Unit 1 chapter 3

Problem-solving: Problem-solving agents, Example problems, Searching for Solutions, Uninformed Search Strategies: Breadth First search, Depth First Search

Problem-solving agents

- A type of intelligent agent designed to address and solve complex problems or tasks in its environment

Problem Statement

- Imagine an agent in the city of Arad, Romania, enjoying a touring holiday.
- The agent's performance : it wants to improve its suntan, improve its omanian, take in the sights, enjoy the nightlife, avoid hangovers, and so on.
- agent has a nonrefundable ticket to fly out of Bucharest the following day.
- 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.
- The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.

- Three roads lead out of Arad, one toward Sibiu, one to Timisoara, and one to Zerind.
- Agent should be familiar with the geography of Romania.

- Consider the agent has a map of Romania.
- map - provide the agent with information about the states it might get itself into and the actions it can take.
- environment is **observable(**agent always knows the current state)**,** environment is **discrete**(sign indicating its presence to arriving drivers)

Problem solving agents

- Goal formulation: Based on current situation and the performance measure.
- Problem formulation is the process of deciding what actions and states to consider ,given a goal.

Problem solving agents

- A **search algorithm** - problem as input , solution in the form of action sequence.
- Once a solution is found, the actions it recommends can be carried out➔ E**xecution** phase.

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

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

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

Well-defined problems and solutions

- **problem** can be defined formally by five components
- **Initial state ,actions, transaction model, goal test, path cost**

❑**initial state :**agent starts in. initial state :Romania as In(Arad).

❑ **actions** :

- 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.
- Each of these actions is **applicable** in s.
- For example, from the state In(Arad), the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

❑ transition model

- A description of what each action does;
- function RESULT(s, a) that returns the state that results from doing action **a** in state **s**.
- **successor** refer to any state reachable from a given state by a single action.
- Example, RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

- initial state, actions, and transition model implicitly define the **state space** of the problem
- **State Space:** 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.

- **goal test**,
  - determines whether a given state is a goal state.
  - Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. {In(Bucharest)}.
- **path cost**
  - Numeric cost to each path.
  - sum of the costs of the individual actions along the path.
  - The step cost c(s, a, s). assume that step costs are nonnegative

Example problem :Romania tour

- On holiday in Romania .Currently in Arad
- Flight leaves tomorrow at Bucharest
- Formulate the goal
  - Be in Bucharest
- Formulate the problem
  - State: various cities
  - Actions: Drive between cities
- Find Solution
  - Sequence of cities eg. Arad, Sibiu, Fagaras, Bucharest

- **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
- **optimal solution** has the lowest path cost among all solutions

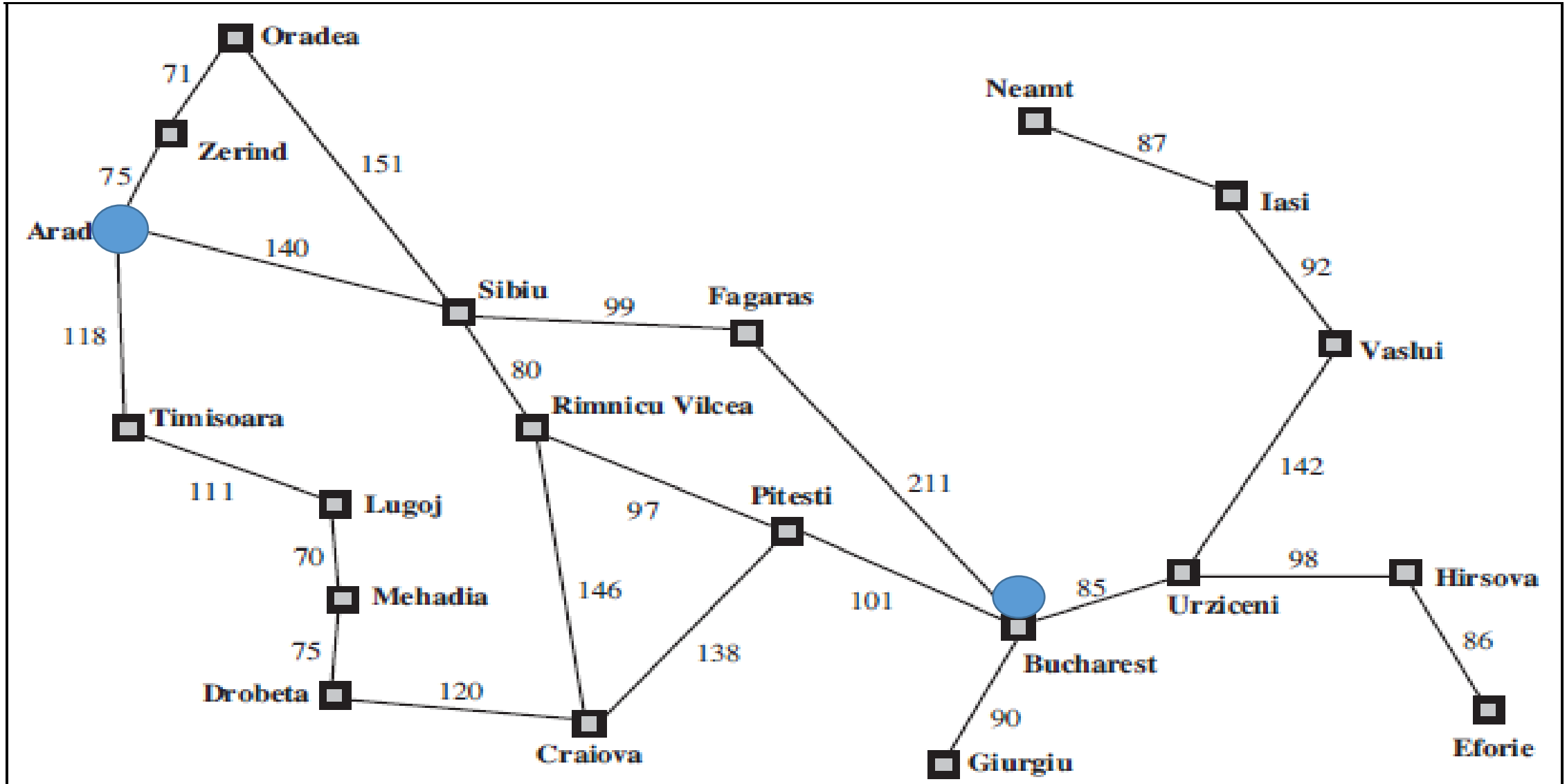


Figure 3.2 A simplified road map of part of Romania.

EXAMPLE PROBLEMS

- **toy problem :**illustrate or exercise problem-solving methods
- A **real-world problem** is one whose solutions people actually care about.

1.toy problem

1. States: Dirt and robot location
2. Actions: left ,right, suck
3. Goal test: no dirt at all locations
4. Path cost :1 per action

