Solving Problem by Searching : Uninformed Search

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Three (3)**	**Week No:**	**Three (3)**	**Semester:**	
Lecturer:	*Dr. Abdus Salam*			*abdus.salam@aiub.edu*	

Lecture Outline

1. Problem-Solving Agents
2. Example Problems
3. Searching for Solutions
4. Uninformed Search Strategies

PROBLEM SOLVING AGENTS

Simple Reflexive Agent

Base their action on a direct mapping from STATEs to ACTIONs
Uses Rule

Goal Based Agent

Consider future Actions and the desirability of their outcomes

Problem Solving Agent

- Goal based Agent
- Use Atomic representation of Environment

Steps in problem solving

- ❖ Goal Formulation
- ❖ Problem Formulation
- ❖ Search
- ❖ Solution
- ❖ Execution

GOAL FORMULATION

Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Goal formulation is the first step in problem solving. It is based on the current situation and the agent’s performance measure,

Goal: A set of world states—exactly those states in which the goal is satisfied.

The agent’s task is to find out how to act, now and in the future, so that it reaches a goal state.

PROBLEM FORMULATION: BASED ON ENVIRONMENT

Problem formulation is the process of deciding what **actions** and **states** to consider, given a goal.

In an **Unknown** environment the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action.

If the agent has no additional information—i.e., if the environment is **unknown** in the sense defined in Section 2.3—then it is having no choice but to try one of the actions at **random**.

*an agent with several immediate options of **unknown value** can decide what to do by first **examining*** **future** ***actions*** *that eventually lead to **states of known value.***

SEARCH-SOLUTION-EXECUTE: OPEN LOOP SYSTEM

The process of looking for a sequence of actions that reaches the goal is called **search**. **Solution** is the sequence of actions that takes any agent to the goal state, exactly those state the agent is satisfied.

A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.

While the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be. An agent that carries out its plans with its eyes closed must be quite certain of what is going on.

Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.

ROMANIAN MAP

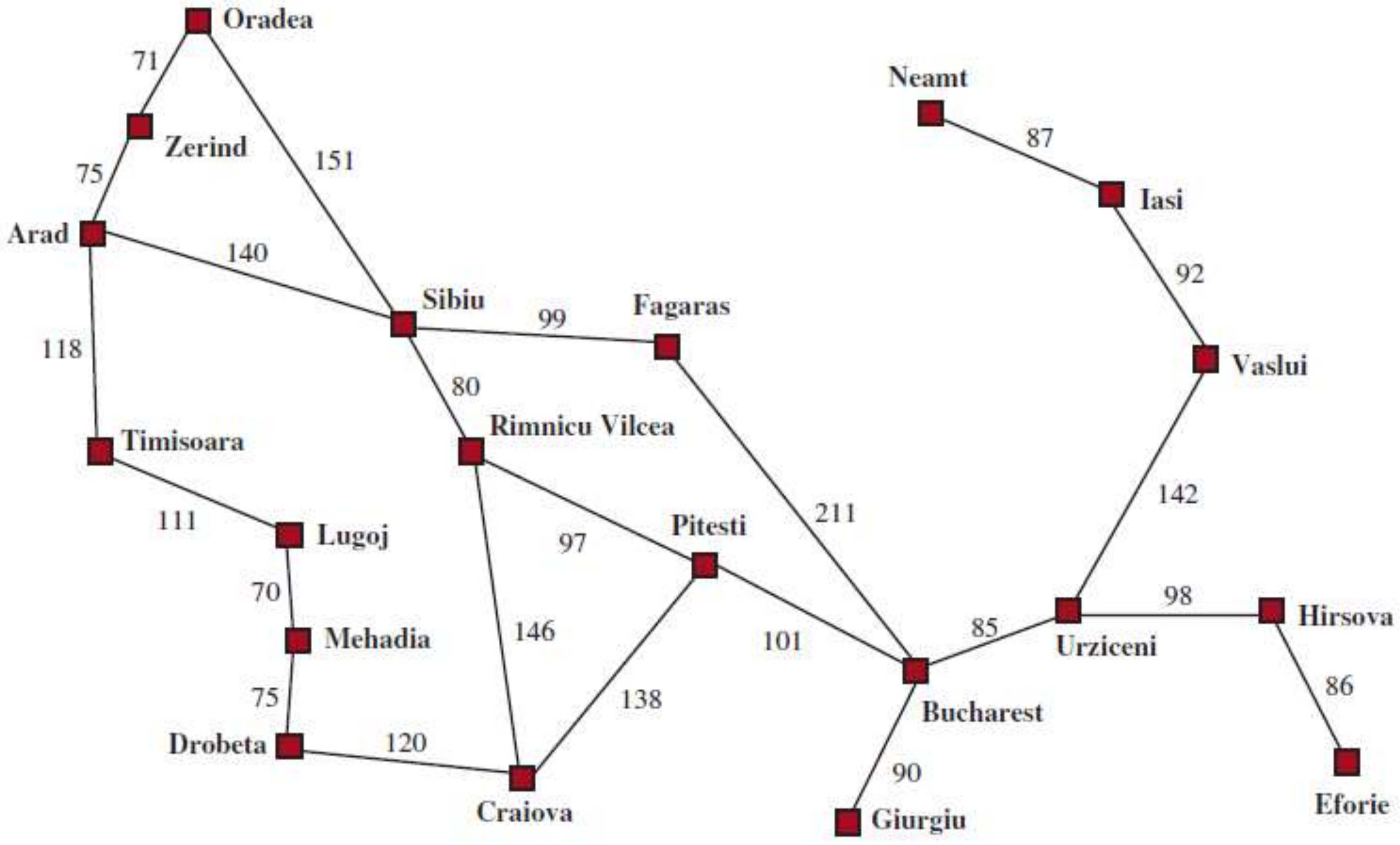


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

WELL-DEFINED PROBLEMS

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

1. The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).

2. 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. We say that each of these actions is **applicable** in s. For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

3. A description of what each action does; the formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.2 For example, we have RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

4. The **goal test**, which determines whether a given state is a goal state.

5. A **path cost** function that assigns a numeric cost to each path.

Vacuum World Problem

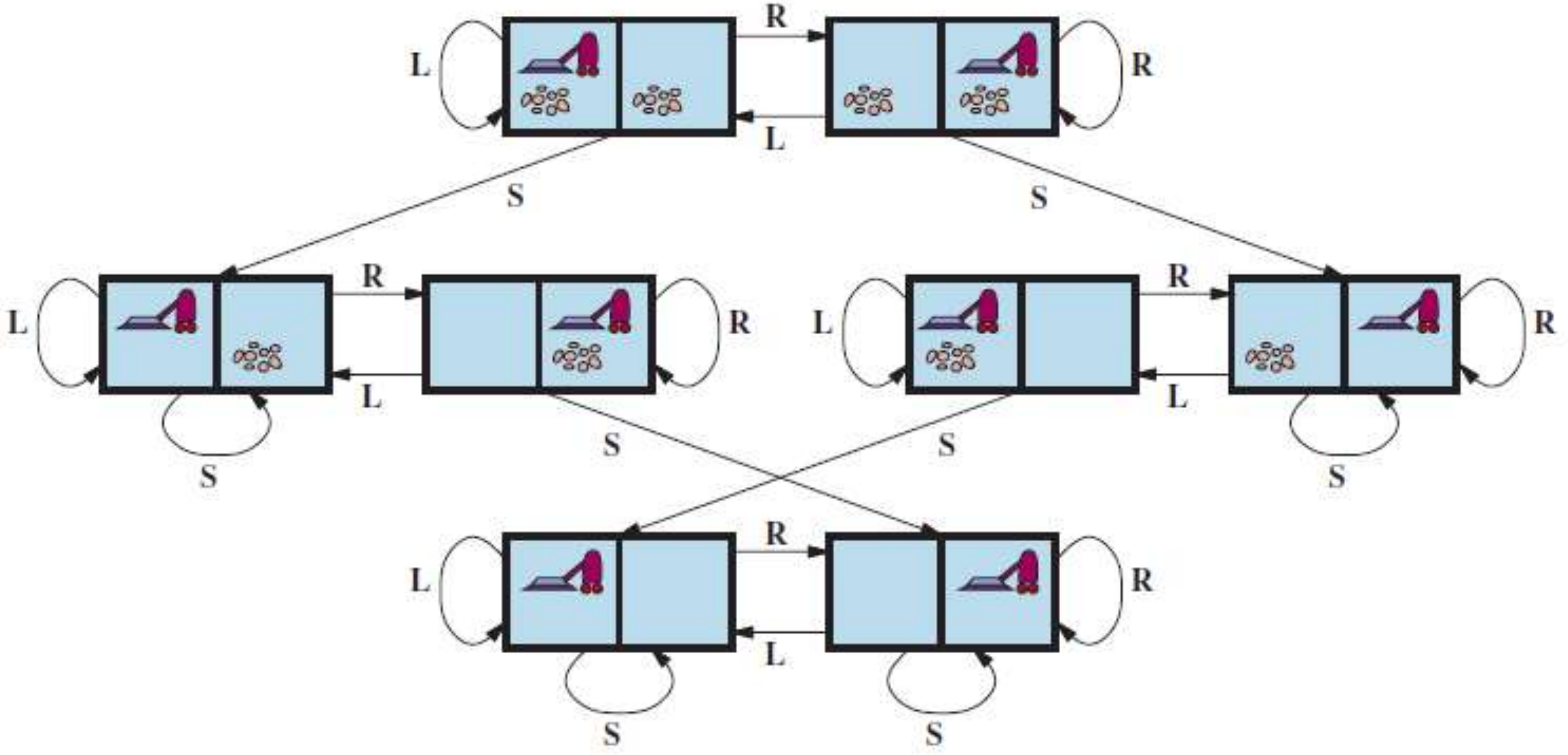
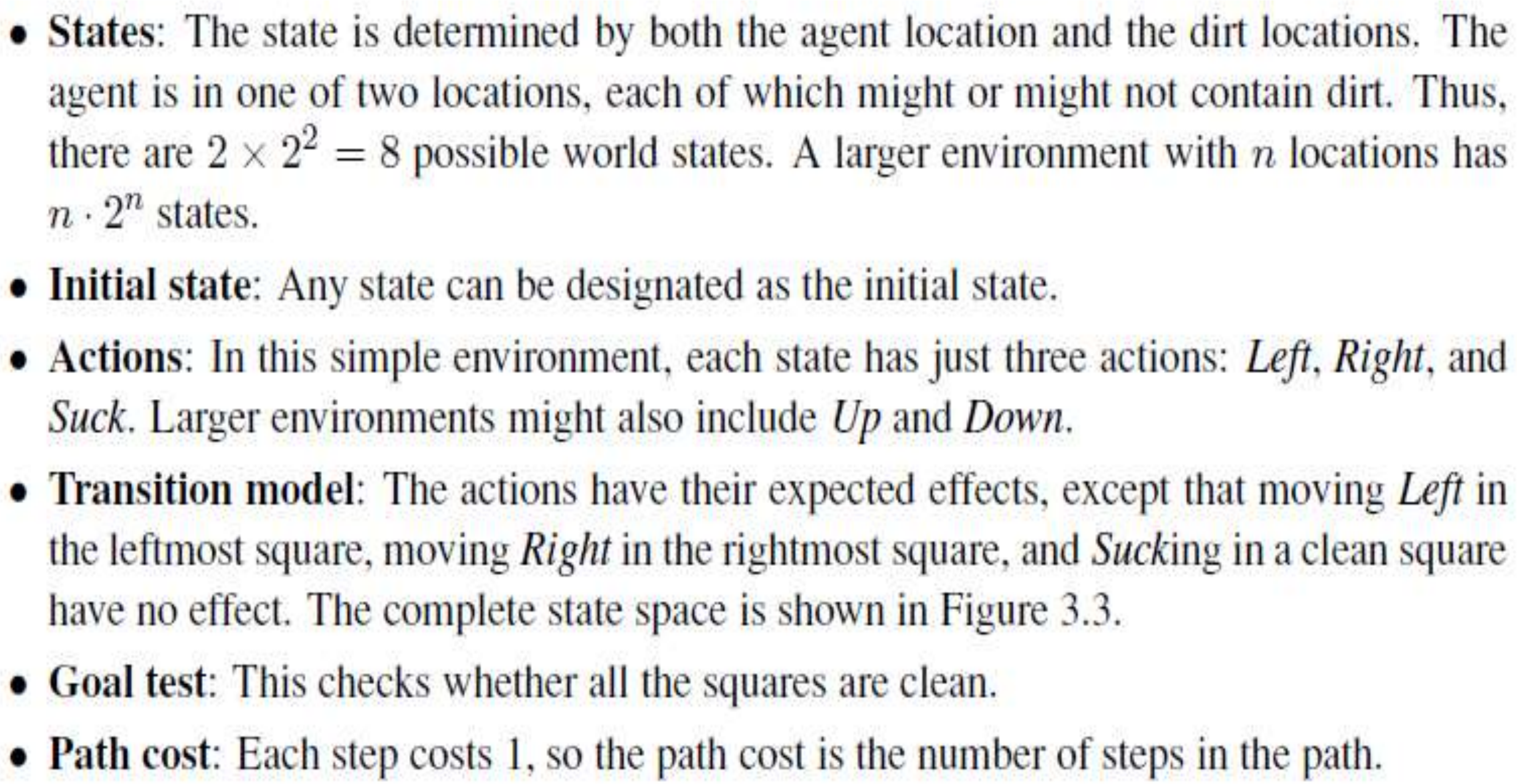


