Chapter 3 - Solving Problems by Searching

Russell, S., & Norvig, P. (2022). *Artificial Intelligence - A Modern Approach* (4th global ed.). Pearson.

Contents

1. Problem-Solving Agents

- When the correct action to take is not immediately obvious, an agent may need to *plan ahead*: to consider a *sequence* of actions that form a path to a goal state. Such an agent is called a **problem-solving agent**, and the computational process it undertakes is called **search**.
- In this chapter, we consider only the simplest environments: *episodic, single agent, fully observable, deterministic, static, discrete*, and ***known***.
- We distinguish between **informed** algorithms, in which the agent can estimate how far it is from the goal, and **uninformed** algorithms, where no such estimate is available.

1. Problem-Solving Agents

- Imagine an agent enjoying a touring vacation in Romania. Now, suppose the agent is currently in the city of Arad and wants to go Bucharest.
- The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind.
- None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.
- If the agent has no additional information—that is, if the environment is **unknown**—then the agent can do no better than to execute one of the actions at random. This is a sad situation.
- In this chapter, we will assume our agents always have access to information about the world, such as the map in Figure 3.1.

1. Problem-Solving Agents

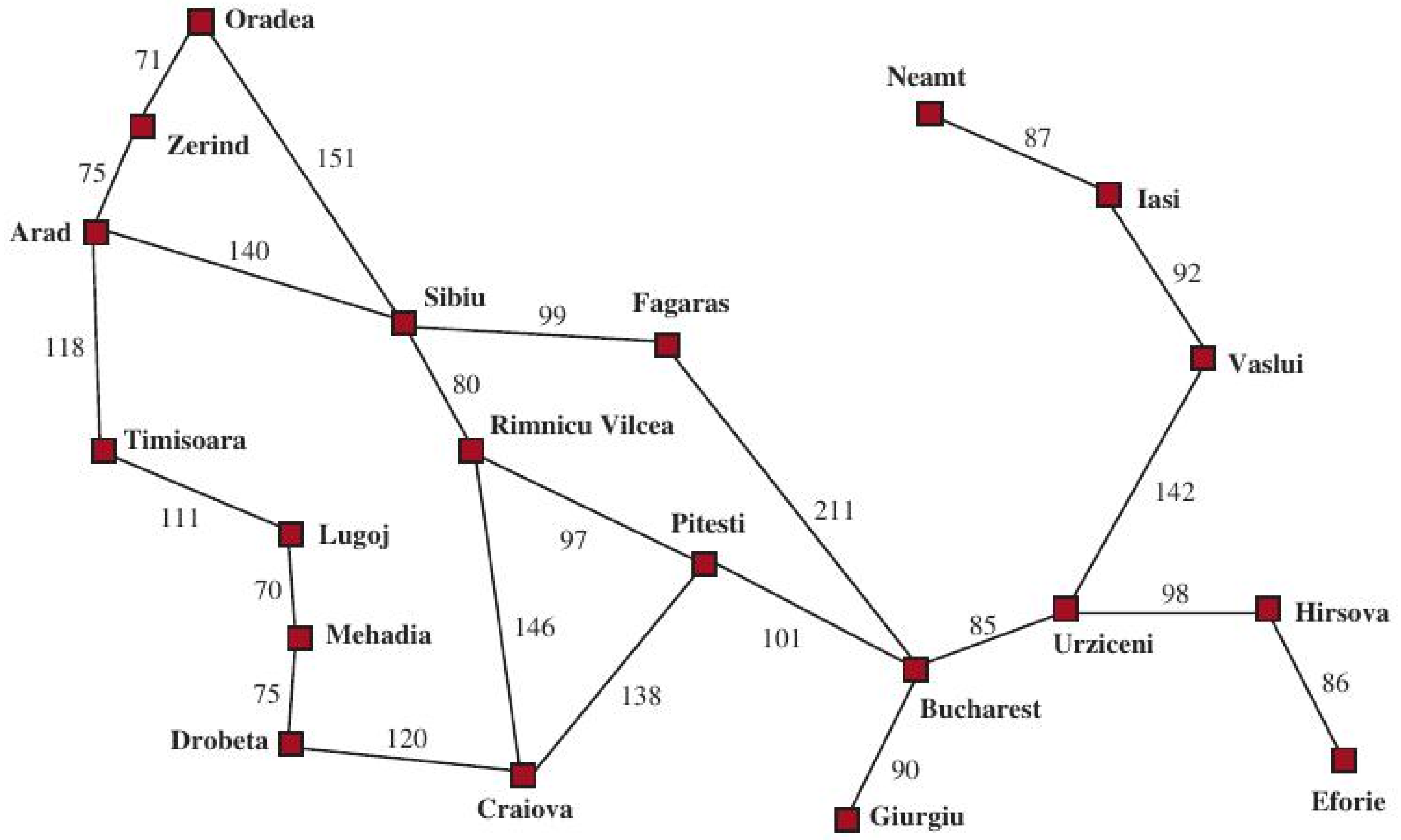


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

1. Problem-Solving Agents

- With that information, the agent can follow this four-phase problem-solving process:
  - **Goal formulation**: The agent adopts the **goal** of reaching Bucharest.
  - **Problem formulation**: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world.
    - For our agent, one good model is to consider the actions of traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.
  - **Search**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
    - Such a sequence is called a **solution**. The agent might have to simulate multiple sequences that do not reach the goal, but eventually it will find a solution (such as going from Arad to Sibiu to Fagaras to Bucharest), or it will find that no solution is possible.
  - **Execution**: The agent can now execute the actions in the solution, one at a time.

1. Problem-Solving Agents

- It is an important property that in a fully observable, deterministic, known environment, *the solution to any problem is a fixed sequence of actions*:
  - Drive to Sibiu, then Fagaras, then Bucharest.
- If the model is correct, then once the agent has found a solution, it can ignore its percepts while it is executing the actions—closing its eyes, so to speak—because the solution is guaranteed to lead to the goal. Control theorists call this an **open-loop** system: ignoring the percepts breaks the loop between agent and environment.
- If there is a chance that the model is incorrect, or the environment is nondeterministic, then the agent would be safer using a **closed-loop** approach that monitors the percepts.

1. Problem-Solving Agents

- In partially observable or nondeterministic environments, a solution would be a branching strategy that recommends different future actions depending on what percepts arrive.
- For example, the agent might plan to drive from Arad to Sibiu but might need a contingency plan in case it arrives in Zerind by accident or finds a sign saying “Drum Închis” (Road Closed).

1. Problem-Solving Agents
Search Problems and Solutions

- A search **problem** can be defined formally as follows:
  - 1. A set of possible **states** that the environment can be in. We call this the **state space**.
  - 2. The **initial state** that the agent starts in. For example: *Arad*.
  - 3. A set of one or more **goal states**. Sometimes there is one goal state (e.g., *Bucharest*), sometimes there is a small set of alternative goal states, and sometimes the goal is defined by a property that applies to many states (potentially an infinite number).
  - 4. The **actions** available to the agent. Given a state *s*, ACTIONS(*s*) returns a finite set of actions that can be executed in *s*. We say that each of these actions is applicable in *s*.
    - ACTIONS(*Arad*) = {*ToSibiu*, *ToTimisoara*, *ToZerind*}

1. Problem-Solving Agents

Search Problems and Solutions

- 5. A **transition model**, which describes what each action does. RESULT(*s*, *a*) returns the state that results from doing action *a* in state *s*. For example,
  - RESULT(*Arad*, *ToZerind*) = *Zerind*.
- 6. An **action cost function**, denoted by ACTION-COST(*s*, *a*, *s'*) when we are programming or *c*(*s*, *a*, *s'*) when we are doing math, that gives the numeric cost of applying action *a* in state *s* to reach state *s'*.
  - A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length in miles, or it might be the time it takes to complete the action.

1. Problem-Solving Agents

Search Problems and Solutions

- Path
  - A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.
  - We assume that action costs are additive; that is, the total cost of a path is the sum of the individual action costs.
- Optimal Solution
  - An **optimal solution** has the lowest path cost among all solutions. In this chapter, we assume that all action costs will be positive, to avoid certain complications.
- Graph
  - The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

1. Problem-Solving Agents
Formulating Problems

- Our formulation of the problem of getting to Bucharest is a **model**—an abstract mathematical description—and not the real thing.
- Compare the simple atomic state description *Arad* to an actual cross-country trip, where the state of the world includes so many things (the condition of the road, the weather, the traffic, and so on). All these considerations are left out of our model because they are irrelevant to the problem of finding a route to Bucharest.
- Abstraction
  - The process of removing detail from a representation is called **abstraction**. A good problem formulation has the right level of detail.
    - If the actions were at the level of "move the right foot forward a centimeter" 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.

1. Problem-Solving Agents
Formulating Problems

- Level of Abstraction
  - The abstraction is *valid* if we can elaborate any abstract solution into a solution in the more detailed world.
  - The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem.
  - The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

2. Example Problems

- Standardized Problem
  - A standardized problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms.
- Real-world Problem
  - A real-world problem, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

2. Example Problems
Standardized Problems - Grid World

- A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell.
- Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally.
- Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell.

2. Example Problems

Standardized Problems - Grid World

- The vacuum world can be formulated as a grid world problem as follows:
  - **States**: A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each cell can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states. In general, a vacuum environment with n cells has $n \cdot 2^n$ states.

2. Example Problems
Standardized Problems - Grid World

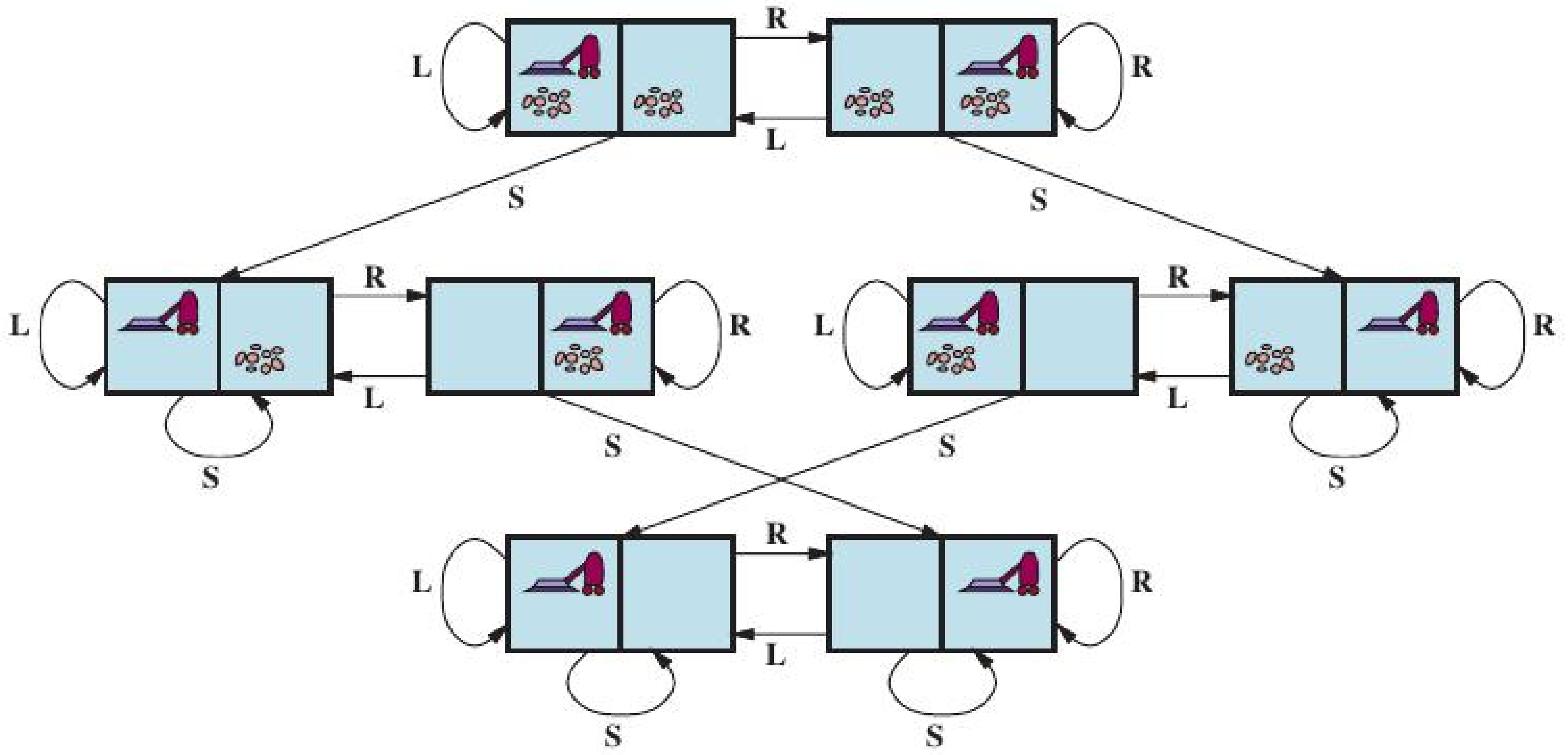


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

2. Example Problems

Standardized Problems - Grid World

- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four absolute movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward, Backward, TurnRight*, and *TurnLeft*.
- **Transition model**: *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90◦
- **Goal states**: The states in which every cell is clean.
- **Action cost**: Each action costs 1.

