Course: Introduction to AI

Prof. Ahmed Guessoum
The National Higher School of AI

Chapter 3

Search

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Outline (cont.)

- Informed (Heuristic) Search Strategies
  - Greedy best-first search
  - A* search
  - Memory-bounded heuristic search
  - Learning to search better
- Heuristic Functions
  - The effect of heuristic accuracy on performance
  - Generating admissible heuristics from relaxed problems
  - Learning heuristics from experience

Problem-Solving Agents

- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- **Goal formulation**: first step in problem solving. It is based on the current situation and the agent's performance measure.
- **Goal**: *the* state of world in which the goal is satisfied.
- **Agent's task**: find out how to act, now and in the future, so that it reaches a goal state ➔ what sorts of actions and states it should consider.
- **Problem formulation:** process of deciding what actions and states to consider, given a goal.

Case: Travelling in Romania

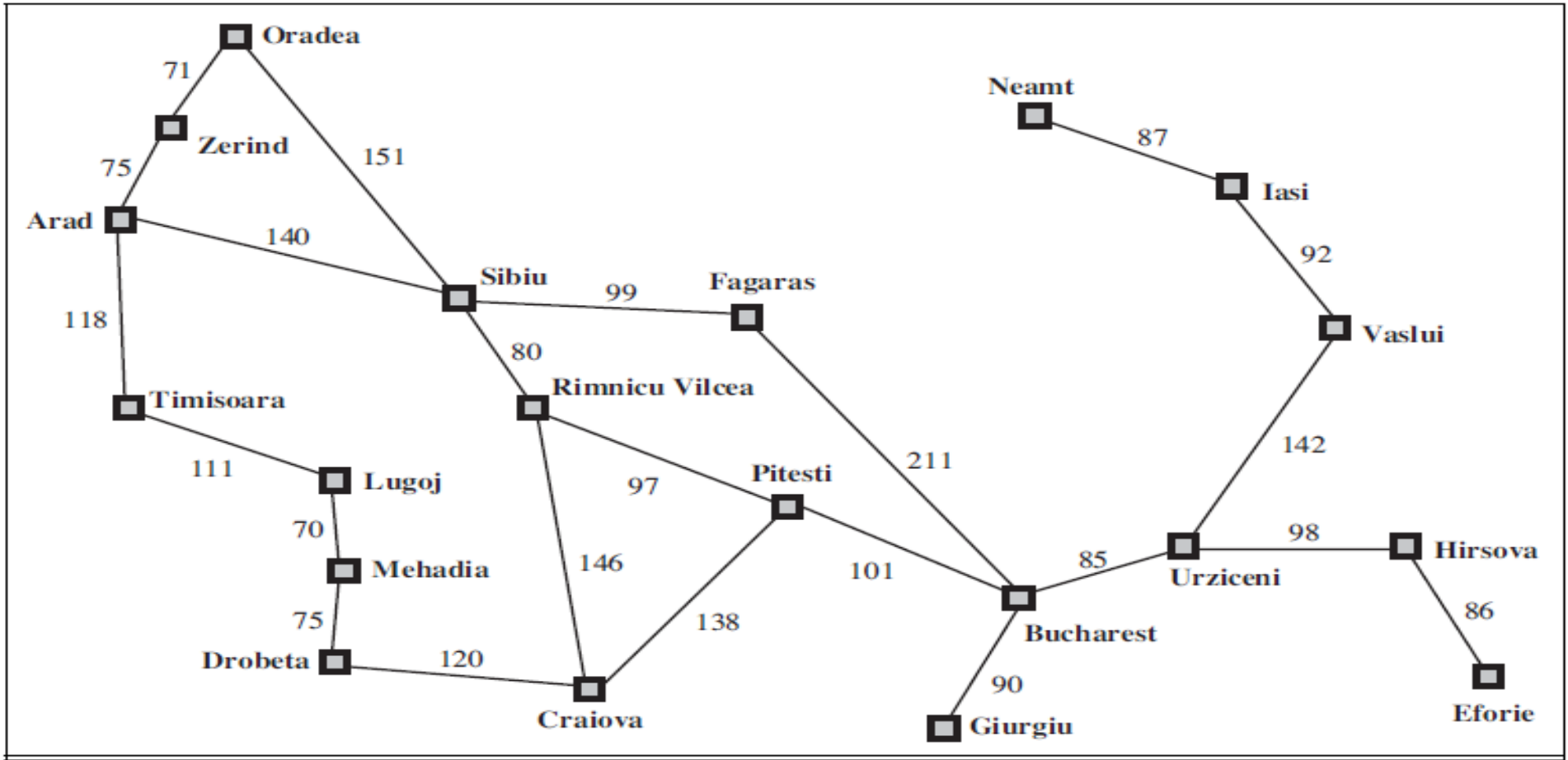


Figure 3.2 A simplified road map of part of Romania.

- Suppose Agent needs to go from Arad to Bucharest.
- Three possible alternative first steps.

Problem-Solving Agents

- If the agent has no additional information, i.e. the environment is **unknown** (Chap. 2), then only choice: try one of the actions at random.
- Suppose the agent has a map of Romania, i.e. the agent has information about the states it might get itself into and the actions it can take, it can find a sequence of states to reach the goal.
- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal by carrying out the driving actions.

Problem Assumptions

We will assume the environment is :

- **Observable**: the agent always knows the current state.
- **Discrete:** at any given state there are only finitely many actions to choose from.
- **Known**: the agent knows which states are reached by each action.
- **Deterministic**: each action has exactly one outcome.
- N.B.: The above are ideal conditions.
- 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 to execute.

Formulate Goal → Formulate problem → Search → Execute

Simple Problem-Solving Agent

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

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

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

Well-defined problems & solutions

Formal definition of a problem:

- The **initial state:** where the agent starts
- A description of all the possible **actions** available to the agent in a given state.
- The **transition model:** description of what each action does.

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 **goal test**: determines whether a given state is a goal state.
- A **path cost** function: assigns a numeric cost to each path. reflects agent's performance measure.

Case of travelling in Romania

- **Initial state:** Arad
- Possible **actions** from *Arad*: {*Go(Sibiu)*, *Go(Timisoara)*, *Go(Zerind)*}
- The **transition model** contains all descriptions like RESULT(In(Arad),Go(Zerind)) = In(Zerind)
- **Goal test**: singleton set {In(Bucharest)}
- **Path cost:** might be length in kilometres.

(In this chapter) We will 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').

Well-defined problems & solutions

- The preceding elements define a problem and can be put into a single data structure that is given as input to a problem-solving algorithm.
- **Solution** to a problem: an action sequence that leads from the initial state to a goal state.
- A **solution quality** is measured by the path cost function
- An **optimal solution** has the lowest path cost among all solutions.

Formulating problems

- In our formulation of a problem (e.g. travelling in Romania), **a lot of details have been left out**: condition of the road, weather, etc.
- The process of removing detail from a representation is called **abstraction**.
- In addition to abstracting the state description, we must abstract the actions themselves.
- A driving action has many effects: takes up time, consumes fuel, generates pollution, … We only consider the change in location, not finer actions.
- The choice of a **good abstraction** involves removing maximum details while retaining validity and ensuring that the abstract actions are easy to carry out.

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Example Problems

Vacuum world:

- **States**: determined by both the agent location and the dirt locations. How many possible states?

  $2 * 2^2$ (in(A, clean), (B, clean)), (in(A, clean), (B, dirty)), etc.

  If *n* locations, $n * 2^n$ possible states
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: just three actions: *Left*, *Right*, and *Suck*. (Larger environments possibly *Up* and *Down)*.
- **Transition model**: expected effects of actions; except moving *Left (Right)* from leftmost (rightmost) square, and *Suck*ing in a clean square: have no effect.
- **Goal test**: check whether all the squares are clean.
- **Path cost**: Each step costs 1 ➔ path cost = number of steps in the path.

