Search Algorithms

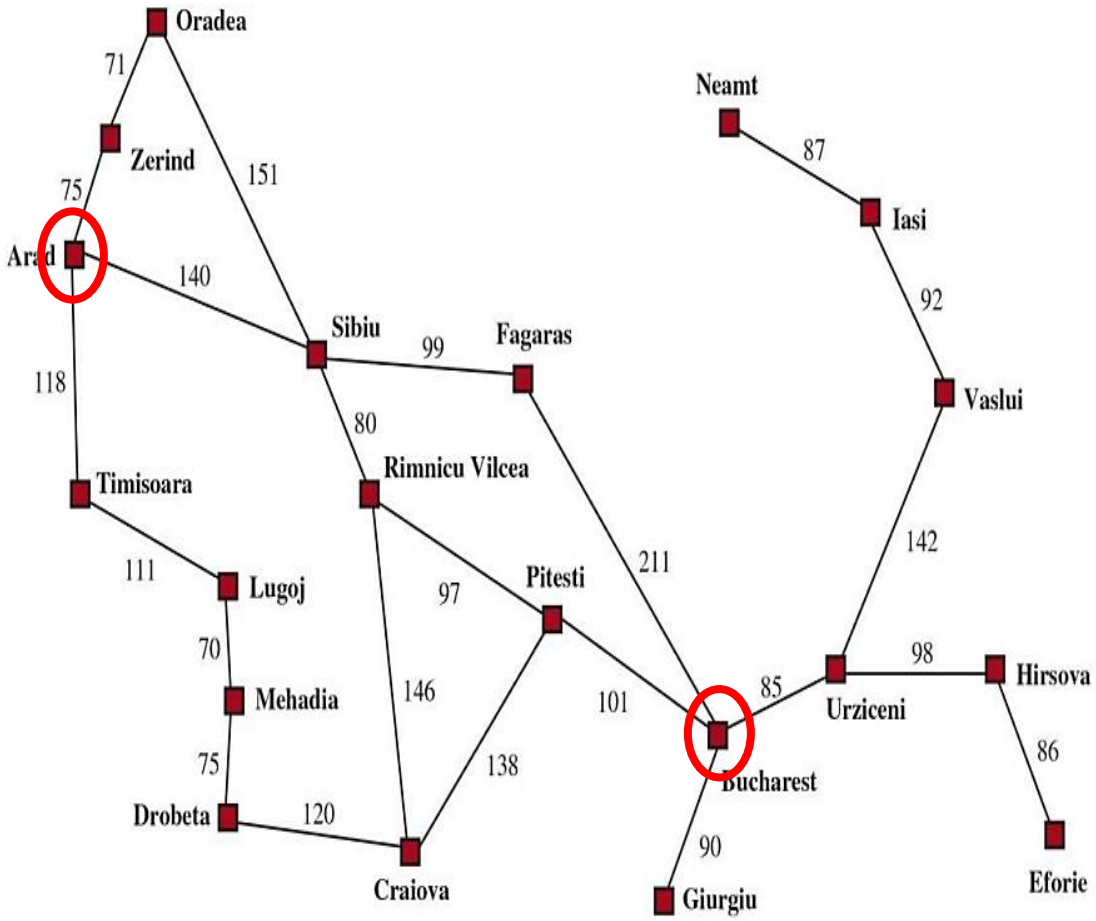
Prof. G. Rama Mohan Babu
Professor & HoD, CSE(AI&ML)
RVR&JCCE, Guntur - 19

