Chapter 3: Solving Problems By Searching

CS-4365 Artificial Intelligence

Chris Irwin Davis, Ph.D.

Email: chrisirwindavis@utdallas.edu
Office: ECSS 4.603

Outline

- Problem Solving Agents
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
- Informed (Heuristic) Search Strategy
- Heuristic Functions

Solving Problems By Searching

- We begin by describing one kind of goal-based agent called a **problem-solving agent**
- Problem-solving agents think about the world using *atomic representations* – that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms
- Goal-based agents that use more advanced factored or structured representations are usually called **planning agents** (discussed in Chapter 7 and 11)

Problems and Solutions

- We start our discussion of problem solving by defining precisely the elements that constitute a “problem” and its “solution,” and give several examples to illustrate these definitions.
- We then describe several general-purpose *search algorithms* that can be used to solve these problems.

Search

- This chapter introduces methods that an agent can use to select *actions* in *environments* that are
  - **deterministic**,
  - **observable**,
  - **static**, and
  - **completely known**.
- In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.

Uniformed Search vs. Informed Search

- We will see several **uninformed search** algorithms—algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently.
- **Informed search** algorithms, on the other hand, can often do quite well given some idea of where to look for solutions.

Static Environment: Fixed Sequence of Actions

- For the time, we limit ourselves to the simplest kind of task environment, for which the solution to a problem is always a *fixed sequence of actions*
- The more general case—where the agent’s *future actions* may vary depending on *future percepts* – is handled in Chapter 4

3.1 – Problem Solving Agents

Problem Definition

- Before an agent can start searching for solutions, a **goal** must be identified and a well-defined **problem** must be formulated.

Performance Measure

- Imagine an *agent* in the city of Arad, Romania, enjoying a touring holiday. The agent's **performance measure** contains many factors:
  - improve its suntan,
  - improve its Romanian,
  - take in the sights,
  - enjoy the nightlife (such as it is),
  - avoid hangovers, and so on.
- The decision problem is a complex one involving many tradeoffs and careful reading of guide-books.

Goal

- Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the **goal** of getting to Bucharest.
- Courses of action that don't reach Bucharest on time can be rejected without further consideration and the agent's decision problem is greatly simplified.
- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

Choosing Appropriate Actions and States

- We will consider a goal to be a *set of world states* – exactly those states in which the goal is satisfied.
- The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.
- Before it can do this, it needs to decide what sorts of actions and states it should consider. If it were to consider actions at the level of "move the left foot forward an inch" or "turn the steering wheel one degree left," the agent would probably never find its way out of the parking lot, let alone to Bucharest, because at that level of detail there is too much uncertainty in the world and there would be too many steps in a solution.

Problem Formation

- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- For now, let us assume that the agent will consider actions at the level of driving from one major town to another.
- Each state therefore corresponds to being in a particular town.

- Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad.
- There are three roads out of Arad,
  - one toward **Sibiu**,
  - one to **Timisoara**, and
  - one to **Zerind**.
- None of these achieves the goal, so unless the agent is very familiar with the geography of Romania, it will not know which road to follow.

Map of Romania

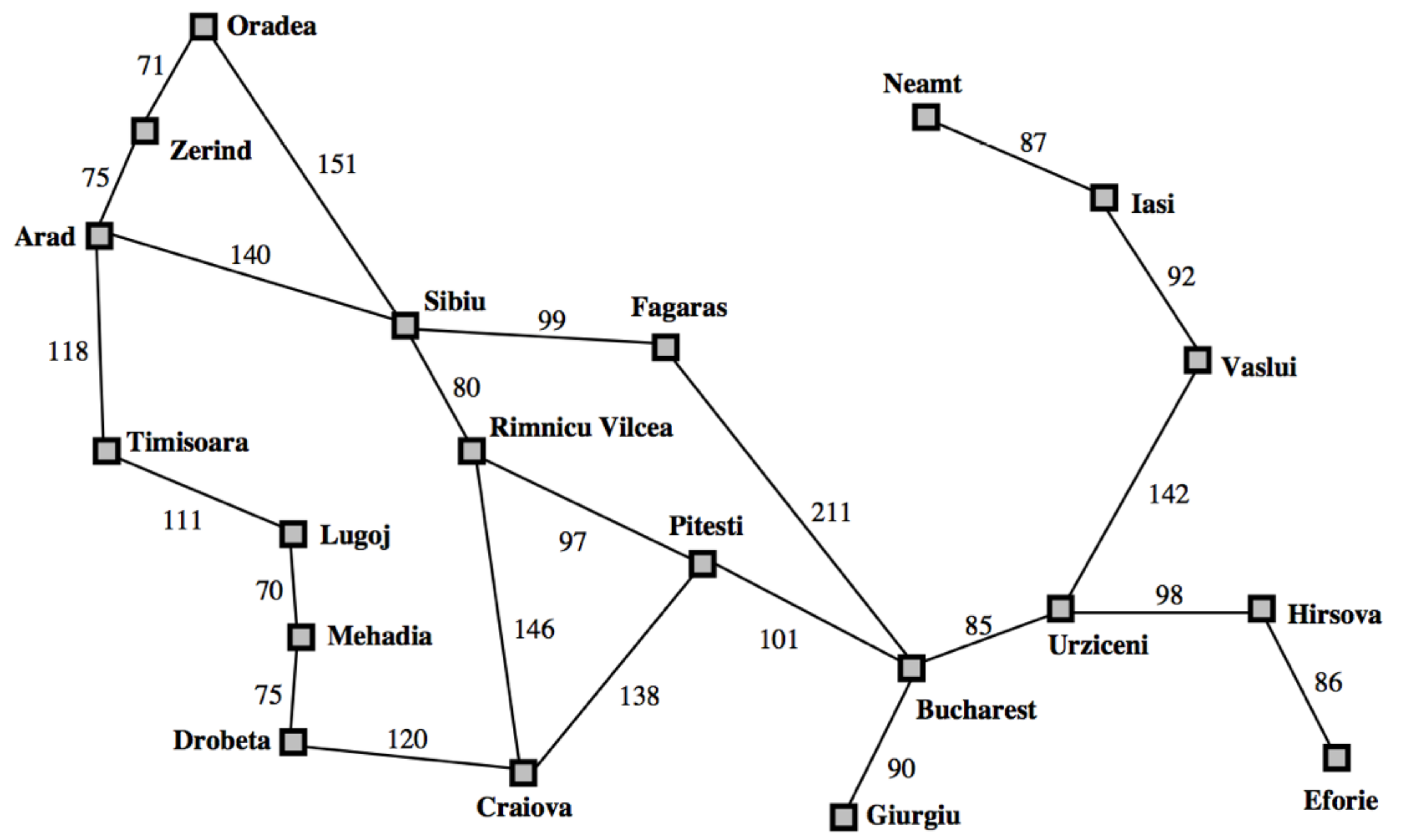


Figure 3.2 A simplified road map of part of Romania.

Simple Problem Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: 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)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Search (revisit)

- An agent can select actions in environments that are
    - deterministic,
    - observable,
    - static, and
    - completely known.
- In such cases, the agent can construct sequences of actions that achieve its goals; this process is called **search**.

Search

- Under these assumptions, the solution to any problem is a fixed sequence of actions.
- In general it could be a branching strategy that recommends different actions in the future depending on what percepts arrive.
- 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.

Well-defined Problems and Solutions

- A problem consists of five parts:
  - the **initial state**,
  - a **set of actions**,
  - a **transition model** describing the results of those actions,
  - a **goal test function**, and
  - a **path cost function**.
- The environment of the problem is represented by a **state space**.
- A **path** through the state space from the initial state to a goal state is a **solution**.

Formulating Problems

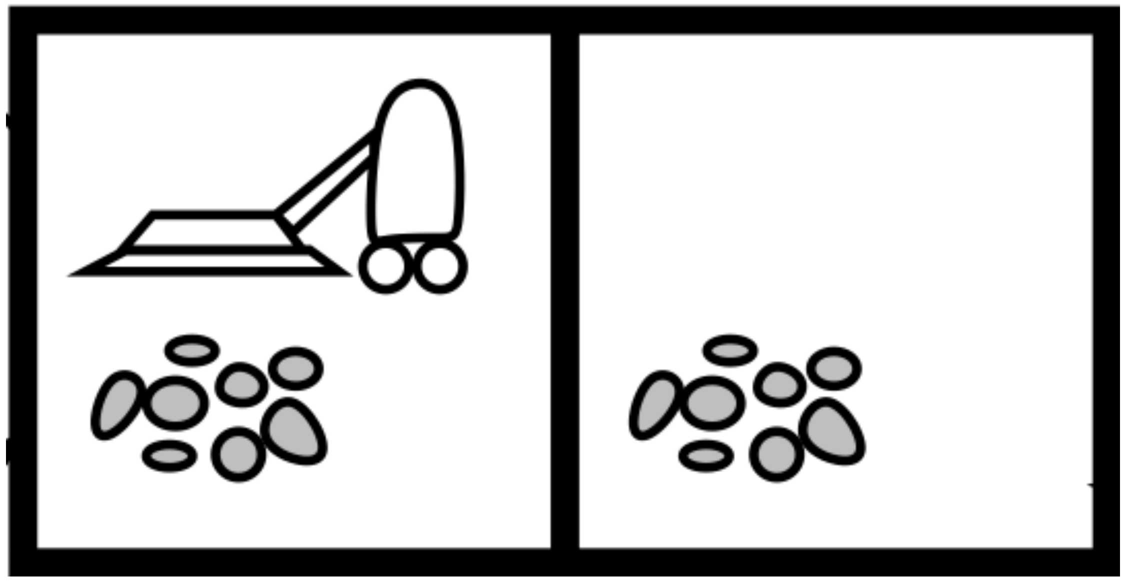
- We proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost.
- This formulation seems reasonable, but it is still a *model* – an abstract mathematical description – and not the real thing.
- The process of removing detail from a representation is called **abstraction**.

3.2 – Example Problems

Toy Problems vs. Real-World Problems

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

Toy Problem: Vacuum Cleaner World



Vacuum Cleaner World Problem Description

- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- **Transition model**: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy Problem: Vacuum Cleaner World

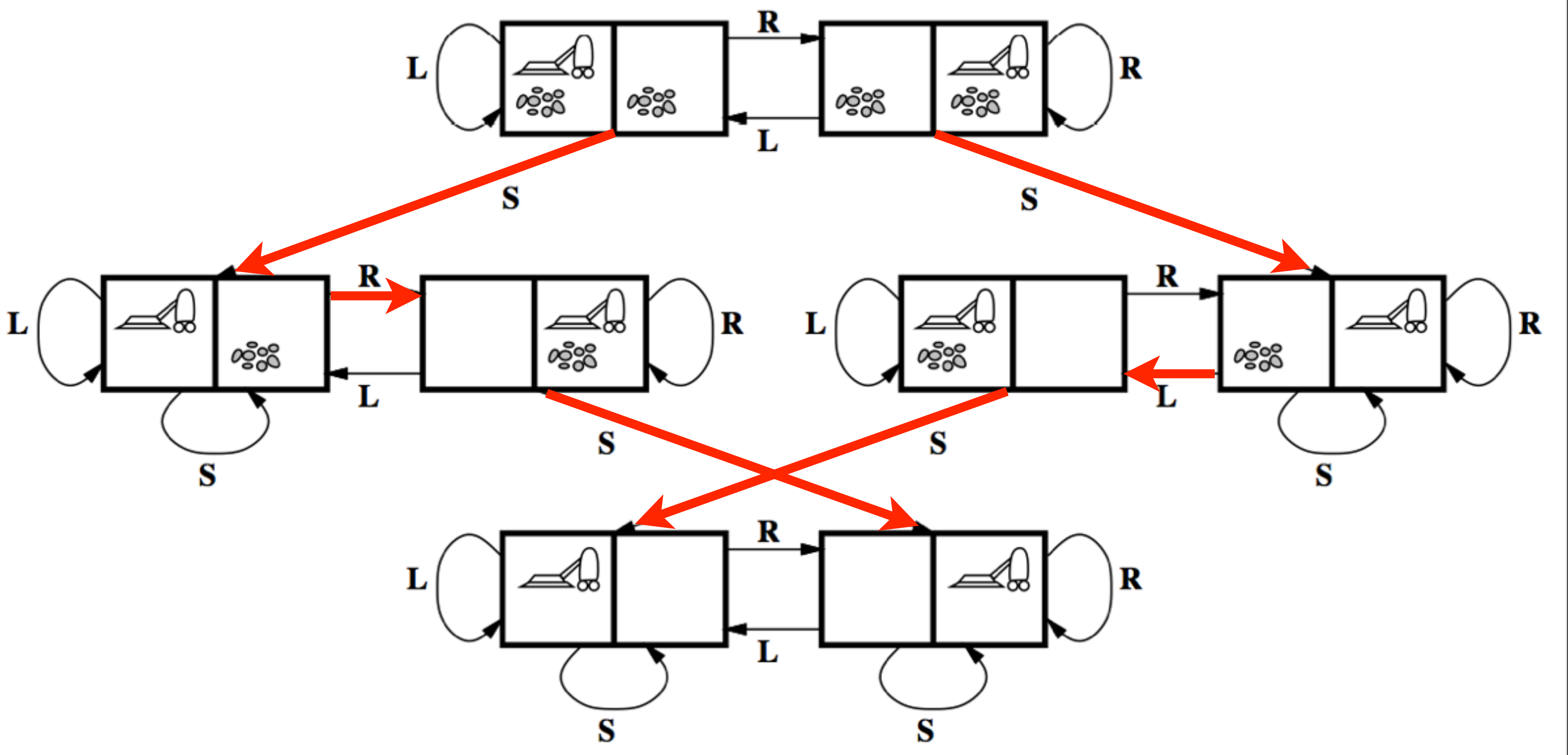


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Toy Problem: 8-Puzzle

- The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI.
- This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next.