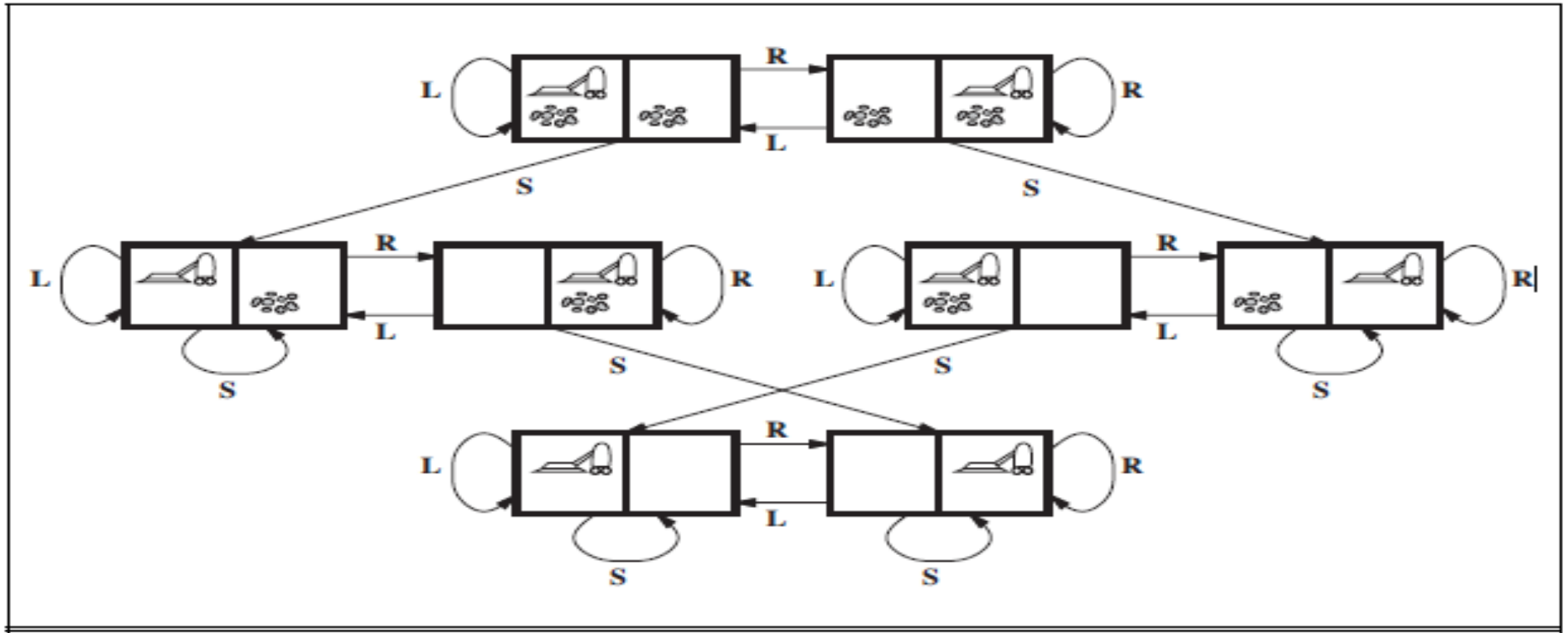


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Toy problems

1. **States:** Dirt and robot location
2. **Initial state**: Any state can be designated as the initial state.
3. **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
4. **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.
5. **Goal test**: This checks whether all the squares are clean
6. **Path cost** :Each step costs 1, so the path cost is the number of steps in the path.

8-puzzle

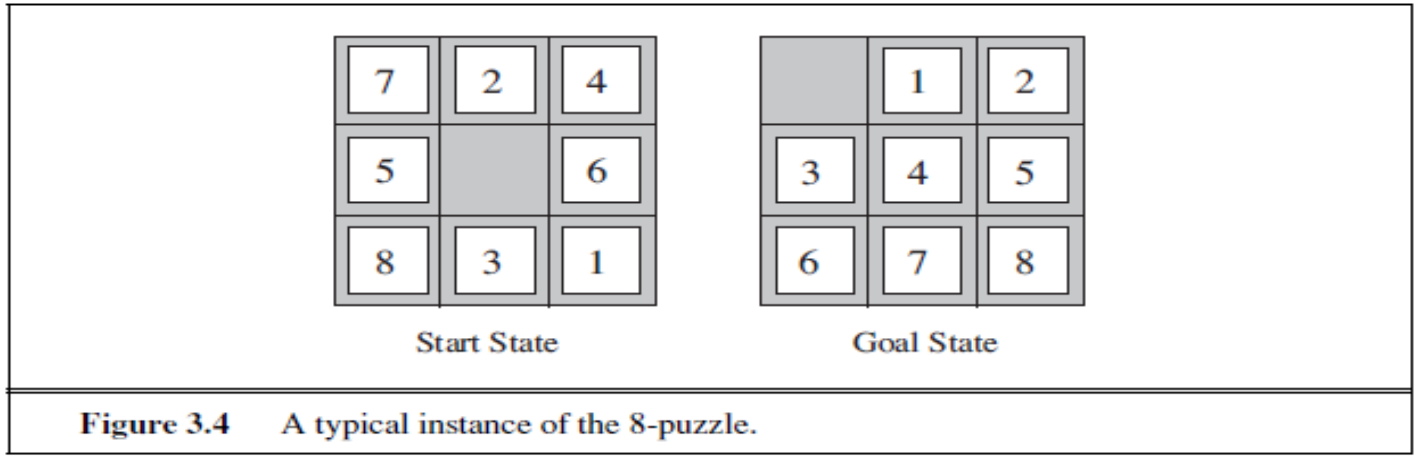


Figure 3.4 A typical instance of the 8-puzzle.

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**: L*eft*, *Right*, *Up*, or *Down*.
- **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 (Other goal configurations are possible.)
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path

8-puzzle

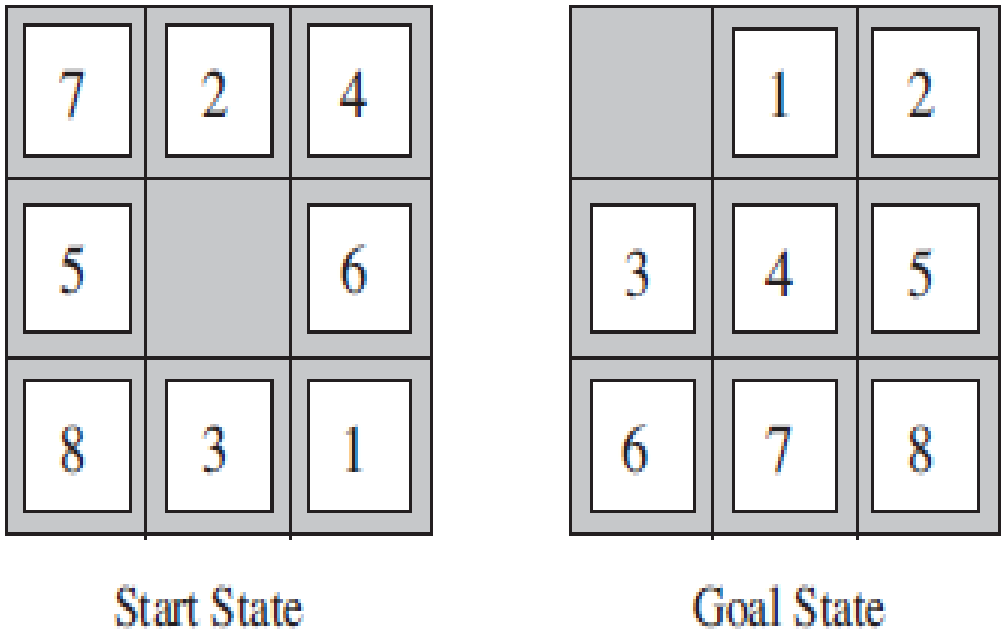
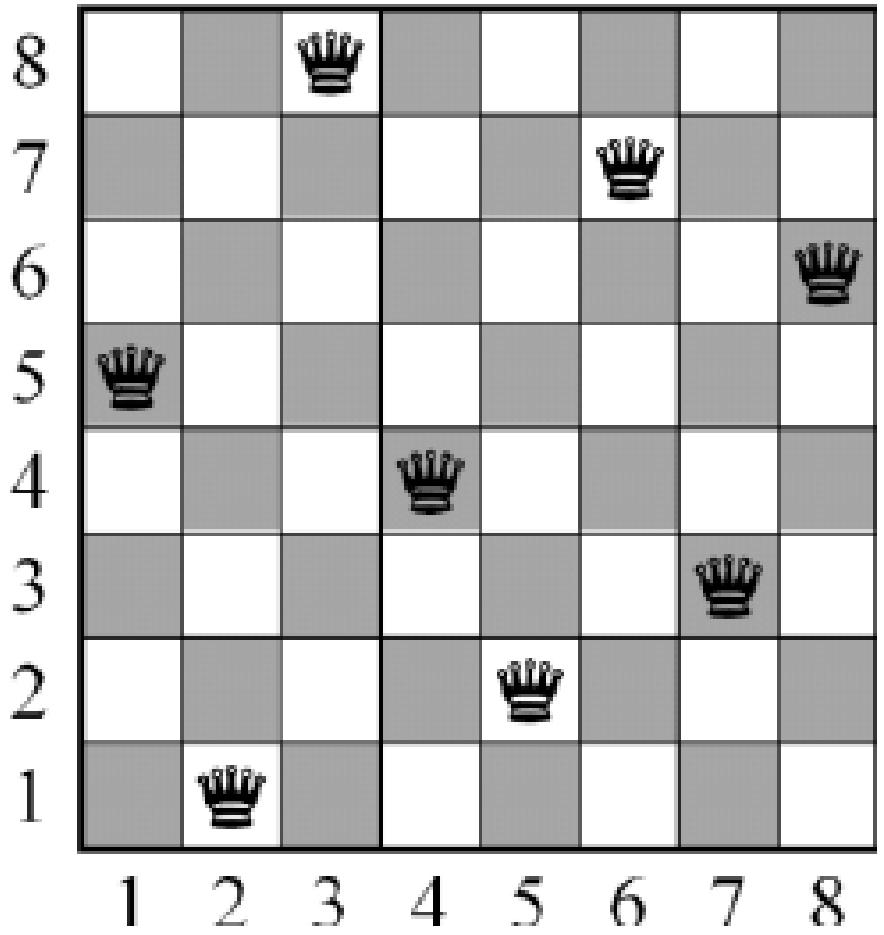
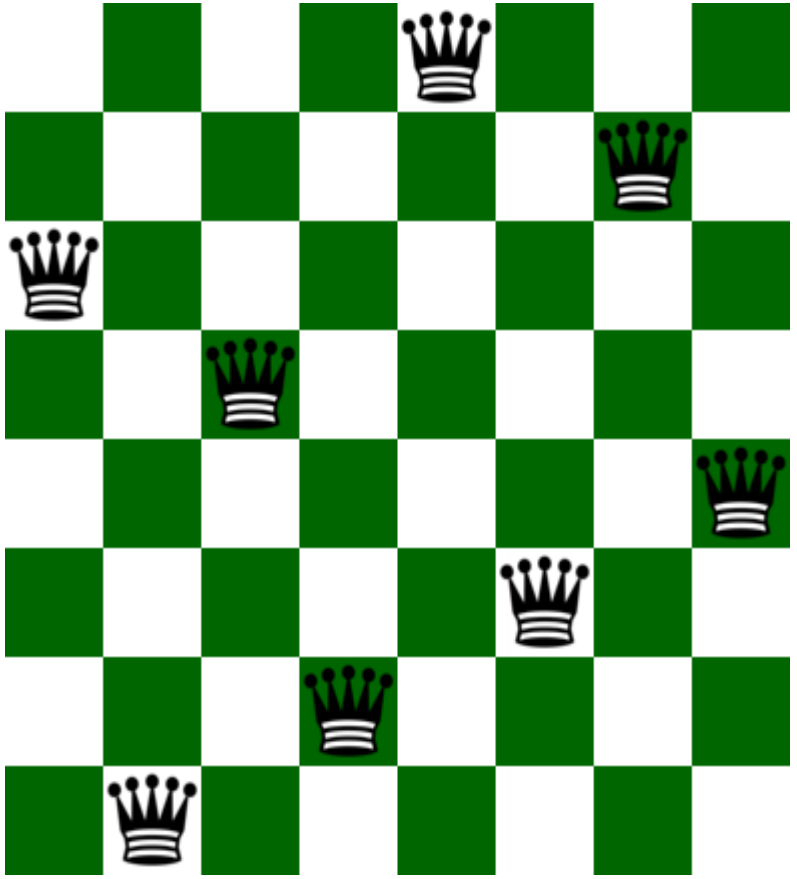


