北京大学

# 搜索

# Solving Problems by Searching

·AIPKU·

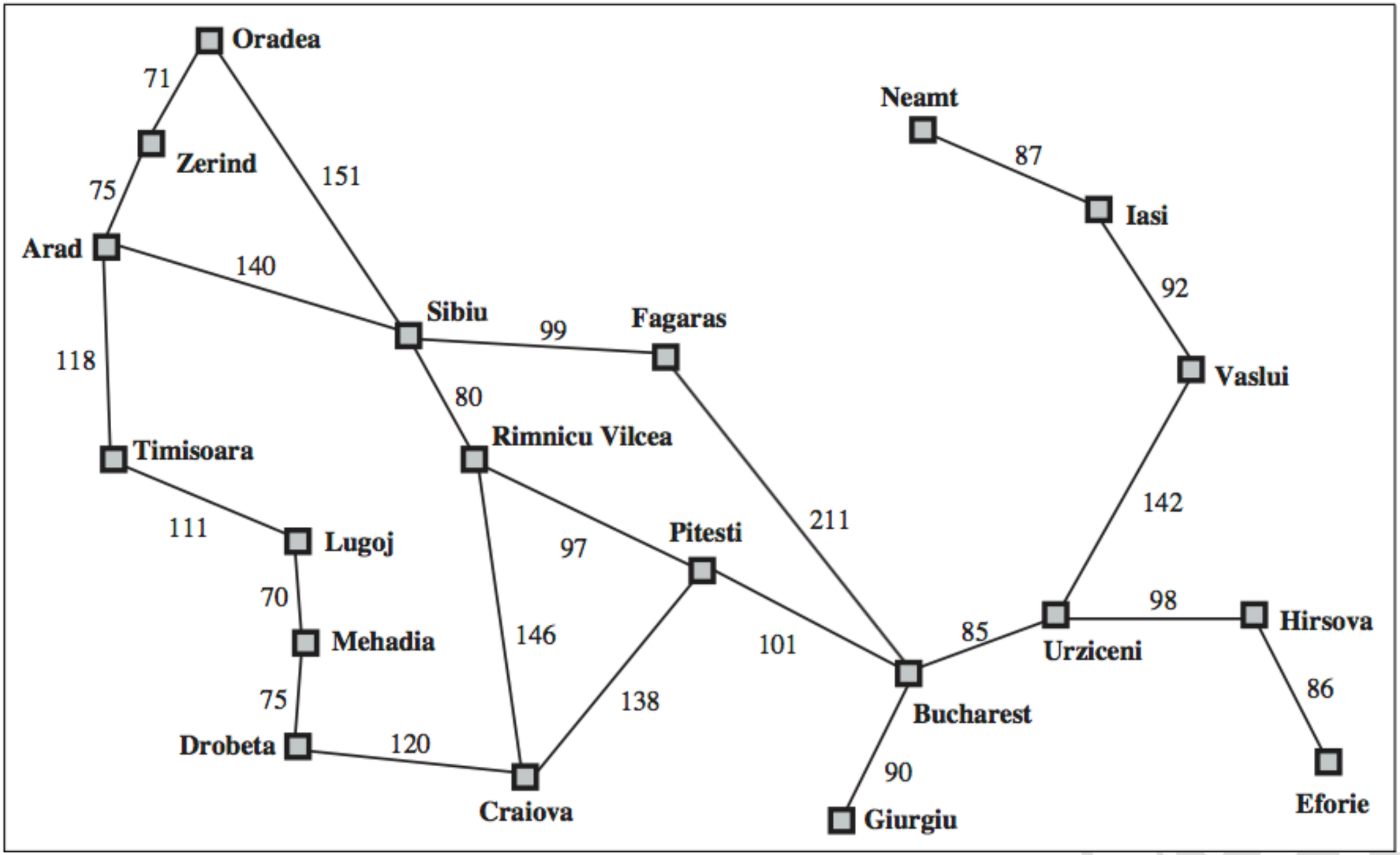
人工智能引论第五课

主讲人：李文新

# 内容提要

- 问题的定义及问题的解
  - 问题描述模型
- 通过搜索对问题求解
  - 搜索树、搜索算法框架、搜索算法的性能评价
- 无信息搜索（盲目搜索）
  - 宽度优先、一致代价、深度优先、深度受限、迭代加深、双向搜索
- 有信息搜索（启发式搜索）
  - 贪婪最佳优先搜索、A* 搜索、启发式函数

# 寻径问题 – 寻找从Arad 到 Bucharest的路径




- 简化版罗马尼亚地图

# 问题描述

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

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

- 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)}.

# 问题描述

- 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. 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. (The map of Romania shown in Figure 3.2 can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each direction.)

- A **path** in the state space is a sequence of states connected by a sequence of actions.

# 问题描述

- 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)}.

- Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called “checkmate,” where the opponent’ s king is under attack and can’ t escape.

# 问题描述

- 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.

- In this chapter, we assume that the cost of a path can be described as the *sum* of the costs of the individual actions along the path. The **step cost** of taking action **a** in state **s** to reach state s′ is denoted by **c(s, a, s′ ).** The step costs for Romania are shown in Figure 3.2 as route distances. We assume that step costs are **nonnegative**.

- The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm.

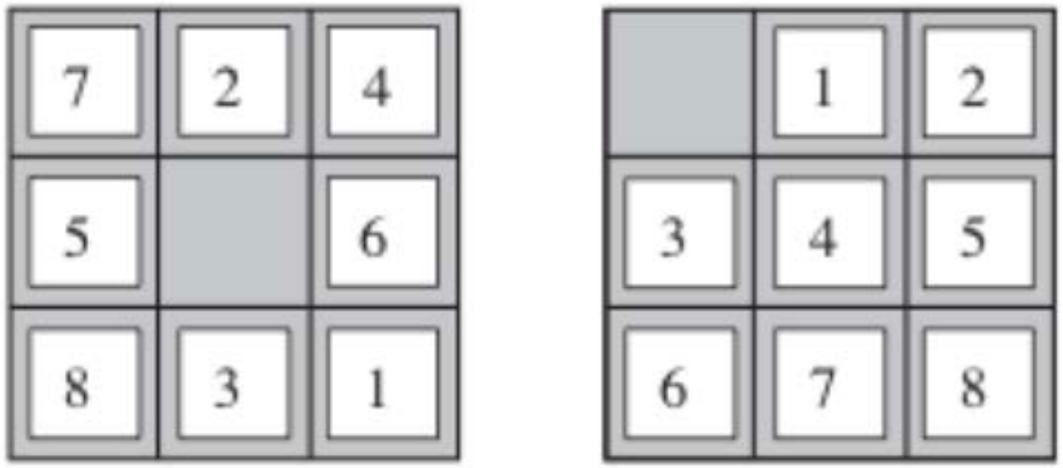
- A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

# 问题描述模型

- **States**:
- **Initial state**:
- **Actions**:
- **Transition model**:
- **Goal test**:
- **Path cost**:

# 八数码问题

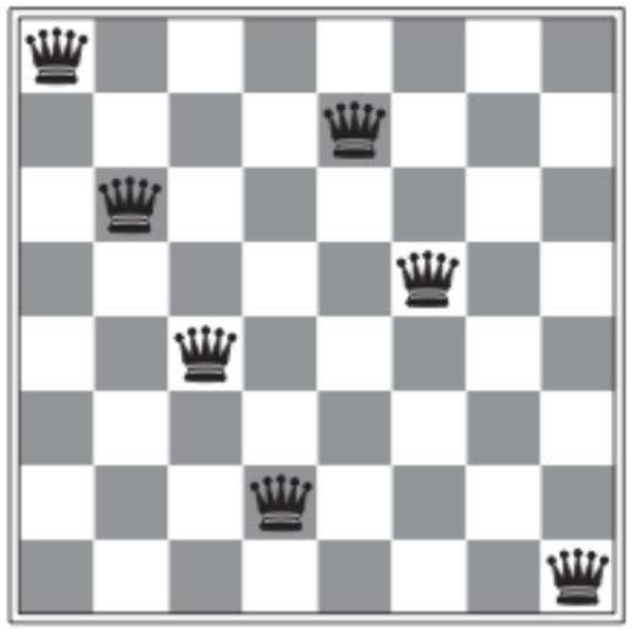
- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.4).
- **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.




八数码问题

# 八皇后问题

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



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

# 八皇后问题的另一种表述

- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ 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 ($0 \leq n \leq 8$), 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 $1.8 \times 10^{14}$ to just 2,057, and solutions are easy to find. On the other hand, for 100 queens the reduction is from roughly $10^{400}$ states to about $10^{52}$ states (Exercise 3.5)—a big improvement, but not enough to make the problem tractable. Section 4.1 describes the complete-state formulation, and Chapter 6 gives a simple algorithm that solves even the million-queens problem with ease.

# 内容提要 - 通过搜索对问题求解

- 搜索树
  - 父节点、子节点、扩展、开节点集、闭节点集、探索集、搜索策略
- 搜索算法基础
- 问题求解算法的性能
  - 完备性、最优性、时间复杂度、空间复杂度

# 通过搜索求解问题

- 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. 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. (Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad." ) 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 父节点** *In(Arad)* leading to three new **child nodes 子节点** : *In(Sibiu), In(Timisoara),* and *In(Zerind)*. Now we must choose which of these three possibilities to consider further.

