# Chapter 3

## SOLVING PROBLEMS BY SEARCHING

Prepared By:
Dipanita Saha
Assistant Professor
Institute of Information Technology(IIT)
Noakhali Science and Technology University

## PROBLEM-SOLVING AGENTS

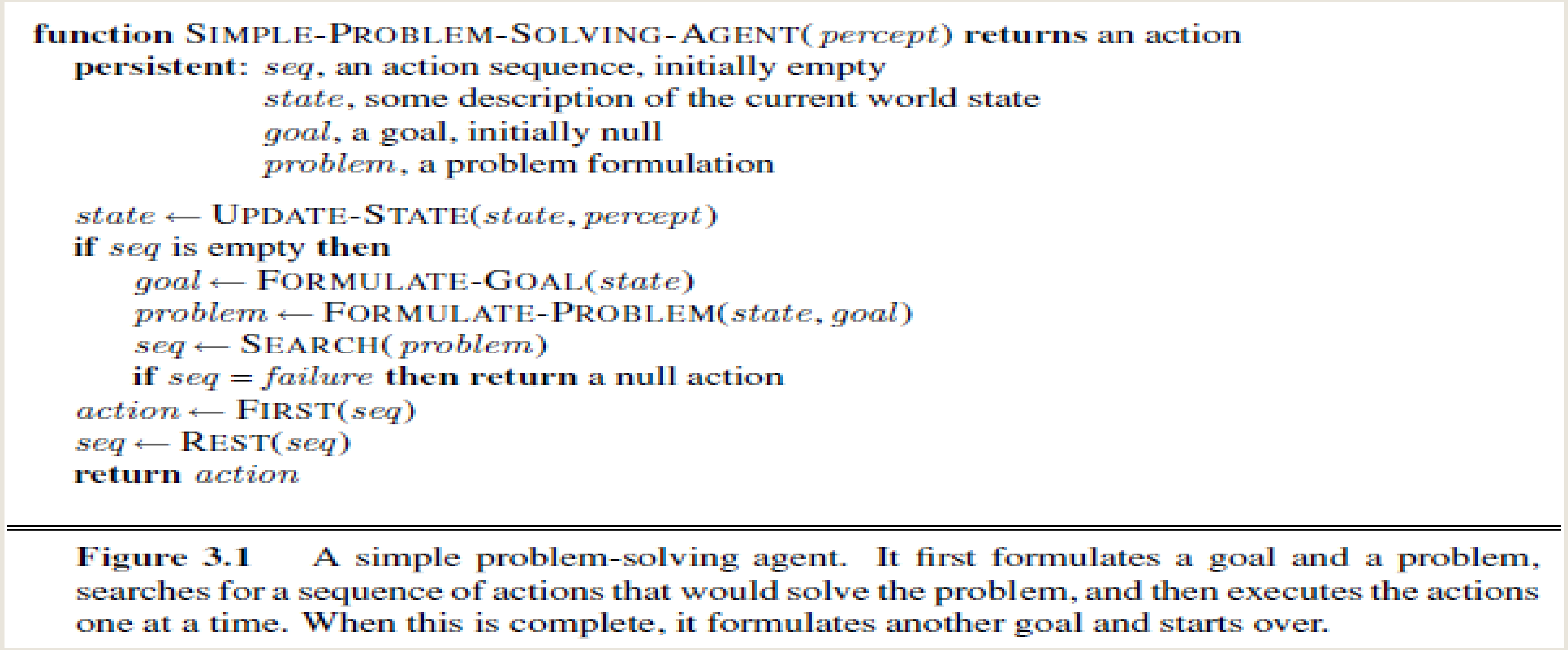
- Intelligent agents are supposed to maximize their performance measure. For achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.
- why and how an agent might do this.
- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
- The agent’s performance measure contains many factors: it wants to improve its suntan, improve its Romanian, take in the sights, enjoy the nightlife (such as it is), avoid hangovers, and so on.
- Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the **goal** of getting to Bucharest.
- 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**, based on the current situation and the agent’s performance measure, is the first step in problem solving.
- We will consider a goal to be a set of world states in which the goal is satisfied.
- The agent’s task is to find out how to act, now and in the future.

- If it were to consider actions at the level of “move the left foot forward an inch” or “turn the steering wheel one degree left,” the agent would probably never find its way out of the parking lot, let alone to Bucharest
- because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal. let us assume that the agent will consider actions at the level of driving from one major town to another.
- Each state therefore corresponds to being in a particular town
- Our agent has now adopted the goal of driving to Bucharest and is considering where to go from Arad.
- Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these achieves the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.
- In other words, 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** then it is has no choice but to try one of the actions at random.

## Well-defined problems and solutions

- 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).*



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

**Figure 3.1** A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

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. **transition model**, A description of what each action does; is 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. For example, we have

*RESULT(In(Arad),Go(Zerind)) = In(Zerind) .*

- Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or **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.

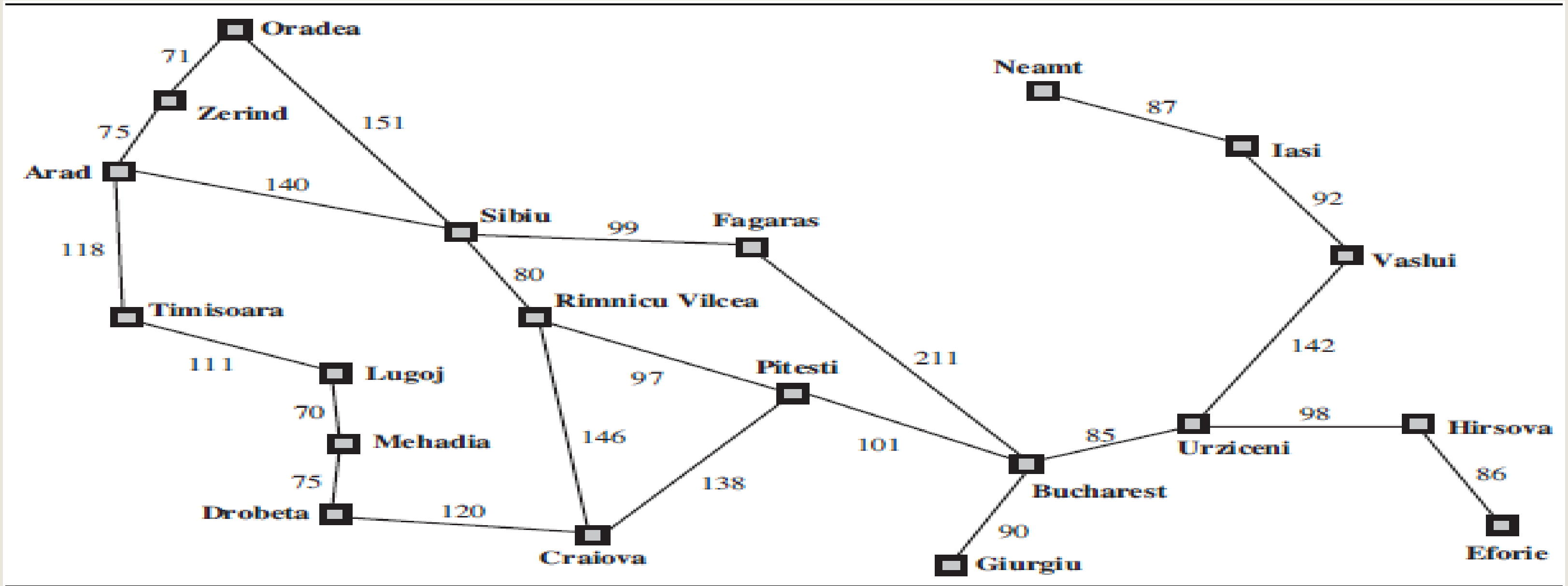


Figure 3.2 A simplified road map of part of Romania.