State Space of Vacuum World – "Toy problem" version

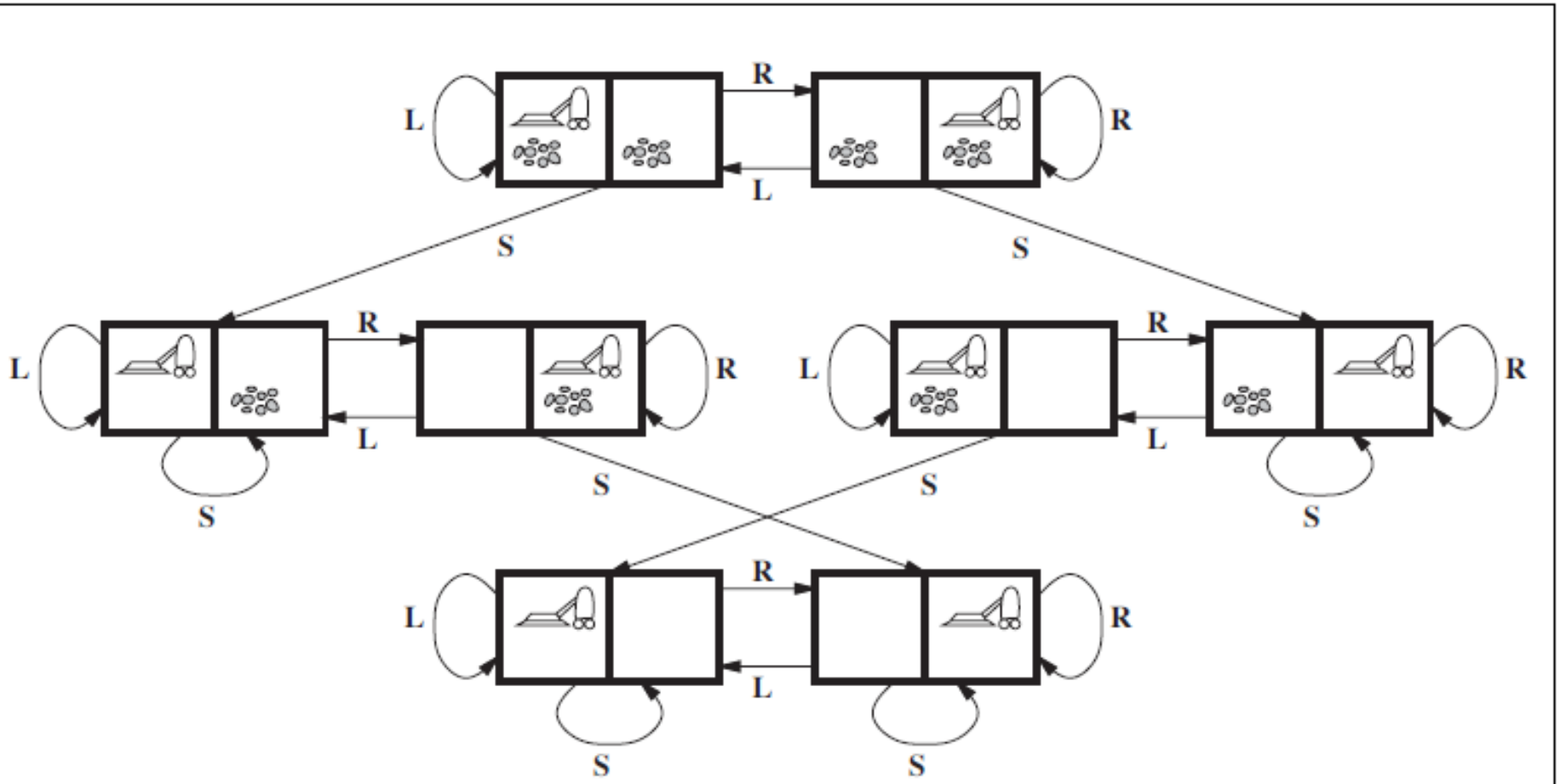


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8-Puzzle problem

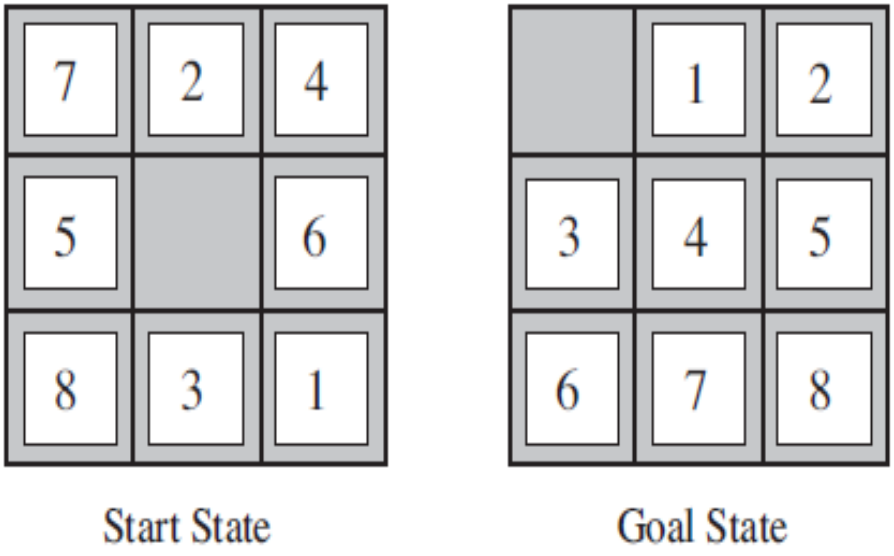


Figure 3.4 A typical instance of the 8-puzzle.

8-Puzzle problem formulation

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Possibly any state.
- **Actions**: simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*.
- **Transition model**: Given a state and action, this returns the resulting state.
- **Goal test**: checks whether the state matches the goal configuration (e.g. the one shown in previous figure).
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Properties of 8-puzzle problem

- The 8-puzzle belongs to the family of **sliding-block puzzles**. This family is known to be NP-Complete.
- The 8-puzzle
  - has 9!/2=181,440 reachable states and
  - is easily solved.
- The 15-puzzle (4×4 board)
  - has around 1.3 trillion states, and
  - random instances can be solved optimally in a few milliseconds by the best search algorithms.
- The 24-puzzle (on a 5 × 5 board)
  - has around 10^{25} states, and
  - random instances take several hours to solve optimally.

The 8-queens problem

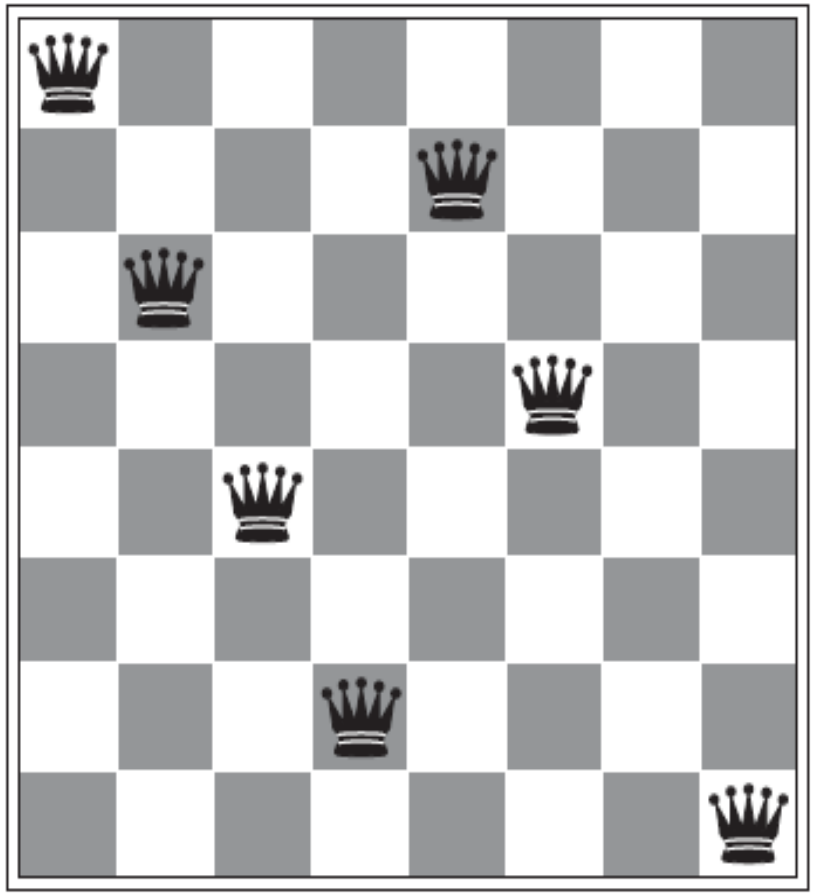


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

The 8-queens problem

- Efficient special-purpose algorithms exist for this problem; but interesting search problem.

Two main kinds of formulation:

- An **incremental formulation** involves operators that *augment* the state description, starting with an empty state. (add a queen at each step)
- **complete-state formulation** starts with all 8 queens on the board and moves them around.

- In both cases, path cost is of no interest; only the final state counts.

8-queens problem - Naïve formulation

The first (naïve) incremental formulation one might try is the following:

- **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.
- In this formulation, we have 64 * 63 * … * 57 ≈ 1.8×10^{14} possible sequences to investigate.

8-queens problem - better formulation

Better formulation: Do not place 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 1.8×10^{14} to just 2,057, and the solutions are easy to find.
- For 100 queens, the reduction is from roughly 10^{400} states to about 10^{52} states, a big improvement, but **not** enough to make the problem **tractable**.

Real-world problems

- The **route-finding problem**: defined in terms of specified locations and transitions along links between them.
- Variety of applications:
  - Relatively straightforward extensions of the Romania example.
  - Those that involve much more complex specifications: routing video streams in computer networks, military operations planning, and airline travel-planning systems,.

Airline travel problems by a travel-planning Web site

- **States**: Each state includes a location (e.g., an airport), the current time, the cost of an action (a flight segment), fare bases, status as domestic or international, etc.
- **Initial state**: specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on monetary cost, waiting time, flight day and flight time, seat quality, type of airplane, frequent-flyer mileage awards, and so on.

Other real-world problems

- **Touring problems:** closely related to route-finding problems. Big difference: e.g. the problem "Visit every city in Romania at least once, starting and ending in Bucharest." ➔ In the state space, each state must include not just the current location but also the *set of cities the agent has visited*.
- The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

Other real-world problems (reading)

- **Traveling salesman problem (TSP)**: touring problem in which each city must be visited **exactly once**.
  - Aim is to find the *shortest* tour.
  - Example Applications: planning movements of automatic circuit-board drills; stocking machines on shop floors.
- **VLSI layout** problem: position millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
  - aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.
  - Channel routing finds a specific route for each wire through the gaps between the cells.
  - Extremely hard problem.

Other real-world problems (reading)

- **Robot navigation:** generalization of the route-finding problem described earlier.
- No following of a discrete set of routes; a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
  - For a robot moving on a flat surface, the space is essentially two-dimensional;
  - If the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional.
  - Advanced techniques are required just to make the search space finite.

Other real-world problems (reading)

Automatic assembly sequencing of complex objects by a robot.

- Aim: find an order in which to assemble the parts of some object.
  - If the wrong order is chosen, no part can later be added in the sequence without undoing some of the work already done.
  - Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation.

  ➔ The generation of legal actions is the expensive part of assembly sequencing.
- The assembly of intricate objects such as electric motors is now economically feasible.
- **Protein design**: important problem where the goal is to find a sequence of amino-acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Searching for Solutions

- A *solution* will be an action sequence to go from the *Start State* to the *Goal State.*
- **Search tree:** the possible action sequences starting at the initial state (the root of the tree); the **branches** are actions and the **nodes** correspond to states in the state space of the problem.
- During the search, at each node (state), the current state is **expanded**; i.e., a new set of states is **generated by** applying each legal action to the current state.

Finding a route from Arad to Bucharest.

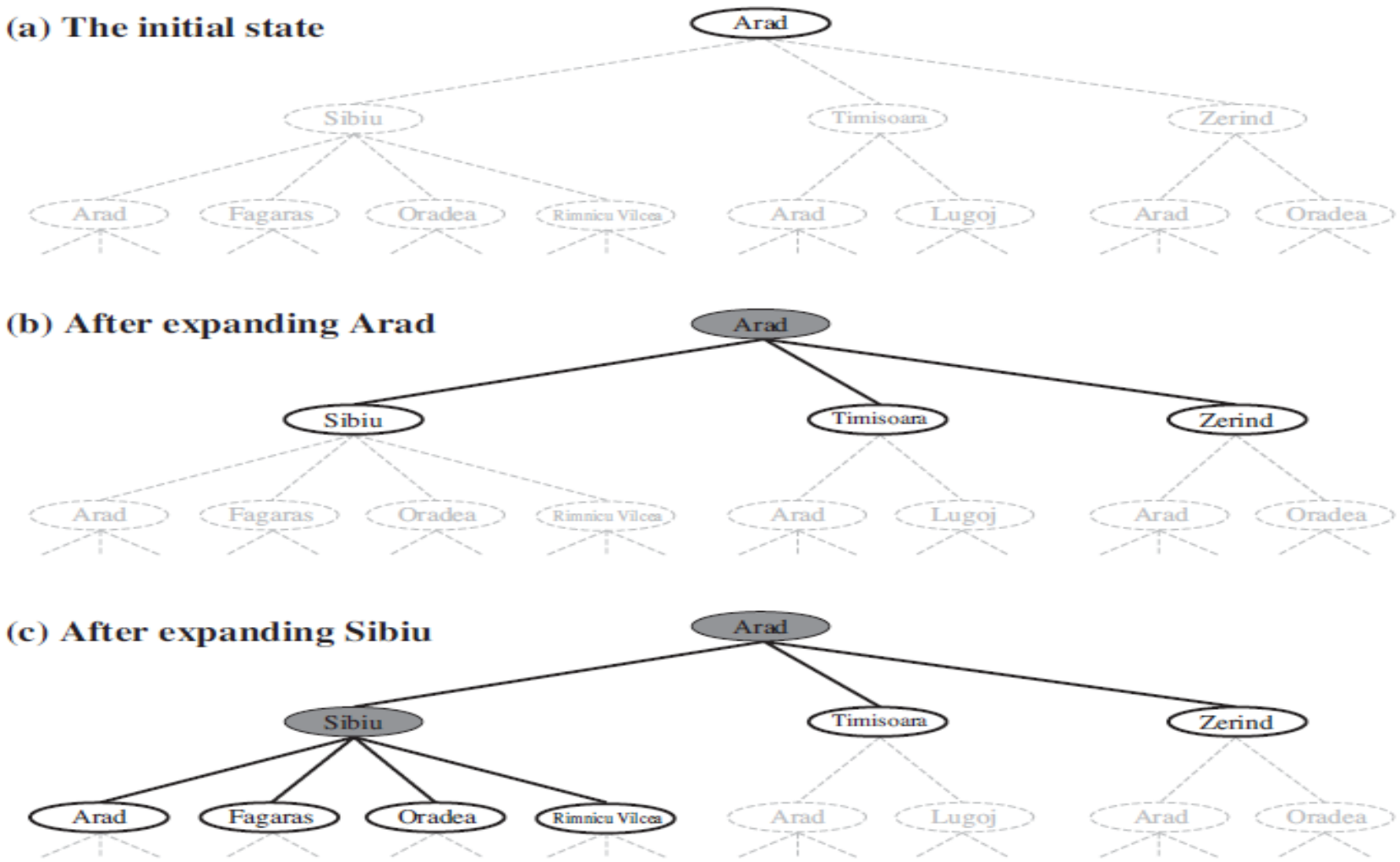


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Searching for Solutions

- A ***key decision***: which of the **children nodes** (of any given **parent node**) do we select for further expansion during the search? E.g. *Sibiu*, *Timisoara*, or *Zerind*?
- We can select one child to expand but keep the other alternatives in case the selected node does not lead to the goal, …
- **Frontier**: the set of all the **leaf** (i.e. not yet expanded) nodes, during the search.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

Searching for Solutions

- Search algorithms vary primarily according to how they choose which state to expand next— called **search strategy.**
- A **loopy path**: a path which contains a **repeated state** (i.e. already explored) ➔ the complete search tree for Romania is *infinite*
- Since path costs are additive and step costs nonnegative, we can avoid falling in loops.
- Loopy paths are a special case of the more general concept of **redundant paths**, which exist whenever there is more than one alternative path. (See Romania map).
- We can define a problem so as to eliminate redundant paths. See 8-Queens problem. From n! different paths to one!

