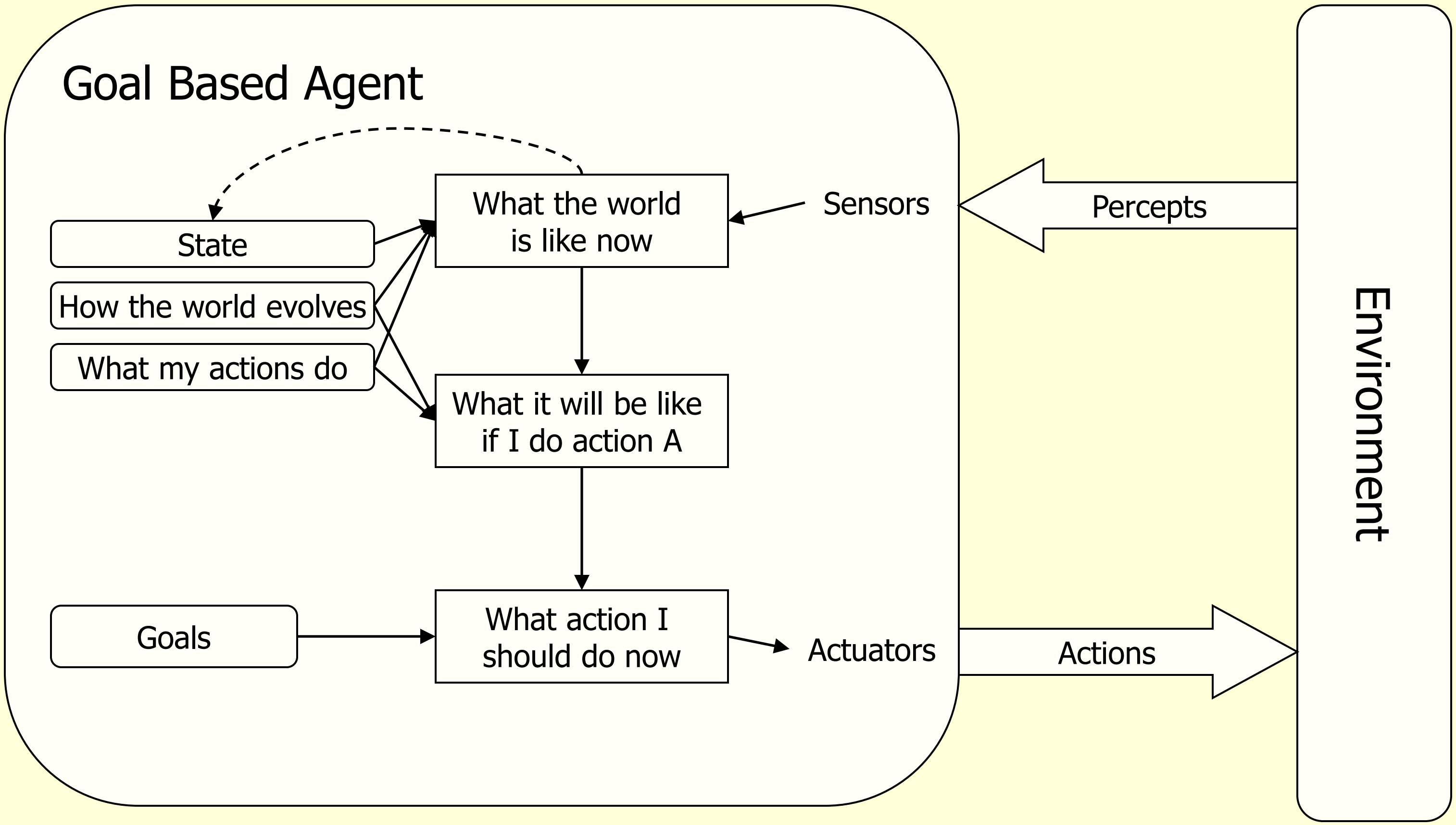
Artificial Intelligence Chapter 3: Solving Problems by Searching

Problem Solving Agents

- Problem solving agent
  - A kind of “goal based” agent
  - Finds <u>sequences of actions</u> that lead to desirable states.

- The algorithms are uninformed
  - No extra information about the problem other than the definition
    - No extra information
    - No heuristics (rules)

Goal Based Agent



Goal Based Agent

```
Function Simple-Problem-Solving-Agent( percept ) returns action

Inputs: percept        a percept
Static: seq            an action sequence initially empty
        state          some description of the current world
        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 )                          # SEARCH
action <- RECOMMENDATION ( seq )                     # SOLUTION
seq <- REMAINDER( seq )
return action                                        # EXECUTION
```

Goal Based Agents

- Assumes the problem environment is:
  - Static
    - The plan remains the same
  - Observable
    - Agent knows the initial state
  - Discrete
    - Agent can enumerate the choices
  - Deterministic
    - Agent can plan a sequence of actions such that each will lead to an intermediate state
- The agent carries out its plans with its eyes closed
  - Certain of what's going on
  - Open loop system

Well Defined Problems and Solutions

- A problem
  - Initial state
  - Actions and Successor Function
  - Goal test
  - Path cost

Example: Water Pouring

- Given a 4 gallon bucket and a 3 gallon bucket, how can we measure exactly 2 gallons into one bucket?
  - There are no markings on the bucket
  - You must fill each bucket completely

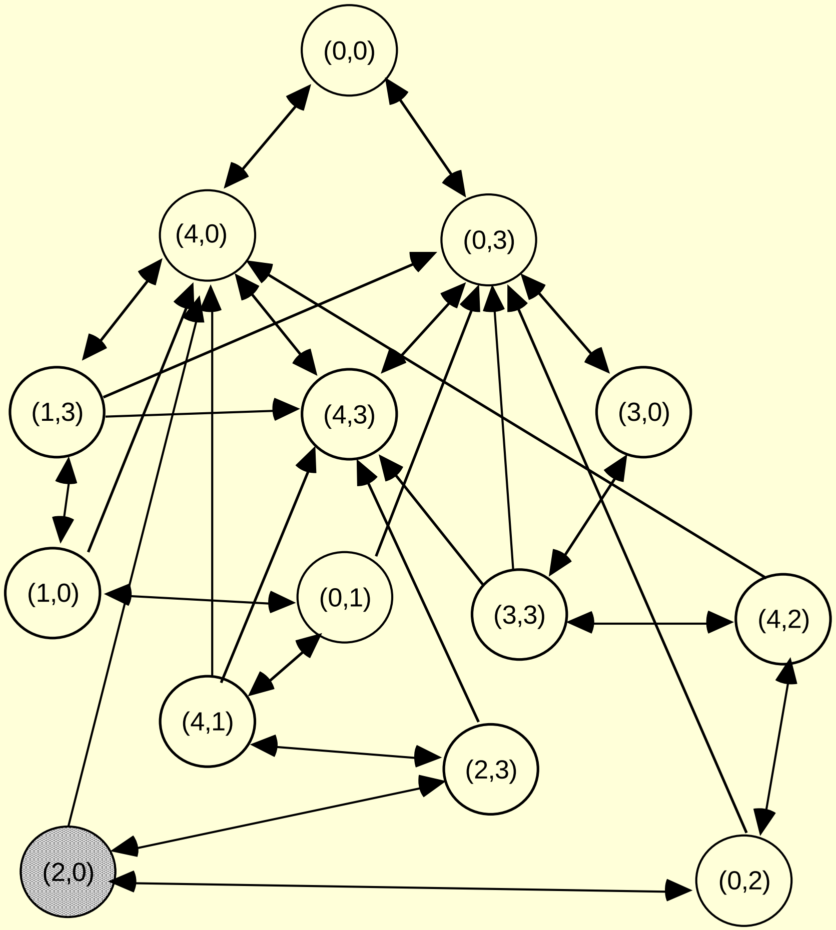
Example: Water Pouring

- Initial state:
  - The buckets are empty
  - Represented by the tuple (0 0)
- Goal state:
  - One of the buckets has two gallons of water in it
  - Represented by either (x 2) or (2 x)
- Path cost:
  - 1 per unit step

Example: Water Pouring

- Actions and Successor Function
  - Fill a bucket
    - (x y) -> (3 y)
    - (x y) -> (x 4)
  - Empty a bucket
    - (x y) -> (0 y)
    - (x y) -> (x 0)
  - Pour contents of one bucket into another
    - (x y) -> (0 x+y) or (x+y-4, 4)
    - (x y) -> (x+y 0) or (3, x+y-3)