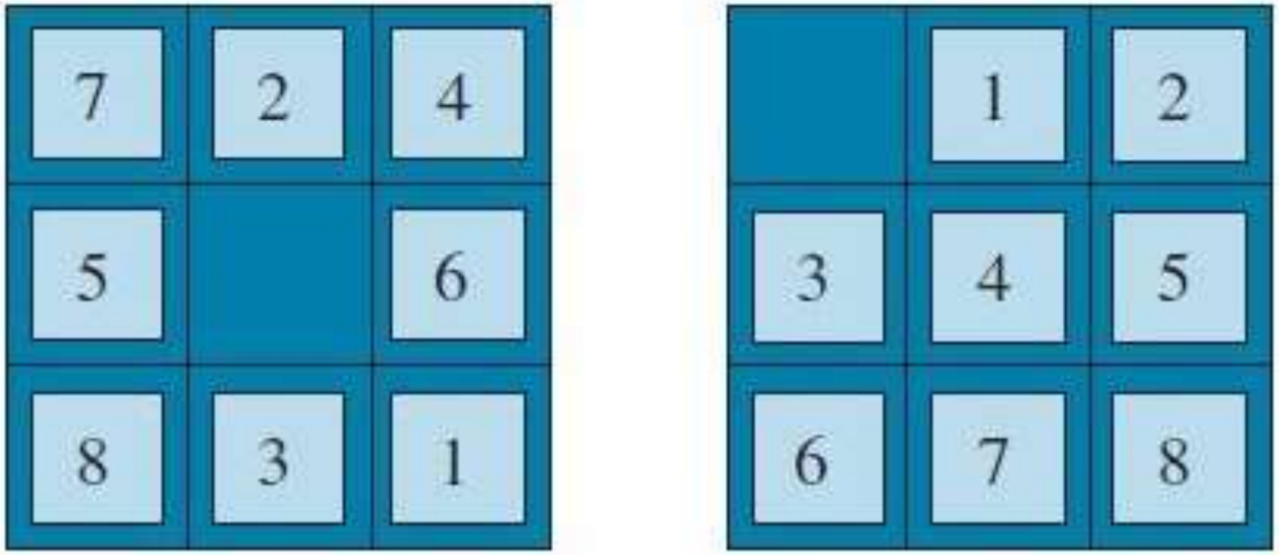
Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

VACUUM WORLD: PROBLEM FORMULATION



- **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. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **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 complete state space is shown in Figure 3.3.
- **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.

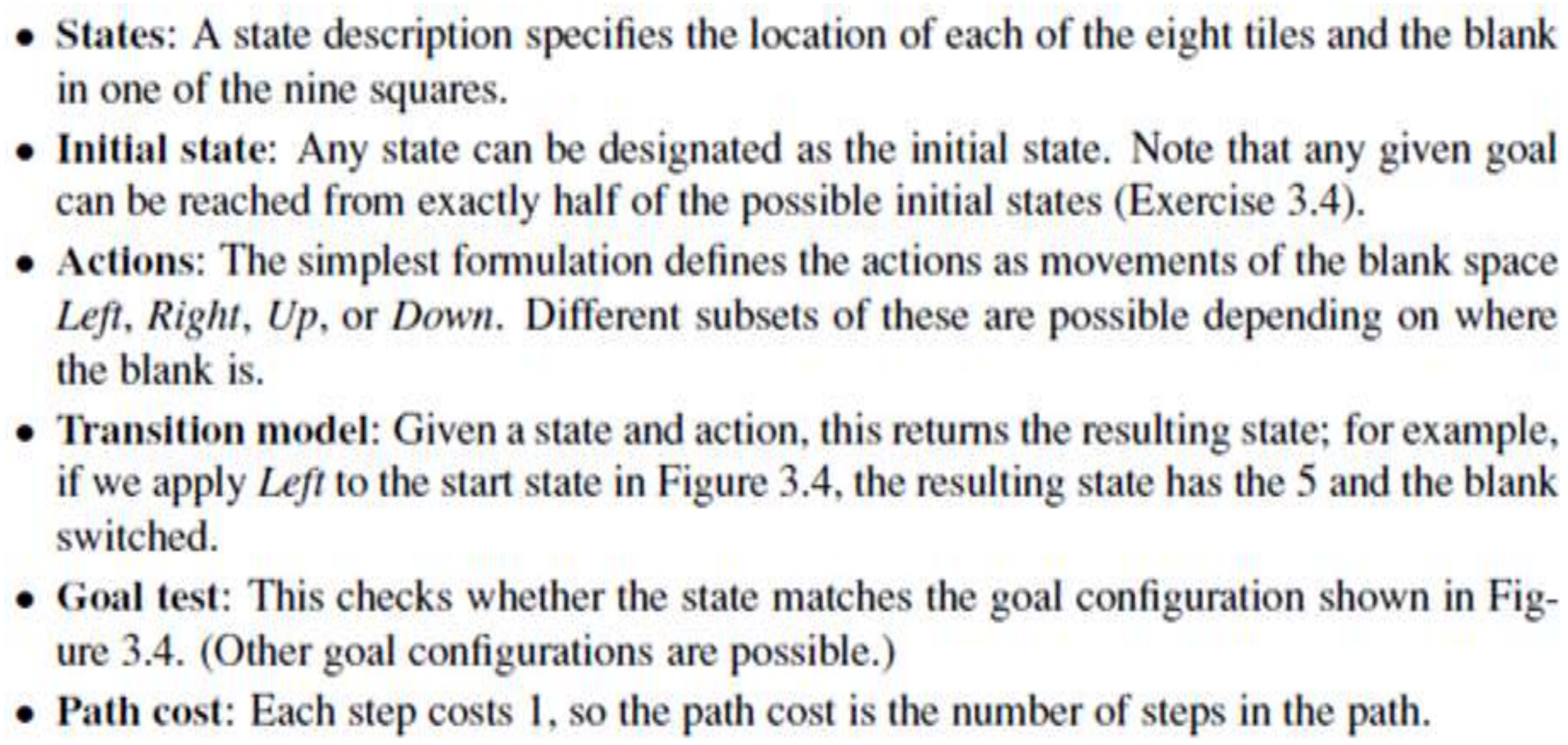
8-PUZZLE: PROBLEM FORMULATION



Start State Goal State

Figure 3.3 A typical instance of the 8-puzzle.

8-PUZZLE: PROBLEM FORMULATION



- **States**: A state description 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 any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **Actions**: The simplest formulation defines the actions as 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; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

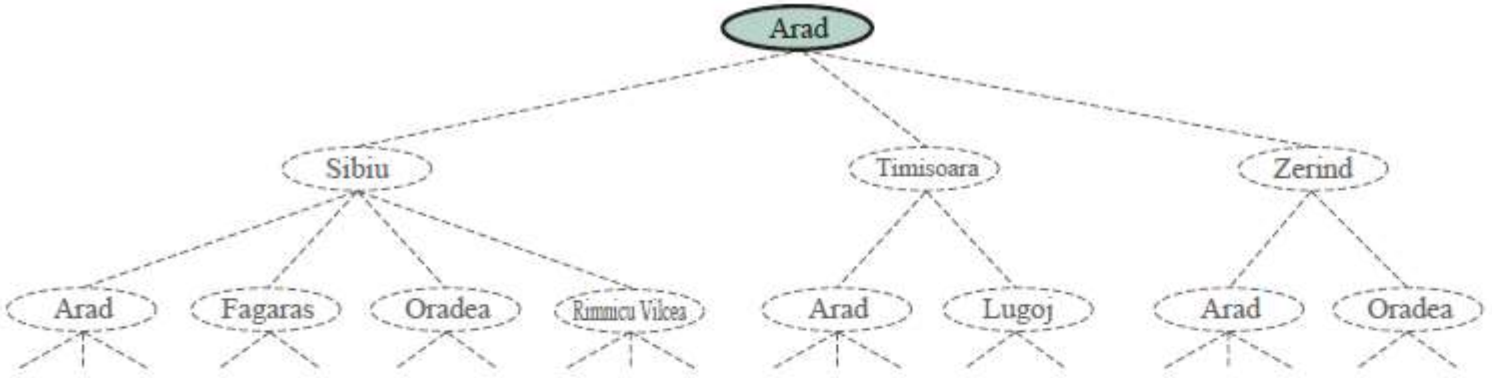
SEARCHING FOR SOLUTIONS

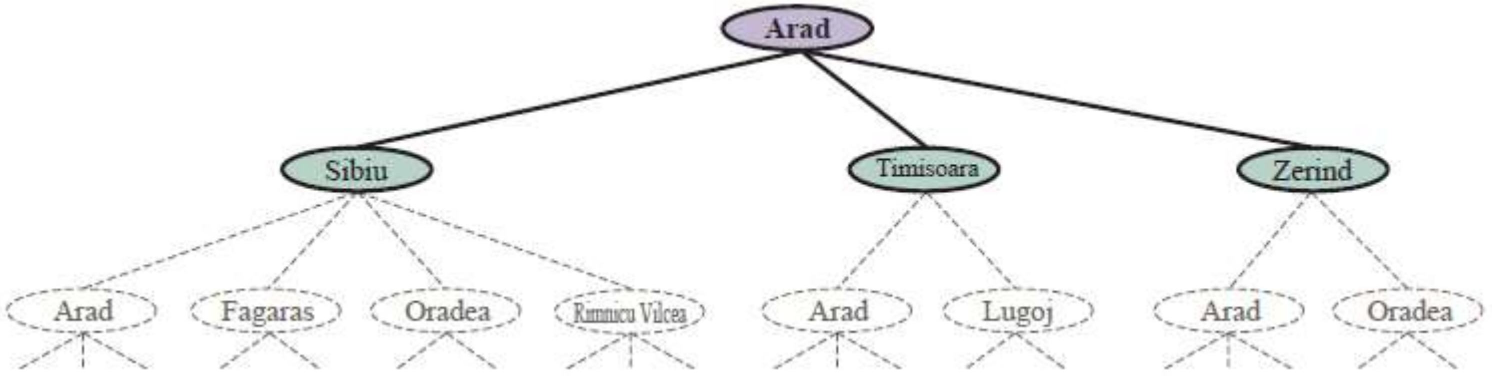
A **solution** is an action sequence, so search algorithms work by considering various possible action sequences