Searching for Solutions

- In other cases, redundant paths are unavoidable. This includes all problems where the actions are reversible.

  e.g. route-finding problems and

  sliding-block puzzles.
- The TREE-SEARCH algorithm can be augmented with a data structure called the **explored set** (or the **closed list**), which remembers every expanded node so as not to repeat expanding it.

General TREE-SEARCH algorithm

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Route finding in Romania

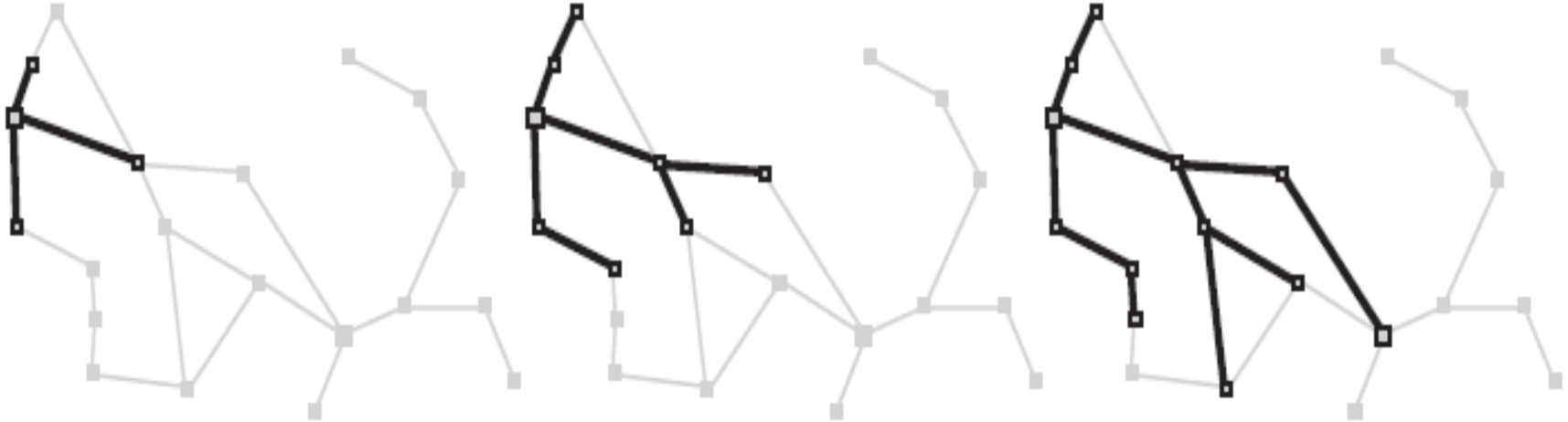


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

At the third stage, the northernmost city (Oradea) has become a dead end: 2 children already explored.

Rectangular-grid problem

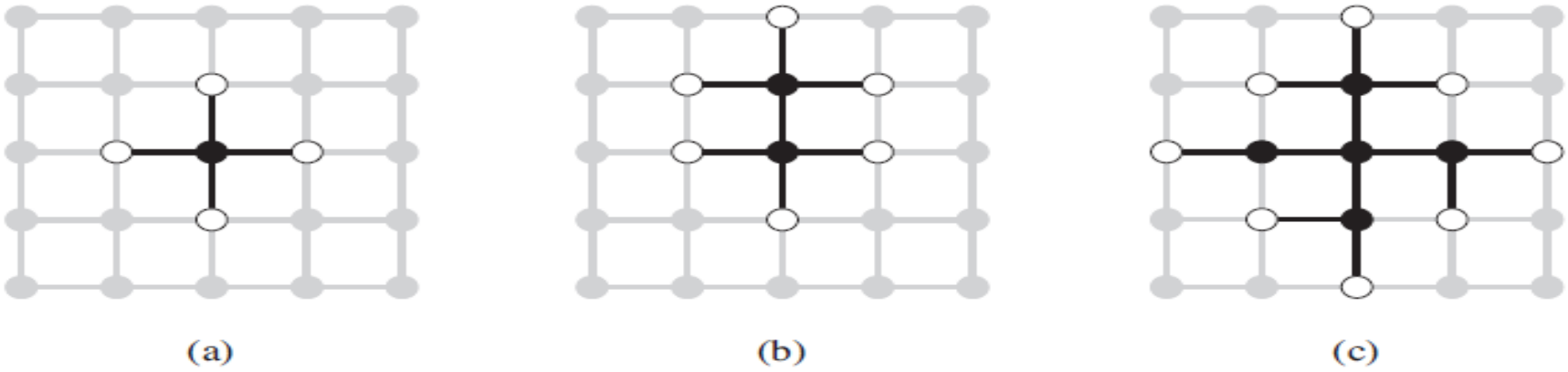


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

- A search tree of depth *d* that includes repeated states has $\mathbf{4^d}$ leaves; but there are only about $\mathbf{2*d^2}$ distinct states within d steps of any given state.
  ➔ If d = 20, about a trillion nodes but only about 800 distinct states.

Infrastructure for search algorithms

- Search algorithms require **a data structure** to keep track of the search tree being constructed.
- For each node n of the tree, we have a DS 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.

Infrastructure for search algorithms

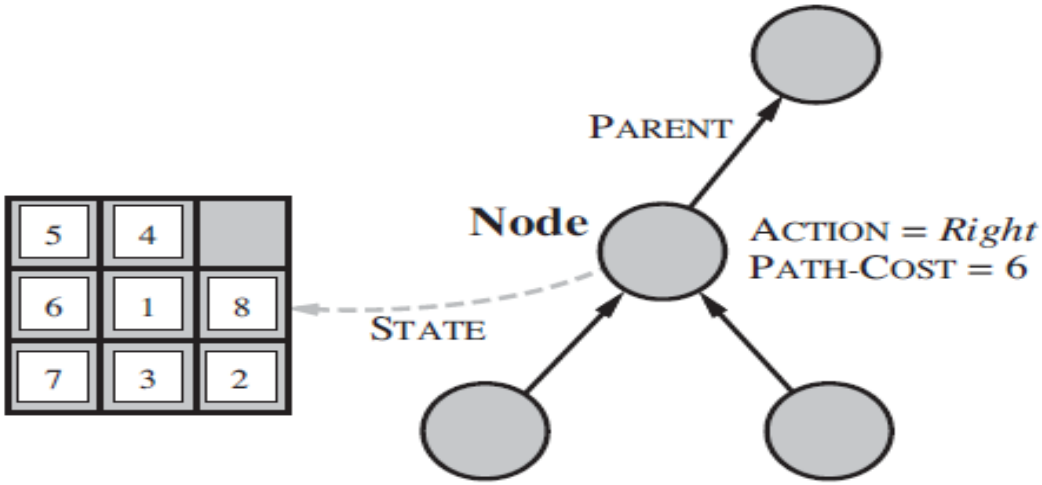


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node.

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Implementation issues

- The frontier is to be stored so that the search algorithm can easily choose the next node to expand according to its *strategy*.
- The appropriate DS here is a **Queue.**
- Operations on a queue: EMPTY?(queue), POP(queue), INSERT(element, queue).
- 3 common variants: **FIFO**, **LIFO** (i.e. **stack**), **Priority Queue**.
- *Explored set* can be implemented with a **hash table** so efficient checking for repeated states ➔ O(1) *insertion* & *lookup*.
  - Implement hash table with equality between states: the hash table needs to know that the set of visited cities {Bucharest, Urziceni, Vaslui} is the same as {Urziceni, Vaslui, Bucharest}.
- Ensure that the DS for states be in some canonical form; i.e., logically equivalent states should map to the same data structure. (e.g. bit-vector representation or a sorted list without repetition for states represented as sets are canonical)

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 (remember the *utility*)?
  - **Time complexity**: How long does it take to find a solution?
  - **Space complexity**: How much memory is needed to perform the search?

Measuring problem-solving performance

- Search complexity is expressed in terms of three quantities:
  - b, the **branching factor** (maximum number of successors of any node);
  - d, the **depth** of the shallowest goal node (the number of steps along the path from the root); and
  - m (the maximum length of any path in the state space).
- *Time complexity*: often measured in terms of the number of nodes generated during the search.
- *Space complexity*: measured in terms of the maximum number of nodes stored in memory.

Measuring problem-solving performance

- **Search cost**: typically depends on the time complexity, i.e. number of nodes generated during the search.
- **Path cost**: cost of the solution found.
- **Total cost:** combines search and path costs.
  - Arad to Bucharest: the search cost is the amount of time taken by the search;
  - the solution Path cost is the total length of the path in kms
  - Total cost: some combination of the two (e.g. estimating time for each km)

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Uninformed Search Strategies

- Also called **blind search strategies.**
- **Uninformed 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.

Breadth-First Search

- Simple strategy: the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, ...

  ➔ nodes at a given depth in the search tree are expanded before any nodes at the next level are expanded.
- It is an instance of the general graph-search algorithm where *shallowest* unexpanded node is chosen for expansion.
- Achieved by using a FIFO queue for the frontier.
- BFS always has the shallowest path to every node on the frontier.
- When all step costs are equal, BFS is optimal because it always expands the *shallowest* unexpanded node.

Breadth-First Search

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

Figure 3.11 Breadth-first search on a graph.

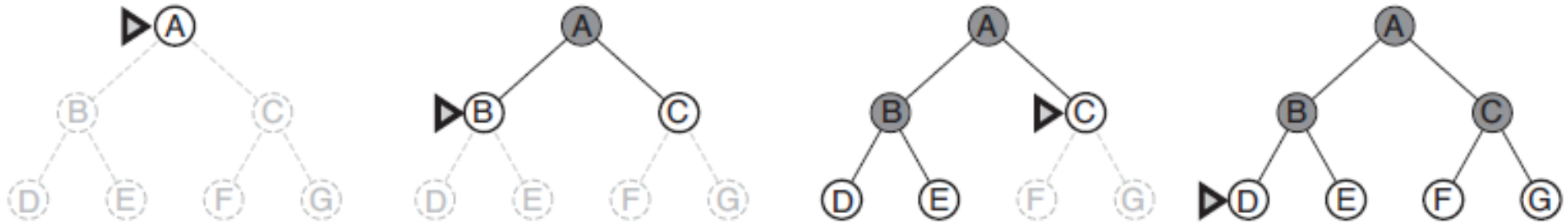


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-First Search complexities

- BFS is complete: If the shallowest goal node is at some finite depth *d*, BFS will find it after generating all shallower nodes (given: branching factor b is finite).
- *Shallowest* goal node not necessarily the *optimal one.*
- BFS is optimal if the path cost is a non-decreasing function of the depth of the node.
- **Worst-case time complexity**: (goal is last node generated at that level) ➔ total number of nodes generated is **???**

  $b + b^2 + b^3 + \cdot \cdot \cdot + b^d = \mathbf{O(b^d)}$
- **Space Complexity**: $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier SO **$O(b^d)$** .

