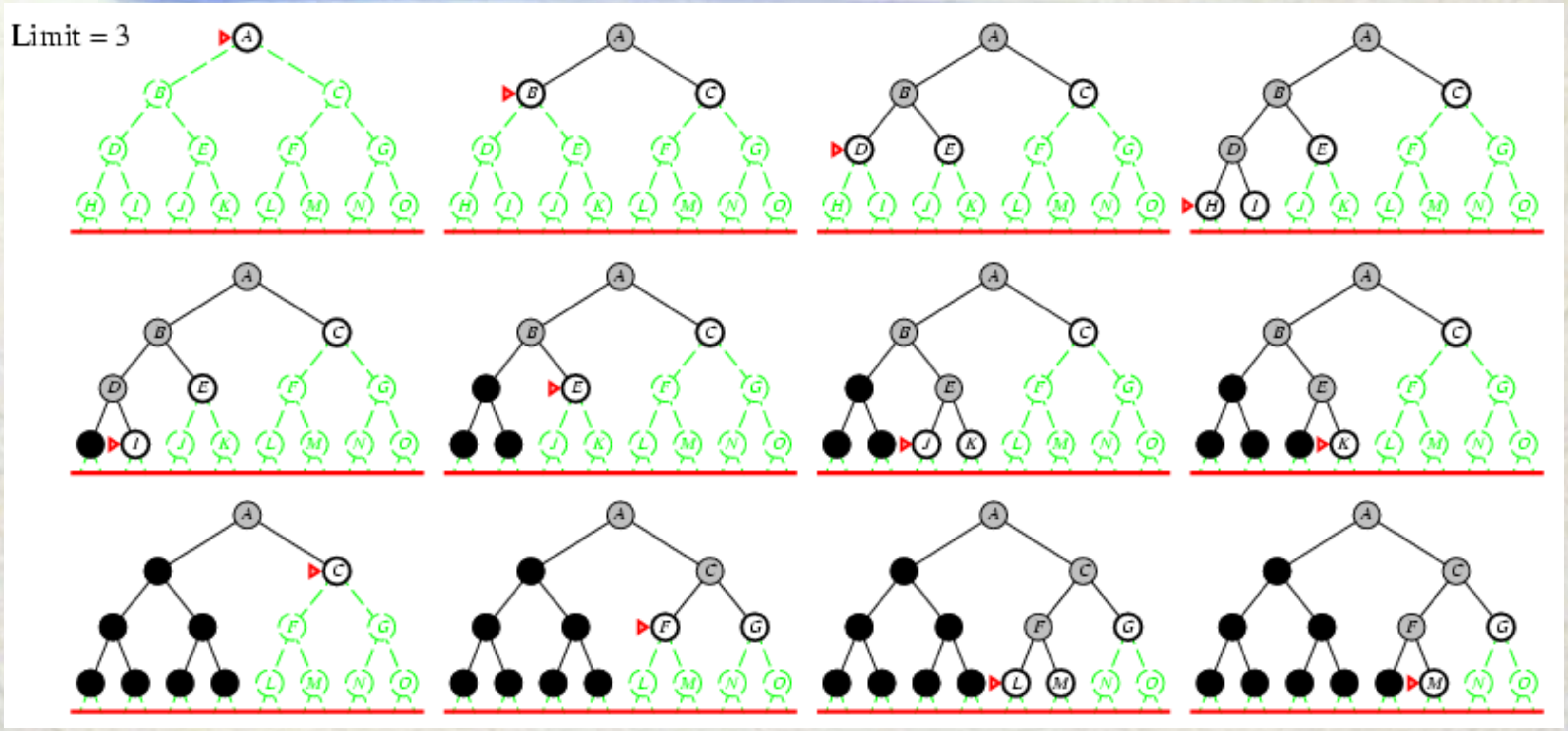
IDS *l* =1



IDS *l* =2



IDS *l* =3



IDS

- 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*:

  $N_{IDS} = (d+1)b^0 + d\, b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$
- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

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

IDS

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

References

- Artificial Intelligence: A Modern Approach. Chapter 3.
- Artificial Intelligence Illuminated. Chapter 4.