國立成功大學
National Cheng Kung University

Introduction to

Artificial Intelligence

Chapter 3
Solving Problems by Searching

Wei-Ta Chu (朱威達)

Introduction

- Goal-based agents consider future actions and the desirability of their outcomes. This chapter describes one kind of goal-based agent called a **problem-solving agent**.
- In this chapter, we limit ourselves to the simplest kind of task environment (PEAS: Performance, Environment, Actuators, Sensors), for which the solution to a problem is always a *fixed sequence* of actions.

Problem-Solving Agents

- The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Problem-Solving Agents

- An example problem: moving from Arad to Bucharest

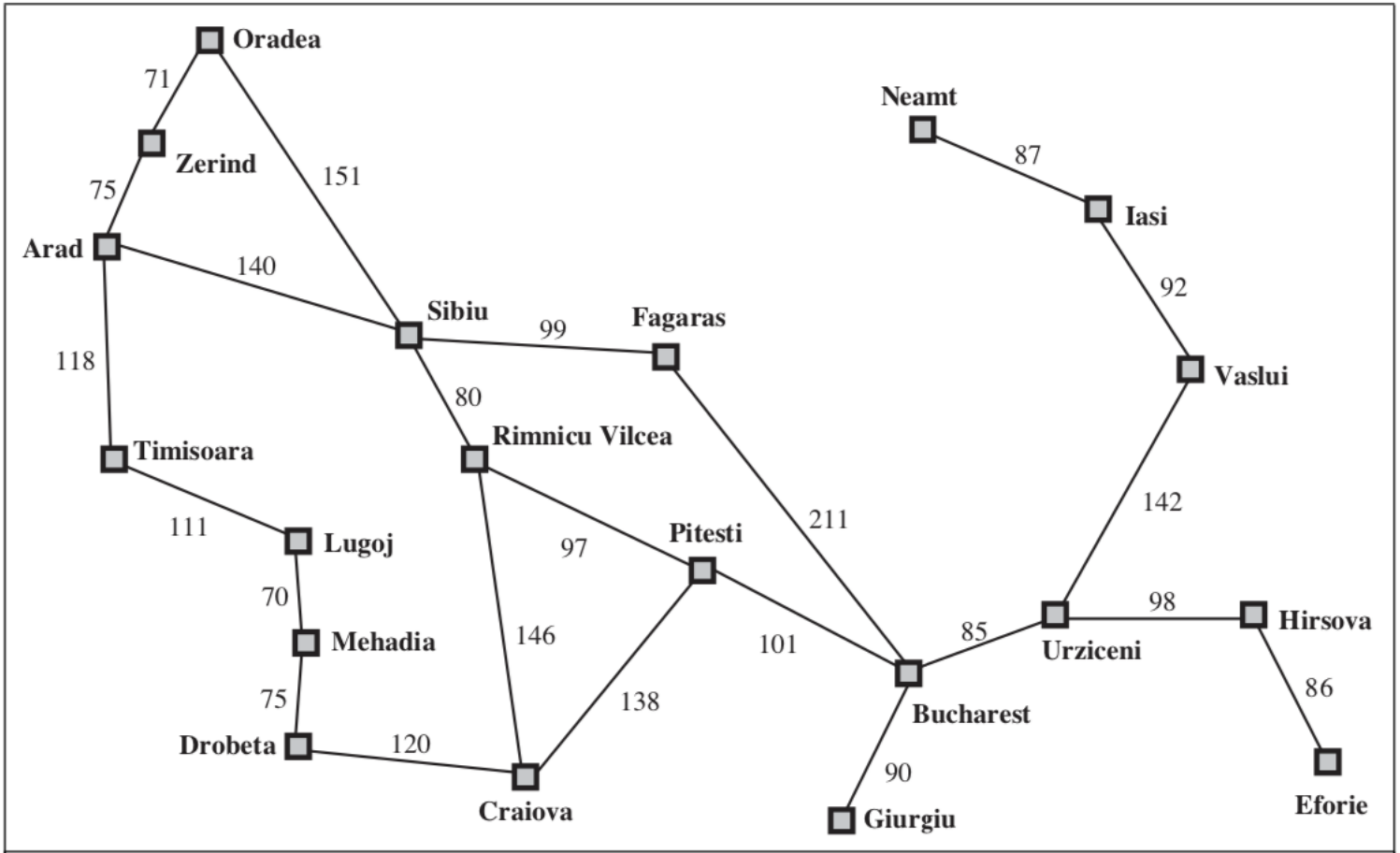


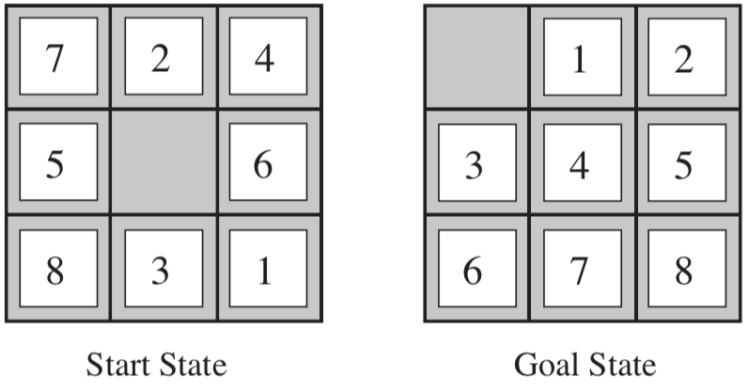
Figure 3.2 A simplified road map of part of Romania.