Breadth-First Search complexities

- Suppose b = 10 and assume that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-Cost Search

- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost g(n)*.
- The frontier is stored as a priority queue ordered by *g*.
- *UCS* is ***optimal*** *with any step cost function.*

Uniform-Cost Search

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

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Uniform-Cost Search : example

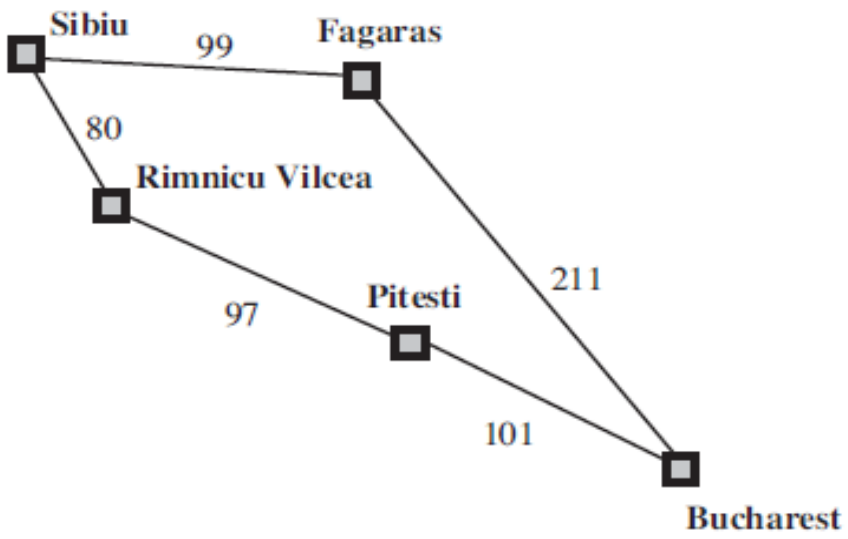


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Sibiu ➔ {Rimnicu Vilcea (80), Fagaras(99)} ➔ Rimnicu Vilcea ➔ {Pitesti (80+97)} ➔ Fagaras(99) ➔ {Bucharest(99+211)} goal reached but UCS expands Pitesti (80+97) ➔ Bucharest(80+97+101} so first path to goal discarded.

Uniform-Cost Search

- *UCS expands nodes in the order of their optimal path cost.*

➔ The first goal node selected for expansion must be the optimal solution.

- *UCS* does not care about the *number* of steps a path has, but only about their total cost.
- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε.

UCS time&space complexities

- Complexity is not easily characterized in terms of b and d.
- Let C^* be the cost of the optimal solution and suppose every action costs at least ϵ, then

  The worst-case time and space complexities are:

$$O(b^{1+\left\lfloor \frac{C^*}{\epsilon} \right\rfloor})$$

- This can be much greater than b^d.
- When all step costs are equal, this is b^{d+1} and UCS is same as BFS but may do unnecessary expansions

Depth-First Search

- **DFS** always expands the *deepest* node in the current frontier of the search tree.

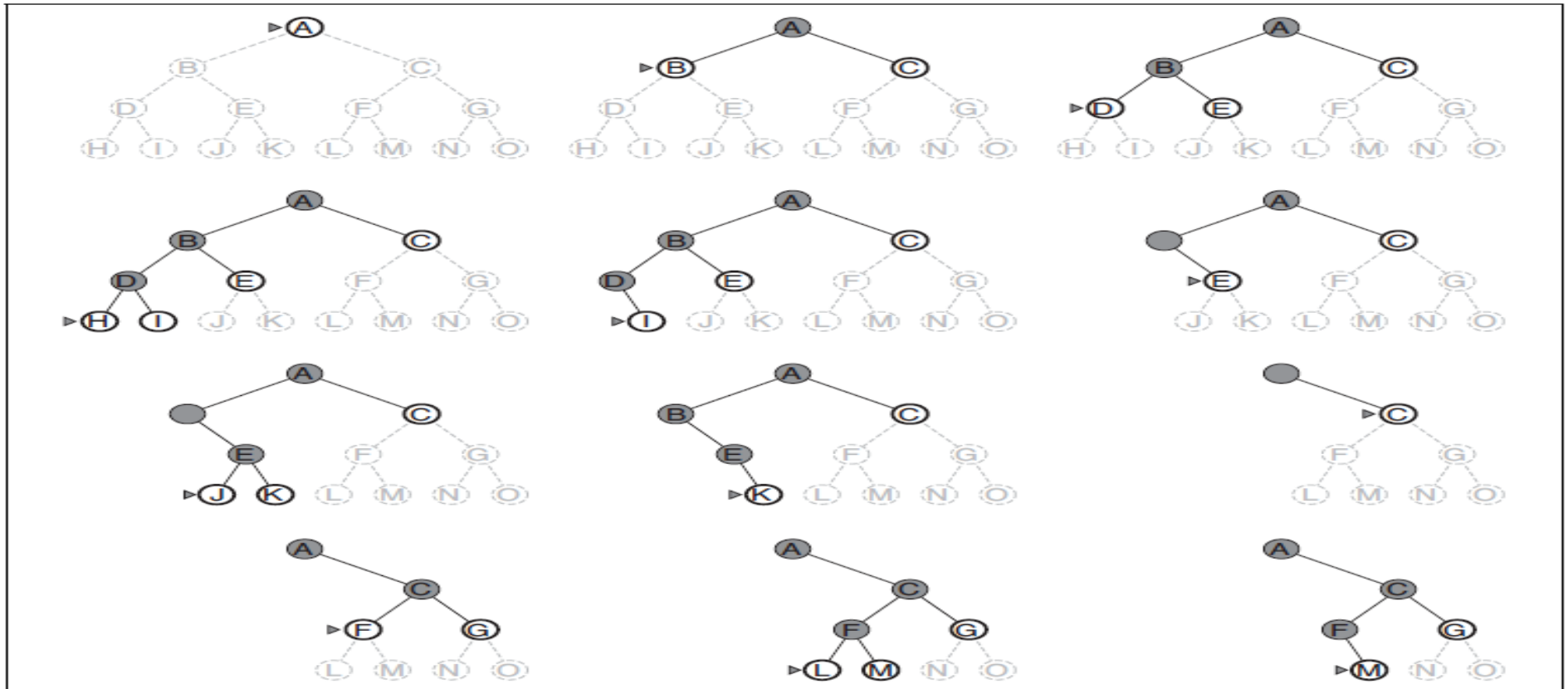


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth-First Search

- As the deepest nodes are expanded, they are dropped from the frontier ➔ the search "backs up" to the next deepest node that still has unexplored successors.
- DFS is an instance of the graph-search algorithm where a LIFO queue is used.
- Alternative to the GRAPH-SEARCH-style implementation, DFS can be implemented with a recursive function that calls itself on each of its children in turn.
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
- The tree-search version, is *not* complete; possibility of loops. E.g. Arad–Sibiu–Arad–Sibiu… (Slide 31)

Properties of DFS – Time

- Both versions of DFS are non-optimal. Example:
  - In Fig. 3.16, Suppose C is a goal node,
    - DFS will explore the entire left subtree first.
    - If node J were also a goal node, then DFS would return it as a solution instead of C.

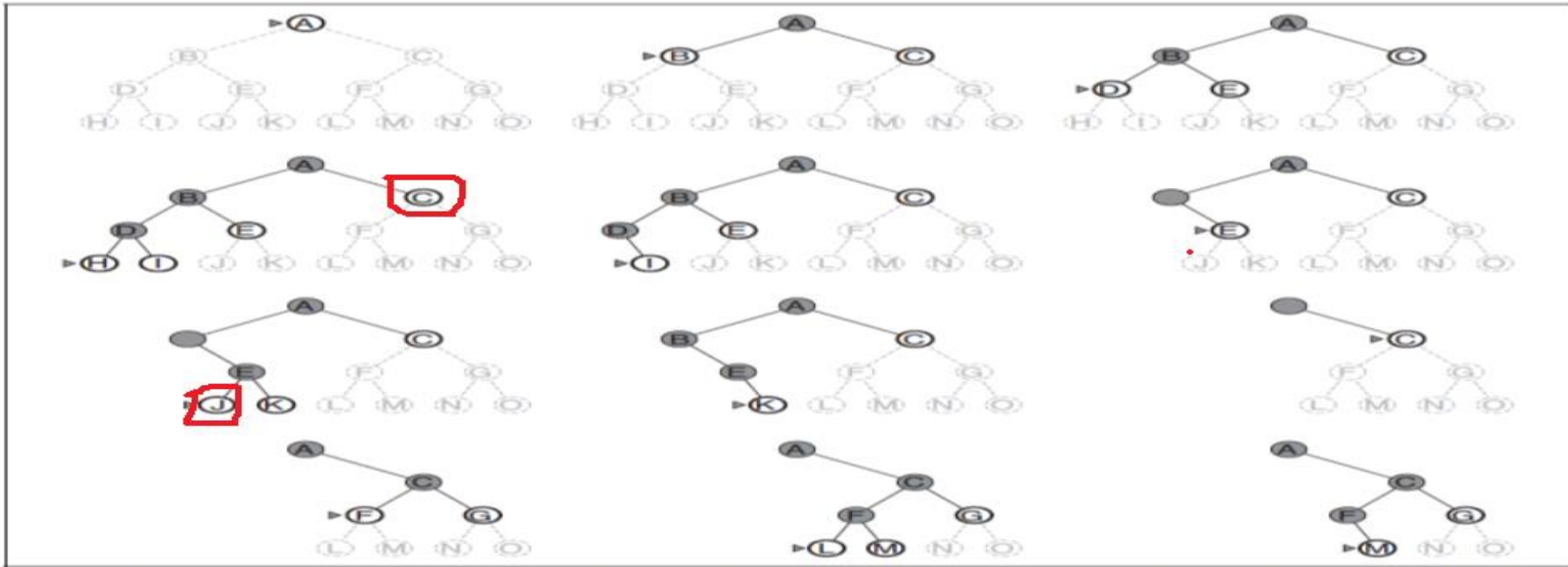


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Properties of DFS – Time

- Time complexity:
  - DF graph search is bounded by the size of the state space (which may be infinite).
  - A DF tree search, may generate all of the $O(b^m)$ nodes in the search tree (m: max depth of any node);
    - this can be much larger than the size of the state space. (m itself can be much larger than d, the depth of the shallowest solution) and is infinite if the tree is unbounded.

Properties of DFS - Space

- Space complexity of DFS much better than BFS.
  - DF tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored (See next slide).
- Space complexity: O(bm) nodes.
- (Figure 3.13, slide 52): assuming that nodes at the same depth as the goal node have no successors, DFS would require 156 kilobytes instead of 10 exabytes at depth d = 16 (i.e. 7 trillion times less space).
- Due to its space complexity, DFS is very frequently used in many AI areas.