3.3. Search for Solutions

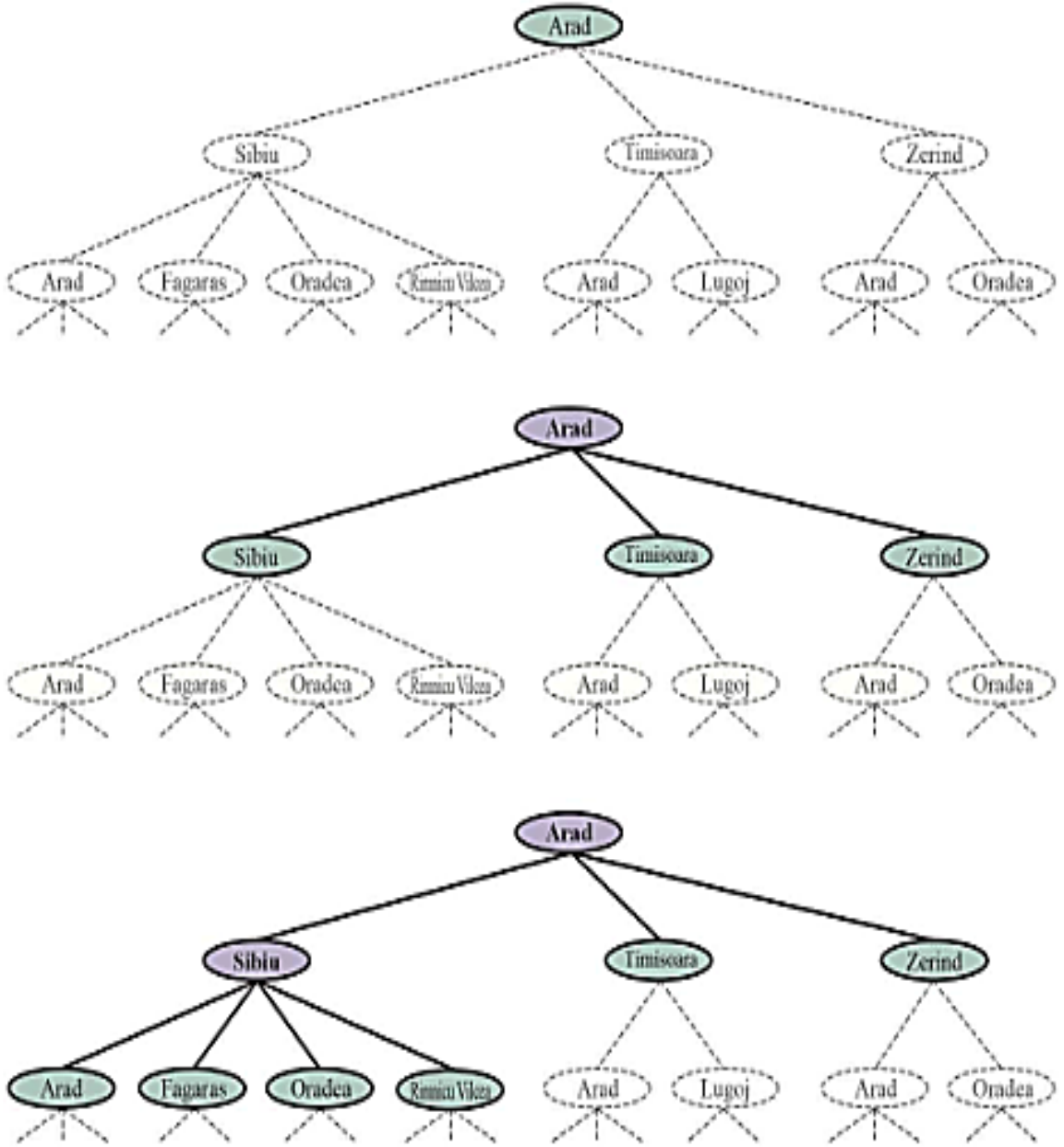
- A solution to the search problem is an action sequence.
- The possible action sequences starting at the *initial state* form a *search tree* with the initial state at the root; the branches are actions and the *nodes* correspond to states in the *state space* of the problem.
- The *state space* describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.
- *Expanding* the current state is by applying each legal action to the current state, thereby generating a new set of states.
  - Example: the parent node In(Arad) leading to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind).
- A *leaf node* 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*. (open list)

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- The basic structure of all search algorithms like TREE-SEARCH algorithm, but they differ in their *search strategy*, which determines the order in which states are expanded.
- *Repeated states* in search algorithms refer to the situation where the same state is encountered multiple times during the search process. This can lead to inefficiencies and redundant work if not handled properly.
- A *loopy path* in search algorithms refers to a path that revisits the same state multiple times, creating a loop. This can lead to infinite loops or redundant explorations, making the search process inefficient.
- Loopy paths are a special case of the more general concept of *redundant paths*, which exist whenever there is more than one way to get from one state to another.
  - Example: Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).

The Romania Problem State Space Tree



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



```
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
```