2. Example Problems

Standardized Problems - Sokoban Puzzle

- Another type of grid world is the **sokoban puzzle,** in which the agent's goal is to push a number of boxes, scattered about the grid, to designated storage locations. There can be at most one box per cell. When an agent moves forward into a cell containing a box and there is an empty cell on the other side of the box, then both the box and the agent move forward.
- The agent can't push a box into another box or a wall.
- For a world with n non-obstacle cells and b boxes, there are $n \times n!/(b!(n-b)!)$ states; for example on an 8×8 grid with a dozen boxes, there are over 200 trillion states.

2. Example Problems

Standardized Problems - Sliding-Tile Puzzle

- In a **sliding-tile puzzle**, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space.
- Perhaps the best-known variant is the **8-puzzle** (see Figure 3.3), which consists of a 3 × 3 grid with eight numbered tiles and one blank space, and the **15-puzzle** on a 4 × 4 grid. The object is to reach a specified goal state, such as the one shown on the right of the figure.

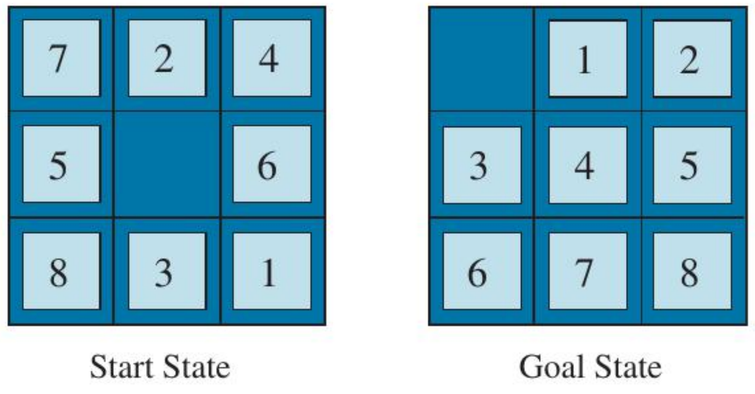


Figure 3.3 A typical instance of the 8-puzzle.

2. Example Problems

Standardized Problems - Sliding-Tile Puzzle

- The standard formulation of the 8-puzzle is as follows:
  - **States**: A state description specifies the location of each of the tiles.
  - **Initial state**: Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states.
  - **Actions**: While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving *Left, Right, Up*, or *Down*. If the blank is at an edge or corner then not all actions will be applicable.
  - **Transition model**: Maps a state and action to a resulting state; for example, if we apply *Left* to the start state in Figure 3.3, the resulting state has the 5 and the blank switched.
  - **Goal state**: Although any state could be the goal, we typically specify a state with the numbers in order, as in Figure 3.3.
  - **Action cost**: Each action costs 1.

2. Example Problems

Standardized Problems - Donald Knuth (1964)

- Our final standardized 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 square root, floor, and factorial operations can reach any desired positive integer.
- For example, we can reach 5 from 4 as follows:

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

2. Example Problems

Standardized Problems - Donald Knuth (1964)

- The problem definition is simple:
  - **States**: Positive real numbers.
  - **Initial state**: 4.
  - **Actions**: Apply square root, floor, or factorial operation (factorial for integers only).
  - **Transition model**: As given by the mathematical definitions of the operations.
  - **Goal state**: The desired positive integer.
  - **Action cost**: Each action costs 1.
- The state space for this problem is infinite: for any integer greater than 2 the factorial operator will always yield a larger integer. The problem is interesting because it explores very large numbers: the shortest path to 5 goes through (4!)! = 620,448,401,733,239,439,360,000.

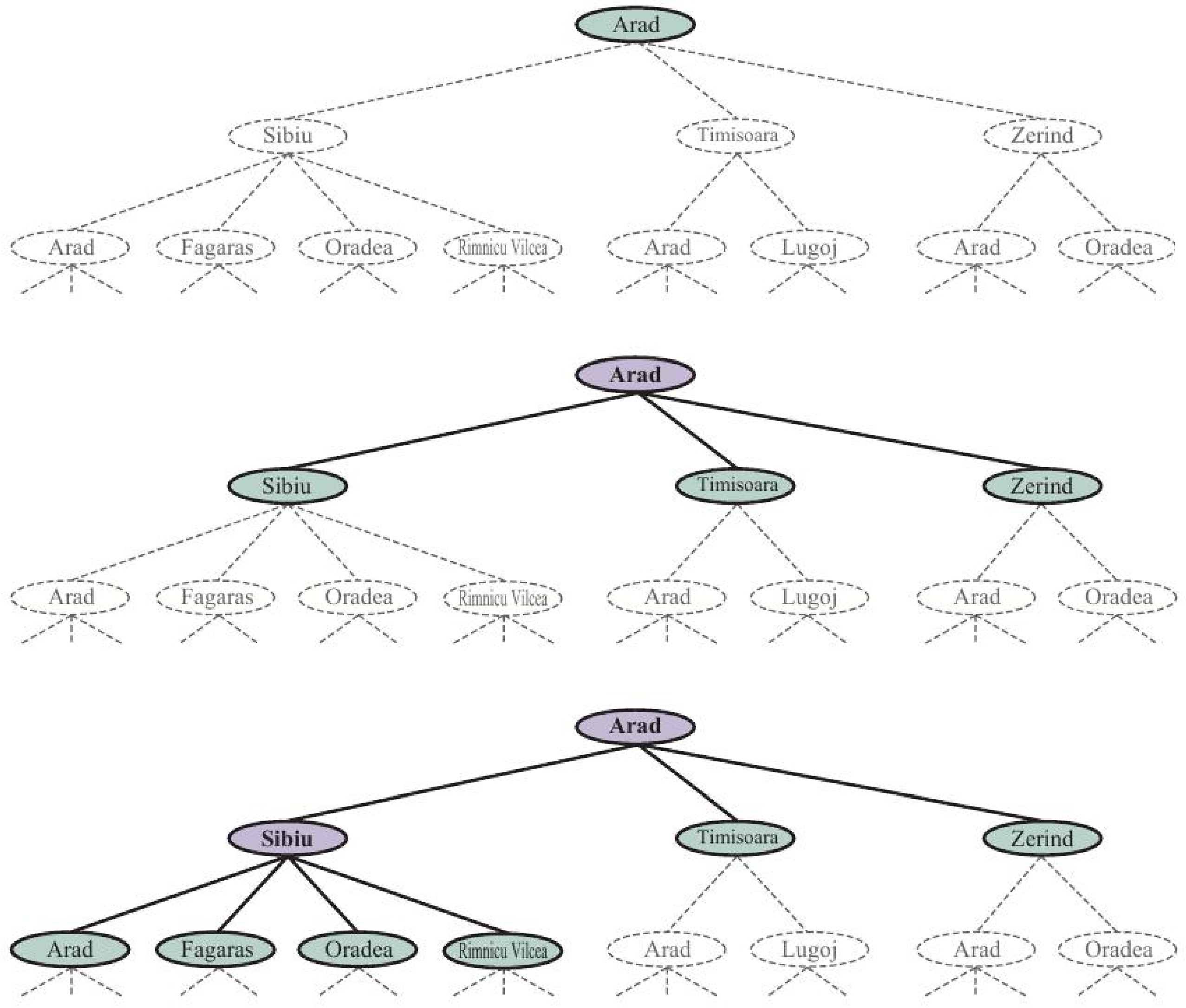
2. Example Problems
Real-World Problems

- Route-Finding Problem
- Routing video streams in computer networks
- Military operations planning
- Airline travel-planning systems
- Touring problems: describe a set of locations that must be visited, rather than a single goal destination.
  - The **traveling salesperson problem** (TSP) is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost $< C$ (or in the optimization version, to find a tour with the lowest cost possible).
  - An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles.

3. Search Algorithms

- A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure.
- In this chapter we consider algorithms that superimpose a search tree over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state.
  - Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.
- It is important to understand the distinction between the state space and the search tree. The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

3. Search Algorithms



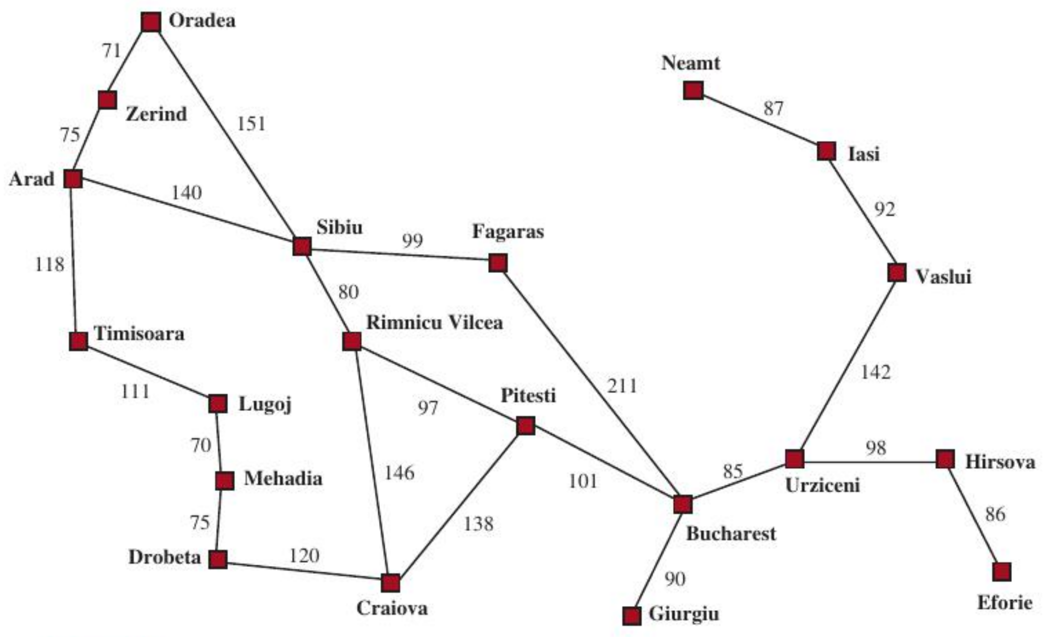


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Figure 3.4 Three partial search trees for finding a route from *Arad* to *Bucharest*.
Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*.
Nodes that could be generated next are shown in faint dashed lines.
Notice in the bottom tree there is a cycle from *Arad* to *Sibiu* to *Arad*; that can't be an optimal path, so search should not continue from there.

3. Search Algorithms

- Figure 3.5 shows the search tree superimposed on the state-space graph.

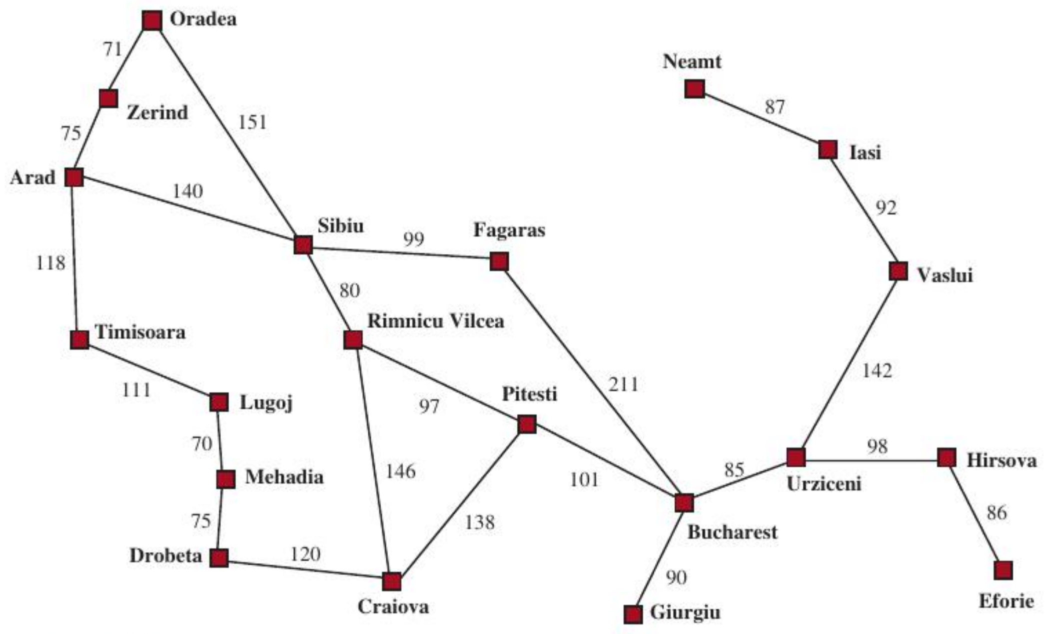


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

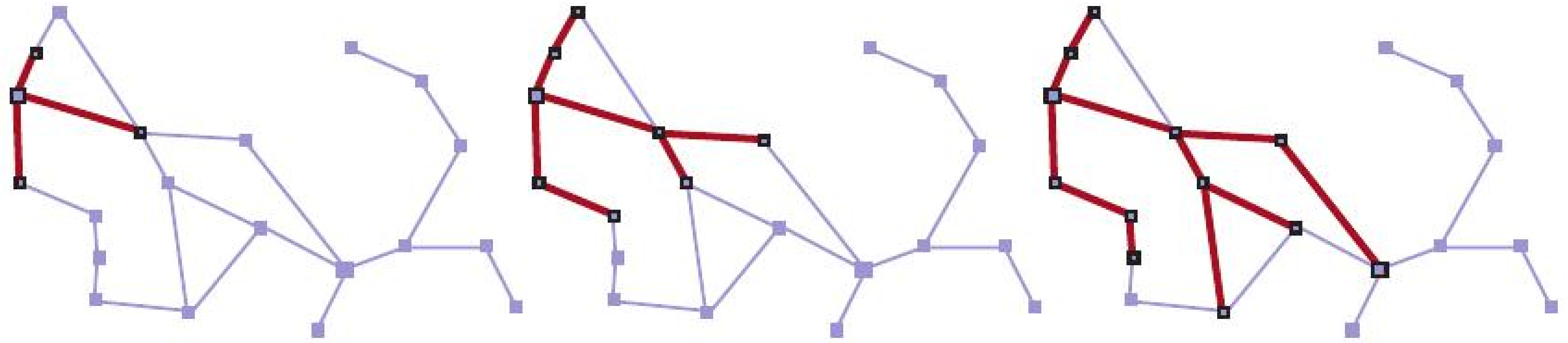


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

3. Search Algorithms

- Note that the frontier **separates** two regions of the state-space graph:
  - An interior region where every state has been expanded,
  - An exterior region of states that have not yet been reached. This property is illustrated in Figure 3.6.