# 搜索树

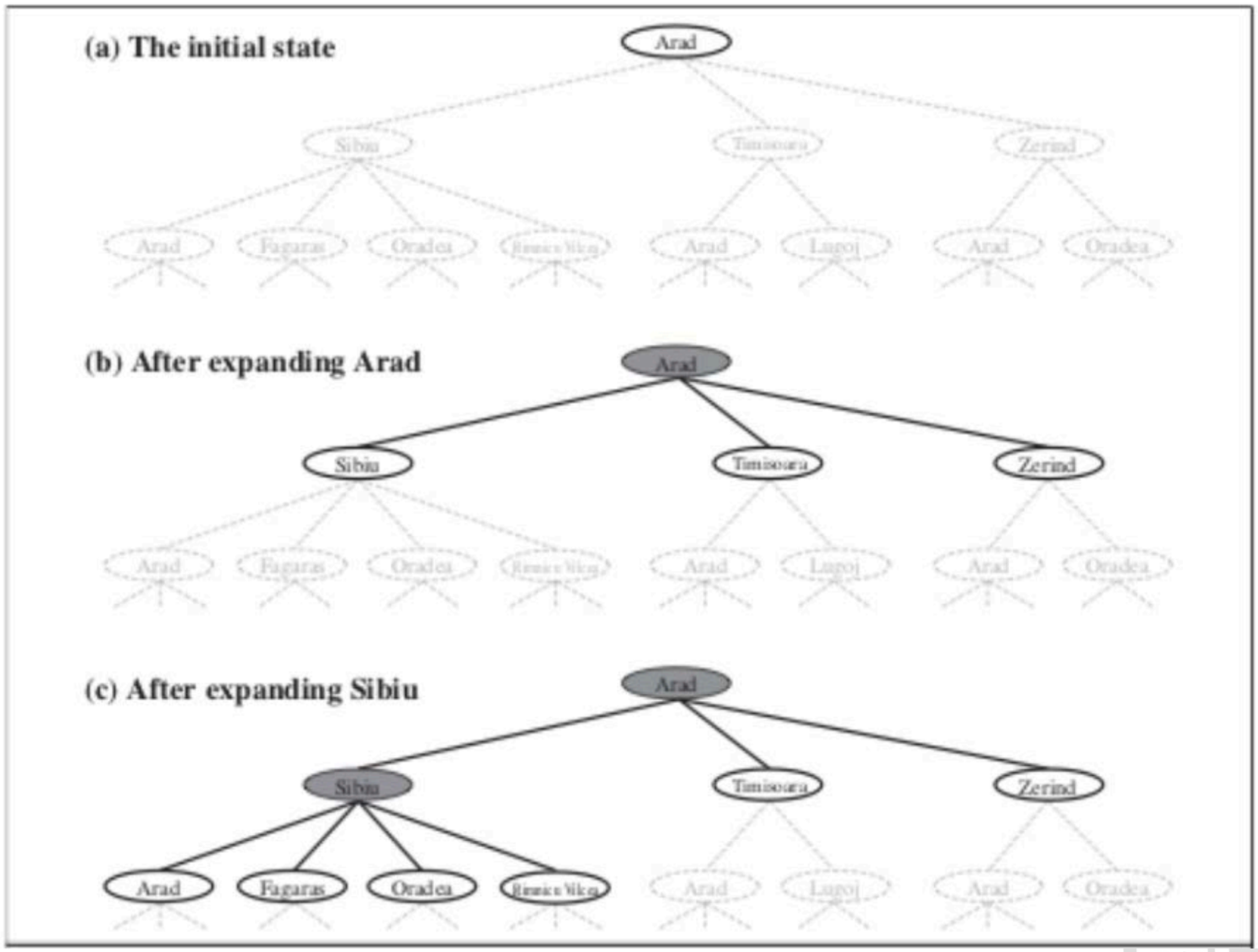


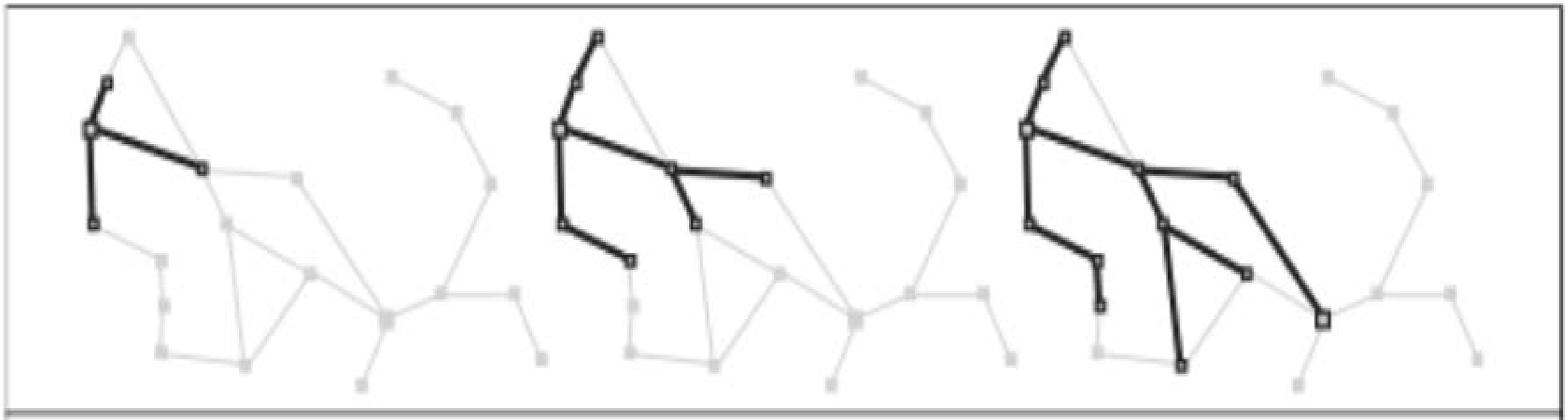

Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

# 搜索树上的几个概念

- Each of these six nodes is a **leaf node** **叶节点**, 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**. Many authors call it the **open list**, **开节点集**

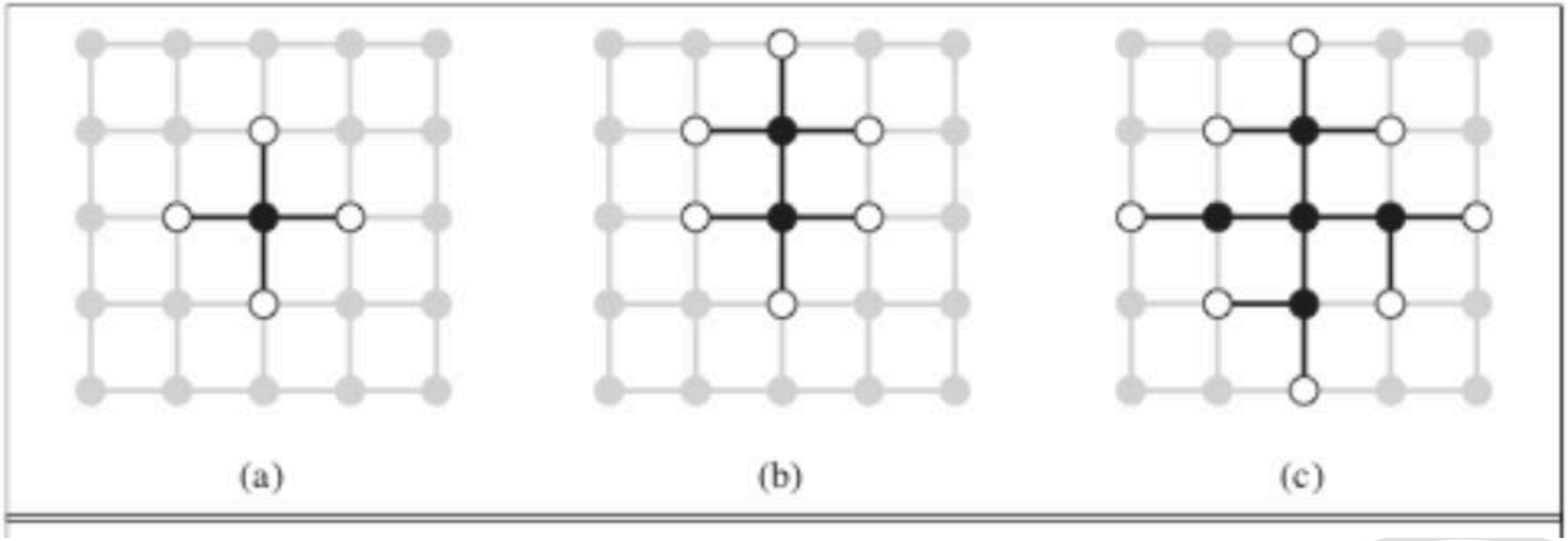
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. how they choose which state to expand next—the so-called **search strategy**. **搜索策略**

# 重复节点



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.

# 图搜索问题




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.

# 记录已访问过的节点

- As the saying goes, *algorithms that forget their history are doomed to repeat it.* The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set 探索集**（ also known as the **closed list 闭节点表** ）, which remembers every expanded node. Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be dis-carded instead of being added to the frontier. The new algorithm, called GRAPH-SEARCH, is shown informally in Figure 3.7. The specific algorithms in this chapter draw on this general design.

- The general TREE-SEARCH algorithm is shown informally in Figure 3.7. Search algorithms all share this basic structure;

**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

树搜索

---

**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***

图搜索

**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.

# 搜索算法的一般存储框架

Search algorithms require a data structure to keep track of the search tree that is being con- structed. 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.

```
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)
```

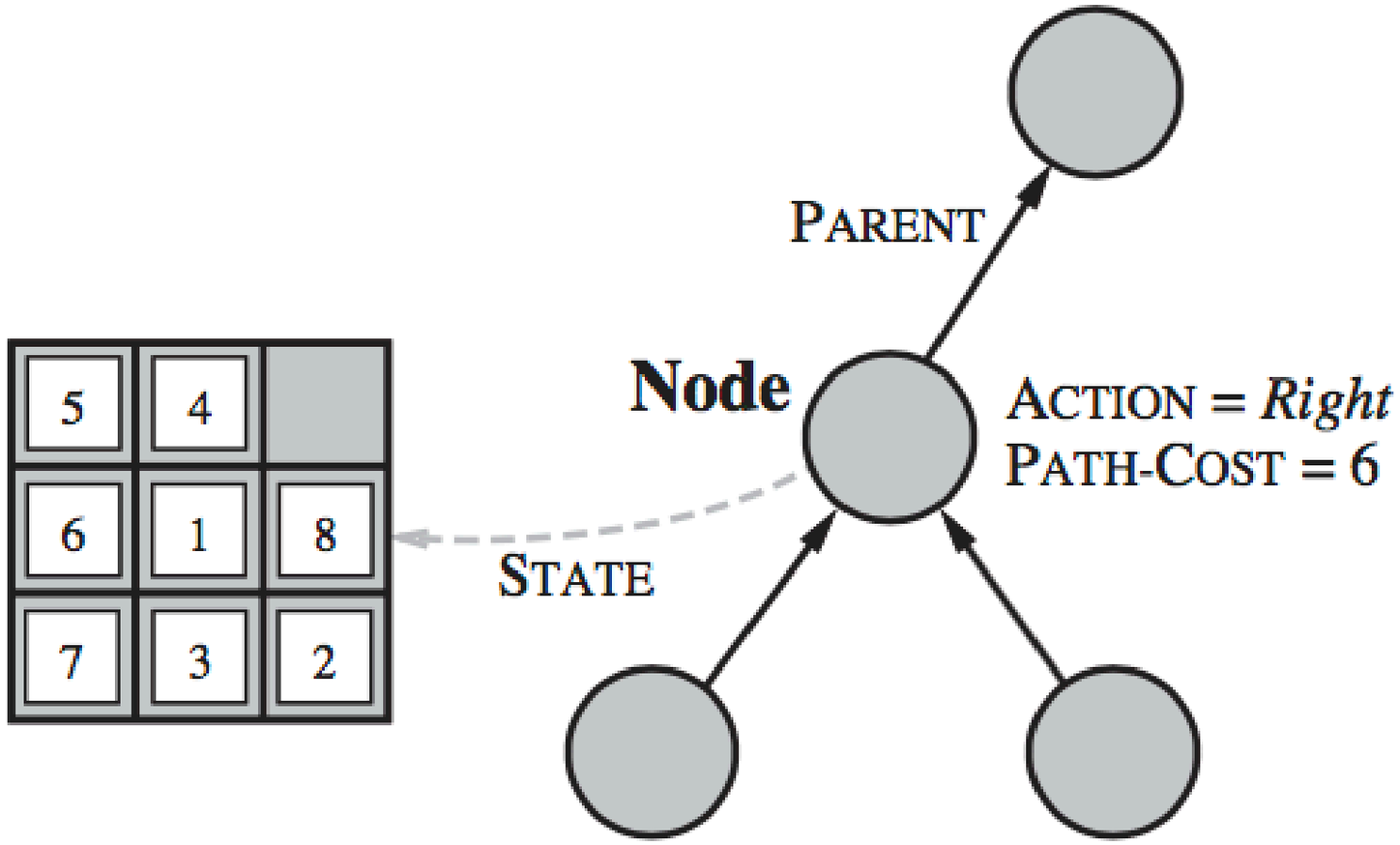



**Figure 3.10** Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

# nodes and states

- Up to now, we have not been very careful to distinguish between **nodes** and **states**, but in writing detailed algorithms it’ s important to make that distinction.

- 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. Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.

# 用队列存储开节点集

Now that we have nodes, we need somewhere to put them. 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 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.

# 用哈希表存储闭节点集 便于查找

- The **explored set** can be implemented with **a hash table** to allow efficient checking for repeated states. With a good implementation, insertion and lookup can be done in roughly constant time no matter how many states are stored.
- **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.
- **{Bucharest,Urziceni,Vaslui} is the same as {Urziceni,Vaslui,Bucharest}**. logically equivalent states should map to the same data structure. In the case of states described by sets, for example, a **bit-vector** representation or a **sorted list** without repetition would be canonical, whereas **an unsorted list would not**.

# Measuring problem-solving

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

**如何评价搜索算法？**

# 问题难度

- Time and space complexity are always considered with respect to some measure of the **problem difficulty**.

- In theoretical computer science, the typical 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). **图问题的规模**

- This is appropriate when the graph is an explicit data structure that is input to the search program. (The map of Romania is an example of this.)

# 搜索树的规模

- 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 (i.e., the number of steps along the path from the root); and
  - **m**, the maximum length of any path in the state space. Time is often measured in terms of the number of nodes generated during the search, and
- **space** in terms of the maximum number of nodes stored in memory.
- For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how "redundant" the paths in the state space are.

