Module 2

Problem-Solving agents

- one kind of goal-based agent called a **problem-solving agent**.
- Intelligent agents are supposed to maximize their performance measure.
- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.

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

After formulating a goal and a problem to solve,the agent calls a search procedure to solve it. It then uses the solution to guide its actions,doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal

```
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.

Searching Process

1. Formulate a goal and a problem to solve,
2. the agent calls a search procedure to solve it
3. Agent uses the solution to guide its actions,
4. do whatever the solution recommends
5. remove that step from the sequence.
6. Once the solution has been executed, the agent will formulate a new goal.

Open-loop system

- while the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be
- An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on is an open loop.
- ignoring the percepts breaks the loop between agent and environment.

Well-defined problems and solutions

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

- The **initial state** that the agent starts in.
- A description of the possible **actions** available to the agent. Given a particular state s, ACTIONS(s) returns the set of actions that can be executed in s. We say that each of these actions is **applicable** in s.

For example, from the state In(Ernakulam), the applicable actions are {Go(Thrissur), Go(Palakkad), Go(Kozhikod)}.

- A description of what each action does; the formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from doing action a in state s. We also use the term **successor** to refer to any state reachable from a given state by a single action.

- Together, the initial state, actions, and transition model implicitly define the **state space** of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or **graph** in which the nodes are states and the links between nodes are actions. **A path** in the state space is a sequence of states connected by a sequence of actions.

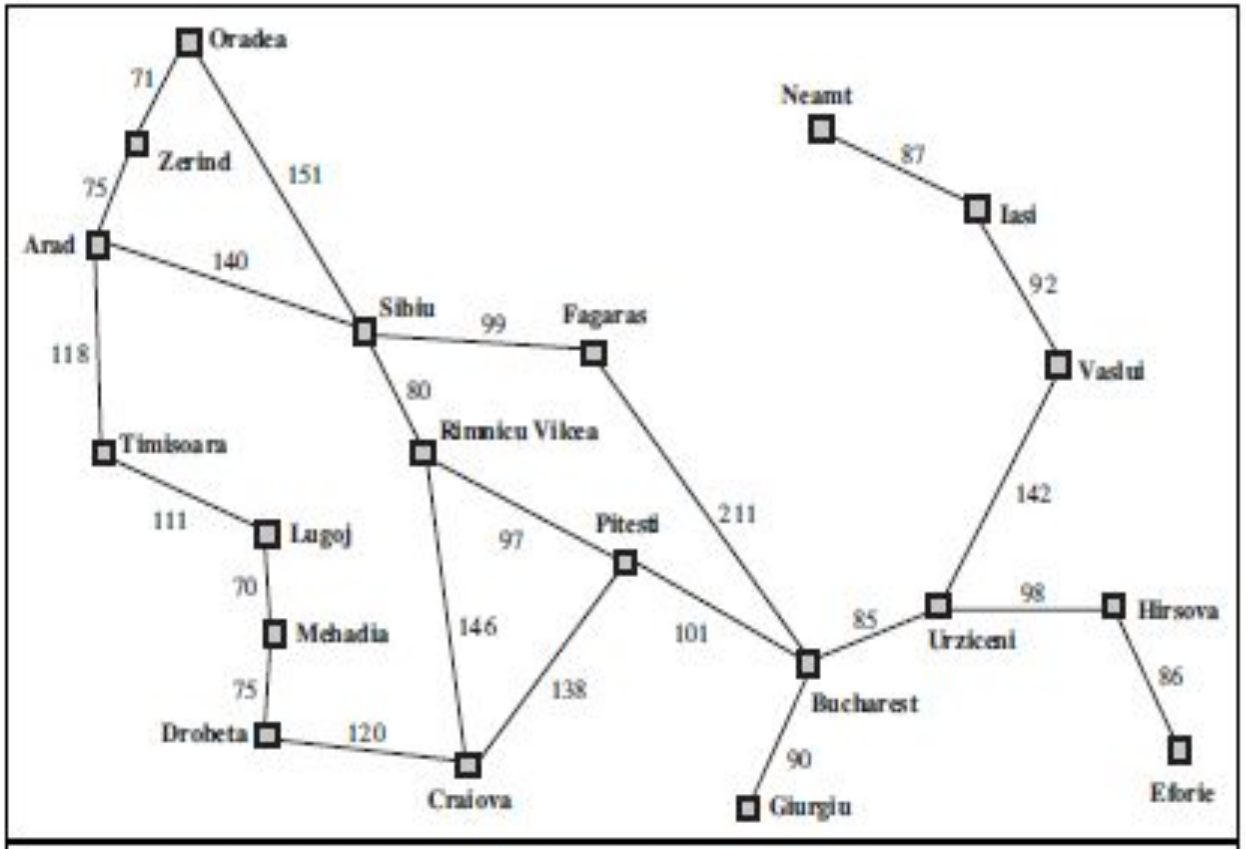


Figure 3.2 A simplified road map of part of Romania.

- The **goal test**, which determines whether a given state is a goal state. For example, in chess, the goal is to reach a state called “checkmate,”where the opponent’s king is under attack and can’t escape
- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.
- A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions

Formulating problems

- **Abstraction**
- the process to take out the irrelevant information
- leave the most essential parts to the description of the states (Remove detail from representation)
- Conclusion: Only the most important parts that are contributing to searching are used
- The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.
- Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

Properties of the Environment

properties of the environment

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

- **Example**:
- Romania On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest Formulate goal:
  - ◦ be in Bucharest

  Formulate problem:

  ◦ states: various cities

  ◦ actions: drive between cities

  Find solution:

  ◦ sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Single-state problem formulation

A problem is defined by four items:

initial state e.g., "at Arad"

1. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) =* {*<Arad □ Zerind, Zerind>, …* }
2. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
3. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

EXAMPLE PROBLEMS

- Toy Problem is intended to illustrate or exercise various problem solving methods. E.g., puzzle, chess, etc.
- A real-world problem is one whose solutions people actually care about. E.g., Design, planning, etc

- **<u>Toy problems</u>**

- The first example we examine is the **vacuum world**

• **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 × 2 ^2= 8 possible world states. A larger environment with n locations has n * 2n 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 *Suck*ing 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.

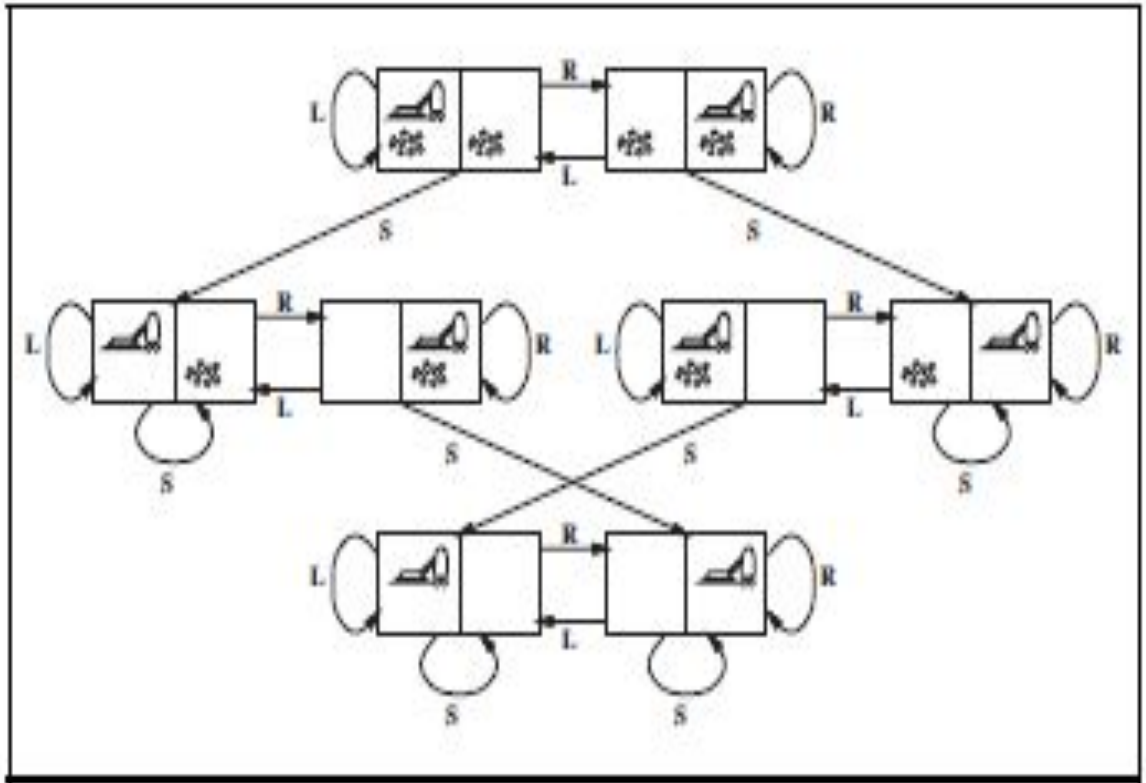


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8-puzzle

- A tile adjacent to the blank space can slide into the space.
- **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.
- **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
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

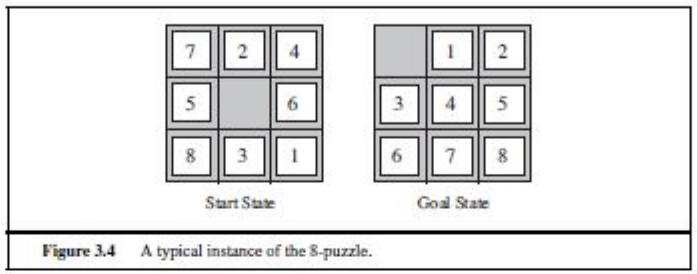


Figure 3.4 A typical instance of the 8-puzzle.

- The 8-puzzle belongs to the family of **sliding-block puzzles**, which are often used as test problems for new search algorithms in AI.
- 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 1025 states, and random instances take several hours to solve optimally

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. A queen attacks any piece in the same row, column or diagonal.

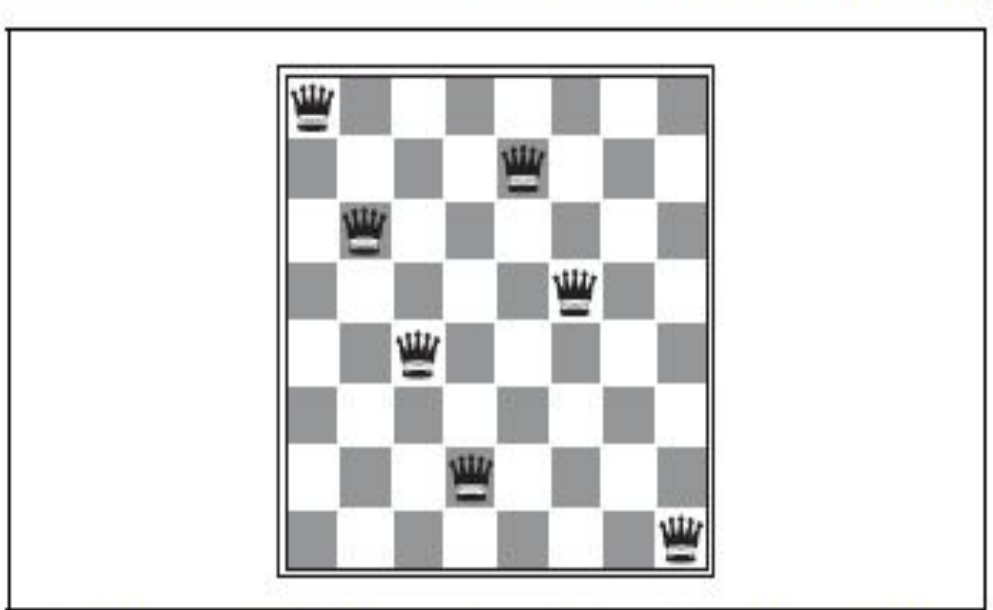


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

- There are two main kinds of formulation

1. An incremental formulation

- involves operators that augment the state description starting from an empty state
- Each action adds a queen to the state
- States: any arrangement of 0 to 8 queens on board