Example: Water Pouring



Example: Eight Puzzle

- States:
  - Description of the eight tiles and location of the blank tile
- Successor Function:
  - Generates the legal states from trying the four actions {*Left, Right, Up, Down*}
- Goal Test:
  - Checks whether the state matches the goal configuration
- Path Cost:
  - Each step costs 1

7	2	4
5		6
8	3	1

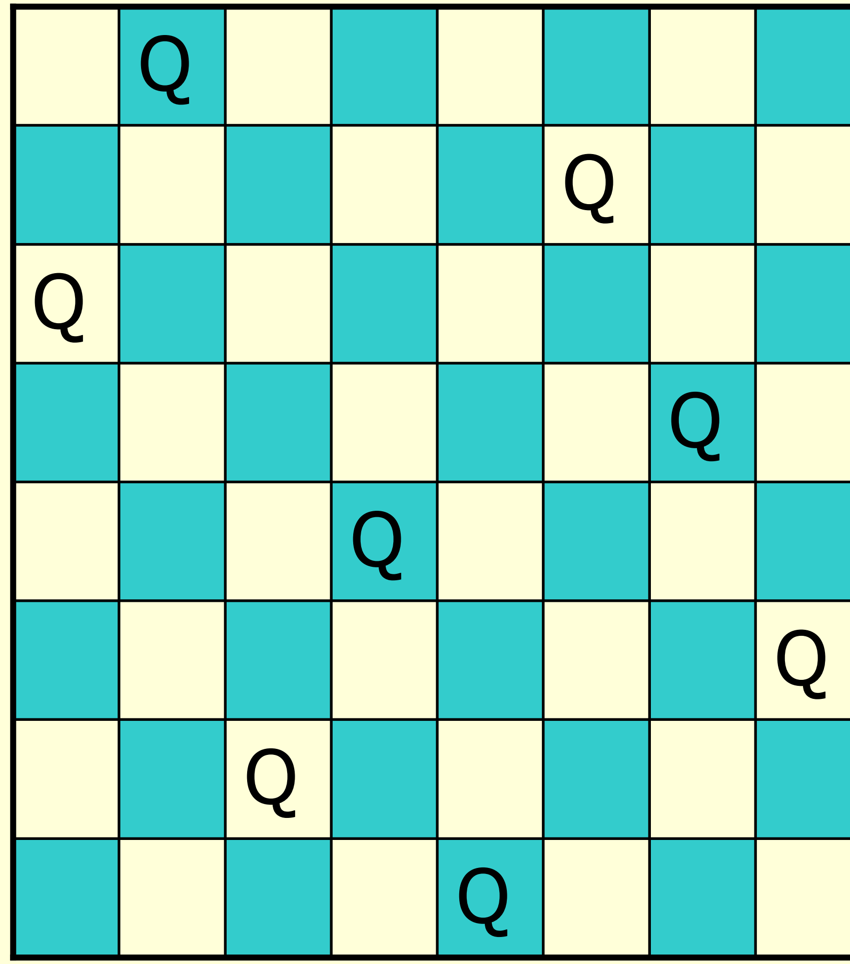
1	2	3
4	5	6
7	8	

Example: Eight Puzzle

- Eight puzzle is from a family of “sliding – block puzzles”
  - NP Complete
  - 8 puzzle has 9!/2 = 181440 states
  - 15 puzzle has approx. $1.3*10^{12}$ states
  - 24 puzzle has approx. $1*10^{25}$ states

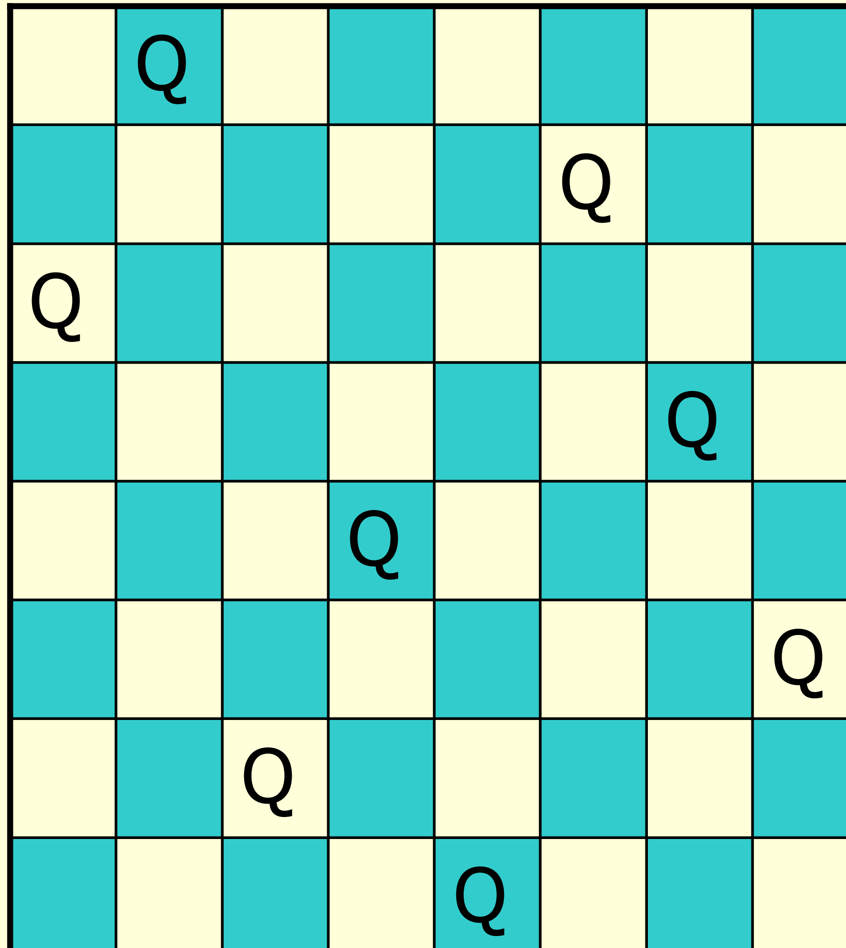
Example: Eight Queens

- Place eight queens on a chess board such that no queen can attack another queen
- No path cost because only the final state counts!
- Incremental formulations
- Complete state formulations



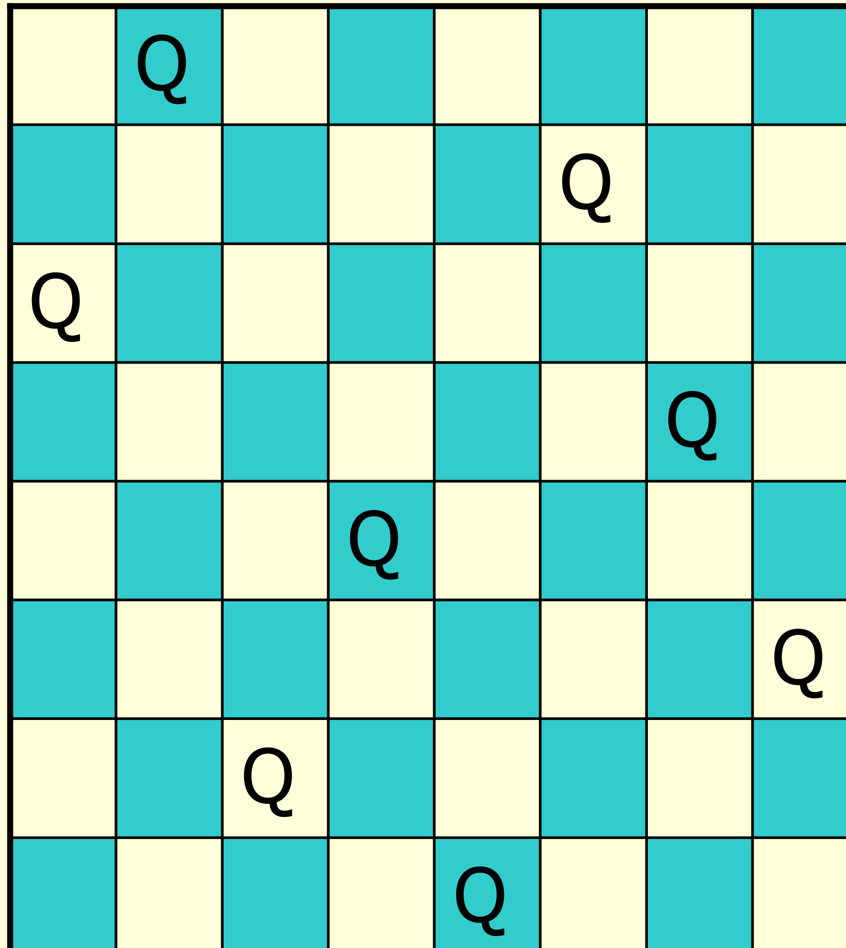
Example: Eight Queens

- States:
  - Any arrangement of 0 to 8 queens on the board
- Initial state:
  - No queens on the board
- Successor function:
  - Add a queen to an empty square
- Goal Test:
  - 8 queens on the board and none are attacked
- $64*63*\ldots*57 = 1.8*10^{14}$ possible sequences
  - Ouch!