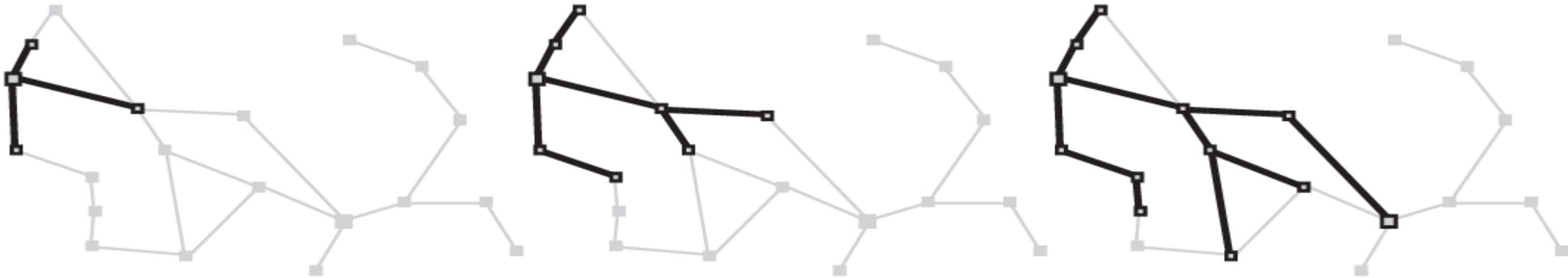


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

- To avoid exploring redundant paths, the TREE-SEARCH algorithm uses a data structure called the *explored set* (closed list), which remembers every expanded node,
- Newly generated nodes that match previously generated nodes in the explored set or the frontier can be discarded instead of being added to the frontier.
- The GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph.
- The algorithm has another nice property that the frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.
- As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

```
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
```

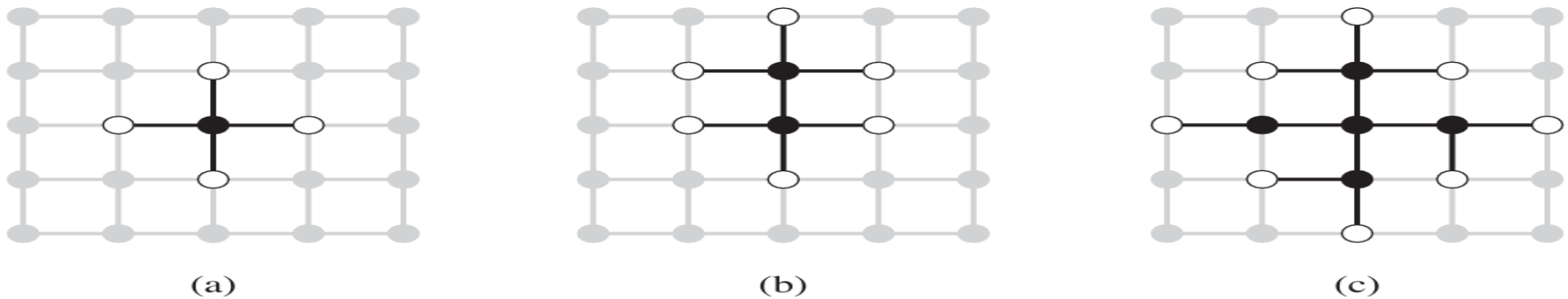


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

3.3.1. Infrastructure for search algorithms

- A node is a bookkeeping data structure used to represent the search tree.
- A state corresponds to a configuration of the world.
- A node in the tree is represented by a data structure with four components:
  - *node.STATE* – the state to which the node corresponds;
  - *node.PARENT* – the node in the tree that generated this node;
  - *node.ACTION* – the action that was applied to the parent's state to generate this node;
  - *node.PATH-COST* – the total cost of the path from the initial state to this node. In mathematical formulas, we use *g(node)* as a synonym for PATH-COST.
- Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution

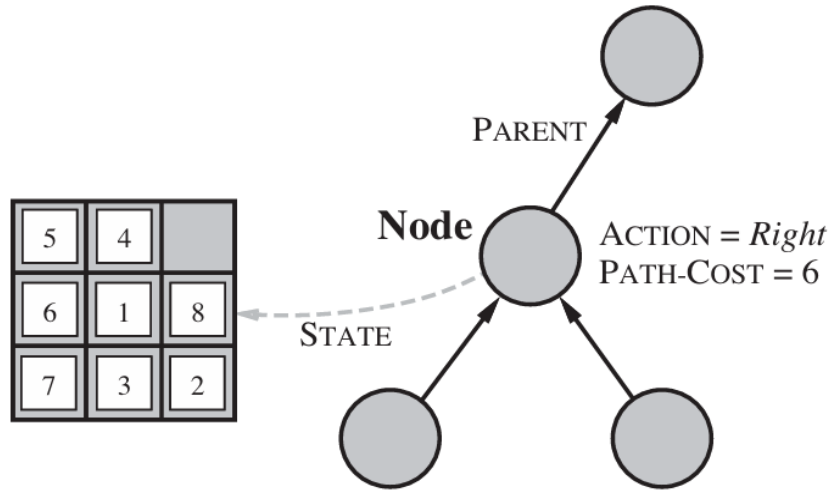


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- Three kinds of queues are used in search algorithms:
  - A *FIFO queue* or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
  - A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.
  - A *priority queue* first pops the node with the minimum cost according to some evaluation function *f*, It is used in best-first search.
- The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.
- *Frontier* is the set of all leaf nodes available for expansion in any given point of time. The operations on frontier as follows
  - *IS-EMPTY(frontier)* returns true only if there are no nodes in the frontier.
  - *POP(frontier)* removes the top node from the frontier and returns it.
  - *TOP(frontier)* returns (but does not remove) the top node of the frontier.
  - *ADD(node, frontier)* inserts node into its proper place in the queue.