4. The **goal test**, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.

- The agent's goal in Romania is the singleton set *{In(Bucharest )}*.

5. 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.

- For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. 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.

## EXAMPLE PROBLEMS

- The problem-solving approach has been applied to a vast array of task environments. *Like toy* and *real-world* problems.
- A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.
- A **real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

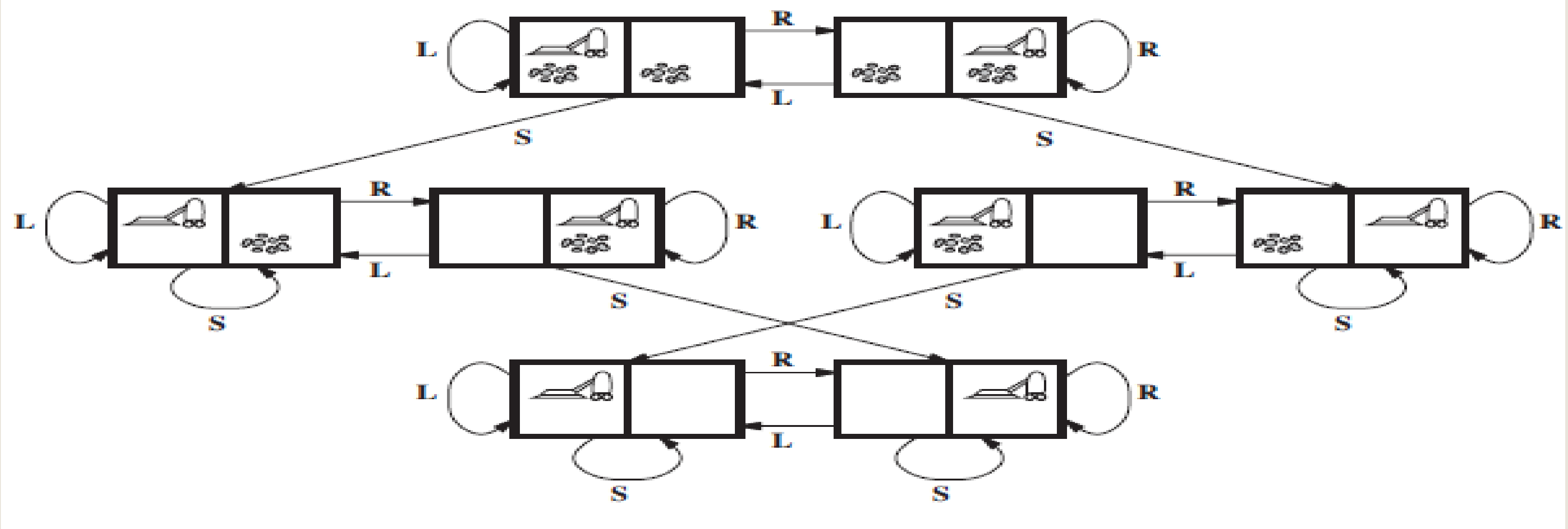


**Figure 3.3** The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

## ■ Toy problems

- The first example we examine is the **vacuum world**. This can be formulated as a problem as follows:
- • **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 × 2^2 = 8 possible world states. A larger environment with n locations has n ・ 2n states.

1. **Initial state**: Any state can be designated as the initial state.
2. **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
3. **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.
4. **Goal test**: This checks whether all the squares are clean.
5. **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

- Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier.

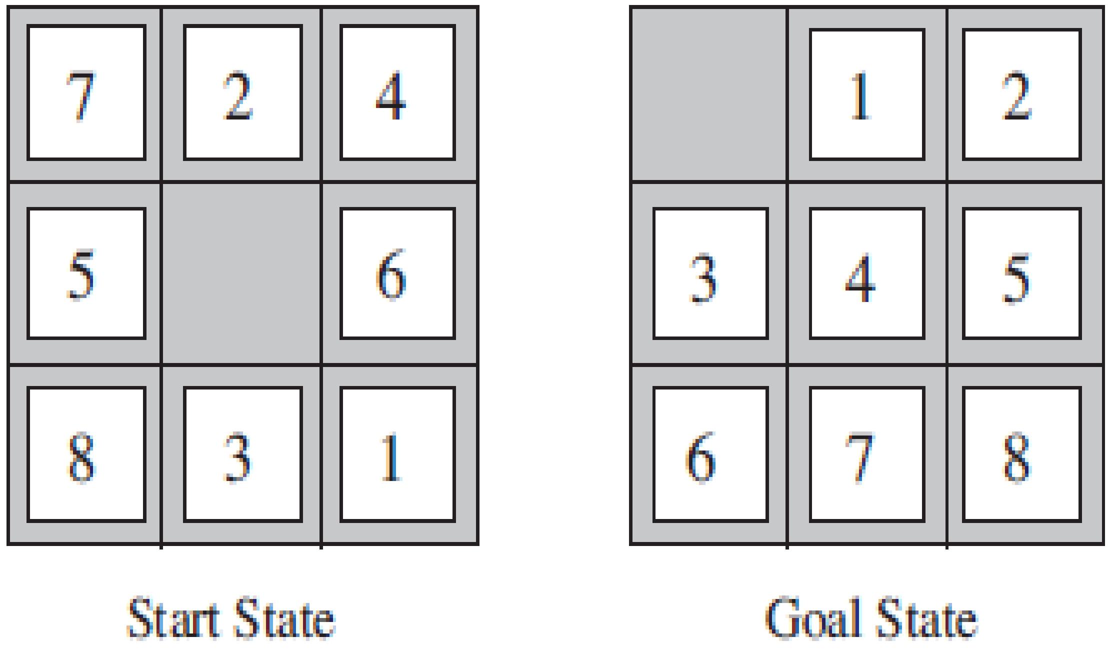


**Figure 3.4** A typical instance of the 8-puzzle.

- 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. The standard formulation is as follows:

1. **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
2. **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.
3. **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.
4. **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.
5. **Goal test**: This checks whether the state matches the goal configuration shown in Figure
6. **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

- The goal of the **8-queens problem** is to place eight queens on a chessboard such that no queen attacks any other.
- Figure shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.
- There are two main kinds of formulation.
- An **incremental formulation** involves operators that *augment* the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts.
- The first incremental formulation one might try is the following:

1. **States**: Any arrangement of 0 to 8 queens on the board is a state.
2. **initial state**: No queens on the board.
3. **Actions**: Add a queen to any empty square.
4. **Transition model**: Returns the board with a queen added to the specified square.
5. **Goal test**: 8 queens are on the board, none attacked.

■ In this formulation, we have 64 · 63 · · · 57 ≈ 1.8×10^14 possible sequences to investigate.