Example: Eight Queens

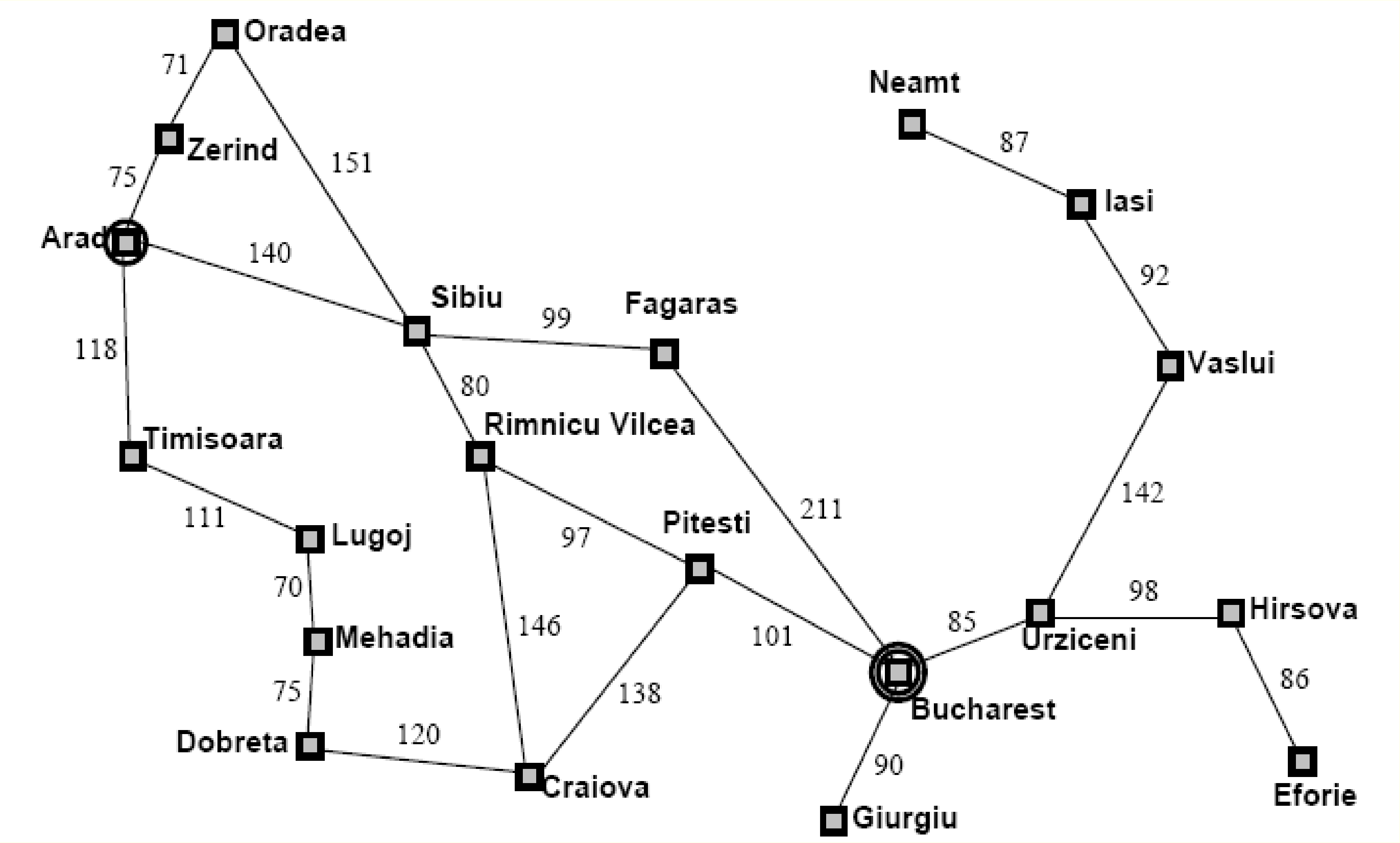
- States:
  - Arrangements of n queens, one per column in the leftmost n columns, with no queen attacking another are states
- Successor function:
  - Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- 2057 sequences to investigate



Other Toy Examples

- Another Example: Jug Fill
- Another Example: Black White Marbles
- Another Example: Row Boat Problem
- Another Example: Sliding Blocks
- Another Example: Triangle Tee

Example: Map Planning



Searching For Solutions

- Initial State
  - e.g. “At Arad”
- Successor Function
  - A set of action state pairs
  - S(Arad) = {(Arad->Zerind, Zerind), …}
- Goal Test
  - e.g. x = “at Bucharest”
- Path Cost
  - sum of the distances traveled

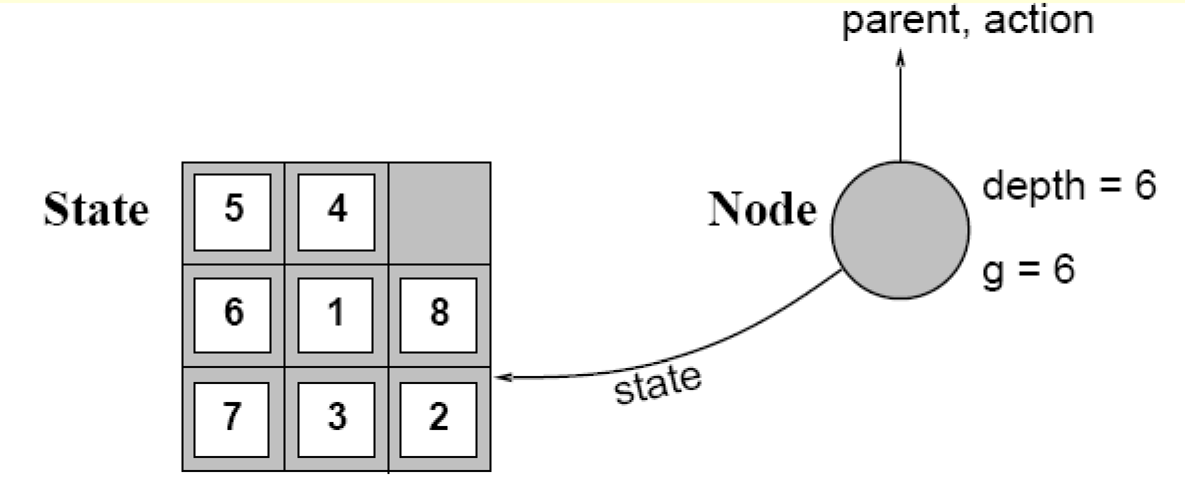
Searching For Solutions

- Having formulated some problems…how do we solve them?
- Search through a state space
- Use a search tree that is generated with an initial state and successor functions that define the state space

Searching For Solutions

- A ***state*** is (a representation of) a physical configuration
- A ***node*** is a data structure constituting part of a search tree
  - Includes parent, children, depth, path cost
- States do not have children, depth, or path cost
- The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSOR function of the problem to create the corresponding states

Searching For Solutions



Searching For Solutions

```
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
    end
```

Searching For Solutions

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*

Uninformed Search Strategies

- ***Uninformed*** strategies use only the information available in the problem definition
  - Also known as blind searching

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

Comparing Uninformed Search Strategies

- Completeness
  - Will a solution always be found if one exists?
- Time
  - How long does it take to find the solution?
  - Often represented as the number of nodes searched
- Space
  - How much memory is needed to perform the search?
  - Often represented as the maximum number of nodes stored at once
- Optimal
  - Will the optimal (least cost) solution be found?