# 评估搜索算法的效率

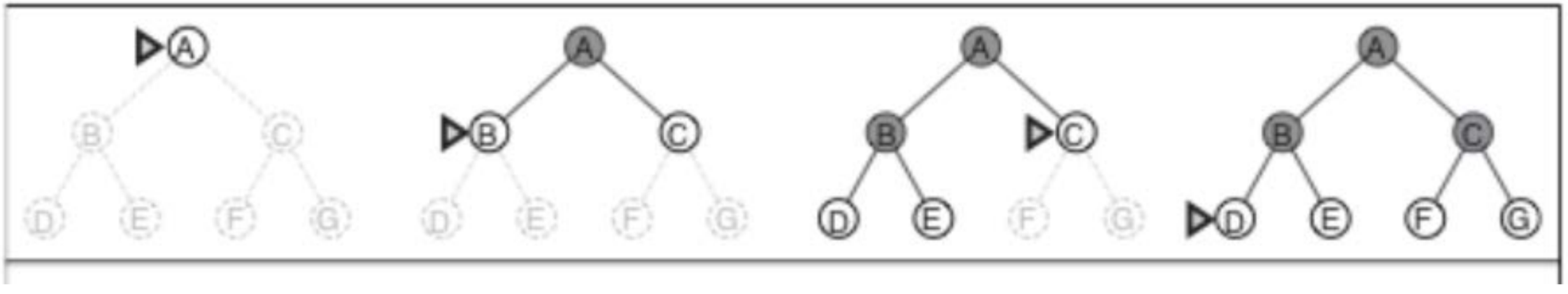
- To assess the effectiveness of a search algorithm, we can consider just the **search cost**— which typically depends on the time complexity but can also include a term for memory usage—or we can use the **total cost**, which combines the **search cost** and the **path cost** of the solution found.
- For the problem of finding a route from Arad to Bucharest, the **search cost** is the amount of time taken by the search and the s**olution cost** is the total length of the path in kilometers.
- Thus, to compute the total cost, we have to add milliseconds and kilometers. There is no "official exchange rate" between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car's average speed (because time is what the agent cares about). This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive. The more general problem of tradeoffs between different goods is taken up in Chapter 16.

# 内容提要 - 无信息搜索（盲目搜索）

- 宽度优先
- 一致代价
- 深度优先
- 深度受限
- 迭代加深
- 双向搜索

# 内容提要 - 无信息搜索 - 宽度优先

- 宽度优先




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

In the worst case, it is the last node generated at that level. Then the total number of nodes generated is $b+b^2+b^3+\cdots+b^d=O(b^d)$.

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 $O(b^d)$.

# 内容提要 - 无信息搜索 - 宽度优先

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Breadth-first search on a graph

| 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.

- **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, 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.*

# 内容提要 - 无信息搜索 - 一致代价

When all **step costs are equal**, breadth-first search is **optimal** because it always expands the *shallowest* unexpanded node.

By a simple extension, we can find an algorithm that is **optimal** with **any step-cost function**. 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. The algorithm is shown in Figure 3.14.

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.

```
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
```

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.

# 内容提要 - 无信息搜索 - 一致代价

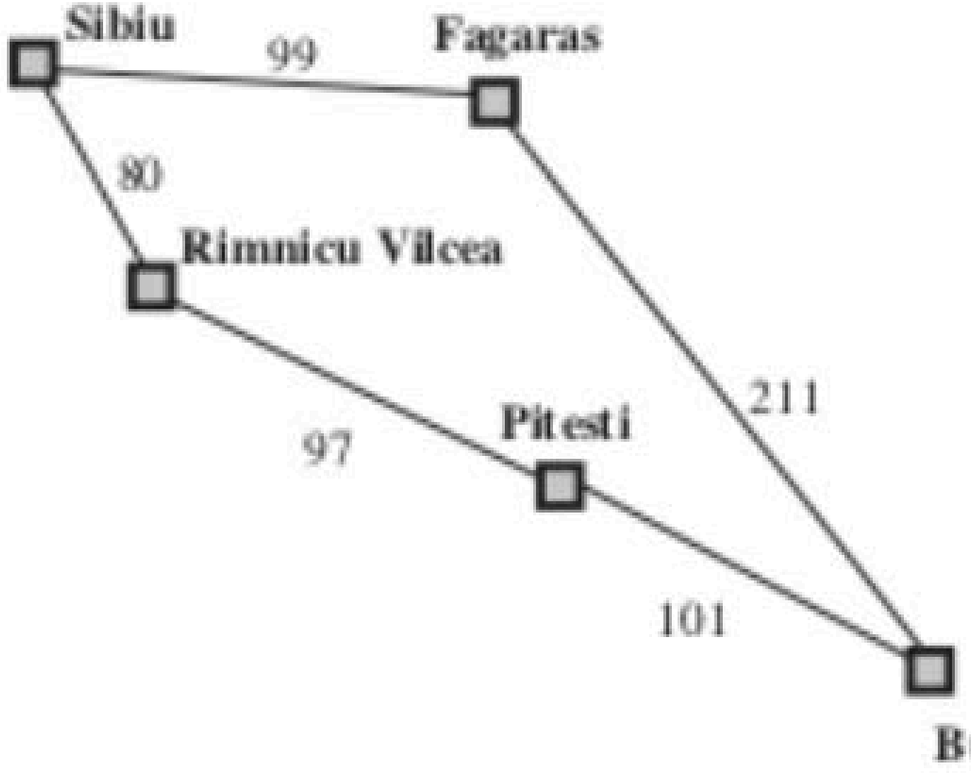



**Figure 3.15** Part of the Romania state space, selected to illustrate uniform-cost search.

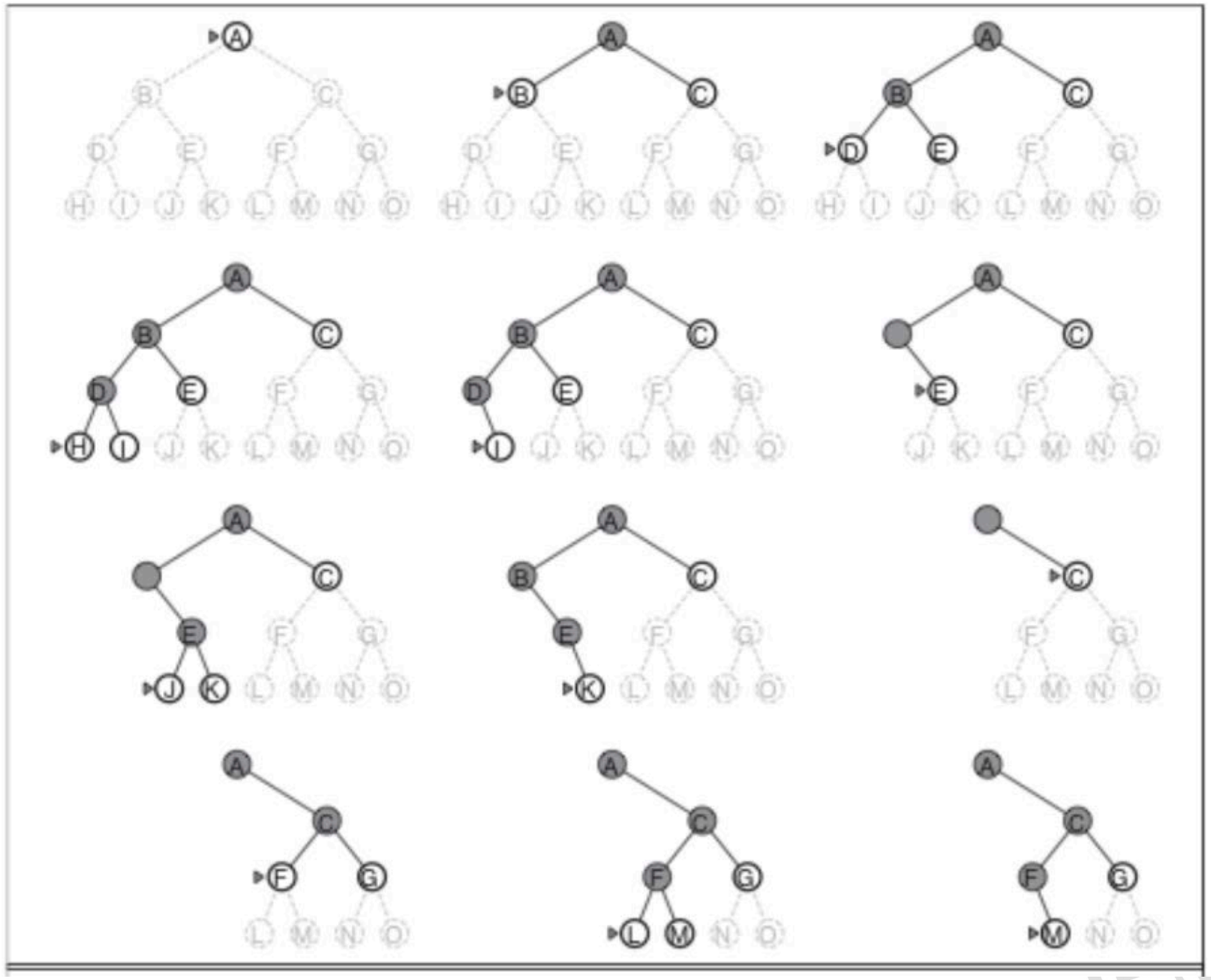
80 + 97 = 177 , 99 + 211 = 310 , 80 + 97 + 101 = 278.

Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

# 内容提要 - 无信息搜索 - 一致代价

- 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 $O(\mathbf{b}^{1+\lfloor C^*/\varepsilon \rfloor})$, which can be much greater than $\mathbf{b}^{\mathbf{d}}$. This is because uniform- cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, $\mathbf{b}^{1+\lfloor C^*/\varepsilon \rfloor}$ is just $\mathbf{b}^{\mathbf{d+1}}$. When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal’ s depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

# 内容提要 - 无信息搜索 - 深度优先



二叉树上的深度优先搜索

# 内容提要 - 无信息搜索 - 深度优先

- The **depth-first search** algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. **A LIFO** queue **栈** 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.

- As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a **recursive** function **递归调用** that calls itself on each of its children in turn.

# 内容提要 - 无信息搜索 - **深度优先**

- So far, depth-first search seems to have **no** clear **advantage** over breadth-first search, so why do we include it? The reason is the **space complexity**. 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.

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

- This has led to the adoption of depth-first tree search as the **basic workhorse of many areas of AI**,

# 内容提要 - 无信息搜索 - 深度受限

- The embarrassing 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**. The depth limit solves the infinite-path problem. 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 $O(\mathbf{b}^{\mathbf{l}})$ and its space complexity is $O(\mathbf{bl})$.
- Depth-first search can be viewed as a special case of depth-limited search with **l = ∞**.

```
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.

- 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 Figure 3.17. 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.**

# 内容提要 - 无信息搜索 - **迭代加深**

- **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.**

```
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.