- *Redundant paths* - refers to a path in a graph or problem space that revisits states or nodes already explored.
  - Do not contribute to finding a solution
  - Can waste computational resources by exploring the same states multiple times.
  - improve efficiency aim to avoid or minimize the exploration of redundant paths.
- *Loopy path* - it revisits at least one node more than once. It can lead to infinite loops or redundant exploration.
- The search algorithm is a *graph search* if it checks for redundant paths and *a tree-like search* if it does not check.

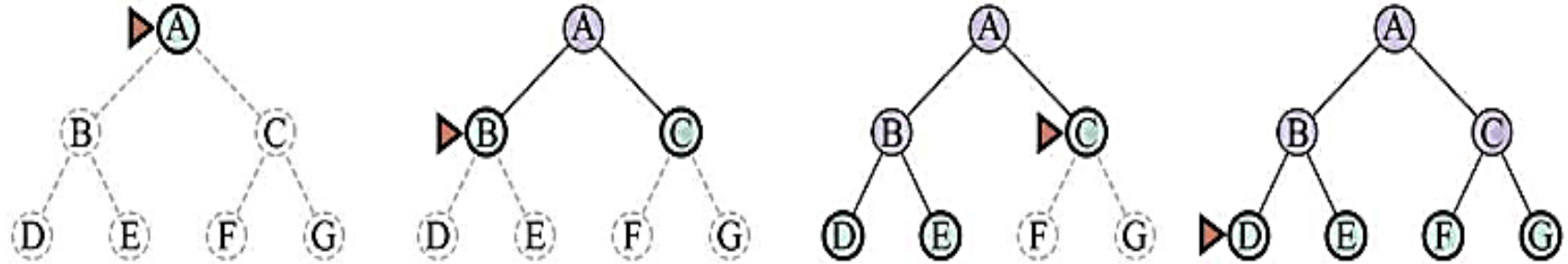
- *Measuring problem-solving performance* - evaluate an algorithm's performance in four ways.
  - *COMPLETENESS* – Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?
  - *COST OPTIMALITY* – Does it find a solution with the lowest path cost of all solutions?
  - *TIME COMPLEXITY* – How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
  - *SPACE COMPLEXITY* – How much memory is needed to perform the search?

3.4. Uninformed/blind Search Strategies

- These strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- These search strategies are distinguished by the order in which nodes are expanded.
- An uninformed search algorithm is given no clue about how close a state is to the goal(s).
  - *Breadth-First Search (BFS)*
  - *Uniform Cost Search (UCS)*
  - *Depth-First Search (DFS)*
  - *Depth-Limited Search (DLS)*
  - *Bidirectional Search*

3.4.1. Breadth-First Search (BFS)

- When all actions have the same cost, an appropriate strategy is breadth-first search, 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.
- In BFS the shallowest unexpanded node is chosen for expansion which is achieved by using a FIFO queue for the frontier.
- The goal test is applied to each node when it is generated rather than when it is selected for expansion.
- It is optimal if the path cost is a nondecreasing function of the depth of the node.
- The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.




```
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)
```

Figure 3.11 Breadth-first search on a graph.

- Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth *d* it has already generated all the nodes at depth *d-1* so if one of them were a solution, it would have been found.
- That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don't have that property.
- The root of the search tree generates nodes, each of which generates more nodes, for a total of at the second level. Each of these generates more nodes, yielding nodes at the third level, and so on. (b is the branching factor)
- Suppose that the solution is at depth *d,* then the total number of nodes generated is: $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- All the nodes remain in memory, so both time and space complexity are $O(b^d)$.
- *In general, exponential complexity search problems cannot be solved by uninformed search for any but the smallest instances.*
- *The memory requirements are a bigger problem for breadth-first search than the execution time.*