- Successor function: add a queen to any empty square

2. A complete-state formulation

- starts with all 8 queens on the board move the queens individually around
- States: any arrangement of 8 queens, one per column in the leftmost columns
- Operators: move an attacked queen to a row, not attacked by any other othe right formulation makes a big difference to the size of the search space

- The first incremental formulation

• **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 $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

• **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.

• **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

Real-world problems- Route-Finding problem

- Route-finding problems
- Touring problems
- Traveling Salesman problem
- VLSI layout problem
- Robot navigation
- Automatic assembly sequencing
- Internet searching

- **airline travel problems**

• **States**: Each state obviously includes a location (e.g., an airport) and the current time.Furthermore, because the cost of an action (a flight segment) may depend on previoussegments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.

• **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 needed.

• **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?

• **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.

- **Touring problems** are closely related to route-finding problems, but with an important difference.
- for example, the problem “Visit every city at least once, starting and ending in Bucharest.” As with route finding, the actions correspond to trips between adjacent cities.
- The state space, however, is quite different.
- Each state must include not just the current location but also the *set of cities the agent has visited*.
- So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest , Urziceni , Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

- The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once.
- The aim is to find the *shortest* tour.
- The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms

- A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**.
- In **cell layout**, the primitive components of the circuit are grouped into cells, each of which performs some recognized function.
- Each cell has a fixed footprint and requires a certain number of connections to each of the other cells.
- ◦The 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.

- **Robot navigation** is a generalization of the route-finding problem described earlier.
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states

- **Automatic assembly sequencing**
- Aim is to find an order in which to assemble the parts of some object.
- If the wrong order is chosen, there will be no way to add some part later 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.
- Thus, the generation of legal actions is the expensive part of assembly sequencing.
- Any practical algorithm must avoid exploring all but a tiny fraction of the state space.
- **protein design** is an automatic assembly problem in which 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.

SEARCHING FOR SOLUTIONS

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

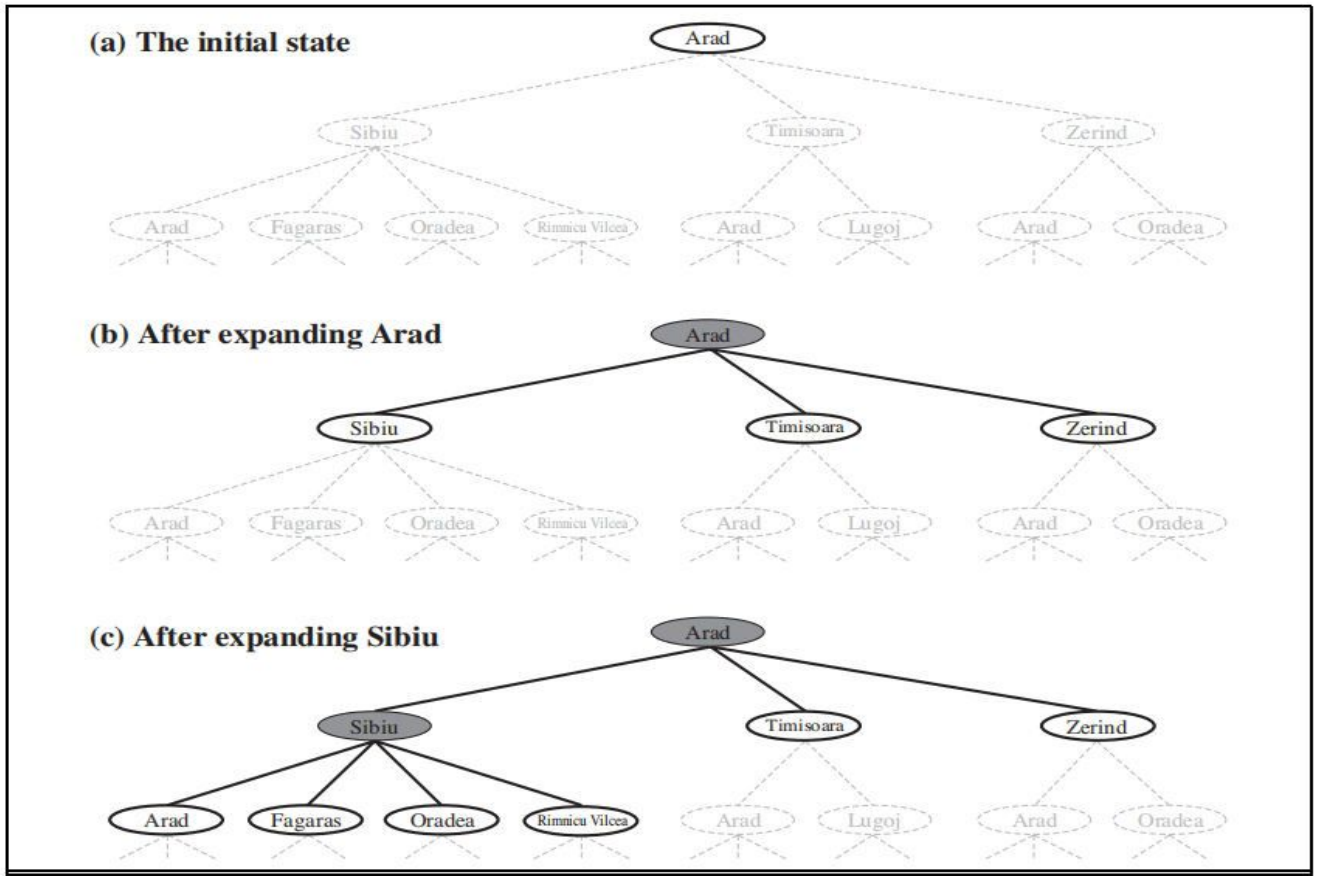


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Search tree for finding a route from Arad to Bucharest

- The root node of the tree corresponds to the initial state, *In(Arad)*.
- The first step is to test whether this is a goal state.
- Then we need to consider taking various actions. We do this by **expanding** the current state;
- applying each legal action to the current state, thereby **generating** a new set of states.
- In this case, we add three branches from the **parent node** *In(Arad)* leading to three new
- **child nodes**: *In(Sibiu), In(Timisoara),* and *In(Zerind)*.
- Now we must choose which of these three possibilities to consider further.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- **search strategy**: how they choose which state to expand next

- **loopy path:** path from Arad to Sibiu and back to Arad again! We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**
- Considering such loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable.
- there is no need to consider loopy paths.
- We can rely on more than intuition for this: because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.
- **redundant paths:** exist whenever there is more than one way to get from one state to another
- eg, the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).

- Frontier
- We reach a state when we identify a path from the start state to it. But, we say that we expanded it if we had followed all its outward edges and reached all its children.
- So, we can also think of a search as a sequence of expansions, and we first have to reach a state before expanding it.
- **Frontier is the reached but unexpanded states because we can expand only them**

- **a search strategy has two components**:
- rule(s) to decide whether or not to place the node in the frontier
- rule(s) to choose the next frontier node for expansion

- TREE-SEARCH algorithm
- with a data structure called the **explored set** (also known as the **closed list**),which remembers every expanded node.
- 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.

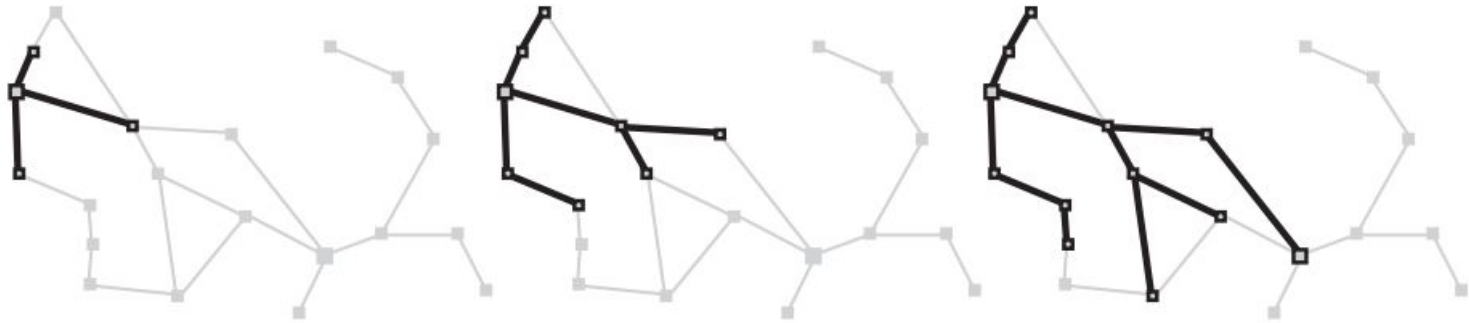


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