# 内容提要 - 无信息搜索 - **迭代加深**

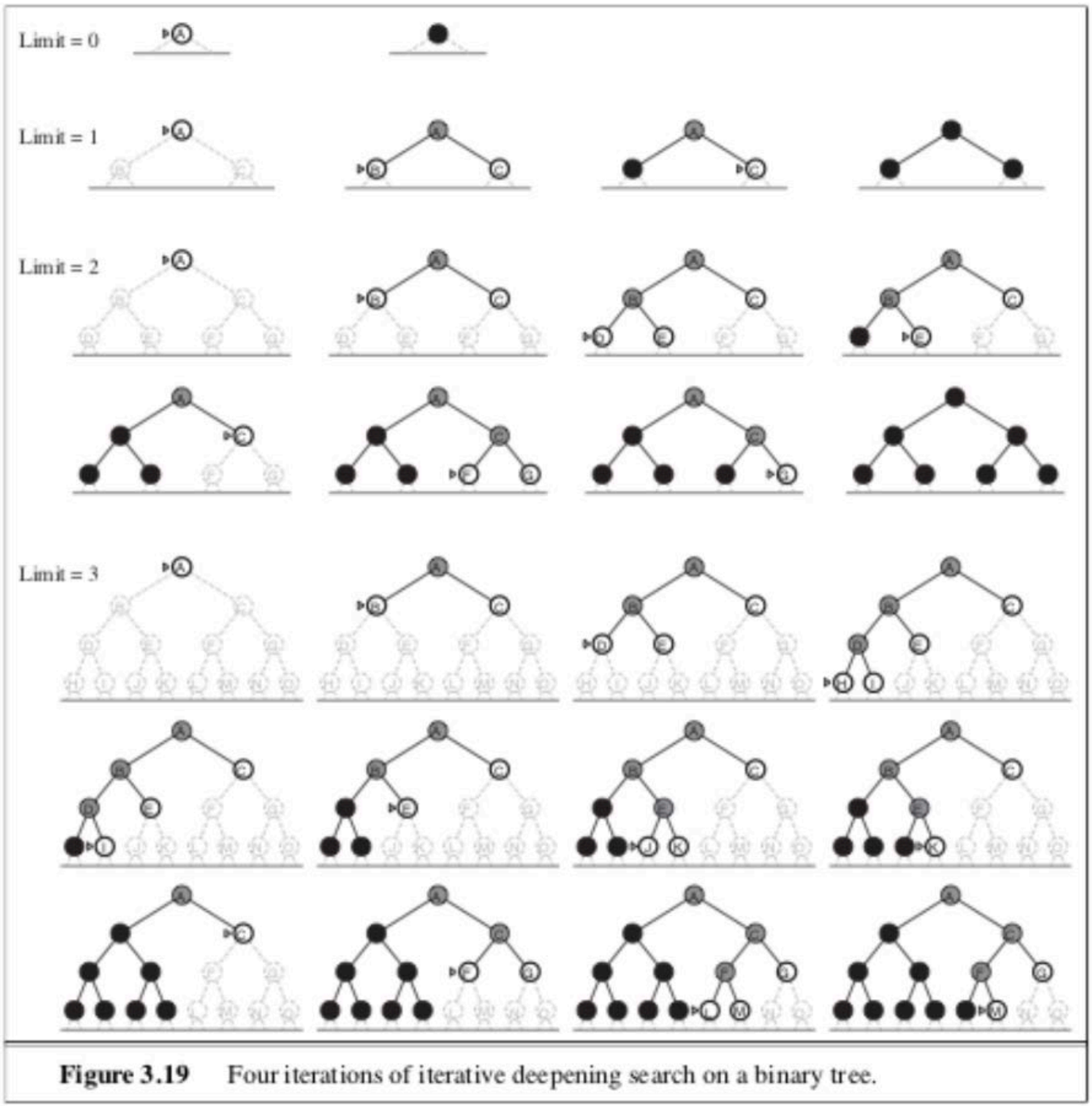



**Figure 3.19** Four iterations of iterative deepening search on a binary tree.

- *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.*

# 内容提要 - 无信息搜索 - **双向搜索**

- 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。 The motivation is that $\mathbf{b^{d/2} + b^{d/2}}$ is much less than $b^d$。 Thus, the time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$. The space complexity is also $\mathbf{O(b^{d/2})}$.

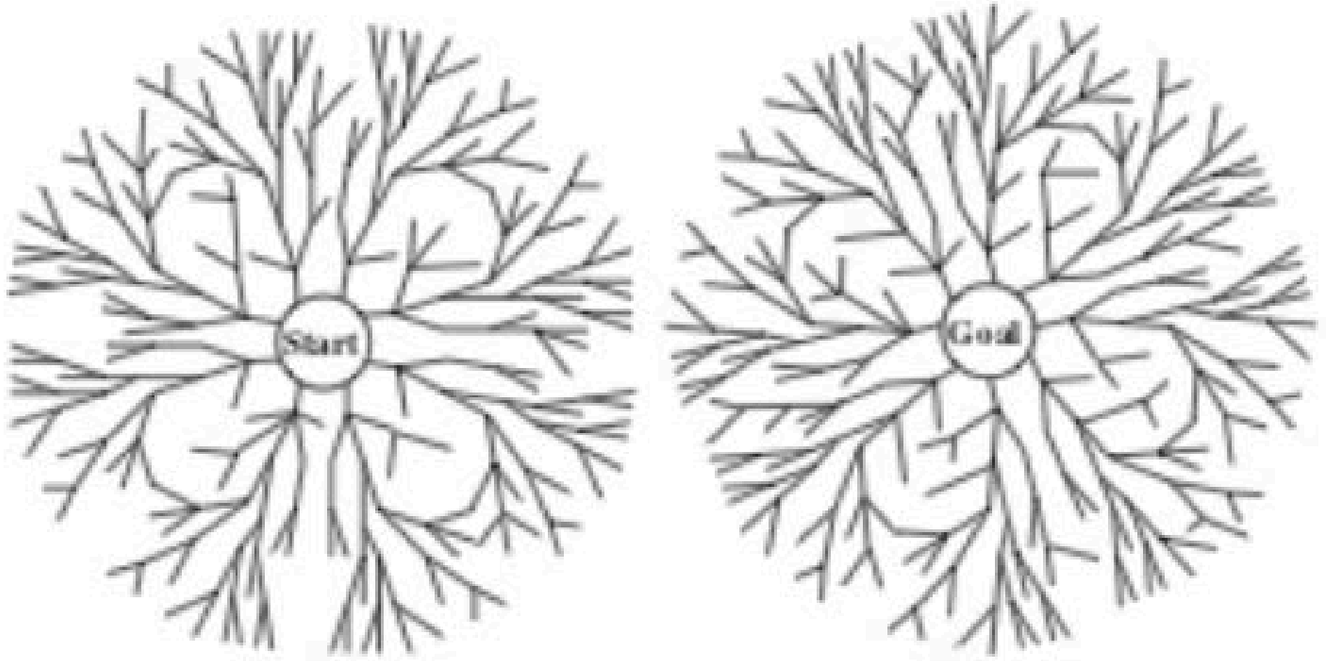



**Figure 3.20** A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

# 内容提要 - 无信息搜索 – 比较

| 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 $\epsilon$; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

# 内容提要 – 有信息搜索（启发式搜索）

- 贪婪最佳优先搜索
- A* 搜索：缩小总评估代价
- 存储受限的启发式搜索
- 学习以促搜索
- 启发式函数

# 内容提要 – 有信息搜索（启发式搜索）

- This section shows how 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.
- The general approach we consider is called **best-first search**. 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.

# 内容提要 – 有信息搜索 - 贪婪最佳优先搜索

- 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** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, ***f (n) = h(n).***

# Arad – Sibiu – Fagras - Bucharest

| | | | |
|---|---|---|---|
| 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 |