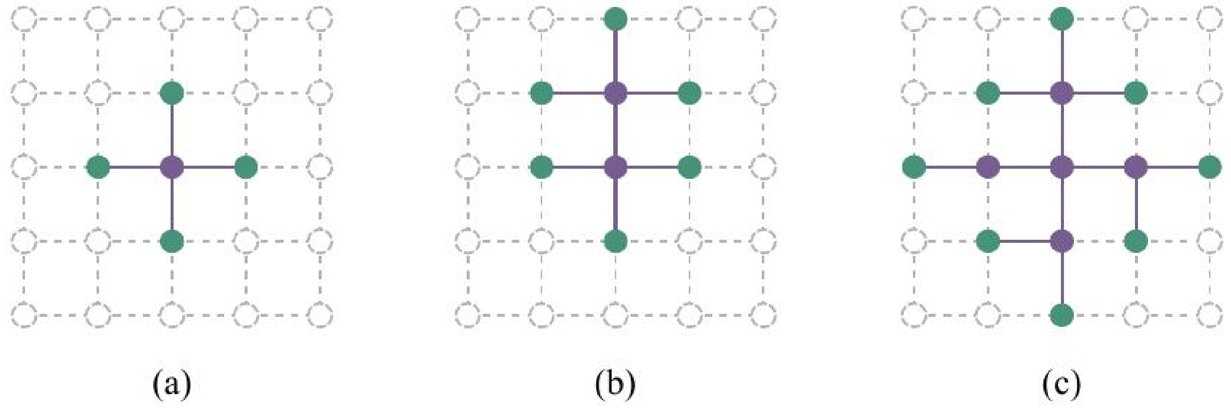


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

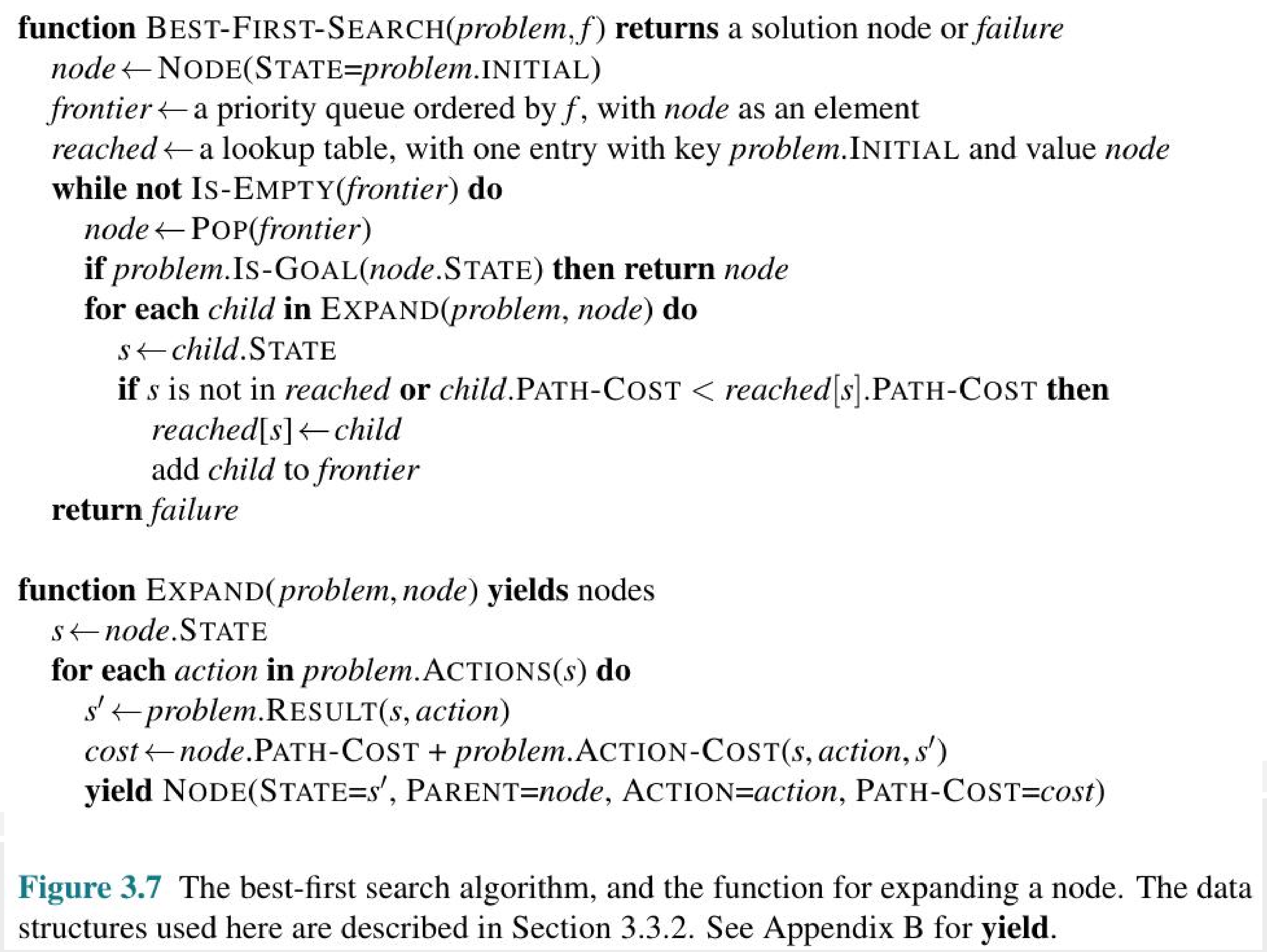
3. Search Algorithms

Best-First Search

- How do we decide which node from the frontier to expand next?
- A very general approach is called **best-first search**, in which we choose a node, n, with minimum value of some evaluation function, $f(n)$.

3. Search Algorithms

Best-First Search



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

3. Search Algorithms

Search Data Structures

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

3. Search Algorithms

Search Data Structures

- We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:
  - IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
  - POP(*frontier*) removes the top node from the frontier and returns it.
  - TOP(*frontier*) returns (but does not remove) the top node of the frontier.
  - ADD(*node*, *frontier*) inserts node into its proper place in the queue.

3. Search Algorithms
Search Data Structures

- Three kinds of queues are used in search algorithms:
  - A **priority queue** first pops the node with the minimum cost according to some evaluation function, *f*. It is used in best-first search.
  - A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.
  - A LIFO queue or last-in-first-out queue (also known as a stack) pops first the most recently added node; we shall see it is used in depth-first search.
- The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

3. Search Algorithms

Redundant Paths

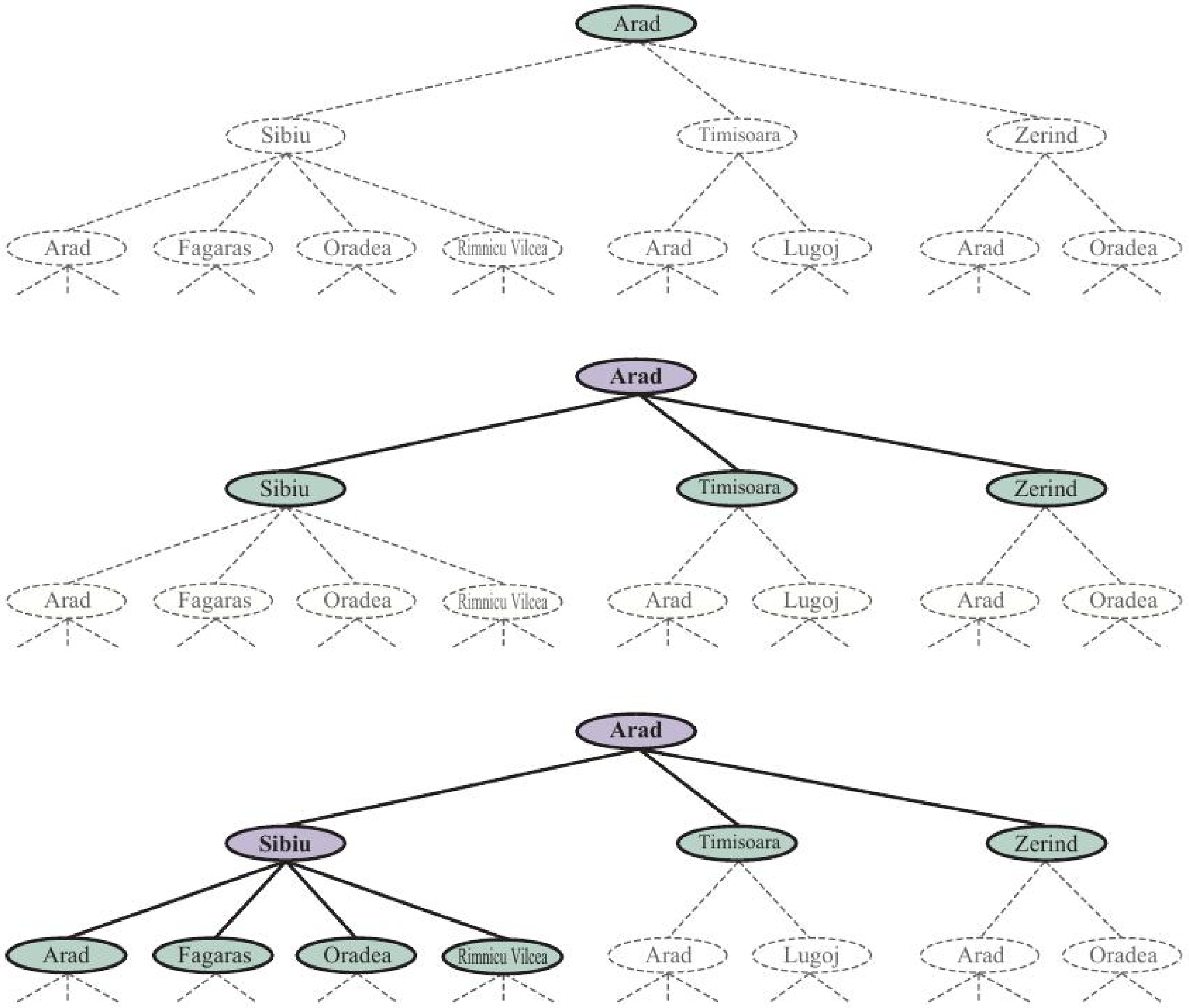


Figure 3.4 Three partial search trees for finding a route from *Arad* to *Bucharest*.

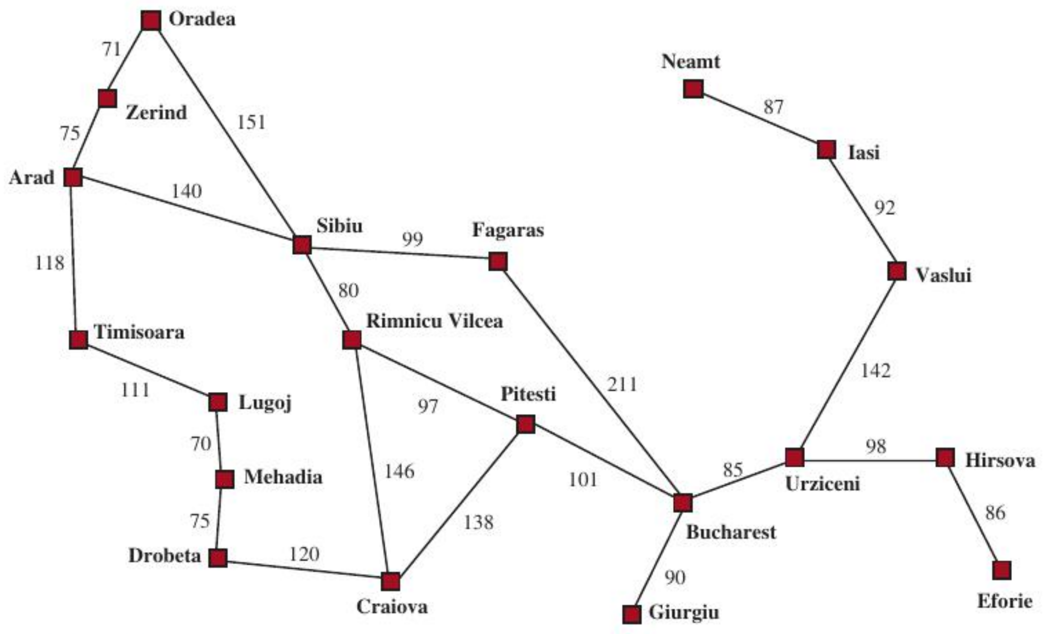


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

- The search tree shown in Figure 3.4 (bottom) includes a path from *Arad* to *Sibiu* and back to *Arad* again. We say that *Arad* is a repeated state in the search tree, generated in this case by a **cycle** (also known as a **loopy path**). So even though the state space has only 20 states, the complete search tree is *infinite* because there is no limit to how often one can traverse a loop.