■ A better formulation would prohibit placing a queen in any square that is already attacked:

❖ **States**: All possible arrangements of n queens (0 ≤ n ≤ 8), one per column in the leftmost n columns, with no queen attacking another.

❖ **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

■ This formulation reduces the 8-queens state space from 1.8×10^14 to just 2,057, and solutions are easy to find.

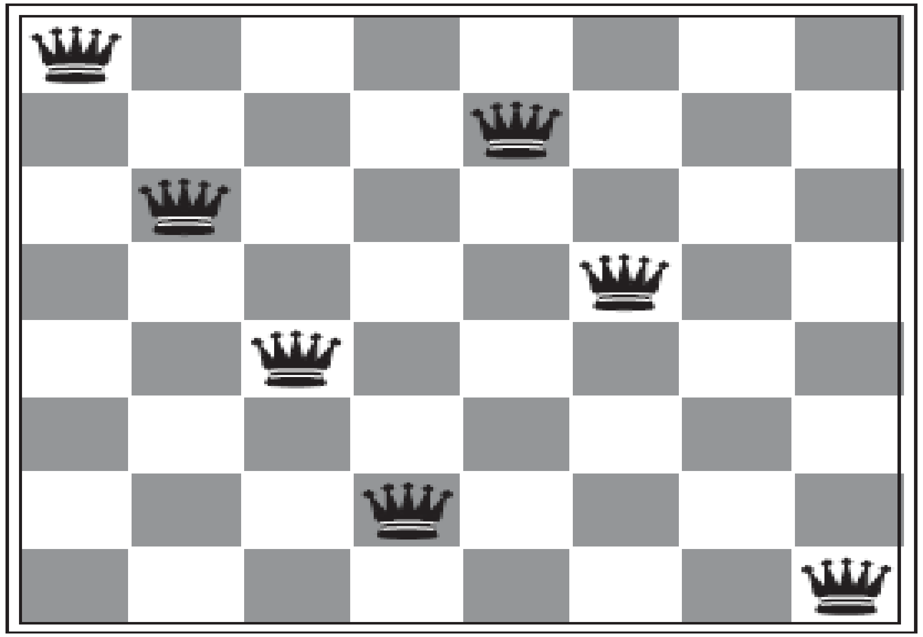
■ Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise.

■ Knuth conjectured that, starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5 \,.$$

■ The problem definition is very simple:

- **States**: Positive numbers.
- **Initial state**: 4.
- **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
- **Transition model**: As given by the mathematical definitions of the operations.
- **Goal test**: State is the desired positive integer.


Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

## Real-world problems

- We have already seen how the **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. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.
- Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.
- Consider the airline travel problems that must be solved by a travel-planning Web site:

1. **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.
2. **Initial state**: This is specified by the user’s query.
3. **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

4. **Transition model**: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
5. **Goal test**: Are we at the final destination specified by the user?
6. **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

- **Touring problems** are closely related to route-finding problems, but with an important difference.
- Consider, for example, the problem “Visit every city in Figure 3.2 at least once, starting and ending in Bucharest.”
- As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different.
- Each state must include not just the current location but also the *set of cities the agent has visited*.
- So the initial state would be In(Bucharest ), Visited({Bucharest}), a typical intermediate state would be *In(Vaslui ), Visited({Bucharest , Urziceni , Vaslui})*, and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

- **traveling salesperson problem** (TSP)
- **VLSI layout**
- **Robot navigation**
- **Automatic assembly sequencing**
- Read these examples from your textbook.

## 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.
- Figure 3.6 shows the first few steps in growing the search tree for finding a route from Arad to Bucharest.

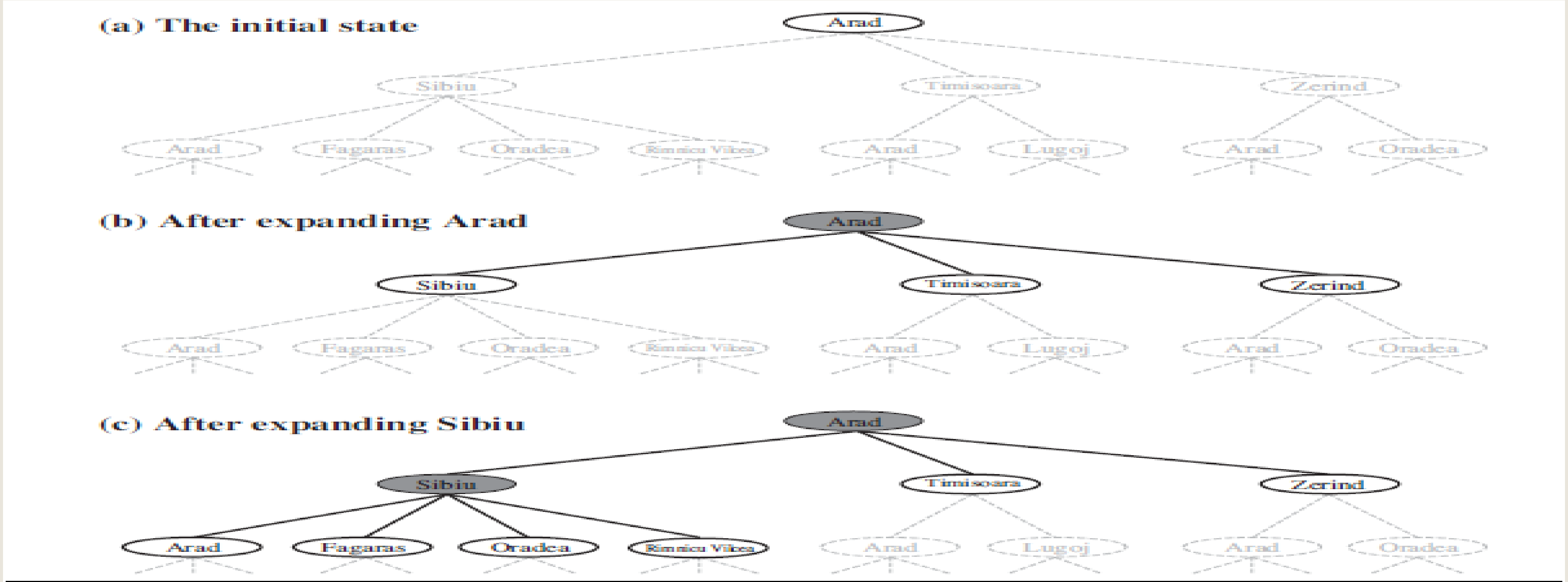


**Figure 3.6** Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

- The root node of the tree corresponds to the initial state, *In(Arad)*.
- The first step is to test whether this is a goal state. Then we need to consider taking various actions. We do this by **expanding** the current state; that is, applying each legal action to the current state, thereby **generating** a new set of states. In this case, we add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu), In(Timisoara),* and *In(Zerind)*
- Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get *In(Arad)*, *In(Fagaras)*, *In(Oradea)*, and *In(RimnicuVilcea)*.
- We can then choose any of these four or go back and choose Timisoara or Zerind. Each of these six nodes is a **leaf node**, that 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**.
- In Figure the frontier of each tree consists of those nodes with bold outlines.
- 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**.

