Fundamentals of Artificial Intelligence

Solving Problems by Searching

Problem-Solving Agent

- A **problem-solving agent** is a *goal-based agent* and use *atomic representations*.
  - In atomic representations, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.
- *Intelligent agents* are supposed to maximize their *performance measure*. Achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.
- **Problem formulation** is the process of deciding what actions and states to consider, given a *goal*.
- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A ***search algorithm*** takes a problem as input and returns a **solution** in the form of an action sequence.
- Once a *solution* is found, the carrying actions it recommends is called the **execution phase**.

- A problem-solving agent has three phases:
  - **problem formulation, searching solution and executing actions in the solution.**

Problem-Solving Agent
Well-defined problems

A **problem** can be defined by five components:

- **initial state, actions, transition model, goal test, path cost.**

INITIAL STATE: The **initial state** that the agent starts in.

ACTIONS: A description of the possible **actions** available to the agent.

- Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s.
- Each of these actions is **applicable** in s.

TRANSITION MODEL: A description of what each action does is known as the **transition model**

- A function RESULT(s,a) that returns the state that results from doing action a in state s.
- The term **successor** to refer to any state reachable from a given state by a single
- The **state space** of the problem is the set of all states reachable from the *initial state* by any sequence of actions.
- The *state space* forms a **graph** in which the nodes are states and the links between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.

Problem-Solving Agent
Well-defined problems

GOAL TEST: The **goal test** determines whether a given state is a goal state.

PATH COST: A **path cost** function that assigns a numeric cost to each path.

- The problem-solving agent chooses a cost function that reflects its own performance measure.
- The **cost of a path** can be described as the sum of the costs of the individual actions along the path.
- The **step cost** of taking action a in state s to reach state s′ is denoted by c(s, a, s′).

- A **SOLUTION** to a problem is an action sequence that leads from the ***initial state*** to a ***goal state***.
- Solution quality is measured by the path cost function, and an **OPTIMAL SOLUTION** has the lowest path cost among all solutions.

Problem Example
Travelling in Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

Formulate goal:

- be in Bucharest

Formulate problem:

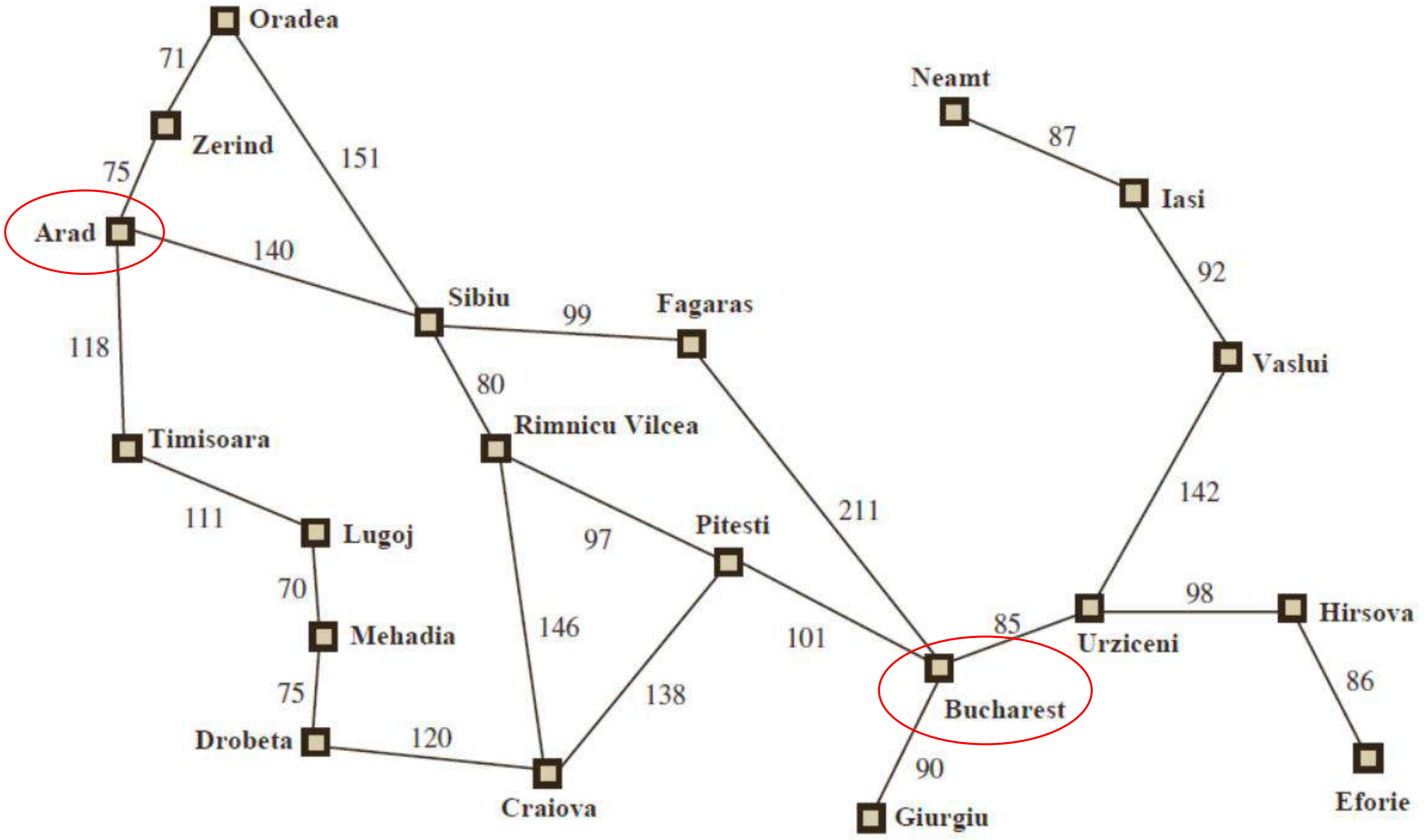
- states: various cities
- actions: drive between cities

Find solution:

- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Problem Example

Travelling in Romania: A simplified road map



Problem Example
Travelling in Romania

- Travelling in Romania problem can be defined by:

initial state: in Arad

actions: transition model: is the graph

successor function S(x) = set of action{state pairs

e.g., S(Arad) = { <Arad → Zerind,Zerind>, <Arad → Sibiu,Sibiu>,

< Arad → Timisoara,Timisoara> }

goal test: in Bucharest

path cost: sum of distances, number of actions executed, etc.

c(x; a; y) is the **step cost**, assumed to be ≥ 0

- A **solution** is a sequence of actions leading from the *initial state* to a *goal state*

Problem Example

vacuum world

States:

- The state is determined by both the agent location and the dirt locations.
- The agent is in one of two locations, each of which might or might not contain dirt.
- Thus, there are $2 \times 2^2 = 8$ possible world states.

Initial state: Any state can be designated as the initial state.

Actions: Each state has just three actions: *Left*, *Right*, and *Suck*.

Transition model:

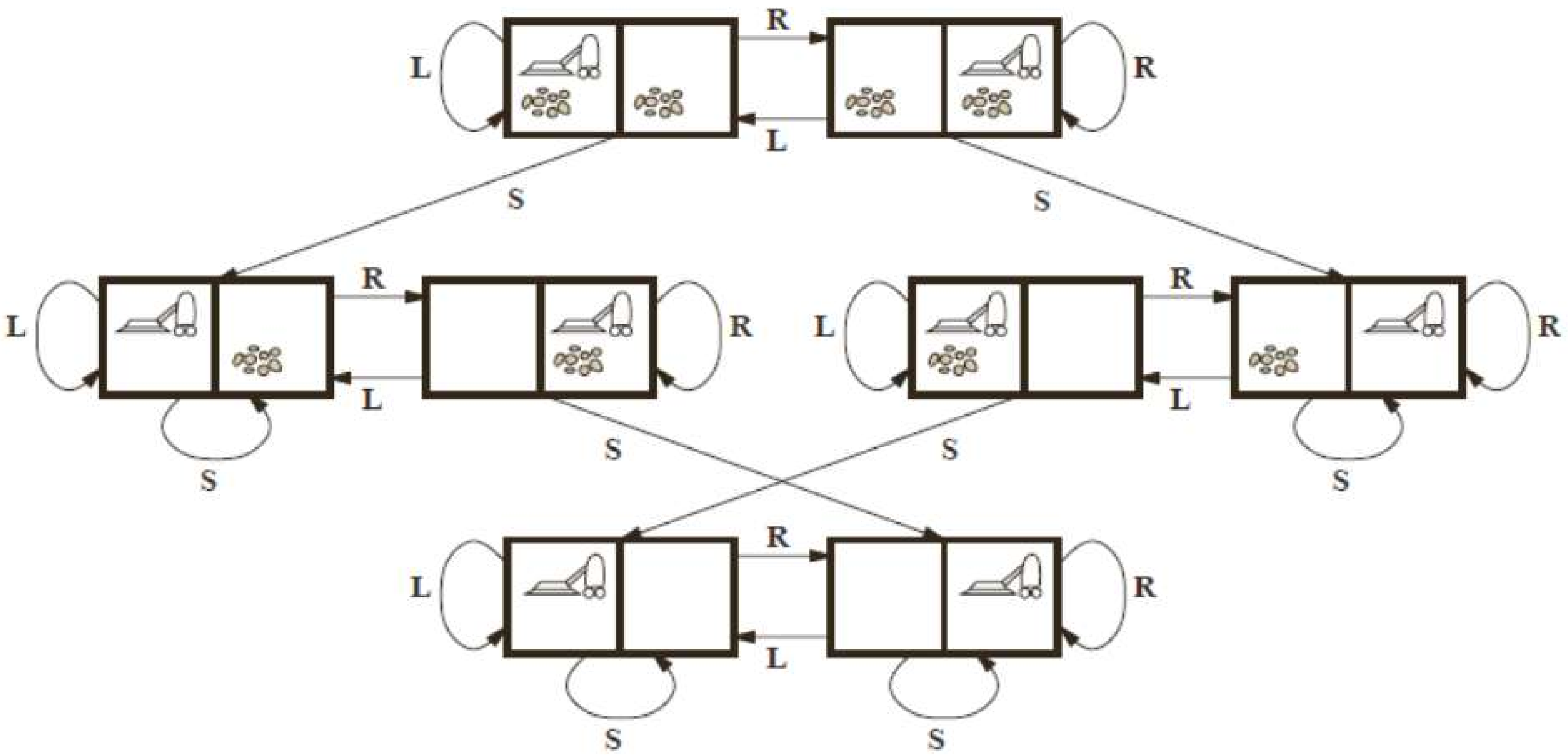
- The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect.
- The transition model defines a state space.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