- A cycle is a special case of a **redundant path**.

3. Search Algorithms

Redundant Paths

- For example, we can get to *Sibiu* via the path *Arad–Sibiu* (140 miles long) or the path *Arad–Zerind–Oradea–Sibiu* (297 miles long).
- This second path is redundant—it's just a worse way to get to the same state—and need not be considered in our quest for optimal paths.
- *Algorithms that cannot remember the past are doomed to repeat it.* There are three approaches to this issue.

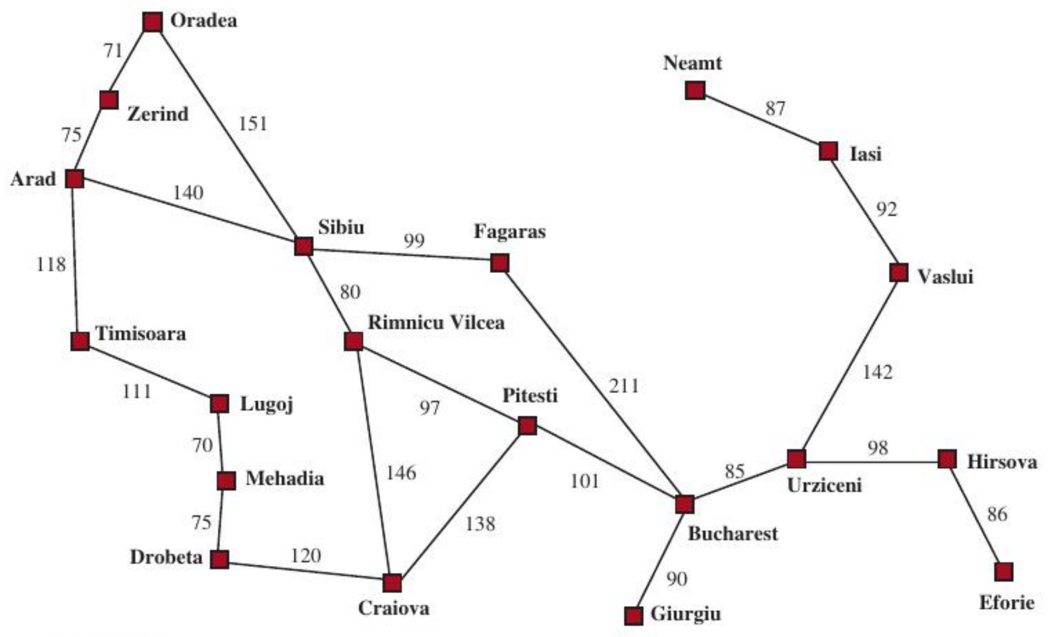


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

3. Search Algorithms
Redundant Paths

- First, we can remember all previously reached states (as best-first search does), allowing us to detect all redundant paths, and keep only the best path to each state.
  - This is appropriate for state spaces where there are many redundant paths, and is the preferred choice when the table of reached states will fit in memory.
- Second, we can not worry about repeating the past. There are some problem formulations where it is rare or impossible for two paths to reach the same state. For those problems, we could save memory space if we don't track reached states and we don't check for redundant paths.
  - We call a search algorithm a **graph search** if it checks for redundant paths and a **tree-like search** if it does not check.
  - The BEST-FIRST-SEARCH algorithm is a graph search algorithm; if we remove all references to *reached* we get a tree-like search that uses less memory but will examine redundant paths to the same state, and thus will run slower.

3. Search Algorithms
Redundant Paths

- Third, we can compromise and check for cycles, but not for redundant paths in general.
  - Since each node has a chain of parent pointers, we can check for cycles with no need for additional memory by following up the chain of parents to see if the state at the end of the path has appeared earlier in the path.
  - Some implementations follow this chain all the way up, and thus eliminate all cycles; other implementations follow only a few links (e.g., to the parent, grandparent, and great-grandparent), and thus take only a constant amount of time, while eliminating all short cycles (and relying on other mechanisms to deal with long cycles).

3. Search Algorithms

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, and to correctly report failure when there is not?
  - **Cost optimality**: Does it find a solution with the lowest path cost of all solutions?
  - **Time complexity**: How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
  - **Space complexity**: How much memory is needed to perform the search?
- To be complete, a search algorithm must be **systematic** in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state.

4. Uninformed Search Strategies

- An uninformed search algorithm is given no clue about how close a state is to the goal(s).
  - For example, consider our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step.
  - In contrast, an informed agent who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.

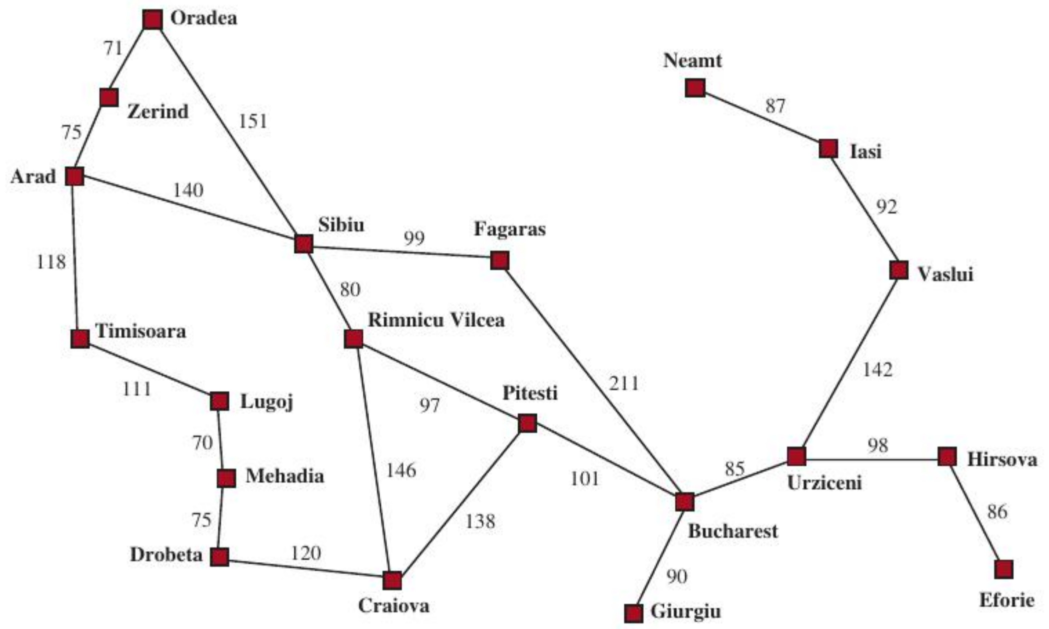


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

4. Uninformed Search Strategies

Breadth-First Search

- When all actions have the same cost, an appropriate strategy is **breadth-first search**, in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- This is a systematic search strategy that is therefore complete even on infinite state spaces.

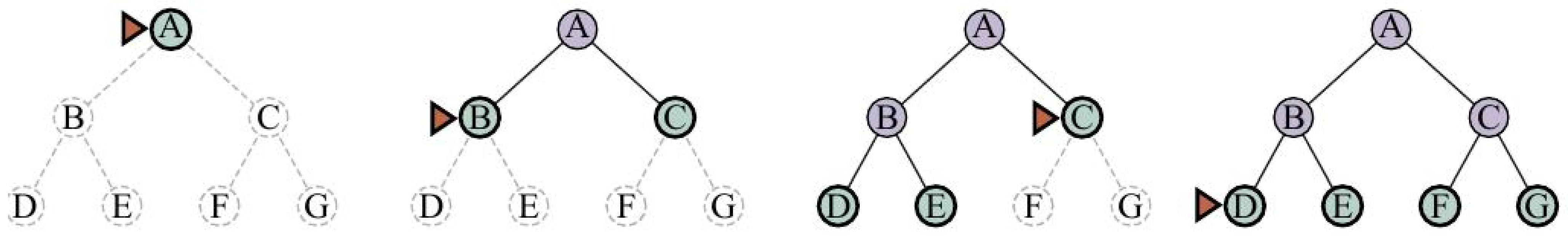


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

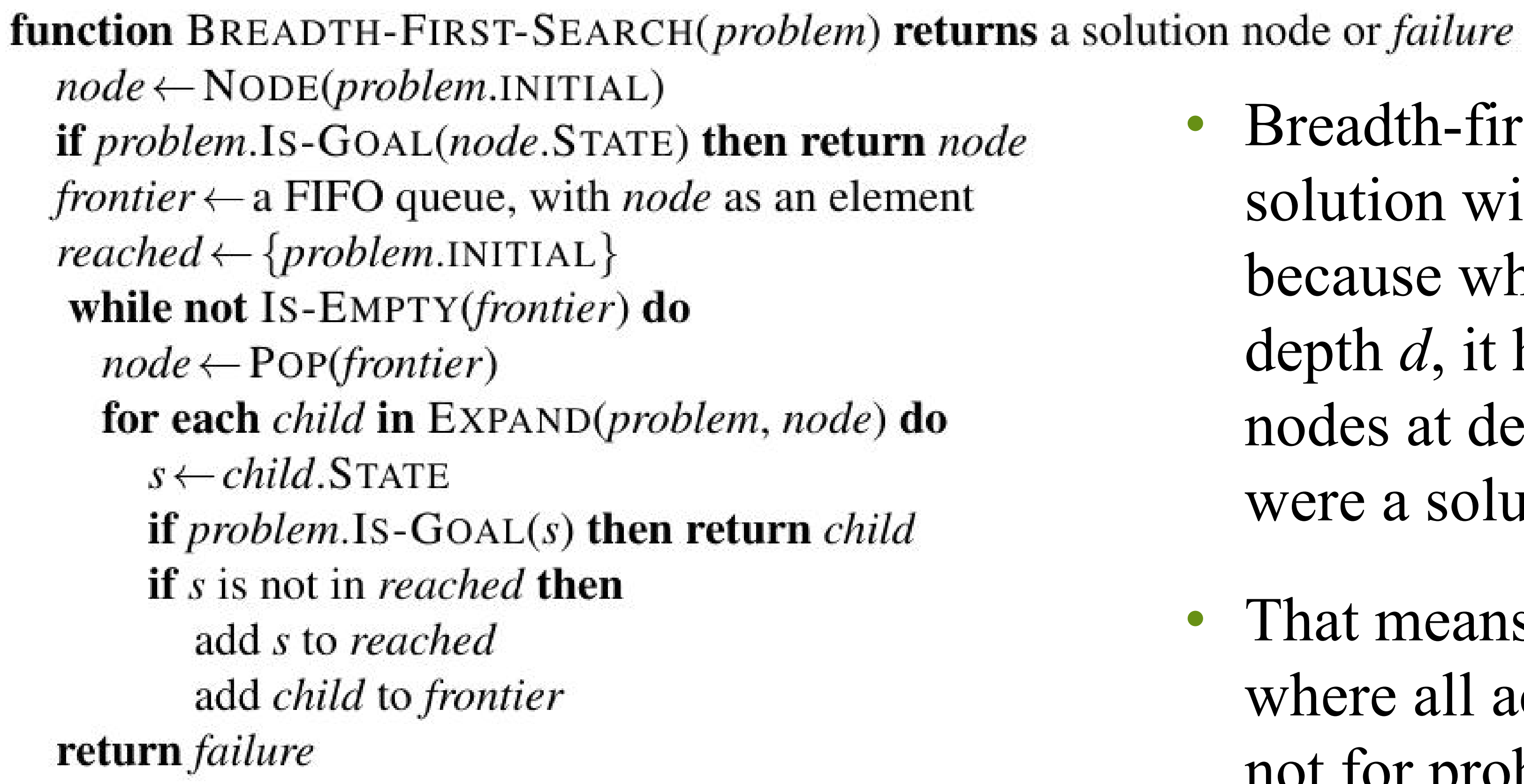
4. Uninformed Search Strategies

Breadth-First Search

- We could implement breadth-first search as a call to BEST-FIRST-SEARCH where
  - The evaluation function $f(n)$ is the depth of the node—that is, the number of actions it takes to reach the node.
  - A first-in-first-out queue will be faster than a priority queue, and will give us the correct order of nodes: 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.
  - In addition, *reached* can be a set of states rather than a mapping from states to nodes, because once we've reached a state, we can never find a better path to the state. That also means we can do an **early goal test**, checking whether a node is a solution as soon as it is generated, rather than the **late goal test** that best-first search uses, waiting until a node is popped off the queue.