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, applying each legal action to the current state, thereby **generating** a new set of states

SEARCH TREE


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


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

SEARCH TREE

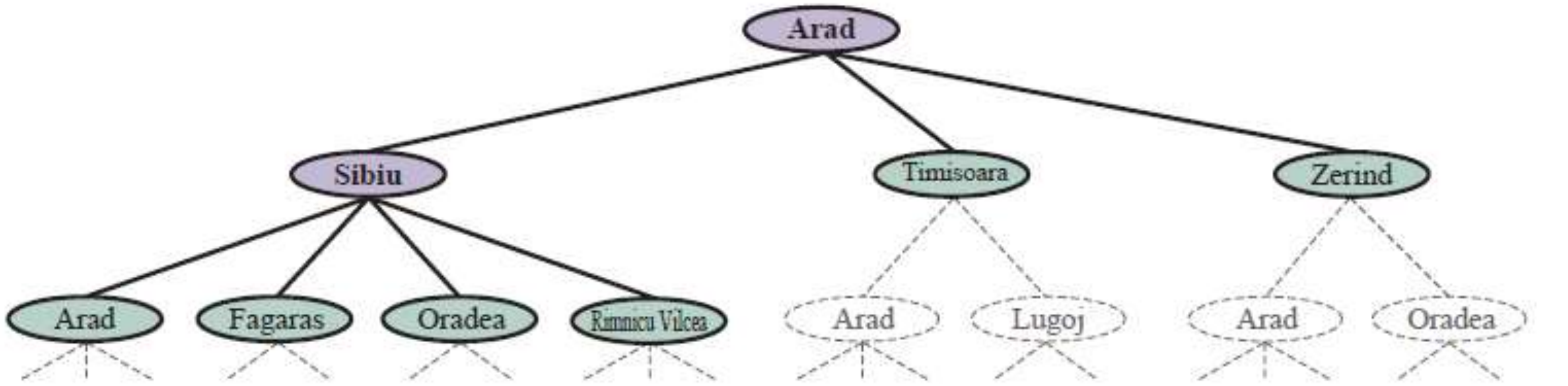


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

SEARCH STRATEGY

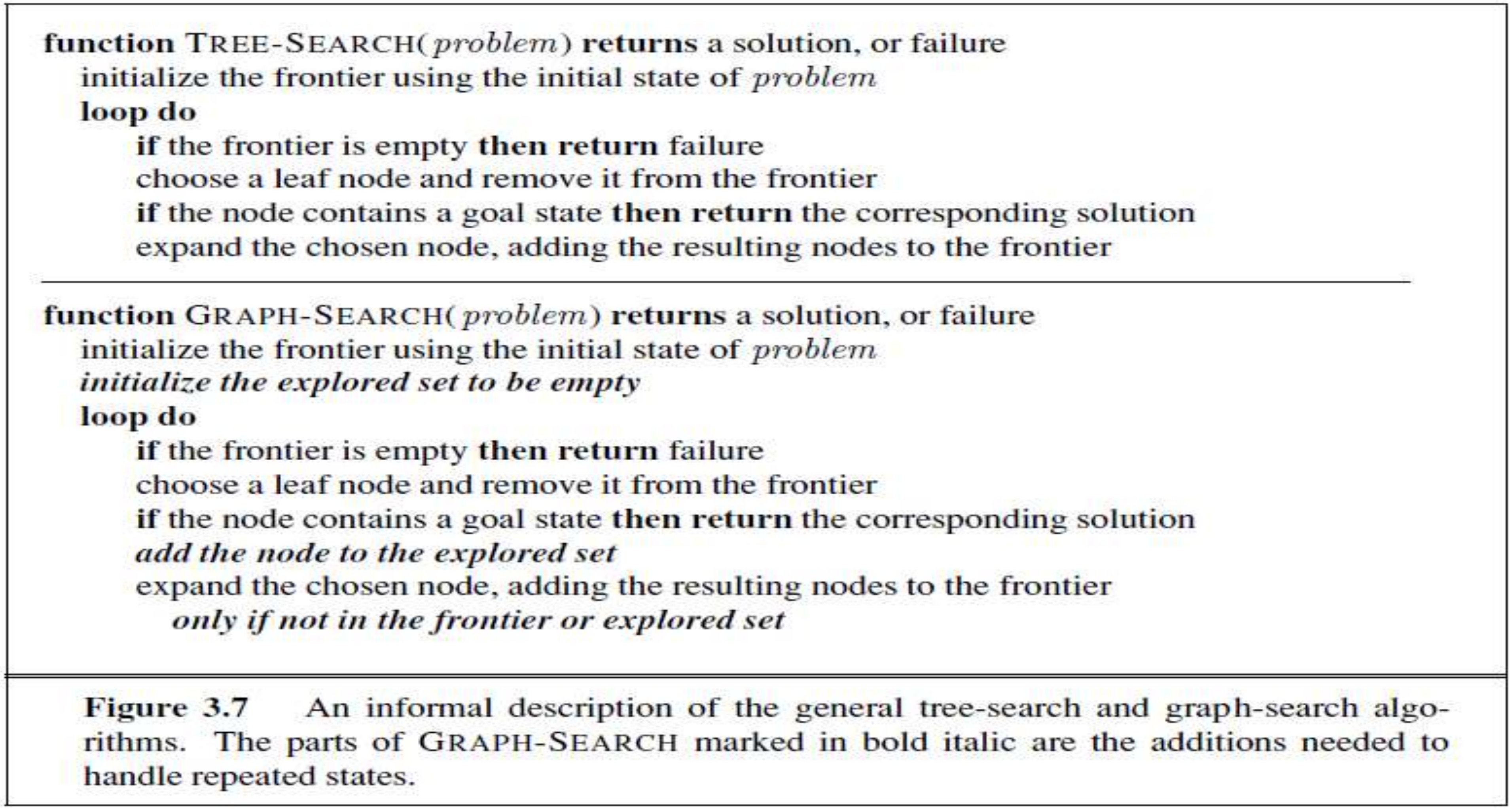
the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.

The set of all leaf nodes available for expansion at any given point is called the **frontier**.

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next —the so-called **search strategy**.

