Problem Solving and Search

Prof. Suresh Kapare

Searching for Solutions- after formulation now need to solve problems

- A solution is an action sequence, so search algorithms work by considering various 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 set of all leaf nodes available for expansion at any given point is called the **frontier**.
- 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.**

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.

Search Tree Example

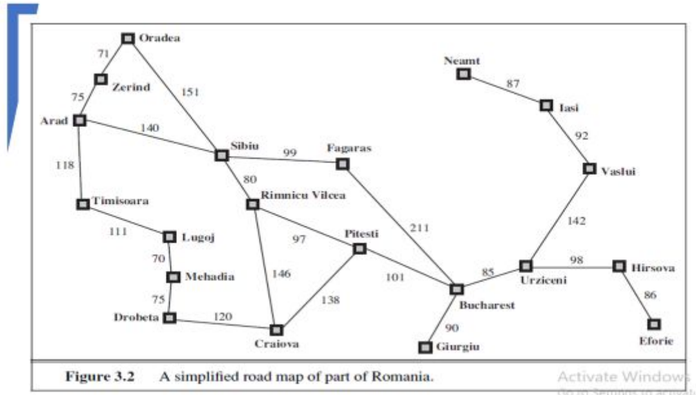

Figure 3.2 A simplified road map of part of Romania.

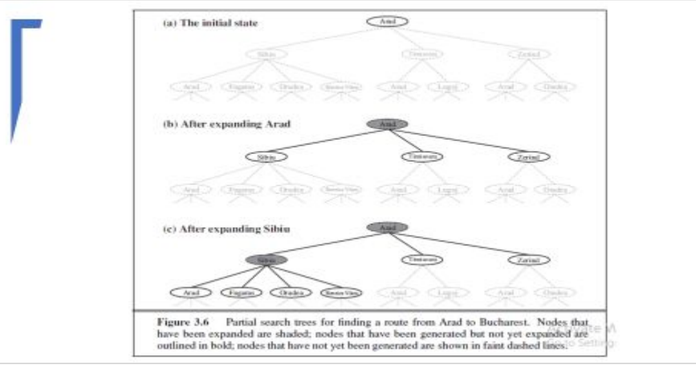

Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

- **Simplified road map**
- **Search Tree**
- In(Arad) is a repeated state in the search tree, generated in this case by a **loopy path**.
- the complete search tree for Romania is *infinite*
- 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.
- Define the problem itself to eliminate redundant paths.

- For example, The 8-queens problem - a queen can be placed in any column, then each state with n queens can be reached by n! different paths;
- Reformulation: each new queen is placed in the leftmost empty column, then each state can be reached only through one path.