**Figure 3.22** Values of $h_{SLD}$—straight-line distances to Bucharest.

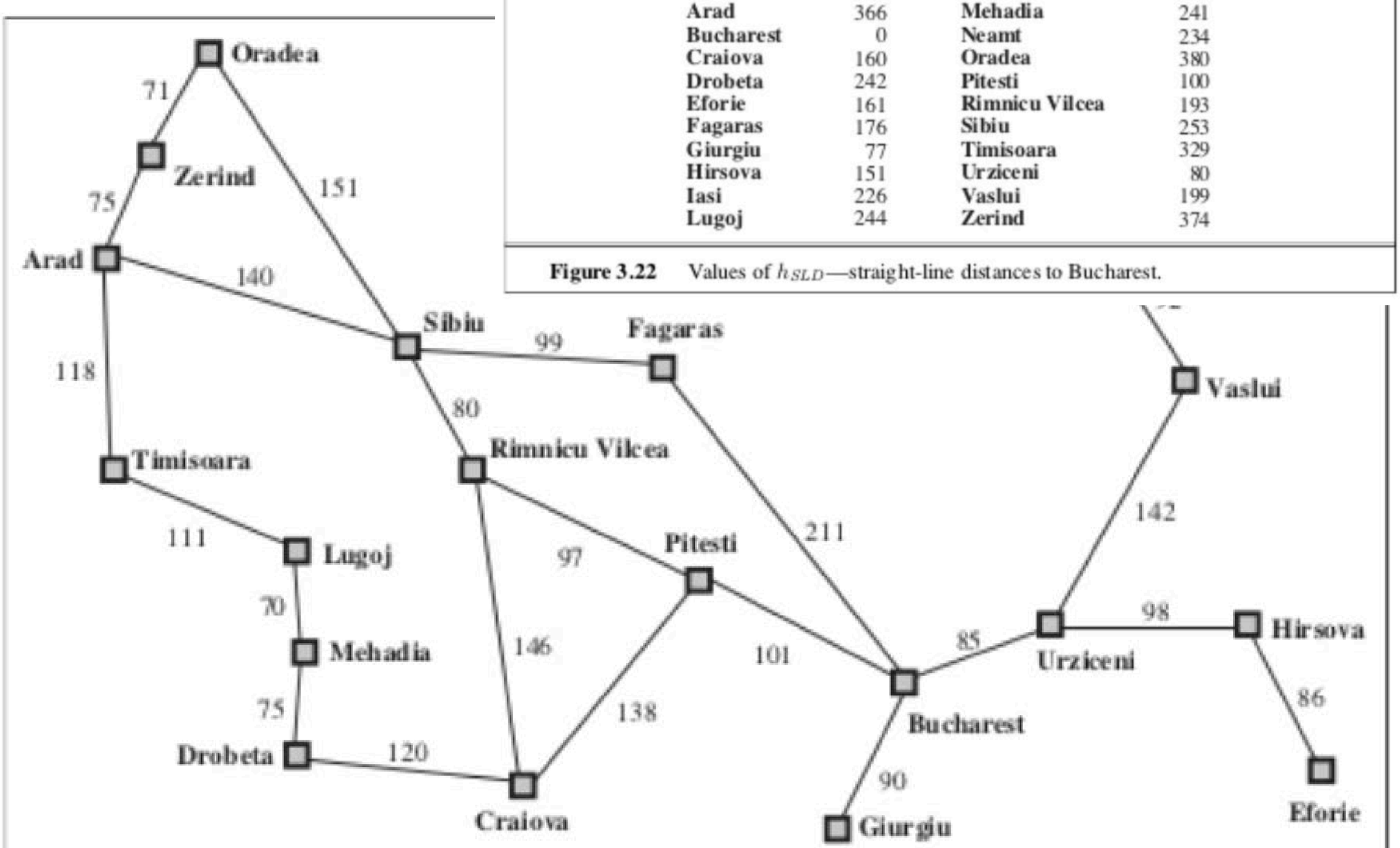



**Figure 3.2** A simplified road map of part of Romania.

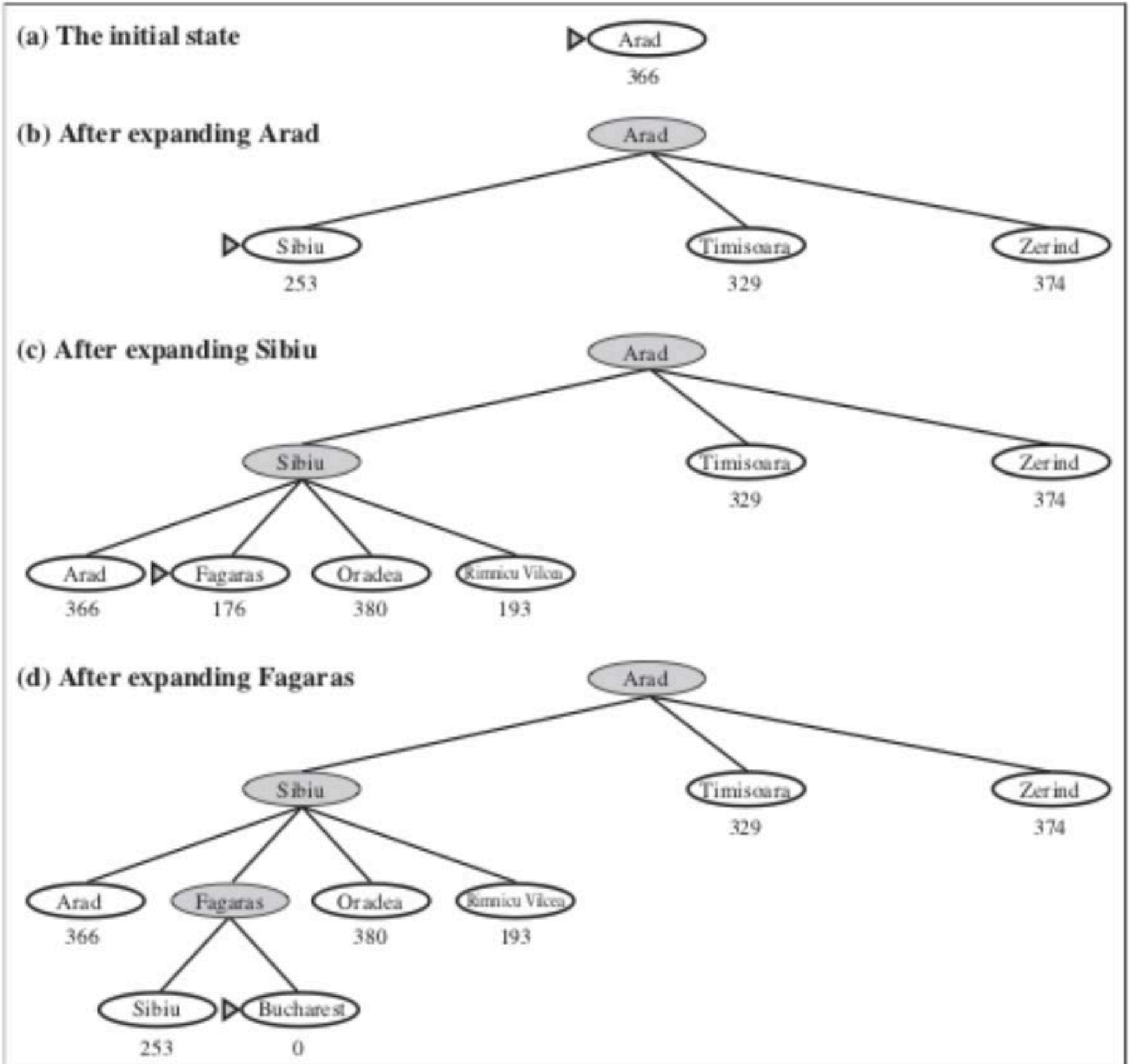



**Figure 3.23** Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their $h$-values.

# 贪心 未必 最优

- For this particular problem, greedy best-first search using $h_{SLD}$ 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.

# 贪心 可能 不完备

- **Greedy best-first tree search is also incomplete** even in a finite state space, much like depth-first search.
- 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. **陷入死循环**
- 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. With a good heuristic function, however, the complexity can be reduced substantially. **The amount of the reduction depends on the particular problem and on the quality of the heuristic.**

# 内容提要 – 有信息搜索 - A* 搜索

- A* 搜索：缩小总评估代价

- The most widely known form of best-first search is called **A* search** (pronounced “A-star 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 .

# 一定条件下，A*算法可以完备且最优

- Thus, 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 $g$.

# 可采纳性和一致性

- **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)***, we have as an immediate consequence that *f(n)* never overestimates the true cost of a solution along the current path through *n*.

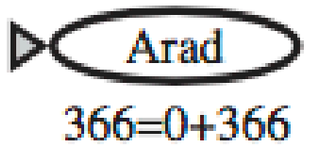
# 以直线距离估计到目的地的路程

- **Admissible heuristics** are by nature optimistic because they think the cost of solving the problem is less than it actually is.

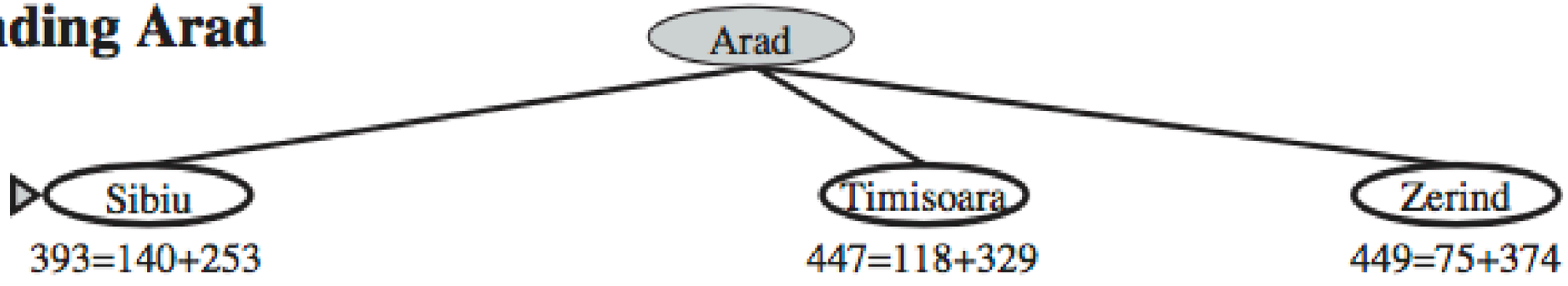
- An obvious example of an admissible heuristic is the straight-line distance $h_{SLD}$ that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate. In Figure 3.24, we show the progress of an A* tree search for Bucharest. The values of ***g*** are computed from the step costs in Figure 3.2, and the values of $h_{SLD}$ are given in Figure 3.22. 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*算法展开过程