TREE SEARCH / GRAPH SEARCH



```
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
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

INFRASTRUCTURE FOR SEARCH ALGORITHMS

- 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.

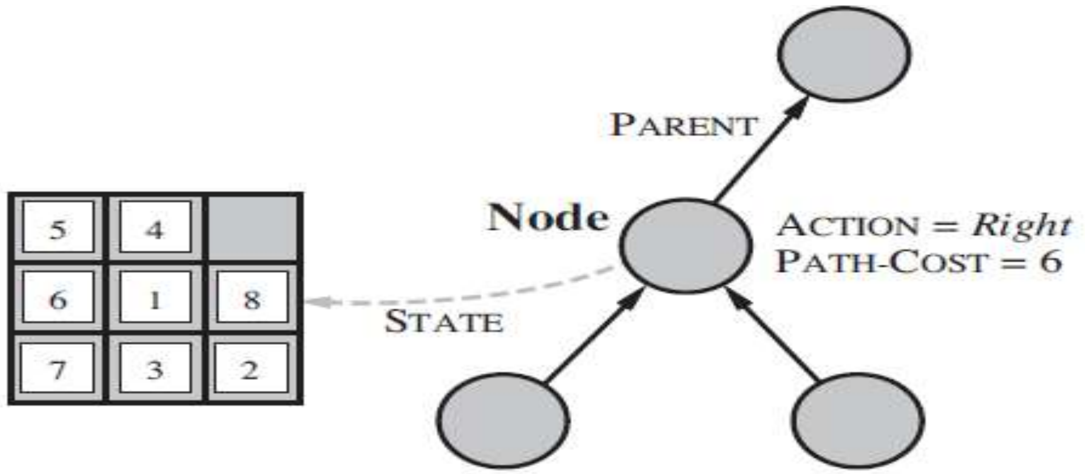


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

FRONTIER [FRINGE]

The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**.

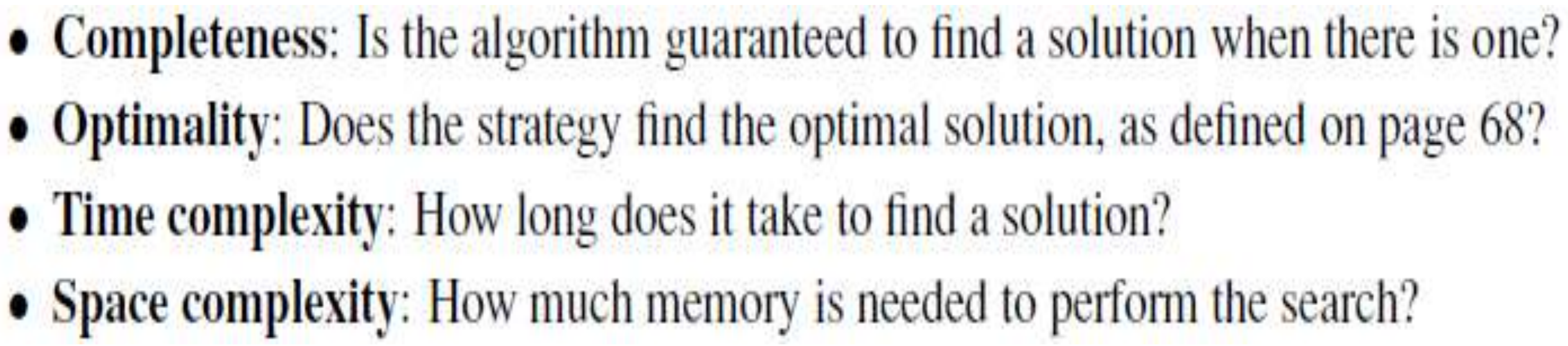
The operations on a queue are as follows:

- **EMPTY(queue)** returns true only if there are no more elements in the queue.
- **POP(queue)** removes the first element of the queue and returns it.
- **INSERT(element, queue)** inserts an element and returns the resulting queue.

Queues are characterized by the *order* in which they store the inserted nodes. Three common variants are -

1. the first-in, first-out FOr **FIFO queue**, which pops the *oldest* element of the queue;
2. the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue; and
3. the **priority queue**,

MEASURING PROBLEM-SOLVING PERFORMANCE



- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution, as defined on page 68?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

SEARCH ALGORITHMS

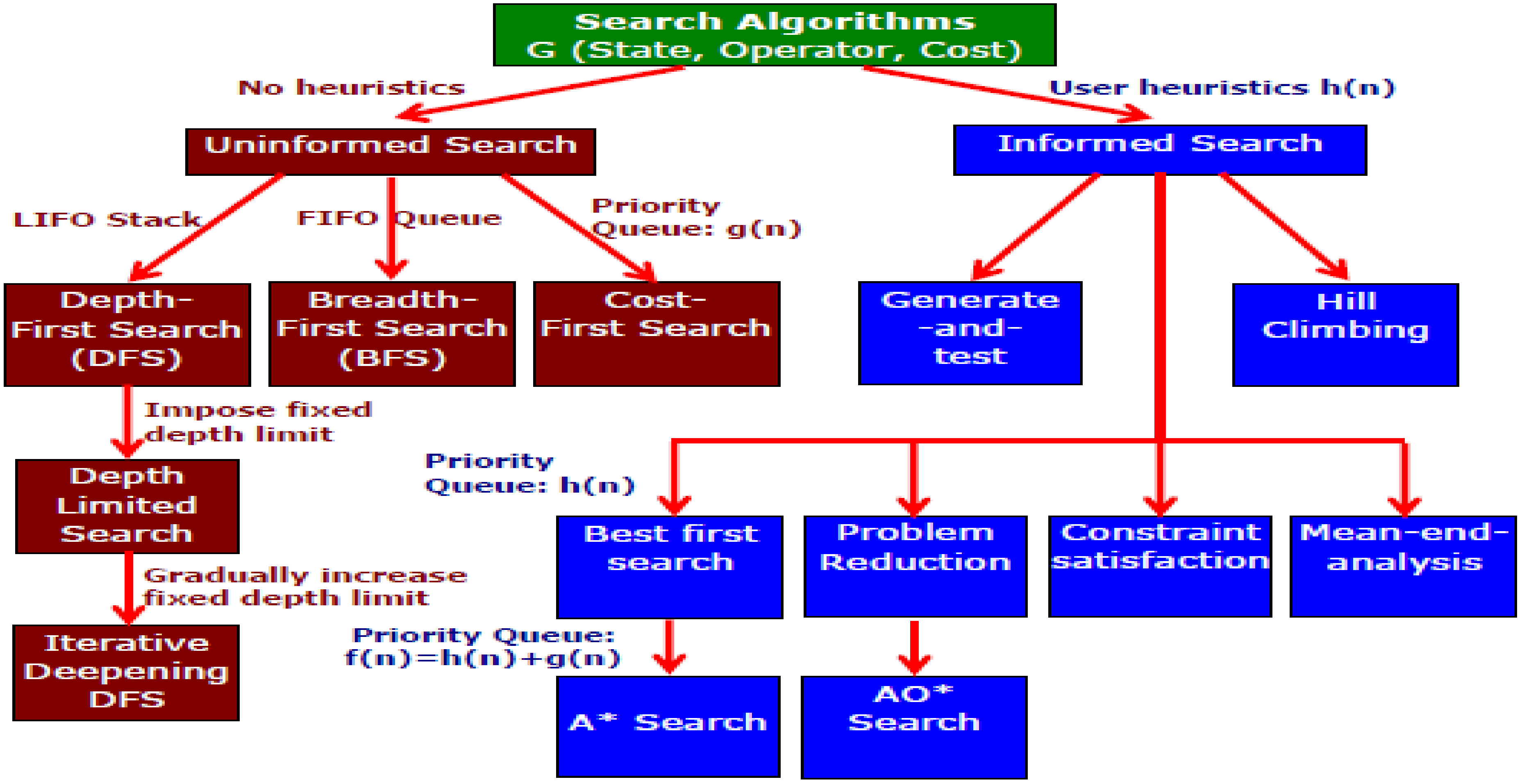
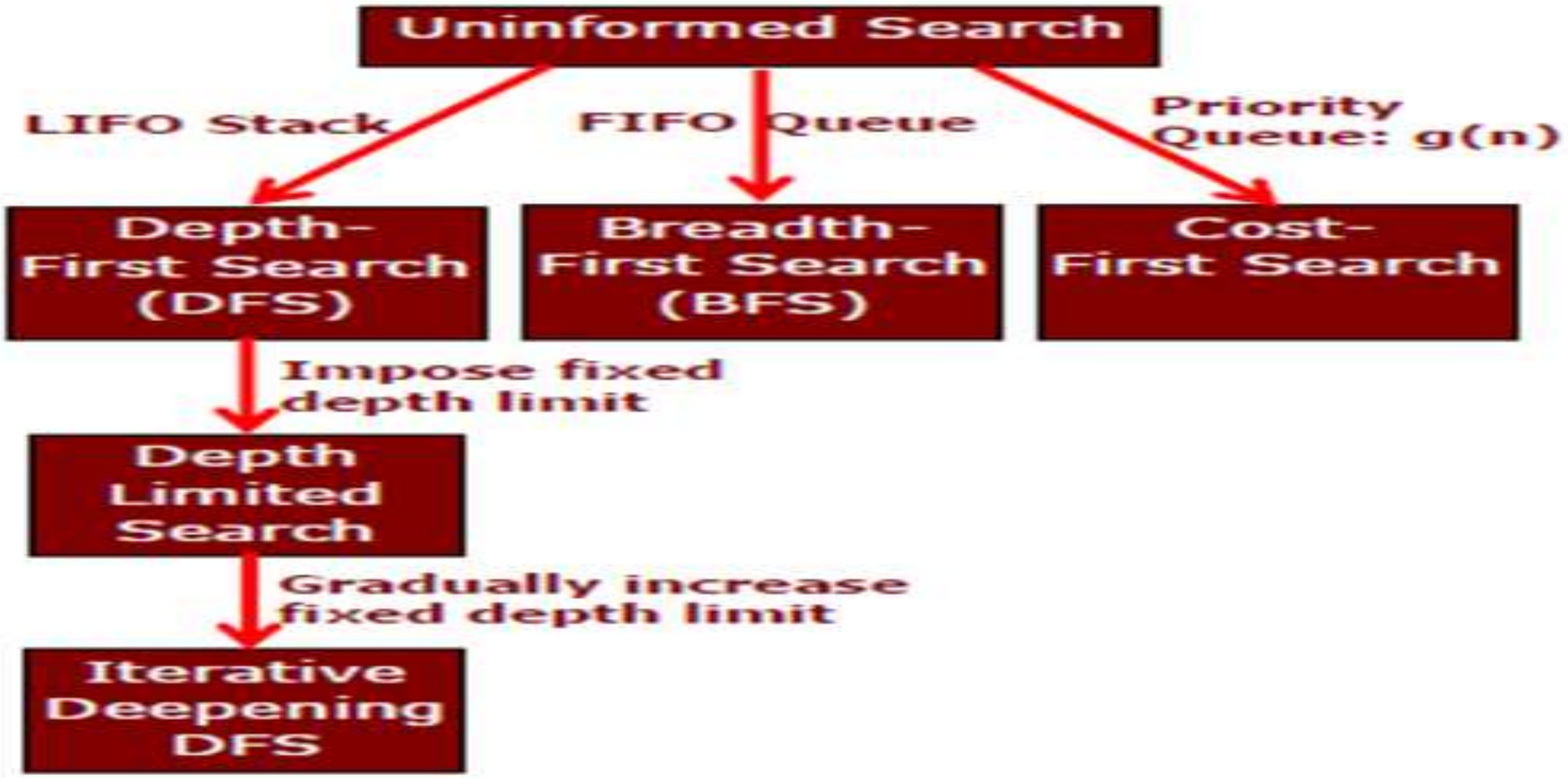


Fig. Different Search Algorithms

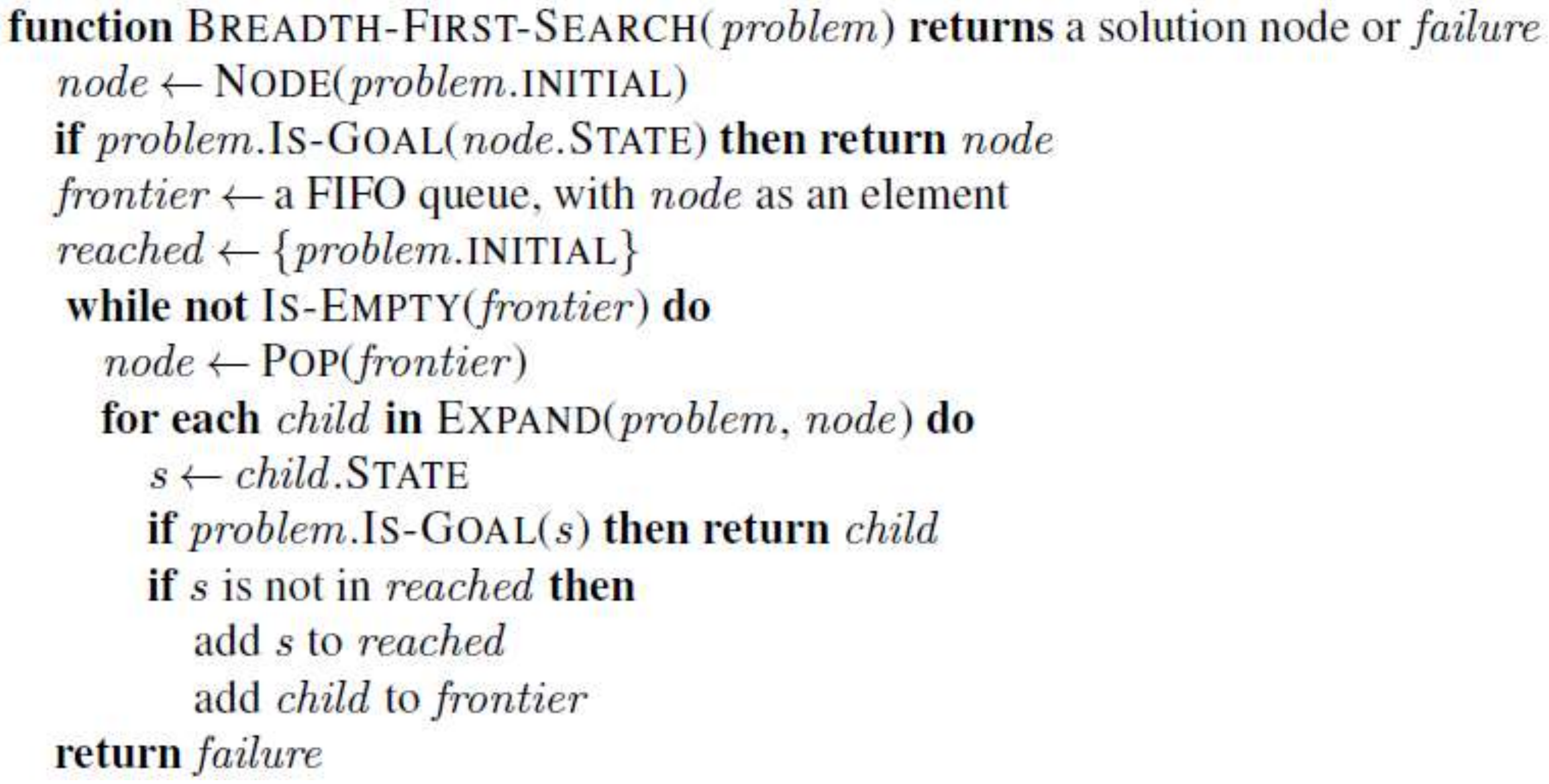
UNINFORMED SEARCH STRATEGIES



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.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breadth-first search is an instance of the general graph-search algorithm (Figure 3.7) in which the *shallowest* unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier.
- Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
- The **goal test** is applied to each node when it is ***generated*** rather than when it is **selected** for expansion.
- Following the general template for graph search, discards any new path to a state already in the frontier or explored set

BREADTH-FIRST SEARCH: PSEUDOCODE