- the search tree shown it includes the path from Arad to Sibiu and back to Arad again!
- We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**. Considering such loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop.
- loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable.
- Fortunately, there is no need to consider loopy paths. because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.
- 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.
- Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long). Obviously, the second path is redundant—it's just a worse way to get to the same state.
- In other cases, redundant paths are unavoidable. This includes all problems where the actions are reversible, such as route-finding problems and sliding-block puzzles.
- Route finding on a **rectangular grid** is a particularly important example in computer games.

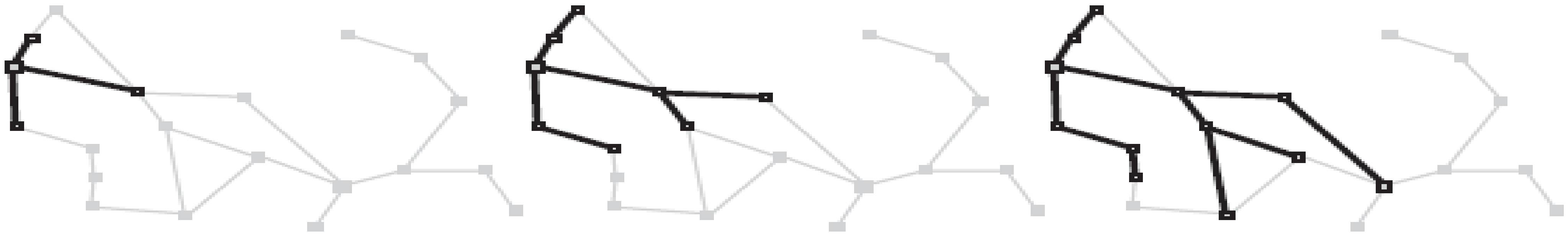


**Figure 3.8** A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other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

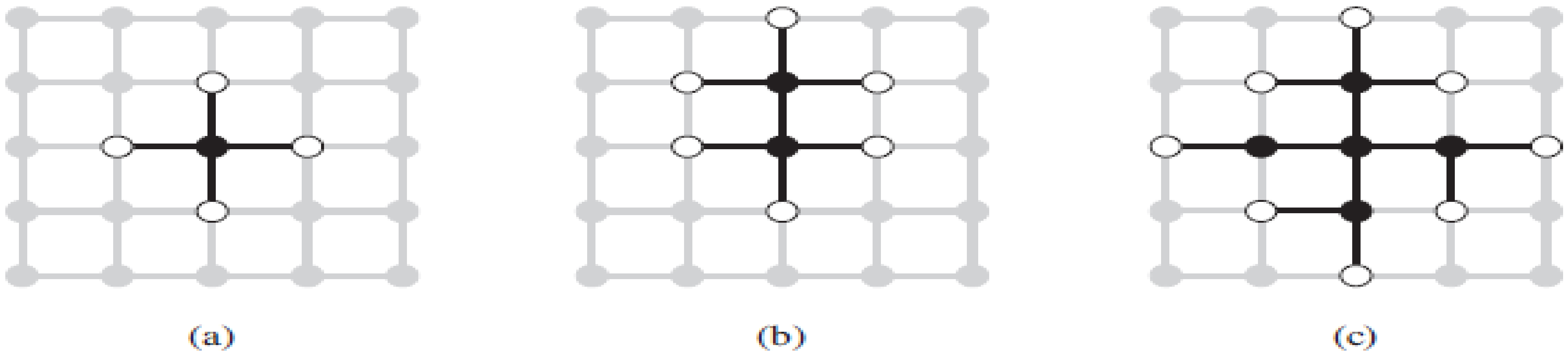


**Figure 3.9** The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has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

## Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- For each node n of the tree, we have a structure that 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, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

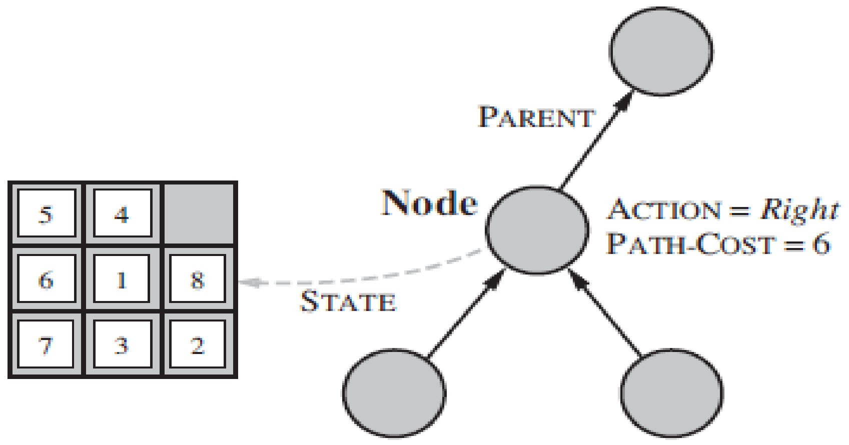
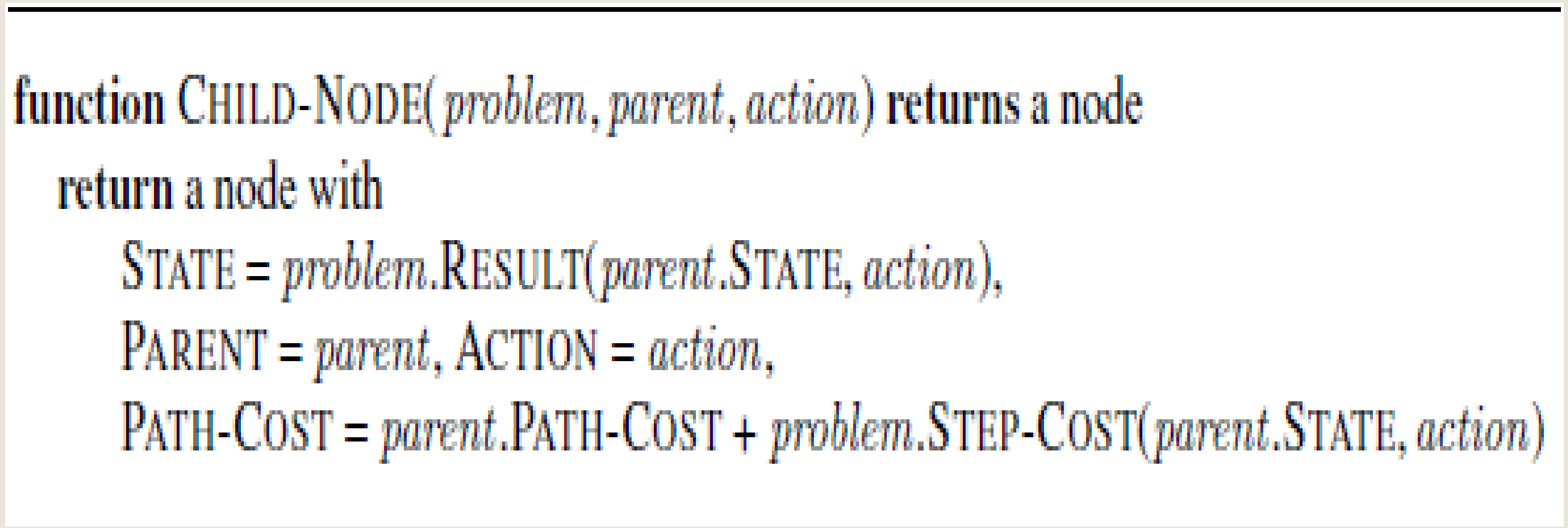


**Figure 3.10** Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

- The function CHILD-NODE takes a parent node and an action and returns the resulting child node:



```
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)
```

- Notice how the PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found;
- A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world.
- Thus, nodes are on particular paths, as defined by PARENT pointers, whereas states are not.