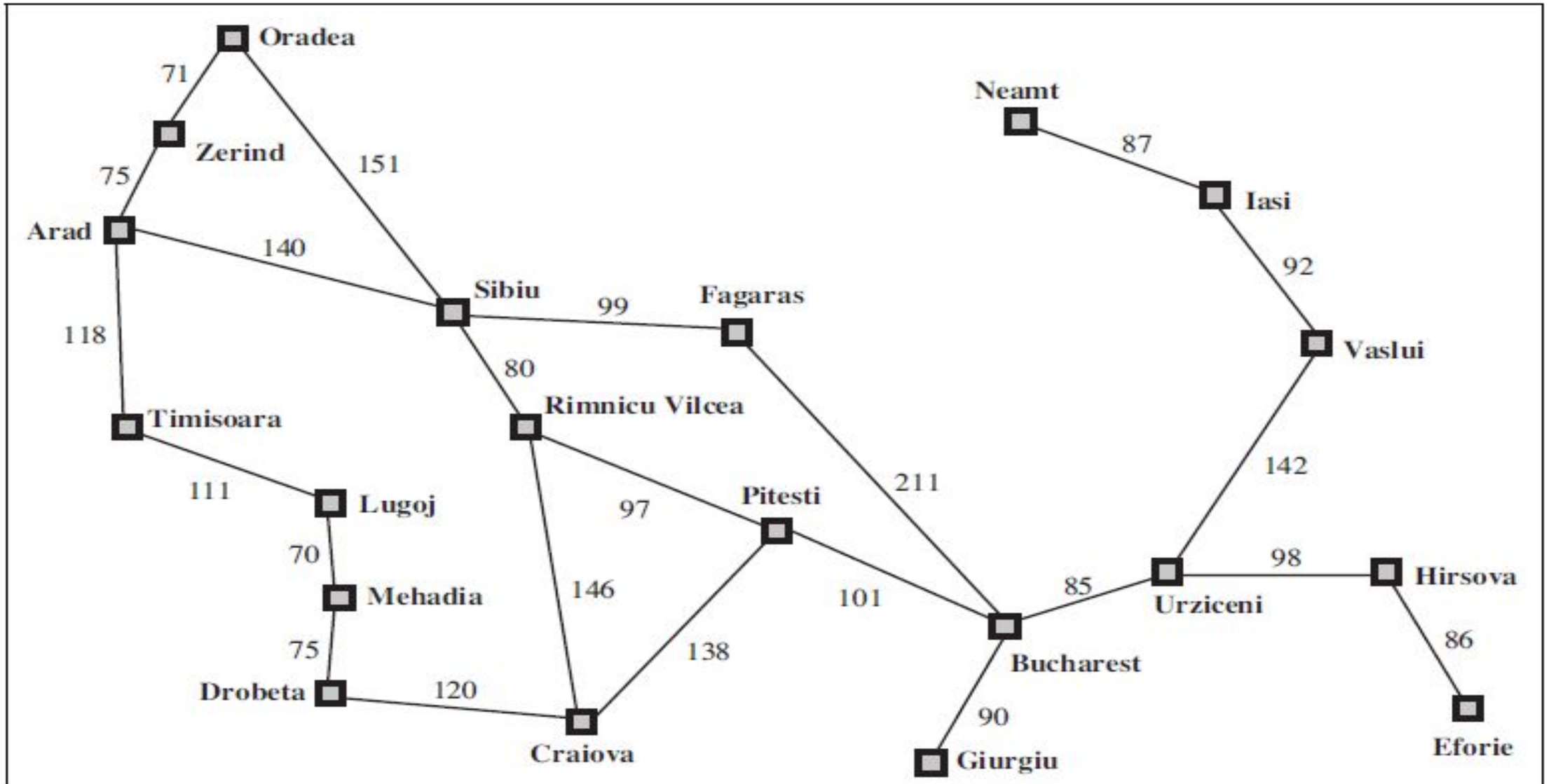

Figure 3.2 A simplified road map of part of Romania.