```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then return node
    frontier ← a FIFO queue, with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if problem.IS-GOAL(s) then return child
            if s is not in reached then
                add s to reached
                add child to frontier
    return failure
```

BREADTH-FIRST SEARCH: FOUR CRITERIA

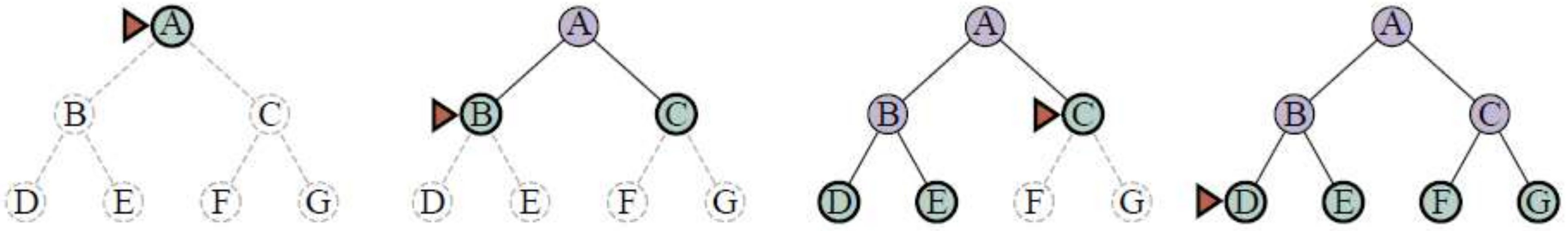
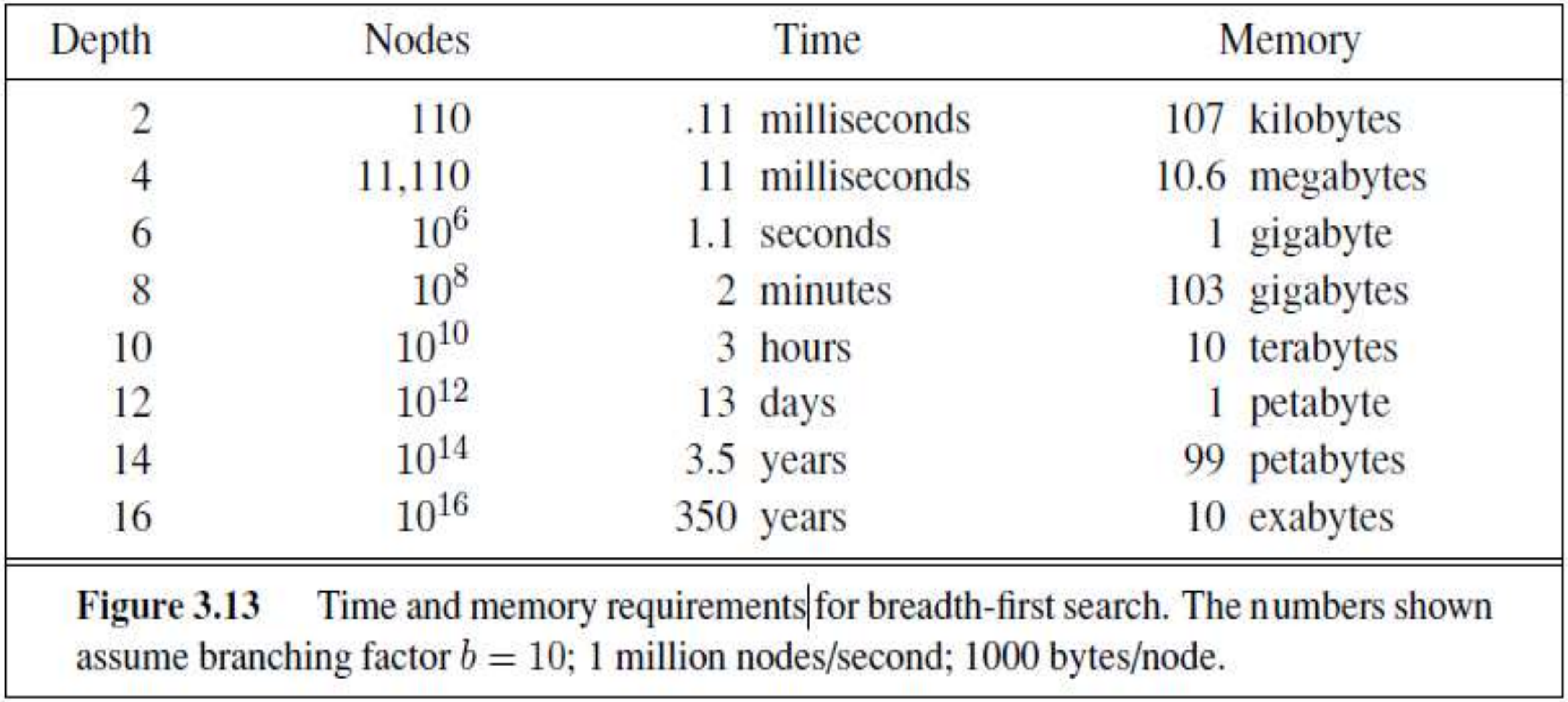


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

- ***complete***—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes.
- Breadth-first search is optimal if the path cost is a non-decreasing function of the depth of the node.
- The time complexity would be ***O(b^{d+1}),*** The space complexity is ***O(b^{d})***
- For uniform tree, the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$

BREADTH-FIRST SEARCH: MEMORY REQUIREMENTS



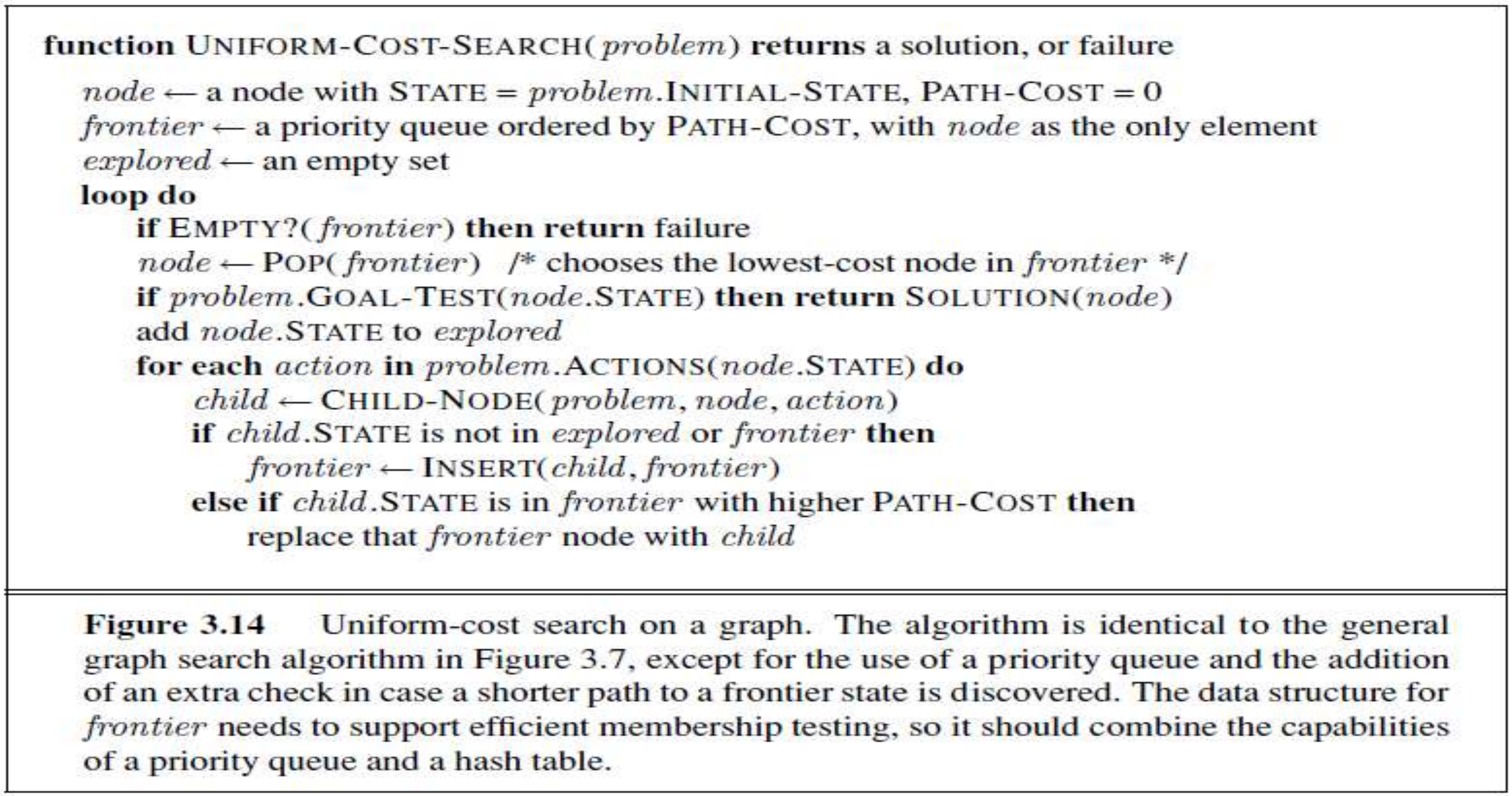
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

UNIFORM-COST SEARCH

- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* **g(n).**
- This is done by **storing** the **frontier as a priority queue** ordered by **g**
- Two other significant differences from breadth-first search are
  1. The **goal test** is applied to a node when it is ***selected for expansion*** rather than when it is first **generated**
  2. A test is added in case a better path is found to a node currently on the frontier.

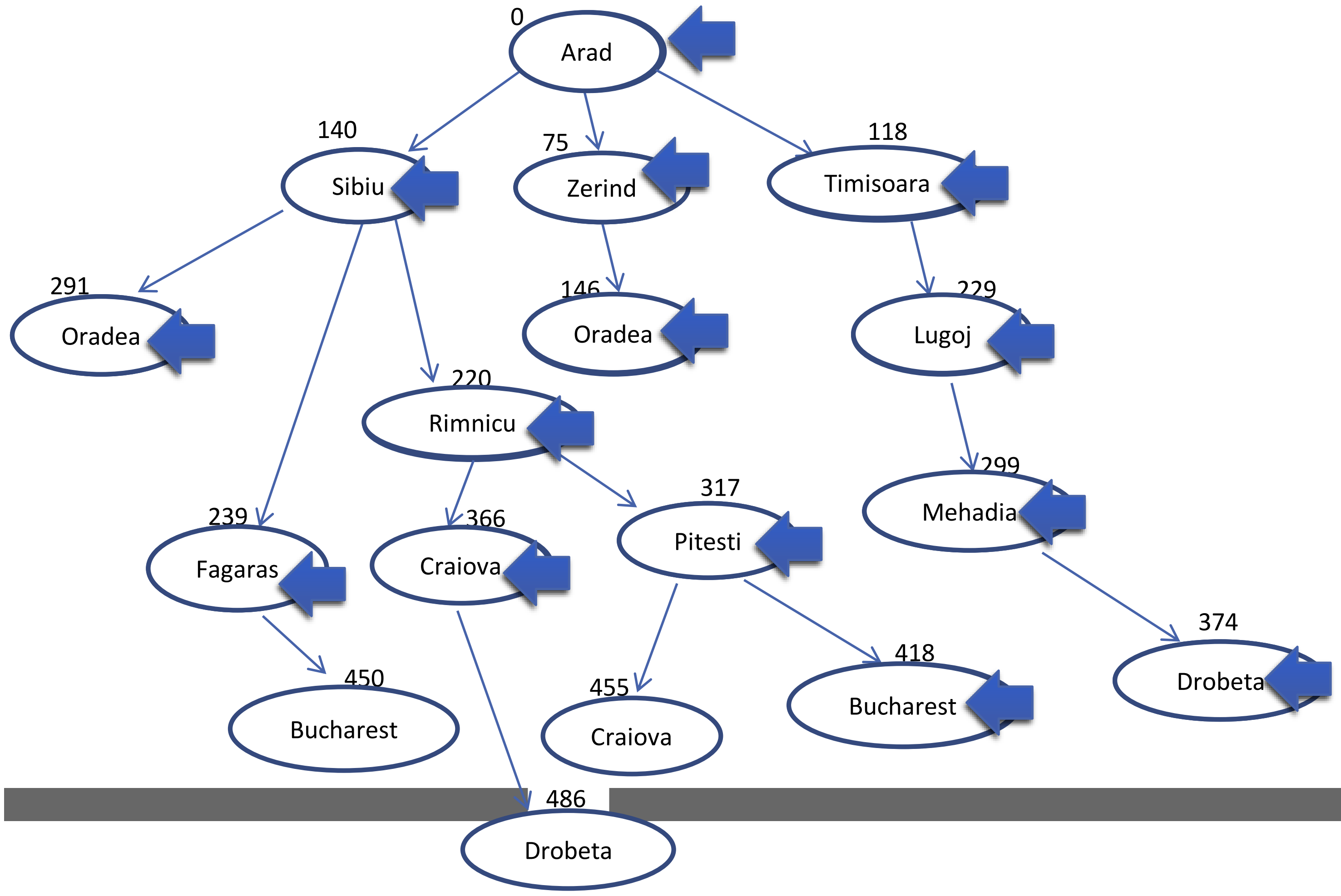
UNIFORM-COST SEARCH: PSEUDOCODE