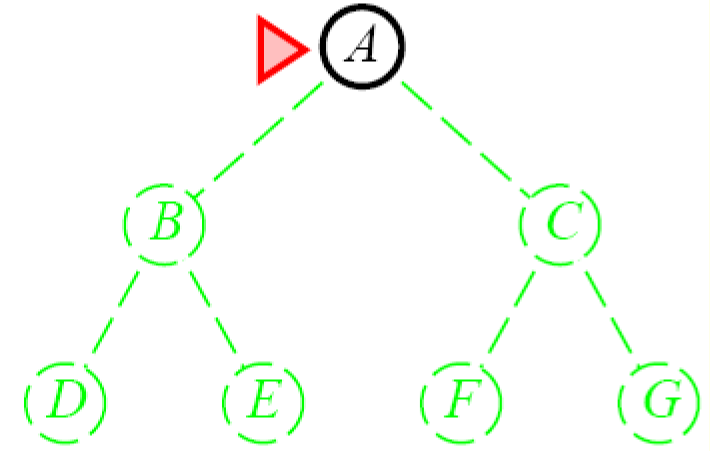
Comparing Uninformed Search Strategies

- Time and space complexity are measured in
  - b – maximum branching factor of the search tree
  - m – maximum depth of the state space
  - d – depth of the least cost solution

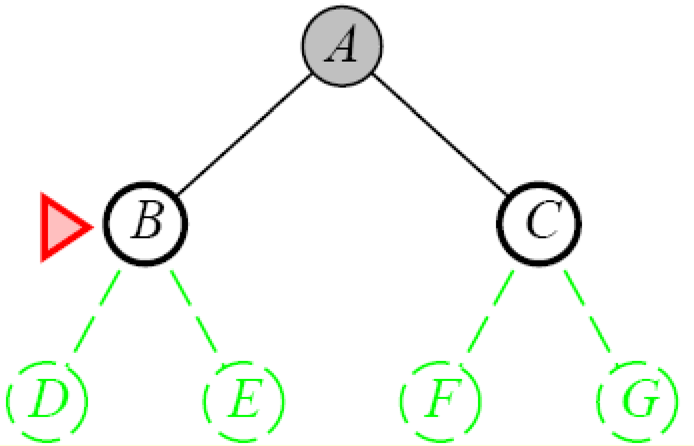
Breadth-First Search

- Recall from Data Structures the basic algorithm for a breadth-first search on a graph or tree
- Expand the ***<u>shallowest</u>*** unexpanded node
- Place all new successors at the end of a FIFO queue

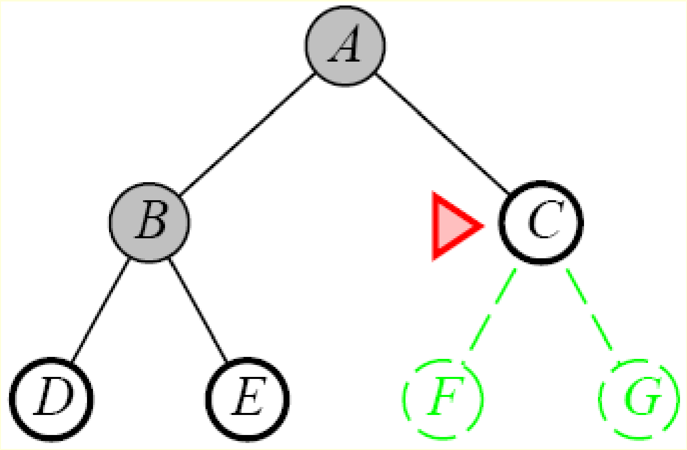
Breadth-First Search



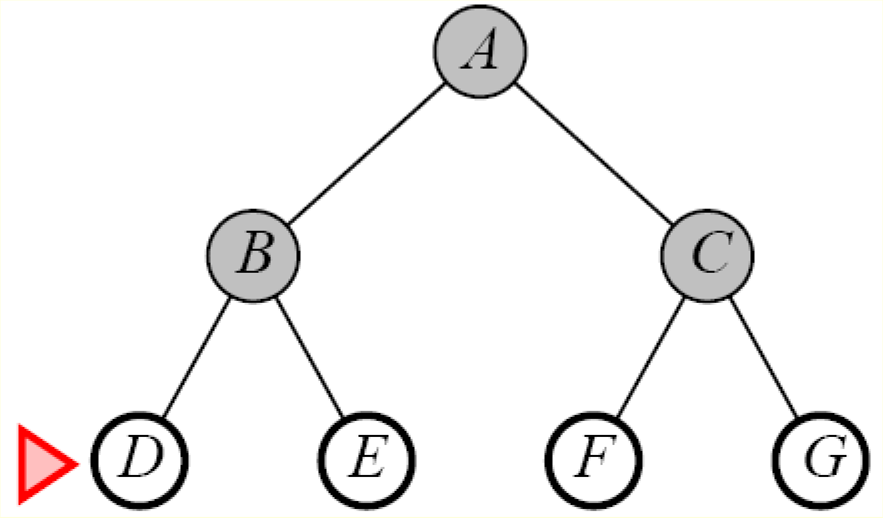
Breadth-First Search



Breadth-First Search



Breadth-First Search



Properties of Breadth-First Search

- Complete
  - Yes if b (max branching factor) is finite
- Time
  - $1 + b + b^2 + \ldots + b^d + b(b^d\text{-}1) = O(b^{d+1})$
  - exponential in d
- Space
  - $O(b^{d+1})$
  - Keeps every node in memory
  - This is the big problem; an agent that generates nodes at 10 MB/sec will produce 860 MB in 24 hours
- Optimal
  - Yes (if cost is 1 per step); not optimal in general

Lessons From Breadth First Search

- The memory requirements are a bigger problem for breadth-first search than is execution time

- Exponential-complexity search problems cannot be solved by uniformed methods for any but the smallest instances

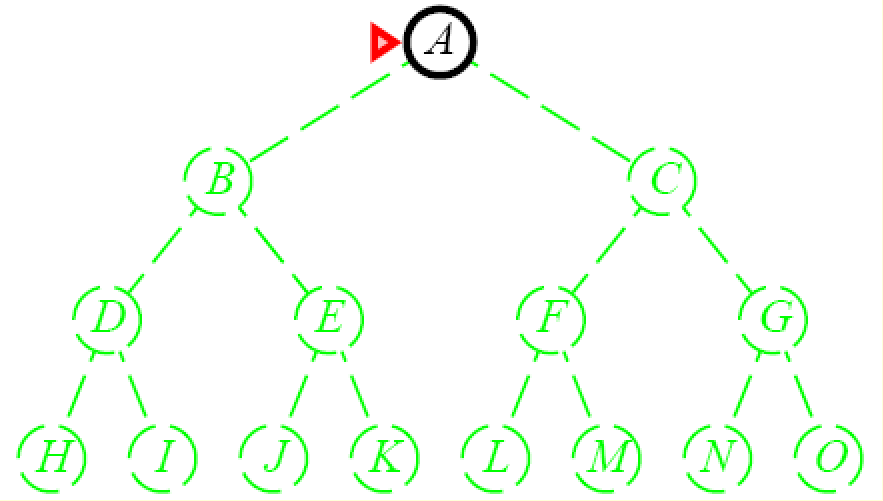
Depth-First Search

- Recall from Data Structures the basic algorithm for a depth-first search on a graph or tree

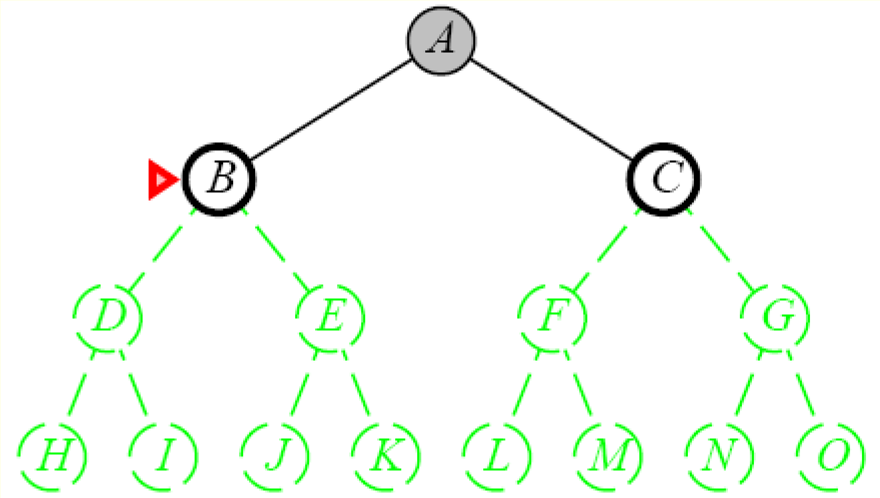
- Expand the ***deepest*** unexpanded node