Well-Defined Problems and Solutions

- A problem can be defined formally by five components
  - **Initial state –** *In*(*Arad*)
  - A description of the possible **actions --** {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)}
  - A description of what each action does (**transition model**) **--** RESULT(*In*(*Arad*), *Go*(*Zerind*)) = *In*(*Zerind*)
  - The **goal test**, which determines whether a given state is a goal state **--** *In*(*Bucharest*)
  - A **path cost** function that assigns a numeric cost to each path
- 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.

Formulating Problems

- 8-puzzle problem



Start State Goal State

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.

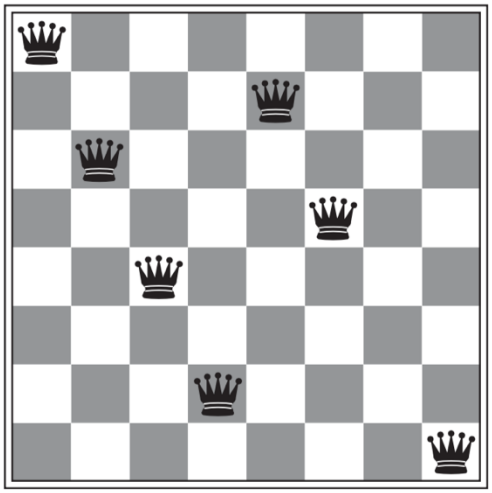
Transition model: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.

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.

Formulating Problems

- 8-queens problem
  - A queen attacks any piece in the same row, column or diagonal.



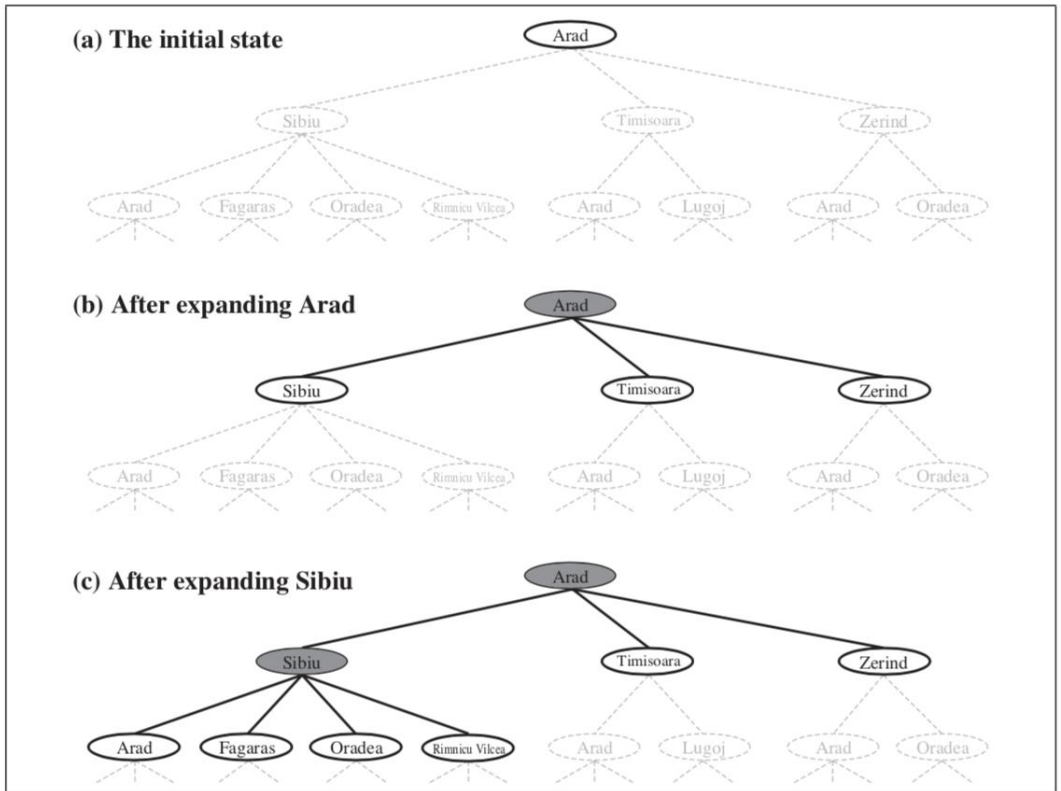
- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