Toy Problems: 8-Puzzle

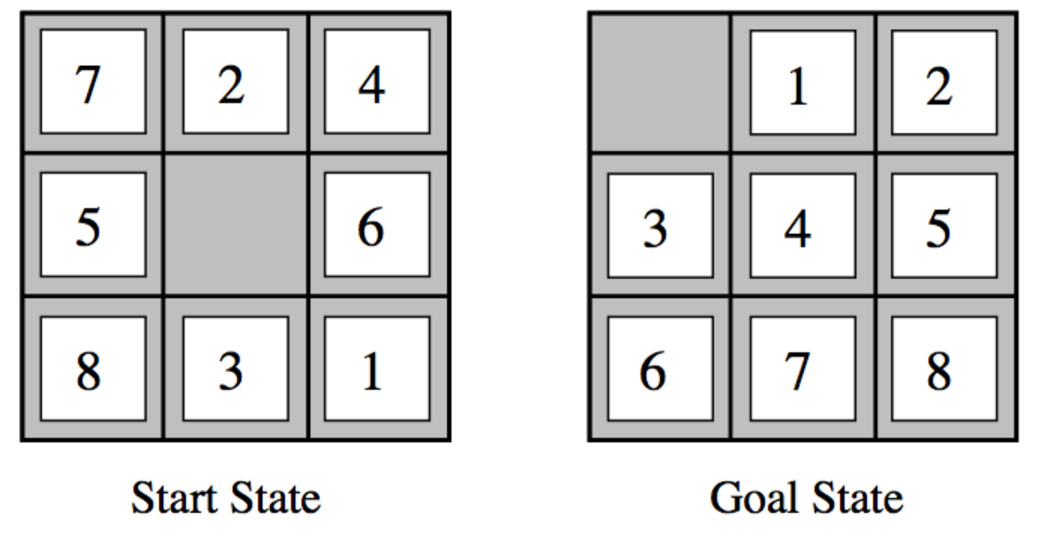


Figure 3.4 A typical instance of the 8-puzzle.

8-Puzzle Problem Description

- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- **Actions**: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy Problems: 8-Puzzle

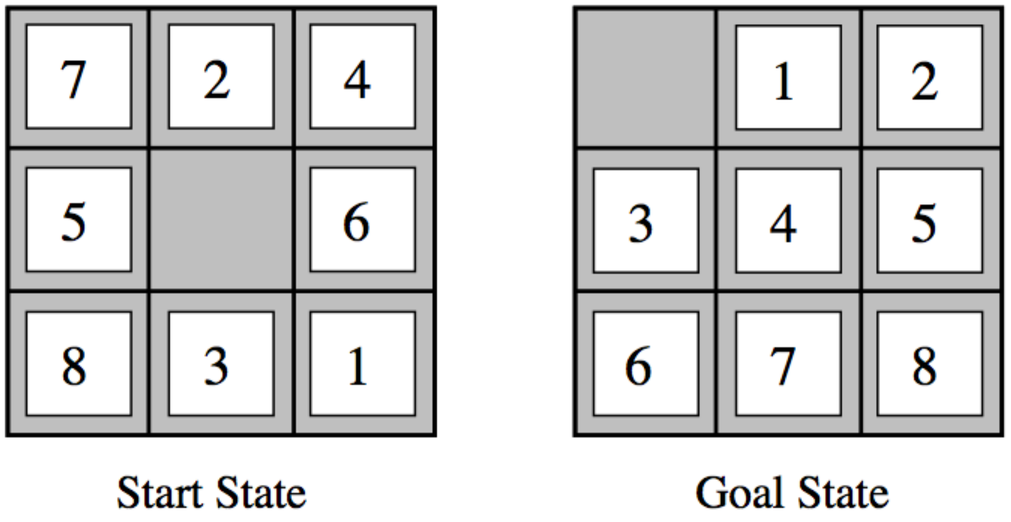


Figure 3.4 A typical instance of the 8-puzzle.

Toy Problems: 8-Puzzle

- The 8-puzzle
  - has $9!/2 = 181,440$ reachable states and is easily solved.
- The 15-puzzle (on a 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.

Toy Problem: 8-Queens Problem

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

8-Queens Problem Problem Description

- **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 the a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

Toy Problems: 8-Queens Problem

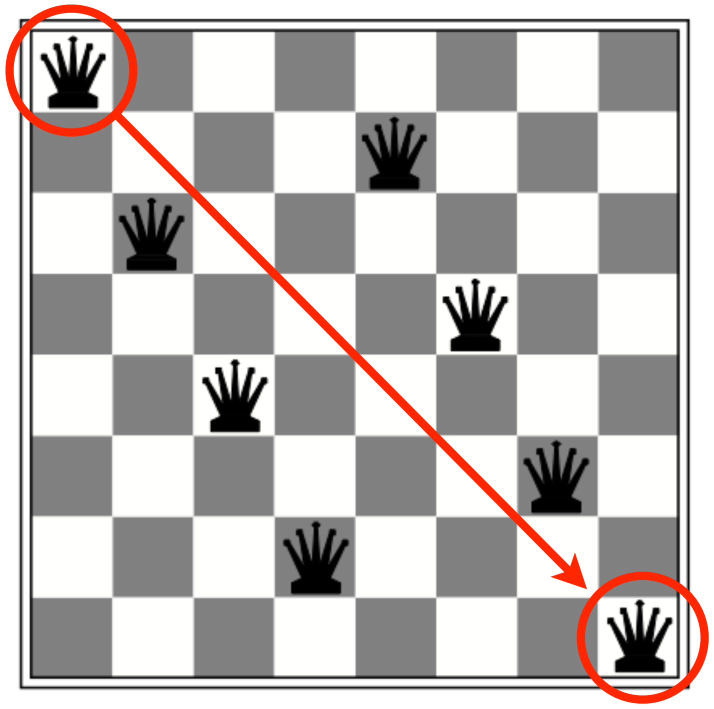


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

8-Queens: Alternative Problem Description

- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.
- A better formulation?
  - Prohibit placing a queen in any square that is already attacked
- **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

Real-World Problems

- Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.
- Consider the airline travel problems that must be solved by a travel planning Web site:

Real-world Problem: Airline Travel

- **States**: Each state obviously includes a location (e.g., an airport) and the current time.
- **Initial state**: This is 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 there is a preceding flight segment.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?

Real-world Problem: Airline Travel

- **Path cost**: This depends on
  - monetary cost,
  - waiting time,
  - flight time,
  - customs and immigration procedures,
  - seat quality,
  - time of day,
  - type of airplane,
  - frequent-flyer mileage awards, and so on.

Real-World Search Problems

- Touring Problems
- Traveling Salesperson Problem (TSP)
- VLSI Layout
- Robot navigation
- Automatic assembly sequencing

3.3 – Searching For Solutions

Search

- Search algorithms treat *states* and *actions* as **atomic**:
  - They do not consider any internal structure they might possess.
- TREE-SEARCH – a general algorithm that considers all possible paths to find a solution
- GRAPH-SEARCH – a algorithm avoids consideration of redundant paths.
- Search algorithms are judged on the basis of
  - **completeness**,
  - **optimality**,
  - **time complexity**, and
  - **space complexity**.

Measuring problem-solving performance

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution, (as defined on page 69)?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

TREE-SEARCH

- The process of choosing and expanding nodes in the ***frontier*** continues until either a solution is found or there are no more states to be expanded.
- Search algorithms all share this basic structure
  - the primary difference is how they choose which state to expand next—the so-called **search strategy**.
- The general TREE-SEARCH algorithm is shown *informally* on the next slide.