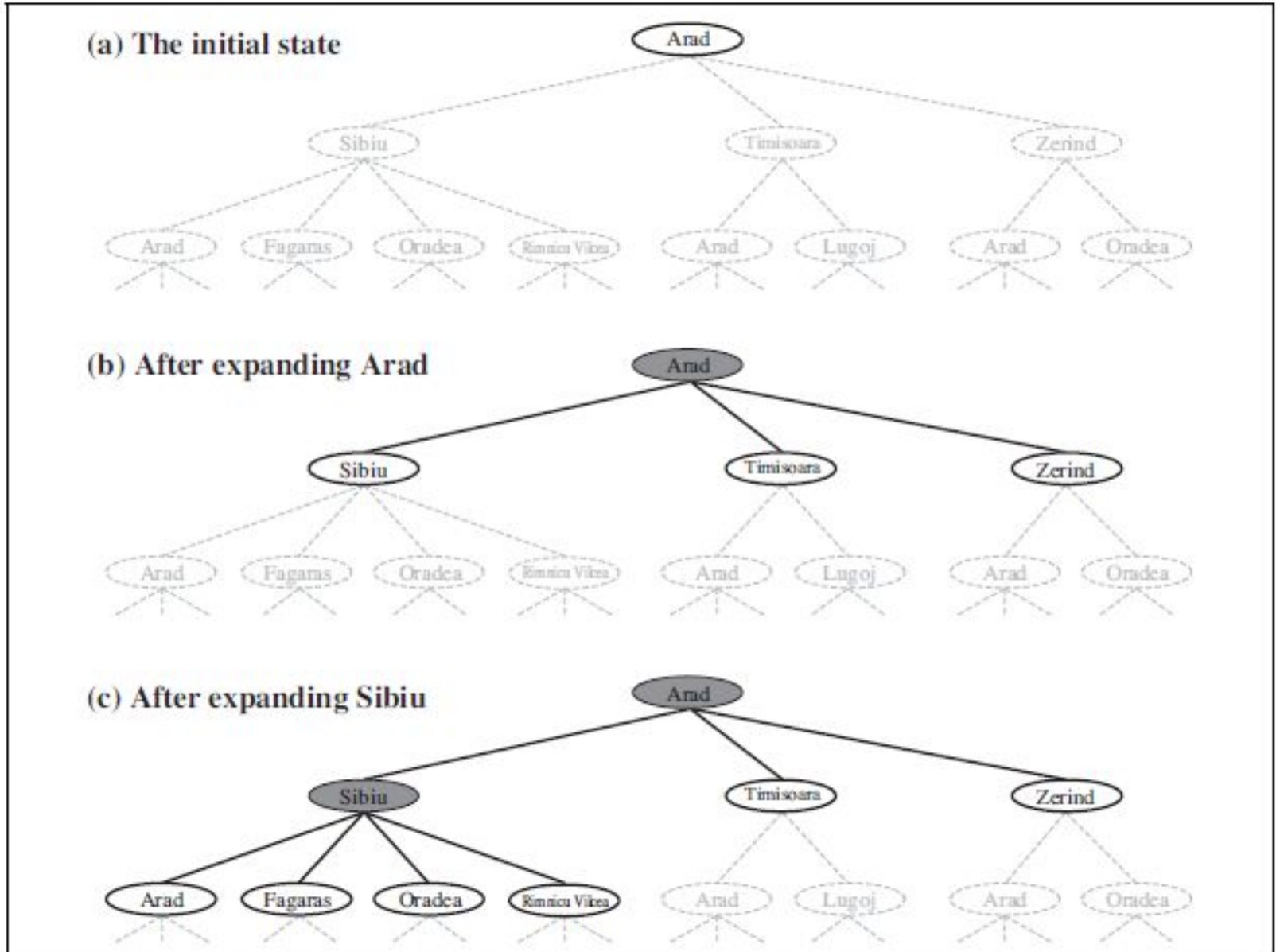

Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Search Tree

- Redundant paths are unavoidable for problems with reversible actions, such as route-finding problems and sliding-block puzzles.

- *Algorithms that forget their history are doomed to repeat it.*

- Way to avoid redundant paths is to remember where one has been.
- To do this, Use TREE-SEARCH algorithm with a data structure called the **explored set/ closed list**, which remembers every expanded node.
- Newly generated nodes that match previously generated nodes can be discarded
- Structure for *n* node -> *n.state, n.parent, n.action, n.path-cost,*
- The new algorithm, called **GRAPH-SEARCH**

Graph-Search

- 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 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.