Real-World Problems

- Touring problems – route-finding problem
- Traveling salesperson problem – route-finding problem
- VLSI layout problem
- Robot navigation -- – route-finding problem
- …

Searching for Solutions

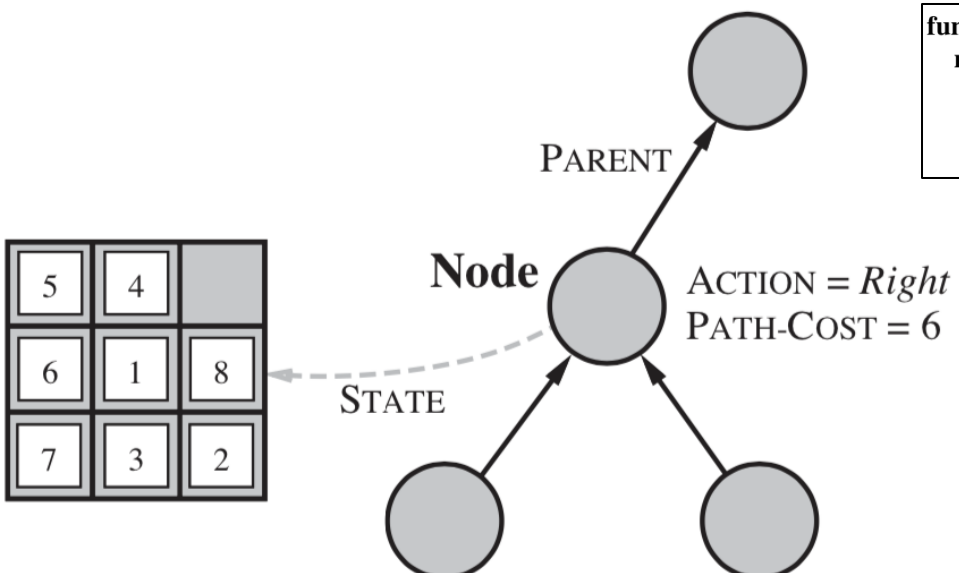
- Search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the *branches* are *actions* and the *nodes* correspond to *states* in the state space of the problem.