4. Uninformed Search Strategies

Breadth-First Search



```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

- Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth *d*, it has already generated all the nodes at depth $d - 1$, so if one of them were a solution, it would have been found.
- That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don’t have that property. It is complete in either case.

4. Uninformed Search Strategies
Breadth-First Search

- In terms of time and space, imagine searching a uniform tree where every state has b successors.
  - The root of the search tree generates b nodes, 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. Then the total number of nodes generated is

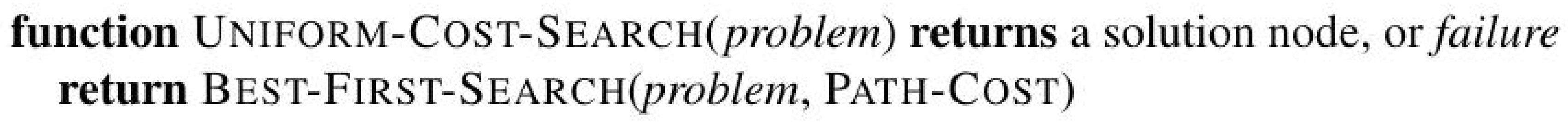
$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

  - All the nodes remain in memory, so both time and space complexity are $O(b^d)$. Exponential bounds like that are scary.
    - As a typical real-world example, consider a problem with branching factor $b = 10$, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth $d = 10$ would take less than 3 hours, but would require 10 terabytes of memory. *The memory requirements are a bigger problem for breadth-first search than the execution time.*
    - At depth $d = 14$, even with infinite memory, the search would take 3.5 years.
  - In general, *exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.*

4. Uninformed Search Strategies

Dijkstra’s Algorithm or Uniform-Cost Search

- When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra’s algorithm by the theoretical computer science community, and **uniform-cost search** by the AI community.
- The algorithm can be implemented as a call to B EST-FIRST-SEARCH with PATH-COST as the evaluation function



```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

4. Uninformed Search Strategies

Dijkstra's Algorithm or Uniform-Cost Search

- The complexity of uniform-cost search is characterized in terms of $C*$, the cost of the optimal solution, and ε, a lower bound on the cost of each action, with $\varepsilon > 0$.
- The algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, which can be much greater than b^d.
  - This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action.
  - When all action costs are equal, $b^{1+\lfloor C^*/\varepsilon \rfloor}$ is just b^{1+d}, and uniform-cost search is similar to breadth-first search.
- Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier.

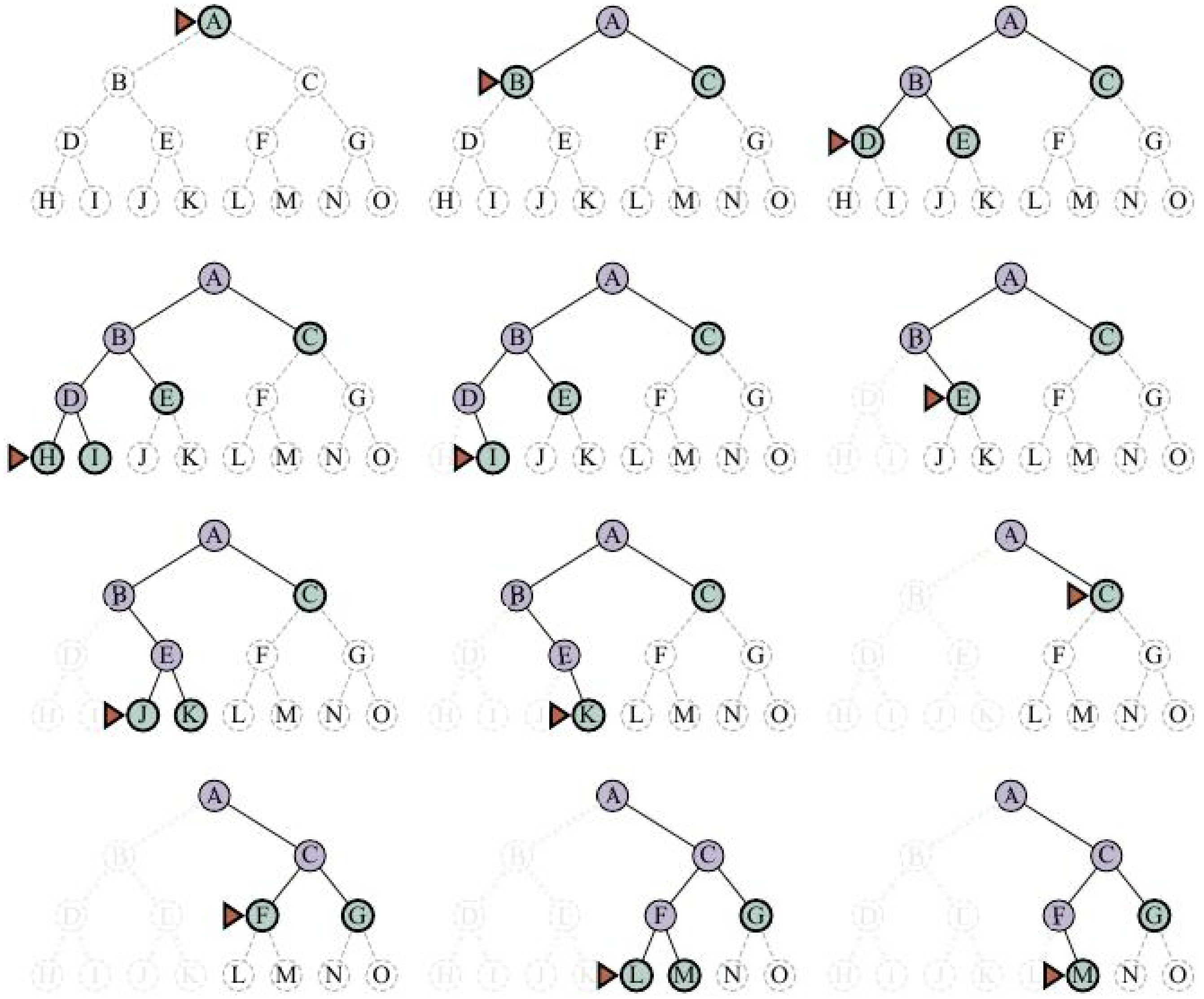
4. Uninformed Search Strategies

Depth-First Search and the Problem of Memory

- **Depth-first search** always expands the deepest node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function *f* is the negative of the depth.
  - However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states.
- Search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then “backs up” to the next deepest node that still has unexpanded successors.
- Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

4. Uninformed Search Strategies

Depth-First Search and the Problem of Memory



- **Figure 3.11** A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M.
- The frontier is in green, with a triangle marking the node to be expanded next.
- Previously expanded nodes are lavender, and potential future nodes have faint dashed lines.
- Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

4. Uninformed Search Strategies

Depth-First Search and the Problem of Memory

- For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.
- In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles.
- Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete.

4. Uninformed Search Strategies

Depth-First Search and the Problem of Memory

- With all this bad news, why would anyone consider using depth-first search rather than breadth-first or best-first?
  - The answer is that for problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don’t keep a *reached* table at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere.
  - For a finite tree-shaped state-space like the one in Figure 3.11, a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only $O(bm)$, where b is the branching factor and m is the maximum depth of the tree.

4. Uninformed Search Strategies

Depth-First Search and the Problem of Memory

- A variant of depth-first search called **backtracking** search uses even less memory.
  - In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by modifying the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of $O(m)$ actions; a significant savings over $O(bm)$ states for depth-first search.
  - With backtracking we also have the option of maintaining an efficient set data structure for the states on the current path, allowing us to check for a cyclic path in $O(1)$ time rather than $O(m)$.
  - For backtracking to work, we must be able to undo each action when we backtrack. Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly.

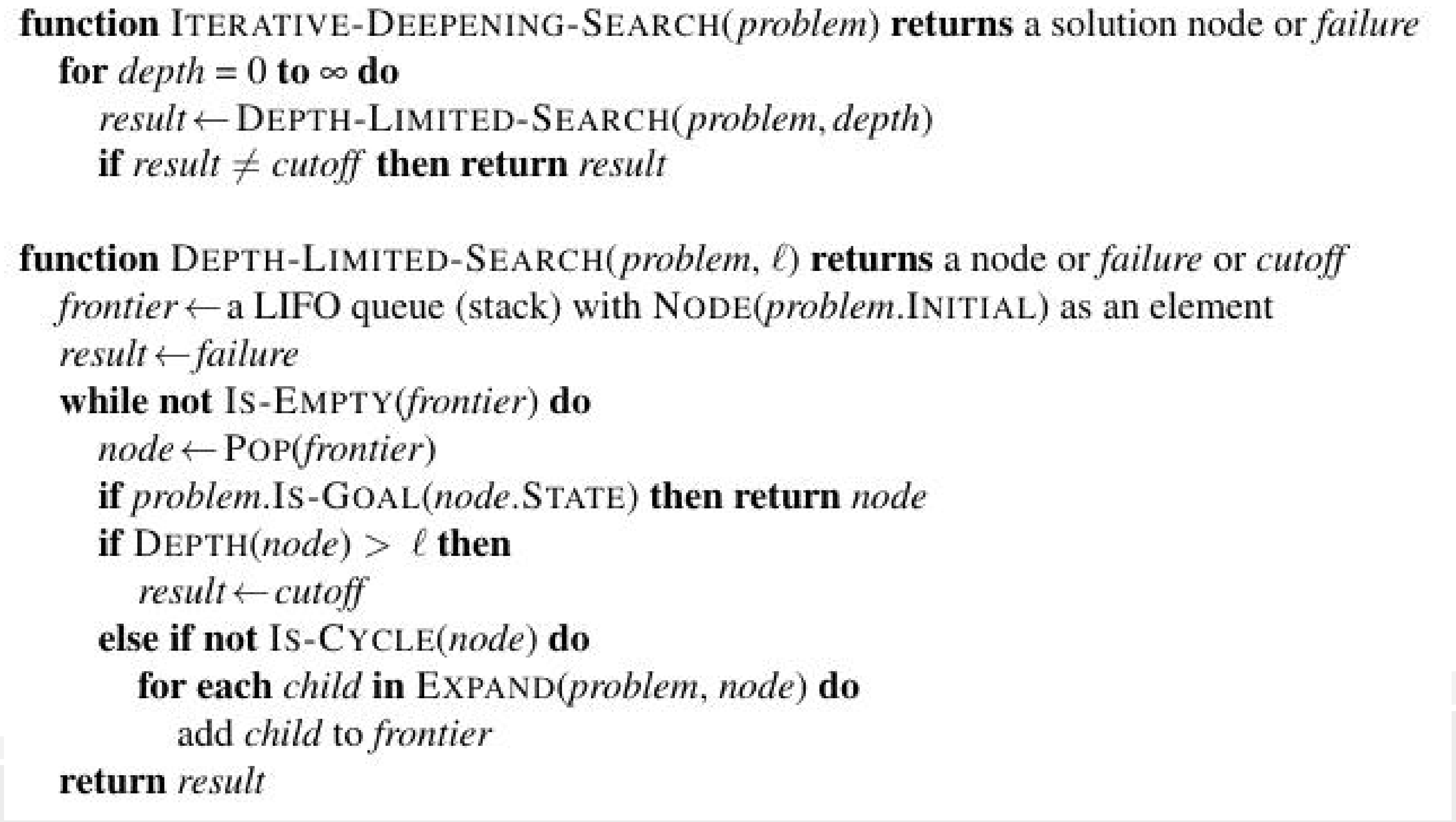
4. Uninformed Search Strategies

Depth-Limited and Iterative Deepening Search

- To keep depth-first search from wandering down an infinite path, we can use **depth-limited search**, a version of depth-first search in which we supply a depth limit, *l*, and treat all nodes at depth *l* as if they had no successors.
- The time complexity is $O(b^l)$ and the space complexity is $O(bl)$.
- Unfortunately, if we make a poor choice for *l* the algorithm will fail to reach the solution, making it incomplete again.