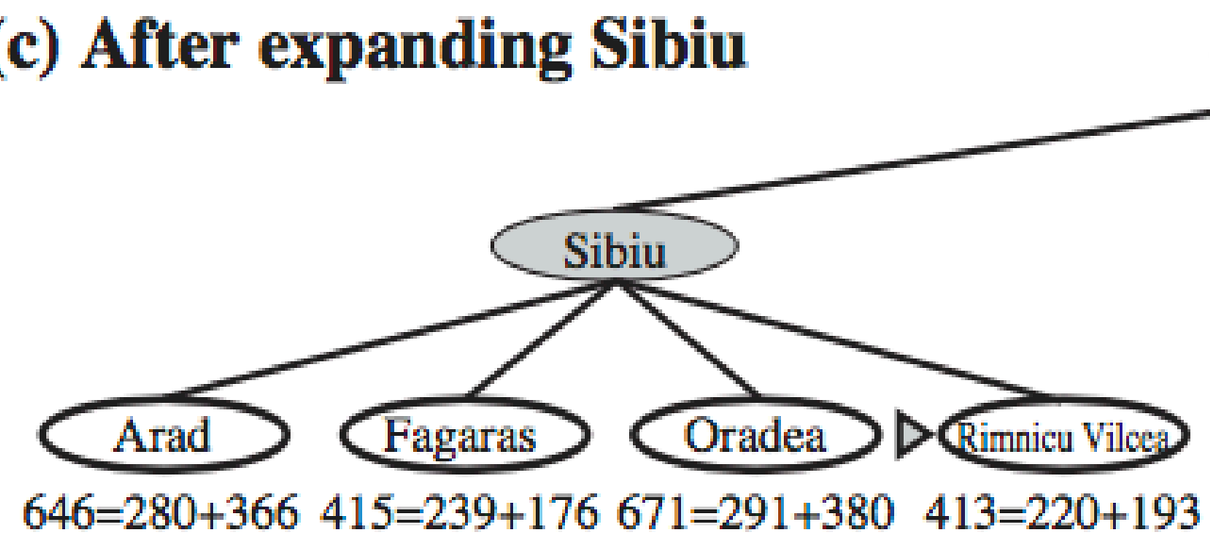
**(a) The initial state**




**(b) After expanding Arad**

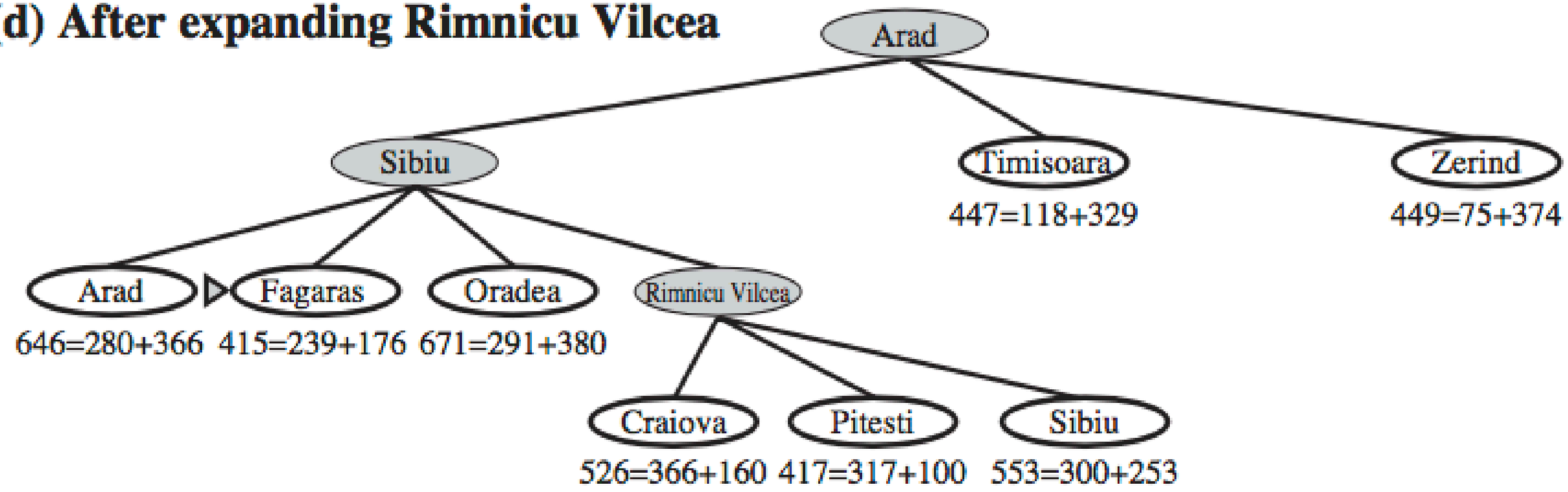



**(c) After expanding Sibiu**

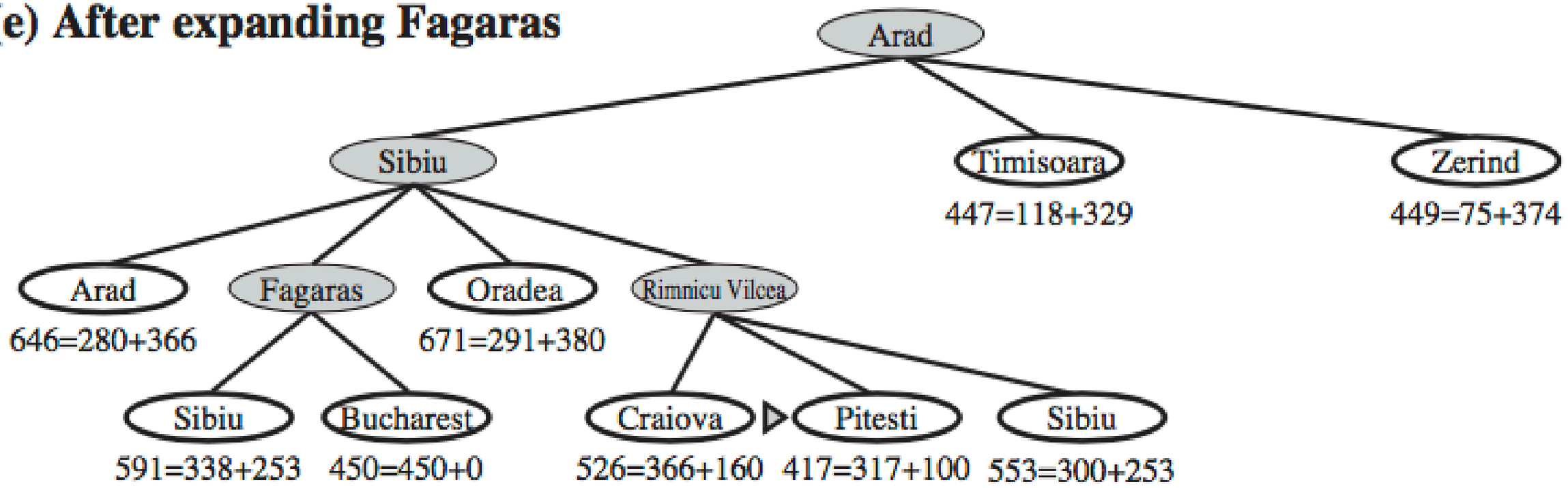


# A*算法展开过程

**(d) After expanding Rimnicu Vilcea**




**(e) After expanding Fagaras**



# A*算法展开过程

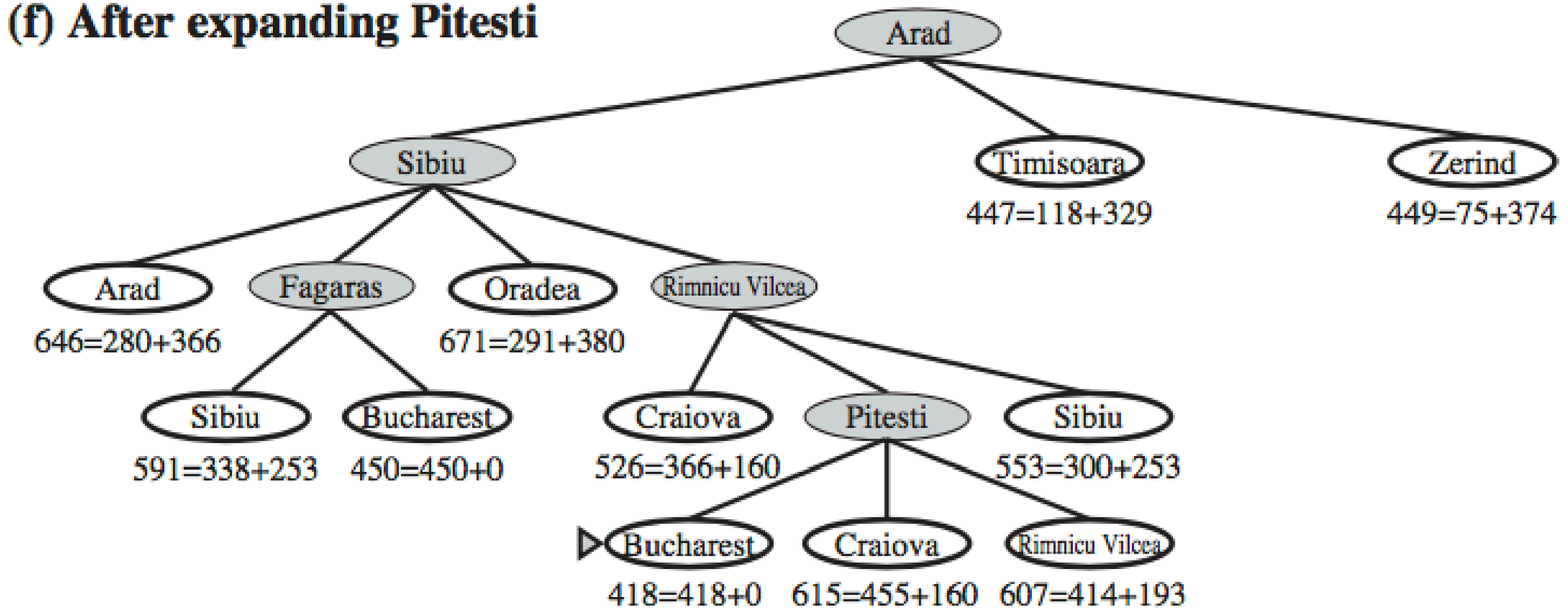



**Figure 3.24** Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The $h$ values are the straight-line distances to Bucharest taken from Figure 3.22.

# 一致性

- A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) 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'***:

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

- This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by ***n, n', ***and the goal ***Gn*** closest to ***n***. For an admissible heuristic, the inequality makes perfect sense: if there were a route from ***n*** to ***Gn*** via ***n'*** that was cheaper than ***h(n)***, that would violate the property that ***h(n)*** is a lower bound on the cost to reach ***Gn***.

# every consistent heuristic is also admissible

- It is fairly easy to show that every consistent heuristic is also admissible**. Consistency is therefore a stricter** requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, $h_{SLD}$. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between $n$ and $n'$ is no greater than $c(n,a,n')$. Hence, $h_{SLD}$ is a consistent heuristic.

# A*算法的最优性

- **Optimality of A***

- As we mentioned earlier, A* has the following properties: *the tree-search version of* A* *is optimal if* ***h(n)*** *is admissible, while the graph-search version is optimal if* ***h(n)*** *is consistent.*

- *if* ***h(n)*** *is consistent, then the values of* ***f(n)*** *along any path are nondecreasing.* The proof follows directly from the definition of consistency. Suppose ***n'*** is a successor of ***n***; then ***g(n') = g(n) + c(n, a, n')*** for some action ***a***, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n) .$$

# A*算法的最优性证明

- *whenever **A*** 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, because f is nondecreasing along any path, n′ would have lower f-cost than n and would have been selected first.

- for expansion must be an optimal solution because ***f*** is the true cost for goal nodes (which have h = 0) and all later goal nodes will be at least as expensive.

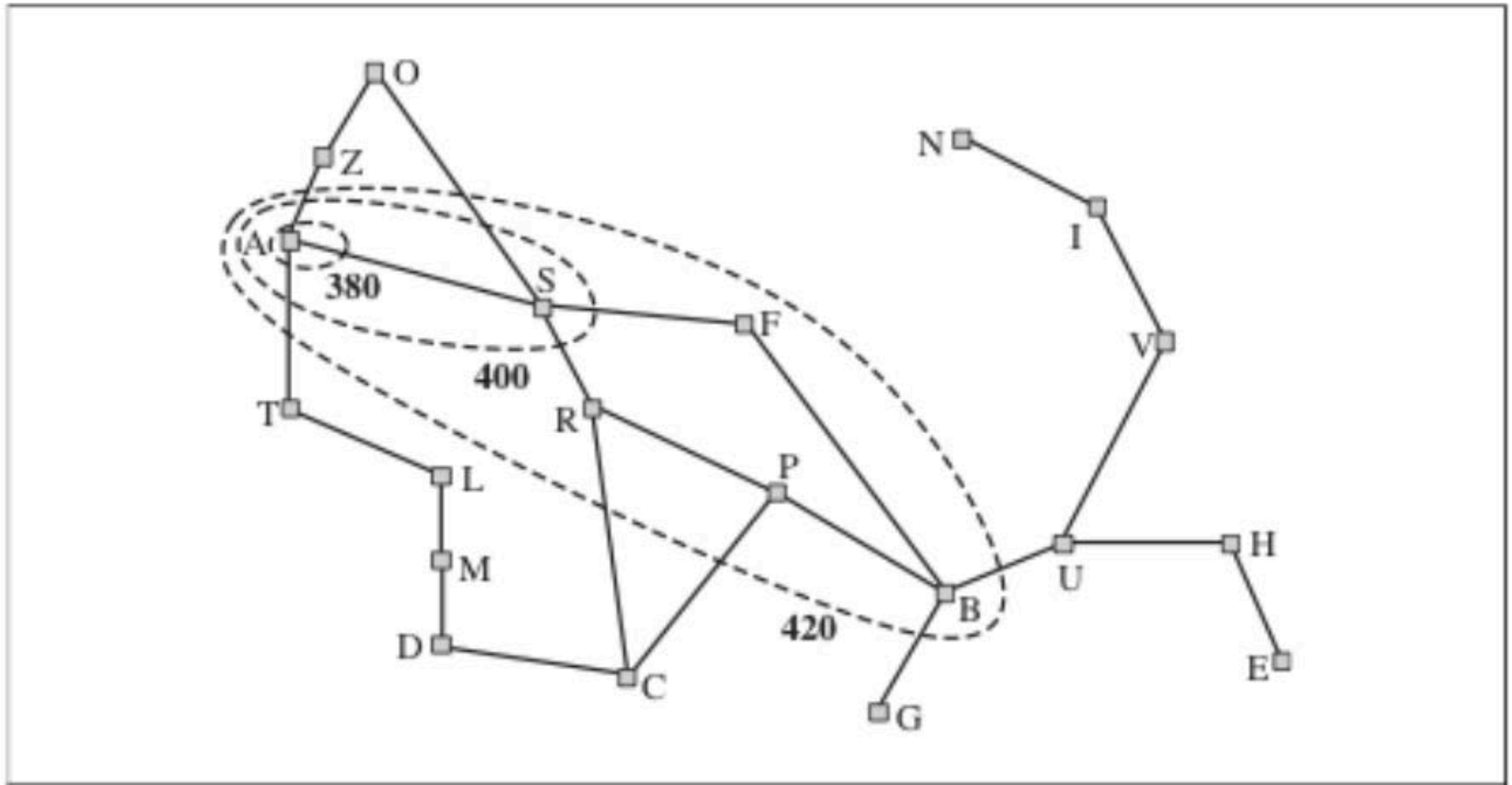



**Figure 3.25** Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f$-costs less than or equal to the contour value.

If $C^*$ is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with $f(n) < C^*$.
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$)

before selecting a goal node.

# optimally efficient

- Completeness requires that there be only finitely many nodes with cost less than or equal to $\mathbf{C}^*$, a condition that is true if all step costs exceed some finite ε and if $b$ is finite.

- The concept of **pruning**—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

- One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root and use the same heuristic information—A* is **optimally efficient** for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with f(n)=C*). This is because any algorithm that *does not* expand all nodes with f(n) < C* runs the risk of missing the optimal solution.

# absolute error or relative error

- That **A*** search is **complete, optimal, and optimally efficient** among all such algorithms is rather satisfying.

- Unfortunately, it does not mean that **A*** is the answer to all our searching needs. The catch is that, for most problems, the **number of states within the goal contour search space is still exponential in the length of the solution**. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth **d** is analyzed in terms of the the **absolute error** or the **relative error** of the heuristic.

- The absolute error is defined as $\Delta \equiv h^* - h$, where $h^*$ is the actual cost of getting from the root to the goal, and the relative error is defined as $\varepsilon \equiv (h^* - h)/h^*$ .

# A*算法的复杂度分析

- The **complexity** results depend very strongly on the assumptions made about the **state space**.

- The **simplest model** studied is a state space that **has a single goal** and is **essentially a tree** with **reversible actions**. (The 8-puzzle satisfies the first and third of these assumptions.) In this case, the time complexity of **A*** is exponential in the maximum absolute error, that is, $\mathbf{O(b^{\Delta})}$. For constant step costs, we can write this as $\mathbf{O(b^{\varepsilon d})}$, where **d** is the solution depth.

- For almost all heuristics in practical use, the **absolute error** is at least proportional to the path cost $h^*$, so **ε** is constant or growing and the time complexity is exponential in **d**.

- We can also see the effect of a more accurate heuristic: $\mathbf{O(b^{\varepsilon d}) = O((b^{\varepsilon})^{d})}$, so the effective branching factor (defined more formally in the next section) is $\mathbf{b^{\varepsilon}}$.

# 出于现实考虑，次优或减少存储

- The complexity of **A*** often makes it impractical to insist on finding an optimal solution. One can use variants of **A*** that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.

- Computation time is not, however, **A***’ s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), **A*** usually runs out of space long before it runs out of time. For this reason, **A*** is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

# 内容提要 – 有信息搜索 - 存储受限的启发式

- **iterative-deepening A* (IDA*)**

- The simplest way to reduce memory requirements for **A*** is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening A* (IDA*)** algorithm.

- **f-cost (g + h)** --- the depth;

- at each iteration, the cutoff value is the smallest **f-cost** of any node that exceeded the cutoff on the previous iteration.