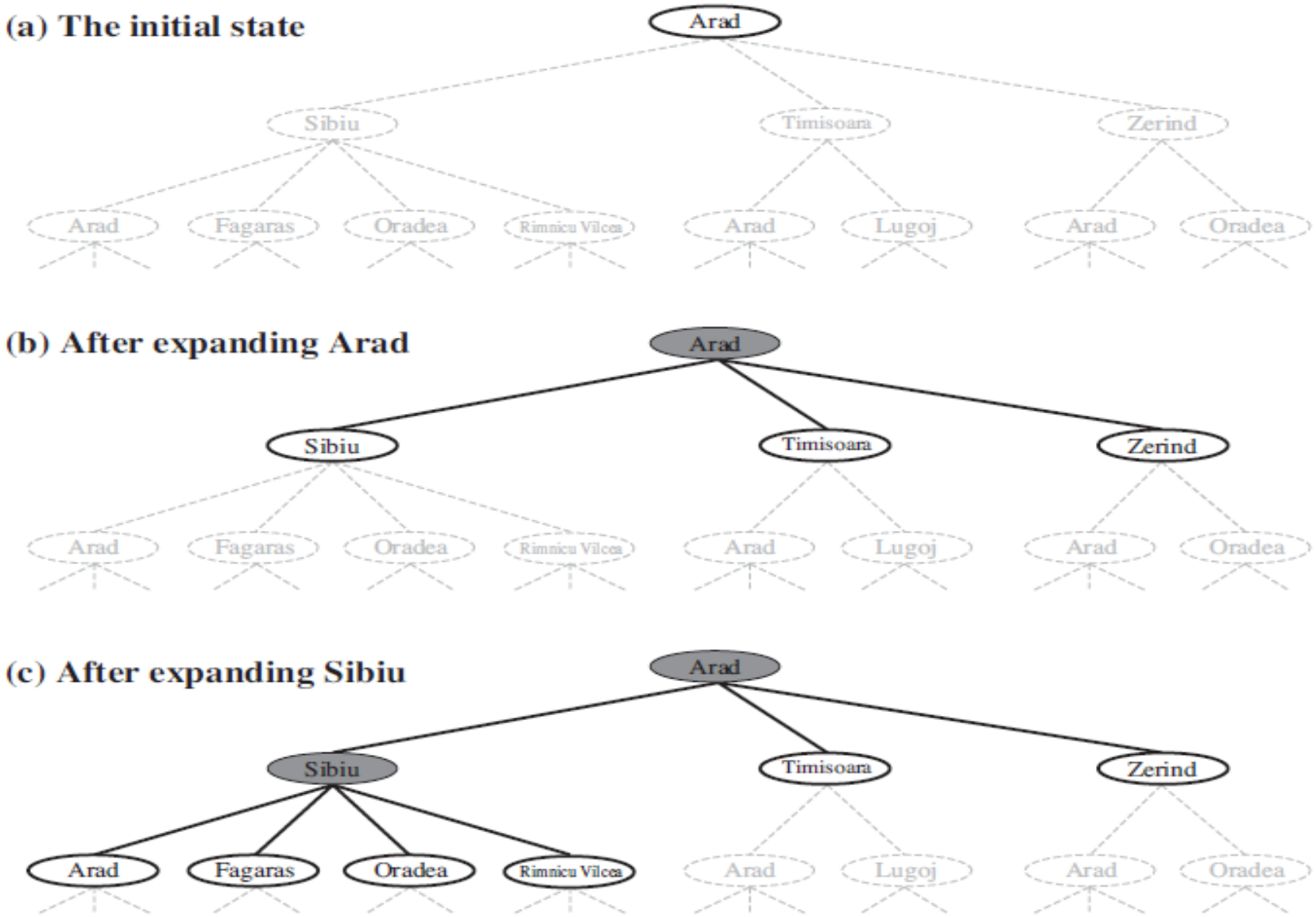


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Depth-Limited Search

- Problem with DFS: risk of infinite loops.
- DLS is DFS with a pre-determined depth limit l for any search path ➔ nodes at depth l are treated as if they have no successors.
- Problem: DLS introduces an additional source of incompleteness if we choose l<d, (the shallowest goal is beyond the depth limit).
- DLS is also non-optimal even if we choose l>d.
- Time complexity $O(b^l)$; space complexity $O(b\,l)$.
- DFS can be viewed as a special case of DLS with $l = \infty$.

Depth-Limited Search

- Sometimes, depth limits can be based on knowledge of the problem. E.g. in map of Romania there are 20 cities ➔any solution is of length at most 19, so $l = 19$.
- Actually, any city reachable from any other in at most 9 steps ➔ $l = 9$: known as the **diameter** of the state space

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

Iterative-Deepening Search

- Iterative Deepening (DFS), often used in combination with depth-first tree search, finds the best depth limit.
- Does DLS but gradually increasing the limit—first 0, then 1, then 2, etc.—until a goal is found.
- This will occur when the depth limit l reaches d, the depth of the shallowest goal node.
- IDS combines the benefits of DFS and BFS.
  - Like DFS, modest memory requirements: *O(bd)*
  - Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a non-decreasing function of the node depth.

Iterative-Deepening 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.

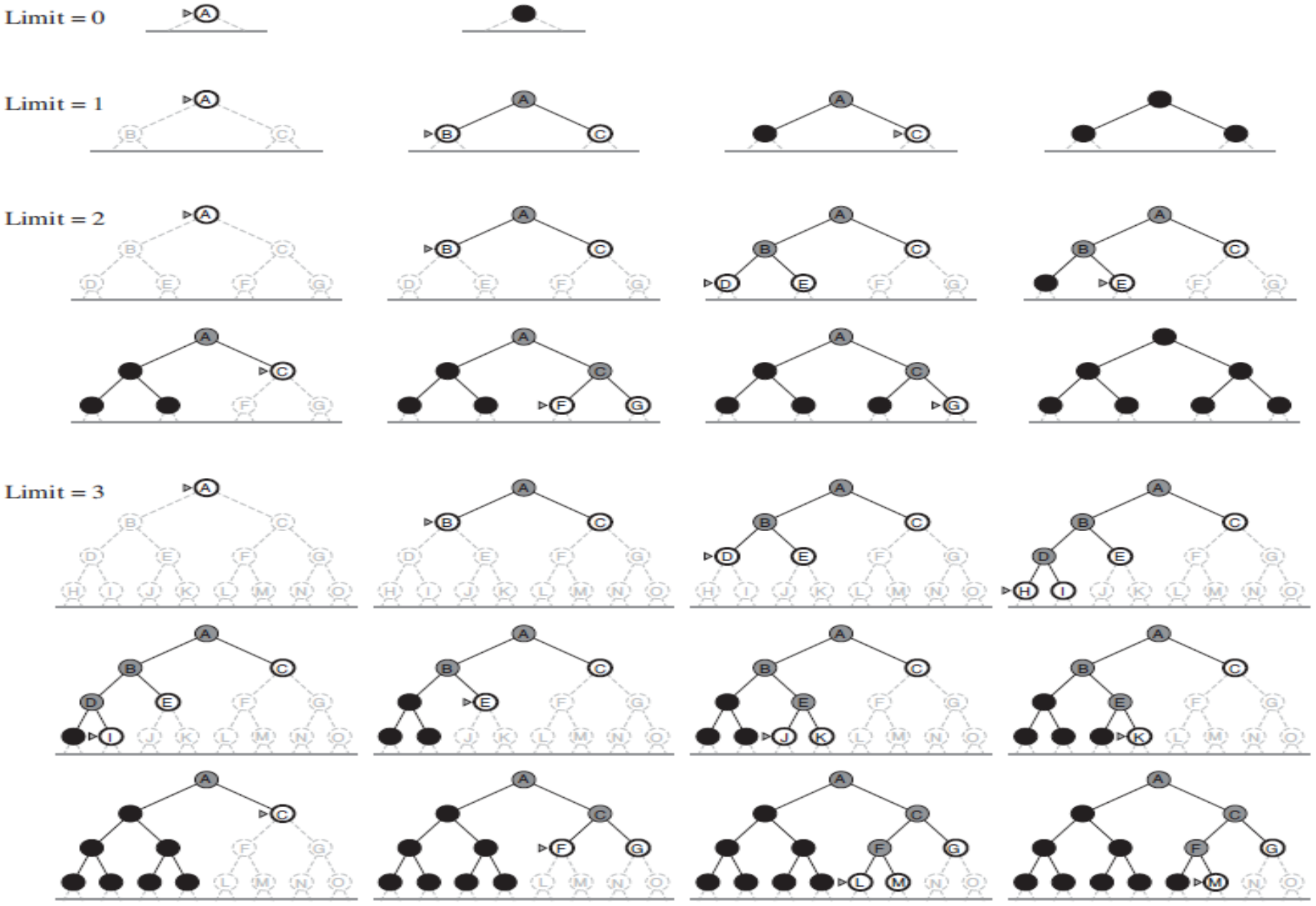


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Iterative-Deepening Search

- In IDS, the nodes on the bottom level (depth *d*) are generated once, those on the next-to-bottom level are generated twice, etc., up to the children of the root, which are generated *d* times.

➔So Nbr of generated nodes:

N(IDS) = $(d)b + (d-1)b^2 + (d-2)b^3 + \ldots + (1)\,b^d$

➔ worst-case time complexity: $O(b^d)$

- *In general, IDS is the preferred uninformed search method when the search space is large and the depth of the solution is not known.*

Bidirectional search

- **Idea**: 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.
- **Motivation**: $O(b^{d/2}) + O(b^{d/2})$ is much less than $O(b^d)$.
- **Implemented** by replacing the goal test with a check to see if the frontiers of the two searches intersect➔ if so, **solution found**!

Bidirectional search

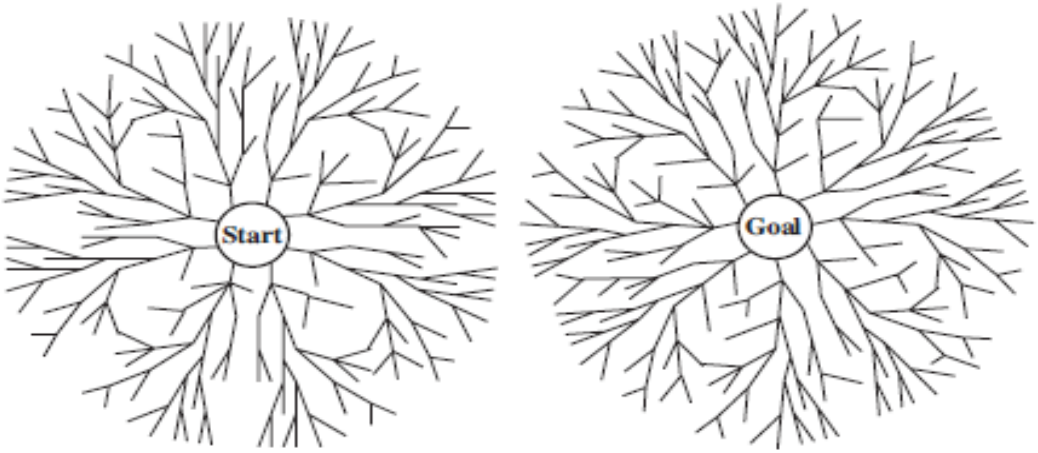


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

- **Question**: how do we search backward?
- Let the **predecessors** of a state x be all those states that have x as a successor.