4. Uninformed Search Strategies

Depth-Limited and Iterative Deepening Search



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
    for depth = 0 to $\infty$ do
        result $\leftarrow$ DEPTH-LIMITED-SEARCH(problem, depth)
        if result $\neq$ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, $\ell$) returns a node or failure or cutoff
    frontier $\leftarrow$ a LIFO queue (stack) with NODE(problem.INITIAL) as an element
    result $\leftarrow$ failure
    while not IS-EMPTY(frontier) do
        node $\leftarrow$ POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        if DEPTH(node) $>$ $\ell$ then
            result $\leftarrow$ cutoff
        else if not IS-CYCLE(node) do
            for each child in EXPAND(problem, node) do
                add child to frontier
    return result
```

4. Uninformed Search Strategies

Depth-Limited and Iterative Deepening Search

- Sometimes a good depth limit can be chosen based on knowledge of the problem.
  - For example, on the map of Romania there are 20 cities. Therefore, $l = 19$ is a valid limit.
  - But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions. This number, known as the **diameter** of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search.
  - However, for most problems we will not know a good depth limit until we have solved the problem.

4. Uninformed Search Strategies

Depth-Limited and Iterative Deepening Search

- **Iterative deepening search** solves the problem of picking a good value for l by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the *failure* value rather than the *cutoff* value.

- Iterative deepening combines many of the benefits of depth-first and breadth-first search.
  - Like depth-first search, its memory requirements are modest: $O(bd)$ when there is a solution, or $O(bm)$ on finite state spaces with no solution.
  - Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.
  - The time complexity is $O(b^d)$ when there is a solution, or $O(bm)$ when there is none.

- *In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known.*

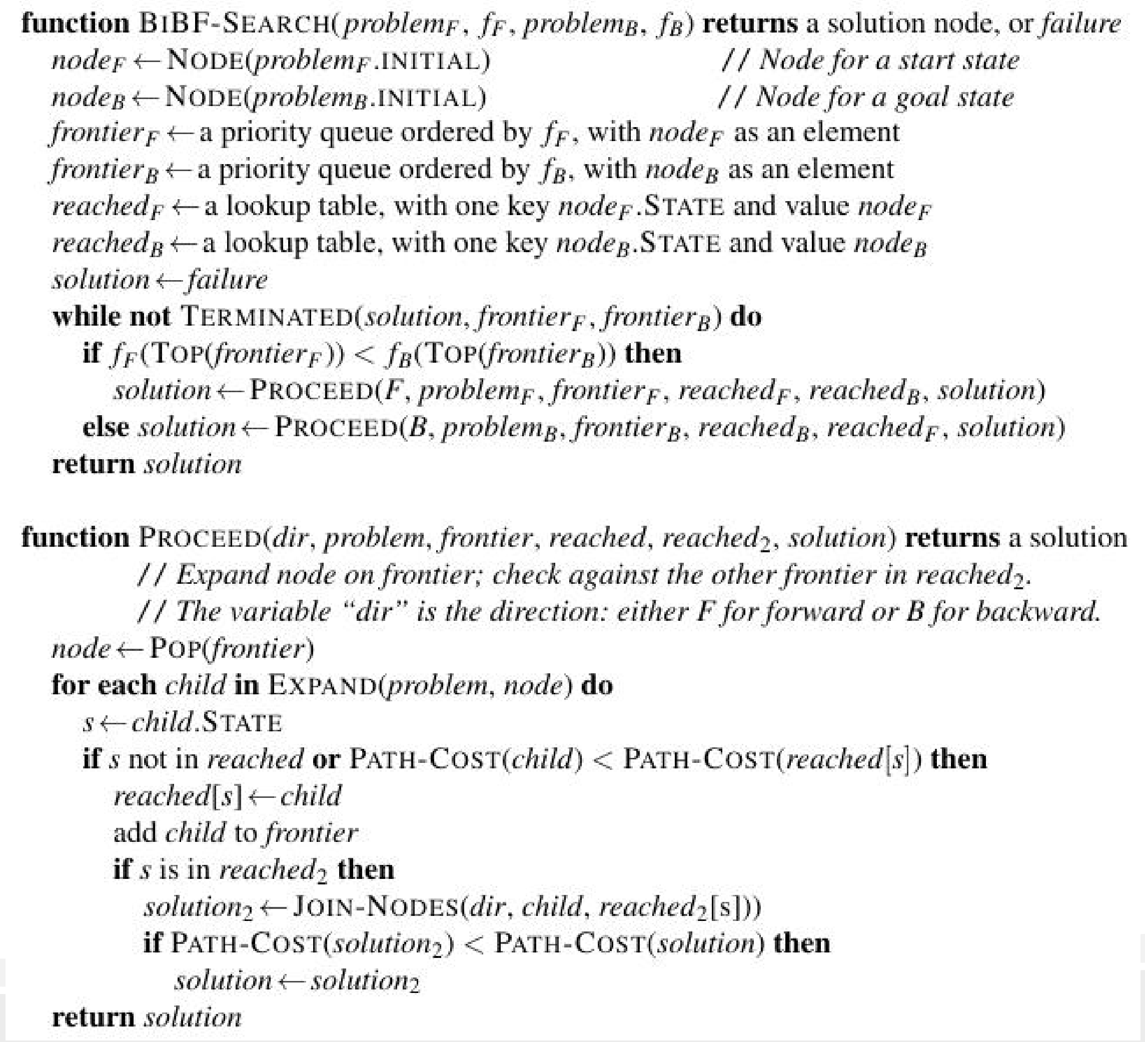
4. Uninformed Search Strategies

Bidirectional Search

- The algorithms we have covered so far start at an initial state and can reach any one of multiple possible goal states. An alternative approach called **bidirectional search** simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.
  - The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^{d} (e.g., 50,000 times less when $b = d = 10$).

4. Uninformed Search Strategies

Bidirectional Search



```
function BIBF-SEARCH($problem_F$, $f_F$, $problem_B$, $f_B$) returns a solution node, or failure
    $node_F \leftarrow$ NODE($problem_F$.INITIAL)                // Node for a start state
    $node_B \leftarrow$ NODE($problem_B$.INITIAL)                // Node for a goal state
    $frontier_F \leftarrow$ a priority queue ordered by $f_F$, with $node_F$ as an element
    $frontier_B \leftarrow$ a priority queue ordered by $f_B$, with $node_B$ as an element
    $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
    $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
    $solution \leftarrow failure$
    while not TERMINATED($solution$, $frontier_F$, $frontier_B$) do
        if $f_F$(TOP($frontier_F$)) $<$ $f_B$(TOP($frontier_B$)) then
            $solution \leftarrow$ PROCEED($F$, $problem_F$, $frontier_F$, $reached_F$, $reached_B$, $solution$)
        else $solution \leftarrow$ PROCEED($B$, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
    return $solution$