- **IDA*** is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

# Recursive best-first search (RBFS)

- **Recursive best-first search** (RBFS)

- mimic the operation of standard best-first search, but using only linear space.

- Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, 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.

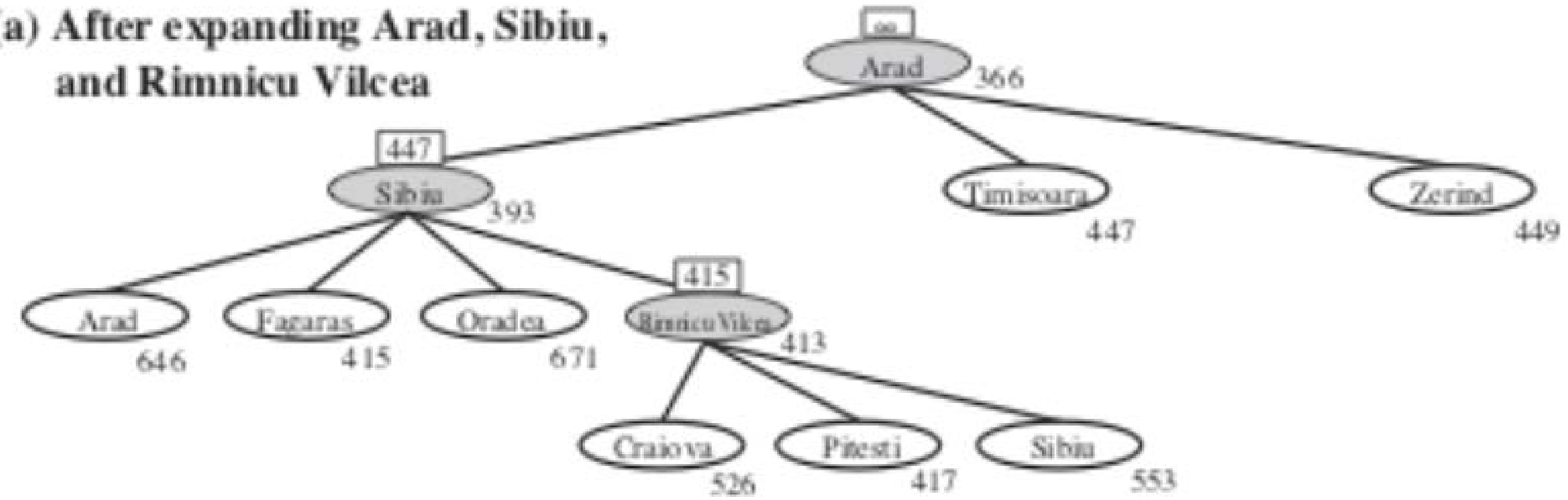
# RBFS

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

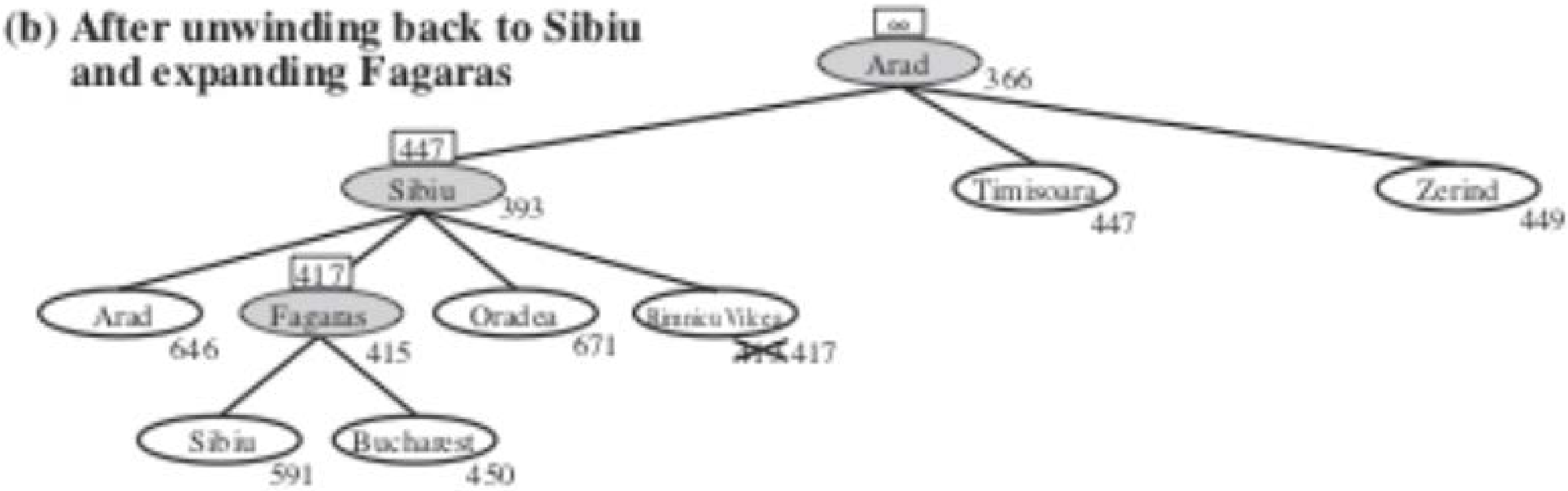
**Figure 3.26** The algorithm for recursive best-first search.

# RBFS 举例 1/3



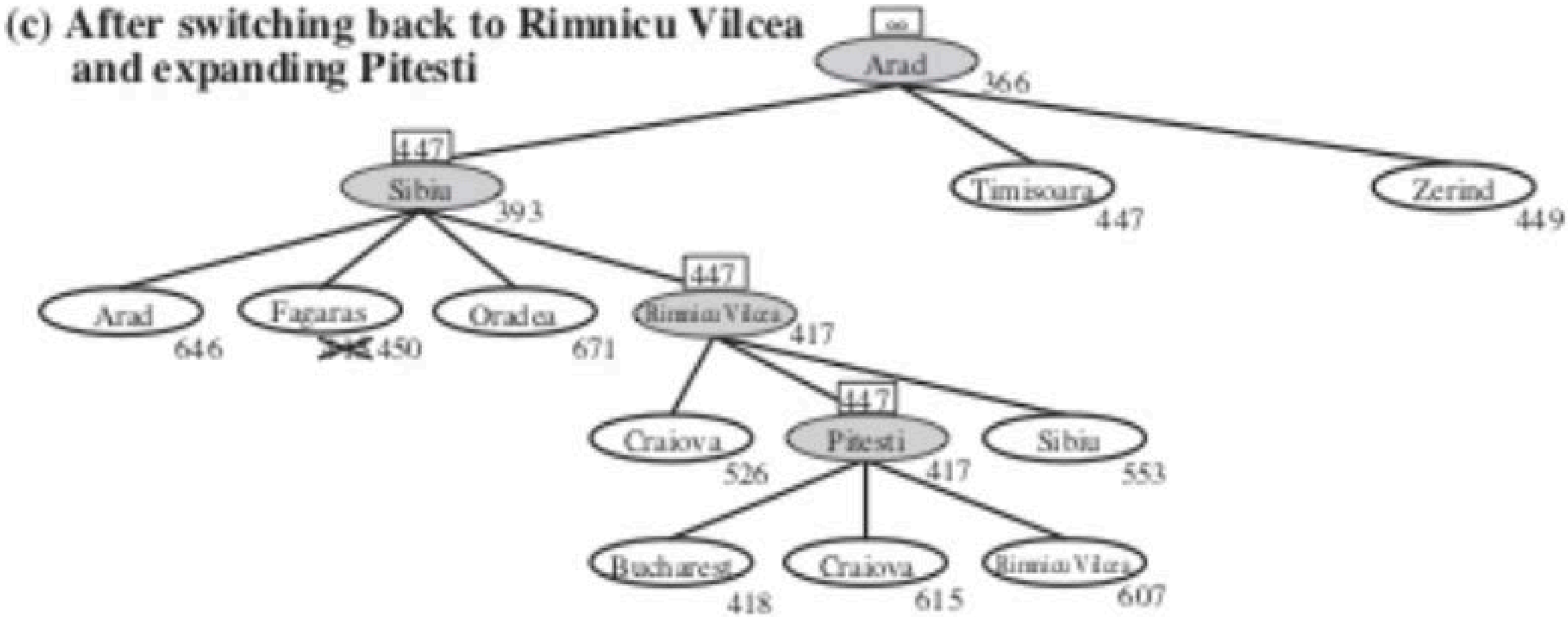

- Stages in an RBFS search for the shortest route to Bucharest. The **f-limit** value for each recursive call is shown on top of each current node, and every node is labeled with its **f -cost**. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

# RBFS 举例 2/3




(b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450.

# RBFS 举例 3/3




(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

# RBFS 的弊端在于更改路径后的重复搜索

- **RBFS** is somewhat more efficient than **IDA***, but still suffers from excessive node regeneration. In the example in Figure 3.27, RBFS follows the path via Rimnicu Vilcea, then “changes its mind” and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its **f-value** is likely to increase — **h** is usually less optimistic for nodes closer to the goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of **IDA*** and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node.

# Use More Memory, MA*, SMA*

- Like **A*** tree search, **RBFS** is an optimal algorithm if the heuristic function **h(n)** is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the **accuracy of the heuristic function** and on how often **the best path changes** as nodes are expanded.

- **IDA*** and **RBFS** suffer from using *too little* memory. Between iterations, **IDA*** retains only a single number: the current **f-cost** limit. **RBFS** retains more information in memory, but it uses only linear space: even if more memory were available, **RBFS** has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexpanding the same states many times over. Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs。

- It seems sensible, therefore, to use all available memory. Two algorithms that do this are **MA*** (memory-bounded A*) and **SMA*** (simplified MA*).

# SMA*, Drop the worst leaf, rem it’s f

- **SMA*** proceeds just like **A***, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one.

- **SMA*** always drops the *worst* leaf node—the one with the **highest f-value**. Like **RBFS**, **SMA*** then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, **SMA*** regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node **n** are forgotten, then we will not know which way to go from **n**, but we will still have an idea of how worthwhile it is to go anywhere from **n**.

# 如何处理估值相同的兄弟节点？

- The complete algorithm is too complicated to reproduce here, but there is one subtlety worth mentioning. We said that **SMA*** expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the **same f-value?** To avoid selecting the same node for deletion and expansion, **SMA*** expands the *newest* **best** leaf and deletes the *oldest* **worst** leaf. These coincide when there is **only one leaf**, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is **not reachable with the available memory**. Therefore, the node can be discarded exactly as if it had no successors.

# 对 SMA* 的分析

- **SMA*** is complete if there is any reachable solution—that is, if **d**, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise**, it returns the best reachable solution**.