TREE-SEARCH

```
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
```

TREE-SEARCH

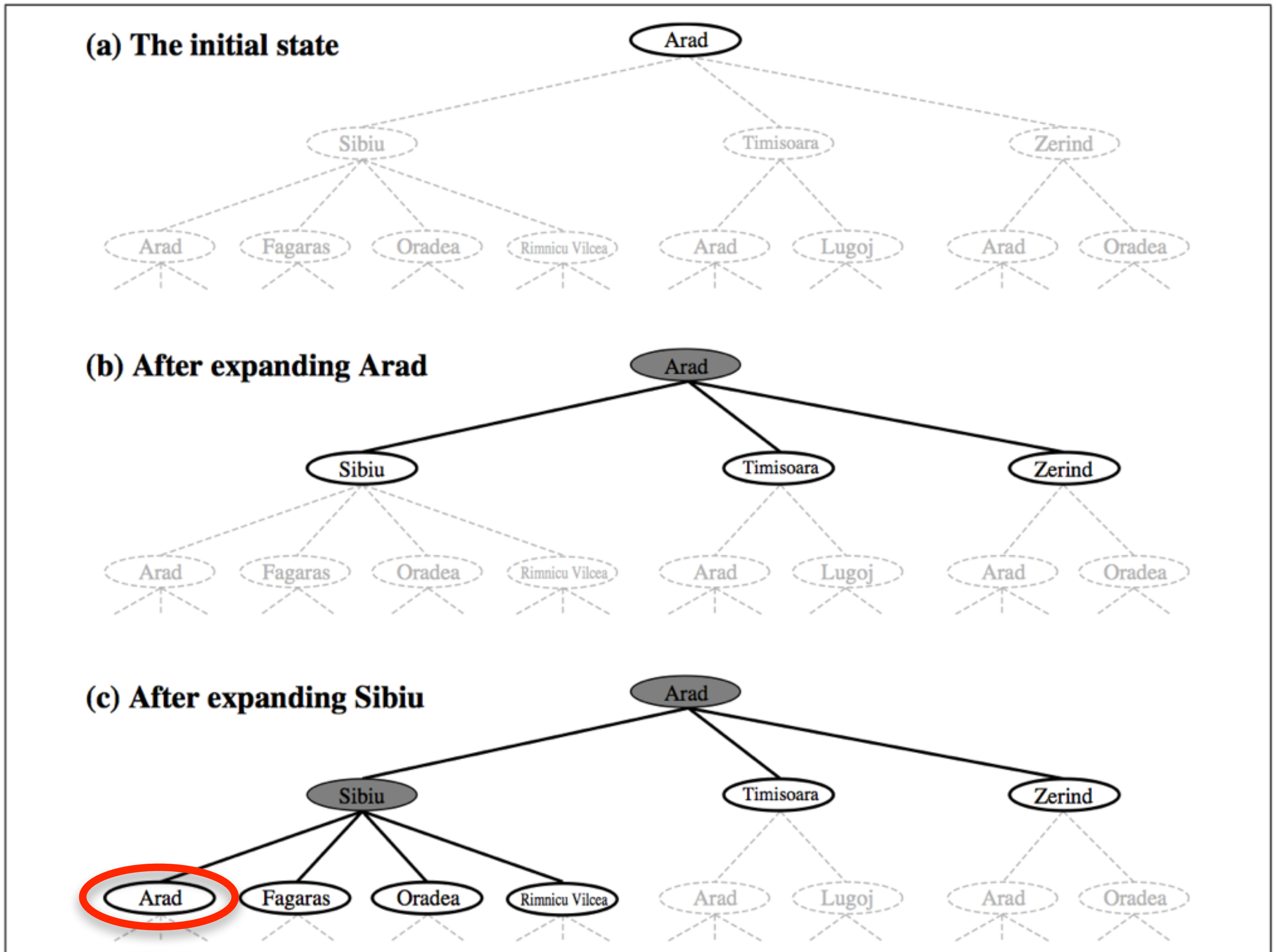
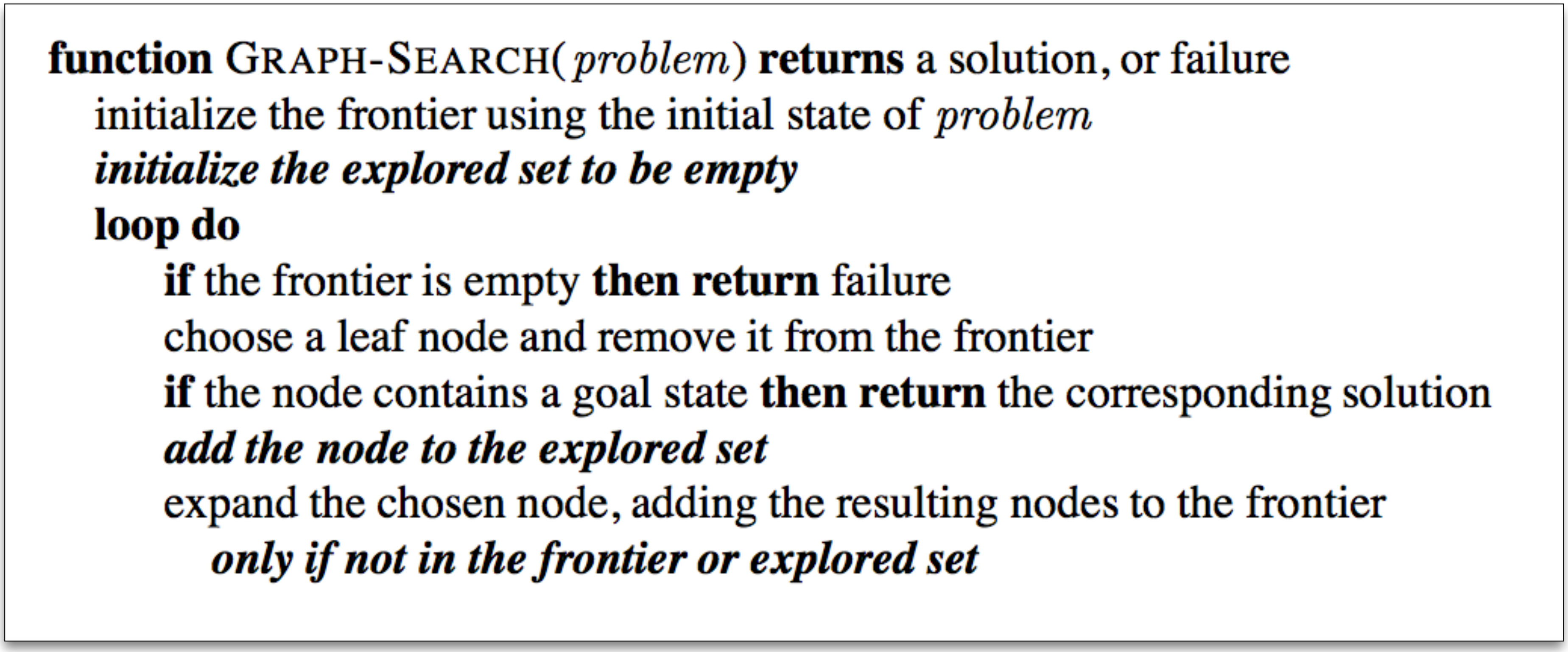


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

GRAPH-SEARCH

- The way to avoid exploring redundant paths is to remember where one has been.
- To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set**, which remembers every expanded node (i.e. the *closed list*)
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.
- The new algorithm, called GRAPH-SEARCH (next slide).
  - The specific algorithms in this chapter are, for the most part, special cases or variants of this general design.

GRAPH-SEARCH



```
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
```

Search Frontier: Tree-Search

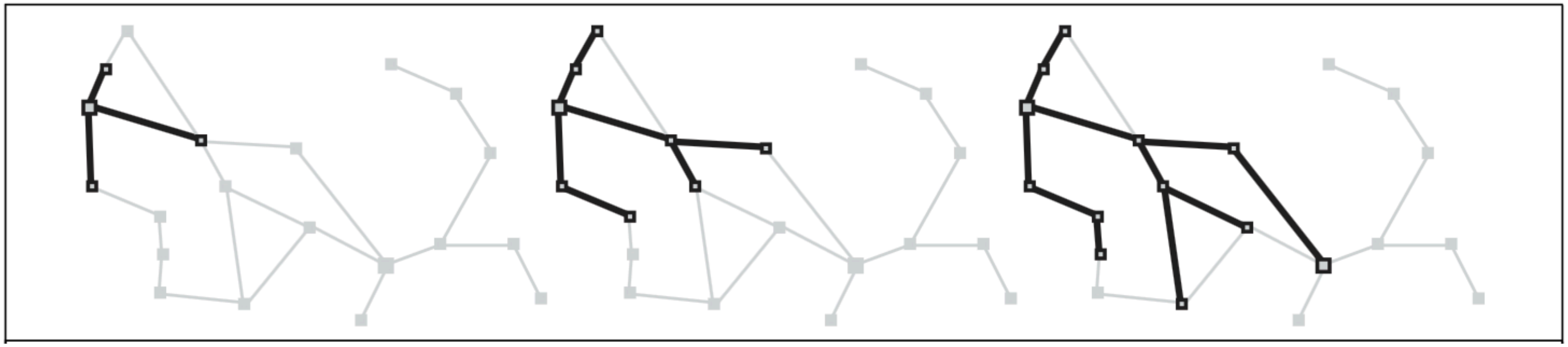


Figure 3.8

- 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.

Search Frontier: Graph-Search

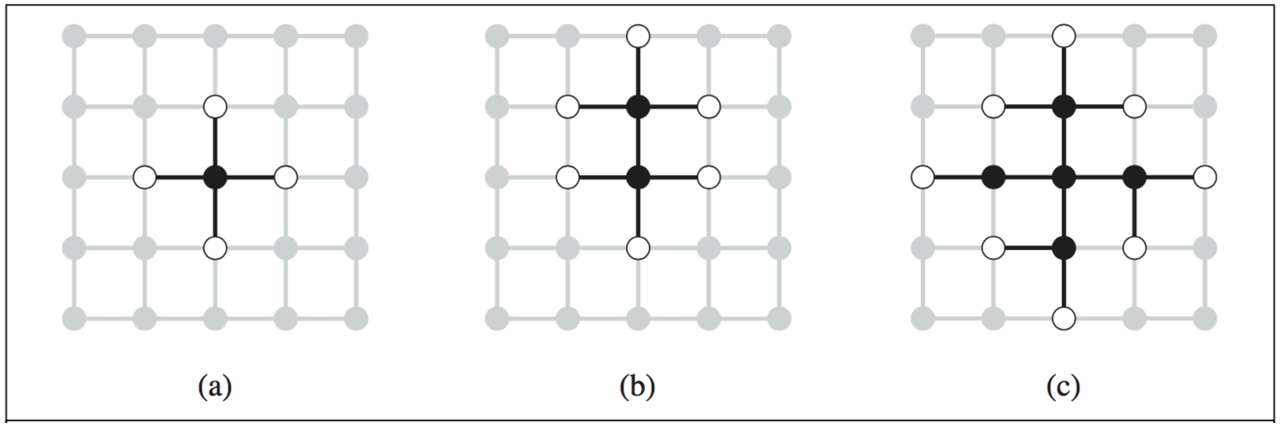


Figure 3.9

- The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes).
  - In (a), just the root has been expanded.
  - In (b), one leaf node has been expanded.
  - In (c), the remaining successors of the root have been expanded in clockwise order.

Infrastructure For Search

- Search algorithms require a **data structure** to keep track of the search tree that is being constructed. For each node *n* of the tree, we will have a structure that contains the following four components:
  - *n*.State
  - *n*.Parent
  - *n*.Action
  - *n*.Path-cost
- This infrastructure used for all GRAPH-SEARCH based algorithms in this chapter

Environment Scope

- Parameters of search
  - **Branching factor** b – how many maximum adjacent nodes
  - **Depth** d – maximum depth of *goal*
  - **Cost** $g(n)$ – cost to reach node n
    - Keep a history of $g(n)$ on the frontier

3.4 – Uninformed Search Strategies

Uninformed Search

- **Uninformed search** – i.e. algorithms that are given no information about a problem except its problem definition.
- Can only 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
- Although some can solve any solvable problem, none of them can do so efficiently

Uninformed Search

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

Breadth-first search

- Breadth-first search is an instance of the general graph search algorithm in which the shallowest unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier.
- Thus, new nodes (which are always deeper than their parents) go to the back of the queue and old nodes, which are shallower than the new nodes, get expanded first.

Breadth-first search

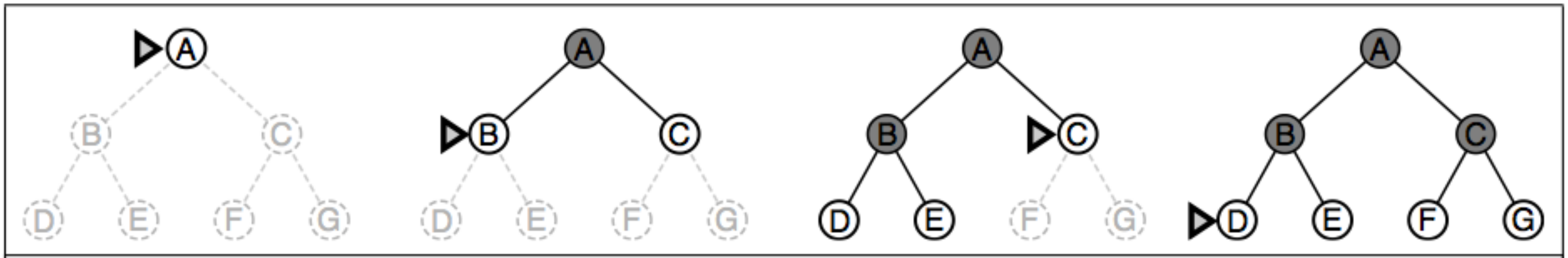
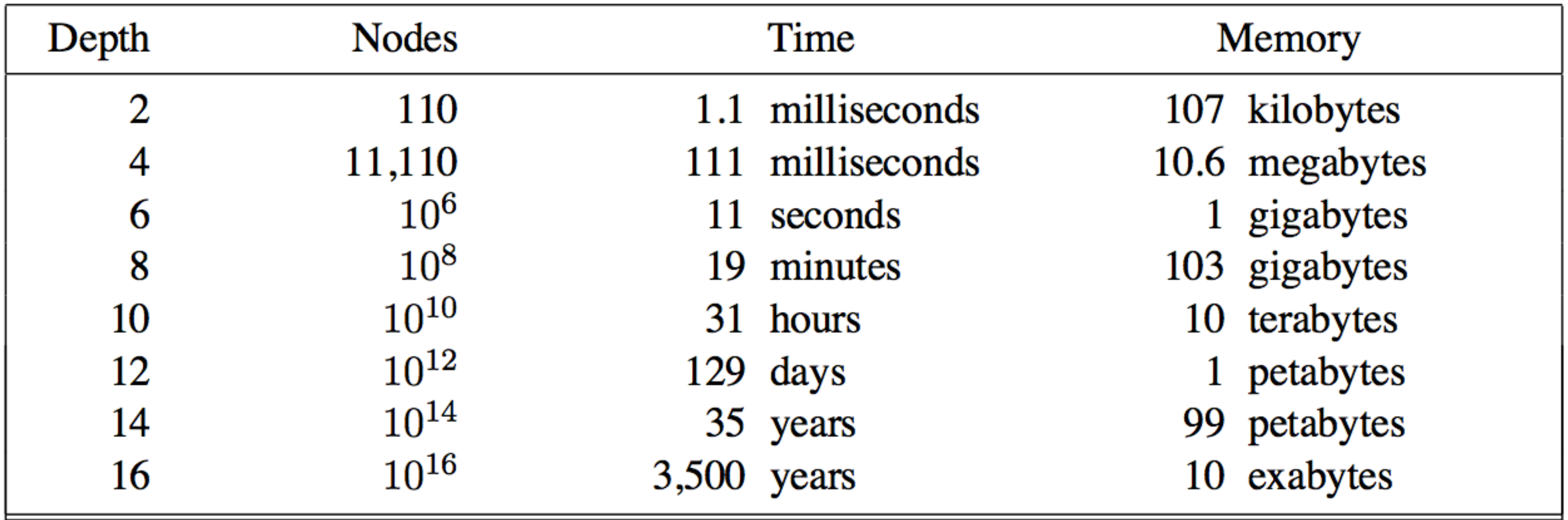


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-first search

- Complete?
  - If the shallowest goal node is at some finite depth *d*, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor *b* is finite).
  - Note that as soon as a goal node is generated, we know ***it is the shallowest goal node*** because all shallower nodes must have been generated already and failed the goal test.
- Optimal?
  - Shallowest goal node is ***not necessarily the optimal one***
  - path cost is a nondecreasing function of the depth of the node. The most common such scenario is when all actions have the same cost

BFS: Time and Memory



Depth	Nodes	Time	Memory
2	110	1.1 milliseconds	107 kilobytes
4	11,110	111 milliseconds	10.6 megabytes
6	10^6	11 seconds	1 gigabytes
8	10^8	19 minutes	103 gigabytes
10	10^{10}	31 hours	10 terabytes
12	10^{12}	129 days	1 petabytes
14	10^{14}	35 years	99 petabytes
16	10^{16}	3,500 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 100,000 nodes/second; 1000 bytes/node.