```
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
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

UNIFORM-COST SEARCH: Arad → Bucharest



UNIFORM-COST SEARCH: ILLUSTRATION

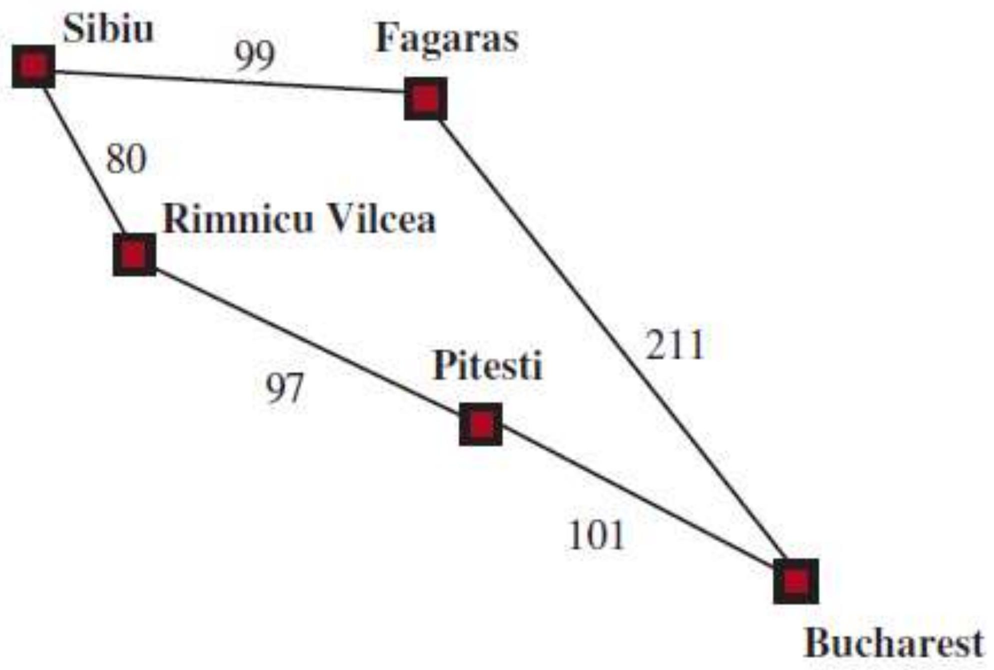


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

UNIFORM-COST SEARCH: OPTIMALITY

- Observe that whenever uniform-cost search selects a **node n** for expansion, the optimal path to that node has been found.
    - Were this not the case, there would have to be another frontier **node n'** on the optimal path from the **start node to n, n'** would have lower g-cost than n and would have been selected first

- Then, because step costs are nonnegative, paths never get shorter as nodes are added

These two facts together imply that

uniform-cost search expands nodes in order of their optimal path cost.

Hence, the first goal node selected for expansion must be the optimal solution.

UNIFORM-COST SEARCH: COMPLETENESS & COMPLEXITY

- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε.
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.
- Instead, let C* be the cost of the optimal solution, and assume that every action costs at least ε .
- Then the algorithm's worst-case time and space complexity is $\mathbf{O(b^{1+[C^*/\varepsilon]})}$, which can be much greater than b^d.
- When all step costs are equal, $\mathbf{b}^{1+[C*/\varepsilon]}$ is just $\mathbf{b^{d+1}}$.

DEPTH-FIRST SEARCH

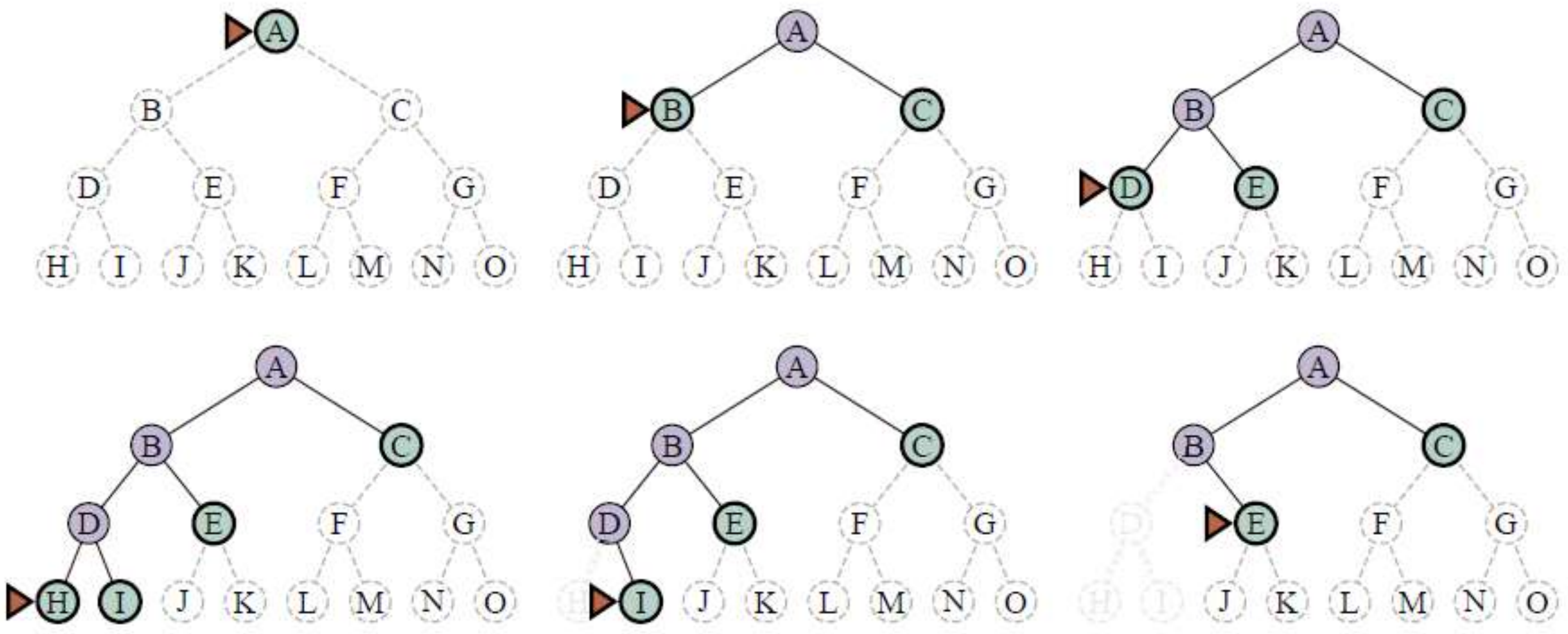
Depth-first search always expands the *deepest* node in the current frontier of the search tree.

The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors

As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm, where it uses a LIFO queue