```
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
```

GRAPH-SEARCH algorithm

- Each state appears in the graph only once. But, it may appear in the tree multiple times contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph,
- Don’t add a node if its state has already been expanded or a node pointing to the same state is already in the frontier.
- so that every path from the initial state to an unexplored state has to pass through a state in the frontier.
- As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier,
- we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

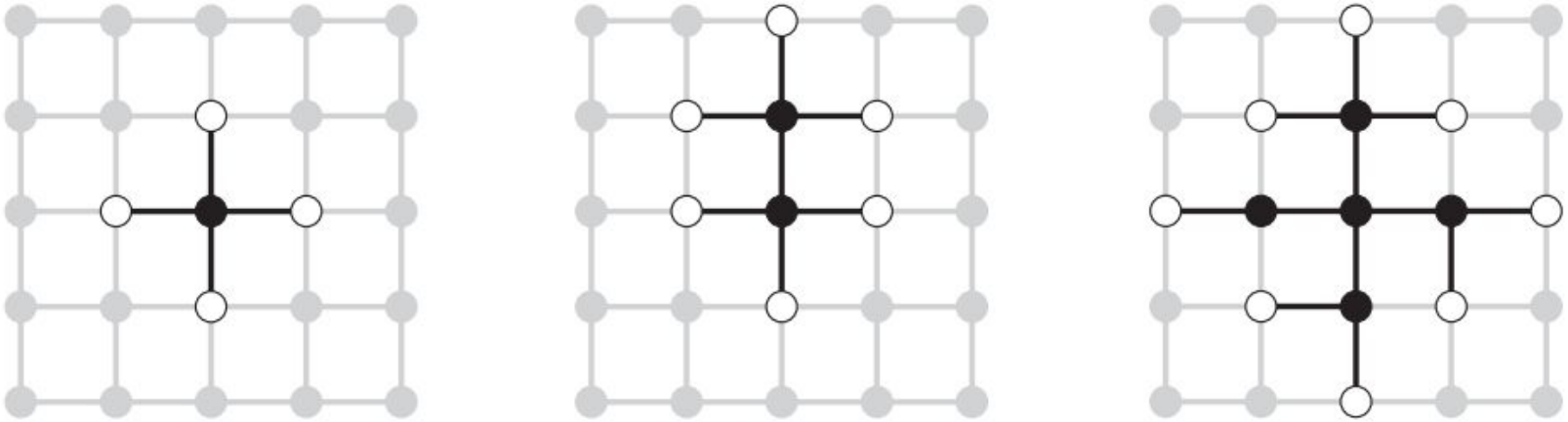


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

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 a rithms. The parts of GRAPH-SEARCH marked in bold italic are the additions neede handle repeated states.

Infrastructure for search algorithms

- 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 have a structure that contains four components:
- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

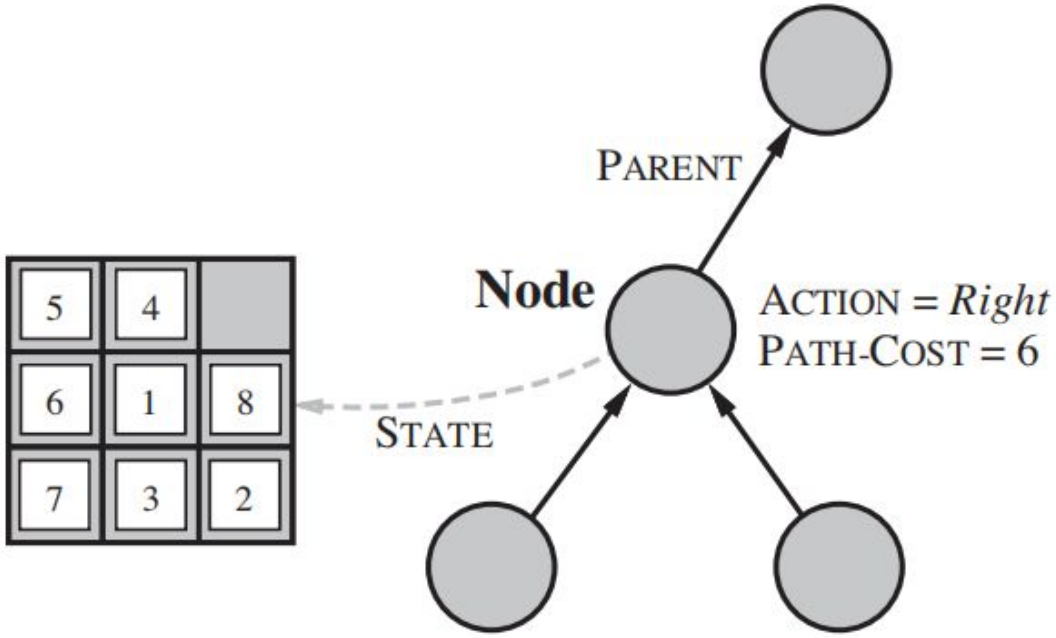


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

- PARENT pointers string the nodes together into a tree structure.
- These pointers also allow the solution path to be extracted when a goal node is found;
- we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.
- A node is a bookkeeping data structure used to represent the search tree.
- A state corresponds to a configuration of the world.
- nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
- Two different nodes can contain the same world state if that state is generated via two different search paths.

- Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting 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)
```

The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy

The appropriate data structure for this is a **queue**. The operations on a queue are as follows:

- EMPTY?(queue) returns true only if there are no more elements in the queue.
- POP(queue) removes the first element of the queue and returns it.
- INSERT(element, queue) inserts an element and returns the resulting queue

Three common variants of queue are

◦ first-in, first-out or **FIFO queue**, which pops the oldest element of the queue;

◦ last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue

◦ **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

- We can evaluate an algorithm's performance in four ways:
- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?
- Complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node;
- d, the **depth** of the shallowest goal node and
- m, the maximum length of any path in the state space.

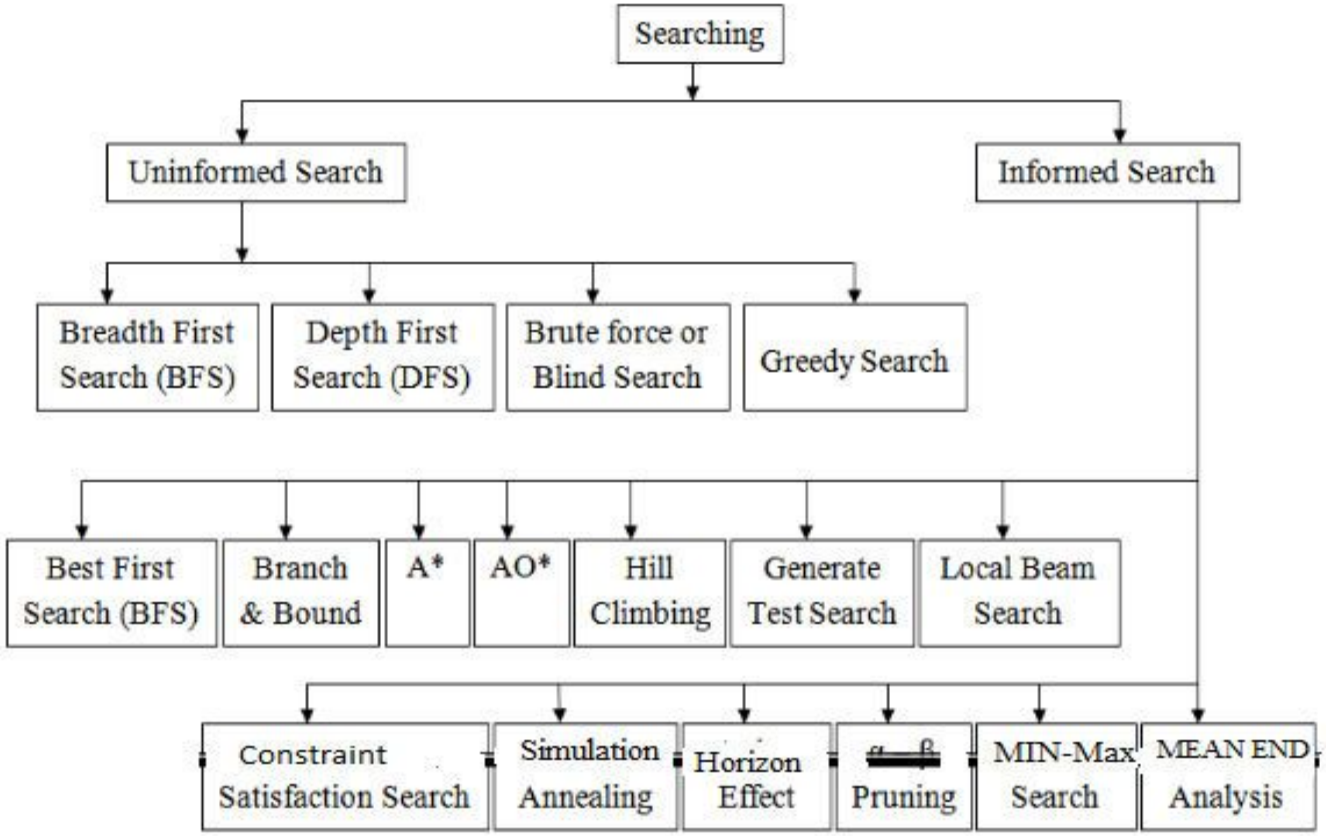
Time and space complexity are measured in terms of

b—maximum branching factor of the search tree

d—depth of the least-cost solution

m—maximum depth of the state space (may be ∞)

DIFFERENT TYPES OF SEARCHING



Uninformed Search Strategies

- 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.
- they that do not take into account the location of the goal. These algorithms ignore where they are going until they find a goal and report success.
- (also called **blind search**).
- All search strategies are distinguished by the *order* in which nodes are expanded.