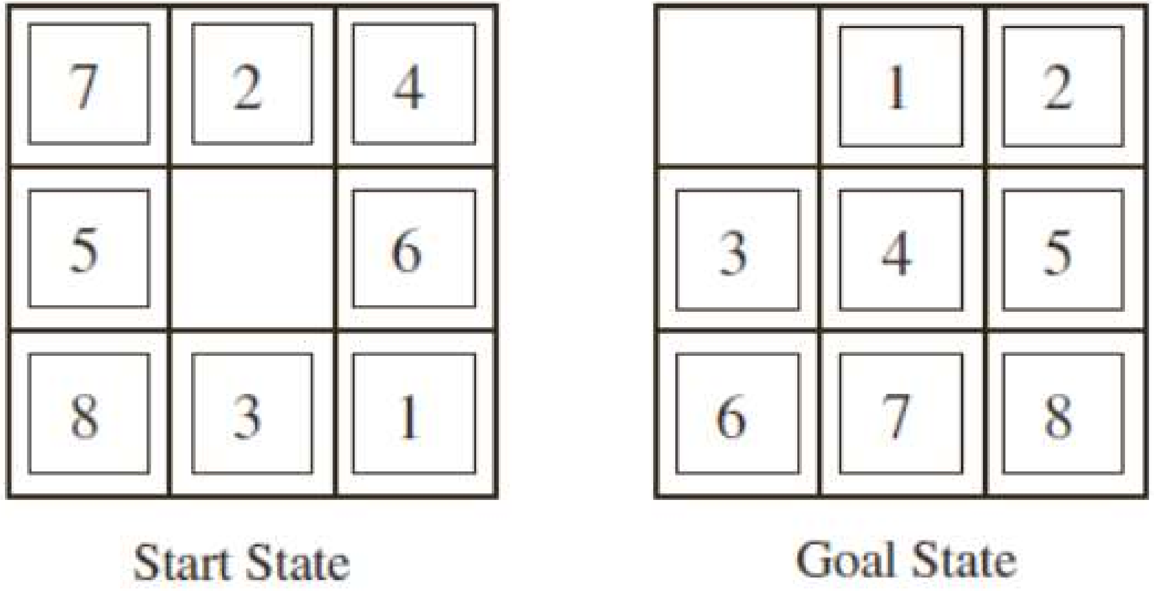
Problem Example
vacuum world: state space for the vacuum world

- Links denote actions: L = Left, R =Right, S = Suck.



Problem Example
8-puzzle

- The **8-puzzle** consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The object is to reach a specified goal state.



Problem Example
8-puzzle

States: A state specifies the location of each of the eight tiles and the blank in one of the nine squares.

Initial state: Any state can be designated as the initial state.

- Note that goal can be reached from exactly half of the possible initial states.

Actions: Movements of the blank space ***Left***, ***Right***, ***Up***, or ***Down***.

- Different subsets of these are possible depending on where the blank is.

Transition model: Given a state and action, this returns the resulting state;

Goal test: This checks whether the state matches the goal configuration

Path Cost: Each step costs 1, so the path cost is the number of steps in the path.

Problem Example
8-puzzle

- The 8-puzzle belongs to the family of **sliding-block puzzles**,
- This family is known to be **NP-complete**.
  - Optimal solution of n-Puzzle family is NP-hard. ie NO polynomial solution for the problem.

- The 8-puzzle has 9!/2=181, 440 reachable states.
- The 15-puzzle (on a 4×4 board) has around 1.3 trillion states,
- The 24-puzzle (on a 5 × 5 board) has around 10^{25} states

Real-World Problems

- **Route-finding problem** is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications such as Web sites and in-car systems that provide driving directions.

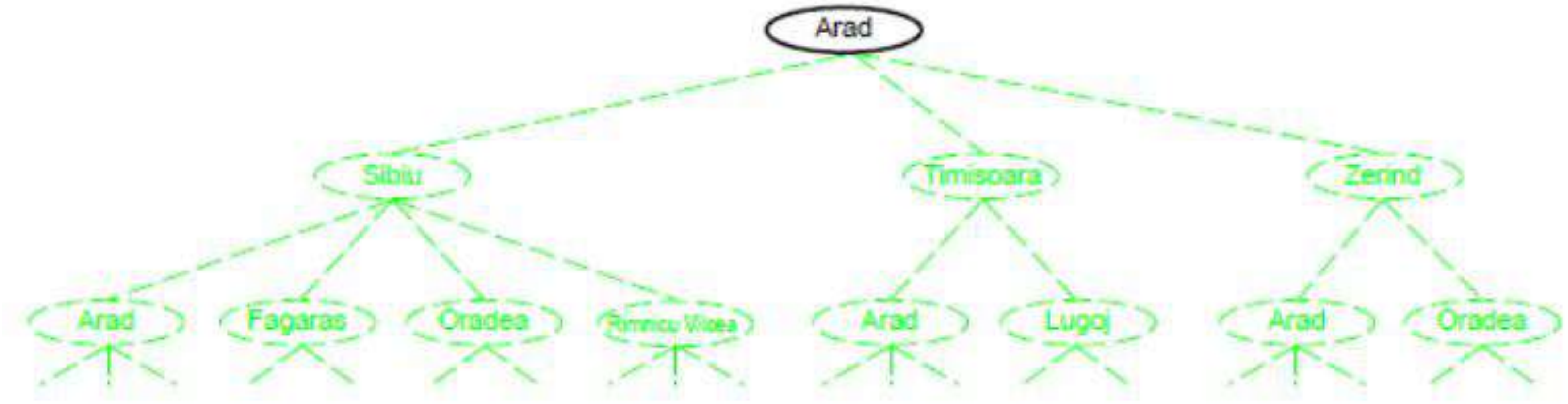
- **VLSI layout problem** requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.

Searching for Solutions

- A solution is an action sequence and **search algorithms** considers various possible action sequences.
- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root;
- The **branches** are **actions** and the **nodes** correspond to **states** in the state space of the problem.
- **Expanding** the current state is application of each legal action to the current state and generation of a new set of states.
  - The current state is the **parent node**, newly generated states are **child nodes**
- **Leaf node** is a node with no children in the tree.
- The ***set of all leaf nodes available*** for expansion at any given point is called the **frontier**.
- **Search algorithms** all share this basic structure; they vary primarily according to how they choose which state to expand next: **search strategy**.

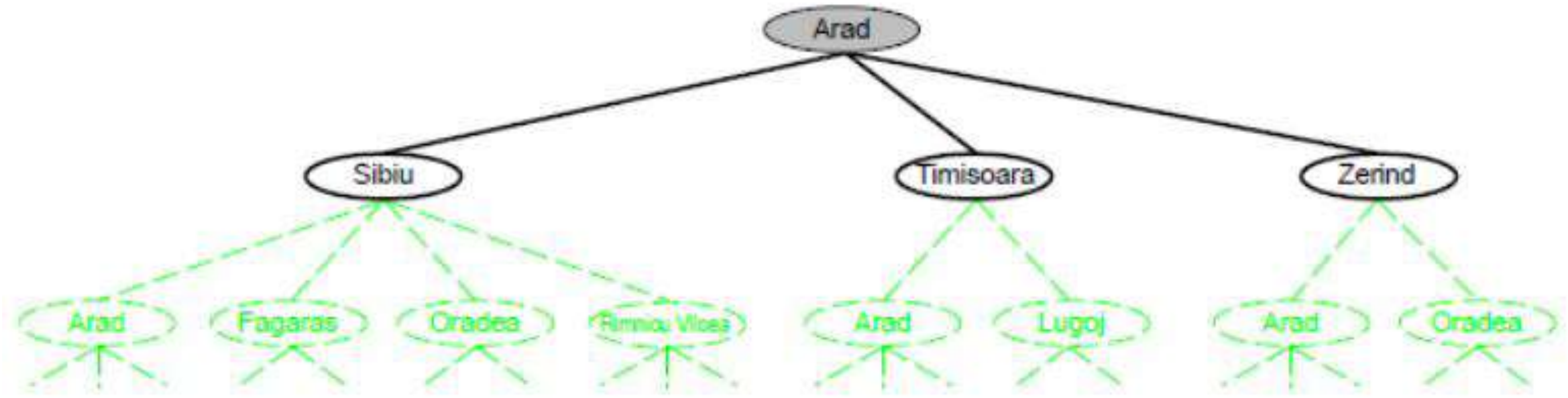
Partial Search Trees for Travelling in Romania

initial state



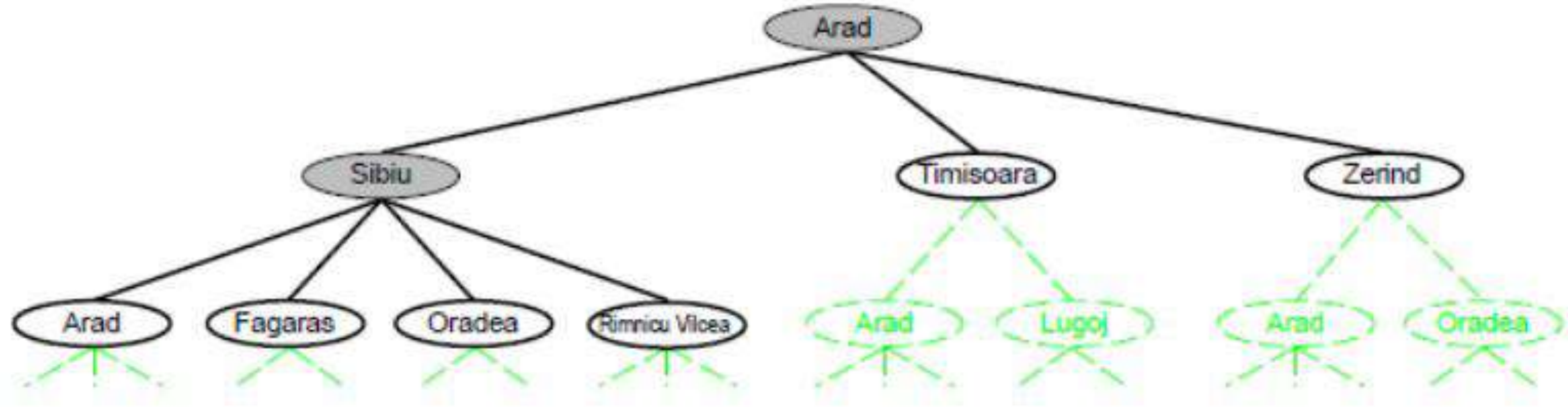
Partial Search Trees for Travelling in Romania