3.4.2. Uniform Cost Search (Dijkstra's Algorithm)

- When all action cost are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- Instead of expanding the shallowest node, uniform-cost search expands the node *n* with the *lowest path cost g(n)*.
- The algorithm's frontier data structure is *priority queue* order by *g*.
- The goal test is applied to a node when it is *selected for expansion* rather than when it is first generated.
- A test is added in case a better path is found to a node currently on the frontier.
- *The 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 is guided by path costs rather than depths, so its complexity is not easily characterized in terms of *b* and *d*.

```
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.

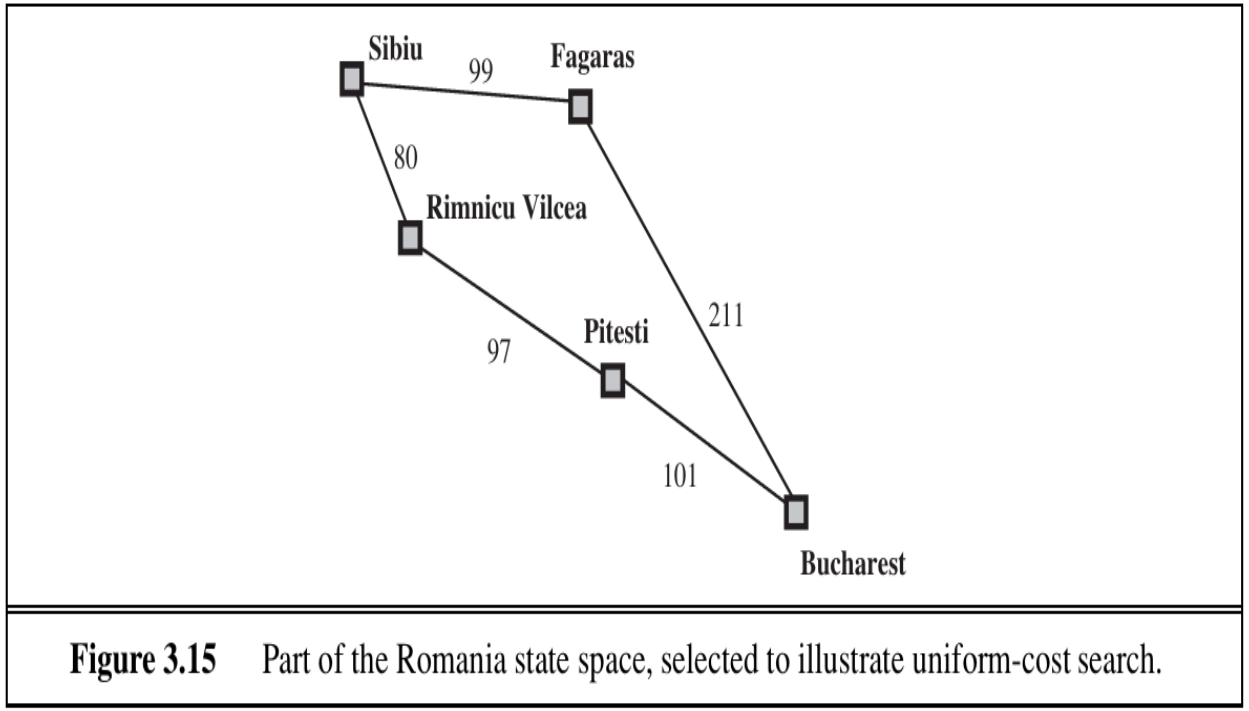
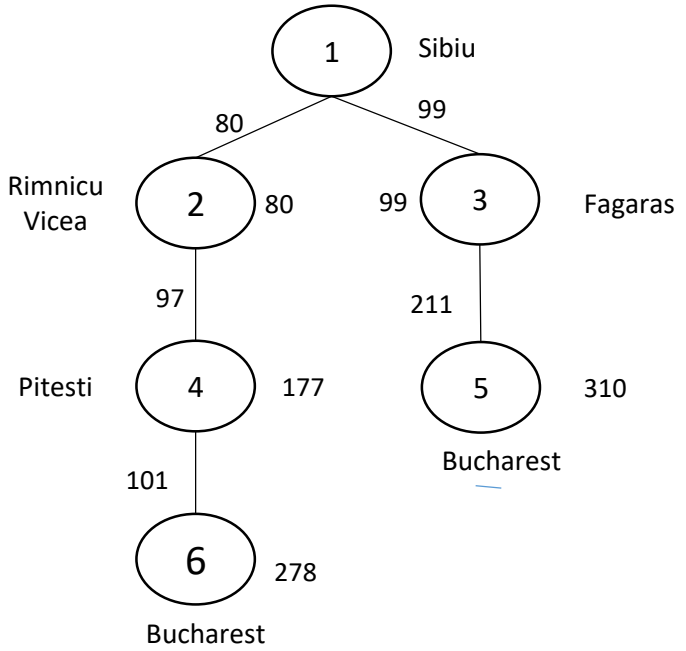