1. Breadth-First search

- root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.
- all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- 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
- 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
- the goal test is applied to each node when it is *generated* rather than when it is selected for expansion
- breadth-first search always has the shallowest path to every node on the frontier

```
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.

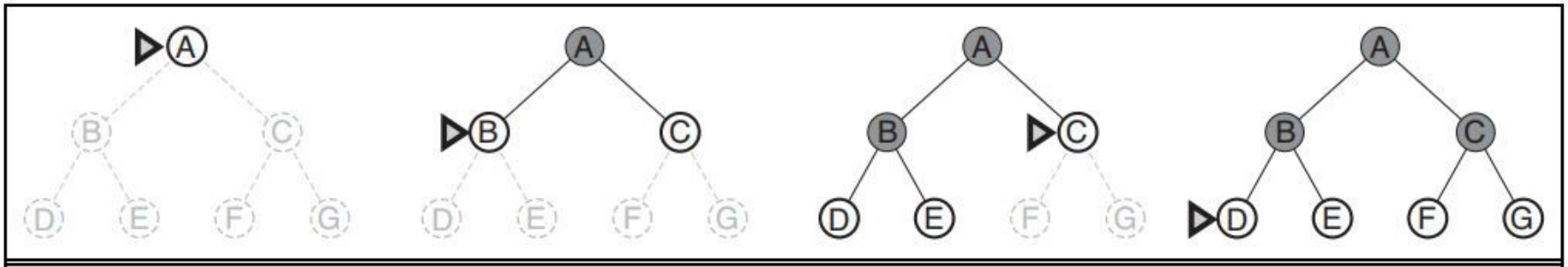


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Problem solving performance-BFS

complete—if the shallowest goal node is at some finite depth d, breadth-first search will

eventually find it after generating all shallower nodes

not optimal one: breadth-first search is optimal if the path cost is a non decreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

Time Complexity: The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of *these* generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

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

Space Complexity: every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier so the space complexity is O(bd), exponential complexity

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.

Summary

- *the memory requirements are a bigger problem for breadth-first search than is the execution time*
- One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take.
- If your problem has a solution at depth 16, then it will take about 350 years for breadth-first search to find it.
- In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

2. Uniform-cost search

- Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n). This is done by storing the frontier as a priority queue ordered by g.
- the goal test is applied to a node when it is *selected for expansion because* 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

```
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 gene graph search algorithm in Figure 3.7, except for the use of a priority queue and the additi of an extra check in case a shorter path to a frontier state is discovered. The data structure f *frontier* needs to support efficient membership testing, so it should combine the capabiliti of a priority queue and a hash table.

- problem is to get from Sibiu to Bucharest
- The successors of Sibiu are Rimnicu Vilcea and
- Fagaras, with costs 80 and 99, respectively.
- The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177.
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 =310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti
- for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

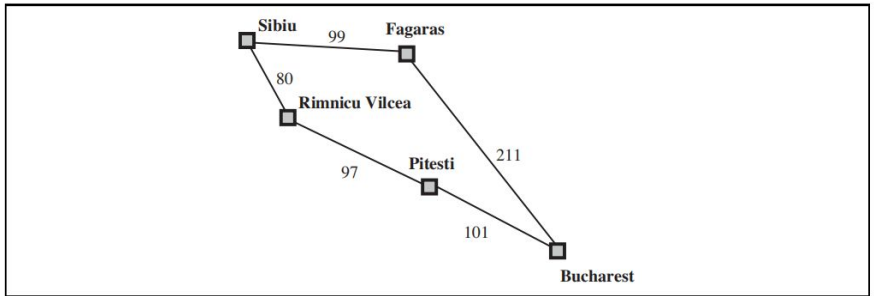


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Problem solving performance-UCS

- **Complete:** guaranteed provided the cost of every step exceeds some small positive constant ε
- **Optimal:** uniform-cost search is optimal in general. *uniform-cost search expands nodes in order of their optimal path cost.*
- **Time Complexity**: Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.The algorithm's worst-case time and space complexity is $O(b^{ceiling(C^*/\varepsilon))}$ where C^* is the cost of the optimal solution
- **Space Complexity**: # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon))}$
- When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost

- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.
- let C* be the cost of the optimal solution and that every action costs at least ε
- Then the algorithm's worst-case time and space complexity $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ is which can be much greater than bd.
- This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps.
- When all step costs are equal $b^{1+\lfloor C^*/\epsilon \rfloor}$ bd+1.
- When all step costs are the same, uniform-cost search is similar to breadth-first search, except that bfs stops as soon as it generates a goal, whereas **uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost**
- thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily

3.Depth-first search

- **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.
- Depth-first search uses a LIFO queue
- A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was was the deepest unexpanded node when it was selected.

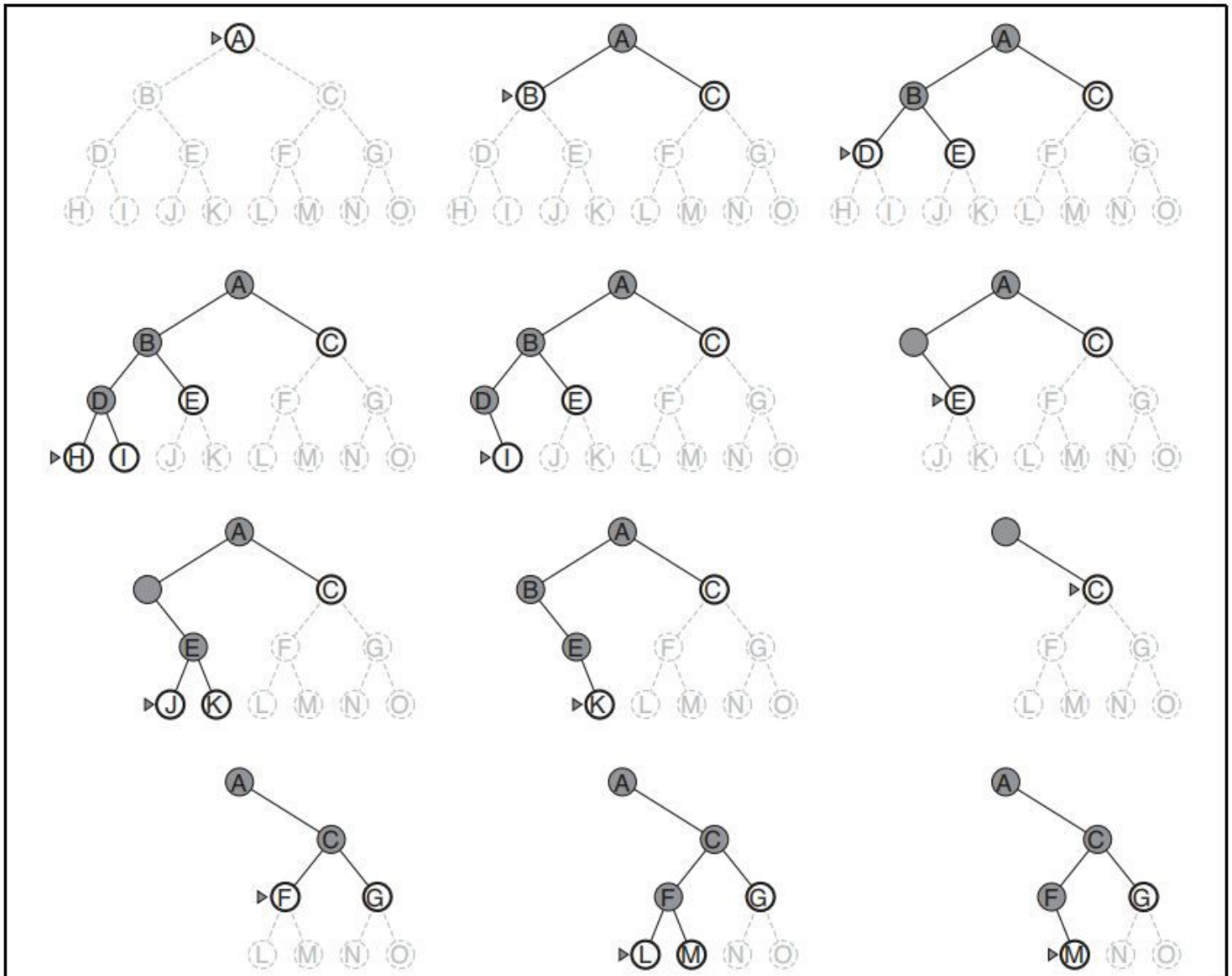


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- 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, on the other hand, is *not* complete.
- both versions are nonoptimal. Depthfirst search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.
- The time complexity of depth-first graph search is bounded by the size of the state space . A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node;
- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(bm) nodes.

- A variant of depth-first search called **backtracking search** uses still less memory**.**
- In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only O(m) memory is needed rather than O(bm).
- Backtracking search facilitates the idea of generating a successor by *modifying* the current state description directly rather than copying it first.
- This reduces the memory requirements to just one state description and O(m) actions.