After expanding Arad



Partial Search Trees for Travelling in Romania

After expanding Sibiu



Infrastructure for Search Algorithms

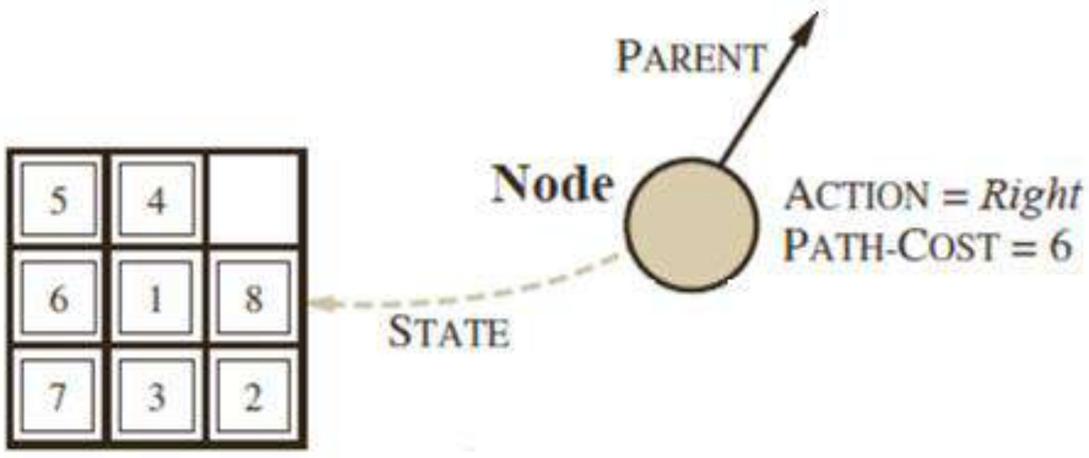
- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- Each node n of the tree contains four components:

  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 of the path from the initial state to the node,



Informal Description of Graph Search Algorithms

function GRAPH-SEARCH(***problem***) **returns** a solution, or failure

- initialize ***frontier*** using ***initial state*** of ***problem***
- initialize ***explored set*** to be empty
- **loop do**
  - **if** ***frontier*** is empty **then return** *failure*
  - choose a ***leaf node*** from ***frontier*** and remove it from there
  - **if** ***node*** contains a ***goal state*** **then return** corresponding *solution*
  - add ***node*** to ***explored set***
  - expand ***node***, adding ***resulting nodes*** to ***frontier*** only if not in ***frontier*** or ***explored set***

Informal Description of Graph Search Algorithms

- To avoid exploring ***redundant paths*** is to remember them.
- **Explored set** (also known as *closed list*) remembers every expanded node.
- The **frontier** needs to be stored in such a way that ***search algorithm*** can easily choose next node to expand according to its ***preferred strategy***.
  - The appropriate data structure for this is a ***queue***.
- **Queues** are characterized by the order in which they store the inserted nodes.
  - First-in, first-out or FIFO queue, which pops the oldest element of the queue; (**QUEUE**)
  - Last-in, first-out or LIFO queue (also known as **STACK**), which pops the newest element
  - **PRIORITY QUEUE**, which pops the element of the queue with the highest priority according to some ordering function.
- The ***explored set*** can be implemented with a hash table to allow efficient checking for repeated states.

Measuring Problem-Solving Performance

- We can evaluate a **search algorithm's performance** in four ways:

  Completeness: Is the algorithm guaranteed to find a solution when there is one?

  Optimality: Does the strategy find the optimal solution?

  Time complexity: How long does it take to find a solution?

  Space complexity: How much memory is needed to perform the search?

Measuring Problem-Solving Performance

- The typical measure for ***time and space complexity*** is the size of the state space graph, $|V| + |E|$, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).
  - This is appropriate when the graph is an explicit data structure.

- In AI, graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:
  - **b - maximum branching factor of the search tree**
  - **d - depth of the least-cost solution**
  - **m - maximum depth of the state space (may be ∞)**

- Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

Uninformed Search Strategies

- **Uninformed strategies** use only the information available in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- All search strategies are distinguished by the order in which nodes are expanded.

- Uninformed (blind) Search Strategies are:
    - **Breadth-first Search**
    - **Uniform-cost Search**
    - **Depth-first Search**
    - **Depth-limited Search**
    - **Iterative Deepening Search**

Breadth-First Search

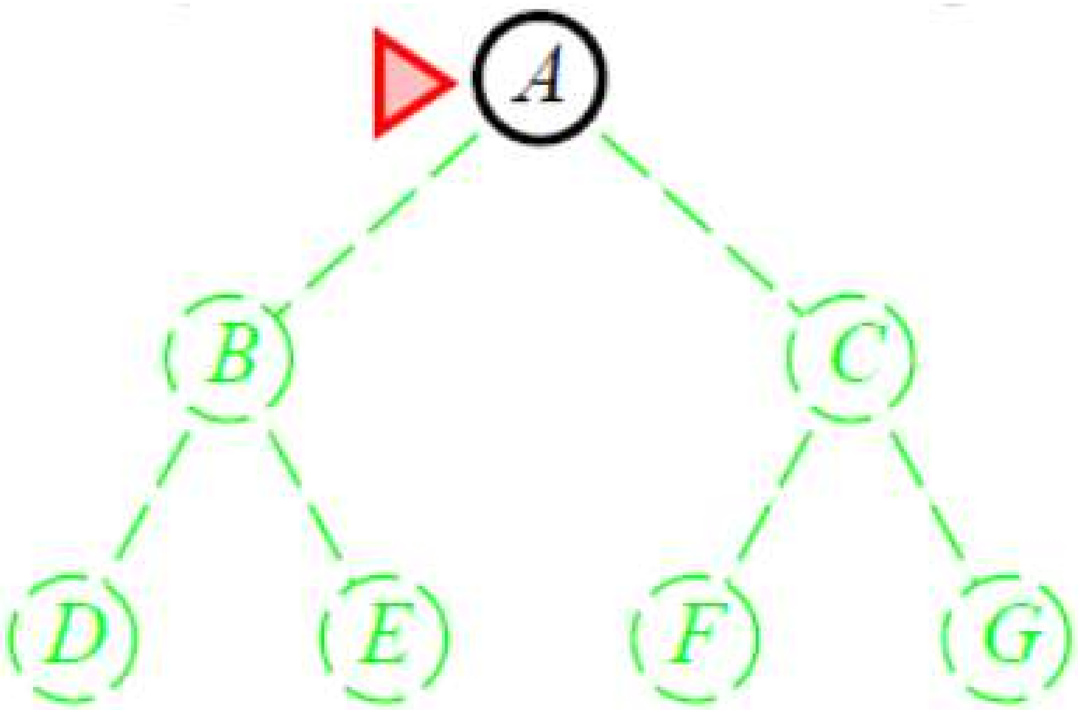
- **Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
    - All nodes are expanded at a given depth in search tree before any nodes at next level are expanded.

- **Breadth-first search** uses a **FIFO** queue for the ***frontier***

Breadth-First Search

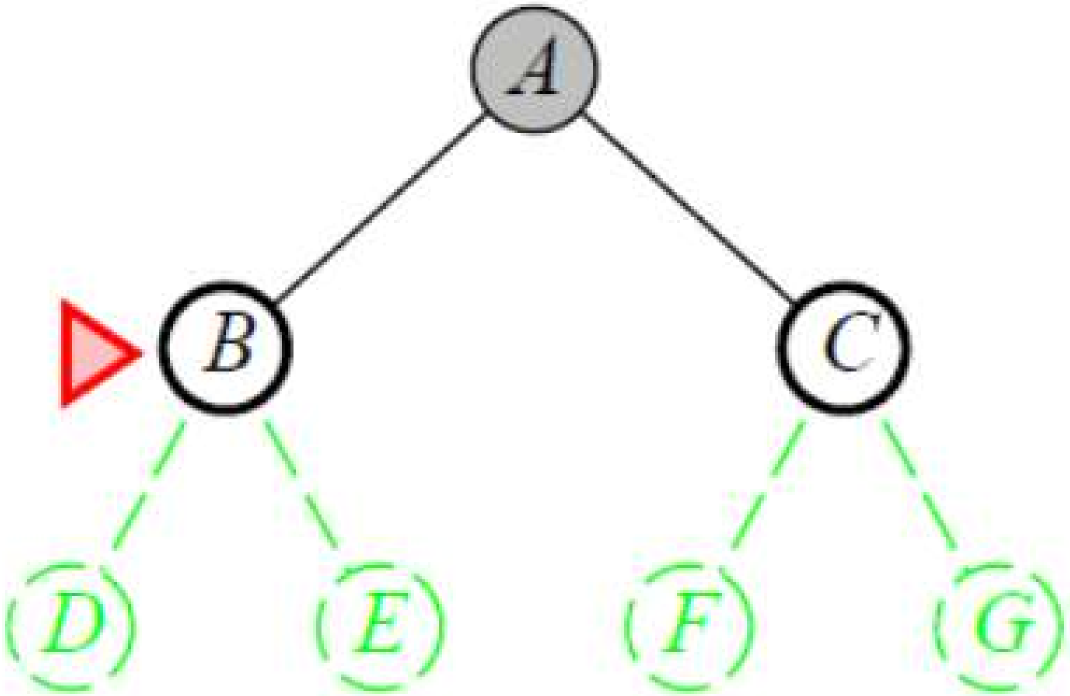
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop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