Figure 3.4 A typical instance of the 8-puzzle.

8-queens problem

- **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.


8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8



Real-world problems

1. **Route finding problems**
- GPS based navigation systems GMAP

2. **Touring Problems**
- **TSP**

3.VLSI Layout Problem: millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances. cell layout and channel routing

4. **Robot Navigation Problem**

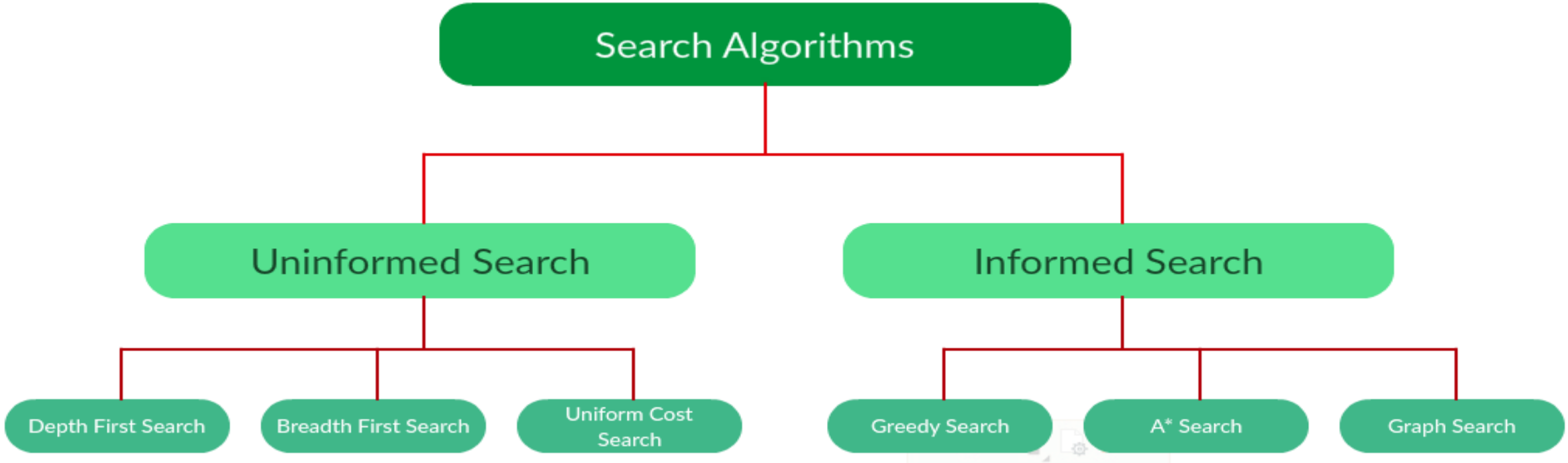
5.Automatic assembly sequencing :

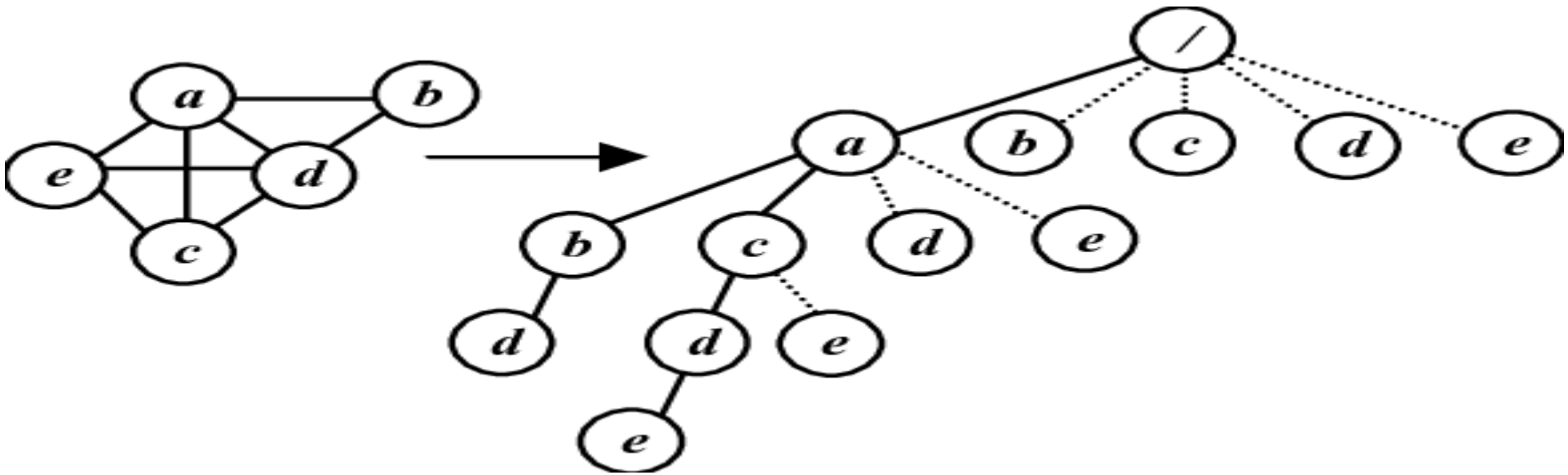
6.Internet Searching

7.Searching paths in metabolic networks in bioinformatics: protein design, 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 .
- a route from **Arad to Bucharest**.
- The root node of the tree initial state, *In(Arad)*.
- The first step is to test whether this is a goal state.
- Do this by **expanding** the current state; thereby **generating** a new set of states


Search Algorithms
Uninformed Search
Informed Search
Depth First Search
Breadth First Search
Uniform Cost Search
Greedy Search
A* Search
Graph Search


a
b
e
d
c
/
a
b
c
d
e
b
c
d
e
d
d
e
e

- Add three branches from the **parent node** *In(Arad)* leading to three new **child nodes**: *In(Sibiu), In(Timisoara),* and *In(Zerind).*
- Suppose we choose Sibiu first.
- Whether it is a goal state (it is not) and then expand it to get *In(Arad)*, *In(Fagaras)*, *In(Oradea)*, and *In(RimnicuVilcea)*.
- Choose any of these four or go back and choose Timisoara or Zerind.
- Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier**.