4. **Depth-limited search**

- failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit . That is, nodes at depth are treated as if they have no successors. This approach is called **depth-limited search**.
- Depth-first search can be viewed as a special case of depth-limited search with l=∞.
- The depth limit solves the infinite-path problem.
- Depth-limited search will also be non optimal if we choose l >d.
- Its time complexity is O(bl) and its space complexity is O(bl).

```
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.

- 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 = 19 is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps.
- This number, known as the **diameter** DIAMETER of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.
- depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

5. **Iterative deepening depth-first 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.

```
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.

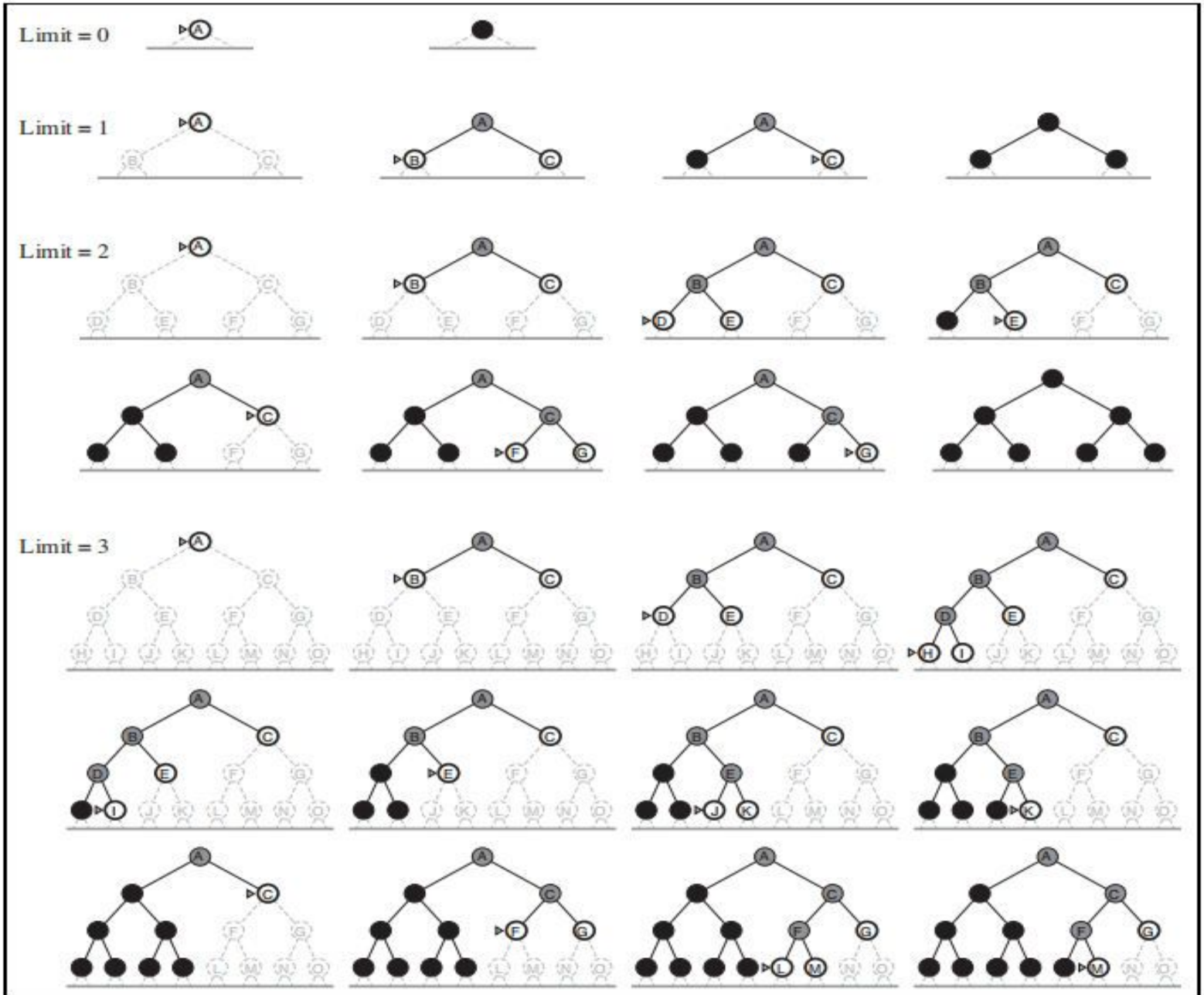


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

- Iterative deepening search may seem wasteful because states are generated multiple times.
- It turns out this is not too costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level,so it does not matter much that the upper levels are generated multiple times.
- In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times.

- So the total number of nodes generated in the worst case is
  - $N(IDS)=(d)b + (d-1)b^2 + \cdots + (1)b^d$,
- which gives a time complexity of $O(b^d)$ asymptotically the same as breadth-first search.
- Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer
- *iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known*

6. **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—hoping that the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than $b^{d.}$
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.
- the time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$. The space complexity is also $O(b^{d/2})$.
- space requirement is the most significant weakness of bidirectional search.

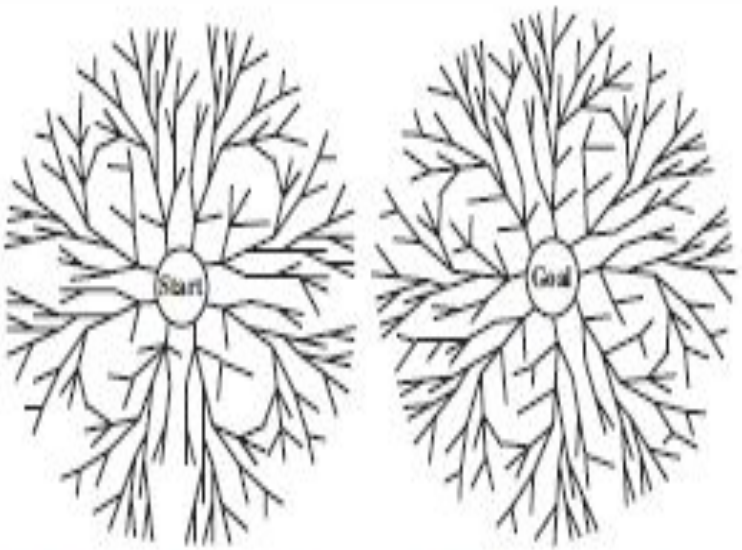


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

The reduction in time complexity makes bidirectional search attractive. Bidirectional search requires a method for computing predecessors.

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.

INFORMED (HEURISTIC) SEARCH STRATEGIES

- an **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.

1. BEST-FIRST SEARCH

- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n).
- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search except for the use of *f* instead of *g* to order the priority queue
- The choice of *f* determines the search strategy.
- Most best-first algorithms include as a component of *f* a **heuristic function**, denoted h(n):
- h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.
- if n is a goal node, then h(n)=0.
- For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

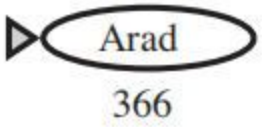
2. **Greedy best-first search**

- **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n).
- for route-finding problems in Romania; we use the **line distance** heuristic, which we will call ^{h}SLD. If the goal is Bucharest, we need to know the straight-line distances to Bucharest,

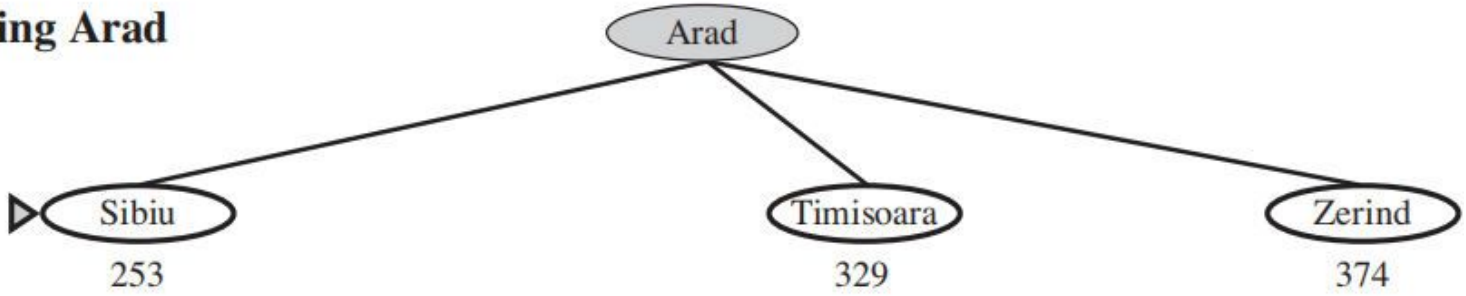
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

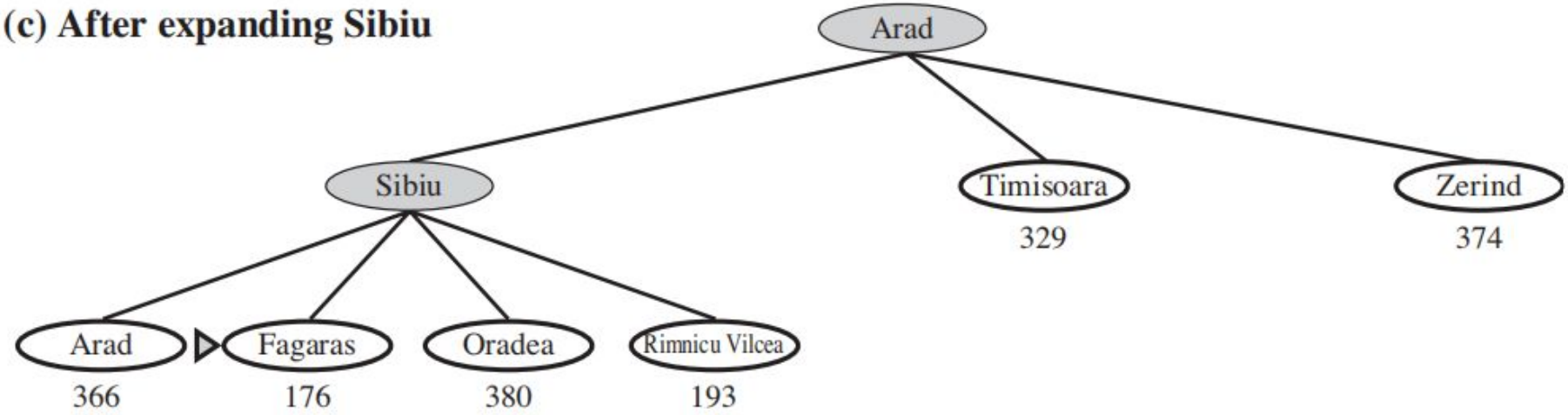
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