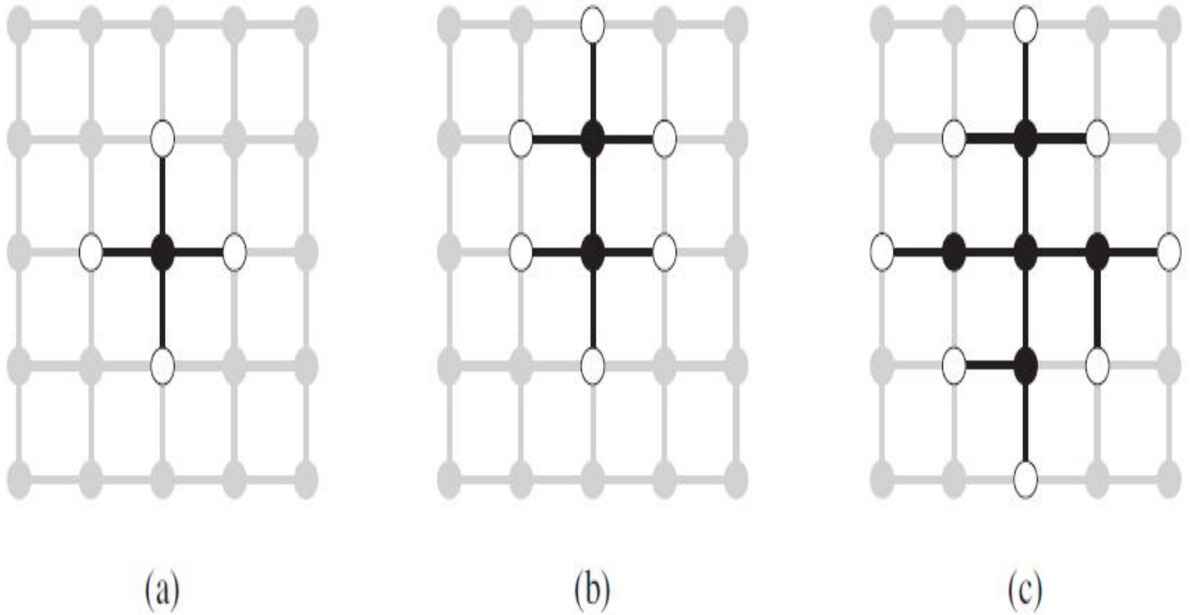

Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

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?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?
- To assess the effectiveness of a search algorithm, we can consider just the search cost—which typically depends on the time complexity but can also include a term for memory usage—or we can use the **total cost**, which combines the search cost and the path cost of the solution found.

Uninformed search strategies

- This section covers five search strategies that come under the heading of **uninformed search** (also called blind search).
- 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 nongoal state.
- Strategies that know whether one non goal state is "more promising" than another are called **informed search** or **heuristic search** strategies.
- All search strategies are distinguished by the order in which nodes are expanded

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.
- Level wise expansion.
- Its an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion.
- Using a FIFO queue for the frontier.
- New nodes at the end of queue, and old nodes get expanded first.
- The goal test is applied to each node when it is *generated* rather than when it is selected for expansion.
- The algorithm, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on 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)
```

Figure 3.11 Breadth-first search on a graph.

- For b successors in every state and for depth d
  $b+b^2+b^3+b^4+\ldots\ldots+b^d=O(b^d)$
- Time Complexity including goal test: **$O(b^{d+1})$**
- Space complexity: stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity -> **$O(b^d)$**.

An exponential complexity bound such as $O(b^d)$ is scary.

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.

- ***the memory requirements are a bigger problem for breadth-first search than is the execution time.***
- ***exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances***

Uniform-cost search

- All step costs are equal, breadth-first search is optimal.
- Any step-cost function……..?
- **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.
  - The goal test is applied to a node when it is selected for expansion.
  - A Test is added in case a better path is found to a node currently on the frontier.

Differs than BST

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
```

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.

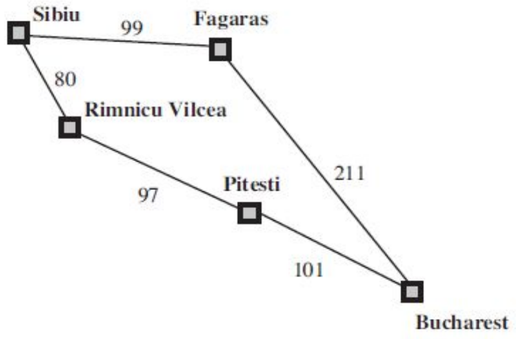

Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

- g-cost = 278
- Uniform-cost search is optimal in general.
- First, we 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, by the graph separation property of Figure 3.9;

by definition, 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.