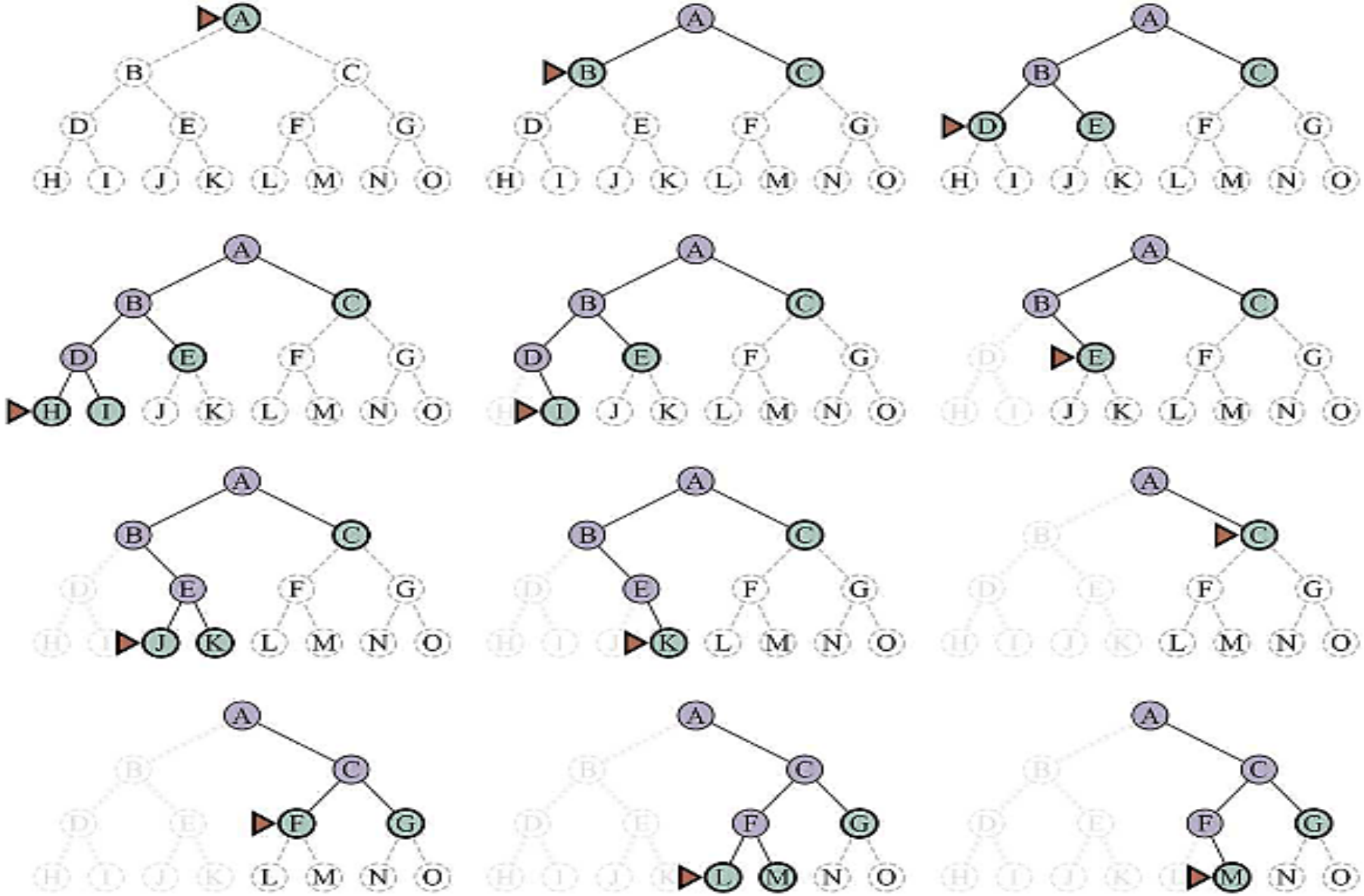
Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.



- The UCS expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution.
- The UCS is effective in finding the least-cost path, its disadvantages, particularly related to memory usage, speed, and inefficiency in certain conditions, make it less suitable for very large or complex graphs.

3.4.3. Depth-first search (DFS)

- 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 and uses a LIFO queue
- Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.
- The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, 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.


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

3.4.4. Depth-limited search

- Depth-first search can fail in infinite state spaces, but this can be fixed by using a predetermined depth limit *l*.