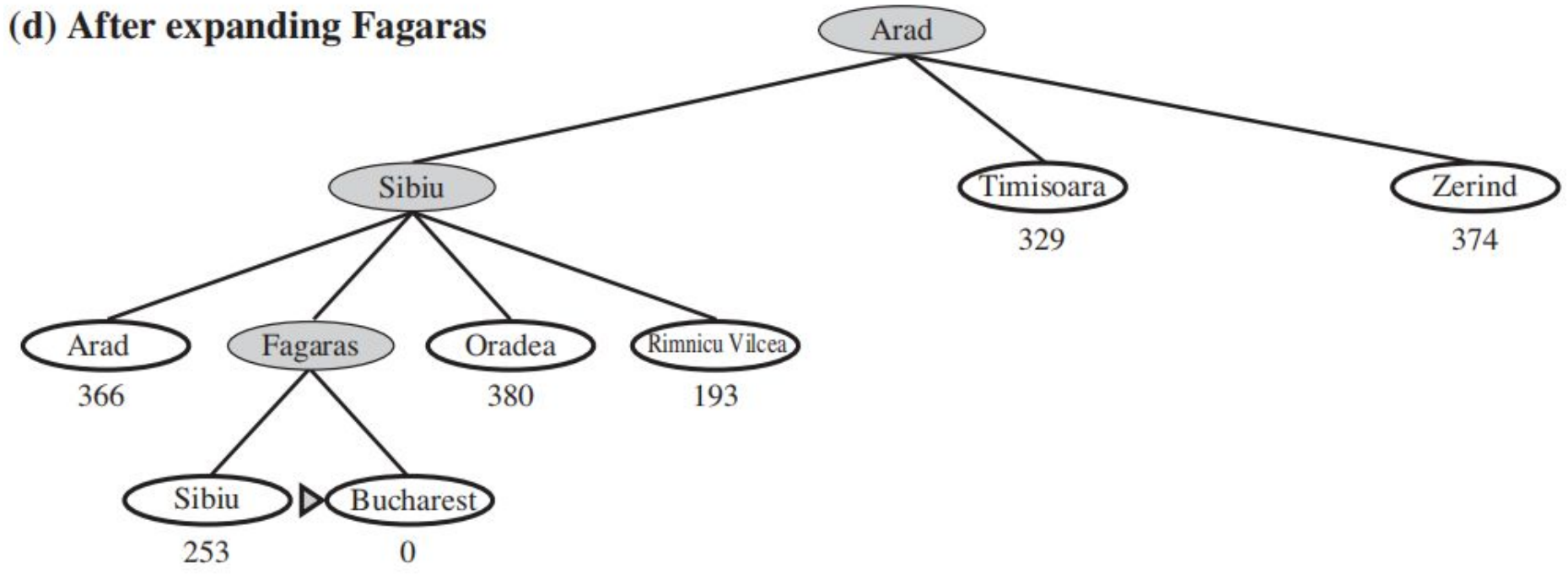


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* search: Minimizing the total estimated solution cost

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

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

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

- if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n).
- It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions,
- **A* search is both complete and optimal.**
- The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses g + h instead of g.

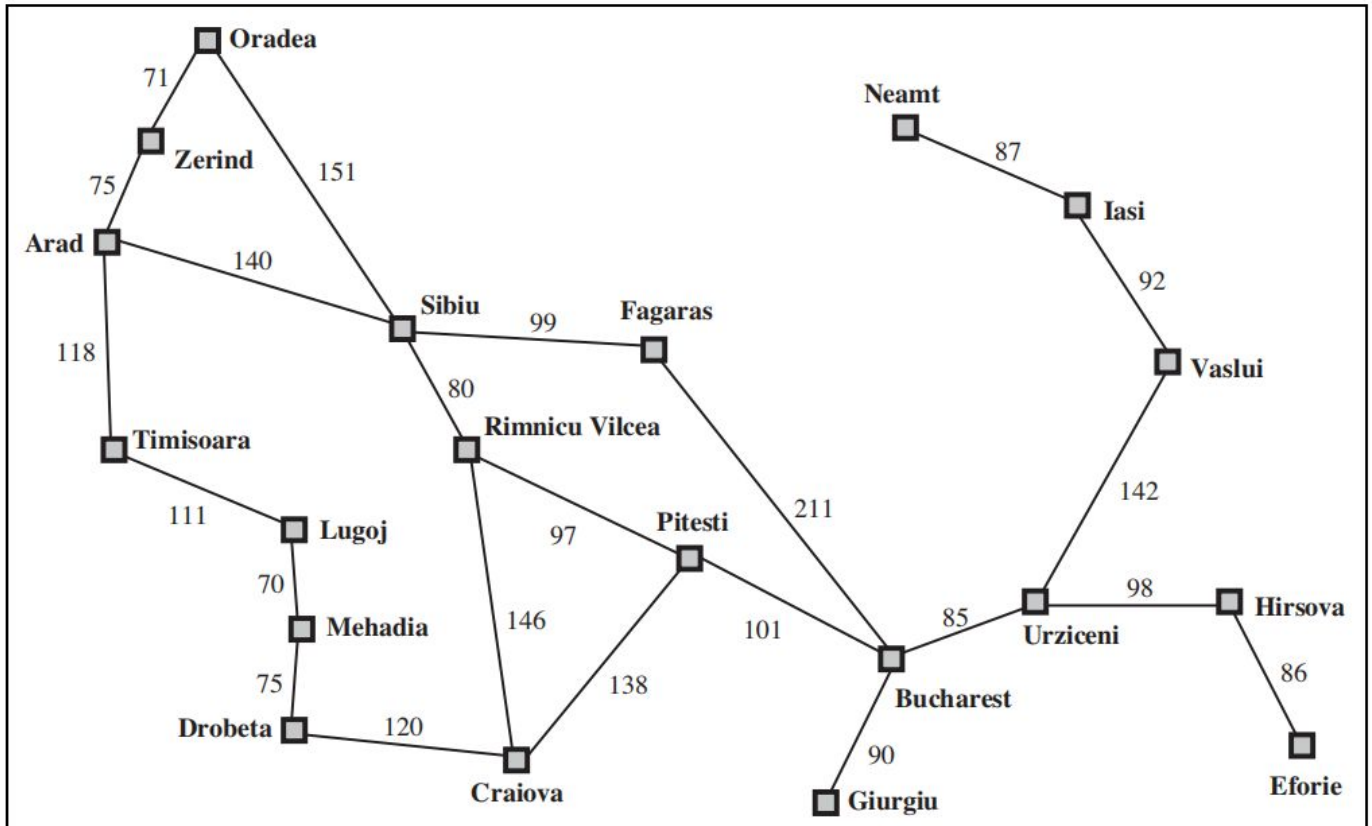
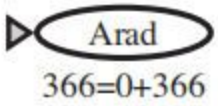


Figure 3.2 A simplified road map of part of Romania.

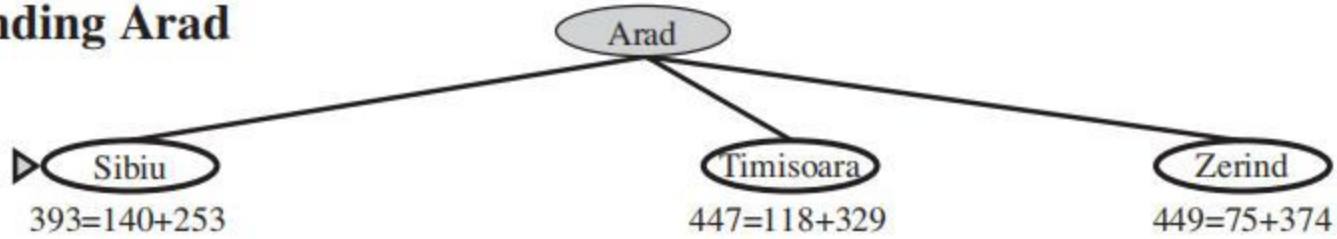
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