- Loopy paths are a special case **redundant paths** whenever there is more than one way to get from one state to another.
- Consider the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).
- second path is redundant

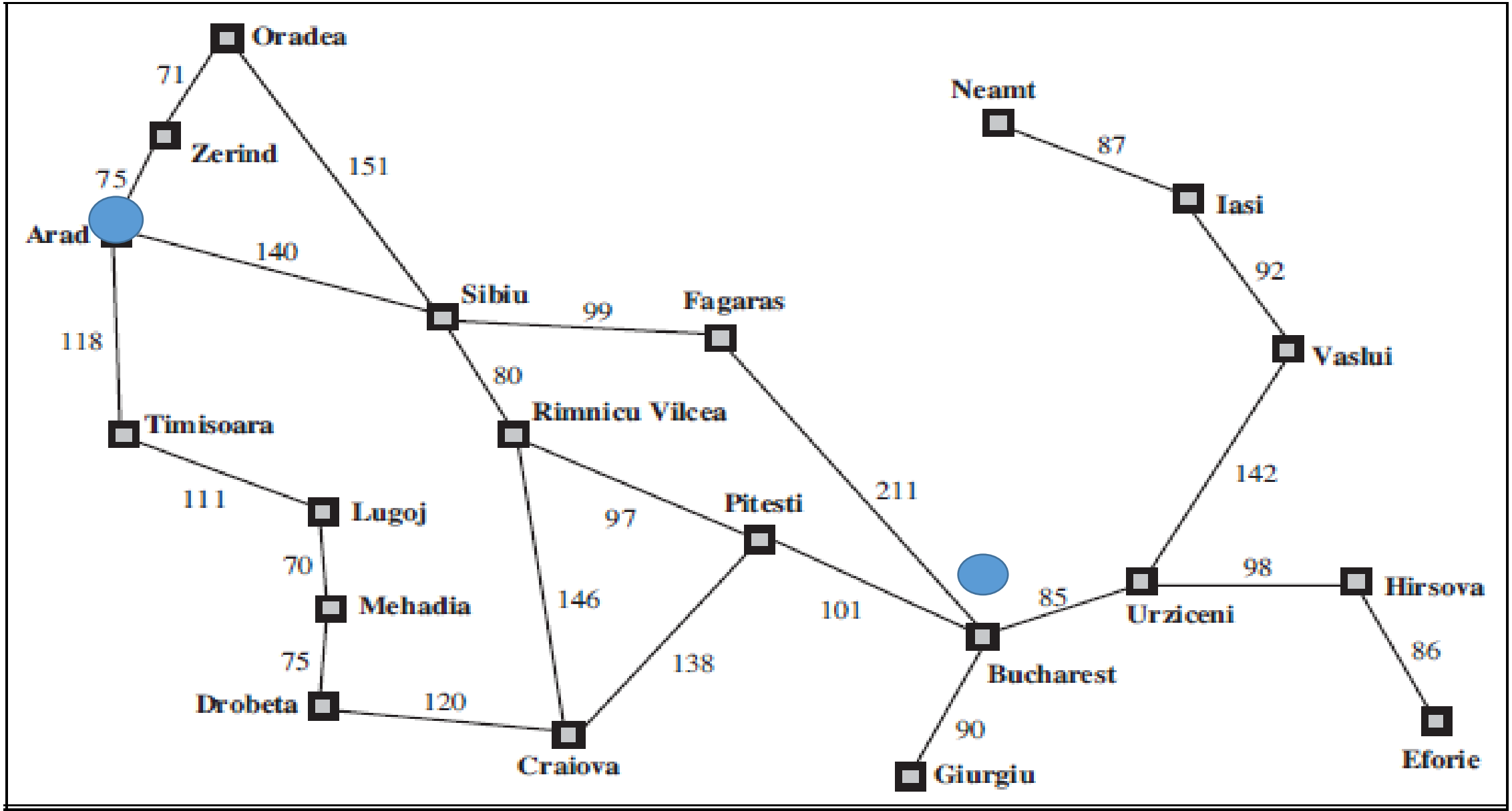


Figure 3.2 A simplified road map of part of Romania.

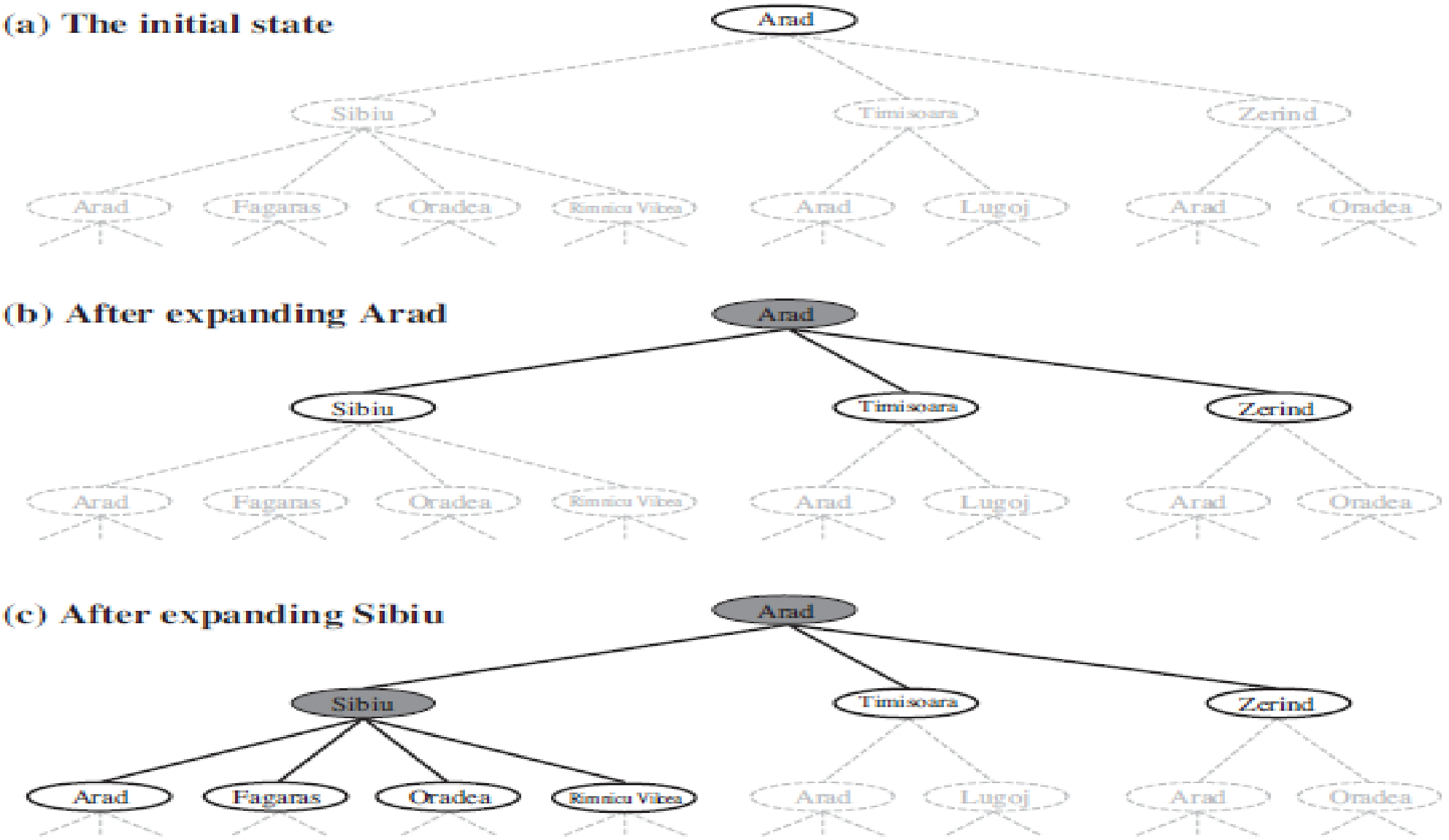


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

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

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

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

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.