function PROCEED($dir$, $problem$, $frontier$, $reached$, $reached_2$, $solution$) returns a solution
            // Expand node on frontier; check against the other frontier in $reached_2$.
            // The variable "dir" is the direction: either F for forward or B for backward.
    $node \leftarrow$ POP($frontier$)
    for each $child$ in EXPAND($problem$, $node$) do
        $s \leftarrow child$.STATE
        if $s$ not in $reached$ or PATH-COST($child$) $<$ PATH-COST($reached[s]$) then
            $reached[s] \leftarrow child$
            add $child$ to $frontier$
            if $s$ is in $reached_2$ then
                $solution_2 \leftarrow$ JOIN-NODES($dir$, $child$, $reached_2$[s])
                if PATH-COST($solution_2$) $<$ PATH-COST($solution$) then
                    $solution \leftarrow solution_2$
    return $solution$
```

- **Figure 3.14** Bidirectional best-first search keeps two frontiers and two tables of reached states.
- When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution.
- The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

4. Uninformed Search Strategies
Bidirectional Search

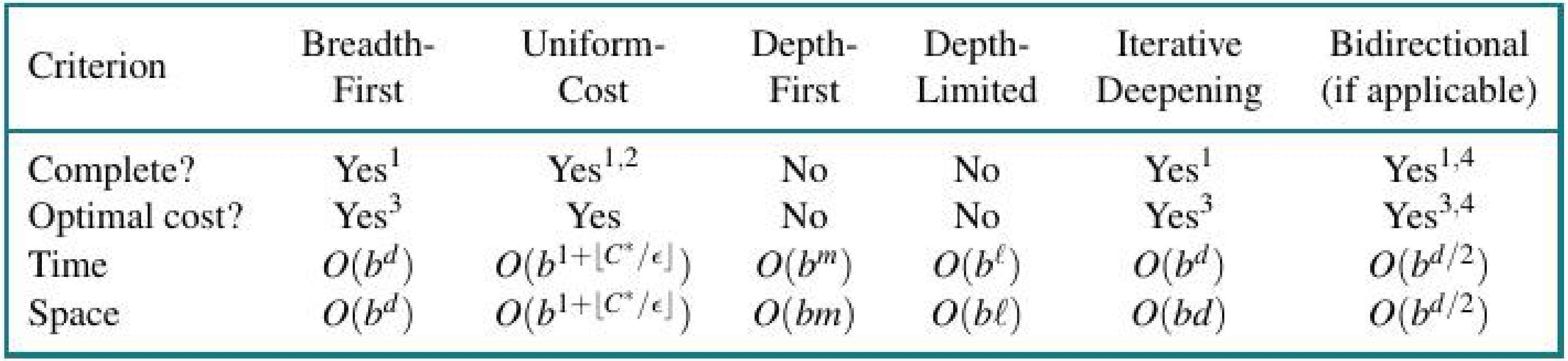
- For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards:
  - If state *s'* is a successor of *s* in the forward direction, then we need to know that *s* is a successor of *s'* in the backward direction. We have a solution when the two frontiers collide.
- There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms.

4. Uninformed Search Strategies
Comparing Uninformed Search Algorithms

- Compares uninformed search algorithms in terms of the four evaluation criteria set forth in Section 3.3.4.
- This comparison is for tree-like search versions which don't check for repeated states. For graph searches which do check, the main differences are that depth-first search is complete for finite state spaces, and the space and time complexities are bounded by the size of the state space (the number of vertices and edges, $|V| + |E|$).

4. Uninformed Search Strategies

Comparing Uninformed Search Algorithms



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

- **Figure 3.15** Evaluation of search algorithms. *b* is the branching factor; *m* is the maximum depth of the search tree; *d* is the depth of the shallowest solution, or is *m* when there is no solution; *l* is the depth limit.

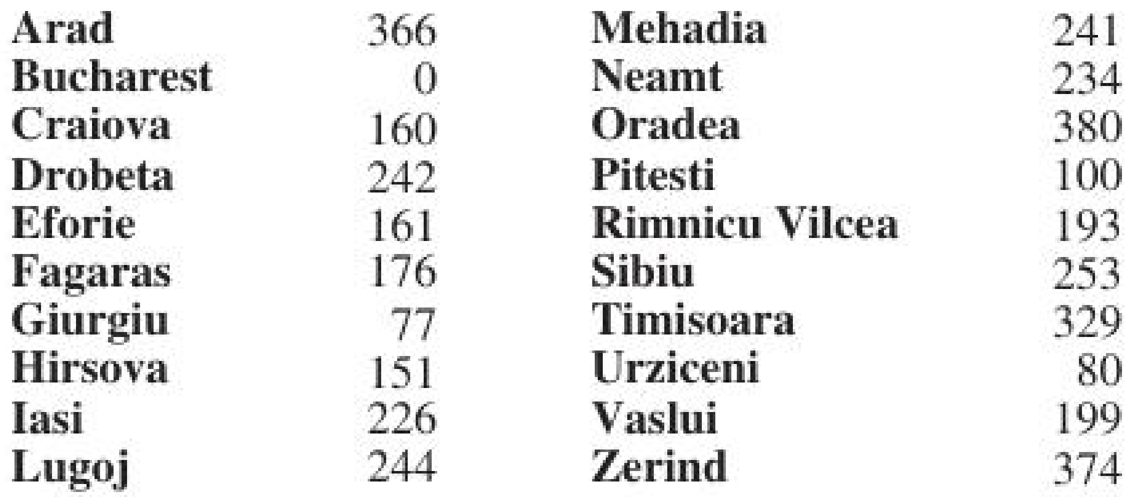
- Superscript caveats are as follows:
  - 1: Complete if *b* is finite, and the state space either has a solution or is finite.
  - 2: Complete if all action costs are $\geq \ \varepsilon > 0$;
  - 3: Cost-optimal if action costs are all identical;
  - 4: If both directions are breadth-first or uniform-cost.

5. Informed (Heuristic) Search Strategies

- This section shows how an **informed search** strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than 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.

- For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points.



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

5. Informed (Heuristic) Search Strategies

Greedy Best-First Search

- **Greedy best-first search** is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly.
  - So the evaluation function $f(n) = h(n)$
- *The algorithm is called "greedy" because on each iteration it tries to get as close to a goal as it can. But greediness can lead to worse results than being careful.*
- Greedy best-first graph search is complete in finite state spaces, but not in infinite ones.
- The worst-case time and space complexity is $O(|V|)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

5. Informed (Heuristic) Search Strategies

Greedy Best-First Search

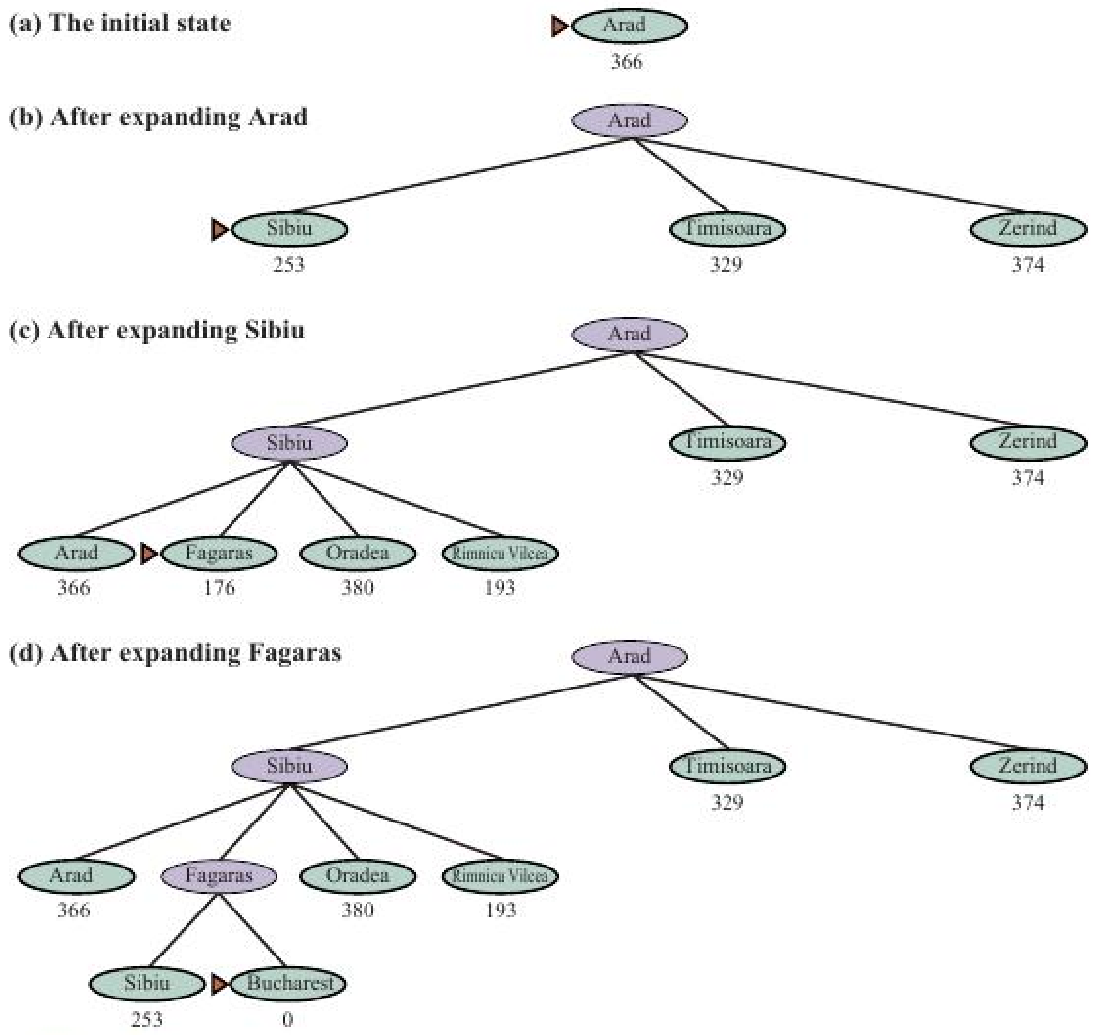


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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.

- Greedy best-first search: Arad→Sibiu→Fagaras→Bucharest
  - Not optimal, 32 miles longer than
    - Arad→Sibiu→Rimnicu Lilcea→Pitesti→Bucharest

5. Informed (Heuristic) Search Strategies

A* Search

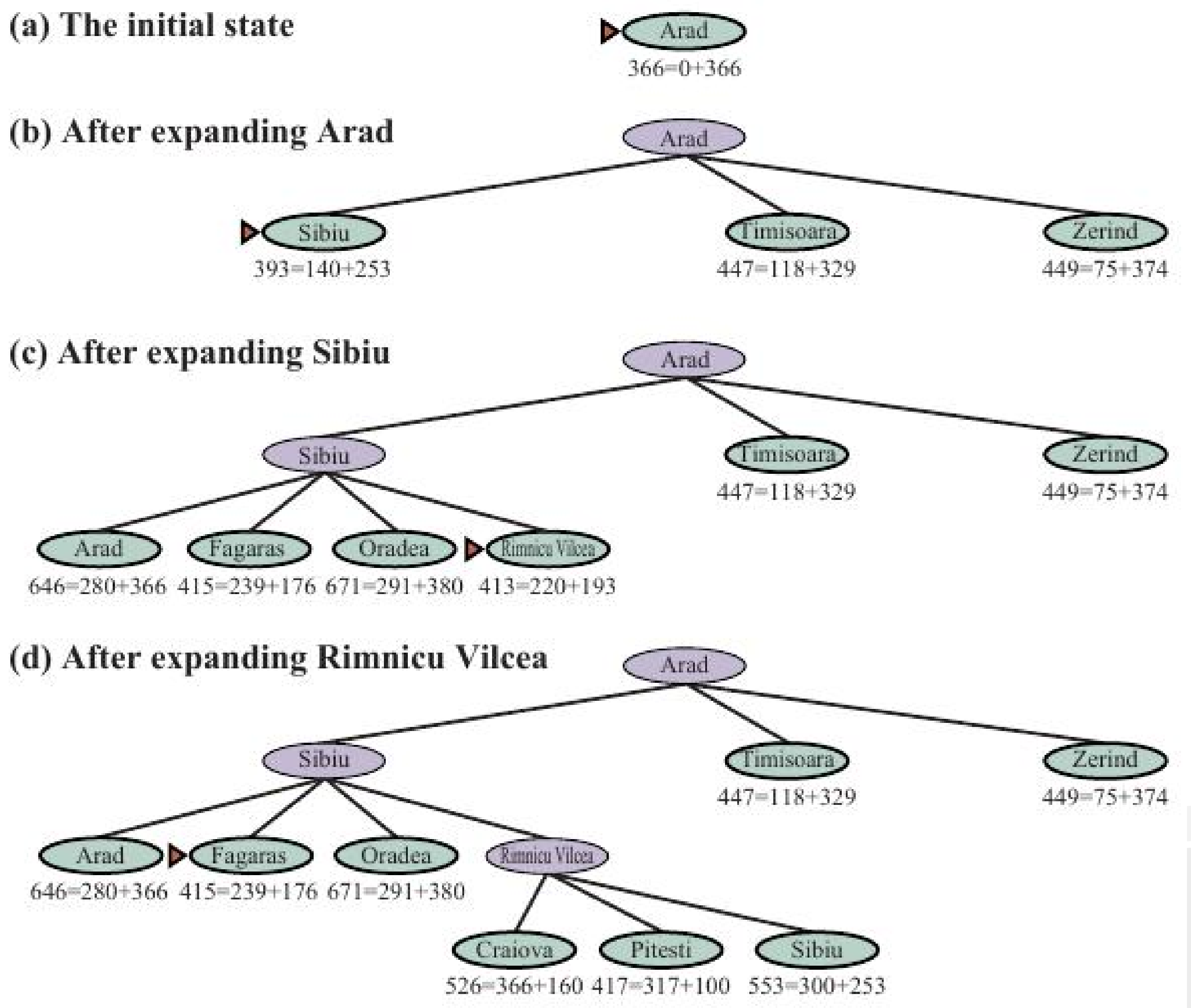
- The most common informed search algorithm is **A* search**, a best-first search that uses the evaluation function
  - $\boldsymbol{f(n) = g(n) + h(n)}$

  where $g(n)$ is the *path cost* from the initial state to node n, and $h(n)$ is the *estimated* cost of the shortest path from n to a goal state.

  - $f(n)$ = estimated cost of the best path that continues from n to a goal.

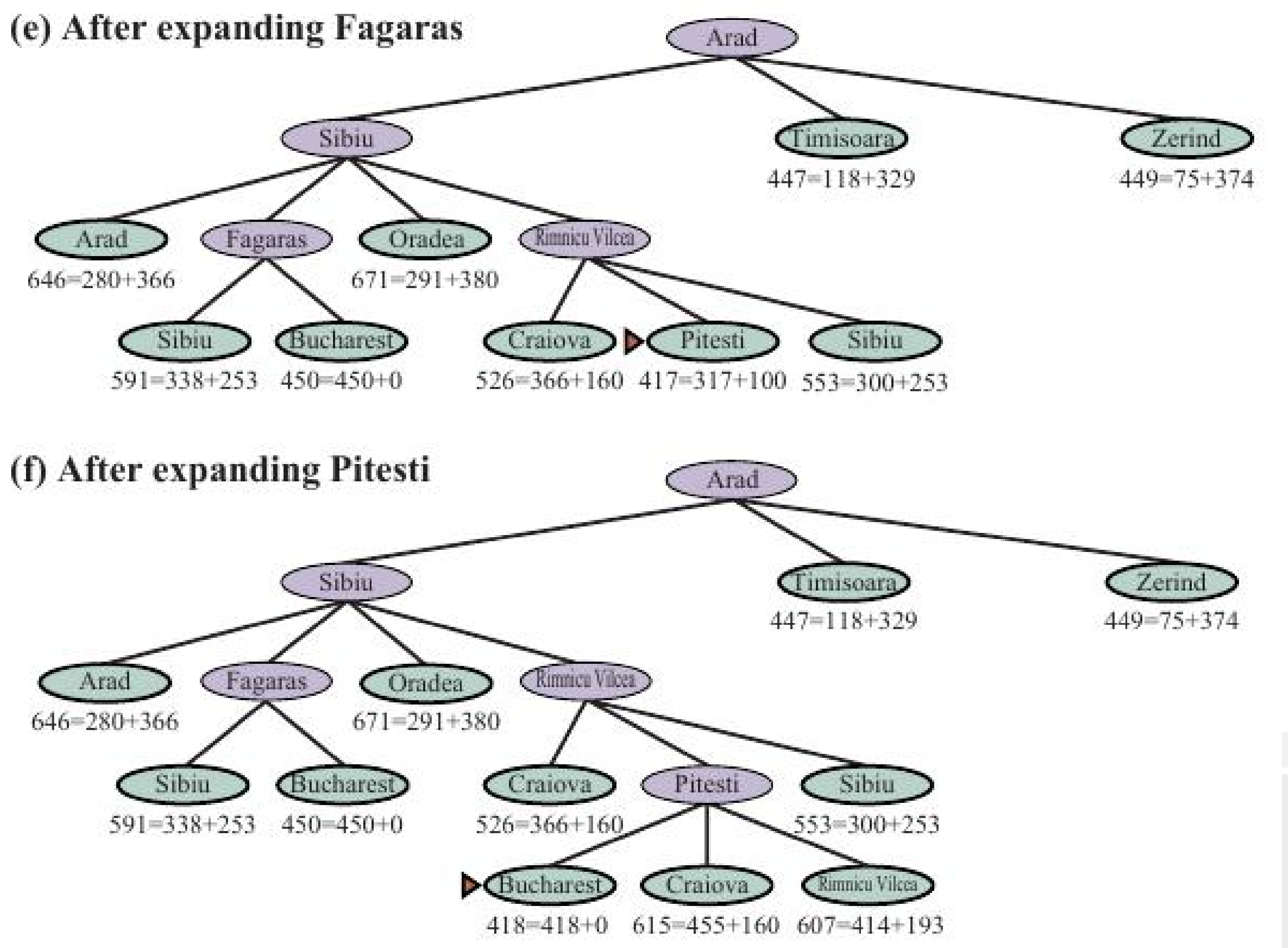
5. Informed (Heuristic) Search Strategies

A* Search



5. Informed (Heuristic) Search Strategies

A* Search



5. Informed (Heuristic) Search Strategies

A* Search

- A* search is complete.
- Whether A* is cost-optimal depends on certain properties of the heuristic.
  - **Admissibility**: an **admissible heuristic** is one that *never overestimates* the cost to reach a goal. (An admissible heuristic is therefore *optimistic*.)
    - With an admissible heuristic, A* is cost-optimal.
  - **Consistency** (stronger condition)**:** A heuristic $h(n)$ is consistent if, for every node *n* and every successor *n'* of *n* generated by an action *a*, we have:

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

This is a form of the **triangle inequality**.

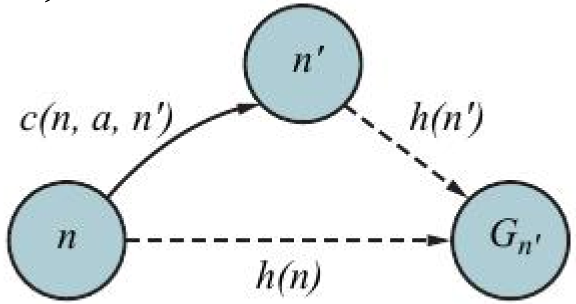


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n,a,a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

5. Informed (Heuristic) Search Strategies

A* Search

- Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A^* is cost-optimal.
- With an inadmissible heuristic, A^* may or may not be cost-optimal. Here are two cases where it is:
  - First, if there is even one cost-optimal path on which $h(n)$ is admissible for all nodes n on the path, then that path will be found, no matter what the heuristic says for states off the path.
  - Second, if the optimal solution has cost C^*, and the second-best has cost C_2, and if $h(n)$ overestimates some costs, but never by more than $C_2 - C^*$, then A^* is guaranteed to return cost-optimal solutions.

5. Informed (Heuristic) Search Strategies

Search Contours

- A useful way to visualize a search is to draw **contours** in the state space, just like the contours in a topographic map.

- Inside the contour labeled 400, all nodes have $f(n) = g(n) + h(n) \leq 400$, and so on.