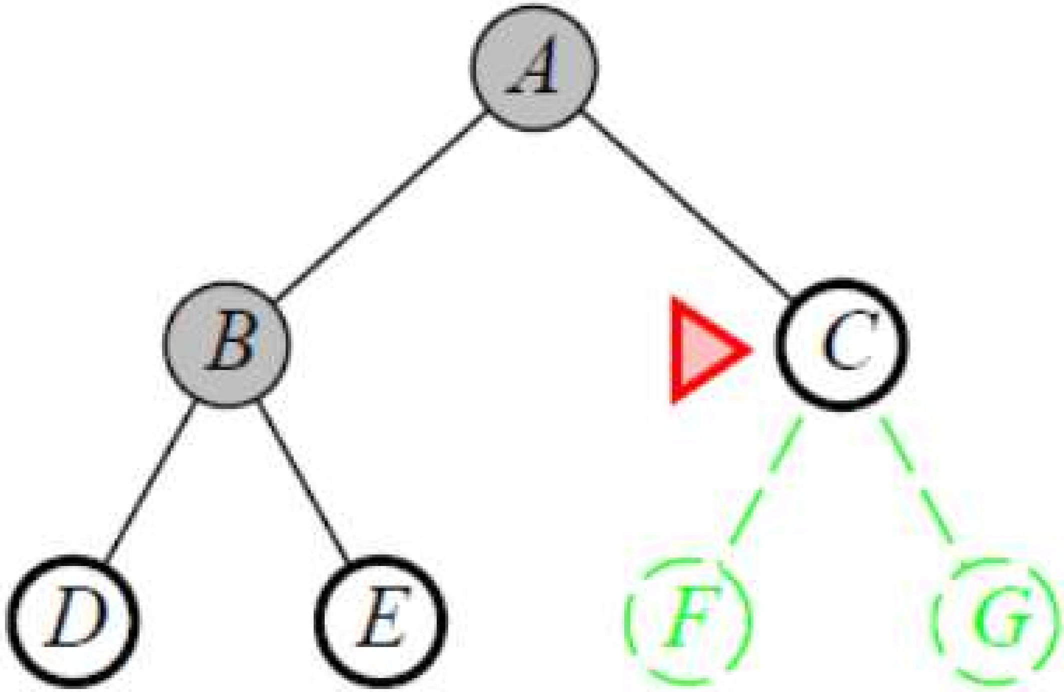
Frontier	Explored
A	empty

Breadth-First Search: Example



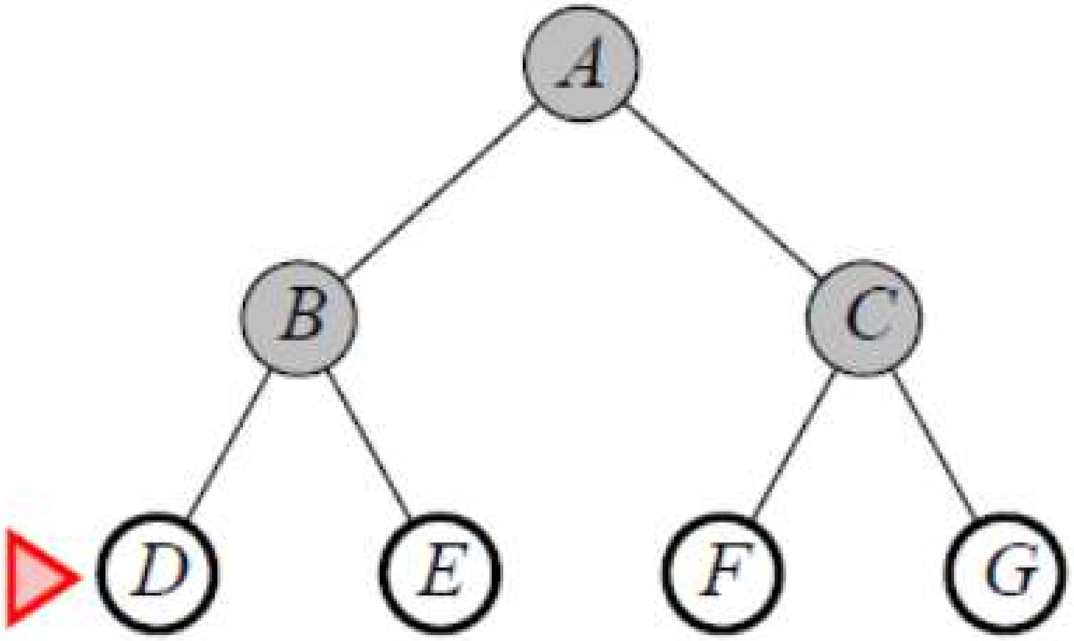
Frontier	Explored
B	A
C	

Breadth-First Search: Example



Frontier	Explored
C	A, B
D	
E	

Breadth-First Search: Example



Frontier	**Explored**
D	A, B, C
E	
F	
G	

Properties of Breadth-First Search

Complete? Yes if branching factor b is finite

Time? $1 + b + b^2 + b^3 + : : : + b^d = O(b^{d+1})$, i.e., exponential in depth d

Space? $O(b^{d+1})$ (keeps every node in memory)

Optimal? Yes (if cost = 1 per step); not optimal in general

Breadth-First Search

- Time and memory requirements for breadth-first search assuming branching factor b = 10; 1 million nodes/second; 1000 bytes/node.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

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

Uniform-Cost Search

- When all ***step costs*** are equal, **breadth-first search** is **optimal** because it always expands the shallowest unexpanded node.
  - In general, it is not optimal when step costs are different.
  - By a simple extension, we can find an algorithm that is **optimal** with ***any step-cost function***.
- **Uniform-cost search** expands node n with the ***lowest path cost g(n)***.
- This is done by storing the ***frontier*** as a ***priority queue*** ordered by g.
- **Uniform-cost search** is equivalent to breadth-first if step costs are all equal