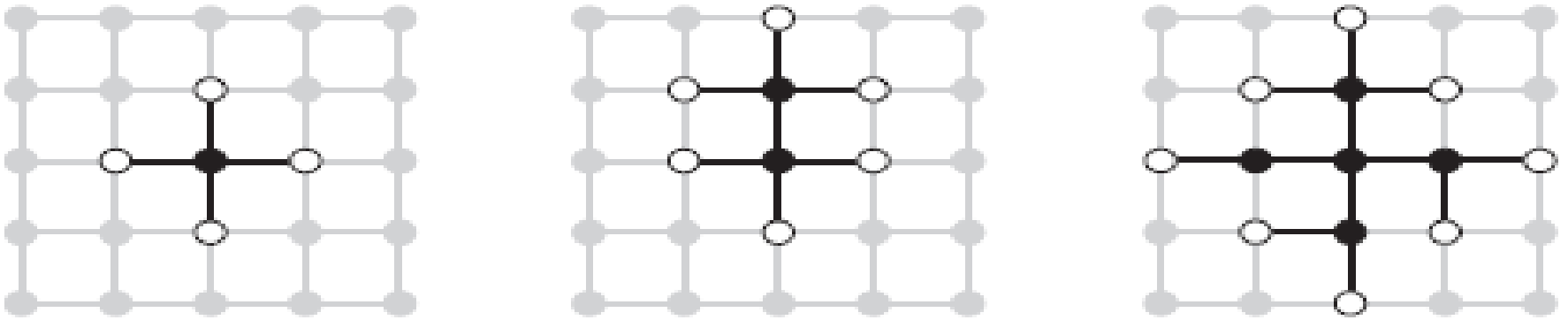


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

Infrastructure for search algorithms

- Search algorithms require a data structure for the search tree. Each node n of the tree, a structure has 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 by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

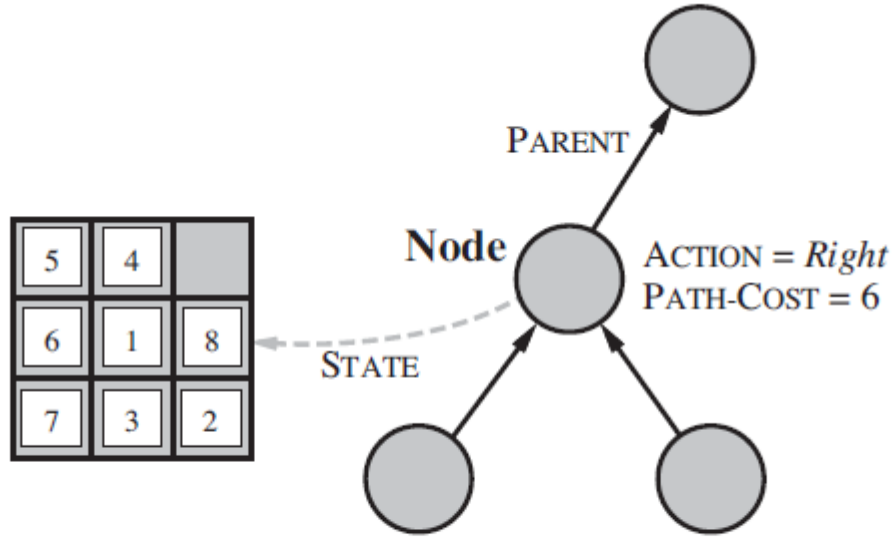


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

```
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 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.

Queues are characterized by the *order* in which they store the inserted nodes.

- first-in, first-out or **FIFO queue**, pops the *oldest* element of the queue;
- last-in, first-out or **LIFO queue** (also known as a **stack**),pops the *newest* element of the queue;
- **priority queue**, pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

design of specific search algorithms

- **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?

- size of the state space graph, |V | + |E|, where V is the set of vertices (nodes) of the graph and E is the set of edges (links).

complexity is expressed in three quantities:

- b:Maximum branch factor of the search tree
- d:depth of the least cost solution to reach the goal
- m:maximum depth of the state space

- **search cost—**
  - can use the **total cost**, which combines the search cost and the path cost of the solution found.
  - Ex:For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the solution cost is the total length of the path in kilometers.
  - to compute the total cost, to add milliseconds and kilometers

UNINFORMED SEARCH STRATEGIES

- Blind search.
- No additional information about states except the one provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.
- search strategies -by the *order* in which nodes are expanded.
- Strategies that know -goal state **informed search** or **heuristic search** strategies;

❖BFS
❖DFS

Breadth First Search(BFS)

- The root node is expanded first
- Then all the successors of the root node are expanded.
- Then all their successors and so on.
- All the of a given depth are expanded before any node of the next depth is expanded.
- Uses a standard queue FIFO as a data structure .

BFS

- BFS is complete(always finds goal if one exists)
- BFS finds the shallowest path to any goal node,if multiple goal nodes exist ,BFS finds the shortest path
- If the shallowest solution is at depth ‘d’ and the goal test is done when each node is **generated** then BFS generates $b+b^2+b^3$ ….$+b^d$ $=O(b^d)$ nodes that has time complexity of $O(b^d)$
- If the goal test is done when each node is **expanded** the time complexity of BFS is $O(b^{d+1})$
- The space complexity (frontier size)is also $O(b^d)$ .This is the biggest drawback of BFS

Limitations of BFS

- **memory requirements** are a bigger problem for breadth-first search than is the execution time.
- Time is still a major factor.

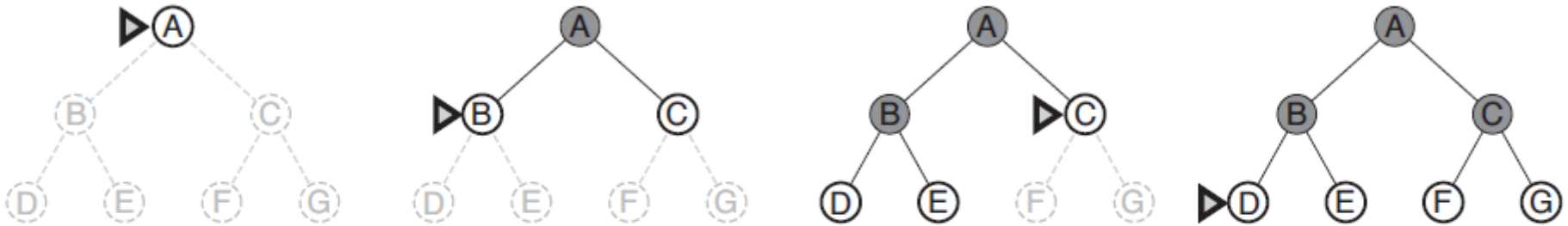
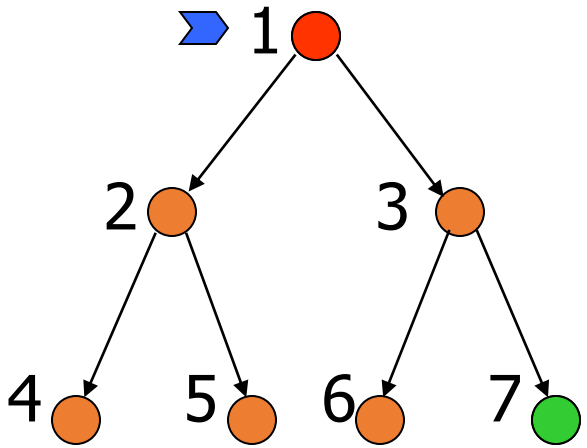


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-First Strategy

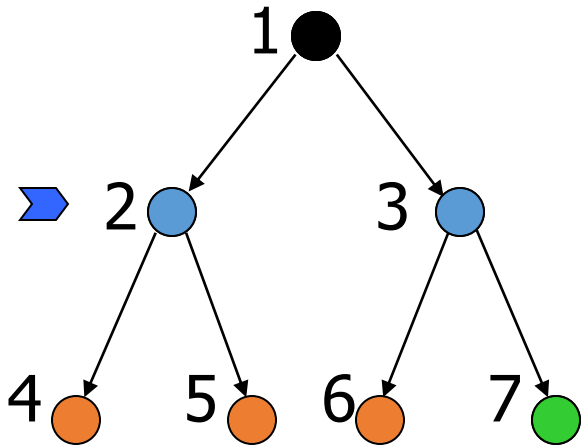
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

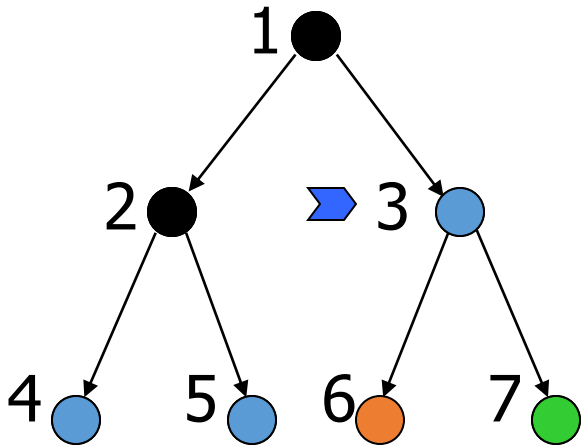
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

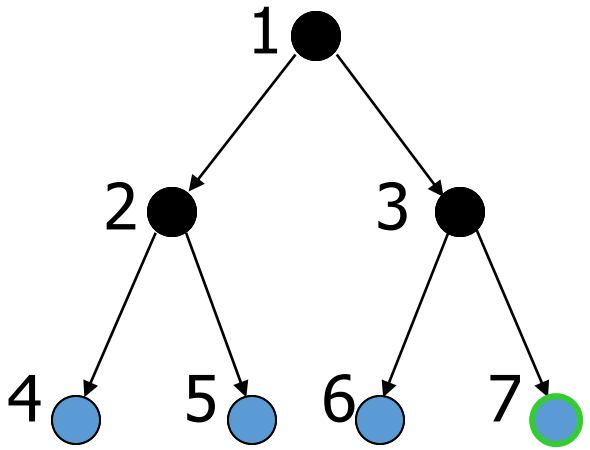
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