➔ Computing predecessors??

- When all actions in the state space are reversible, the predecessors of x are just its successors. (e.g. neighbouring city).
- Other cases may require substantial ingenuity.

Bidirectional search

- What does *goal* mean in searching "*backward from the goal*"?
- For *8-puzzle* and for *finding a route in Romania*, there is just one goal state, so backward search is like forward search. ➔ no problem.
- If several *explicitly listed* goal states, e.g. vacuum cleaner pb.—then we can construct a new dummy goal state whose immediate predecessors are all the actual goal states.
- if the goal is an abstract description, e.g. "no queen attacks any other", then bidirectional search is difficult to use.

Comparing uninformed search strategies

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

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Outline (cont.)

- Informed (Heuristic) Search Strategies
  - Greedy best-first search
  - A* search
  - Memory-bounded heuristic search
  - Learning to search better
- Heuristic Functions
  - The effect of heuristic accuracy on performance
  - Generating admissible heuristics from relaxed problems
  - Learning heuristics from experience

Informed (Heuristic) Search Strategies

- **Informed search** strategy: one that uses problem-specific knowledge beyond the definition of the problem itself.
- Informed search strategies can find solutions more efficiently than can an uninformed strategy.
- **Best-first search** is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, *f(n)*, taken as a **cost estimate**, so the node with the *lowest* evaluation is expanded first.
- Best First Search is implemented as UCS except for *f* instead of *g* for ordering the priority Queue.

Informed (Heuristic) Search Strategies

- The choice of *f* determines the search strategy.
- Most best-first algorithms include as a component of *f* a **heuristic function**, denoted *h(n)*, where
  - h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.
  - E.g. Straight-line distance could be an estimate of the distance between *Arad* and *Bucharest*.
- Heuristic functions are considered to be arbitrary, non-negative, problem-specific functions, with one constraint: if *n* is a goal node, then *h(n)=0*.
- Heuristic functions are selected with a deep understanding of the problem.

Greedy best-first search

- **Greedy best-first search** tries to expand the node that is closest to the goal, by using just the heuristic function. that is, *f(n) = h(n)*.
- **Aim**: this is likely to lead to a solution quickly.
- E.g.: use the **straight-line distance** heuristic, denoted here h_{SLD}. SLD to 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.

Case: Travelling in Romania

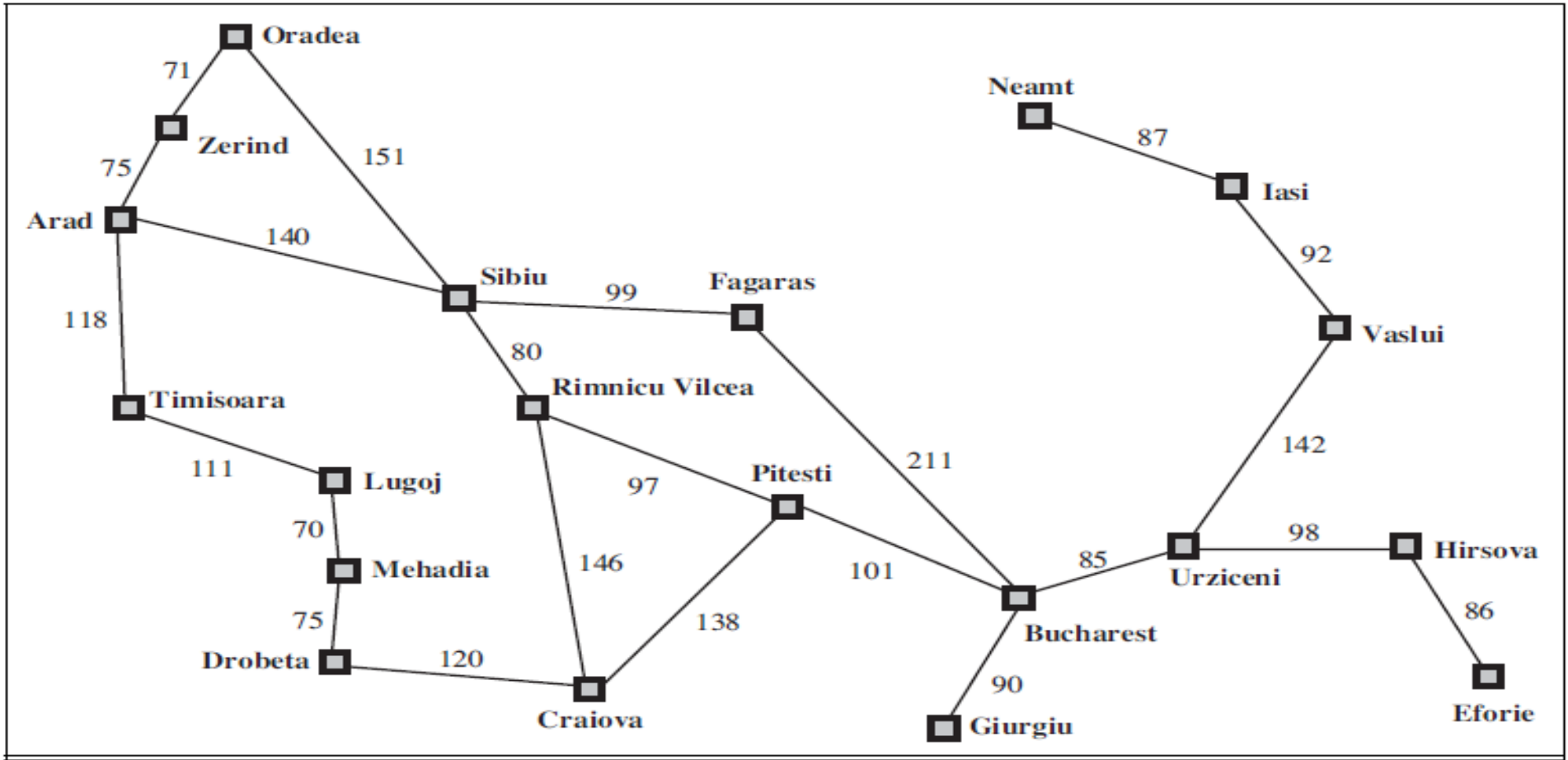


Figure 3.2 A simplified road map of part of Romania.

- Suppose Agent needs to go from Arad to Bucharest.
- Three possible alternative first steps.