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
```

Properties of Uniform-Cost Search

Complete? Yes if *step cost* $\geq \epsilon$ where cost of every step exceeds positive constant ϵ.

Time? $O(b^{C/e})$ where C is the cost of the optimal solution and e is ϵ

The cost of generating all nodes whose costs $\leq$ cost of optimal solution

Space? $O(b^{C/e})$ where C is the cost of the optimal solution and e is ϵ

Keeping all nodes whose *costs* $\leq$ *cost of optimal solution*

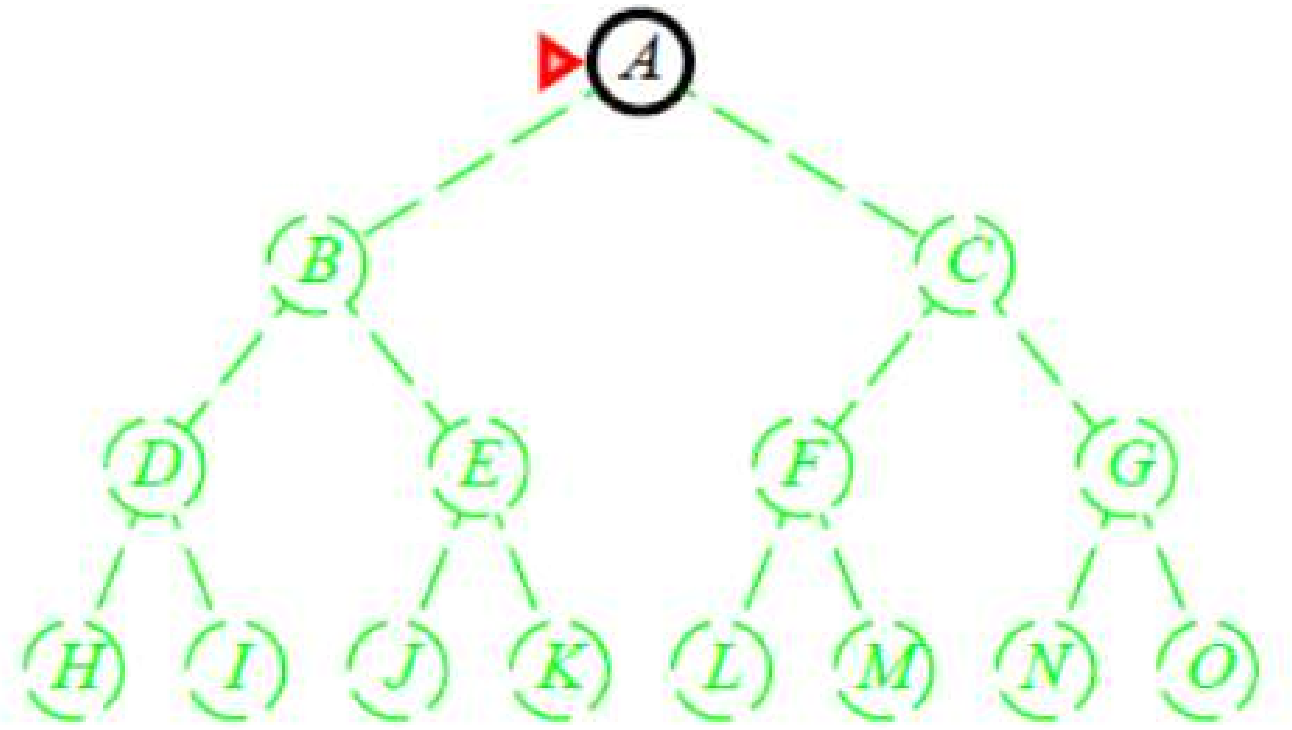
Optimal? Yes nodes expanded in increasing order of path cost

Depth-First Search

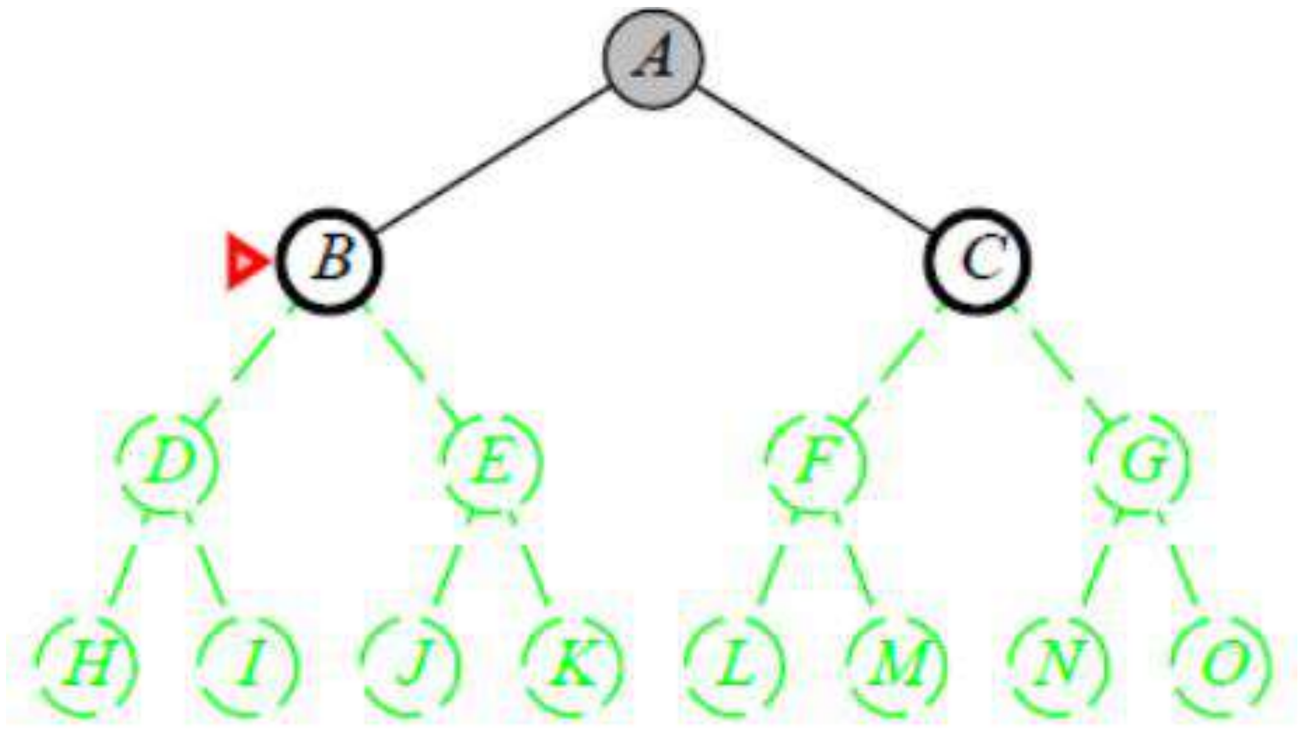
- **Depth-first search** always expands *deepest unexpanded node* in *frontier* of search tree.
  - As nodes are expanded, they are dropped from frontier, so then search “backs up” to next deepest node that still has unexplored successors.
- **Depth-first search** uses a **LIFO queue (STACK).**
  - A LIFO queue means that the most recently generated node is chosen for expansion.
  - This must be the deepest unexpanded node because it is one deeper than its parent.

Depth-First Search

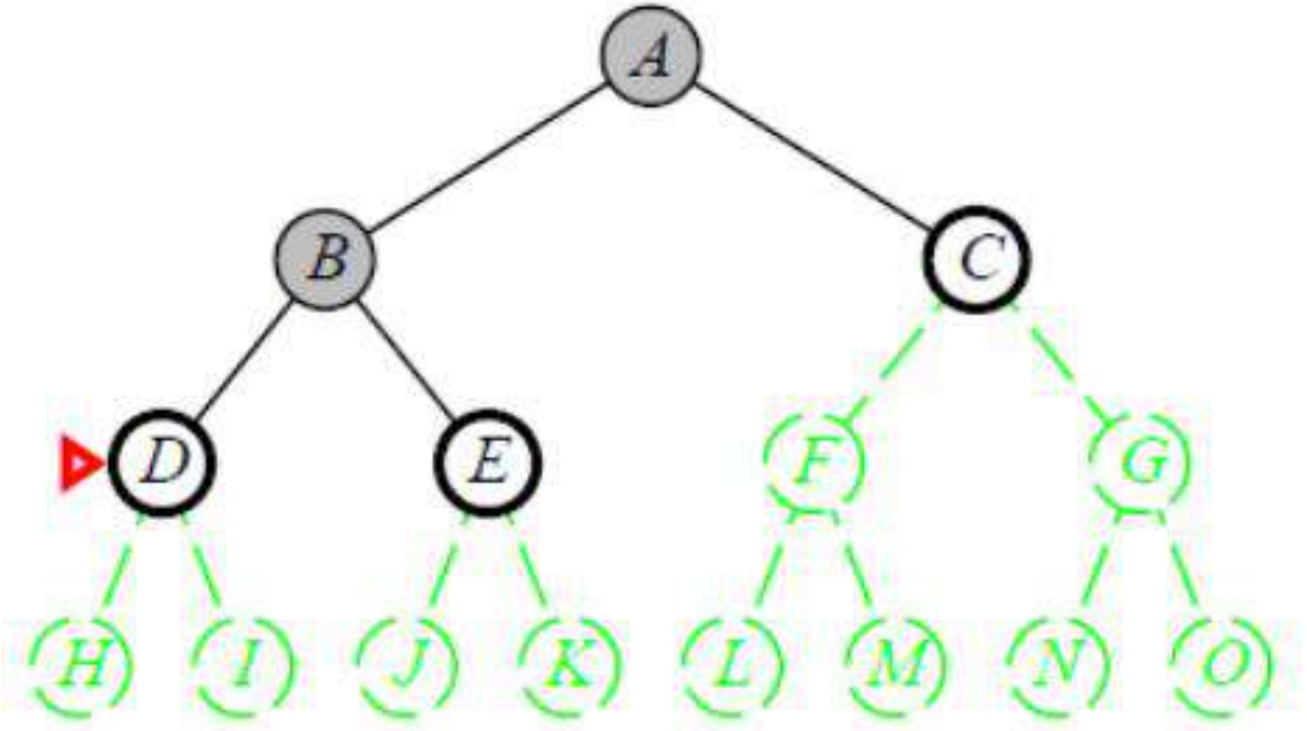
- Depth-first search on a binary tree.
- Explored nodes with no descendants in the frontier are removed from memory.
- Nodes at depth 3 have no successors and M is the only goal node.



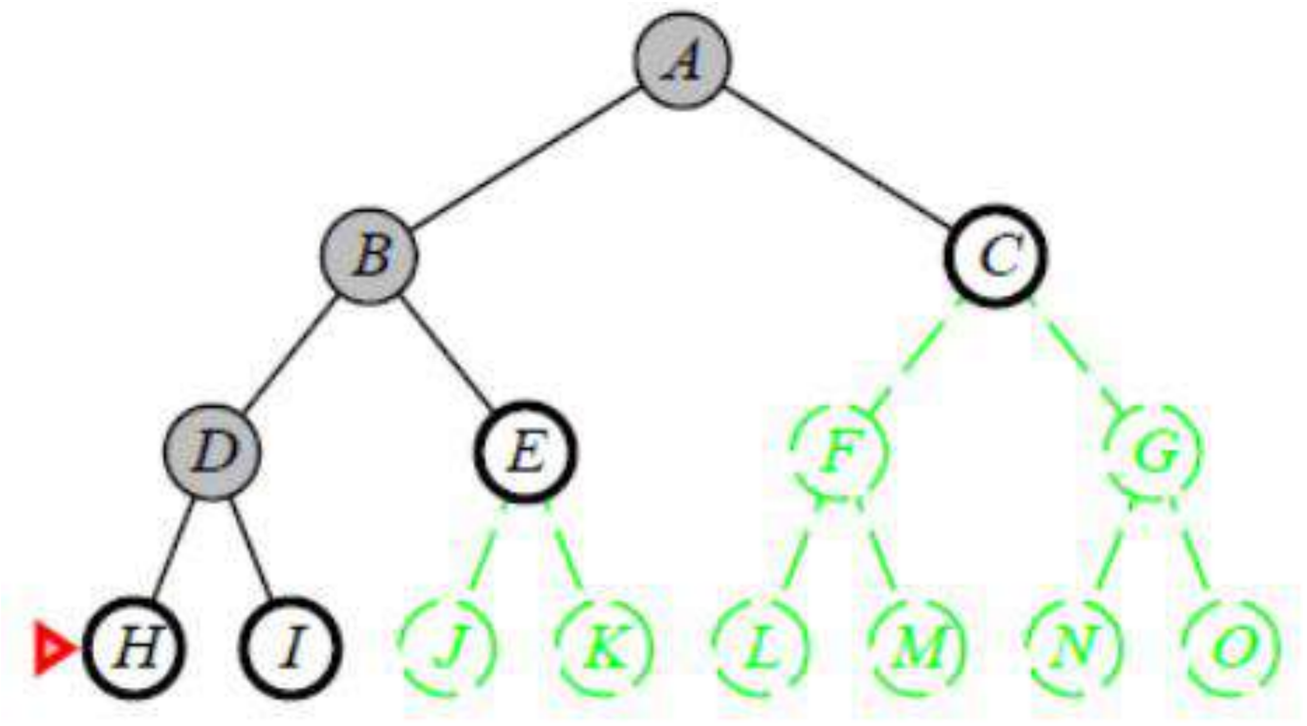
Depth-First Search



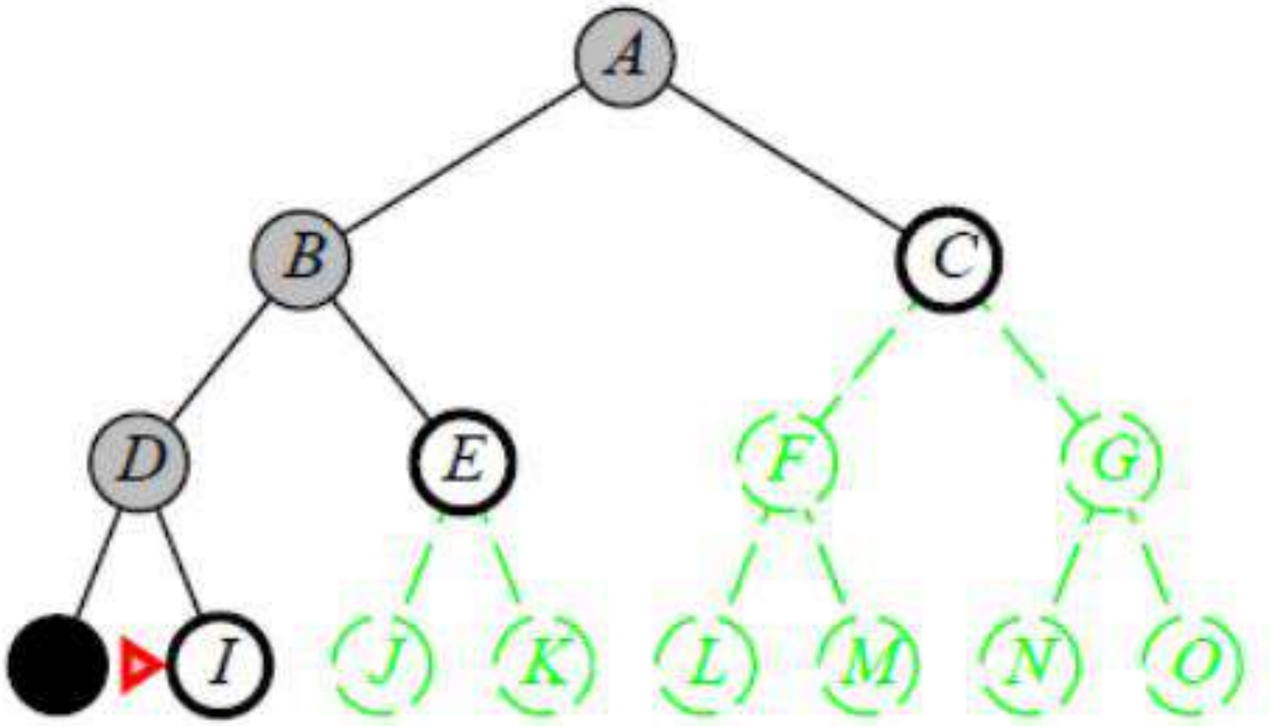
Depth-First Search