```
import collections
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
     while queue:
      # Dequeue a vertex from queue
       vertex = queue.popleft()
       print(str(vertex) + " ", end="")
        # If not visited, mark it as visited, and enqueue it
       for neighbour in graph[vertex]:
           if neighbour not in visited:
               visited.add(neighbour)
               queue.append(neighbour)
 if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

```
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.

Depth-first search

- Expands the *deepest* node in the current frontier of the search tree.
- It uses Stack (LIFO queue),means that the most recently generated node is chosen for expansion
- DFS is frequently programmed recursively.

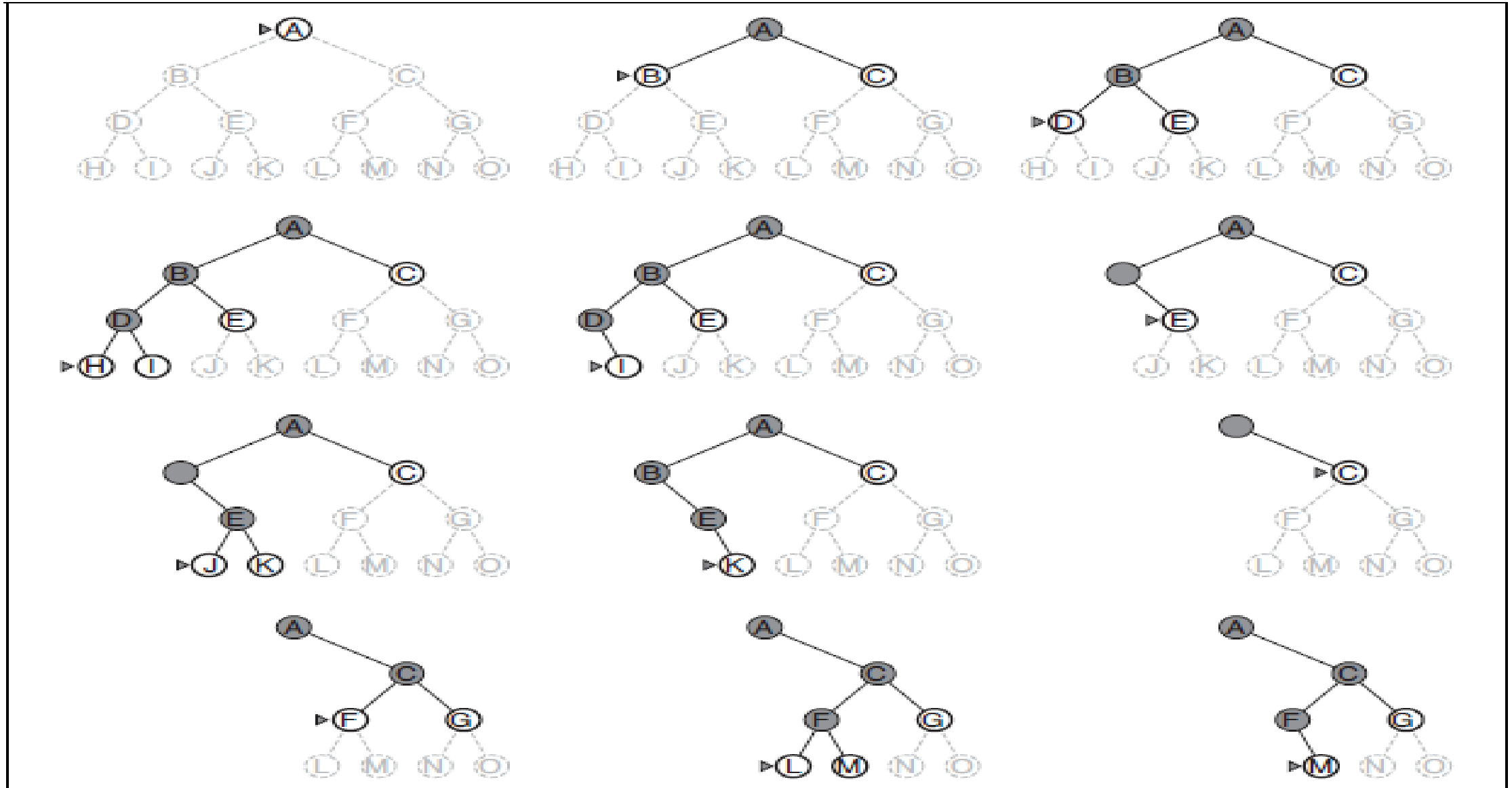


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