- Unexplored successors are placed on a stack until fully explored

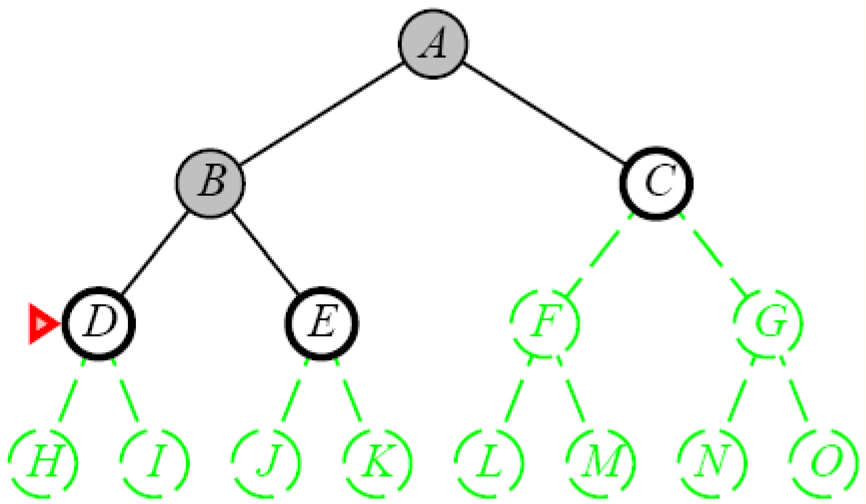
Depth-First Search



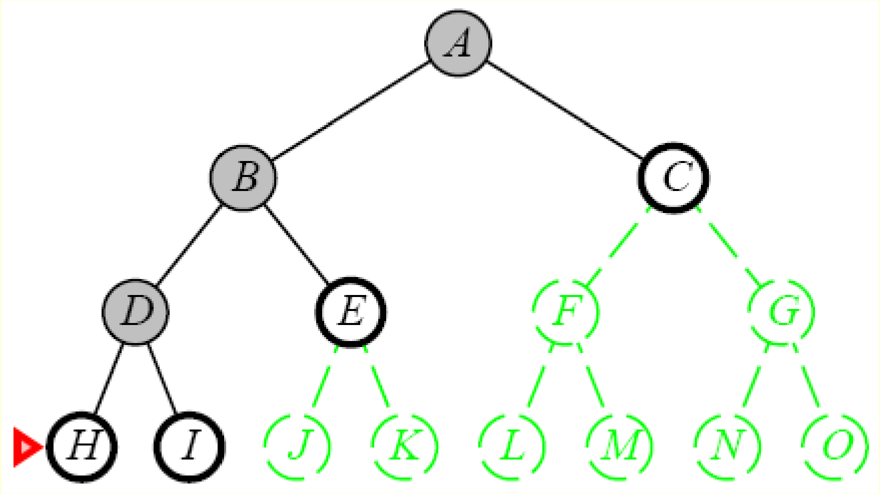
Depth-First Search



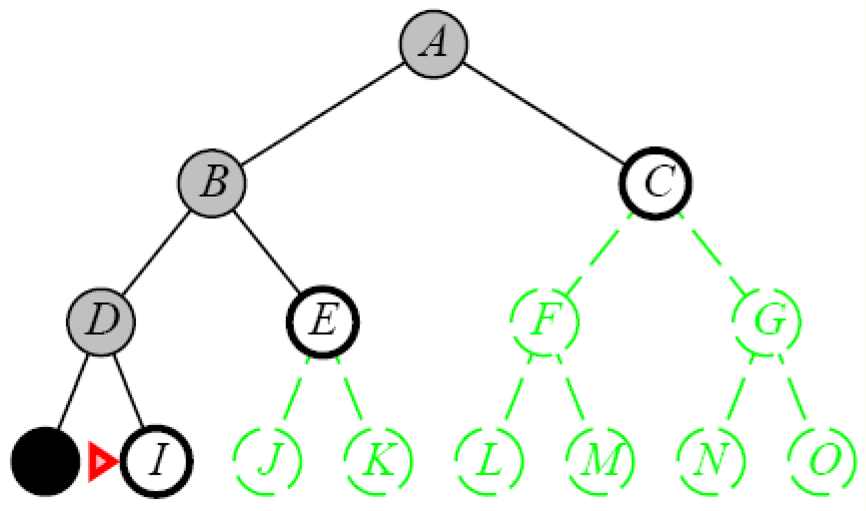
Depth-First Search



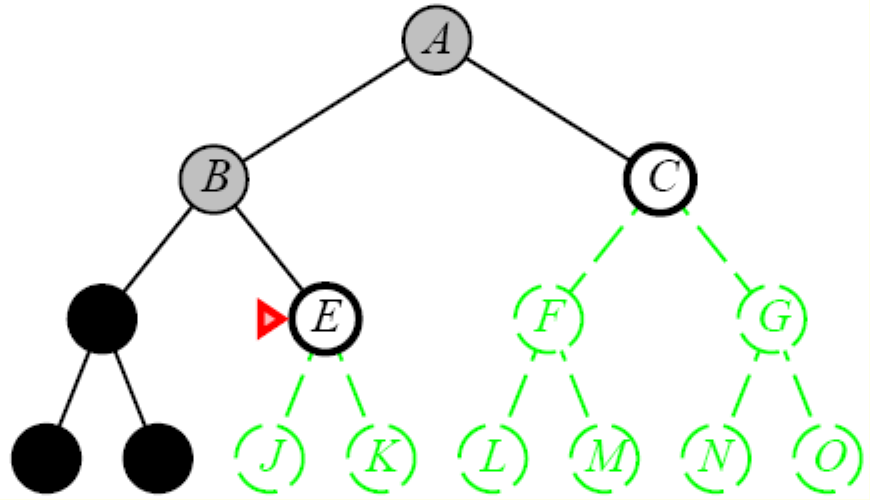
Depth-First Search



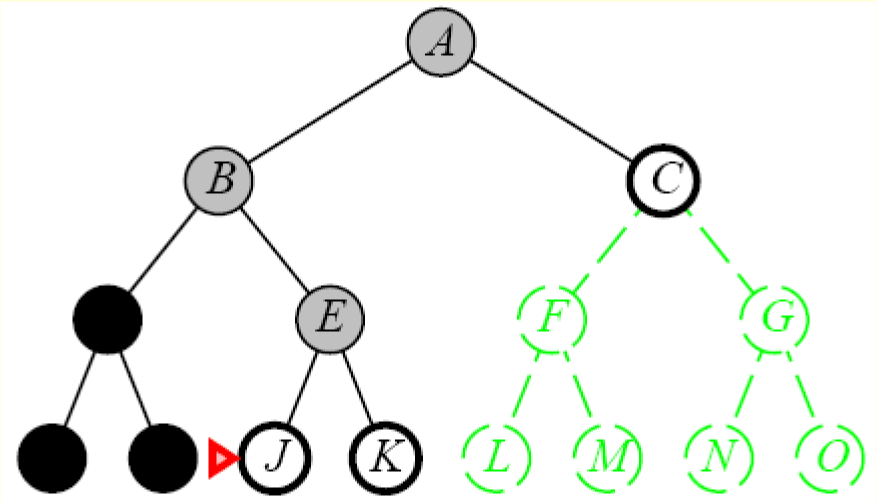
Depth-First Search



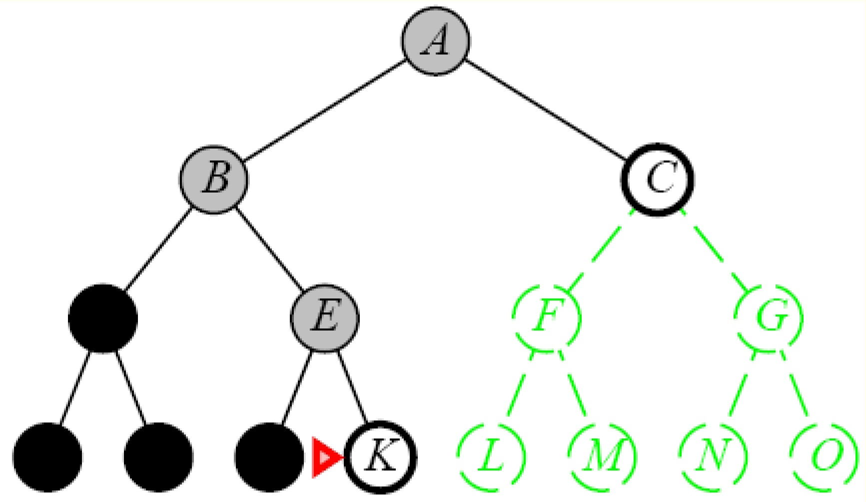
Depth-First Search



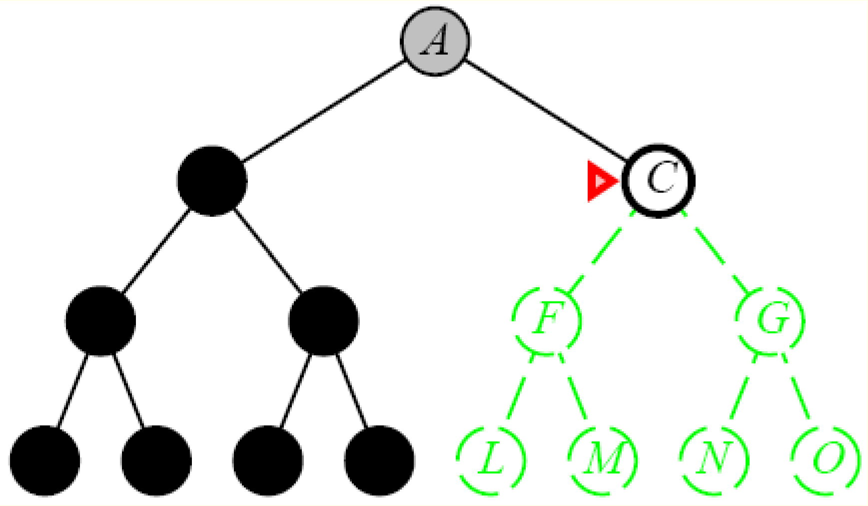
Depth-First Search



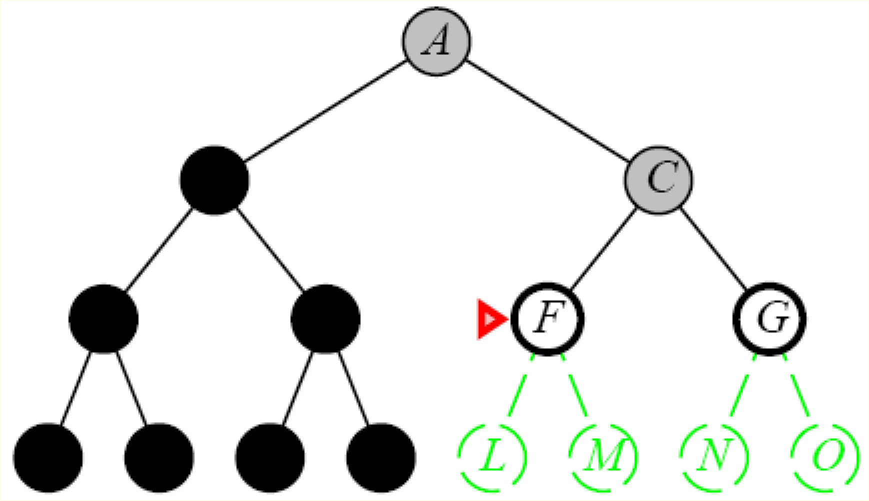
Depth-First Search