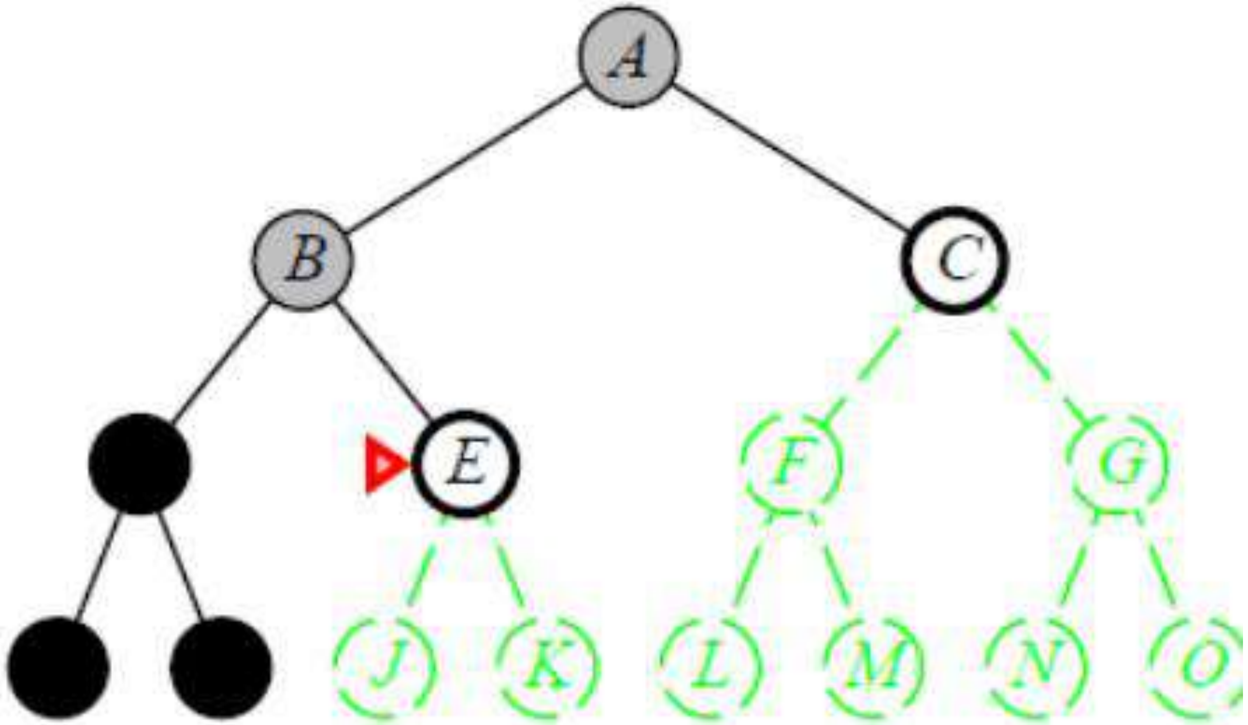
Depth-First Search



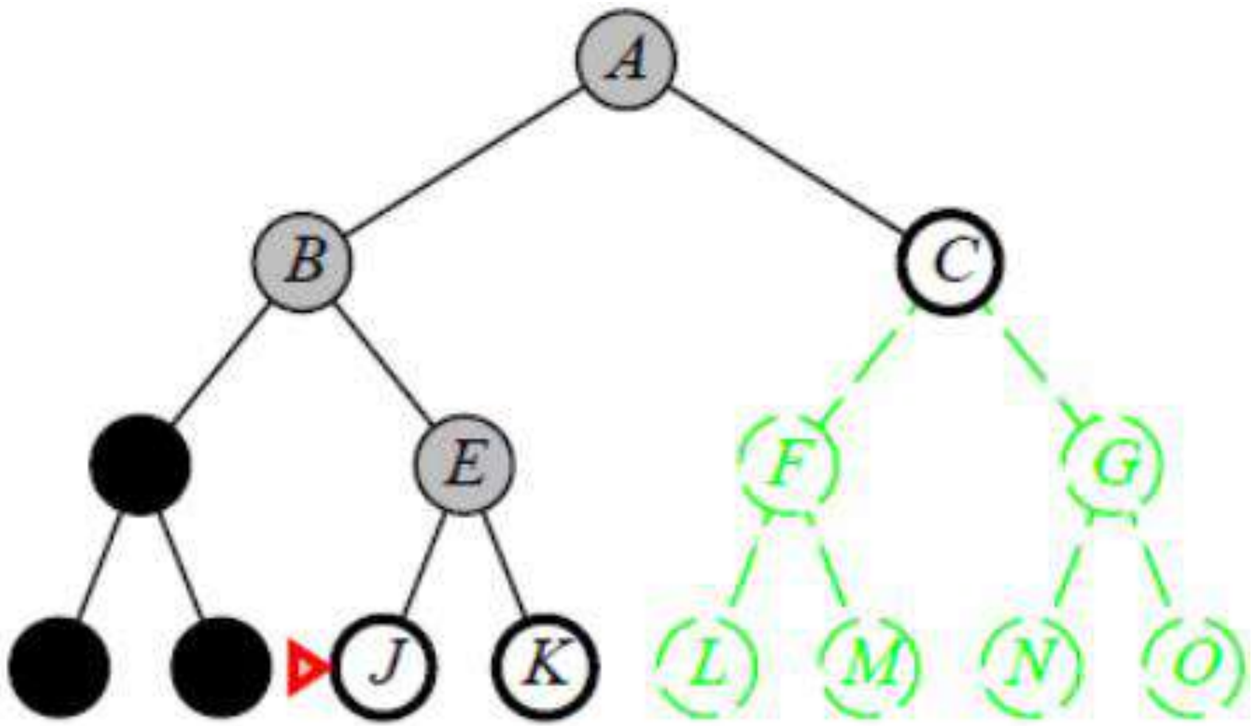
Depth-First Search



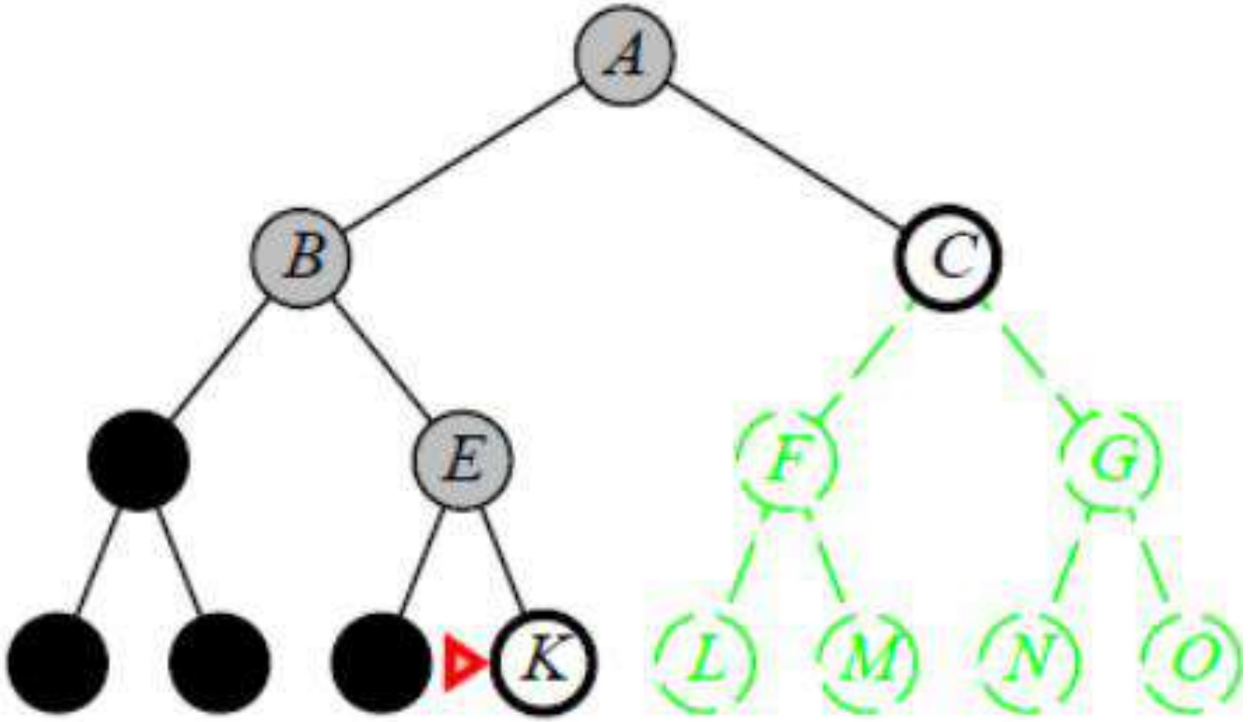
Depth-First Search



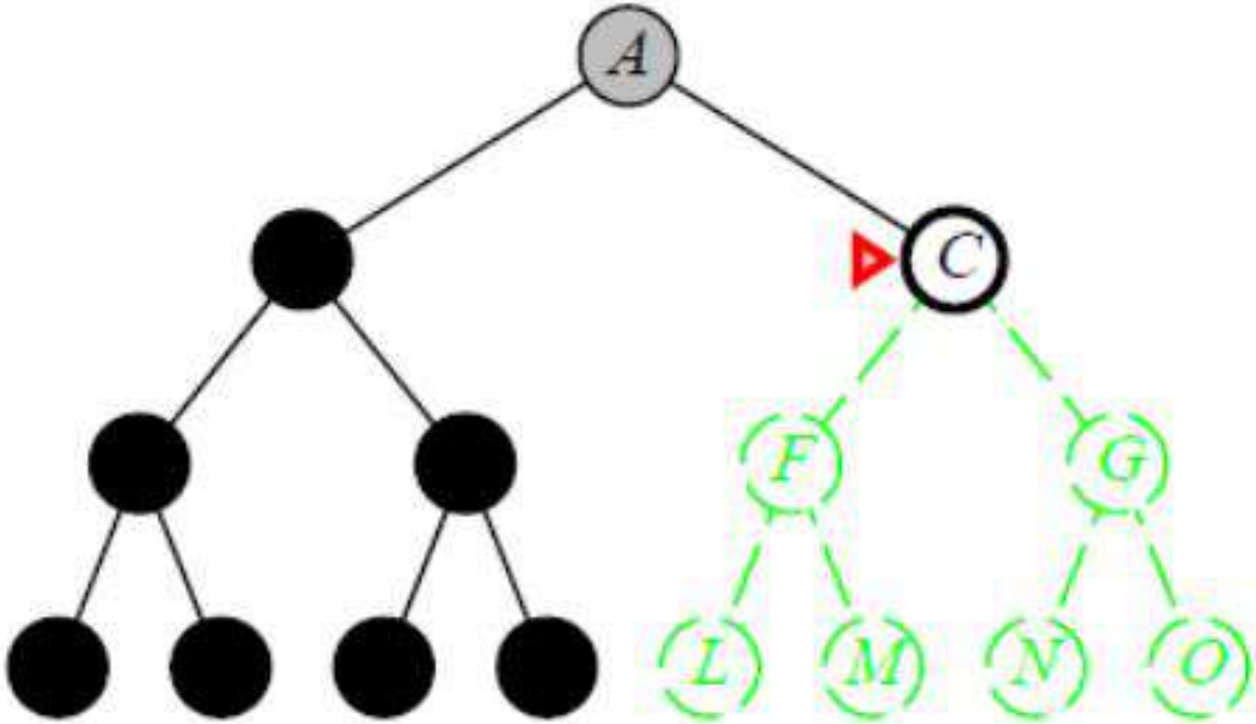
Depth-First Search



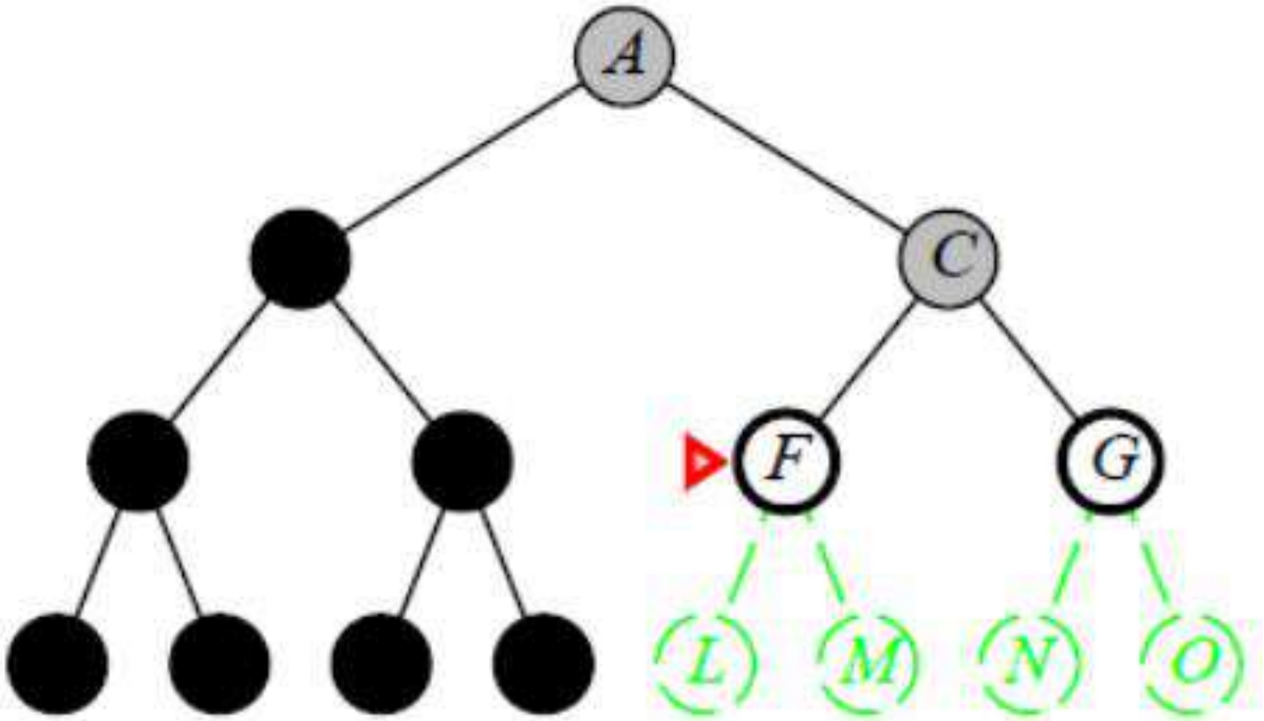
Depth-First Search



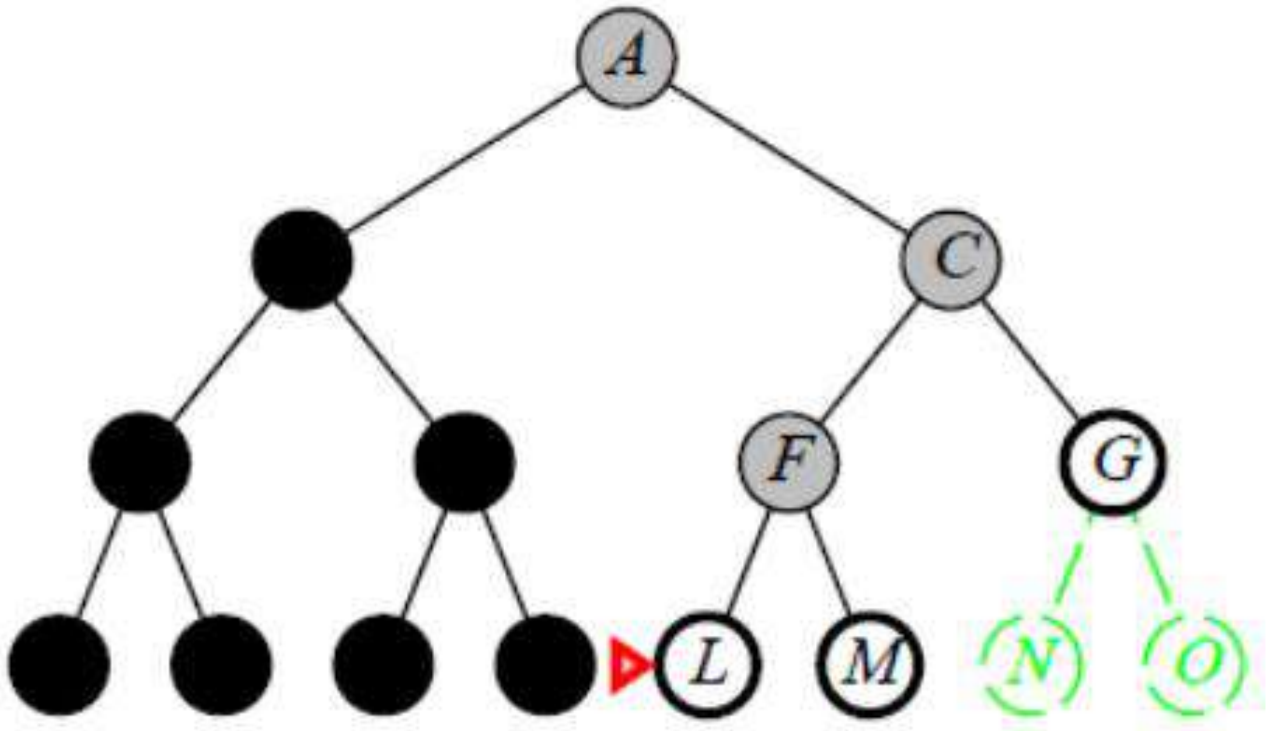
Depth-First Search



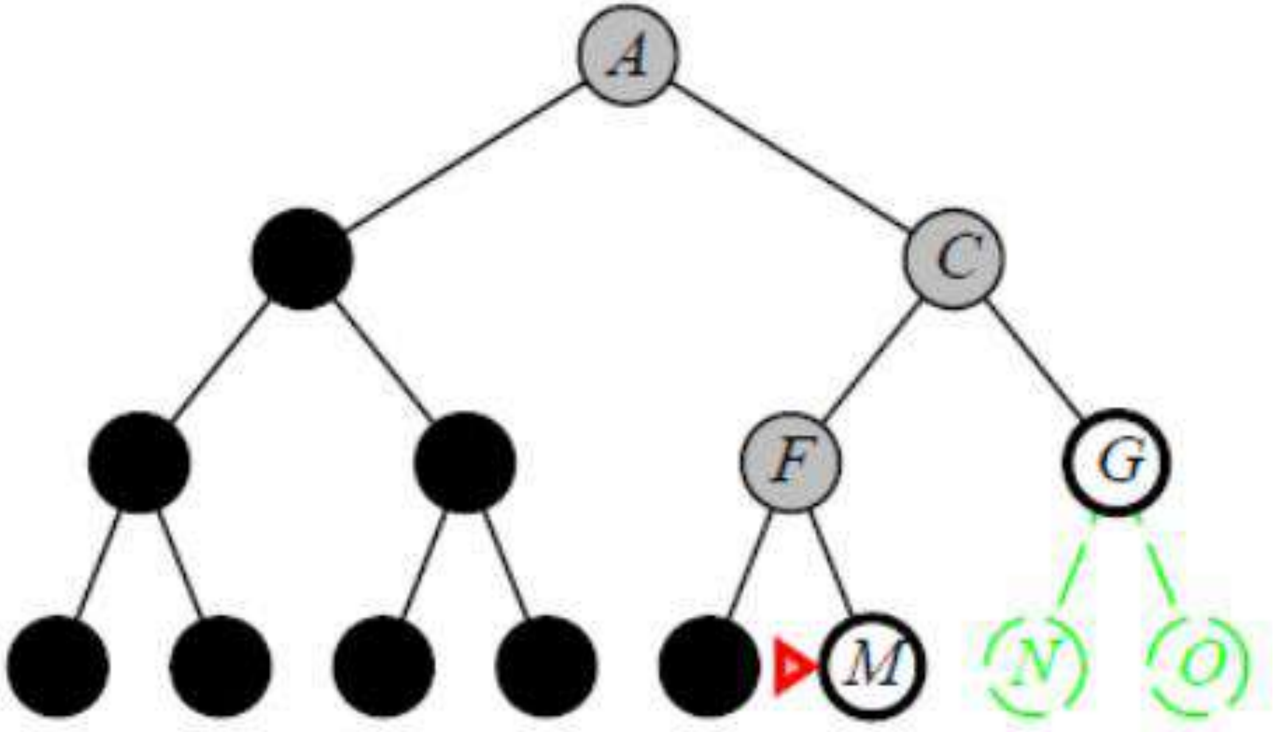
Depth-First Search



Depth-First Search



Depth-First Search



- Goal (M) is found

Properties of depth-first search

Complete? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path ➔ complete finite spaces

Time? $O(b^m)$ where m is the maximum depth of search tree

terrible if m is much larger than d (depth of shallowest solution)

but if solutions are dense, may be much faster than breadth-first

Space? O(bm) linear space

Optimal? No

Depth-Limited Search

- The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ℓ.
- Nodes at depth ℓ are treated as if they have no successors.
- This approach is called **depth-limited search**.

- The depth limit solves the infinite-path problem.
- Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit.
- Depth-limited search will also be non-optimal if we choose $\ell > d$.
- Its time complexity is $O(b^\ell)$ and its space complexity is $O(b\ell)$.
- Depth-first search can be viewed as a special case of depth-limited search with $\ell=\infty$.

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
```