- Then, because A* expands the frontier node of lowest *f*-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing *f*-cost.

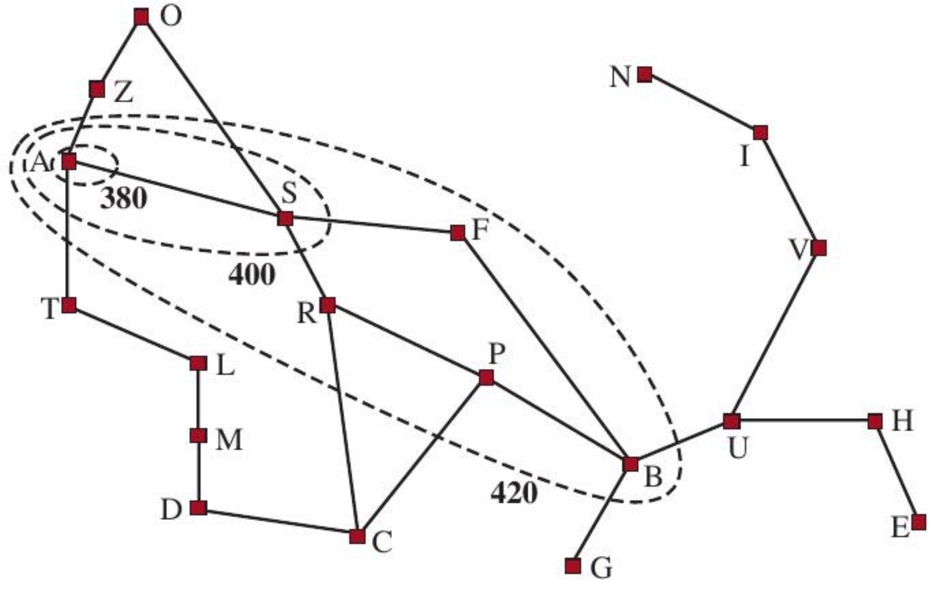


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

5. Informed (Heuristic) Search Strategies

Search Contours

- With uniform-cost search, we also have contours, but of *g*-cost, not $g + h$.
- The contours with uniform-cost search will be “circular” around the start state, spreading out equally in all directions with no preference towards the goal.
- With A* search using a good heuristic, the $g + h$ bands will stretch toward a goal state and become more narrowly focused around an optimal path.

5. Informed (Heuristic) Search Strategies
Satisficing Search: Inadmissible Heuristics and Weighted A*

- A* search has many good qualities, but it expands a lot of nodes.
- We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are "good enough"—what we call **satisficing** solutions.
- If we allow A* search to use an **inadmissible heuristic**—one that may overestimate—then we risk missing the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of nodes expanded.
  - For example, road engineers know the concept of a **detour index**, which is a multiplier applied to the straight-line distance to account for the typical curvature of roads. A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles. For most localities, the detour index ranges between 1.2 and 1.6.

5. Informed (Heuristic) Search Strategies

Satisficing Search: Inadmissible Heuristics and Weighted A*

- We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A* search** where we weight the heuristic value more heavily, giving us the evaluation function

  $f(n) = g(n) + W \times h(n)$, for some $W > 1$.

- In general, if the optimal solution costs C^*, a weighted A* search will find a solution that costs somewhere between C^* and $W \times C^*$; but in practice we usually get results much closer to C^* than $W \times C^*$.

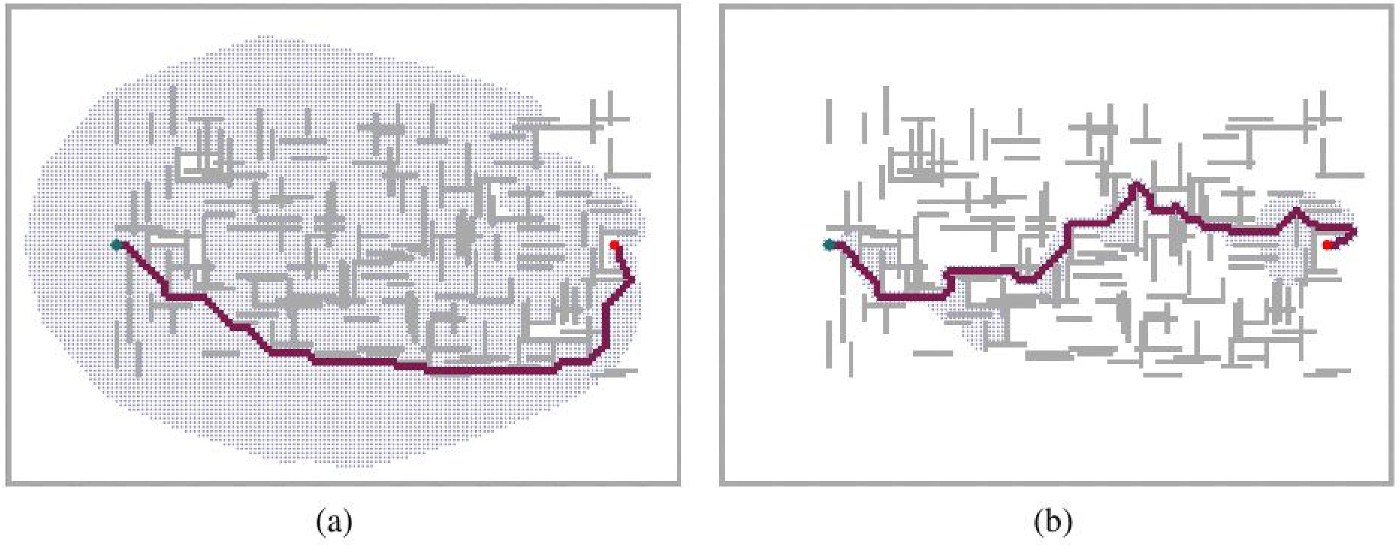


Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

5. Informed (Heuristic) Search Strategies

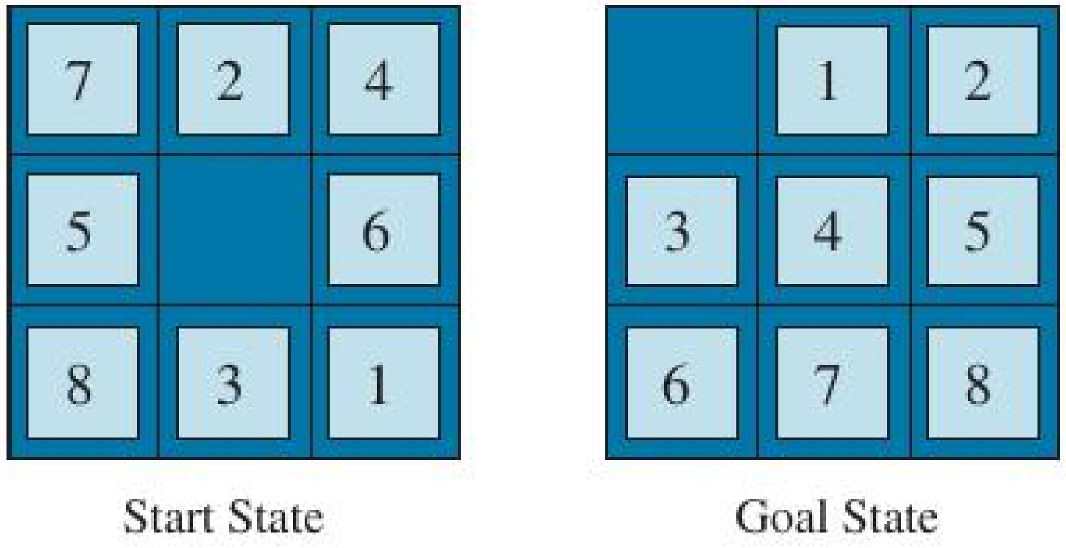
Satisficing Search: Inadmissible Heuristics and Weighted A*

- We have considered searches that evaluate states by combining g and h in various ways; weighted A* can be seen as a generalization of the others:

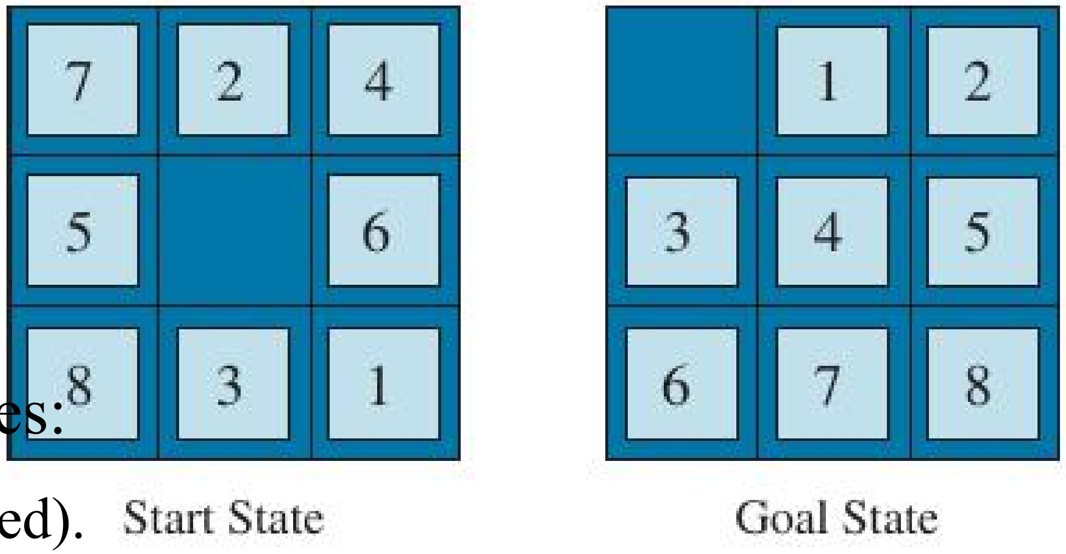
A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform-cost search:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$
Weighted A* search:	$g(n) + W \times h(n)$	$(1 < W < \infty)$

6. Heuristic Functions

- 8-puzzle
  - 9!/2 = 181,400 reachable states → a search could easily keep them all in memory.
- 15-puzzle
  - 16!/2 states—over 10 trillion—so to search that space we will need the help of a good admissible heuristic function.



6. Heuristic Functions



- There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:
  - h_1 = the number of misplaced tiles (blank not included).
    - For Figure 3.25, all eight tiles are out of position, so the start state has h1 = 8.
    - h_1 is an admissible heuristic because any tile that is out of place will require at least one move to get it to the right place.
  - h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance is the sum of the horizontal and vertical distances—sometimes called the **city-block distance** or **Manhattan distance**.
    - h_2 is also admissible because all any move can do is move one tile one step closer to the goal.
    - Tiles 1 to 8 in the start state of Figure 3.25 give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

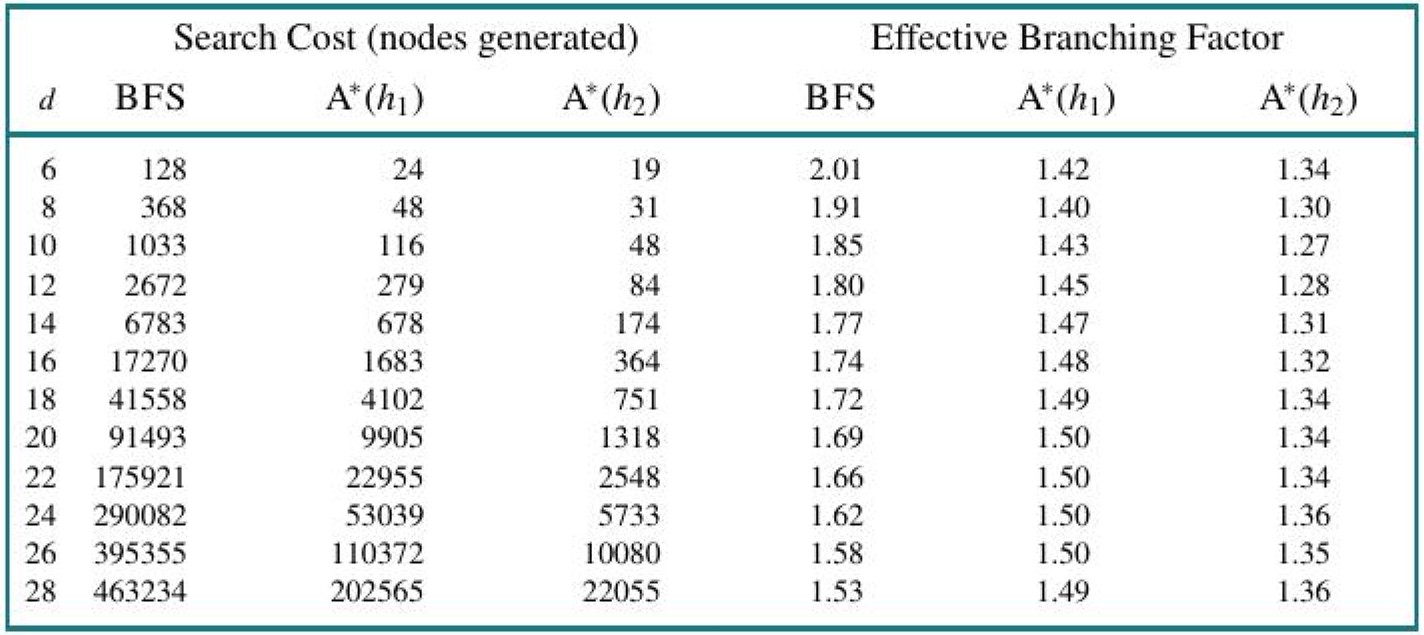
5. Informed (Heuristic) Search Strategies
The Effect of Heuristic Accuracy on Performance

- 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.

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d .$$

5. Informed (Heuristic) Search Strategies

The Effect of Heuristic Accuracy on Performance



	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A^* with h_1 (misplaced tiles), and A^* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

- The results suggest that h_2 is better than h_1, and both are better than no heuristic at all.

5. Informed (Heuristic) Search Strategies
The Effect of Heuristic Accuracy on Performance

- One might ask whether h_2 is always better than h_1. The answer is "Essentially, yes."
- It is easy to see from the definitions of the two heuristics that for any node n, $h_2(n) \geq h_1(n)$.
  - We thus say that $h_2(n)$ **dominates** $h_1(n)$. Domination translates directly into efficiency: A^* using $h_2(n)$ will never expand more nodes than A^* using $h_1(n)$ (except in the case of breaking ties unluckily).

5. Informed (Heuristic) Search Strategies

Generating Heuristics From Relaxed Problems

- We have seen that both h_1 (misplaced tiles) and h_2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h_2 is better.
  - How might one have come up with h_2?
  - Is it possible for a computer to invent such a heuristic mechanically?

5. Informed (Heuristic) Search Strategies

Generating Heuristics From Relaxed Problems

- h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle.
  - If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h_1 would give the exact length of the shortest solution.
  - Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact length of the shortest solution.
- 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.

5. Informed (Heuristic) Search Strategies

Generating Heuristics From Relaxed Problems

- Because the relaxed problem adds edges to the state-space graph, 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 shortcuts. 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.

5. Informed (Heuristic) Search Strategies

Generating Heuristics From Relaxed Problems

- If a collection of admissible heuristics h_1 ... h_m is available for a problem and none of them is clearly better than the others, which should we choose? As it turns out, we can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_k(n)\}\,.$$

- This composite heuristic picks whichever function is most accurate on the node in question. Because the h_i components are admissible, h is admissible (and if h_i are all consistent, h is consistent).
- Furthermore, h dominates all of its component heuristics. The only drawback is that $h(n)$ takes longer to compute.