- In depth-limited search, nodes at the specified depth *l* are treated as having no successors. This method solves the problem of infinite paths.
- Unfortunately, it also introduces an additional source of incompleteness if we choose $l<d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.) Depth-limited search will also be nonoptimal if we choose $l>d$.
- Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

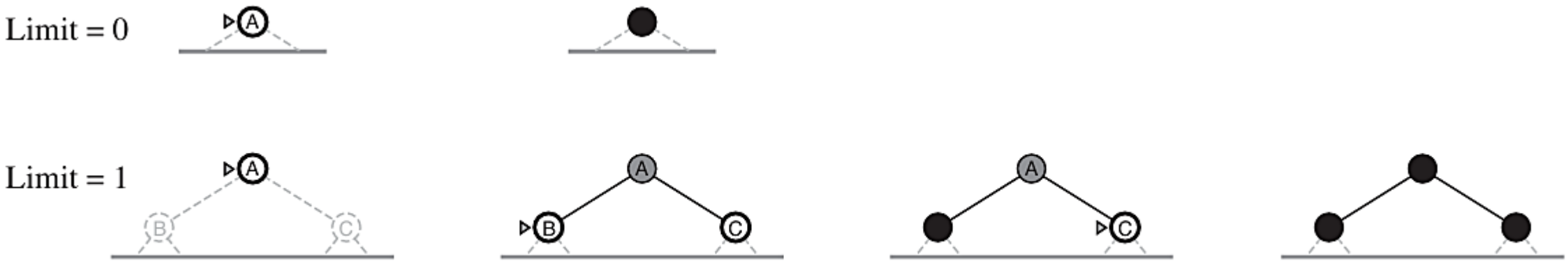
Figure 3.17 A recursive implementation of depth-limited tree search.

3.4.5. Iterative deepening depth-first search

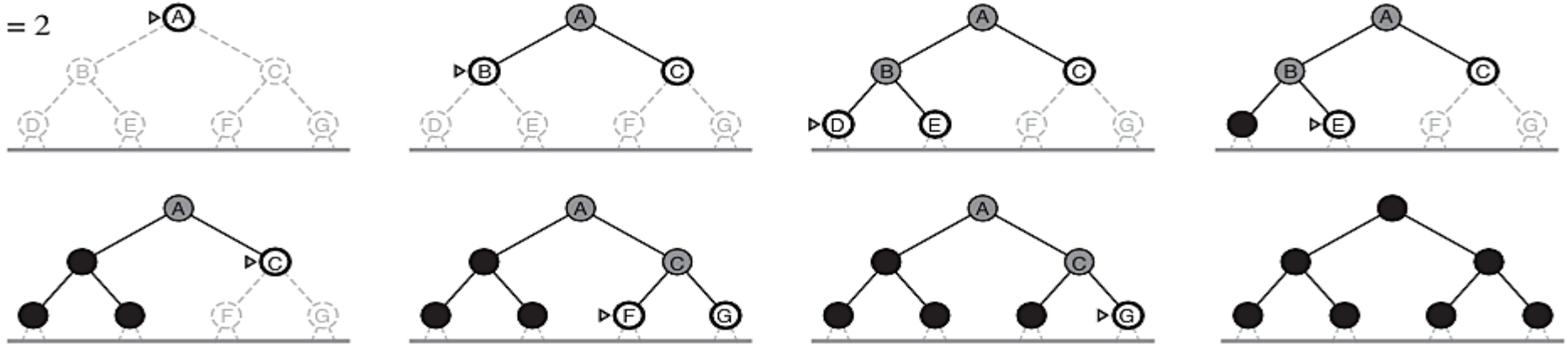
- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

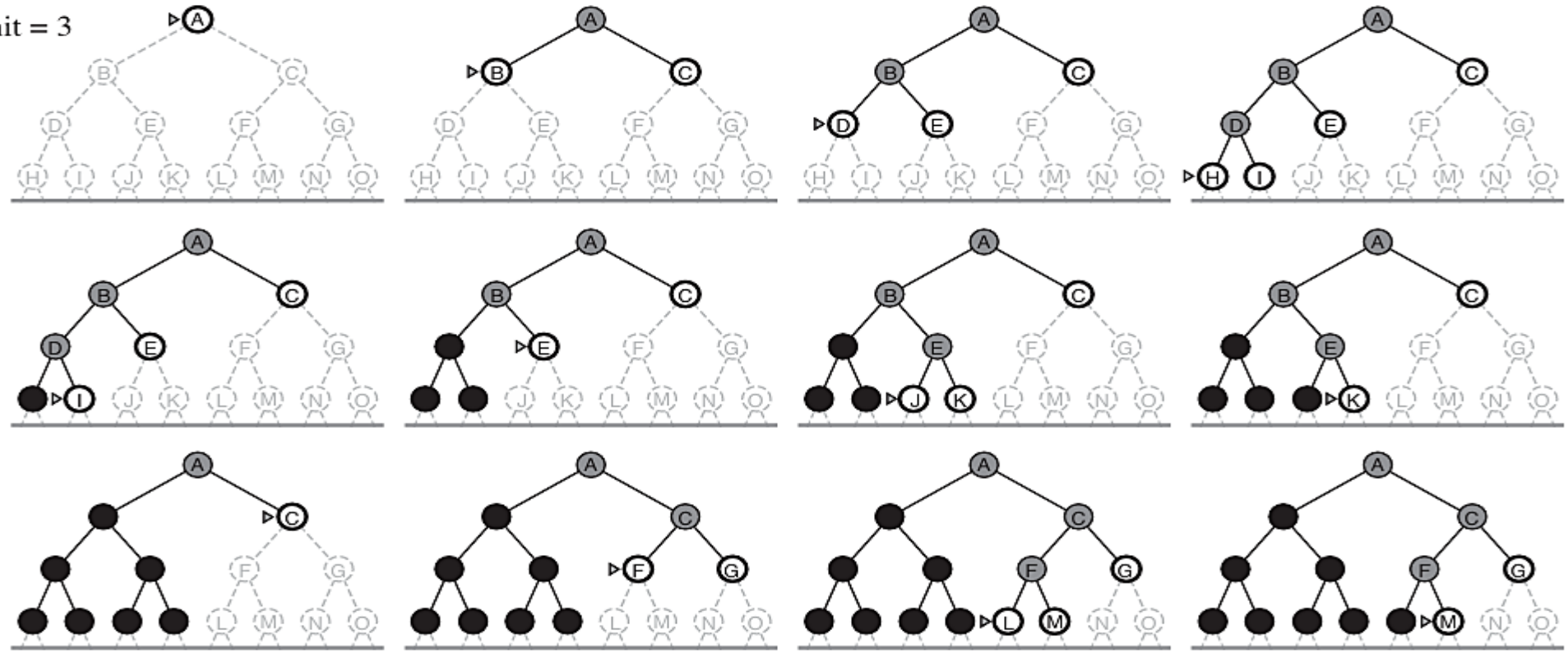
Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.



Limit = 2



Limit = 3



3.4.6. Bidirectional search