BFS: Analysis

- There are two lessons to be learned
  - **The memory requirements are a bigger problem for BFS than is the execution time**.
    31 hours would not be too long to wait for the solution to an important problem of depth 10, but few computers have the 10 terabytes of main memory it would take. Fortunately, there are other search strategies that require less memory.
  - **The time requirements are still a major factor**.
    If your problem has a solution at depth 16, then (given our assumptions) it will take about 3,500 years for breadth-first search (or indeed any uninformed search) to find it. In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

Uniform-cost search

- Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any *step cost* function.
- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost $g(n)$.
- This is done by storing the frontier as a priority queue ordered by g.

Uniform-cost search

- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.
    - The the goal test is applied to a node when it is selected for expansion (like generic graph search algorithm, Figure 3.7) rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path.
    - A test is added in case a better path is found to a node currently on the frontier.

Depth-first search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- Explored nodes with no descendants in the frontier are removed from memory.
  - Not a big advantage with Breadth-first Search. Why?

Depth-first search

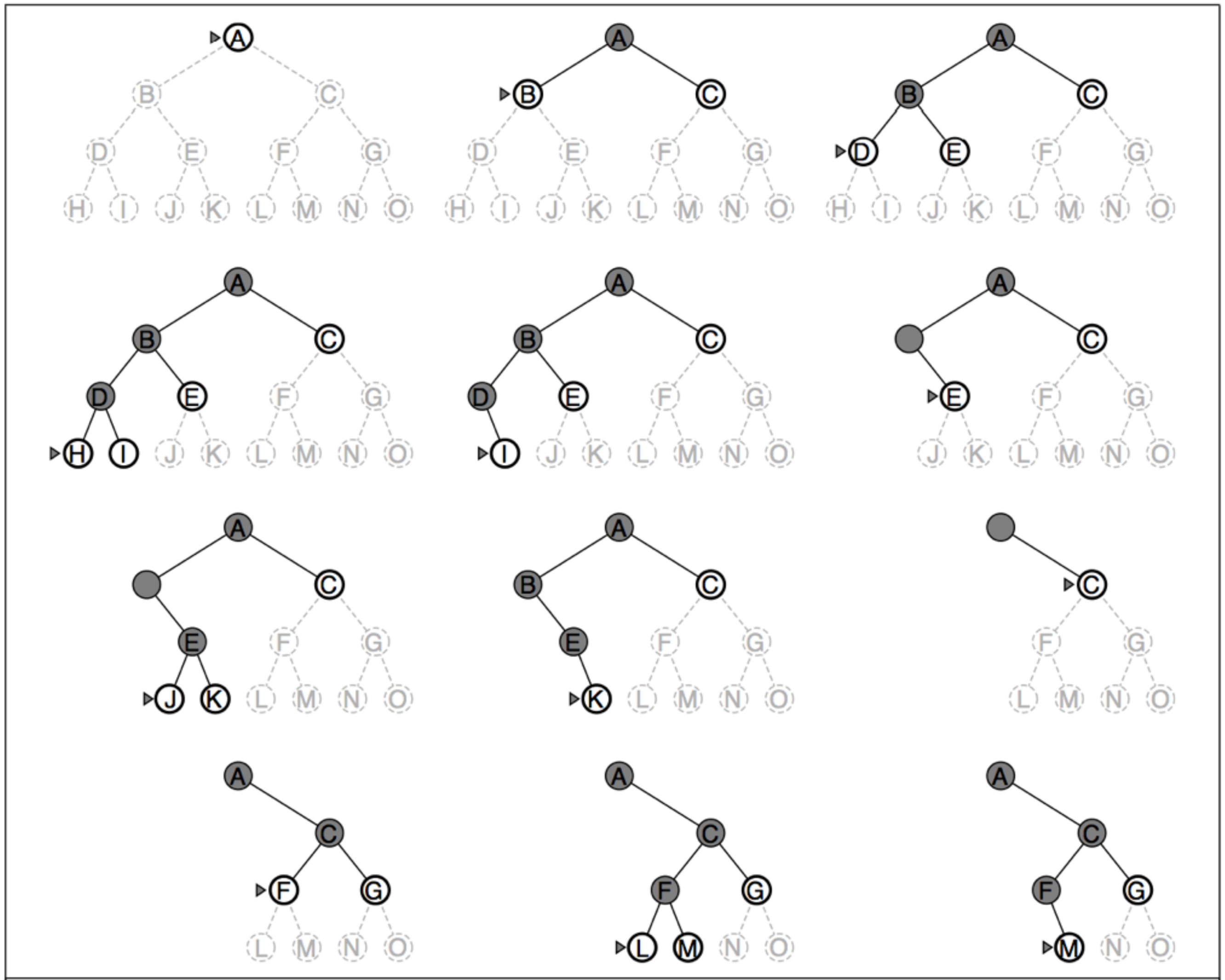


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors, and M is the only goal node.

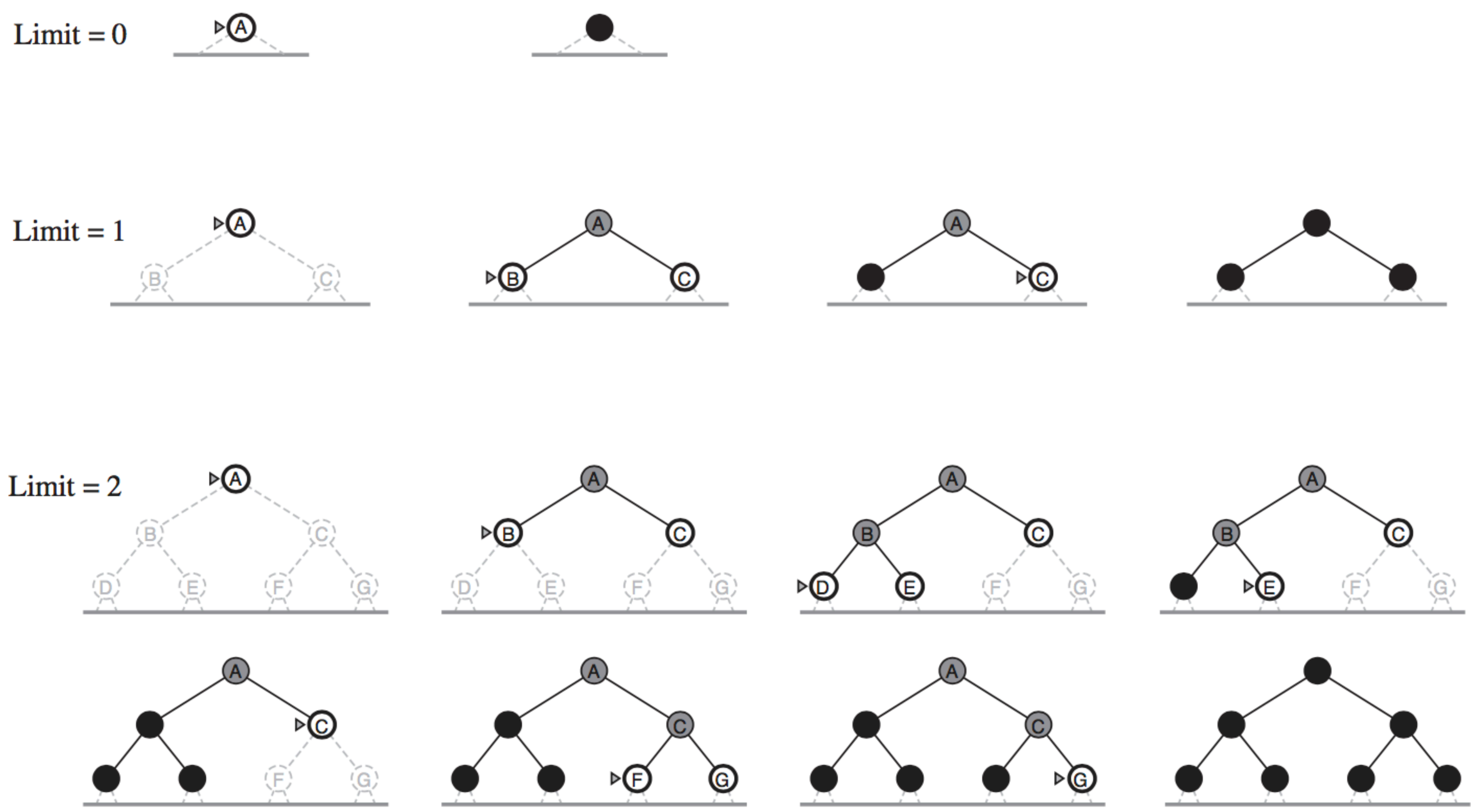
Depth-limited search

- Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.
- Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $l = 19$ is a possible choice.
- Implementation
  - Depth-limited search can be implemented as a simple modification to the general tree or graph search algorithm.
  - Alternatively, it can be implemented as a simple recursive algorithm.