Infrastructure for Search Algorithms

- For each node *n* of the tree, we have a structure containing 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)
```

Measuring Problem-Solving Performance

We can evaluate an algorithm's performance in four ways:

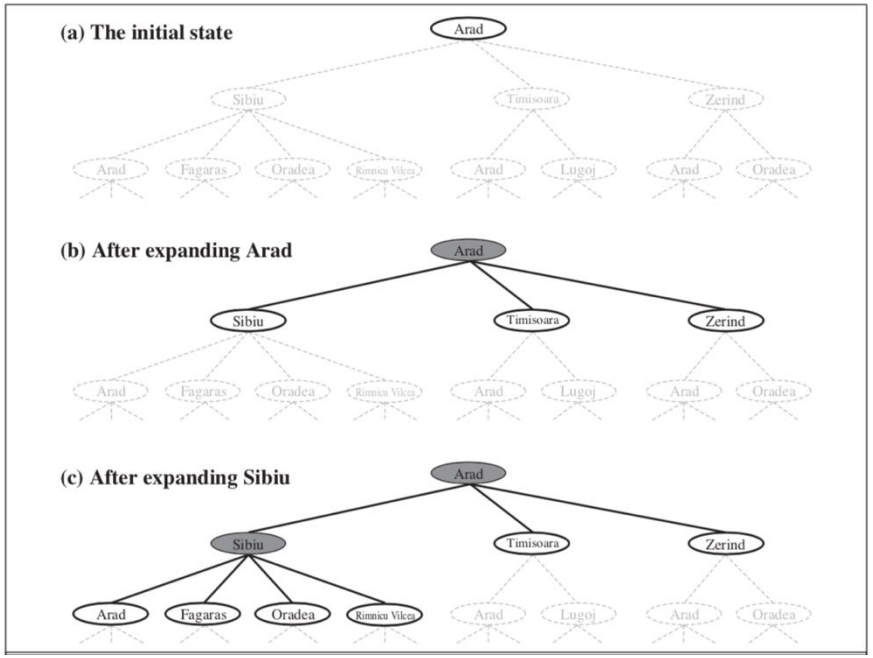
- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Uninformed Search Strategies

- **Uninformed search** (also called **blind search**)
- 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.
- All search strategies are distinguished by the *order* in which nodes are expanded.
- Strategies that know whether one non-goal state is "more promising" than another are called **informed search** or **heuristic search** strategies.

Uninformed Search Strategies

- **Breadth-first search (BFS)**
  - The root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.



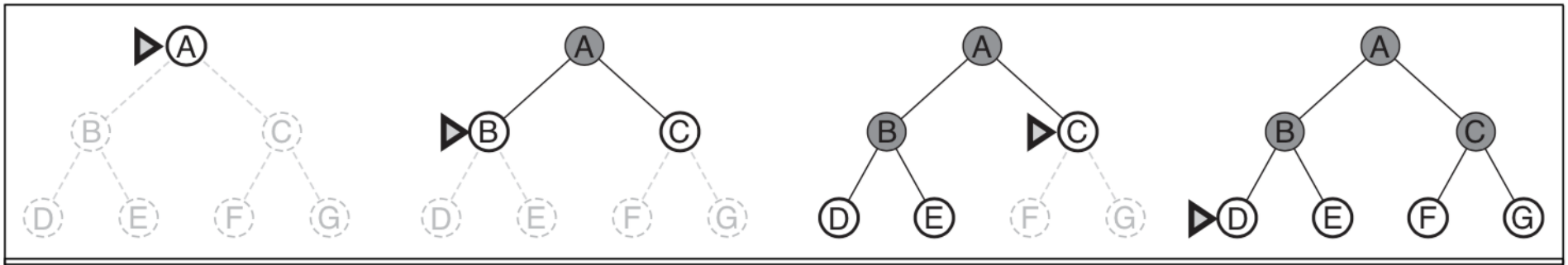


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Uninformed Search Strategies

- **Breadth-first search (BFS)**
  - 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.
  - Suppose that the solution is at depth *d*. The total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Uninformed Search Strategies

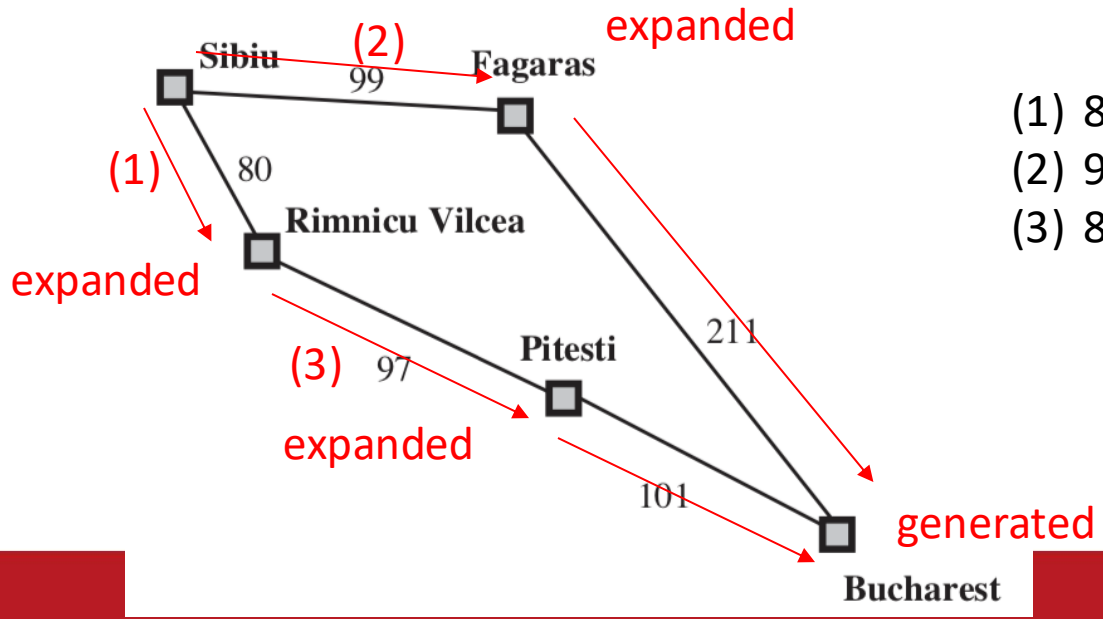
- **Breadth-first search (BFS)**

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.

Uninformed Search Strategies