```
graph = {
    '0':['1', '2'], '1':['2'],'2':['3'],'3':['1', '2']
}
visited =set()
def dfs(visited,graph,root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbour in graph[root]:
            dfs(visited,graph,neighbour)
dfs(visited,graph,'0')
```

INFORMED (HEURISTIC) SEARCH STRATEGIES

- uses problem-specific knowledge -can find solutions more efficiently than can an uninformed strategy.
- General approach is **best-first search**.
- Best-first search is an instance of TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n).

INFORMED (HEURISTIC) SEARCH STRATEGIES

- The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first
- Informed search algorithms include a **heuristic function h(n) as a part of f(n)**
- f(n)=g(n)+h(n) g(n)→ the cost to reach the node
- h(n)=estimate of the cheapest cost form the state at node n to a goal state;
- h(goal)=0

Greedy best-first search

- GFS tries to expand the node that it estimates as being closest to the goal.
- likely to lead to a solution quickly
- It uses only the heuristic function h(n).
- f(n)=h(n).
- Use a straight line distance(SLD) heuristic h_{SLD} for the route finding in Romania to the goal Bucharest.
- h_{SLD}(In(Arad))=366. values of h_{SLD} cannot be computed from the problem description itself.
- It takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

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

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

- greedy best-first search using h_{SLD} to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal.
- For this particular problem, greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

- It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. **“greedy”**—at each step it tries to get as close to the goal as it can.

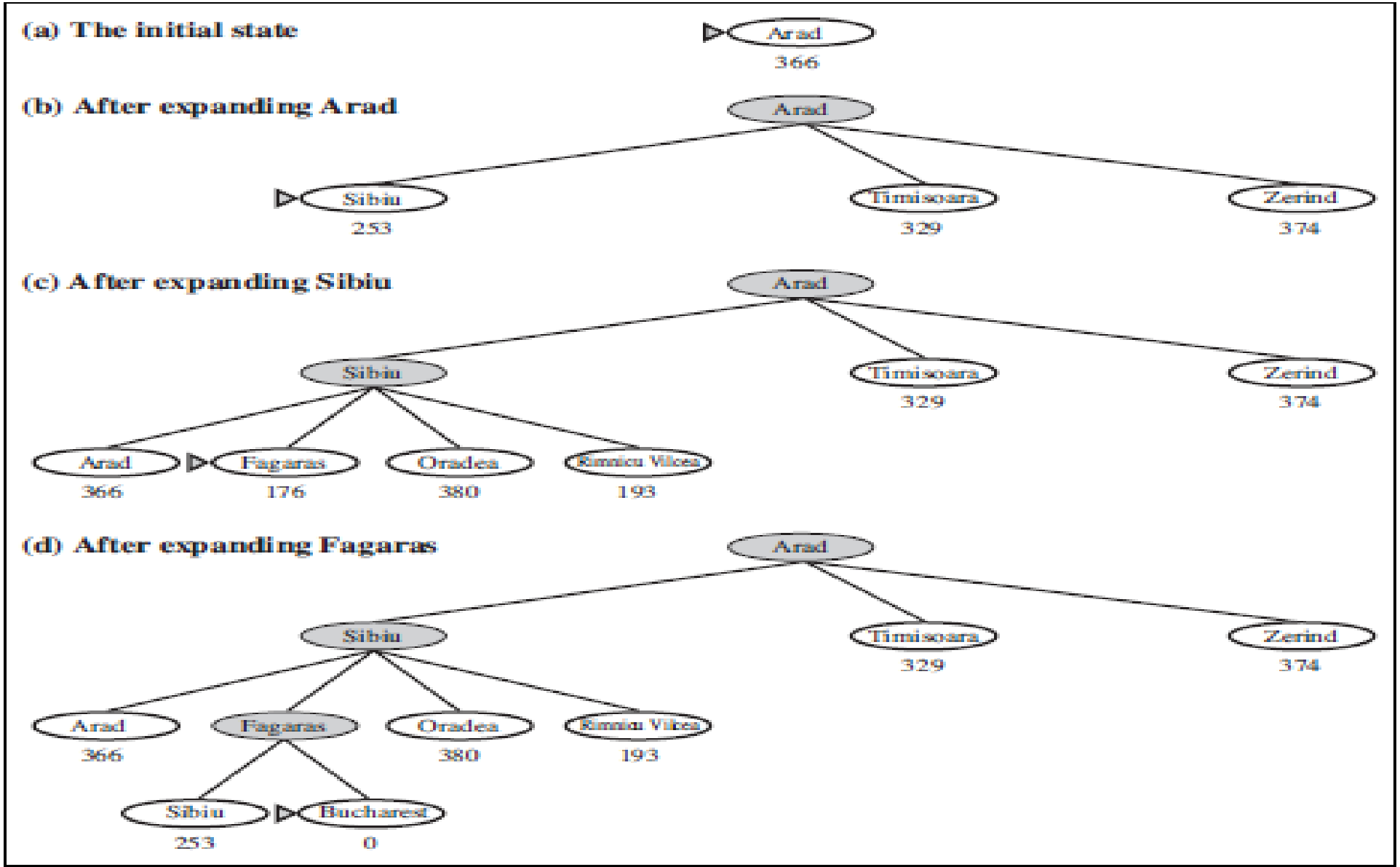


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- The algorithm is called greedy because in each step the algorithm greedily tries to get as close to the goal as possible.
- time and space complexity for the tree version is $O(b^m)$,
- Suggestion for improvement :use the accumulated path distance g(n) plus a heuristic h(n) as cost function f(n).This leads to A*

- Consider the problem of getting from Iasi to Fagaras.
- The heuristic suggests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end.
- The solution is to go first to Vaslui—a step that is actually farther from the goal according to the heuristic—and then to continue to Urziceni, Bucharest, and Fagaras.
- The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop.

- The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space.

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) .
- g(n) gives the path cost from the start node to node n,
- h(n) is the estimated cost to reach the goal.
- f(n) = estimated cost of the cheapest solution through n .
- Under certain conditions A∗ search is both complete and optimal.

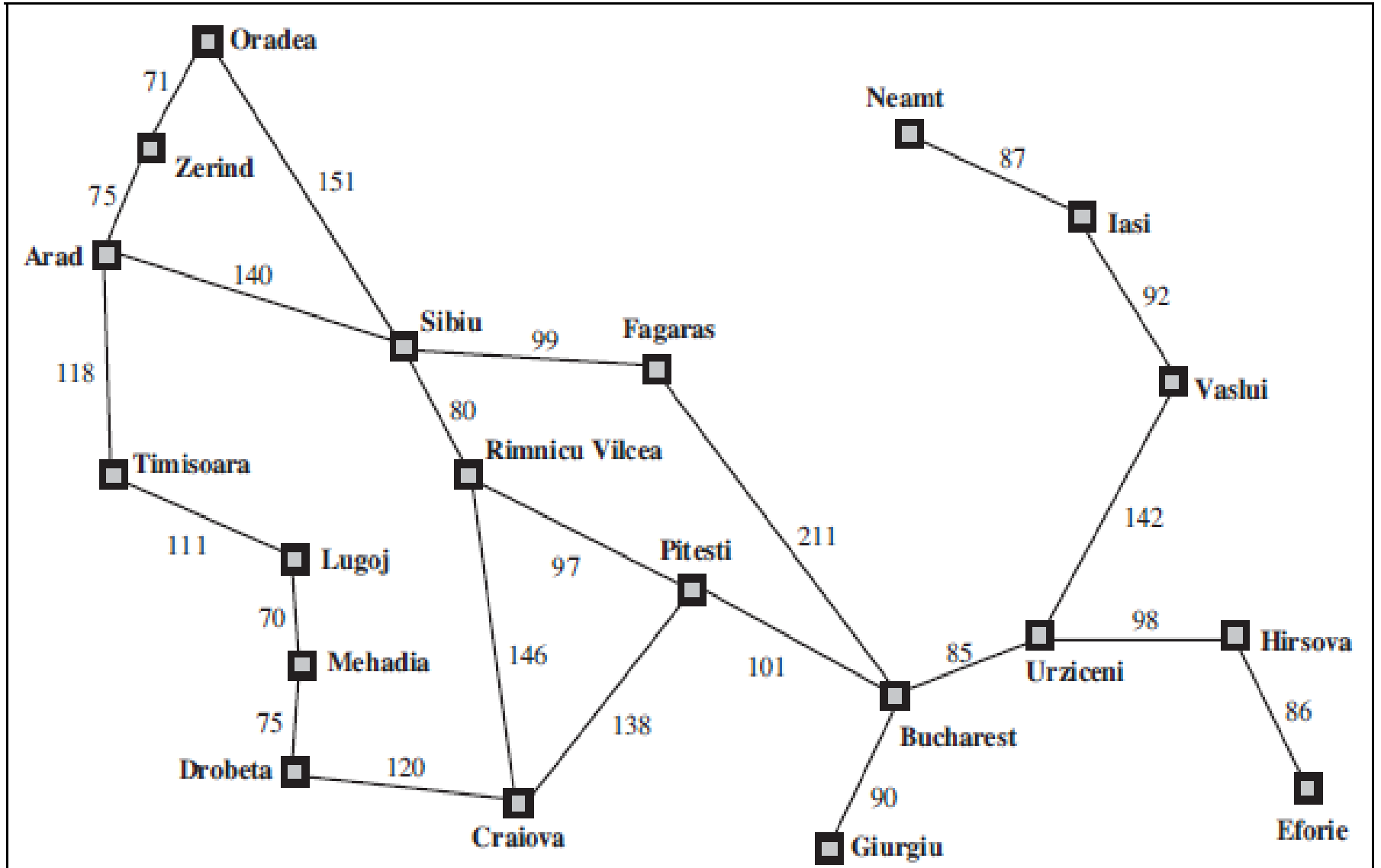


Figure 3.2 A simplified road map of part of Romania.

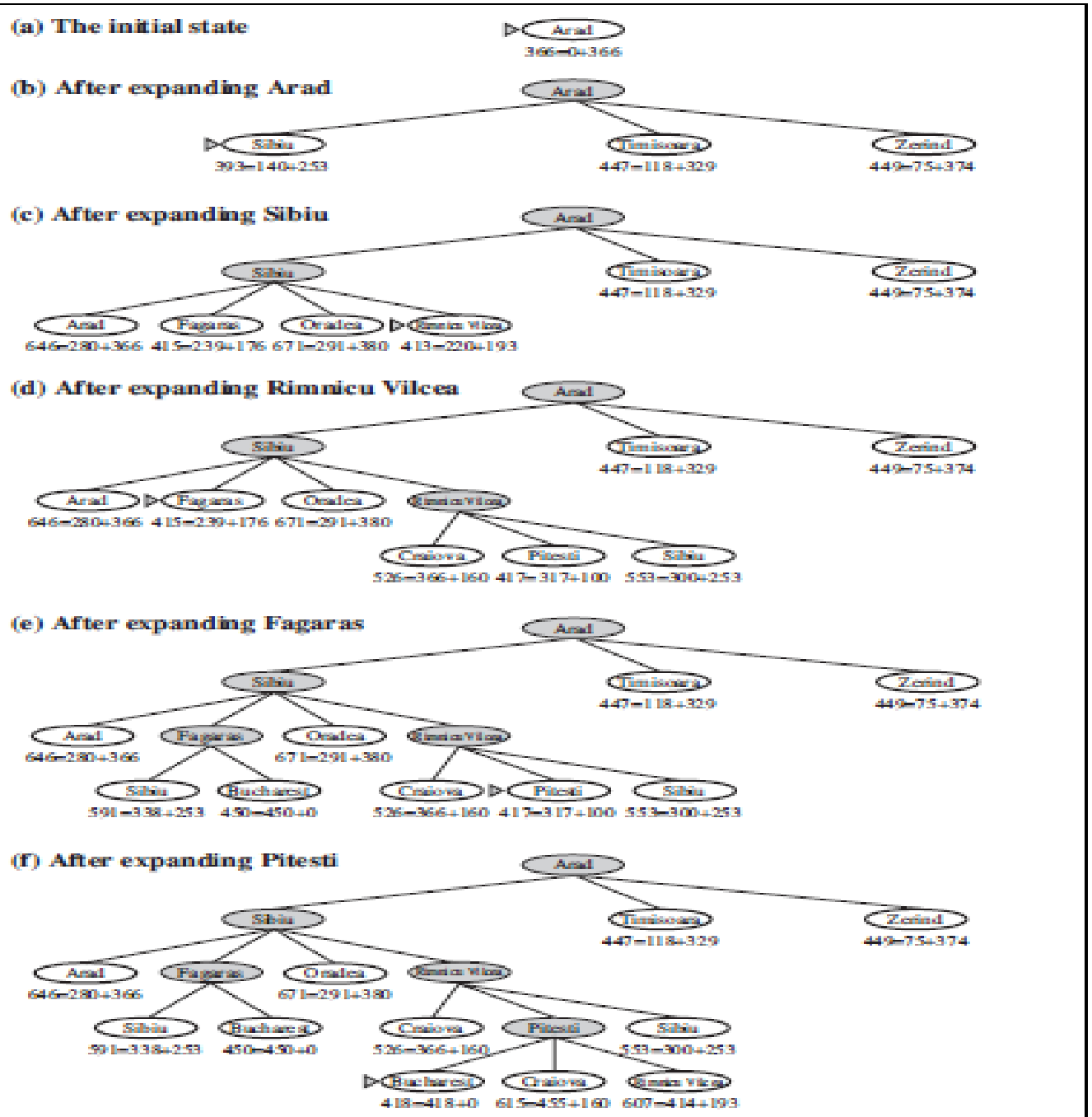


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

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.