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.
- Uses LIFO (Stack)

Depth-first search

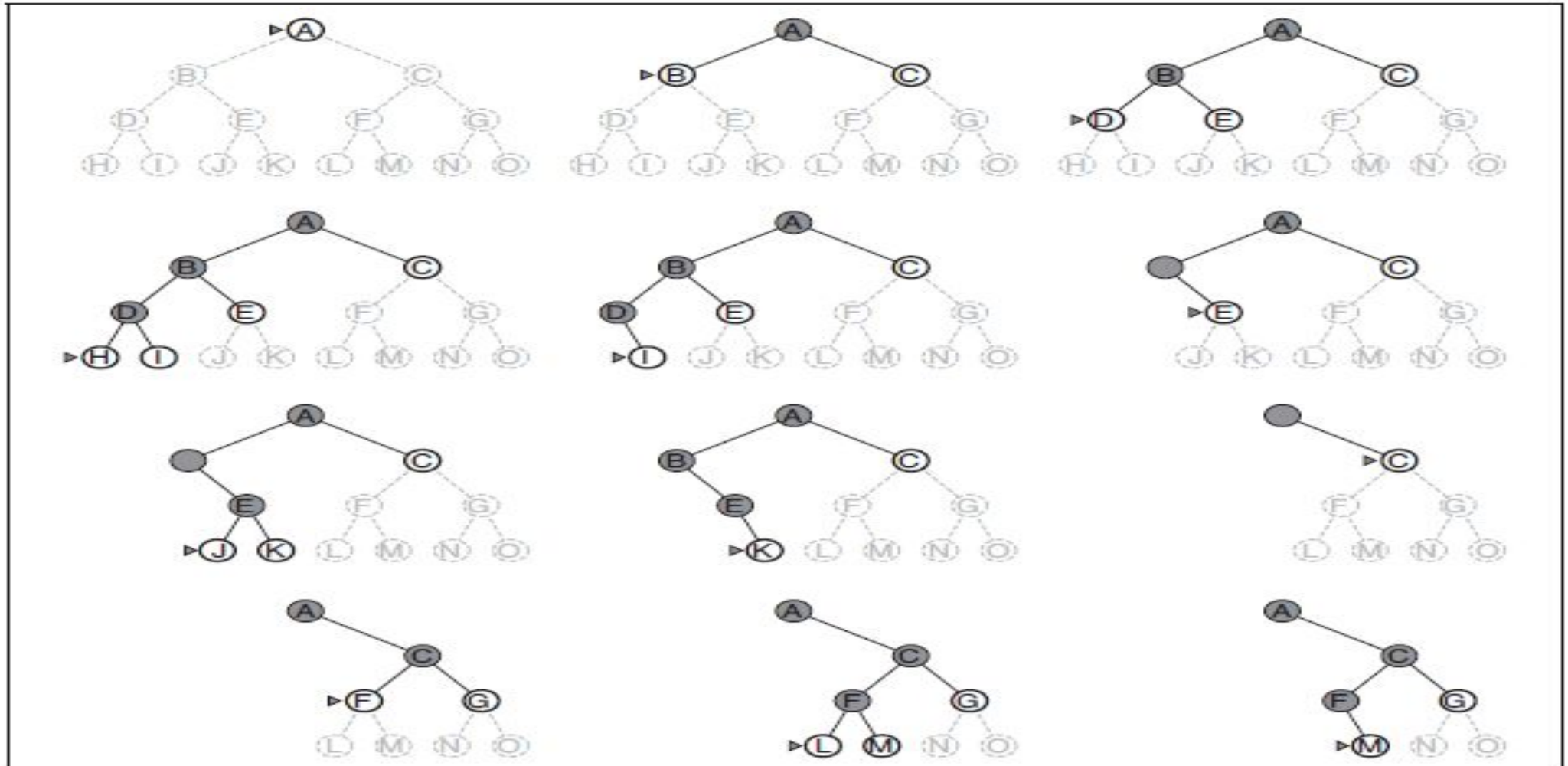

Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth-First Search

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths.
- In infinite state spaces, both versions fail if an infinite non-goal path is encountered.

- The time complexity of depth-first graph search is bounded by the size of the state space
(which may be infinite).
- It 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. Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.
- For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only $O(b^m)$ nodes.
- we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth d = 16, a factor of 7 trillion times less space.
- This has led to the adoption of depth-first tree search as the basic workhorse of many areas of AI, including constraint satisfaction, propositional satisfiability, and logic programming.