Greedy BF search – Map of Romania

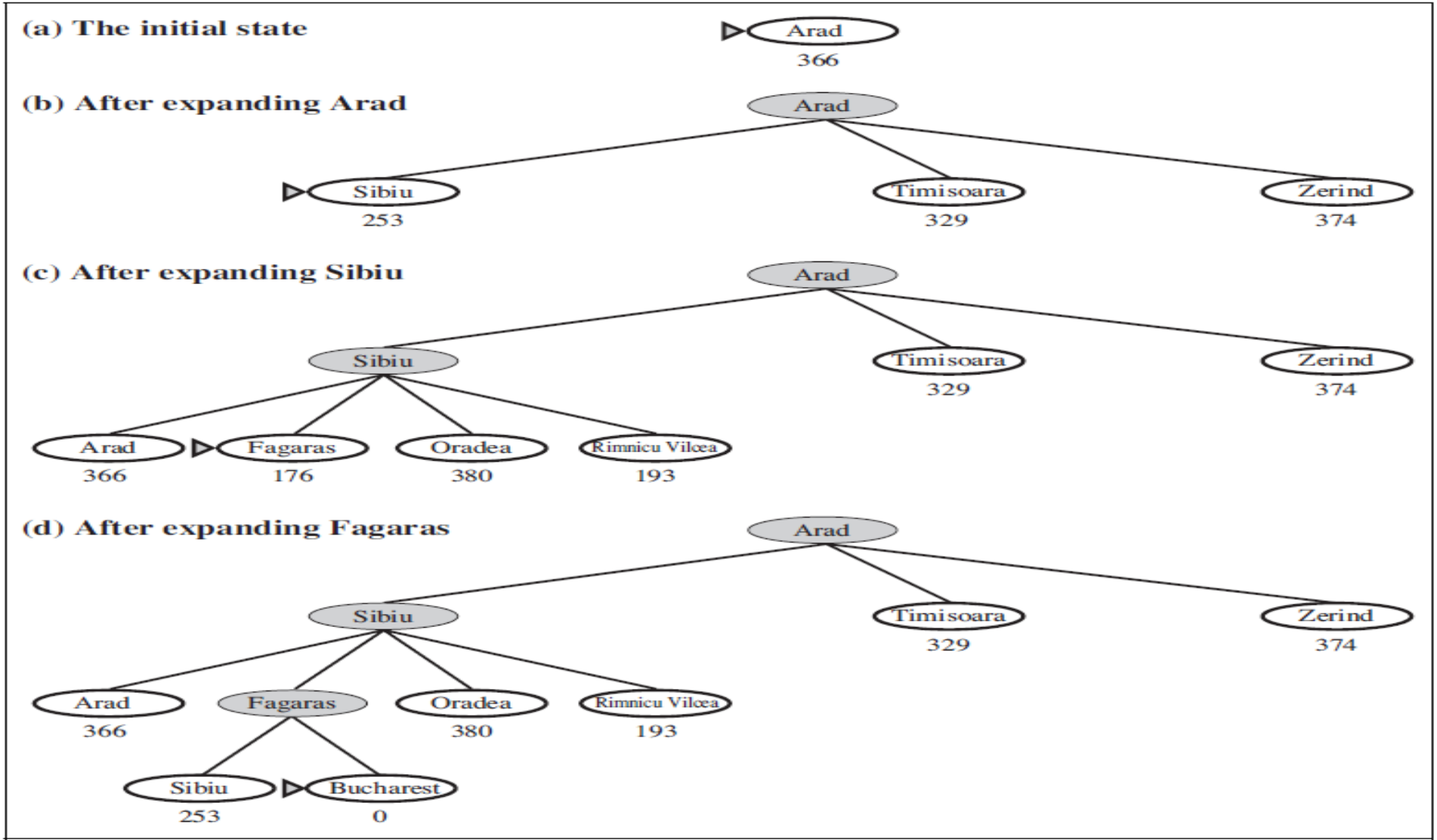


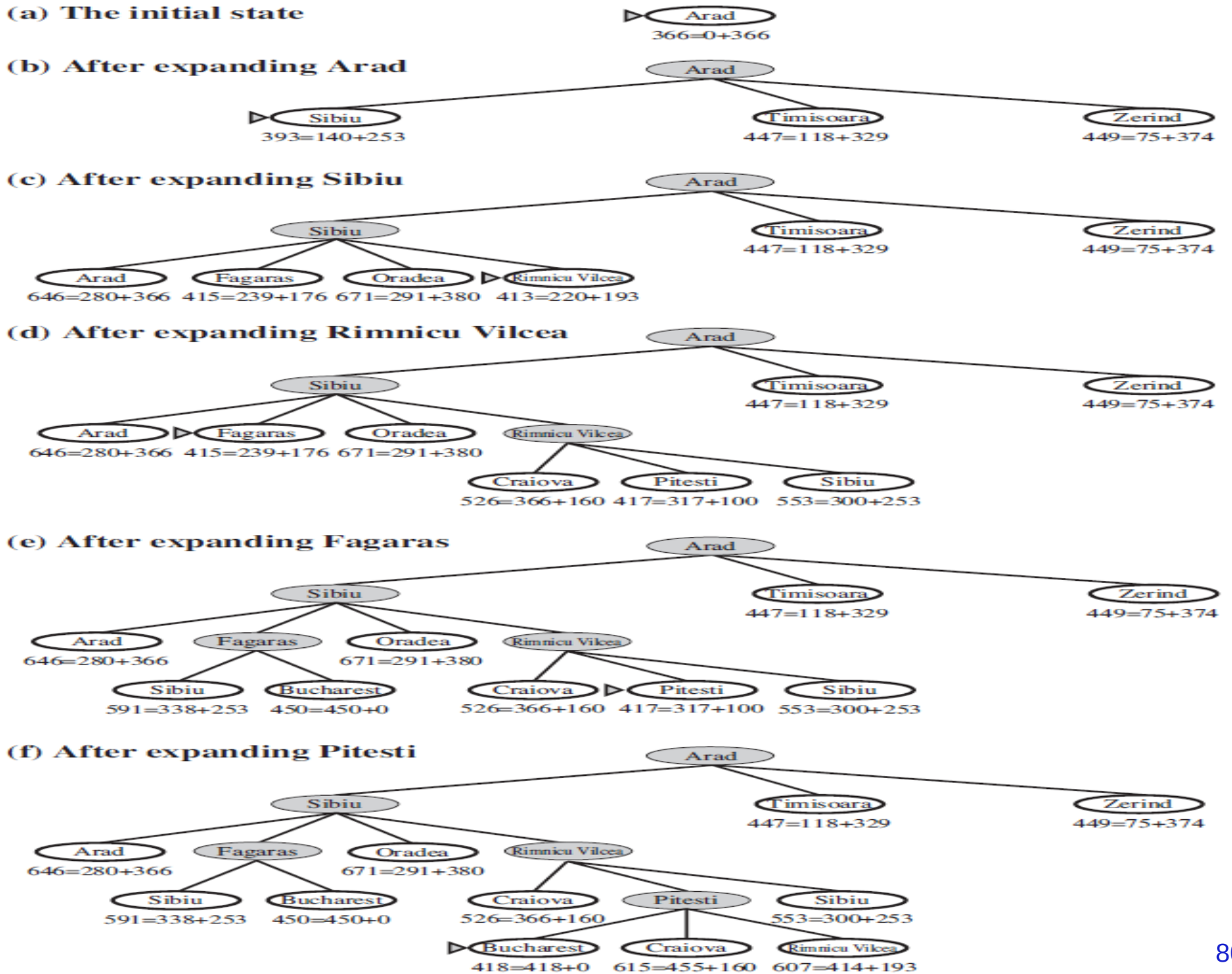
Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Greedy best-first search

- Greedy Best-First tree S. is **not optimal**.
  - E.g. 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 **incomplete** even in a finite state space, just like DFS.
  - E.g. travel from *Iasi* to *Fagaras*:
    - The heuristic suggests that *Neamt* be expanded first because it is closest to *Fagaras*.
    - Expanding *Neamt* takes back to *Iasi*. ➔ infinite loop!
- Worst-case time and space complexity for the tree version is **$O(b^m)$** where m is the max depth of the search space.
- A good heuristic can reduce the complexity substantially.

A* search

- It evaluates nodes by combining *g(n)*, the cost to reach the node, and *h(n)*, the estimated cost to get from the node to the goal:
  - $f(n) = g(n) + h(n)$.
  - $f(n)$ = estimated cost of the cheapest solution **through *n*** .
- The algorithm is identical to Uniform-Cost-Search except that A∗ uses $g + h$ instead of g.
- Under some conditions on the heuristic function *h(n)*, A∗ search is both complete and optimal.


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

Conditions for optimality

- Two conditions for optimality of A* Search:
  - **Admissibility**: a heuristic function is **admissible** if it *never overestimates* the cost to reach the goal.
    ➔ ***f(n)*** never overestimates the true cost of a solution through *n.*
    - *E.g. Straight-Line Distance is admissible.*
  - **Consistency** (sometimes called **monotonicity**): A heuristic *h(n)* is consistent if, for every node n and every successor *n′* of *n* generated by any action *a* :

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

    - This condition is required only for applications of A∗ to graph search. (Think about it!)
- Every consistent heuristic is also admissible.
- Some admissible heuristics are not consistent. (Left as exercises)

Optimality of A* Search

- A^* has the following properties:
  - *The tree-search version of $A*$ is optimal if h is admissible.*
  - *The graph-search version is optimal if h is consistent.*
  - *Proof is straightforward (See textbook).*
- 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 where $f(n) = C^*$ before selecting a goal node.

Properties of A* Search

- **If** all step costs exceed some finite ε and if b is finite, then A* search is **complete**.
- Notice that A* expands no nodes with f(n) > C*.

  ➔ **pruning** of subtrees during the search.
  - E.g. Timisoara not expanded! Because h_{SLD} is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality. Pruning is important in various AI areas.
- A* is **optimally efficient** for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A*.

Properties of A* Search

- A* search is complete, optimal, and optimally efficient among all such algorithms.
- Unfortunately, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution.
- Main time complexity results are given for constant-step problems in terms of the **absolute error** or the **relative error** of the heuristic.
  - **Absolute error**: $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and
  - **Relative error** is defined as $\varepsilon \equiv (h^* - h)/h^*$

Properties of A* Search

- The complexity results for A* depend very strongly on the assumptions made about the state space.
- For 8-puzzle, the time complexity of A* is $O(b^{\Delta})$.
- For constant step costs, we can write this as $O(b^{\varepsilon d})$, where *d* is the solution depth, so the effective branching factor is b^{ε}.
- In the general case of a graph, the situation is even worse. There can be exponentially many states with $f(n) < C^*$ even if the absolute error is bounded by a constant.

Properties of A* Search

- 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
  - sometimes design heuristics that are more accurate but not strictly admissible.
- Still, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search
- 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.

Memory-bounded heuristic search

Iterative-deepening A∗ (IDA∗):

- Using the idea of iterative-deepening search to the heuristic search context.
- Cutoff is based on the cost function *f (g+h)* rather than the depth.
- It is the simplest way to reduce memory requirements for A∗.
- IDA∗ is practical for many problems with unit step costs.

Memory-bounded heuristic search

Recursive best-first search (RBFS):

- Mimics the operation of standard best-first search, but using only linear space.
- The algorithm structure is similar to that of a recursive DFS, 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.

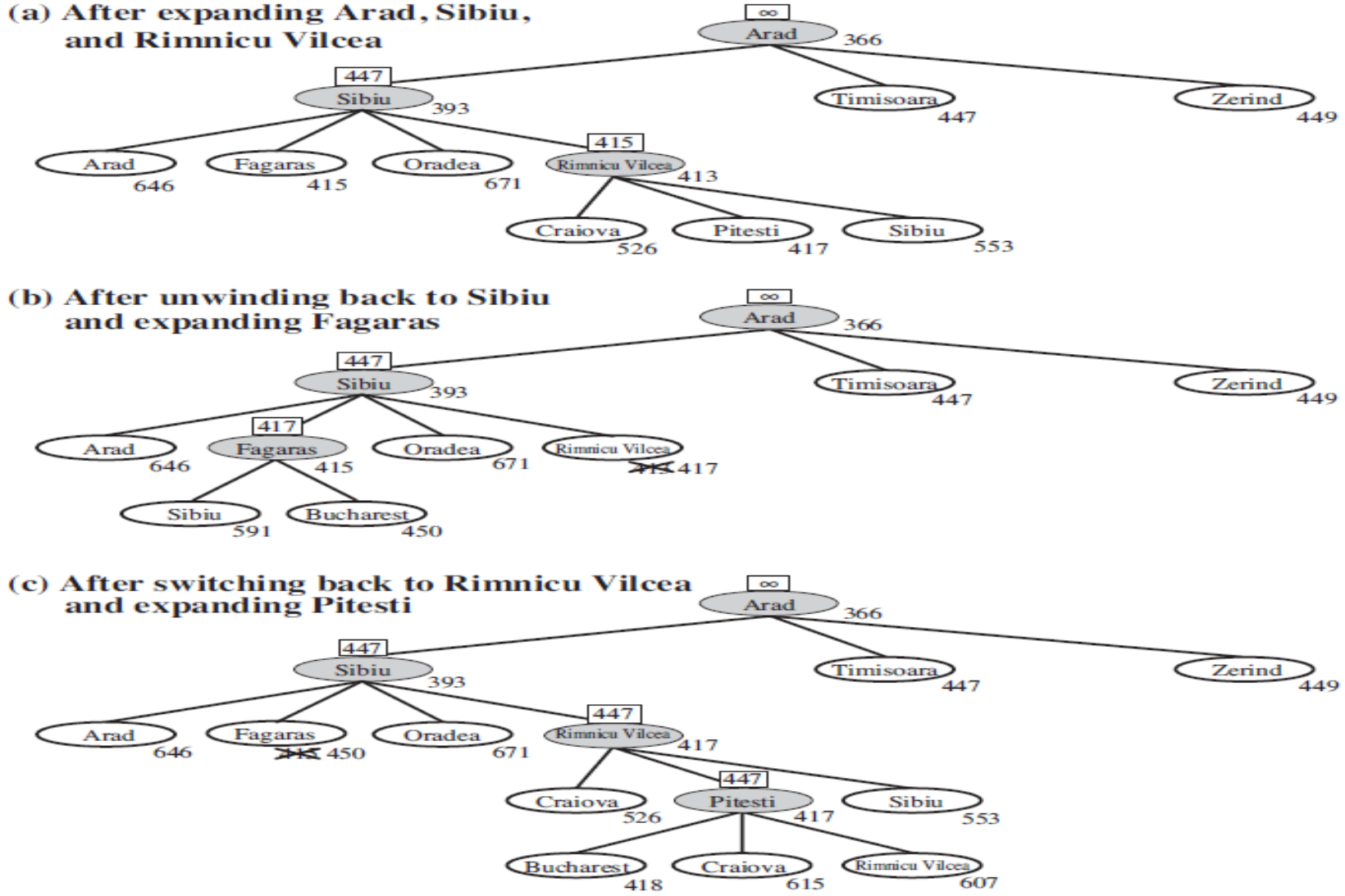


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (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. (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.

Properties of RBFS

- RBFS is somewhat more efficient than IDA∗, but still suffers from excessive node regeneration due to "mind changes".
- Like A∗, RBFS is an **optimal** algorithm if the heuristic function *h(n)* is **admissible**.
- RBFS space complexity is linear in the depth of the deepest optimal solution
- RBFS time complexity: (potentially exponential) but 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.

Learning to search better

Could an agent *learn* how to search better?

- The method rests on an important concept called the **metalevel state space**.
- Each state in a meta-level state space captures the internal (computational) state of a program that is searching in an **object-level state space** such as Romania.
- The internal state of the A∗ algorithm consists of the current search tree.
- Each action in the meta-level state space is a computation step that alters the internal state.
- For example, each computation step in A∗ expands a leaf node and adds its successors to the tree.