Iterative deepening search

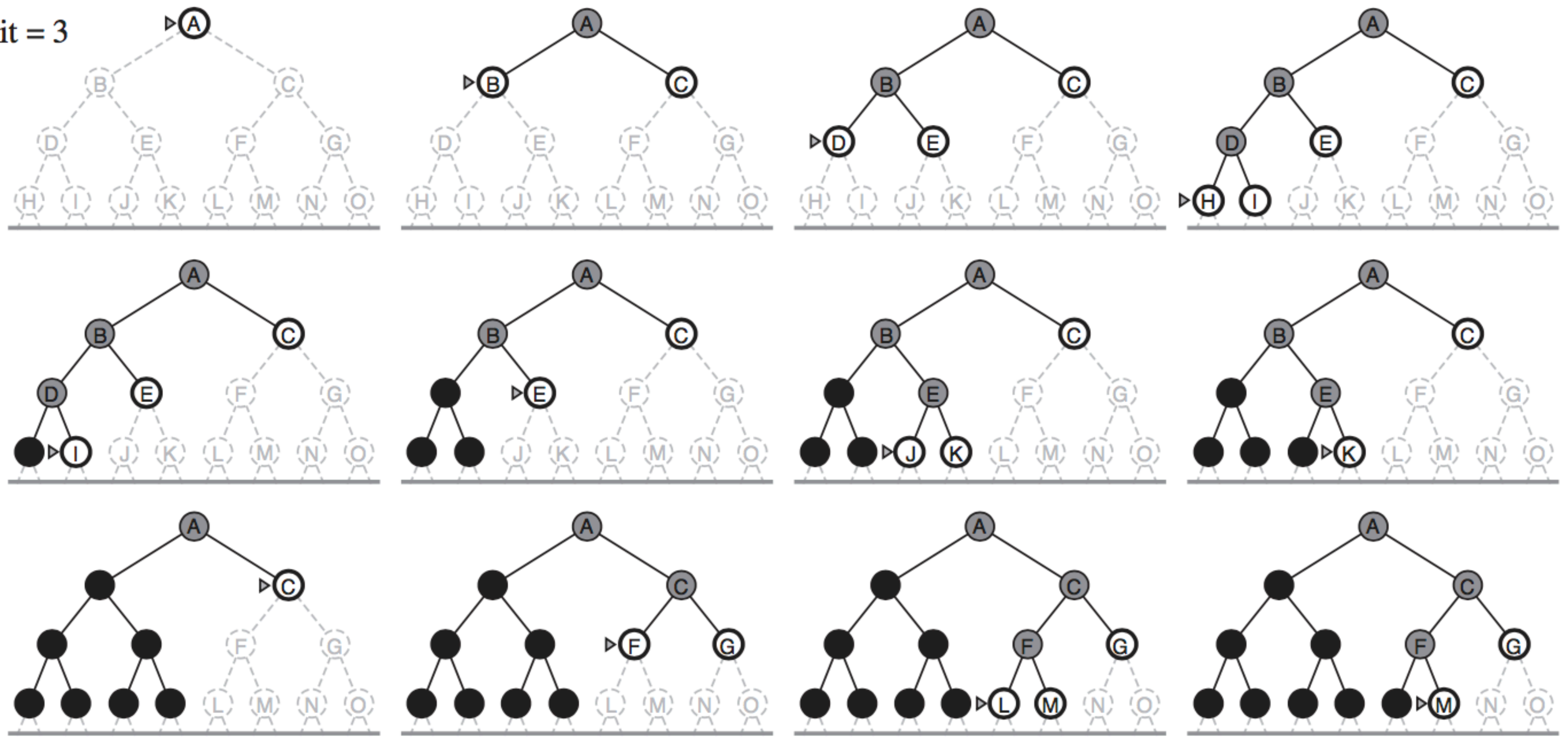
- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
  - This will occur when the depth limit reaches d, the depth of the shallowest goal node.

Iterative deepening search



Iterative deepening search

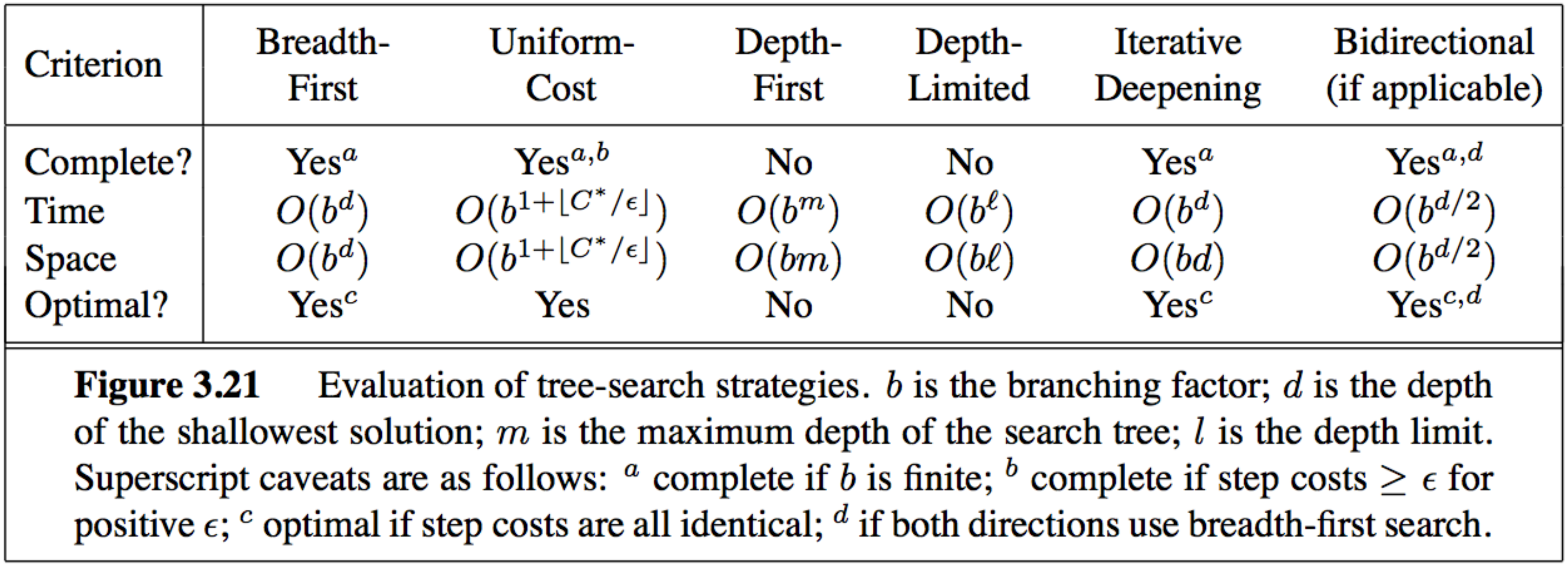
Limit = 3



Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—stopping when the two searches meet in the middle
- For example, if a problem has solution depth $d = 6$, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3.

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.

3.5 – Informed Search Strategies

Informed Search

- **Informed Search** can do well given some guidance on where to look for solutions.
- Makes an *informed* decision on which node to expand next.
- Informed Search methods may have access to a heuristic function $h(n)$ that estimates the cost of a solution from n.
- Notice that $h(n)$ takes a node as input, but, unlike $g(n)$, it depends only on the state at that node.

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

Informed Search

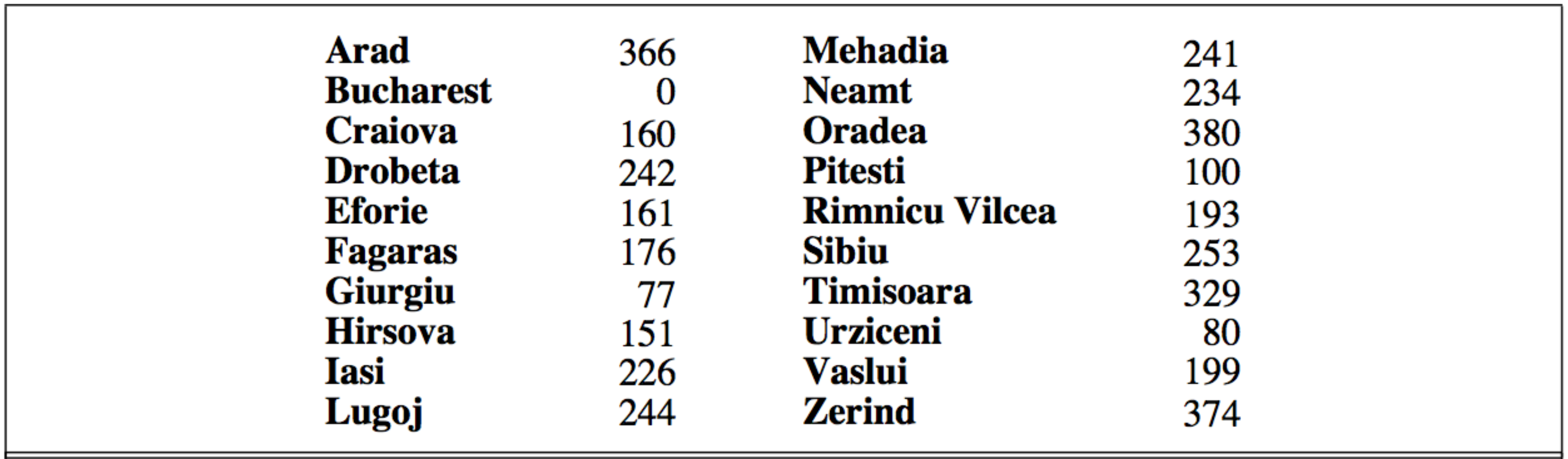
- **This is not an exhaustive list**
  - Best-first Search
  - Greedy Best-first Search
  - A* Search
  - Space-bounded A*
    - Iterative-Deepening A* (IDA*)
    - Recursive Best-First Search (RBFS)
    - Memory-bounded A* (MA*)
    - Simplified Memory-bounded A* (SMA*)

Greedy Best-first Search

- Tries to expand the node on the frontier that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function:

$$f(n) = h(n)$$

Proposed $h(n)$ Values for Romanian Map

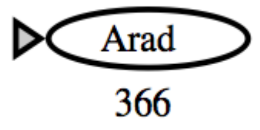


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

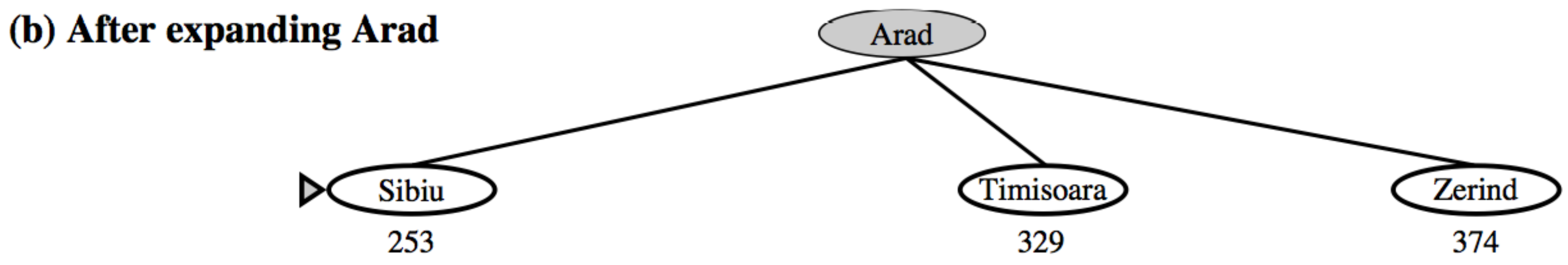
Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