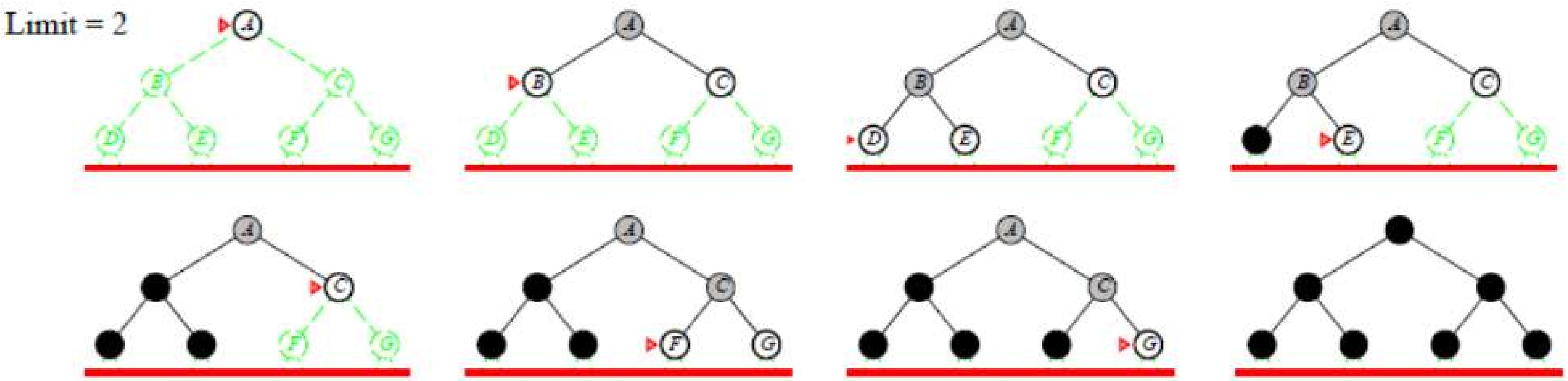
Iterative Deepening Depth-First Search

```
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
```