- **Uniform-cost search (AI) or Dijkstra's algorithm (Theoretical CS)**
  - Uniform-cost search expands the node *n* with the ***lowest path cost*** *g*(*n*).
  - The algorithm tests for goals only when it **expands** a node, not when it **generates** a node



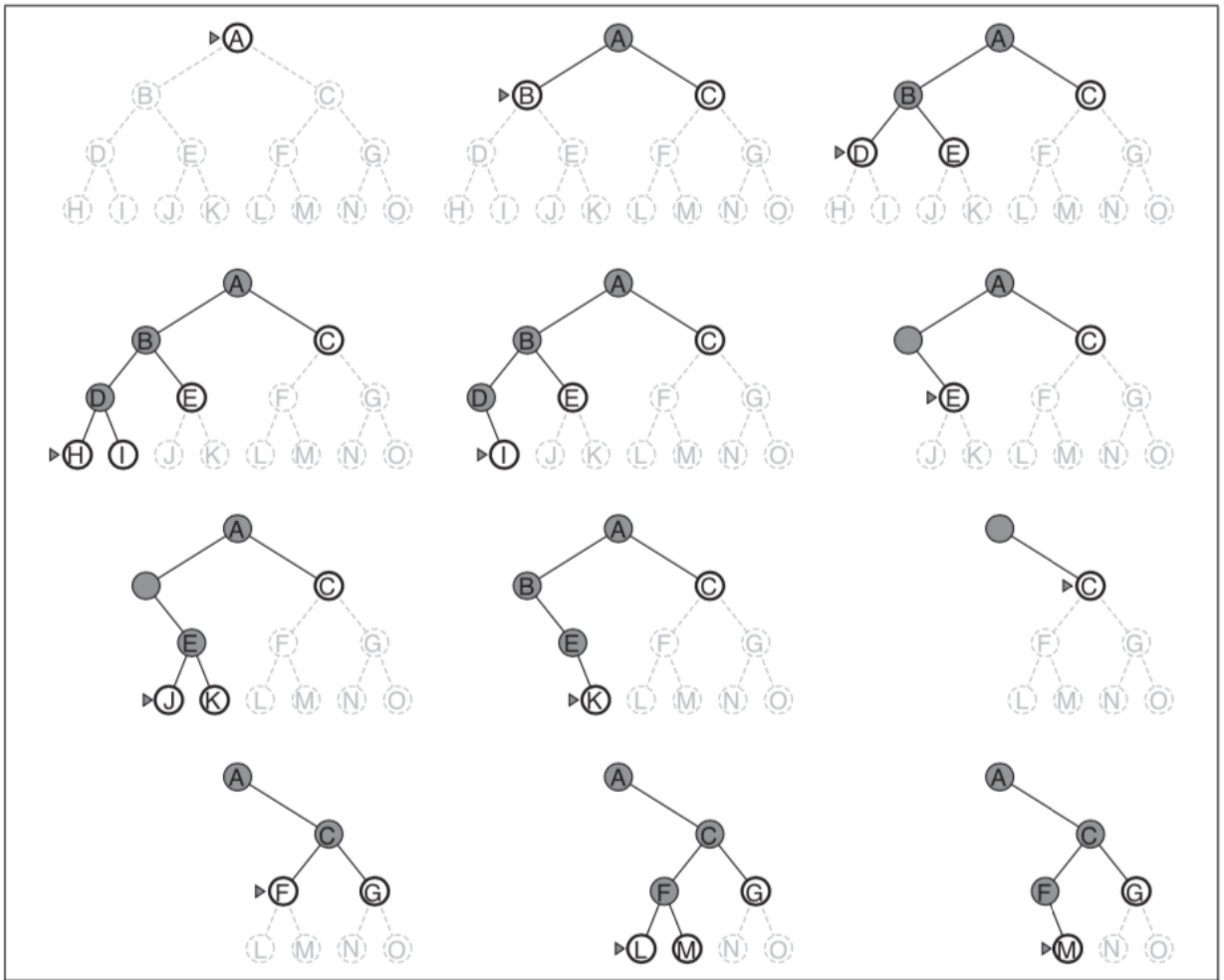
(1) 80+97=177
(2) 99+211=310
(3) 80+97+101=278

Uninformed Search Strategies

- **Uniform-cost search (AI) or Dijkstra's algorithm (Theoretical CS)**
  - Uniform-cost search is optimal in general. Uniform-cost search expands nodes in order of their optimal path cost.
  - Uniform-cost search is *guided by path costs rather than depths*, so its complexity is not easily characterized in terms of b and d.
  - 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. $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d.
  - When all step costs are the same, uniform-cost search is similar to breadth-first search.

Uninformed Search Strategies

- **Depth-first search (DFS)**
  - Expands the *deepest* node in the current frontier of the search tree.



Uninformed Search Strategies

- **Depth-first search (DFS)**
  - Time complexity: 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. *m* can be much larger than *d* (the depth of the shallowest solution).

Uninformed Search Strategies

- **Depth-first search (DFS)**
  - Space complexity: 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*, DFS requires storage of only $O(bm)$ nodes.