Depth-First Search

- A variant of depth-first search called backtracking BACKTRACKING search uses still less memory.
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
- In this way, only O(m) memory is needed rather than $O(b^m)$.
- Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first.
- This reduces the memory requirements to just one state description and O(m) actions.
- For this to work, we must be able to undo each modification when we go back to generate the next successor.
- For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

Depth-limited search

- failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called **depth-limited search**.
- Solves infinite loop problem.
- **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 $O(bl)$.
- Depth-first search can be viewed as a special case of depth-limited search with $l=\infty$.
- Any city of example can be reached from any other city in at most 9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.
- For most problems, however, we will not know a good depth limit until we have solved the problem.
- Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm.
- Alternatively, it can be implemented as a simple recursive algorithm as shown in next Figure.
- **Notice that depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.**

```
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.

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.
- Iterative deepening combines the benefits of depth-first and breadth-first search.
- Like depth-first search, its memory requirements are modest: O(bd) to be precise.
- 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.

```
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.

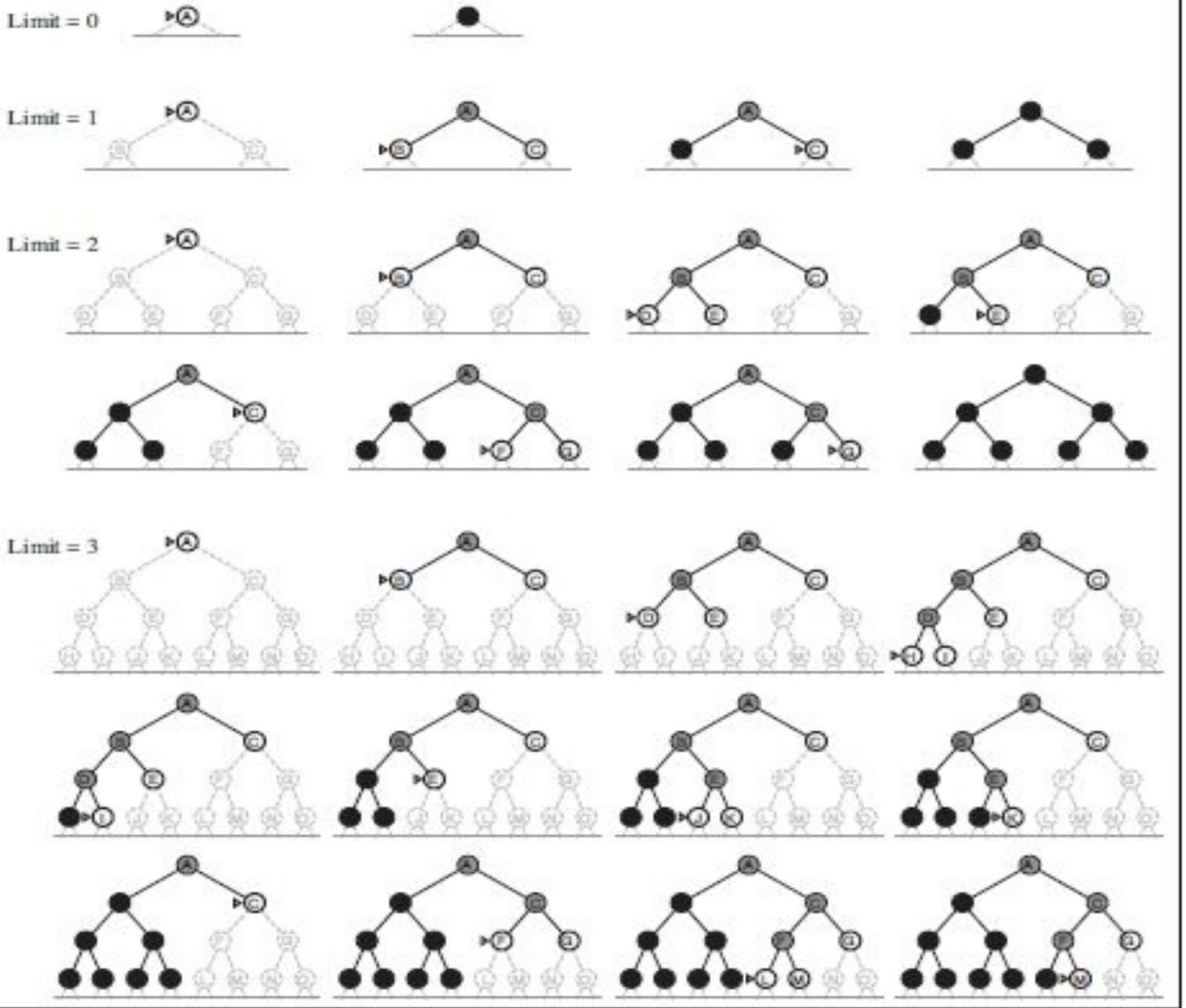

Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Iterative deepening depth-first search

- May seem wasteful because states are generated multiple times.
- It turns out this is not too costly as 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.
- 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.

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

Gives time complexity $O(b^d)$ like BFS.

- Some extra cost for generating the upper levels multiple times, but it is not large.
- **About repeating the repetition**, can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier.
- Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.
- Analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer.
- Worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm's optimality guarantees while avoiding its memory requirements.
- Increasing path-cost limits instead of increasing depth limits, called **iterative lengthening search.**
- It turns out, unfortunately, incurs substantial overhead compared to uniform-cost search.

Bidirectional Search

- 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.
- **The motivation:** $b^{d/2} + b^{d/2}$ is much less than bd, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.
- **Solution Found:** by replacing the goal test with a check to see whether the frontiers of the two searches intersect;
- First solution found may not be optimal, additional search required to make sure no another short-cut across. Done when each node is generated or selected for expansion with a hash table, will take constant time.
- E.g, if a problem has solution depth d=6, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. For b=10, this means a total of 2,220 node generations, compared with 1,111,110 for a standard breadth-first search. Thus, the time complexity of bidirectional search using breadth-first searches in both directions is $\mathbf{O(b^{d/2})}$. The space complexity is also $\mathbf{O(b^{d/2})}$.
- **Significant weakness:** Can reduce complexity roughly by half if one of two searches done by iterative deepening, but at least one of the frontiers must be in **memory** so intersection check can be done.

Bidirectional Search

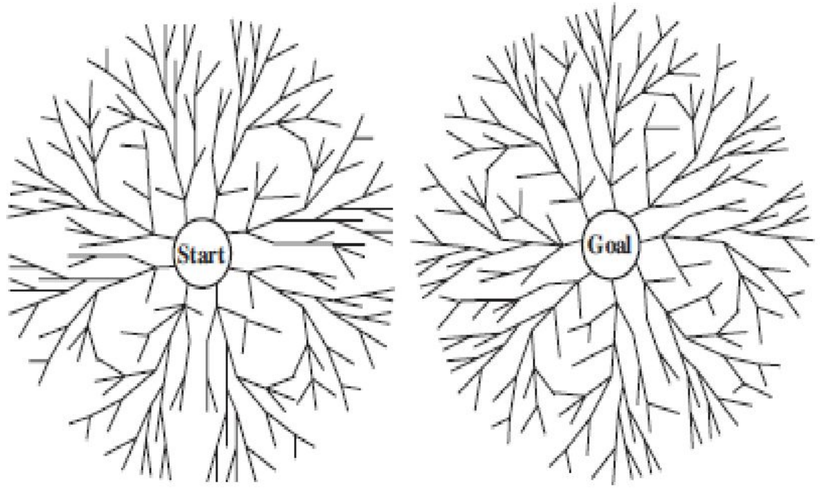

Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

- Attractive due to reduction in time complexity but how do we search backward?
- Let the predecessors of a state x be all those states that have x as a successor.
- Requires a method for computing predecessors.
- When all the actions in the state space are reversible, the predecessors of x are just its successors.
- Consider the question of what we mean by “the goal” in searching “backward from the goal.”
- If one goal state like 80 puzzle and route finding, the backward search is very much like the forward search.
- If several listed goal states like two dirt-free goal states in Vacuum world then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states.
- 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.

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.