Greedy Best-First Search

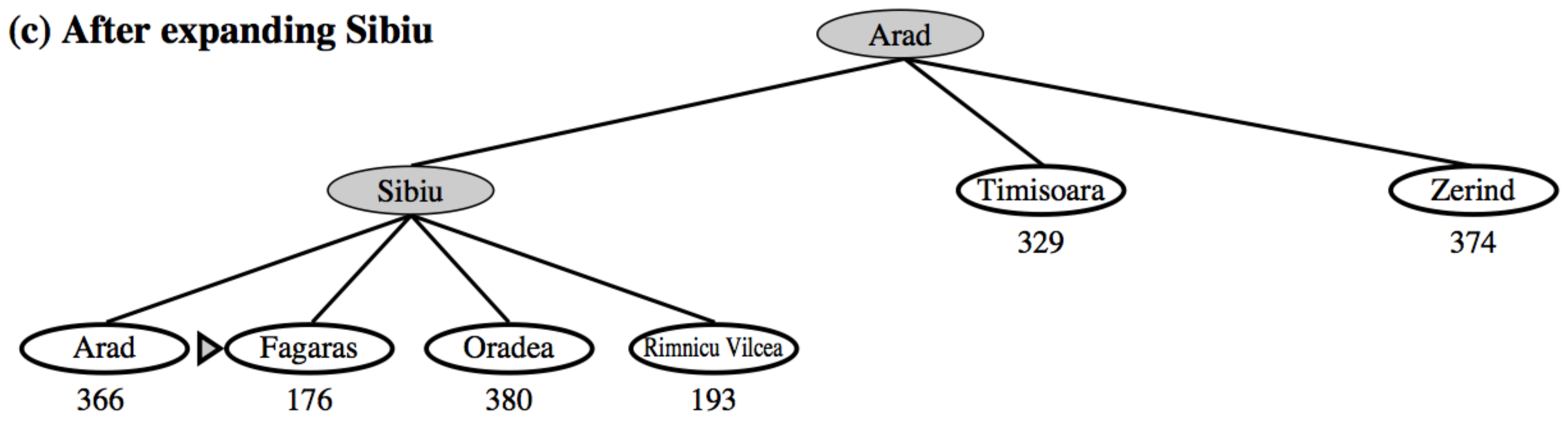
(a) The initial state



Greedy Best-First Search

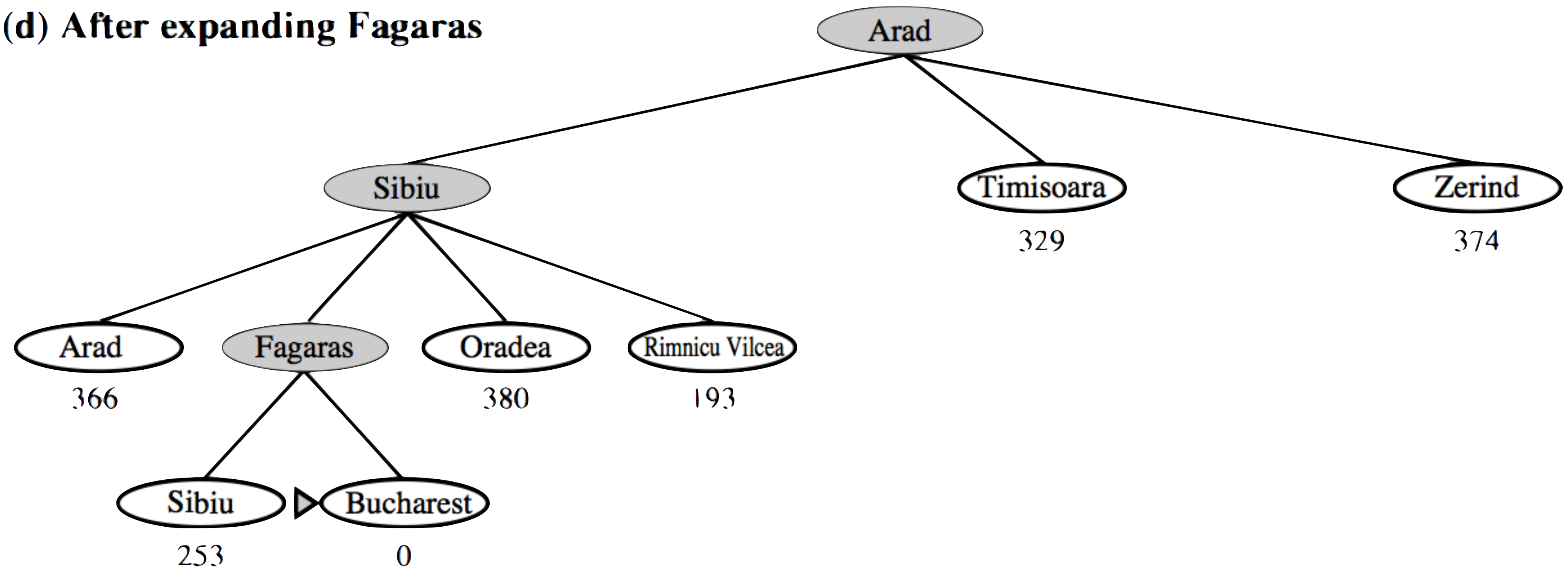


Greedy Best-First Search



Greedy Best-First Search

(d) After expanding Fagaras



A* Search

- Evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

$$f(n) = g(n) + h(n)$$

- Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have:

$f(n)$ = *estimated* cost of the cheapest solution through n

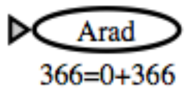
Conditions for Optimality

- Admissibility
  - One that never overestimates the cost to reach the goal
  - Because $g(n)$ is the actual cost to reach n, and $f(n) = g(n) + h(n)$, we have as immediate consequence that $f(n)$ never overestimates the true cost of a solution through n
- Consistency
  - Required only for the graph-search version of A*
  - For every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