Uninformed Search Strategies

- **Depth-limited search**
  - Supply depth-first search with a predetermined depth limit ℓ
  - Incompleteness
  - Nonoptimal
  - Sometimes, depth limits can be based on knowledge of the problem. For example, if we check carefully, we would discover that any city can be reached from any other city in at most 9 steps, i.e., $\ell = 9$ leads to a more efficient depth-limited search.

Uninformed Search Strategies

- **Iterative deepening DFS**
  - Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
  - It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
  - Iterative deepening combines the benefits of depth-first and breadth-first search. Like DFS, its memory requirements are modest: $O(bd)$ to be precise. Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth.

Uninformed Search Strategies

- **Iterative deepening DFS**
- 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.

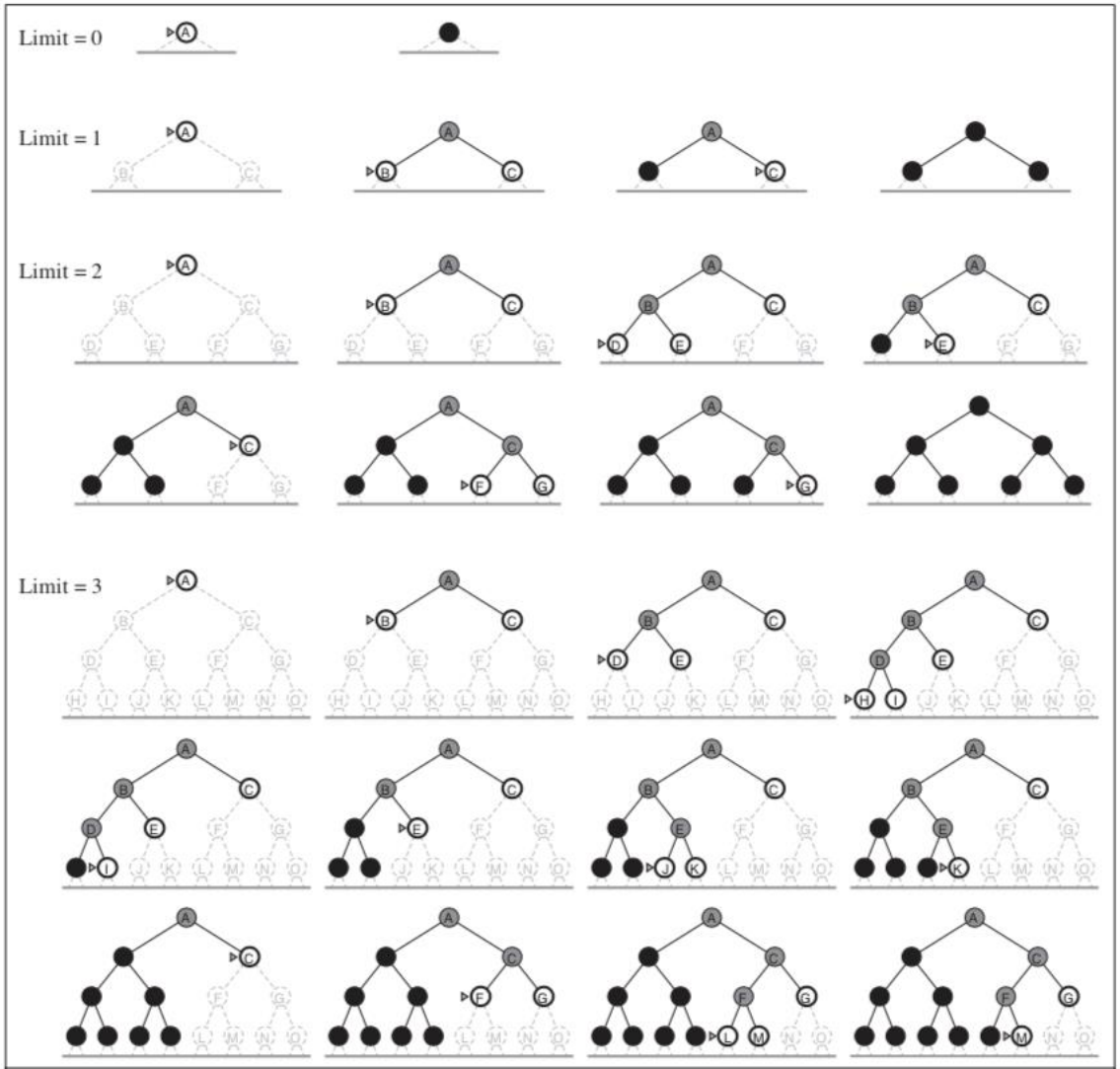


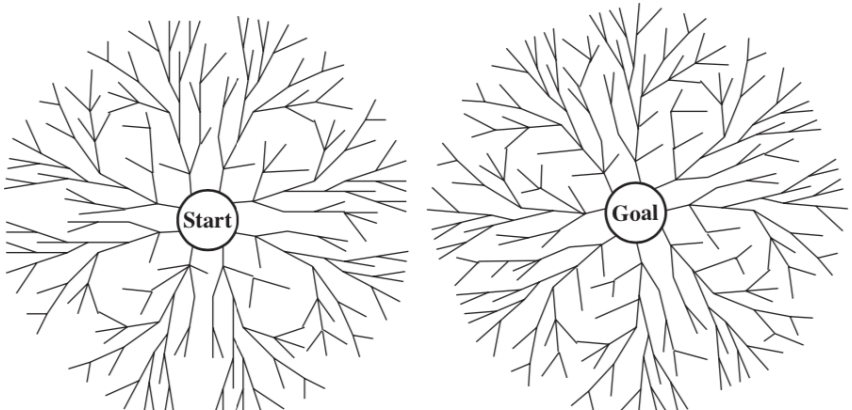
Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Uninformed Search Strategies

- **Iterative deepening DFS**
  - Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly.
  - The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

Uninformed Search Strategies

- **Bidirectional search**
  - Run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
  - The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
  - Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.



Uninformed Search Strategies

- **Bidirectional search**
  - How to search backward? This is not easy.
  - Let the **predecessors** of a state *x* be all those states that have *x* as a successor. Bidirectional search requires a method for computing predecessors. When all the actions in the state space are reversible, the predecessors of *x* are just its successors.
  - For the 8-puzzle and for finding a route in Romania, there is just one goal state, so the backward search is very much like the forward search.
  - If the goal is an abstract description, such as the goal that "no queen attacks another queen", then bidirectional search is difficult to use.

Informed (Heuristic) Search Strategies

- Informed search strategy—one that uses problem-specific knowledge hints about the location of goals—can find solutions more efficiently than can an uninformed strategy.
- The hints come in the form of a **heuristic function**, denoted $h(n)$:
  - $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

Informed (Heuristic) Search Strategies

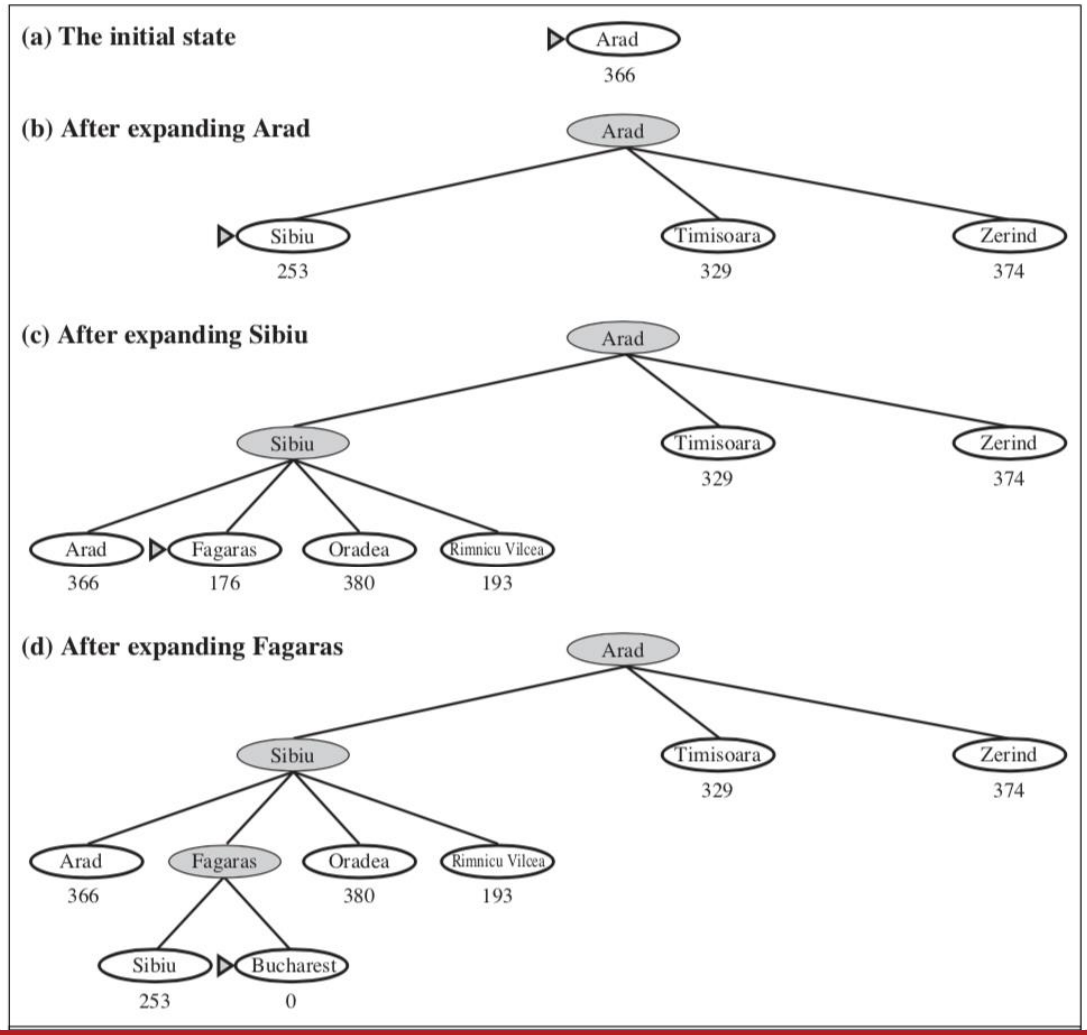
- **Greedy best-first search**
  - Expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.
  - In this example, use the straight-line distance heuristic

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.

Informed (Heuristic) Search Strategies