- 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 the first-in, first-out or **FIFO queue**, which pops the *oldest* element of the queue;
- the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue;
- and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

## Measuring problem-solving performance

- We can evaluate an 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?

- In theoretical computer science, the Time and space complexity measure 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).
- In AI, the 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, the **branching factor** or maximum number of successors of any node; d, the **depth** of the shallowest goal node; and m, the maximum length of any path in the state space.

## UNINFORMED SEARCH STRATEGIES

- The term means that the 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.

## 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 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 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.

- The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors.
- 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.
- Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level.
- Then the total number of nodes generated is b + b^2 + b^3 + · · · + b^d = O(b^d) .
- As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity.
- For breadth-first graph search in particular, every node generated remains in memory. There will be O(b^d−1) nodes in the explored set and O(b^d) nodes in the frontier, so the space complexity is O(b^d), i.e., it is dominated by the size of the frontier.

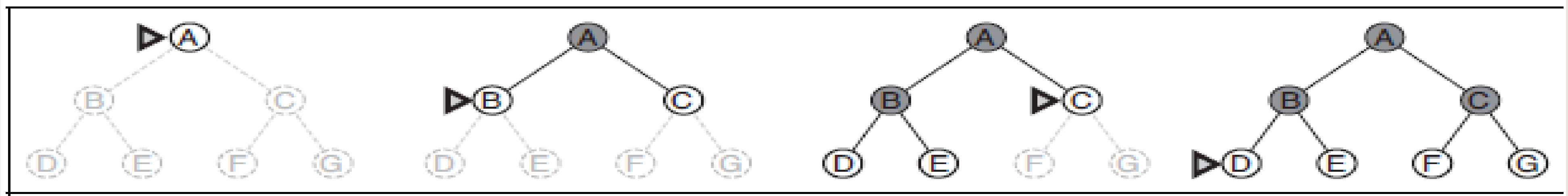
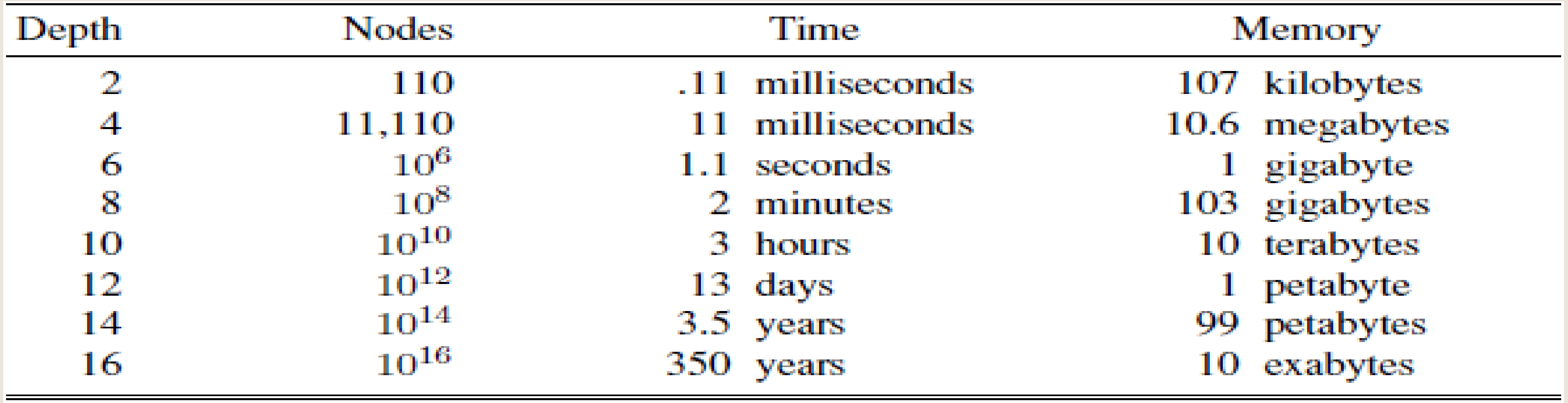


**Figure 3.12** Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- Two lessons can be learned
- **First**, *the memory requirements are a bigger problem for breadth-first search than is the execution time.* One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory.
- The **second** lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it.
- In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*



| 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

- When all step costs 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). This is done by storing the frontier as a priority queue ordered by g.
- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.
- The first is that the goal test is applied to a node when it is *selected for expansion* rather than when it is first generated.
- The reason is that the first goal node that is *generated* may be on a suboptimal path. The second difference is that a test is added in case a better path is found to a node currently on the frontier.

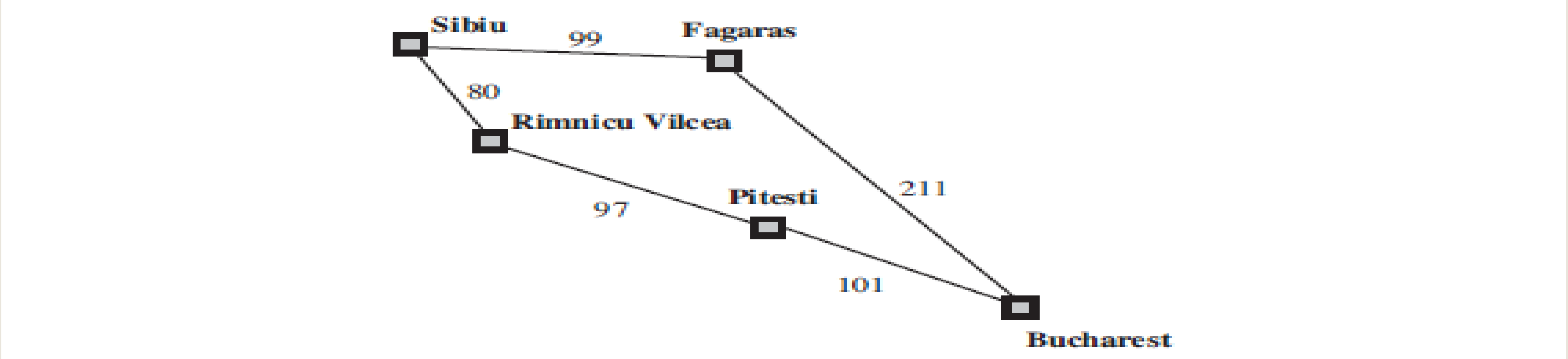


**Figure 3.15** Part of the Romania state space, selected to illustrate uniform-cost search.

- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras,
- so it is expanded, adding Bucharest with cost 99+211=310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.
- It is easy to see that 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.
- 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 does not care about the *number* of steps a path has, but only about their total cost.