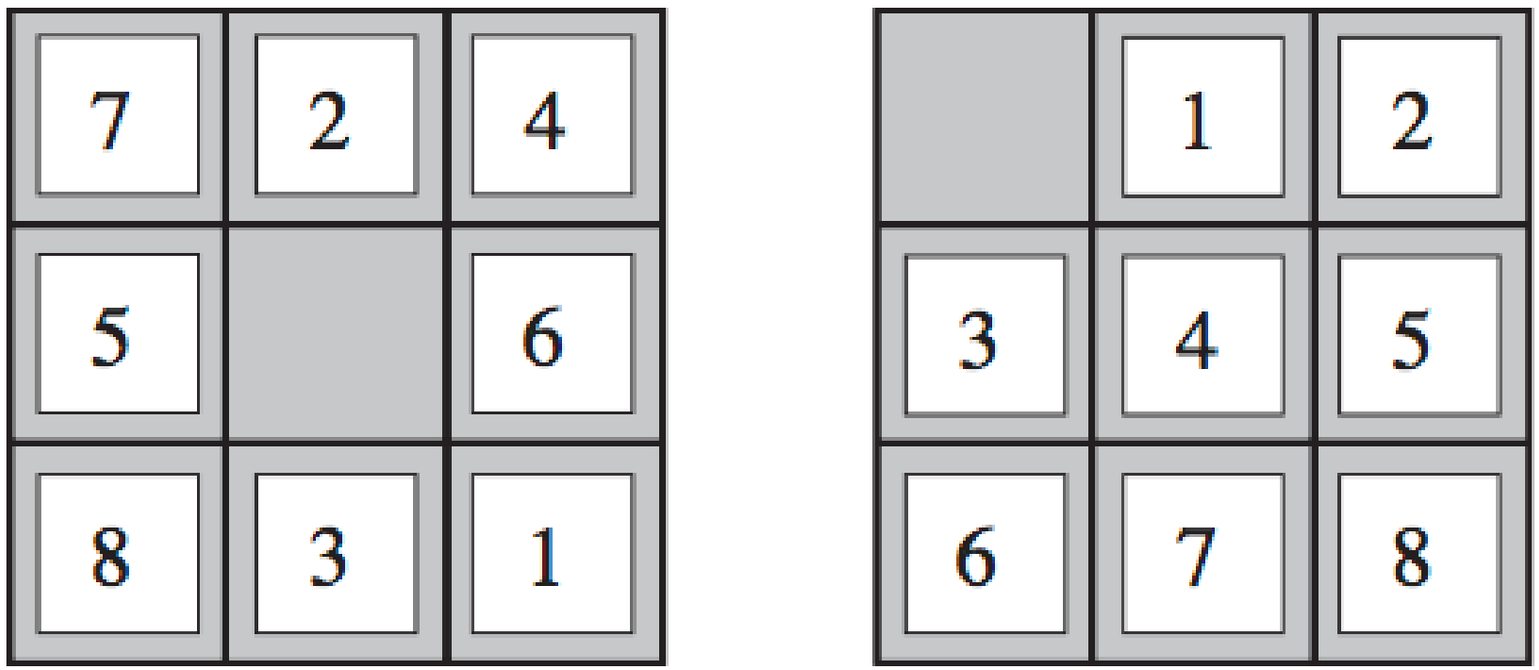
- In practical terms, **SMA*** is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and **node generation** is expensive compared to the overhead of maintaining the frontier and the explored set.

# tradeoff between time and memory

- On **very hard problems**, however, it will often be the case that **SMA*** is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory.

- Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by **A***, given unlimited memory, become intractable for **SMA***. That is to say, ***memory limitations can make a problem intractable from the point of view of computation time.*** Although no current theory explains the **tradeoff between time and memory**, it seems that this is an inescapable problem. The only way out is to **drop the optimality requirement**.

# 内容提要 – 有信息搜索 - 启发式函数



Start State　　Goal State

A typical instance of the 8-puzzle. The solution is 26 steps long.

- **h1** = the number of misplaced tiles. **h1 = 8**. h1 is an admissible heuristic
- **h2** = the sum of the distances of the tiles from their goal positions. **city block distance** or **Manhattan distance**. h is also admissible

**h2 =3+1+2+2+2+3+3+2=18**

# effective branching factor b*

- **The effect of heuristic accuracy on performance**

- One way to characterize the quality of a heuristic is the **effective branching factor b***. If the total number of nodes generated by **A*** for a particular problem is **N** and the solution depth is **d**, then **b*** is the branching factor that a uniform tree of depth **d** would have to have in order to contain **N + 1** nodes. Thus,

- $\mathbf{N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d}$ .

- For example, if **A*** finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

# Comparation of IDS, A*h1, A*h2

- To test the heuristic functions h1 and h2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with **iterative deepening search** and with **A*** tree search using both **h1** and **h2**.

- Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor.

- The results suggest that **h2 is better than h1**, and is far better than using iterative deepening search. Even for small problems with **d** = 12, **A*** with **h2** is 50,000 times more efficient than uninformed iterative deepening search.

# Experimental results

| | Search Cost (nodes generated) | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| $d$ | IDS | A*($h_1$) | A*($h_2$) | IDS | A*($h_1$) | A*($h_2$) |
| 2 | 10 | 6 | 6 | 2.45 | 1.79 | 1.79 |
| 4 | 112 | 13 | 12 | 2.87 | 1.48 | 1.45 |
| 6 | 680 | 20 | 18 | 2.73 | 1.34 | 1.30 |
| 8 | 6384 | 39 | 25 | 2.80 | 1.33 | 1.24 |
| 10 | 47127 | 93 | 39 | 2.79 | 1.38 | 1.22 |
| 12 | 3644035 | 227 | 73 | 2.78 | 1.42 | 1.24 |
| 14 | – | 539 | 113 | – | 1.44 | 1.23 |
| 16 | – | 1301 | 211 | – | 1.45 | 1.25 |
| 18 | – | 3056 | 363 | – | 1.46 | 1.26 |
| 20 | – | 7276 | 676 | – | 1.47 | 1.27 |
| 22 | – | 18094 | 1219 | – | 1.48 | 1.28 |
| 24 | – | 39135 | 1641 | – | 1.48 | 1.26 |

**Figure 3.29** Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with $h_1$, $h_2$. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths $d$.

# h2 is *always* better than h1 ?

- **The answer is "Essentially, yes."**

- It is easy to see from the definitions of the two heuristics that, for any node n, h2(n) ≥ h1(n). We thus say that h2 **dominates** h1.

- Domination translates directly into efficiency: **A*** using **h2** will never expand more nodes than **A*** using **h1** (except possibly for some nodes with **f(n)=C***).

- The argument is simple. Recall the observation on page 97 that every node with **f(n) < C*** will surely be expanded. This is the same as saying that every node with **h(n) < C* – g(n)** will surely be expanded. But because **h2** is at least as big as h1 for all nodes, every node that is surely expanded by **A*** search with **h2** will also surely be expanded with **h1**, and **h1** might cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with **higher values**, provided it is consistent and that the computation time for the heuristic is not too long.

# 启发式函数的设计方法

- **h function derived by relaxed problem**

- A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

- Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have *better* solutions if the added edges provide short cuts. Hence, ***the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.*** Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent** (see page 95).

# 放松限制条件产生松弛问题

- **Generating admissible heuristics from relaxed problems**
- A tile can move from square A to square B if
  A is horizontally or vertically adjacent to B **and** B is blank,
- we can generate three relaxed problems by removing one or both of the conditions:
- (a) A tile can move from square A to square B if A is adjacent to B.
- (b) A tile can move from square A to square B if B is blank.
  (c) A tile can move from square A to square B.

# 启发式函数的产生过程需简单

- From (a), we can derive **h2** (Manhattan distance). The reasoning is that **h2** would be the proper score if we moved each tile in turn to its destination.

- The heuristic derived from (b) is discussed in Exercise 3.31. From (c), we can derive **h1** (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step.

- Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially ***without search***, because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.

# 复合式启发函数

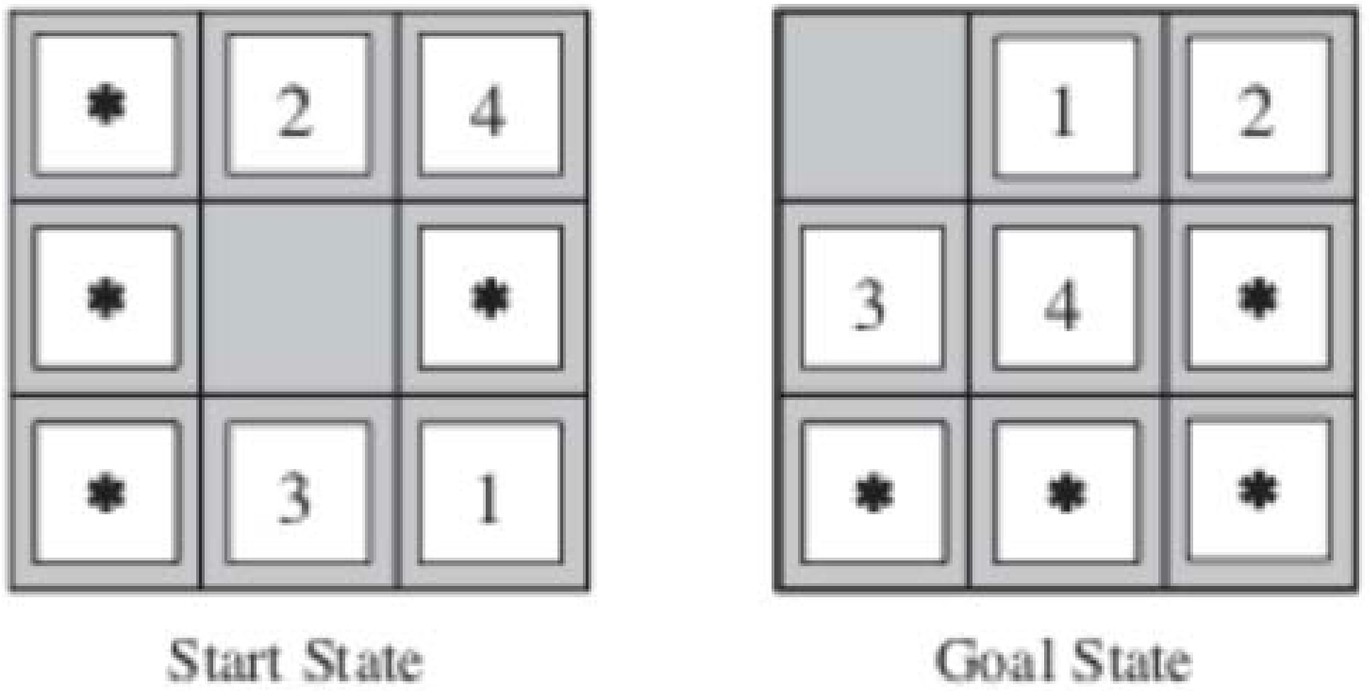
- One problem with generating new heuristic functions is that one often fails to get a single "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

- **h(n) = max{h1(n), . . . , hm(n)} .**

- This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, **h** is admissible; it is also easy to prove that **h** is consistent. Furthermore, **h** dominates all of its component heuristics.

# 用子问题的解形成模式库

- **Generating admissible heuristics from subproblems: Pattern databases**




A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- **Learning heuristics from experience**

# SUMMARY 1/3

- This chapter has introduced methods that an agent can use to select actions in environments that are **deterministic, observable, static, and completely known**. In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.
- Before an agent can start searching for solutions, a **goal** must be identified and a well- defined **problem** must be formulated.
- A problem consists of five parts: the **initial state**, a set of **actions**, a **transition model** describing the results of those actions, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a **state space**. A **path** through the state space from the initial state to a goal state is a **solution**.
- Search algorithms treat states and actions as **atomic**: they do not consider any internal structure they might possess.
- A general TREE-SEARCH algorithm considers all possible paths to find a solution, whereas a GRAPH-SEARCH algorithm avoids consideration of redundant paths.
- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on b, the branching factor in the state space, and d, the depth of the shallowest solution.

# SUMMARY 2/3

- **Uninformed search** methods have access only to the problem definition. The basic algorithms are as follows:
  - – **Breadth-first search** expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.
  - – **Uniform-cost search** expands the node with lowest path cost, g(n), and is optimal for general step costs.
  - – **Depth-first search** expands the deepest unexpanded node first. It is neither com- plete nor optimal, but has linear space complexity. **Depth-limited search** adds a depth bound.
  - – **Iterative deepening search** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity.
  - – **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space.

# SUMMARY 3/3

- **Informed search** methods may have access to a **heuristic** function h(n) that estimates the cost of a solution from n.
  - – The generic **best-first search** algorithm selects a node for expansion according to an **evaluation function**.
  - – **Greedy best-first search** expands nodes with minimal h(n). It is not optimal but is often efficient.
- – **A∗ search** expands nodes with minimal f (n) = g(n) + h(n). A∗ is complete and optimal, provided that h(n) is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A∗ is still prohibitive.
- – **RBFS** (recursive best-first search) and **SMA∗** (simplified memory-bounded A∗) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A∗ cannot solve because it runs out of memory.
- • The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem defi- nition, by storing precomputed solution costs for subproblems in a pattern database, or by learning from experience with the problem class.