- **Greedy best-first search**
  - 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.
  - However, this solution is not optimal



Informed (Heuristic) Search Strategies

- **Greedy best-first search**
  - This solution is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
  - Greedy best-first tree search is complete in finite state spaces, but not in infinite ones.
  - With a good heuristic function, the complexity can be reduced substantially.

Informed (Heuristic) Search Strategies

- **A* search**
  - The most widely known form of best-first 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.
  - $g(n)$ 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.
  - $f(n) = g(n) + h(n)$ is the estimated cost of the cheapest solution through n.

Informed (Heuristic) Search Strategies

- **A* search: 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.
  - Example: straight-line distance that we used in getting to Bucharest

Informed (Heuristic) Search Strategies

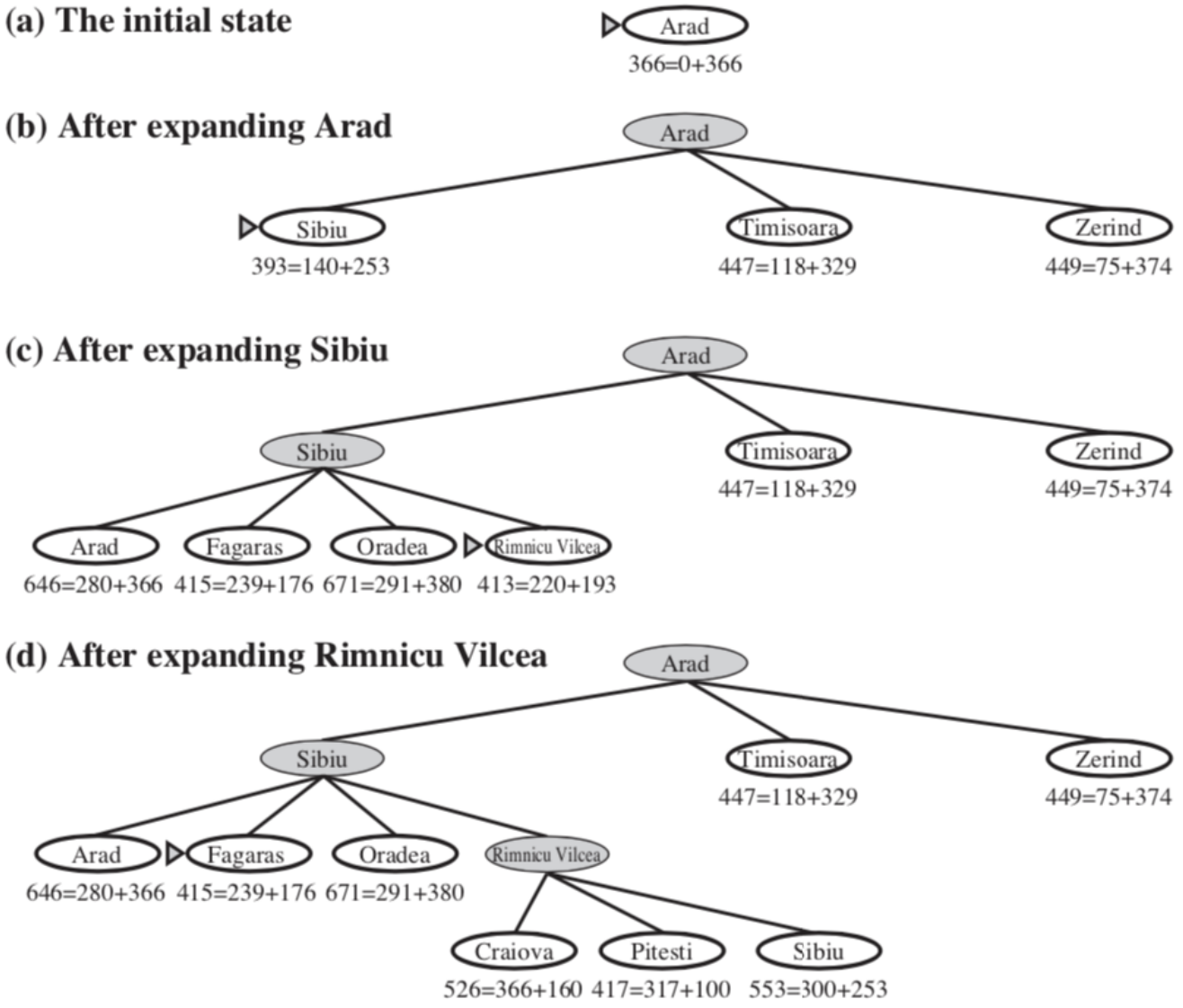
- **A* search: conditions for optimality – admissibility and consistency**
  - A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A* to graph search.
  - A heuristic $h(n)$ is **consistent** if, for every node n and every 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**.

Informed (Heuristic) Search Strategies

- **A* search**



Informed (Heuristic) Search Strategies

- **A* search**