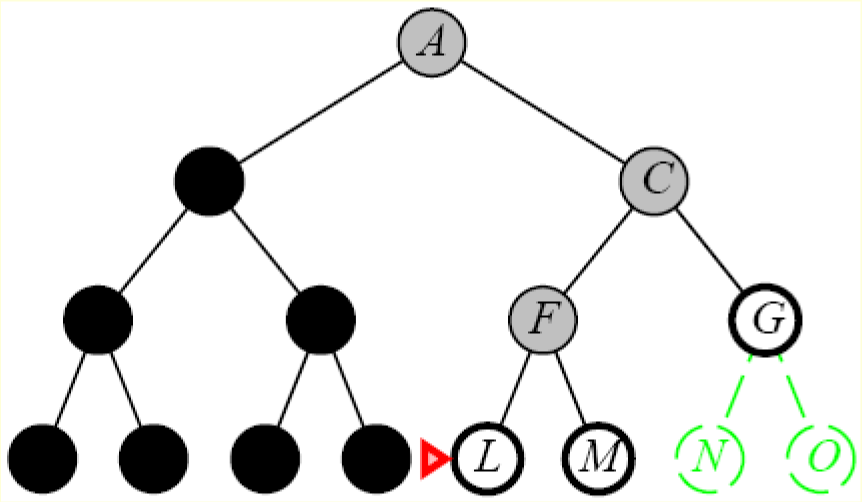
Depth-First Search



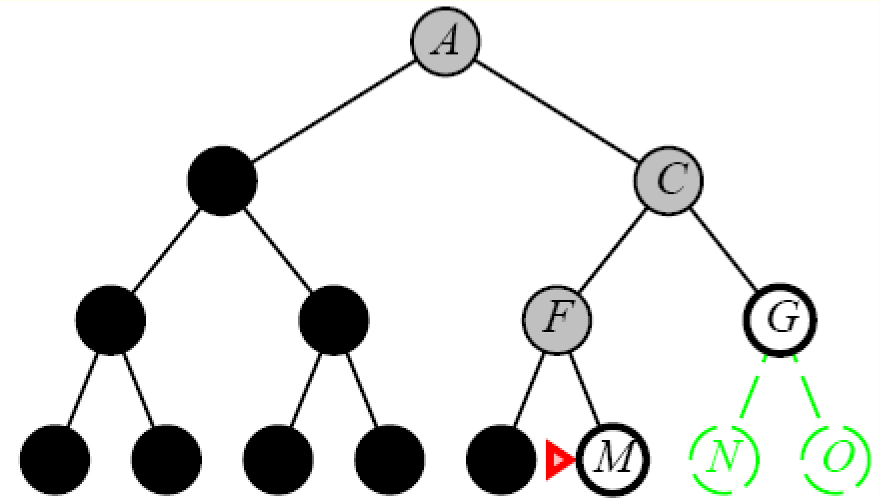
Depth-First Search



Depth-First Search



Depth-First Search



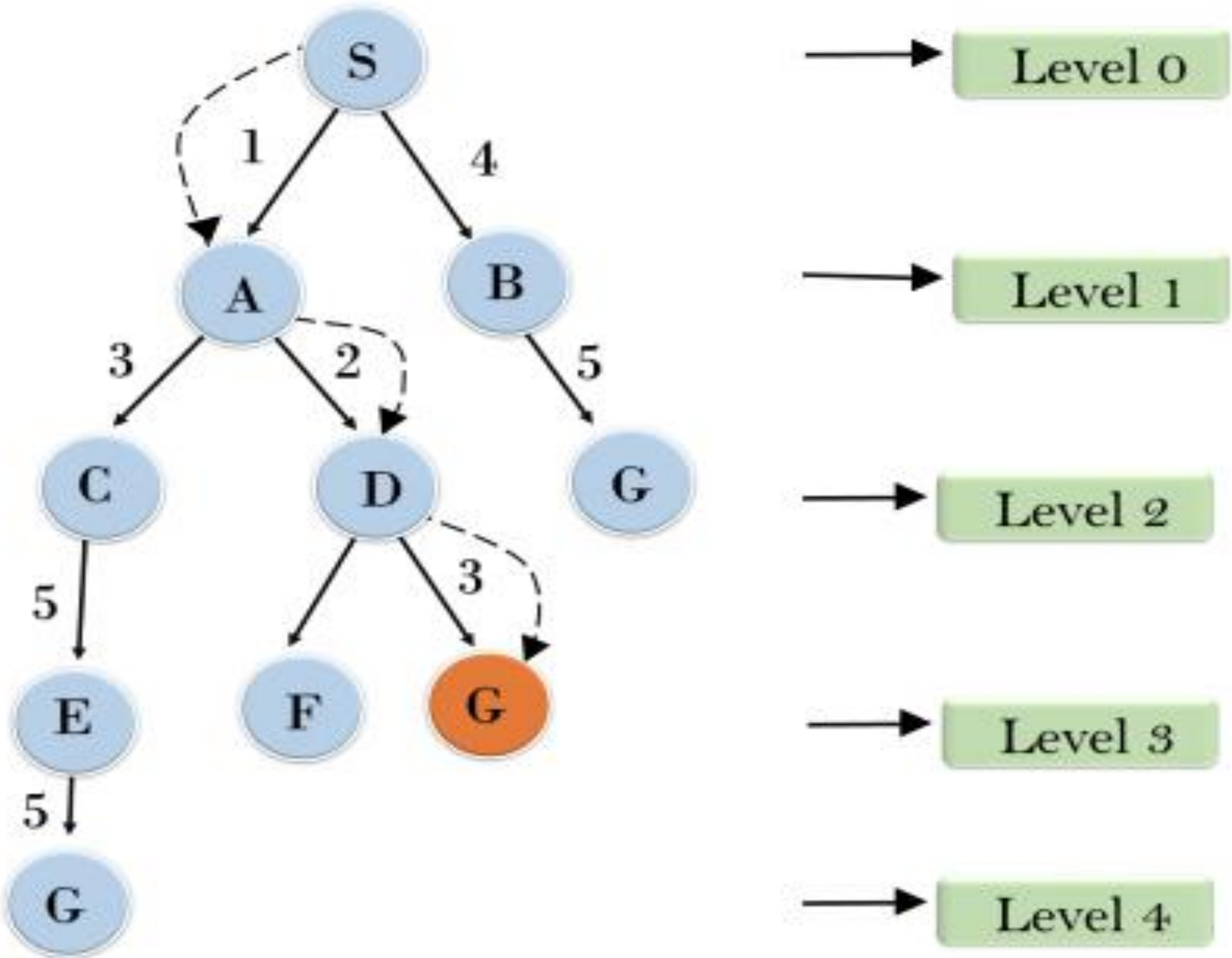
Depth-First Search

- Complete
  - No: fails in infinite-depth spaces, spaces with loops
    - Modify to avoid repeated spaces along path
  - Yes: in finite spaces
- Time
  - $O(b^m)$
  - Not great if m is much larger than d
  - But if the solutions are dense, this may be faster than breadth-first search
- Space
  - O(bm)…linear space
- Optimal
  - No