- Bidirectional search runs two simultaneous searches: one forward from the initial state and one backward from the goal, aiming for the two searches to 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; if they do, a solution has been found.
- The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time.
- *For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search.*
- *But if the goal is an abstract description, such as the goal that "no queen attacks another queen" in the n-queens problem, then bidirectional search is difficult to use*

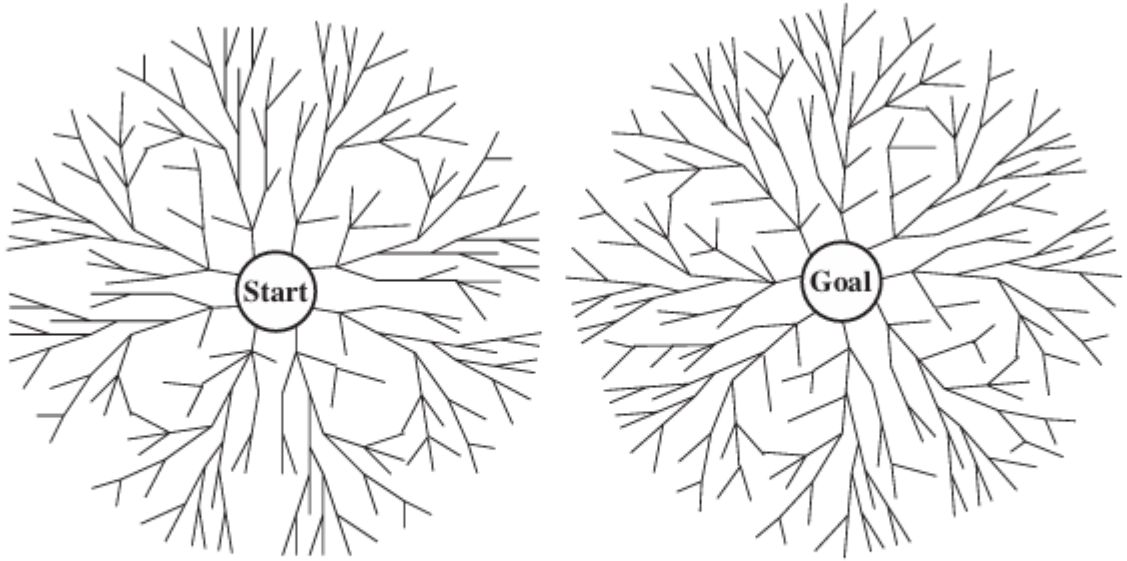


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

3.4.7. Comparing Uninformed Search Strategies

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

Figure 3.21 Evaluation of tree-search strategies. 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. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.