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

A* Search

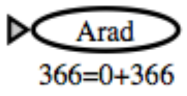
(a) The initial state

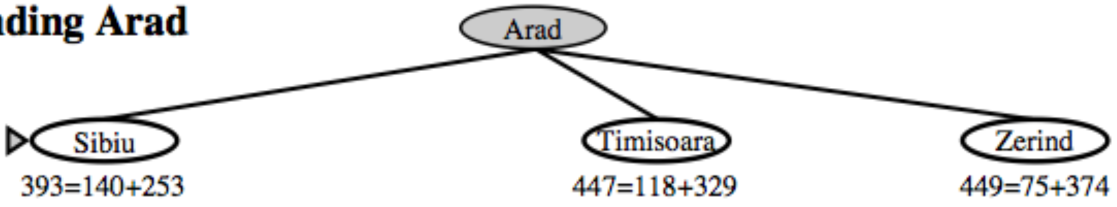


A* Search

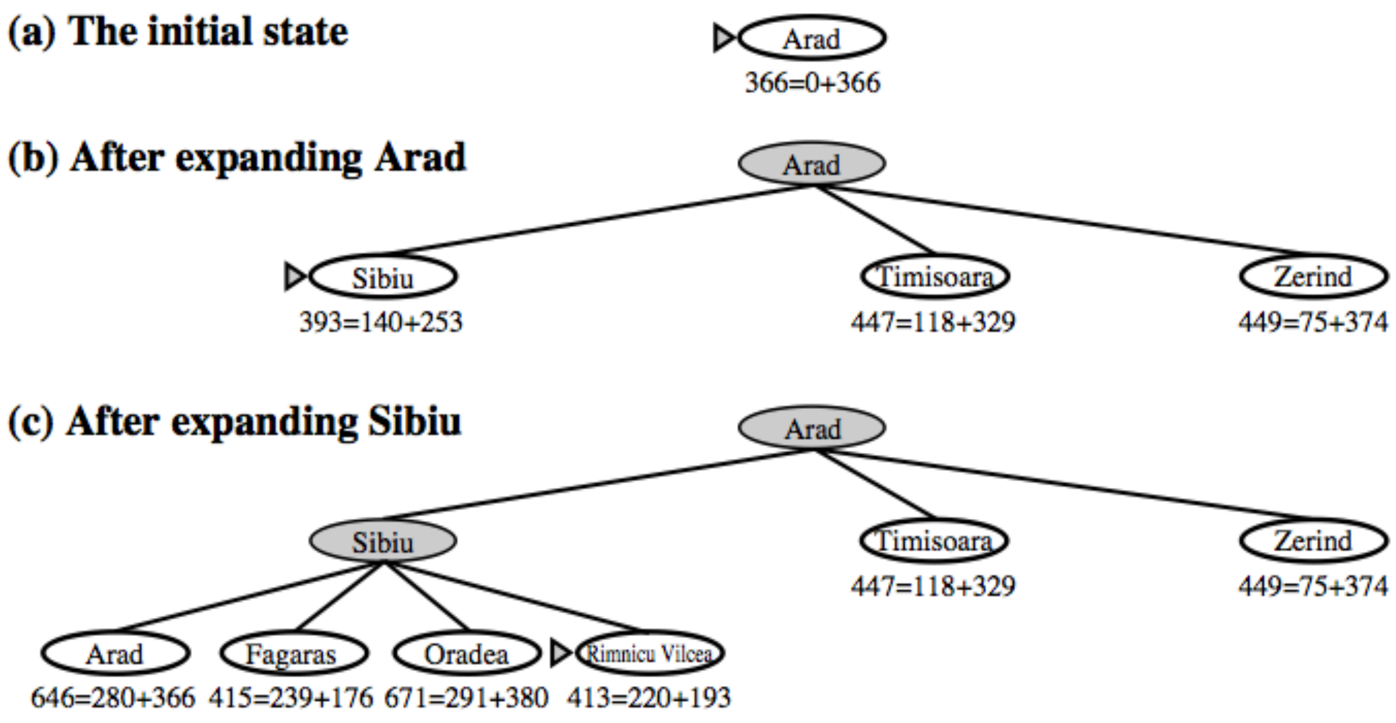
(a) The initial state

(b) After expanding Arad

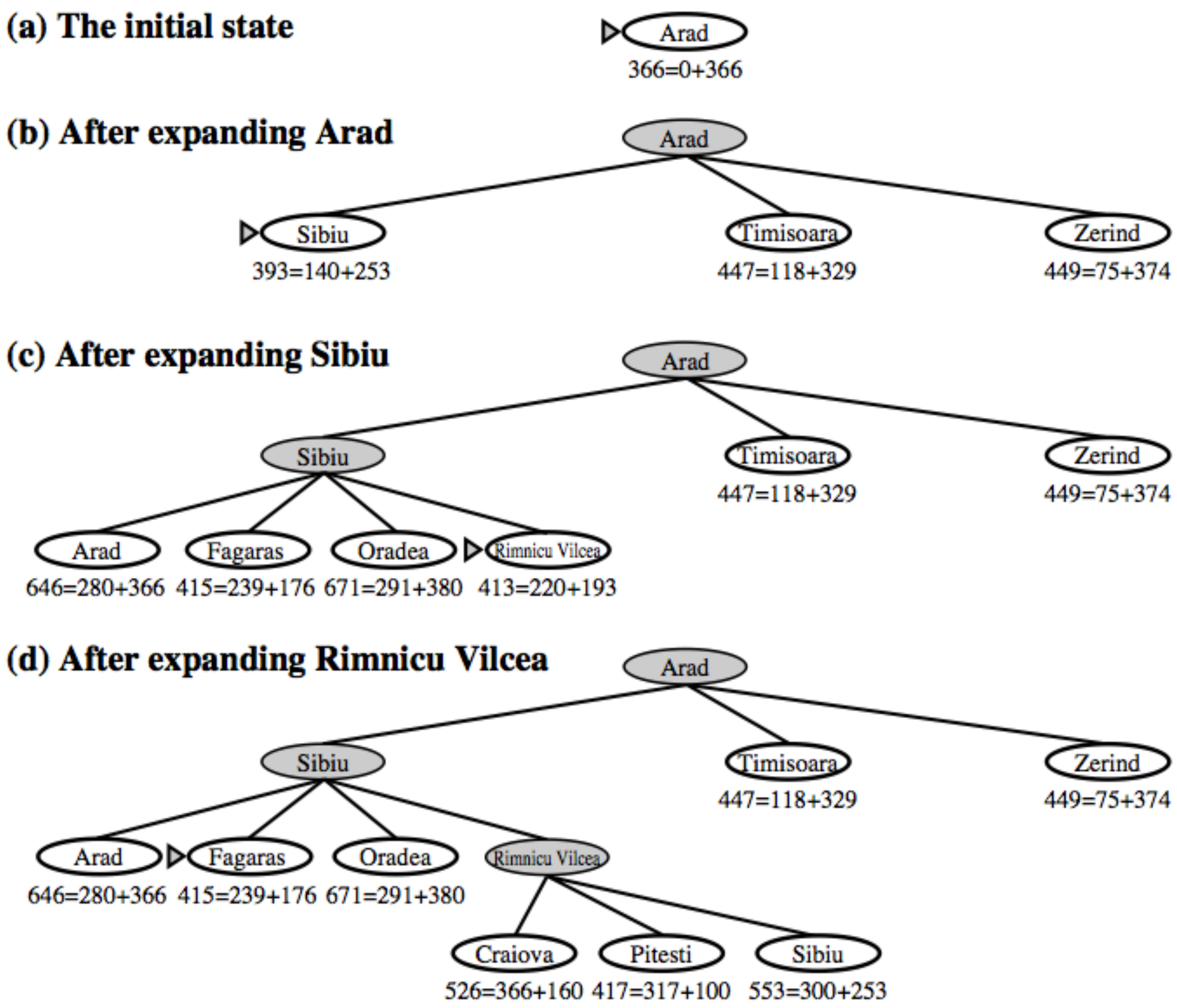




A* Search

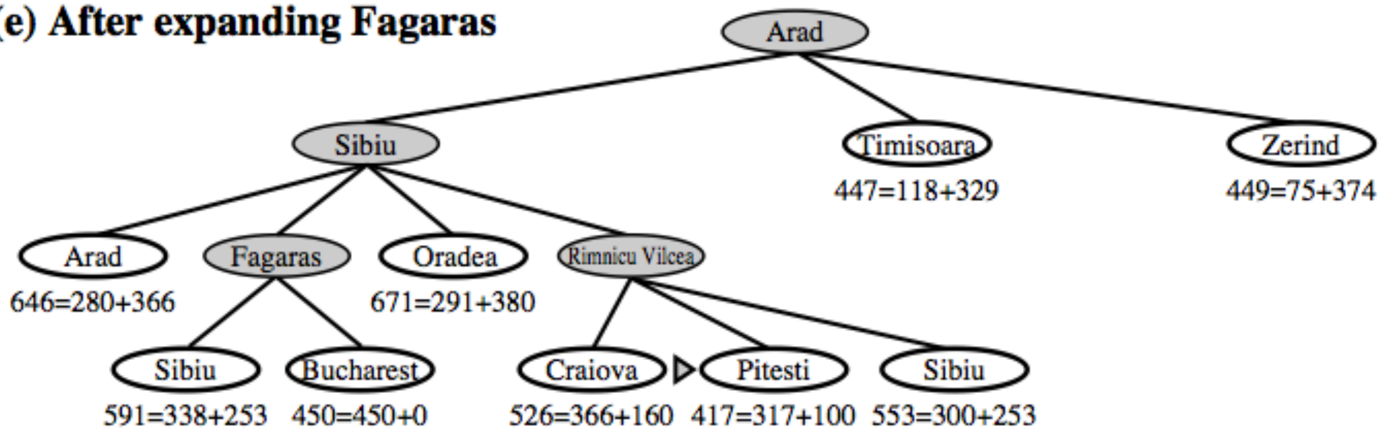

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* Search

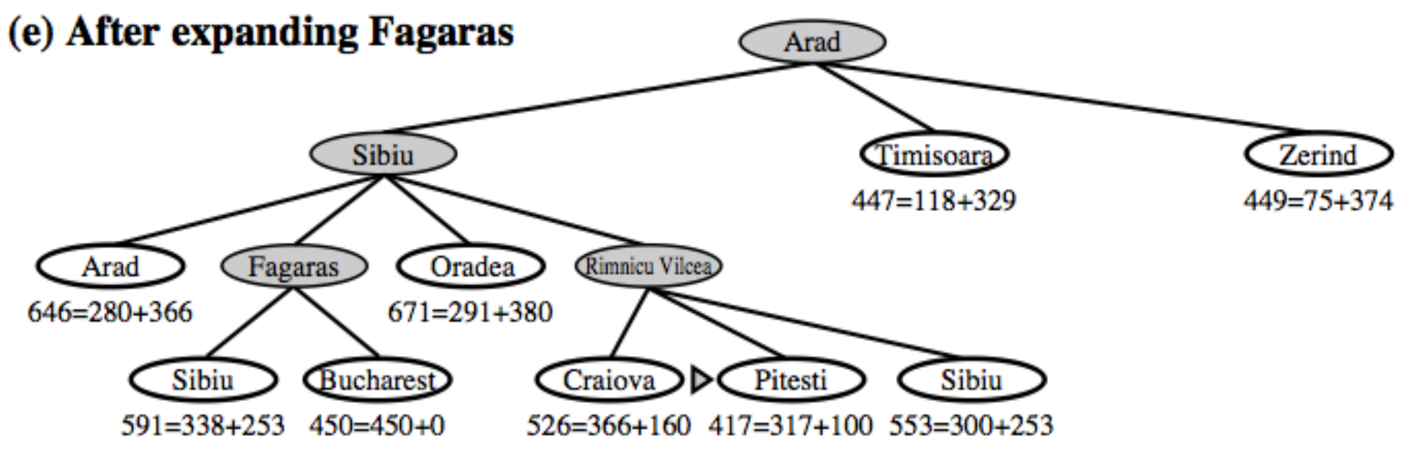

(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
(d) After expanding 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

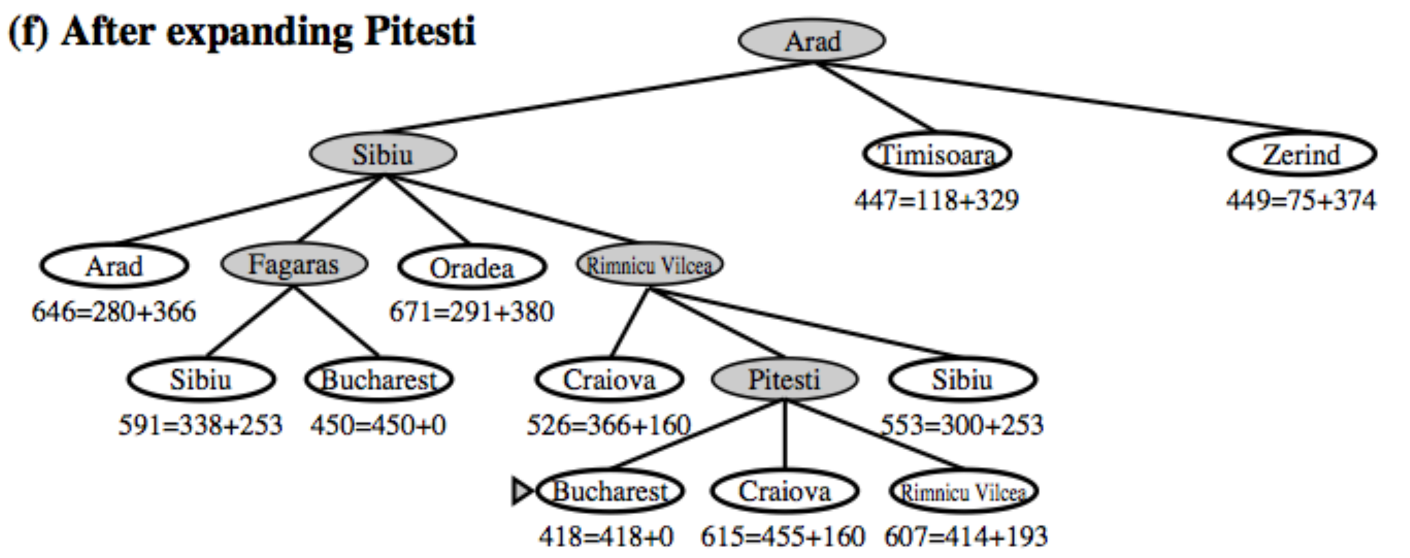
A* Search

(e) After expanding Fagaras



A* Search


(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253


(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

A* Evaluation

- The complexity of A* often makes it impractical to insist on finding an optimal solution.
  - One can use variants of A* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible.
  - In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.
- Computation time is not, however, A*'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A* usually runs out of **space** long before it runs out of time. For this reason, A∗ is not practical for many large-scale problems.

Space-bounded A*

- Iterative-Deepening A* (IDA*)
- Recursive Best-First Search (RBFS)
- Memory-bounded A* (MA*)
- Simplified Memory-bounded A* (SMA*)

Iterative-Deepening A* (IDA*)

- Simple way to reduce memory requirements for A*
- Adapt the idea of iterative deepening to the heuristic search
- The main difference between IDA* and standard iterative deepening is that the cutoff used is the *f*-cost ($g + h$) rather than the depth; at each iteration, the cutoff value is the smallest *f*-cost of any node that exceeded the cutoff on the previous iteration.
- IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

Iterative-Deepening A* (IDA*)

- IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.
- Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search described in Exercise 3.24. This section briefly examines two more recent memory-bounded algorithms, called RBFS and MA*.

Recursive Best-First Search (RBFS)

- Similar to recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f*-limit variable to keep track of the *f*-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path.
- As the recursion unwinds, RBFS replaces the *f*-value of each node along the path with backed-up value—the best *f*-value of its children. In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time.

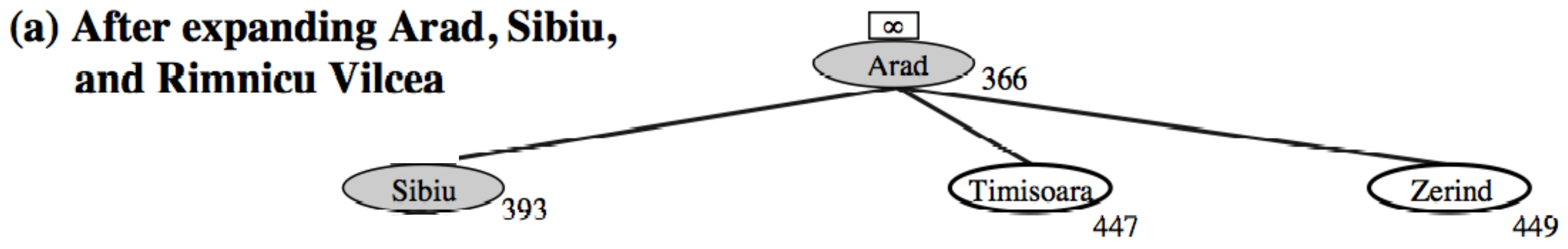
Recursive Best-First Search (RBFS)

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

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

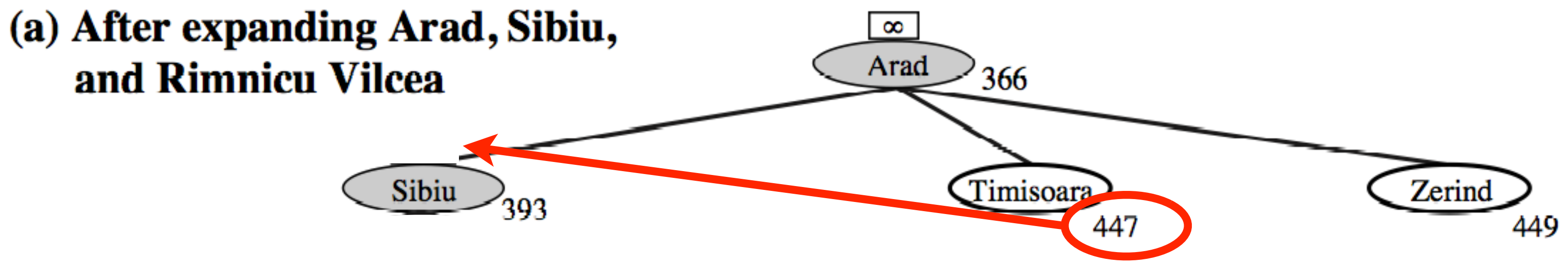
Figure 3.26 The algorithm for recursive best-first search.

Recursive Best-First Search (RBFS)



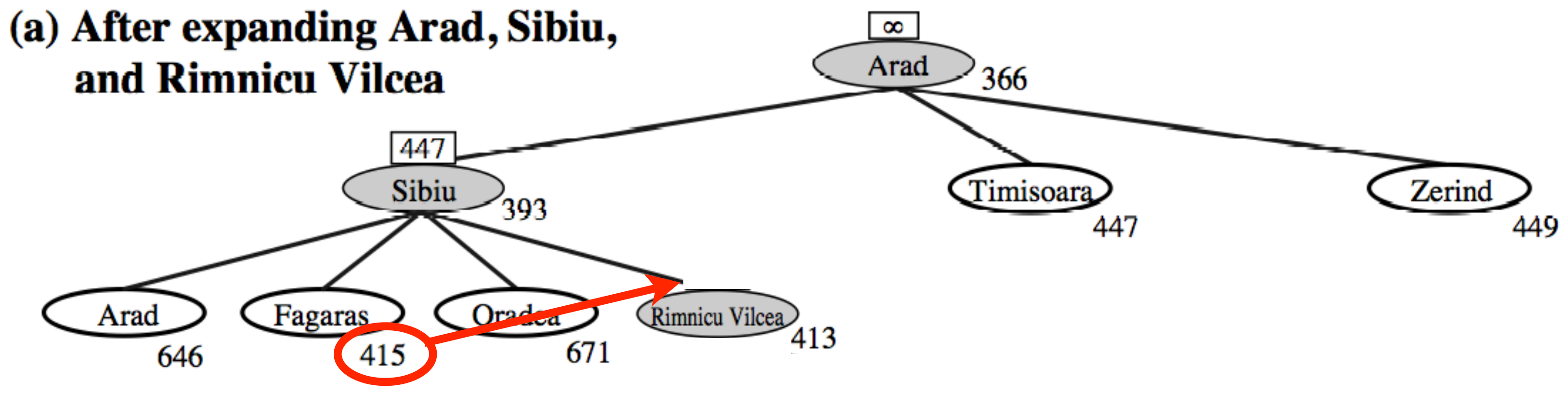
- Stages in an RBFS search for the shortest route to Bucharest.
- The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost.