## ■ 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.
- whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue.
- 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—which, in turn, was the deepest unexpanded node when it was selected.
- it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

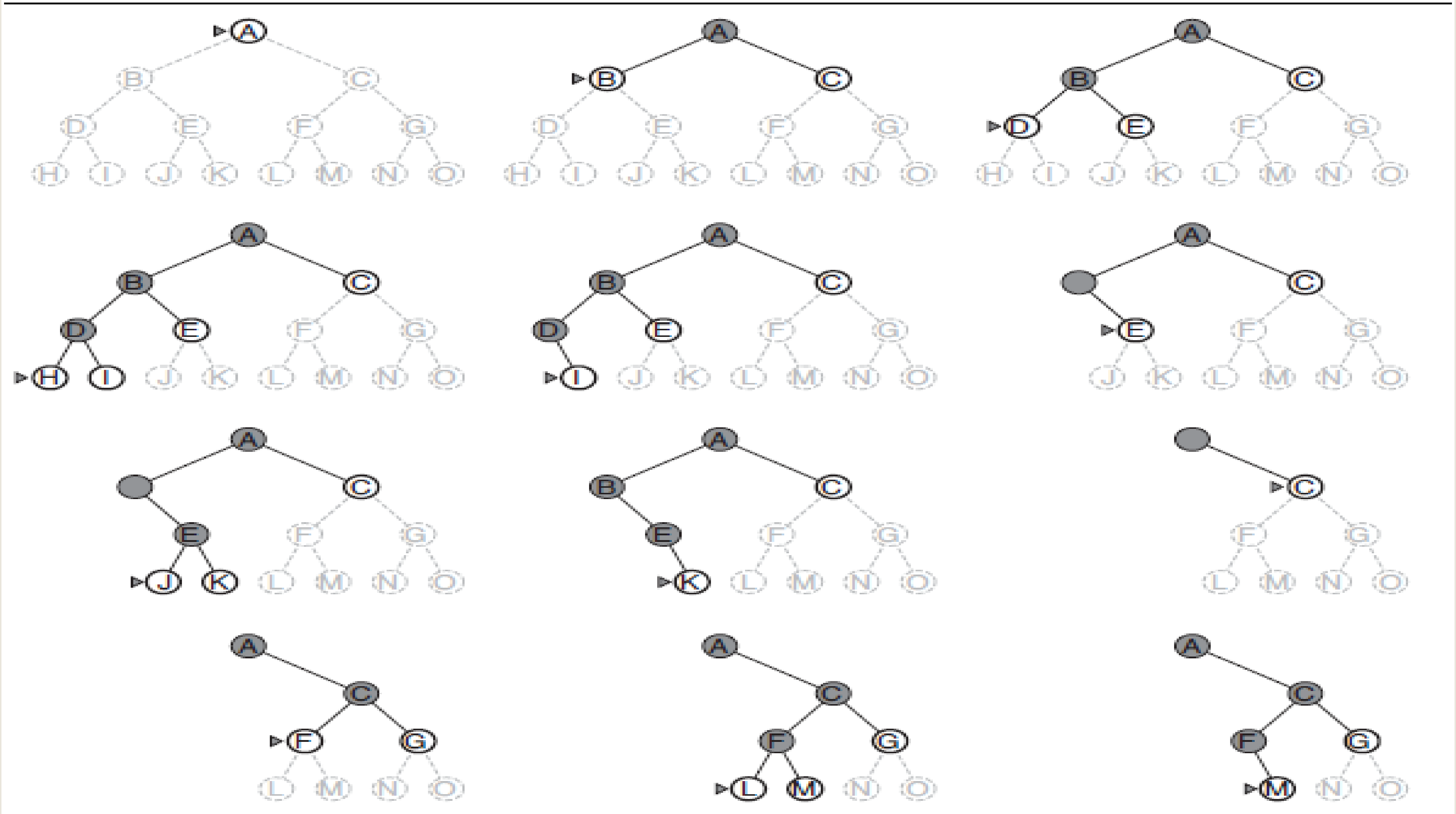


**Figure 3.16** Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and $M$ is the only goal node.

- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
- 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.
- 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.
- Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

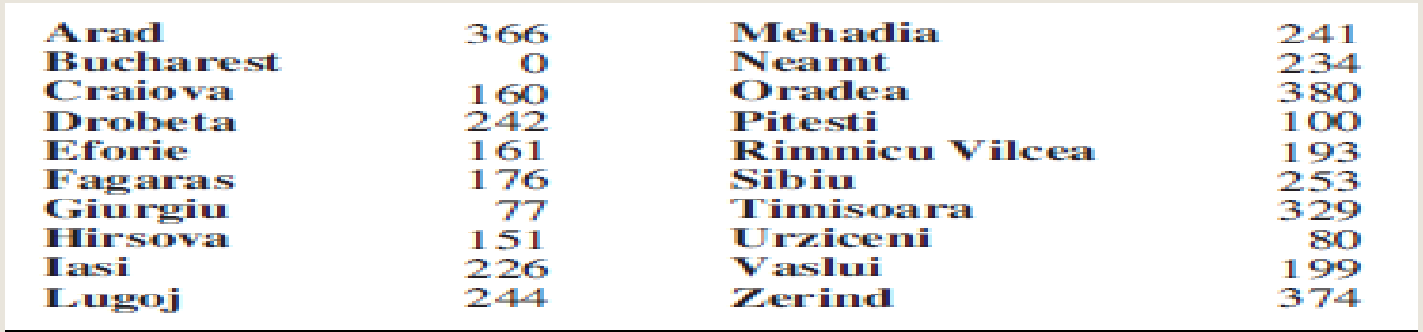
## INFORMED (HEURISTIC) SEARCH STRATEGIES

- an **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, *f(n)*. The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search except for the use of *f* instead of g to order the priority queue.
- The choice of *f* determines the search strategy. Most best-first algorithms include as a component of *f* a **heuristic function**, denoted h(n):

h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

## Greedy best-first search

- **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- it evaluates nodes by using just the heuristic function; that is, f(n) = h(n).
- In **Straight line distance** heuristic, called hSLD, If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure



| | | | |
|---|---|---|---|
| Arad | 366 | Mehadia | 241 |
| Bucharest | 0 | Neamt | 234 |
| Craiova | 160 | Oradea | 380 |
| Drobeta | 242 | Pitesti | 100 |
| Eforie | 161 | Rimnicu Vilcea | 193 |
| Fagaras | 176 | Sibiu | 253 |
| Giurgiu | 77 | Timisoara | 329 |
| Hirsova | 151 | Urziceni | 80 |
| Iasi | 226 | Vaslui | 199 |
| Lugoj | 244 | Zerind | 374 |

Values of $h_{SLD}$—straight-line distances to Bucharest.

- the values of hSLD cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that hSLD is correlated with actual road distances and is, therefore, a useful heuristic.
- Figure shows the progress of a greedy best-first search using hSLD to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest.
- Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.
- It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
- This shows why the algorithm is called “greedy”—at each step it tries to get as close to the goal as it can.