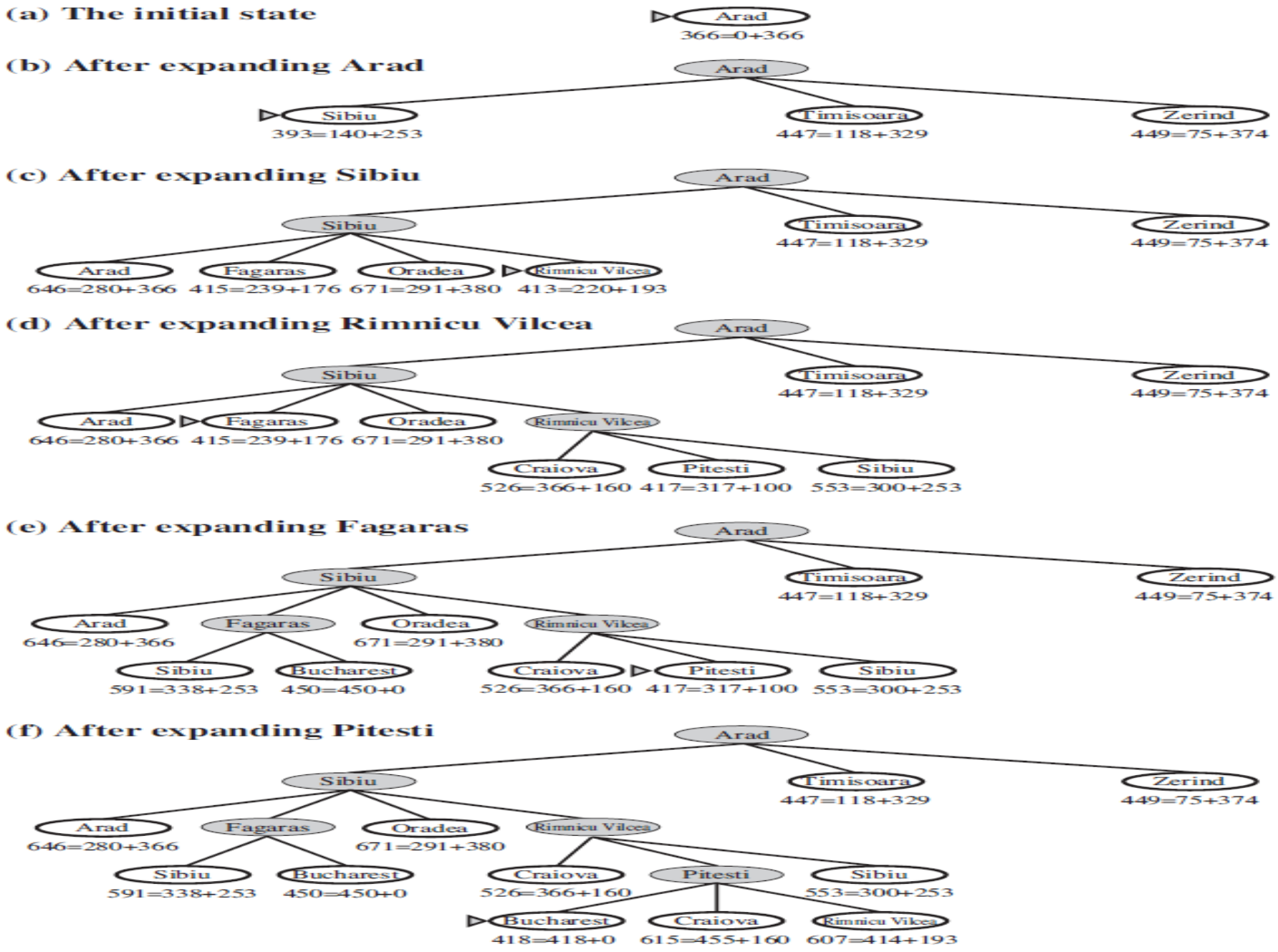


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. T
h values are the straight-line distances to Bucharest taken from Figure 3.22.

Learning to search better

- The previous figure of A* search shows a sequence of larger and larger search trees.
- It can be seen as depicting a path in the meta-level state space where each state on the path is an object-level search tree.
- A **metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees.
- The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost

Outline

- Problem-Solving Agents
  - Well-defined problems and solutions
  - Formulating problems
- Example Problems
- Searching for Solutions
- Infrastructure for search algorithms
  - Measuring problem-solving performance
- Uninformed Search Strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening depth-first search
  - Bidirectional search

Outline (cont.)

- Informed (Heuristic) Search Strategies
  - Greedy best-first search
  - A* search
  - Memory-bounded heuristic search
  - Learning to search better
- Heuristic Functions
  - The effect of heuristic accuracy on performance
  - Generating admissible heuristics from relaxed problems
  - Learning heuristics from experience

Heuristic Functions

- We consider heuristics for the 8-puzzle to better understand heuristics.
- The 8-puzzle was one of the earliest heuristic search problems.
- Branching factor is ≈ **3** and average solution cost for a randomly generated 8-puzzle instance is **about 22 steps**.

➔ An exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states.

- With graph search: reduction by a factor of ≈ 170,000 because only 9!/2 = 181, 440 distinct states are reachable.
- BUT, the corresponding number for the 15-puzzle is ≈ **10^{13}**

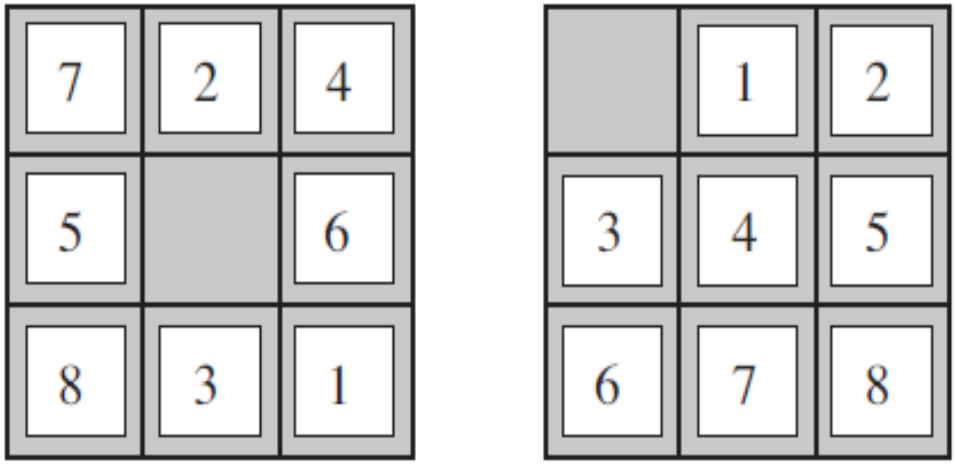


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

Heuristics for 15-Puzzle

- To use A*, we need to look for an admissible heuristic function. Two commonly used candidates for 15-puzzle:
  - h_1 = the number of misplaced tiles. E.g. for previous 8-puzzle example: h_1 =8 for initial state. Clearly, h1 is admissible because any tile that is out of place must be moved at least once.
  - h_2 = the sum of the distances of the tiles from their goal positions. (Since moves are only horizontal or vertical, the function is sometimes called the **city block distance** or **Manhattan distance**.)
    In 8-puzzle example, for the initial state,
    h_2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18
    (Convince yourself h_2 is admissible!)

Heuristic accuracy & 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,
- $N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$
  - E.g. if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.
- The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems.
- For a well-designed heuristic, $b^* \approx 1$, allowing fairly large problems to be solved at reasonable computational cost.

Generated 1200 random problems with solution lengths from 2 to 24 and solved them with IDS and with A* using both h_1 and h_2.

	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.

h_2 is better than h_1, and is far better than IDS. Even for small problems with d=12, A* with h_2 is 50,000 times more efficient than IDS

On Heuristics

- Is h_2 *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 **dominates** h_1.
- Domination translates directly into efficiency: A* using h_2 will never expand more nodes than A* using h_1
- BUT, can such heuristics, especially h_2, be constructed mechanically?

Best heuristic?

- What if one fails to get a single "clearly best" heuristic (which is often the case)?
- Suppose 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 one to select?
- Take $h(n) = max\{h_1(n), \ldots, h_m(n)\}$
- Since the component heuristics are **admissible**, so is h;
- It is also easy to prove that h is **consistent**.

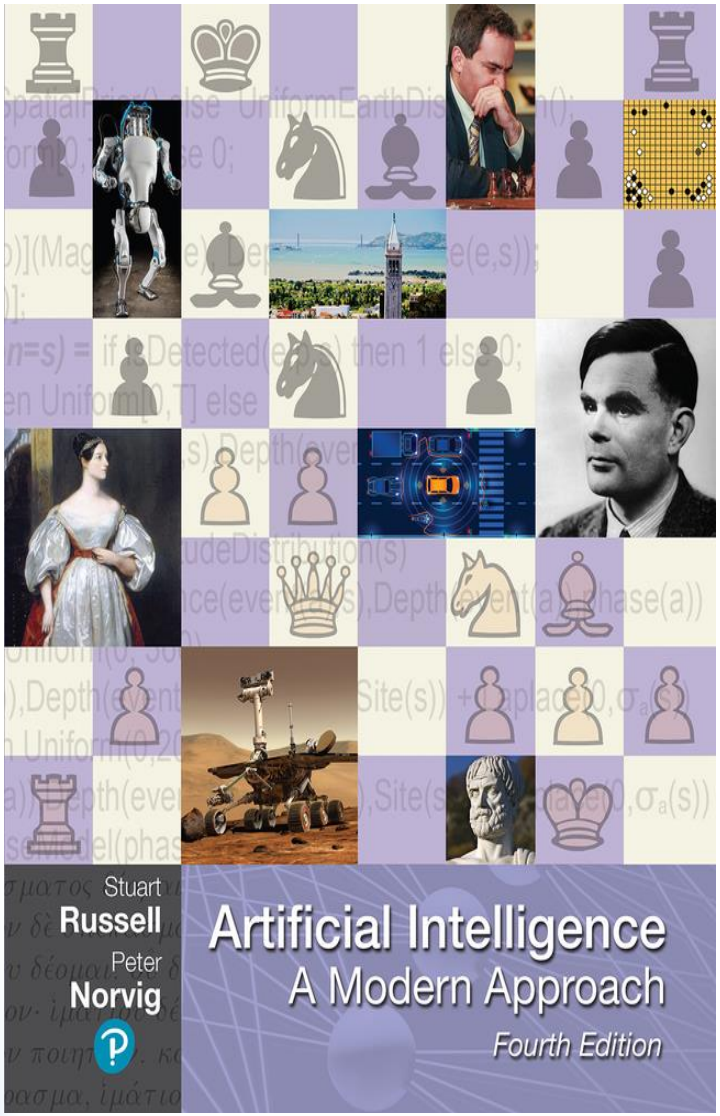
Learning heuristics from experience

- A heuristic function *h(n)* can be learned from experience (i.e. examples).
- Example learning a heuristic for 8-puzzle:
  - solving lots of 8-puzzles, for instance.
  - Each optimal solution to an 8-puzzle problem provides examples from which *h(n)* can be learned.
  - Each example consists of a state from the solution path and the actual cost of the solution from that point.
  - A learning algorithm can use these examples to learn a model of a function *h(n)* that can predict solution costs for other states that arise during search.

Learning heuristics from experience

- Learning can be achieved using Neural Networks, Decision trees, reinforcement learning, etc., or even Inductive Learning.
- **Inductive Learning** works best when supplied with **features** of a state that are relevant to predicting the state's value. E.g.:
  - feature $x_1(n)$: "number of misplaced tiles" might be helpful in predicting the actual distance of a state from the goal.
  - feature $x_2(n)$: "number of pairs of adjacent tiles that are not adjacent in the goal state."
  - etc.
  - Find averages for $x_1(n)$ and $x_2(n)$ on a statistically significant number of random configurations and their actual solution costs.
- Take $h(n) = c_1x_1(n) + c_2x_2(n)$ *and find* c_1 *and* c_2 that give the best fit to the actual data on solution costs.

Slides based on the textbook



- Russel, S. and Norvig, P. (2020) Artificial Intelligence, A Modern Approach (4th Edition), Pearson Education Limited.