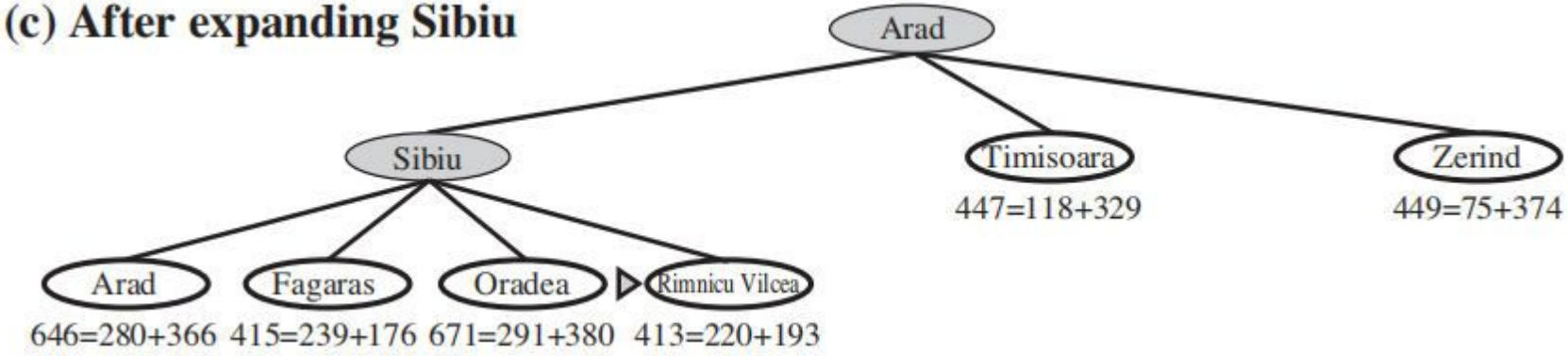
(a) The initial state

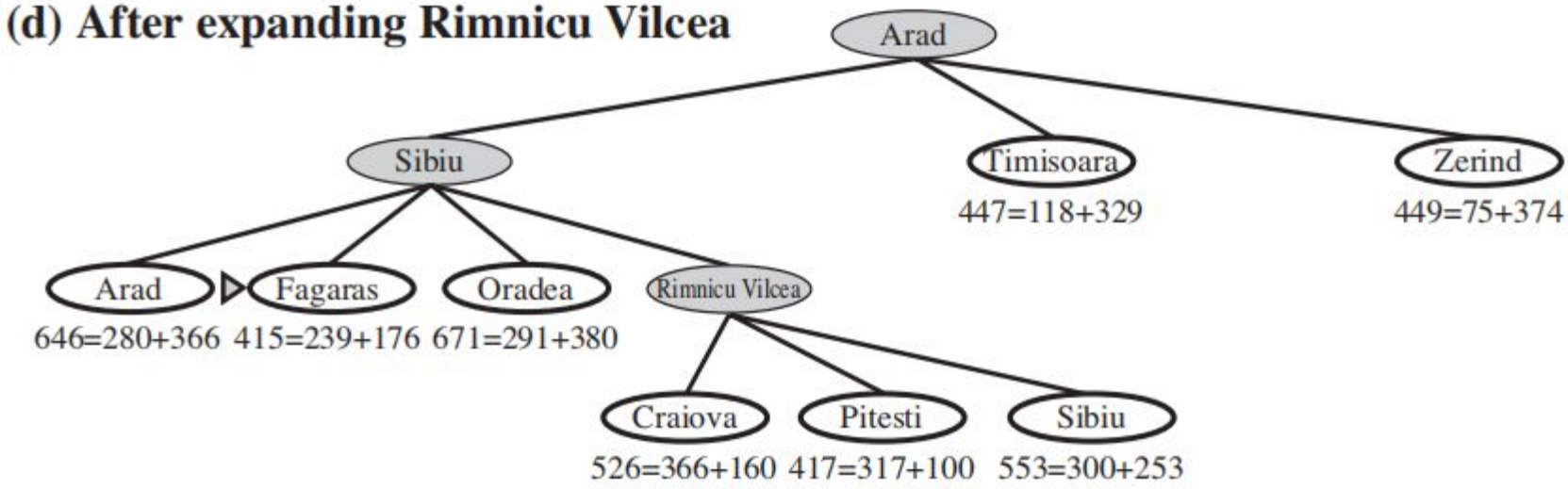


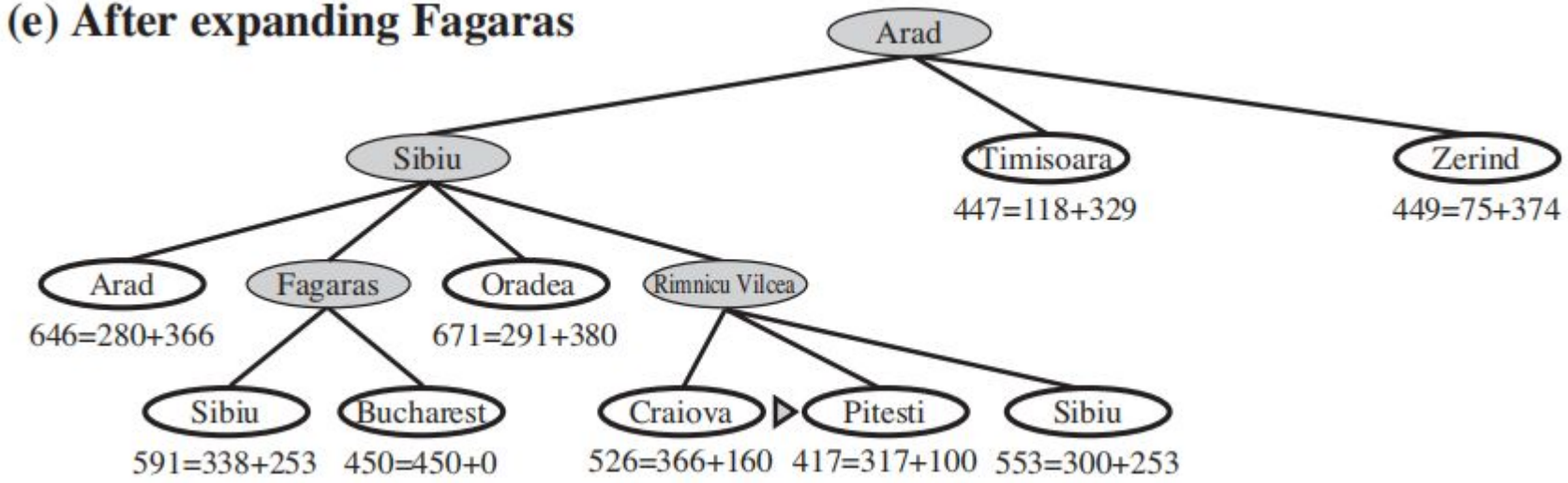
(b) After expanding Arad



(c) After expanding Sibiu







Bucharest first appears on the frontier here but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417).
Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450.

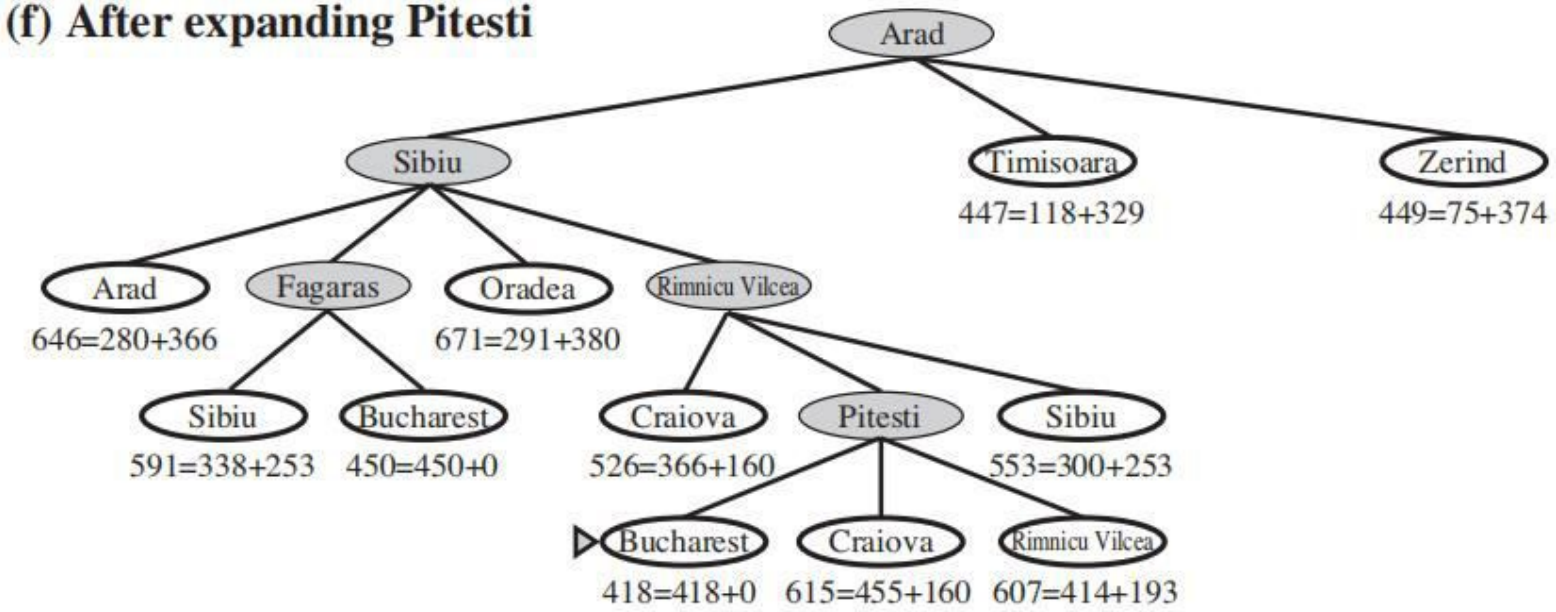


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Conditions for optimality: Admissibility

- The first condition we require for optimality is that h(n) be an **admissible heuristic**.
- An admissible heuristic is one that *never overestimates* the cost to reach the goal.
- g(n) is the actual cost to reach n along the current path, and
- f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is
- ◦ Eg, straight-line distance hSLD that we used in getting to Bucharest is n admissible heuristic
- ◦ because the shortest path between any two points is a straight line
- ◦ Straight line cannot be an overestimate

Conditions for optimality: Consistency

- It is required only for applications of A* to graph search
- A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n :

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

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

Optimality of A*

- *the tree-search version of* A∗ *is optimal if* h(n) *is admissible, while the graph-search version is optimal if* h(n) *is consistent.*
- A∗ expands no nodes with f(n) > C∗—for example, Timisoara is not expanded in even though it is a child of the root
- the subtree below Timisoara is **pruned**; because hSLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality
- Pruning eliminates possibilities from consideration without having to examine them
- A∗ is **optimally efficient** for any given consistent heuristic.
- ◦ That is, no other optimal algorithm is guaranteed to expand fewer nodes than A∗
- ◦ This is because any algorithm that *does not* expand all nodes with f(n) < C∗ runs the risk of missing the optimal solution.