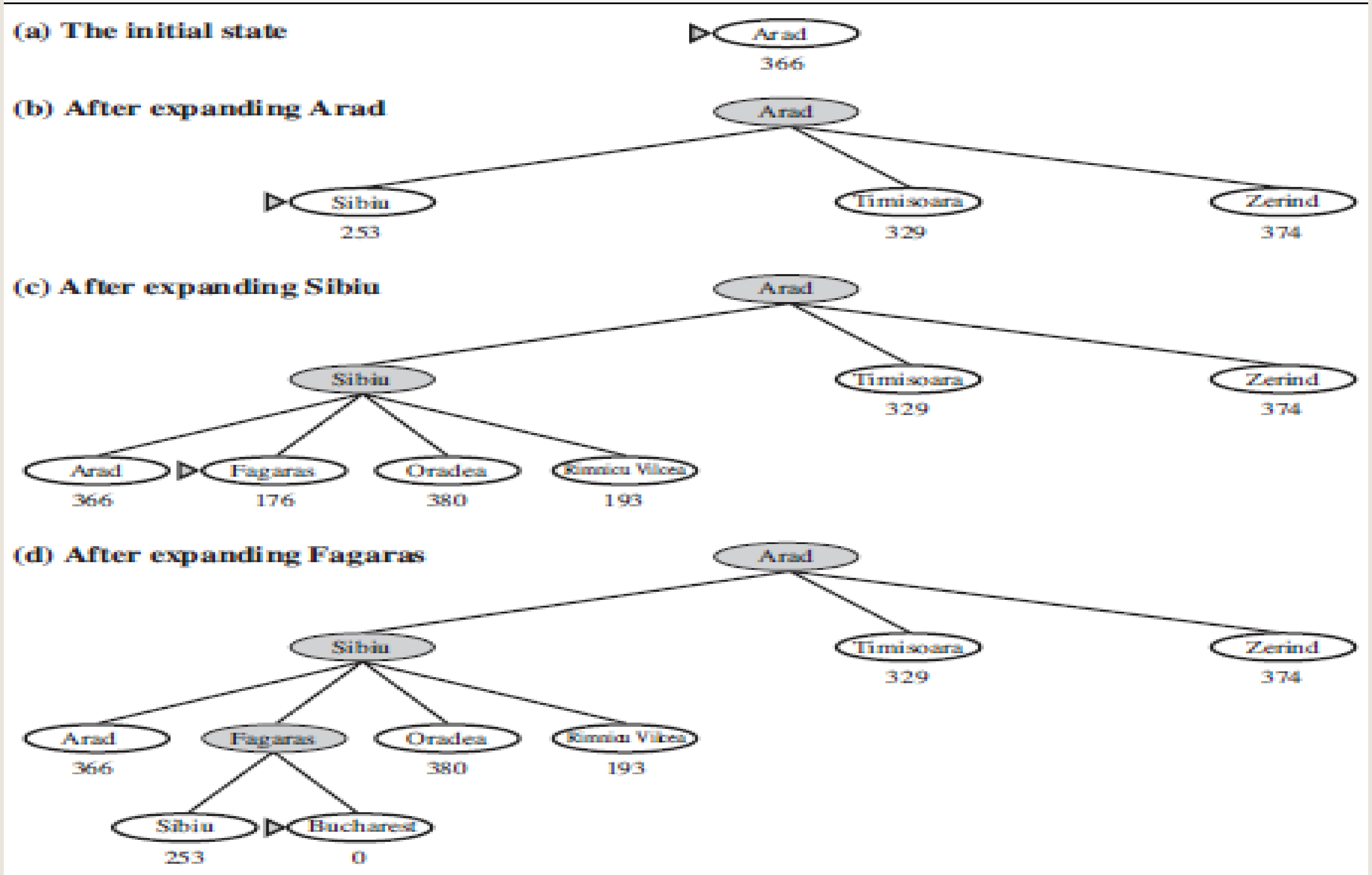


**Figure 3.23** Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their $h$-values.

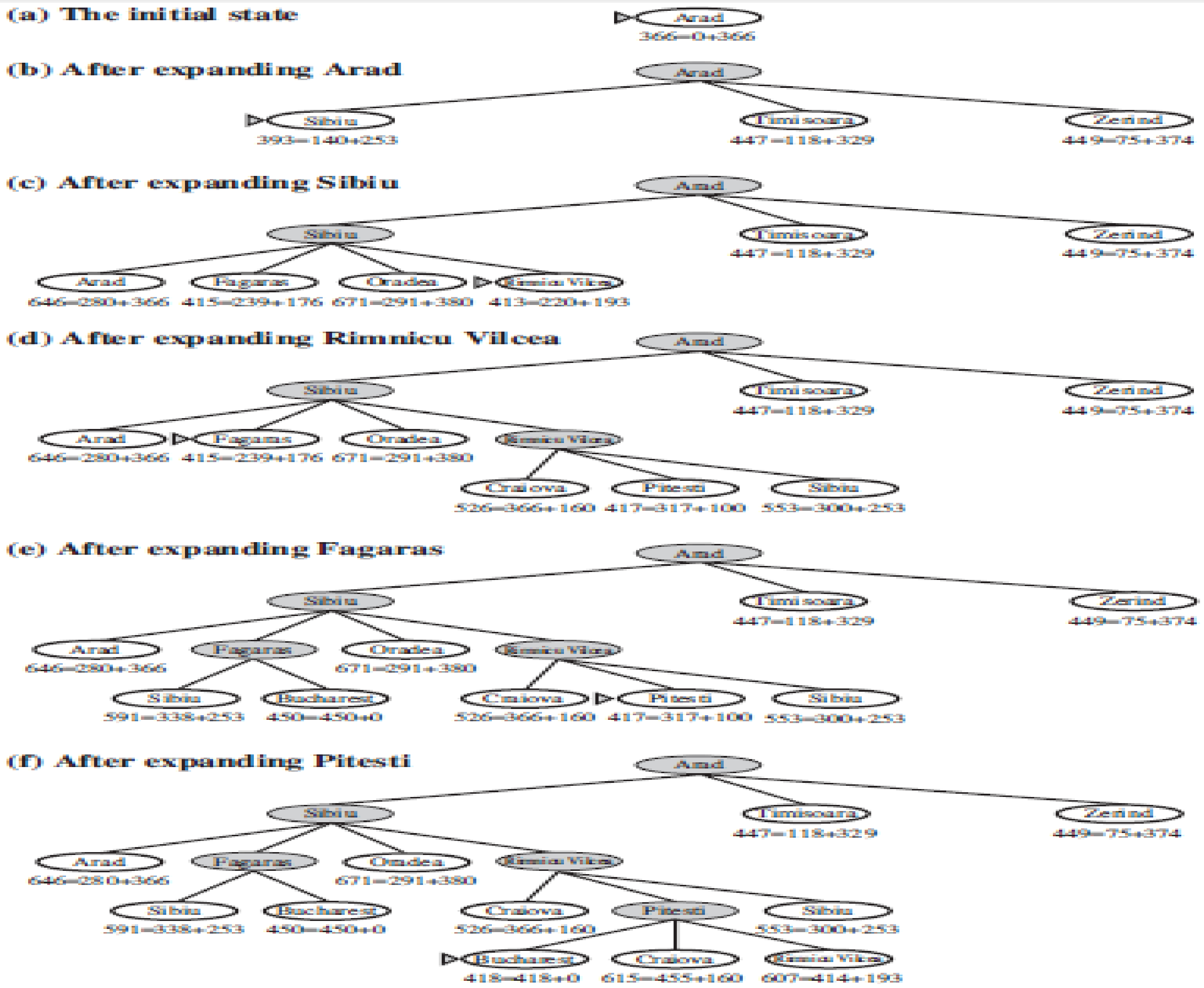
- Greedy best-first tree search is also incomplete even in a finite state space.
- Consider the problem of getting from Iasi to Fagaras. The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end.
- The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras.
- The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop.
- The worst-case time and space complexity for the tree version is $O(b^m)$, where *m* is the maximum depth of the search space.