DEPTH-FIRST SEARCH: SEARCH TREE


A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O

DEPTH-FIRST SEARCH: SEARCH TREE

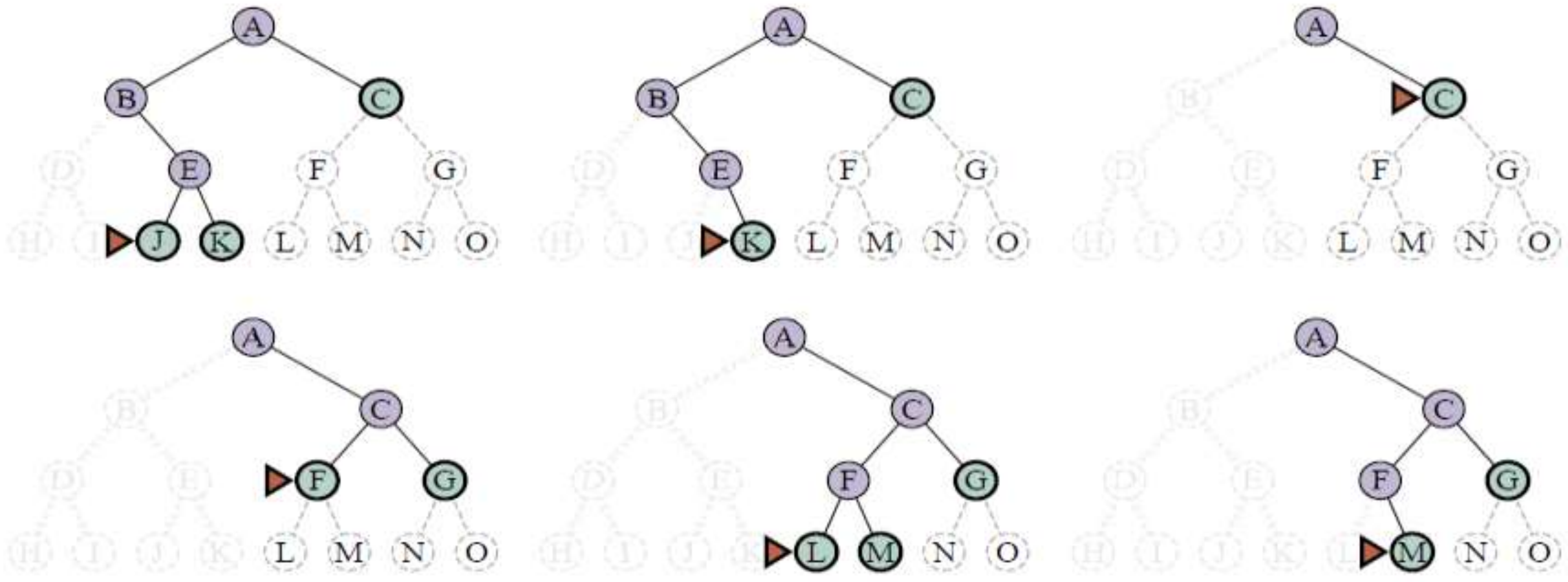

A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O

Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

DEPTH-FIRST SEARCH:
Optimality, Complexity, Completeness

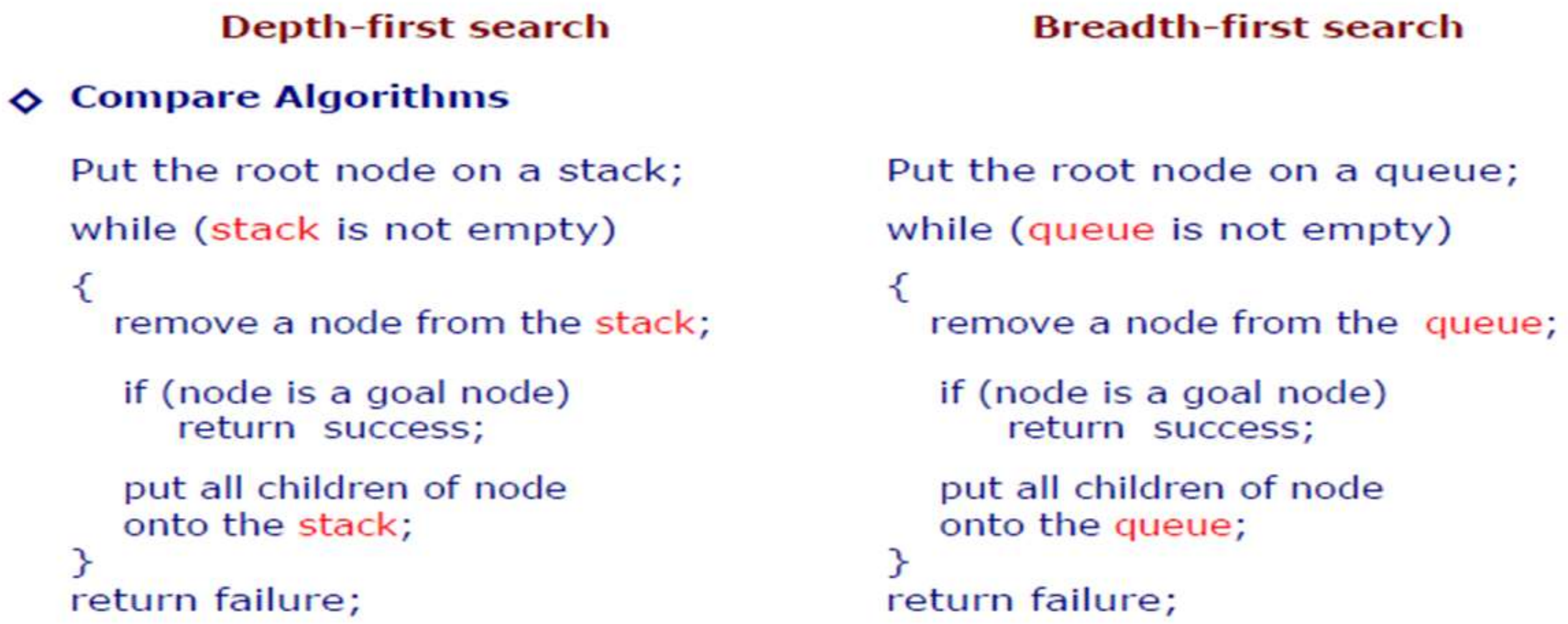
The graph-search version is **complete** in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is ***not* complete**

Both versions are **nonoptimal**. For example, in Figure 3.16, depth first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.

A depth-first tree search, may generate all of the **$O(b^m)$** nodes in the search tree, where **m** is the maximum depth of any node; this can be much greater than the size of the state space.

For a state **space** with branching factor b and maximum depth m, depth-first search requires storage of only **O(bm)** nodes.

DFS v/s BFS



Depth-first search

◇ **Compare Algorithms**

```
Put the root node on a stack;
while (stack is not empty)
{
  remove a node from the stack;

   if (node is a goal node)
      return  success;

   put all children of node
   onto the stack;
}
return failure;
```

Breadth-first search

```
Put the root node on a queue;
while (queue is not empty)
{
  remove a node from the  queue;

   if (node is a goal node)
      return  success;

   put all children of node
   onto the queue;
}
return failure;
```

DFS v/s BFS

- **Compare Features**

  - ‡ When succeeds, the goal node found is not necessarily minimum depth
  - ‡ Large tree, may take excessive long time to find even a nearby goal node

  - ‡ When succeeds, it finds a minimum-depth (nearest to root) goal node
  - ‡ Large tree, may require excessive memory

- **How to over come limitations of DFS and BFS ?**

  - ‡ Requires, how to combine the advantages and avoid disadvantages?
  - ‡ The answer is "*Depth-limited search*" .

    This means, perform Depth-first searches with a depth limit.

DEPTH-LIMITED SEARCH

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit .

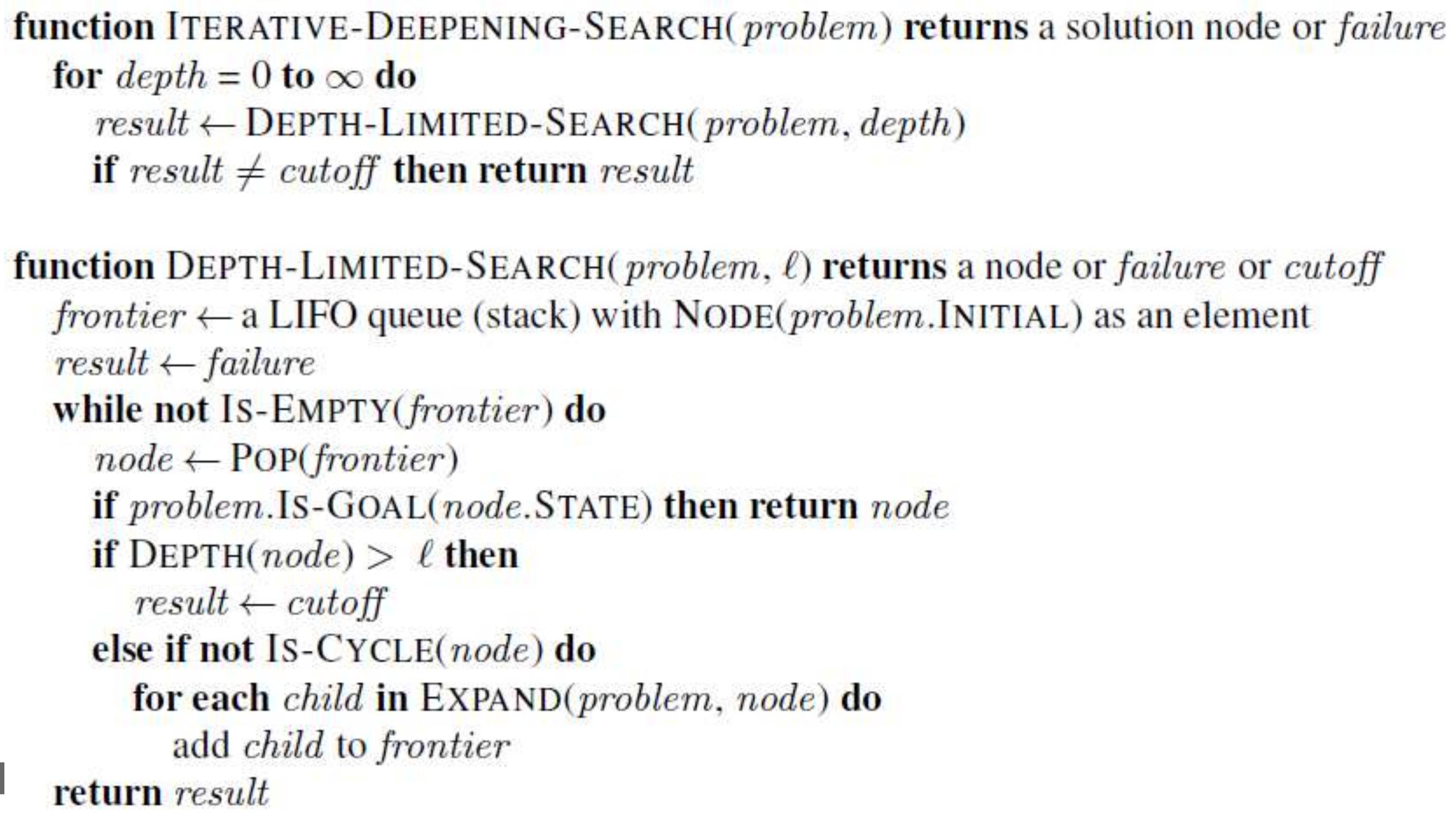
- That is, nodes at depth are treated as if they have no successors.

- This approach is called **depth-limited search**

- Depth-limited search will also be nonoptimal if we choose **ℓ> d**. Its time complexity is **O(b^{ℓ})** and its space complexity is **O(bℓ).**

- Depth-first search can be viewed as a special case of depth-limited search with $\ell=\infty$.