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

- For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the the **absolute error** or the **relative error** of the heuristic.
- The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as $\equiv (h^* - h)/h^*$.
- In this case, the time complexity of A* is exponential in the maximum absolute error, that is,$O(b^{\Delta})$.
- 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.

Memory-bounded heuristic search

- The simplest way to reduce memory requirements for A* is to adapt the idea of iterative
- **iterative-deepening A*** (IDA*) algorithm
- deepening to the heuristic search context, resulting in the **iterative-deepening A*** (IDA*) algorithm. 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 smallestf-cost of any node that exceeded the cutoff on the previous iteration.
- **Recursive best-first search** (RBFS)
- **Recursive best-first search** (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.

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

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

Figure 3.26 The algorithm for recursive best-first search.

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

  reexpanding the subtree at some later time.

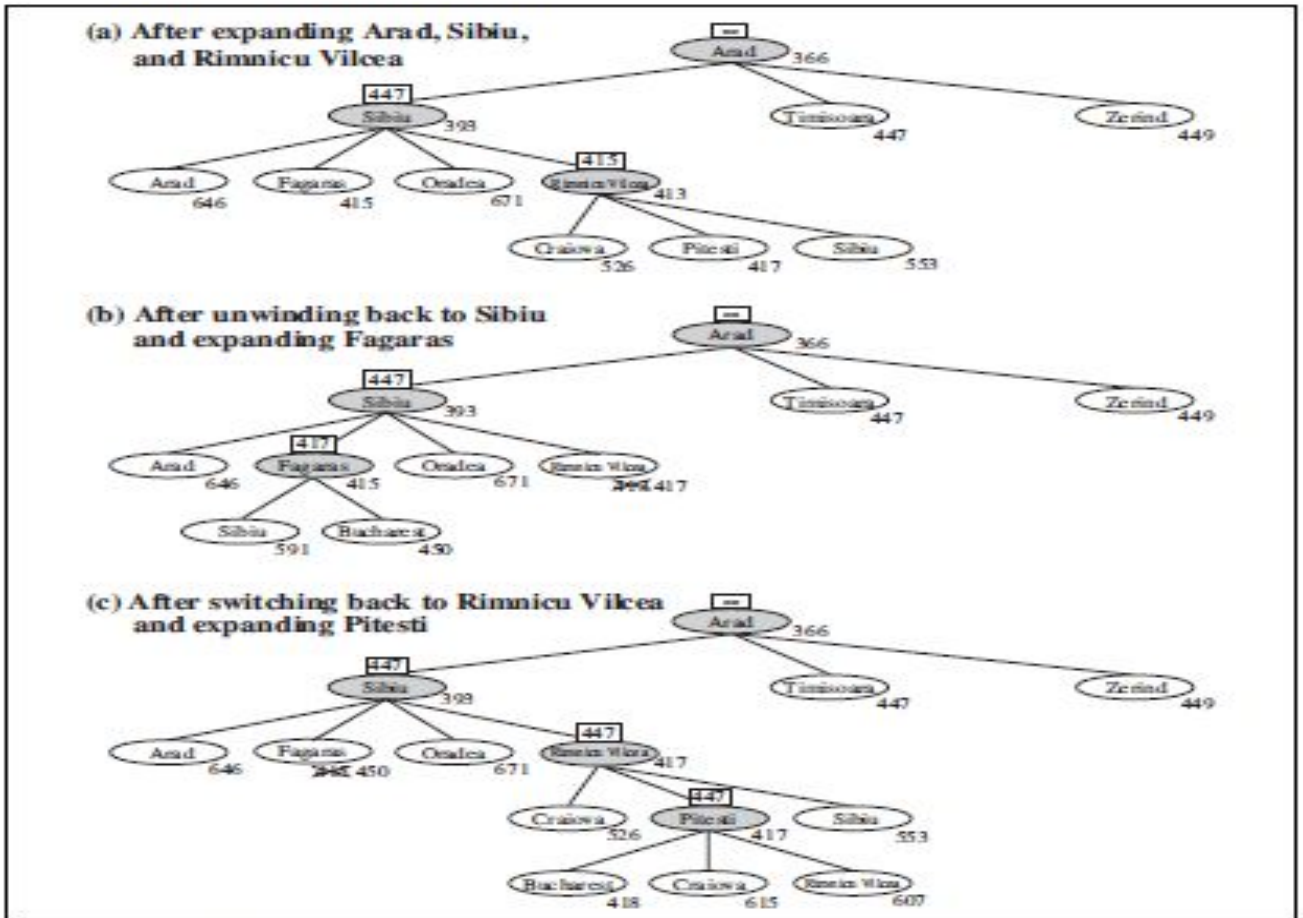


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse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

- RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Its space complexity is linear in the depth of the deepest optimal solution.
- Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory,but it uses only linear space.
- Two algorithms to use all available memory are **MA*** (memory-bounded A*) and MA* **SMA*** (simplified MA*).
- **SMA***
- SMA* proceeds like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f-value.

- SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes).
- It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution.
- SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, step costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the explored set.
- **Learning to search better**
- an important concept called the **metalevel state space is used by an agent to search better**. Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an **object-level state space.**
- The goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost. **Metalevel learning** algorithm can learn from these experiences to avoid exploring unpromising subtrees.

HEURISTIC FUNCTIONS

- In an 8-puzzle problem two commonly used candidates:
- • h1 = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
- • h2 = 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**. h2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of
- h2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18 .
- As expected, neither of these overestimates the true solution cost, which is 26.

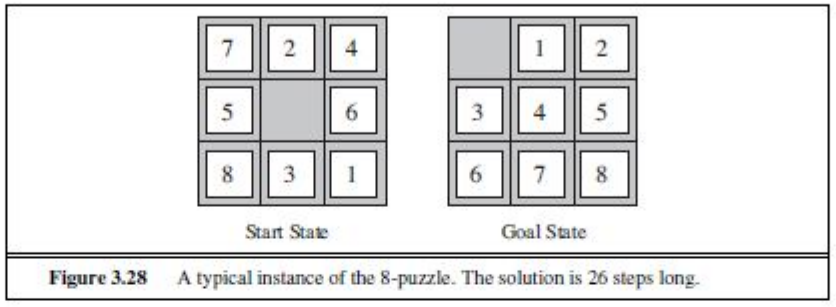


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

- The performance of heuristic search algorithms depends on the quality of the heuristic function.
- One can construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for subproblems in a pattern database, or by learning from experience with the problem class

The effect of heuristic accuracy on performance

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

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

- h2 is *always* better than h1. It is easy to see from the definitions of the two heuristics that, for any node n, h2(n) ≥ h1(n).
- h2 **dominates** h1. Domination translates directly into efficiency: A∗ using h2 will never expand more nodes than A∗ using h1.
- it is generally better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long.

- **Generating admissible heuristics from relaxed problems**

- If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square,then h1 would give the exact number of steps in the shortest solution.
- Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution.
- A problem with fewer restrictions on the actions is called **a relaxed problem**.
- The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph.

- *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.*
- It must obey the triangle inequality and is therefore **consistent**
- If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically. For example, if the 8-puzzle actions are described as
- **A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank**,
- we can generate three relaxed problems by removing one or both of the conditions:
- (a) A tile can move from square A to square B if A is adjacent to B.
- (b) A tile can move from square A to square B if B is blank.
- (c) A tile can move from square A to square B.
- From (a), we can derive h2 (Manhattan distance). The reasoning is that h2 would be the proper score if we moved each tile in turn to its destination. From (c), we can derive h1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step

- If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.
- If a collection of admissible heuristics h1 . . .hm is available for a problem. We can have the best of all worlds, by defining

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

Generating admissible heuristics from subproblems: Pattern databases

- Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem.
- The idea behind **pattern databases** is to store these exact solution costs for every possible subproblem instance—in our example, every possible configuration of the four tiles and the blank.
- Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database

- Disjoint pattern databases work for sliding-tile puzzles because the problem can be divided up in such a way that each move affects only one subproblem—because only one tile is moved at a time.
- The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8,for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value. A combined heuristic of this kind is much more accurate

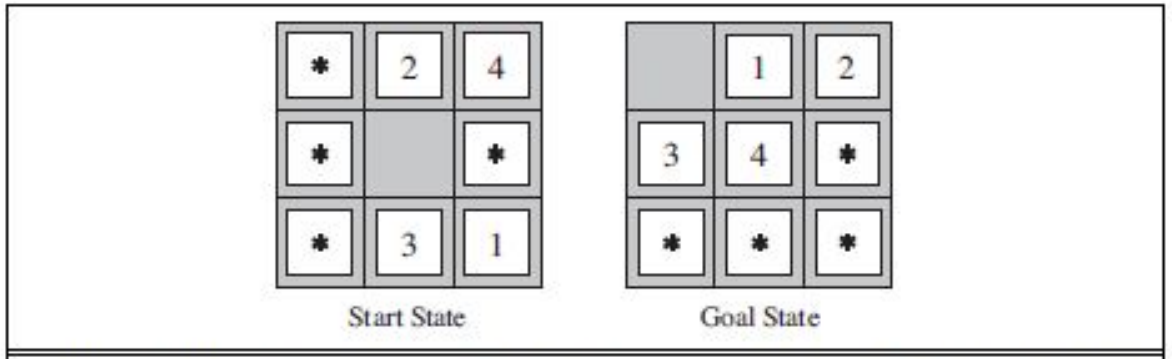


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- **Learning heuristics from experience**
- A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n.
- Another solution is to learn from experience. “Experience” here means 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. From these examples, a learning algorithm can be used to construct a function h(n) that can predict solution costs for other states that arise during search.
- Inductive learning methods work best when supplied with **features** of a state that are relevant to predicting the state’s value, rather than with just the raw state description. For example, the feature “number of misplaced tiles” might be helpful in predicting the actual distance of a state from the goal. we can use several features.
- A common approach is to use a linear combination:

$$h(n) = c1x1(n) + c2x2(n) .$$

- The constants c1 and c2 are adjusted to give the best fit to the actual data on solution costs.