- **A* search: Minimizing the total estimated solution cost**

- The most widely known form of best-first search is called **A∗ search** . It evaluates nodes by combining *g(n)*, the cost to reach the node, and *h(n)*, the cost to get from the node to the goal:
  - $f(n) = g(n) + h(n)$ .
- Since *g(n)* gives the path cost from the start node to node *n*, and *h(n)* is the estimated cost of the cheapest path from *n* to the goal, we have
  - *f(n)* = estimated cost of the cheapest solution through *n* .
- if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$.
- It turns out that this strategy is more than just reasonable: provided that the heuristic function *h(n)* satisfies certain conditions, A∗ search is both complete and optimal.
- The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses $g + h$ instead of *g*.

## ■ Conditions for optimality: Admissibility and consistency

- The first condition we require for optimality is that *h(n)* be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal.
- Because *g(n)* is the actual cost to reach n along the current path, and *f(n)=g(n) + h(n),*
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.
- An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest.
- Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.
- we show the progress of an A∗ tree search for Bucharest.
- Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417).
- Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

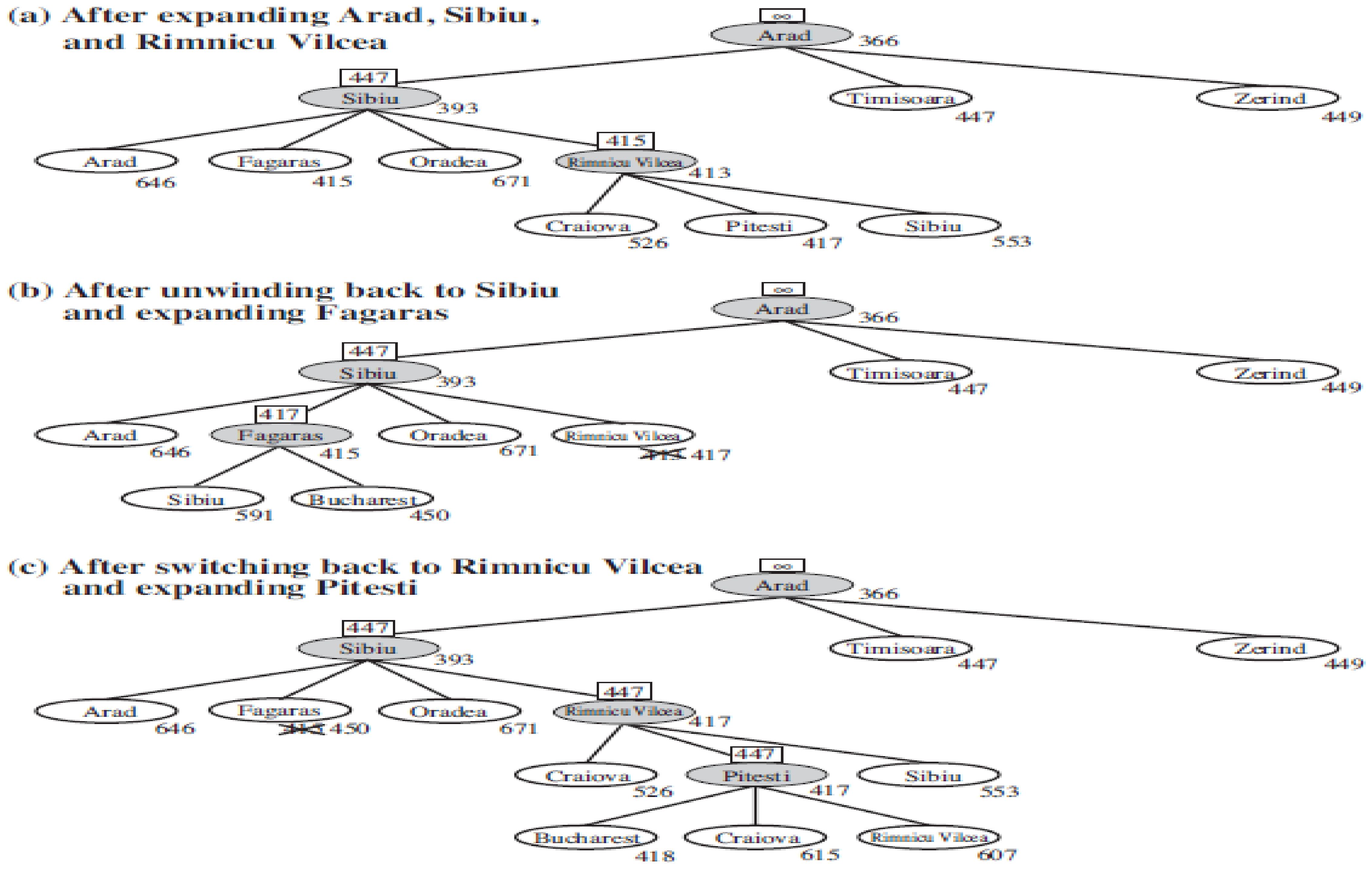

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253
(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

- A second, slightly stronger condition called **consistency** is required only for applications of A∗ to graph search.
- A heuristic *h(n)* is consistent if, for every node n and every successor *n´* of *n* generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to *n´* plus the estimated cost of reaching the goal from n´ :

$$h(n) \leq c(n, a, n') + h(n')$$

## Memory-bounded heuristic search

- memory-bounded algorithms, are RBFS and MA∗.
- **Recursive best-first search** (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.
- it uses the *f_limit* variable to keep track of the *f*-value of the best *alternative* path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the f-value of each node along the path with a **backed-up value**—the best *f*-value of its children.
- In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and can therefore decide whether it’s worth reexpanding the subtree at some later time.


(a) After expanding Arad, Sibiu, and Rimnicu Vilcea
∞
Arad 366
447
Sibiu 393
Timisoara 447
Zerind 449
Arad 646
Fagaras 415
Oradea 671
415
Rimnicu Vilcea 413
Craiova 526
Pitesti 417
Sibiu 553
(b) After unwinding back to Sibiu and expanding Fagaras
∞
Arad 366
447
Sibiu 393
Timisoara 447
Zerind 449
Arad 646
417
Fagaras 415
Oradea 671
Rimnicu Vilcea ~~413~~ 417
Sibiu 591
Bucharest 450
(c) After switching back to Rimnicu Vilcea and expanding Pitesti
∞
Arad 366
447
Sibiu 393
Timisoara 447
Zerind 449
Arad 646
Fagaras ~~415~~ 450
Oradea 671
447
Rimnicu Vilcea 417
Craiova 526
447
Pitesti 417
Sibiu 553
Bucharest 418
Craiova 615
Rimnicu Vilcea 607