DEPTH-LIMITED SEARCH: PSEUDOCODE



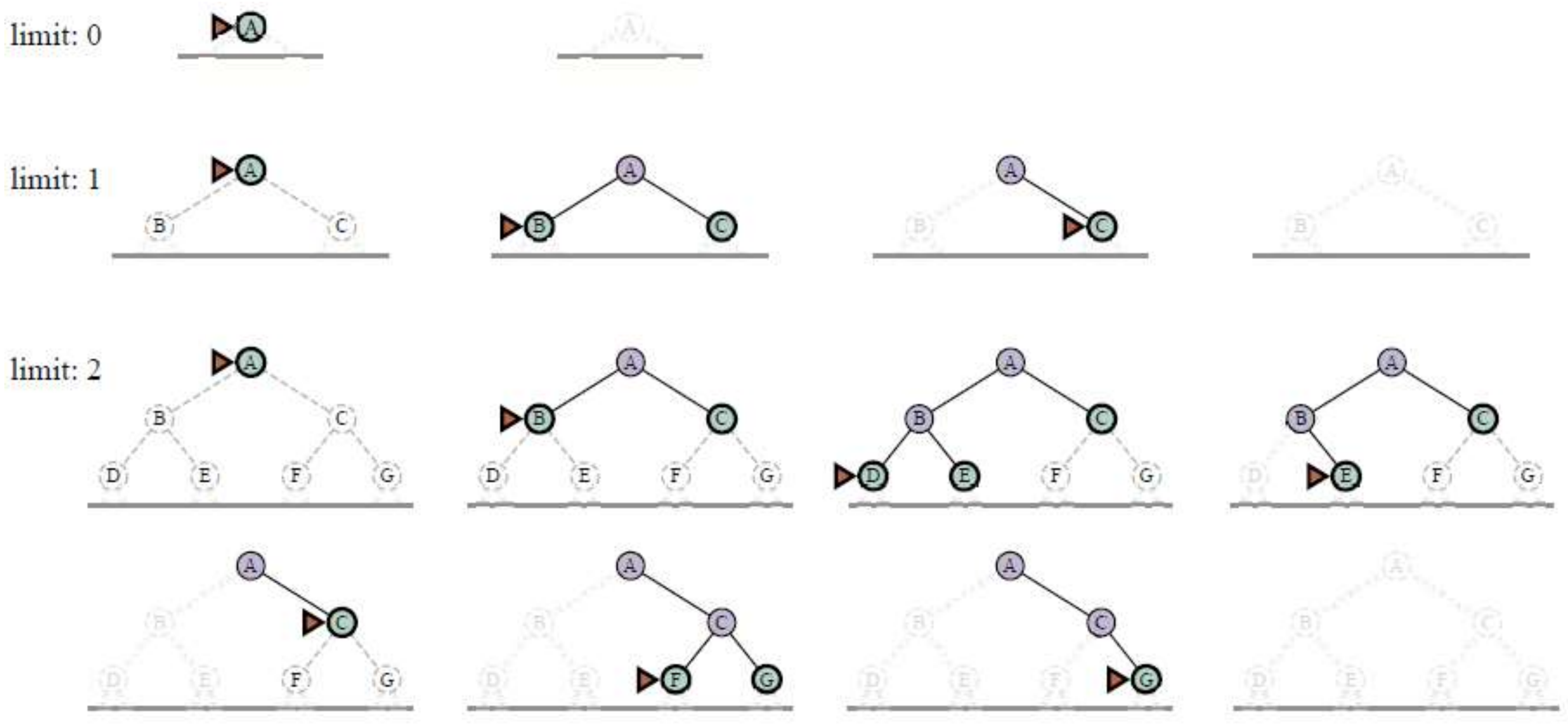
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to $\infty$ do
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
    if result $\neq$ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, $\ell$) returns a node or failure or cutoff
  frontier $\leftarrow$ a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result $\leftarrow$ failure
  while not IS-EMPTY(frontier) do
    node $\leftarrow$ POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) $>$ $\ell$ then
      result $\leftarrow$ cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

ITERATIVE DEEPENING DEPTH-FIRST SEARCH

- **Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found
- Iterative deepening combines the benefits of depth-first and breadth-first search.

1. Like depth-first search, its **memory requirements are modest: O(bd)** to be precise.
2. Like breadth-first search, it is complete when the branching factor is finite **and optimal** when the path cost is a nondecreasing function of the depth of the node

ITERATIVE DEEPENING DEPTH-FIRST SEARCH TREE



ITERATIVE DEEPENING DEPTH-FIRST SEARCH TREE

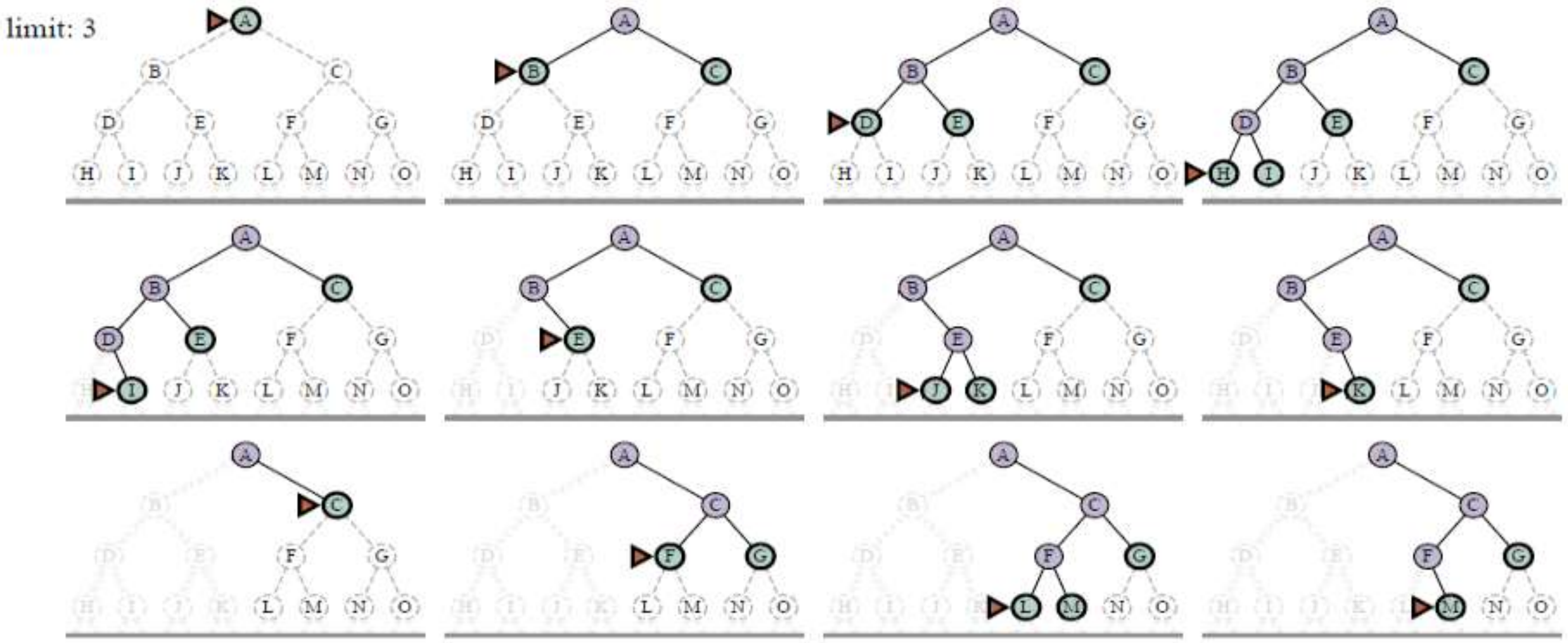


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

IDP: COMPLEXITY

- Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly.

- The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

- In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times.

- So the total number of nodes generated in the worst case is

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

BIDIRECTIONAL SEARCH

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect;

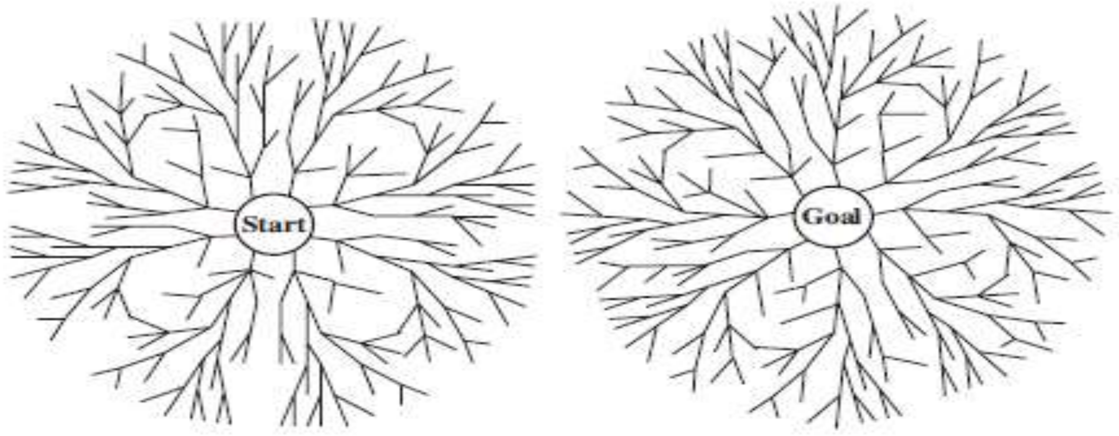
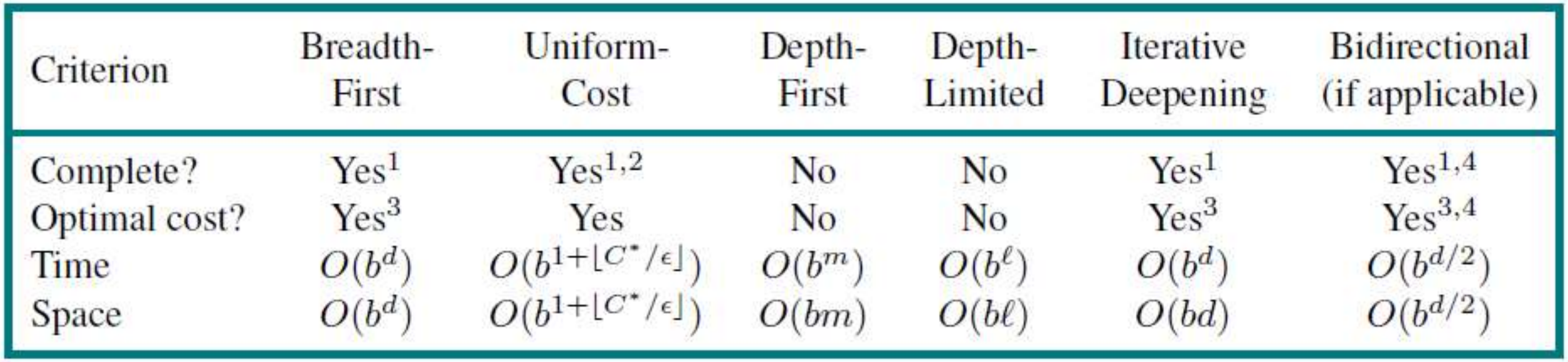


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

COMPARING UNINFORMED SEARCH STRATEGIES



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

References

1. Chapter 3: Solving Problem by Searching , Pages 64-91
"Artificial Intelligence: A Modern Approach," by Stuart J. Russell and Peter Norvig,

Books

1. “Artificial Intelligence: A Modern Approach,” by Stuart J. Russell and Peter Norvig.
2. "Artificial Intelligence: Structures and Strategies for Complex Problem Solving", by George F. Luger, (2002)
3. "Artificial Intelligence: Theory and Practice", by Thomas Dean.
4. "AI: A New Synthesis", by Nils J. Nilsson.
5. “Programming for machine learning,” by J. Ross Quinlan,
6. “Neural Computing Theory and Practice,” by Philip D. Wasserman, .
7. “Neural Network Design,” by Martin T. Hagan, Howard B. Demuth, Mark H. Beale, .
8. “Practical Genetic Algorithms,” by Randy L. Haupt and Sue Ellen Haupt.
9. “Genetic Algorithms in Search, optimization and Machine learning,” by David E. Goldberg.
10. "Computational Intelligence: A Logical Approach", by David Poole, Alan Mackworth, and Randy Goebel.
11. “Introduction to Turbo Prolog”, by Carl Townsend.