Recursive Best-First Search (RBFS)



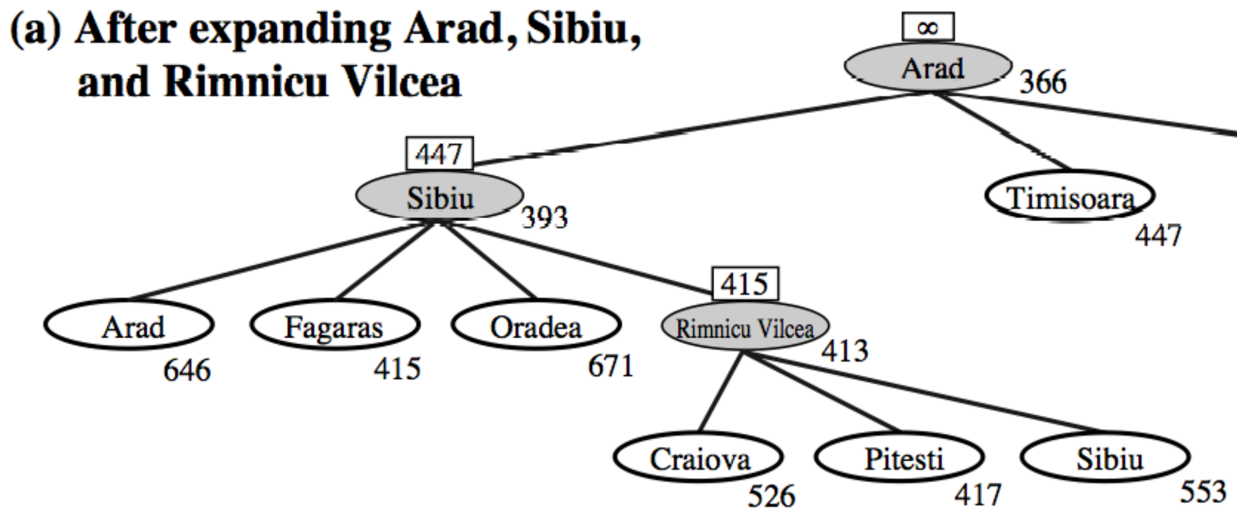
- Stages in an RBFS search for the shortest route to Bucharest.
- The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost.

Recursive Best-First Search (RBFS)



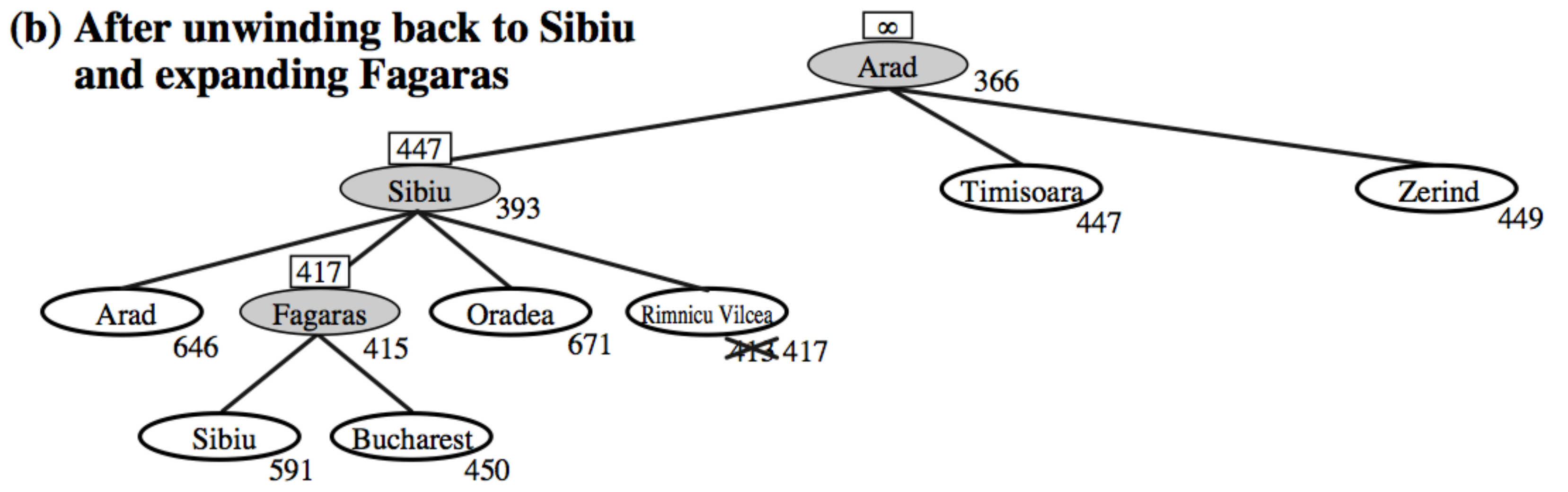
- Stages in an RBFS search for the shortest route to Bucharest.
- The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost.

Recursive Best-First Search (RBFS)



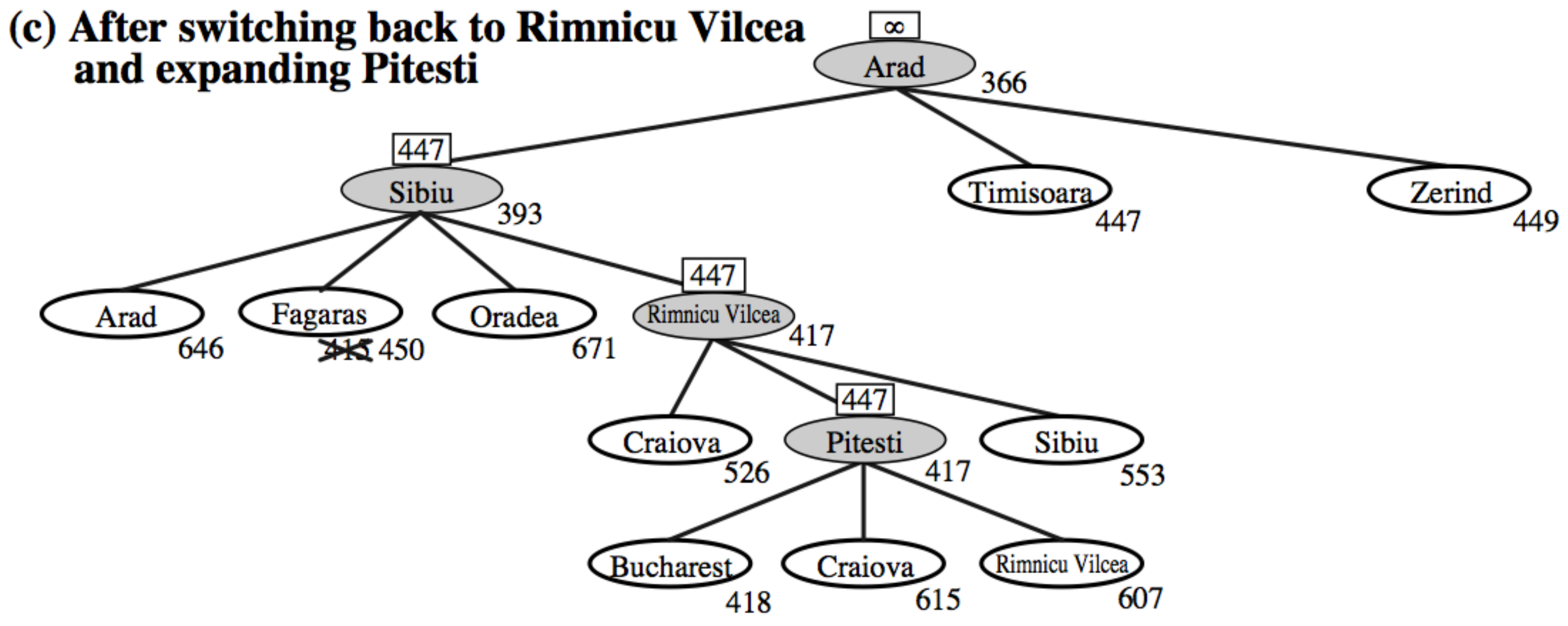
- Stages in an RBFS search for the shortest route to Bucharest.
- The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost.
- (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse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

Recursive Best-First Search (RBFS)



- (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.

Recursive Best-First Search (RBFS)



- (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.

Informed Search Summary

- General Best First Search
- Greedy Best-First Search
  - Complete? **Yes**
  - Optimal? No (but usually efficient)
- A*
  - Complete? **Yes**
  - Optimal? **Yes** (but space expensive)
- Recursive Best-First Search (RBFS)
  - Complete? **Yes**
  - Optimal? **Yes** (provided heuristic is admissible)

3.6 – Heuristic Functions

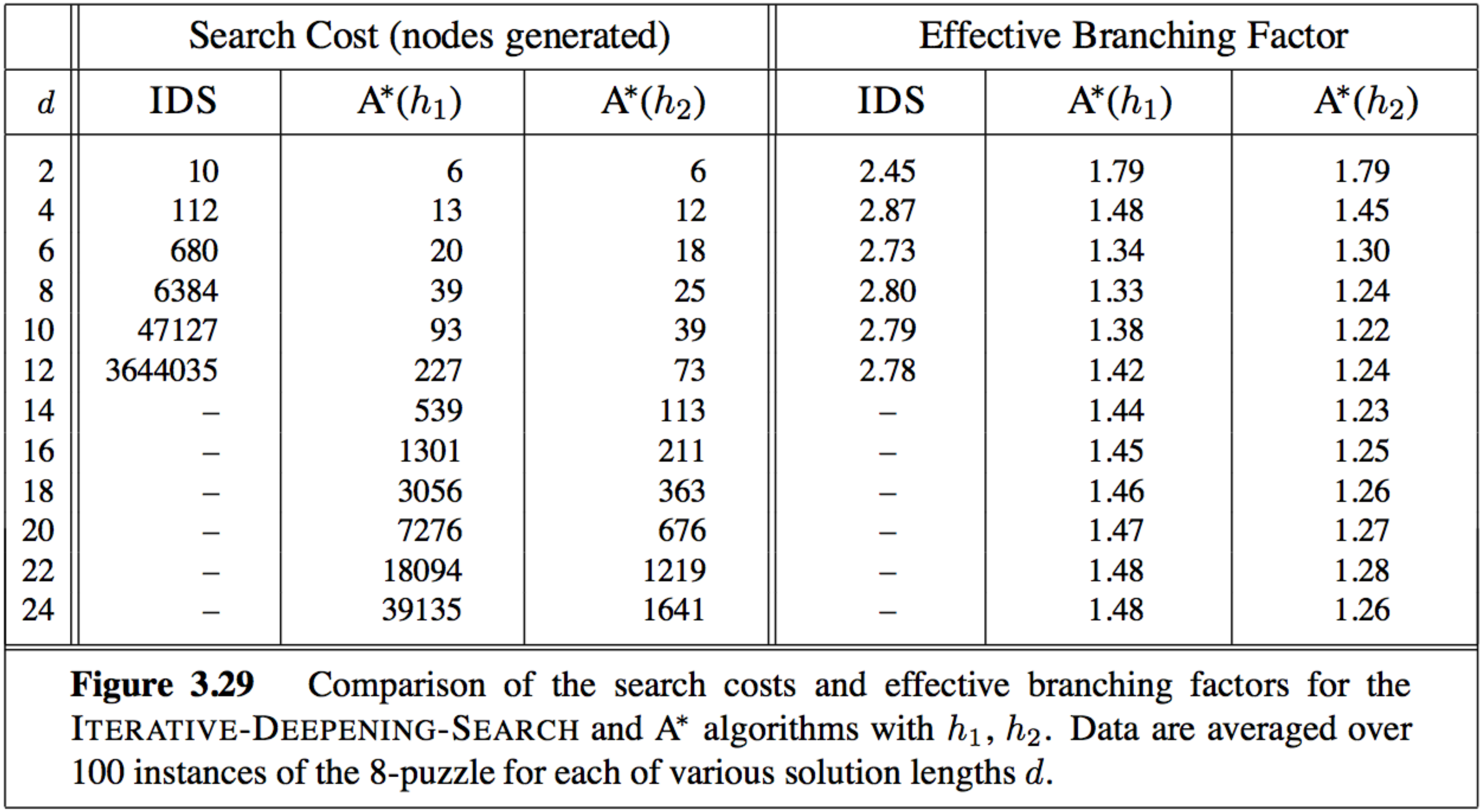
Heuristics for 8-Puzzle

- h_1 = the number of misplaced tiles
  - h_1 is an admissible heuristic, because it is clear that 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. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is 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

Effect of Heuristic Accuracy on Performance

- One way to characterize the quality of a heuristic is the **effective branching factor** b*.
- If the total number of nodes generated by A* for a particular problem is N, and the solution depth is d, then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain $N + 1$ nodes. Thus,
  - $N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$
- A well-designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved

8-Puzzle Heuristic Comparison



	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.

8-Puzzle Heuristic Comparison

- Is h_2 always better than h_1?
  - Yes
- For any node n, $h_2(n) \geq h_1(n)$
  - We say that h_2 **dominates** h_1
  - Domination translates directly into efficiency