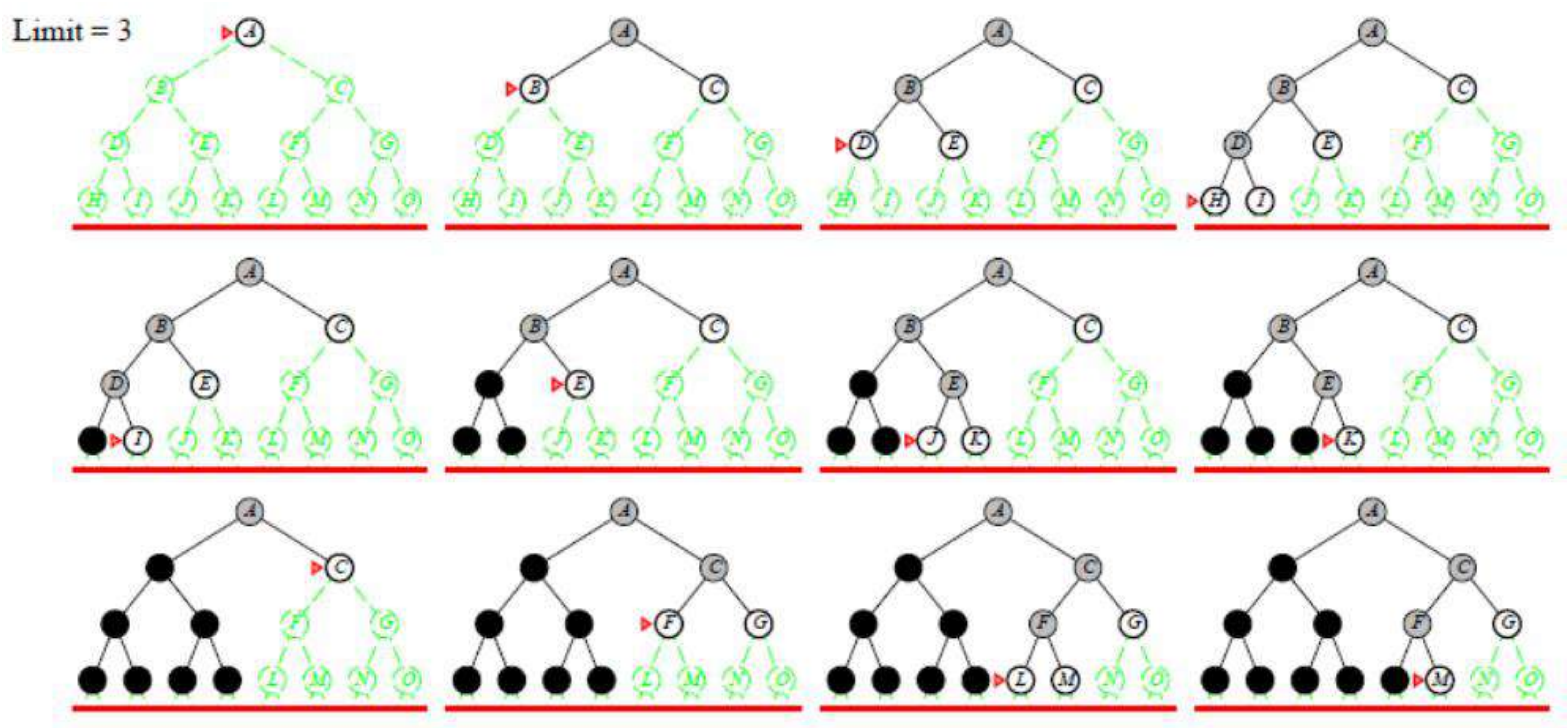
Iterative deepening search l = 0

Iterative deepening search l = 1

Iterative deepening search l = 2



Iterative deepening search l = 3



Properties of iterative deepening search

Complete?	Yes	
Time?	$O(b^d)$	
Space?	$O(bd)$	linear space
Optimal?	Yes	if step cost = 1

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

- b is the branching factor; d is the depth of the shallowest solution;
- m is the maximum depth of the search tree; l is the depth limit.
- Superscripts:
  - a complete if b is finite;
  - b complete if step costs $\geq \epsilon$ for positive ϵ;
  - c optimal if step costs are all identical;