Uniform-Cost Search

- Same idea as the algorithm for breadth-first search…but…
  - Expand the ***<u>least-cost</u>*** unexpanded node
  - FIFO queue is ordered by cost
  - Equivalent to regular breadth-first search if all step costs are equal

Uniform Cost Search



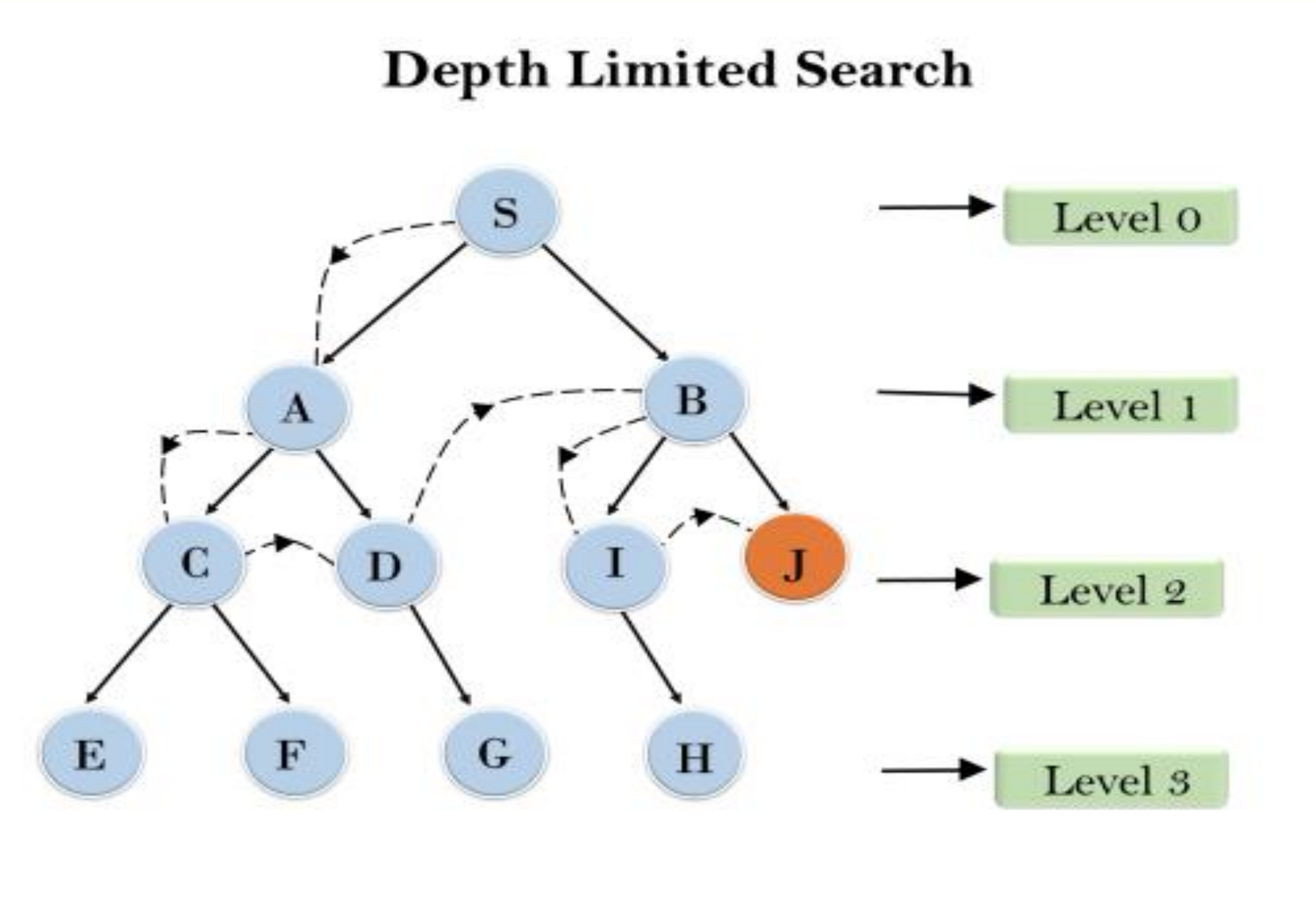
Uniform-Cost Search

- Complete
  - Yes if the cost is greater than some threshold
  - step cost >= ε
- Time
  - Complexity cannot be determined easily by d or d
  - Let C* be the cost of the optimal solution
  - $O(b^{ceil(C^*/ \varepsilon)})$
- Space
  - $O(b^{ceil(C^*/ \varepsilon)})$
- Optimal
  - Yes, Nodes are expanded in increasing order

Depth-Limited Search

- A variation of depth-first search that uses a depth limit
  - Alleviates the problem of unbounded trees
  - Search to a predetermined depth ℓ ("ell")
  - Nodes at depth ℓ have no successors

- Same as depth-first search if $\ell = \infty$
- Can terminate for failure and cutoff