```
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1),('H', 7)] ,
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],

}
```

```
 def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
```

```
def h(n):
        H_dist = {
            'A': 10,
            'B': 8,
            'C': 5,
            'D': 7,
            'E': 3,
            'F': 6,
            'G': 5,
            'H': 3,
            'I': 1,
            'J': 0
        }
    return H_dist[n]
```

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {}
        parents = {}
        g[start_node] = 0
        parents[start_node] = start_node

        while len(open_set) > 0:
            n = None

            for v in open_set:
                if n == None or g[v] + h(v) < g[n] + h(n):
                    n = v

            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight

                    else:
                        if g[m] > g[n] + weight:
                            g[m] = g[n] + weight
                            parents[m] = n
                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)
```

```
    if n == None:
                    print('Path does not
    exist!')
                    return None
                if n == stop_node:
                    path = []

                    while parents[n] != n:
                        path.append(n)
                        n = parents[n]

path.append(start_node)
                    path.reverse()
                    print('Path found: {}'.format(path))
                   return path
            open_set.remove(n)
            closed_set.add(n)

        print('Path does not exist!')
        return None
 aStarAlgo('A', 'J')
```

Conditions for optimality of A*

tree-search version of A* is optimal

- h(n) must be admissible, i.e it never overestimates the cost to reach the goal.
- Then, as a consequence ,f(n)=g(n)+h(n) never overestimates the true cost of a solution along the current path through n.
- h(n) must be consistent (monotonic)in a graph search, i.e for every node n and every successor $n^{|}$ of n generated by action a ,
- $h(n) \leq c(n, a, n^{|}) + h(n^{|})$
- This is a form of the triangular inequality.
- Every consistent heuristic is also admissible.

Heuristic functions

A typical instance of the 8-puzzle. The solution is 5 steps long

The 8-puzzle was one of the earliest heuristic search problems.

1	2	3
4	8	-
7	6	5

1	2	3
4	5	6
7	8	-

Heuristic functions

- 8-puzzle was one of the earliest heuristic search problems.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3. (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.

- an exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{10}$ states.
- h1 = the number of misplaced tiles.
- 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.

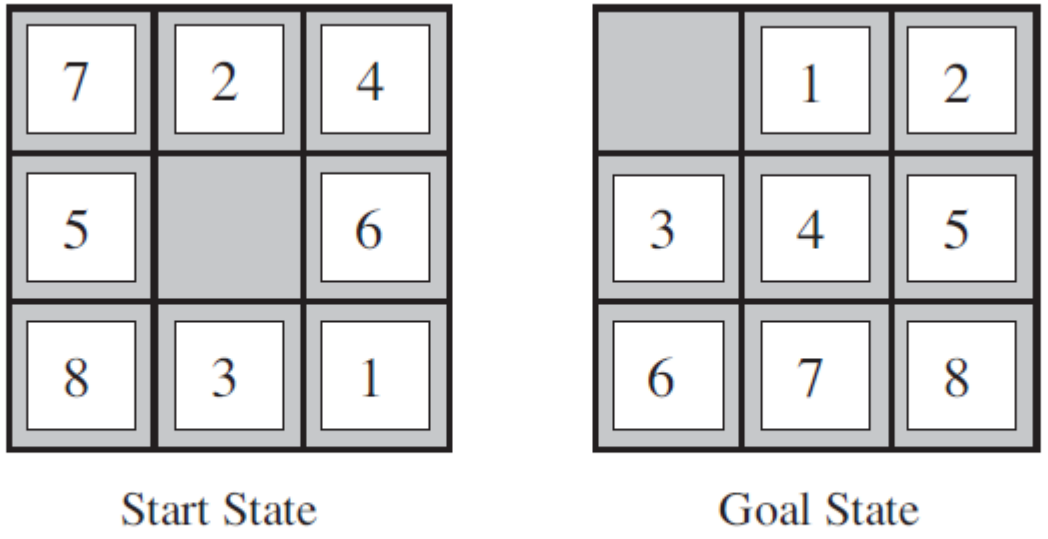


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

- 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.**
- **city block distance** or **Manhattan distance.**
- h2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18 .

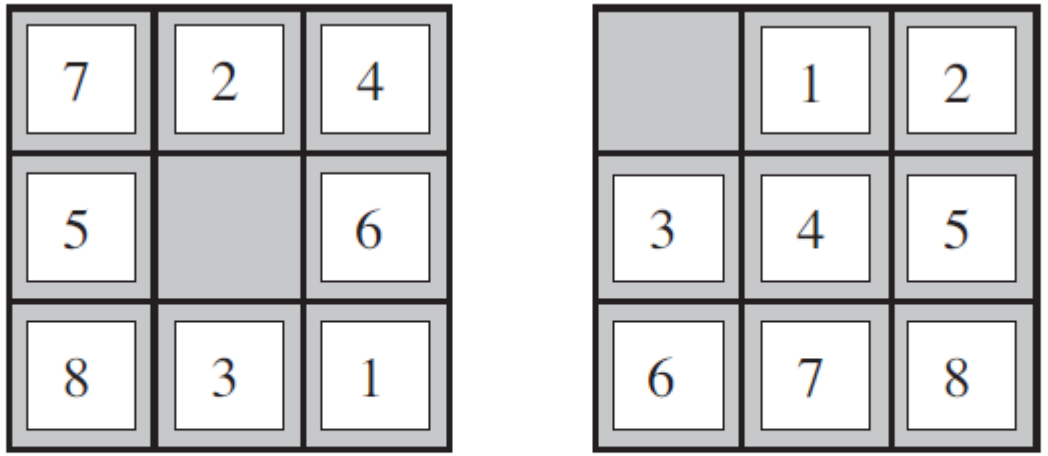


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.