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

Depth-Limited Search

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 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
```

Depth-Limited Search

- Complete
  - Yes if $\ell < d$
- Time
  - $O(b^{\ell})$
- Space
  - $O(b\ell)$
- Optimal
  - No if $\ell > d$

Iterative Deepening Search

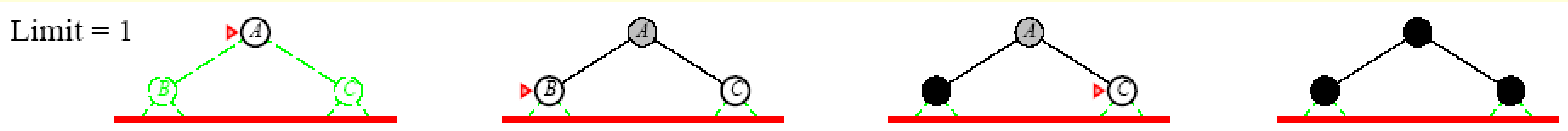
- Iterative deepening depth-first search
  - Uses depth-first search
  - Finds the best depth limit
    - Gradually increases the depth limit; 0, 1, 2, … until a goal is found

Iterative Deepening Search

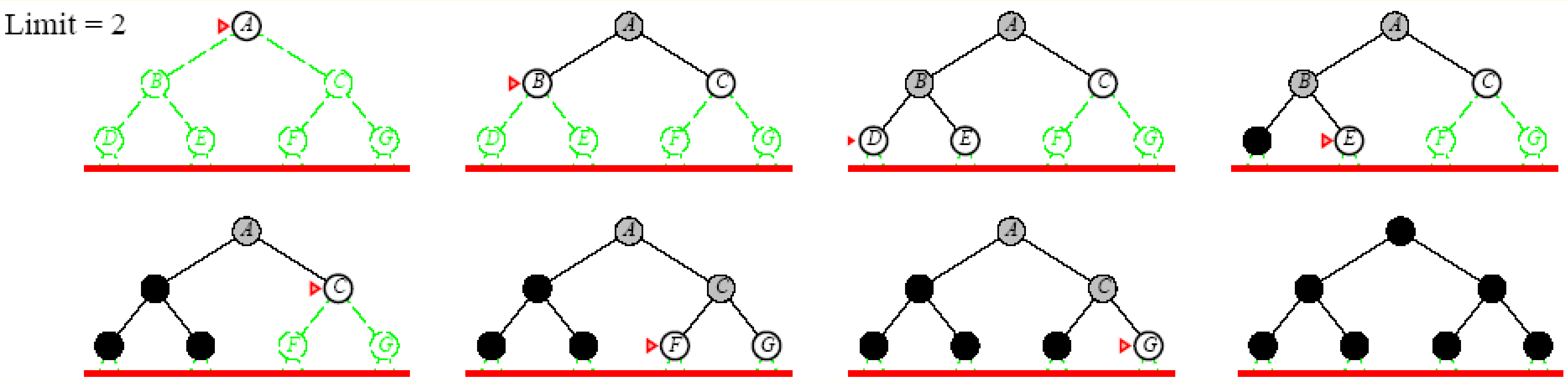
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

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

Iterative Deepening Search

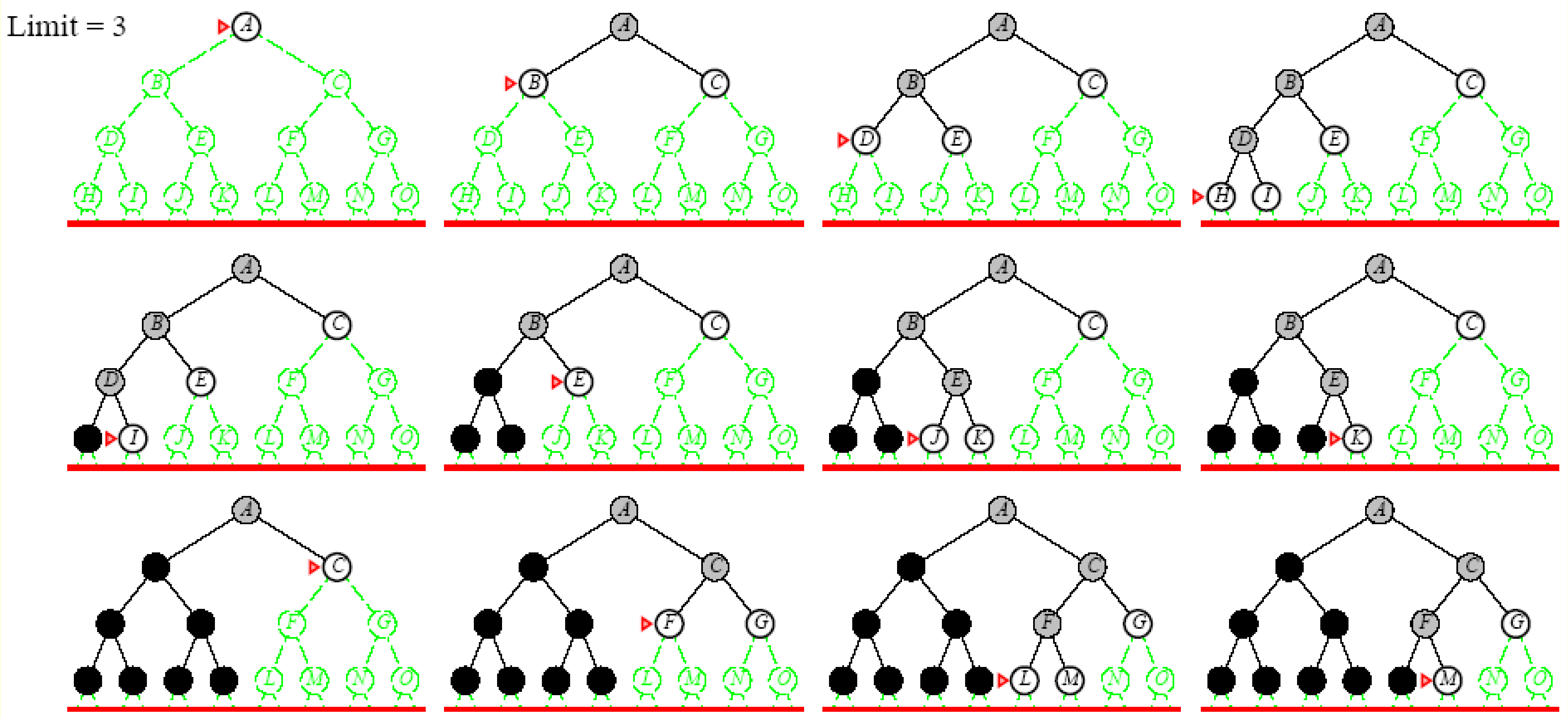
Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search

- Complete
  - Yes
- Time
  - $O(b^d)$
- Space
  - $O(bd)$
- Optimal
  - Yes if step cost = 1
  - Can be modified to explore uniform cost tree

Lessons From Iterative Deepening Search

- Faster than BFS even though IDS generates repeated states
  - BFS generates nodes up to level d+1
  - IDS only generates nodes up to level d

- In general, iterative deepening search is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

Avoiding Repeated States

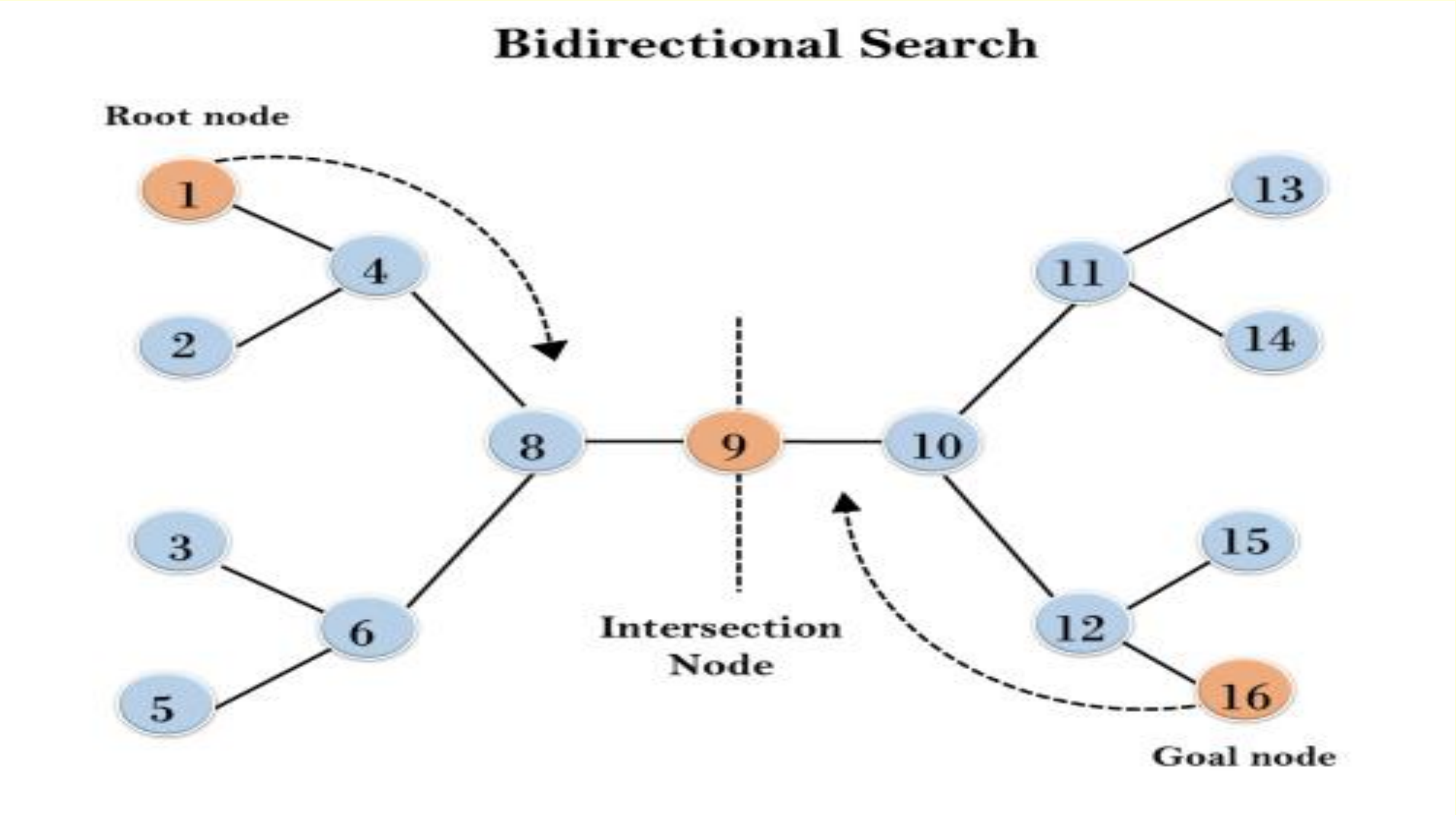
- Complication of wasting time by expanding states that have already been encountered and expanded before
  - Failure to detect repeated states can turn a linear problem into an exponential one

- Sometimes, repeated states are unavoidable
  - Problems where the actions are reversable
    - Route finding
    - Sliding blocks puzzles

Bidirectional Search Algorithm

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Bidirectional Search Algorithm



Bidirectional Search Algorithm

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Bidirectional Search Algorithm

- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is $\mathbf{O(b^{d/2})}$.
- **Space Complexity:** Space complexity of bidirectional search is $\mathbf{O(b^{d/2})}$.
- **Optimal:** Bidirectional search is Optimal.