Artificial Intelligence

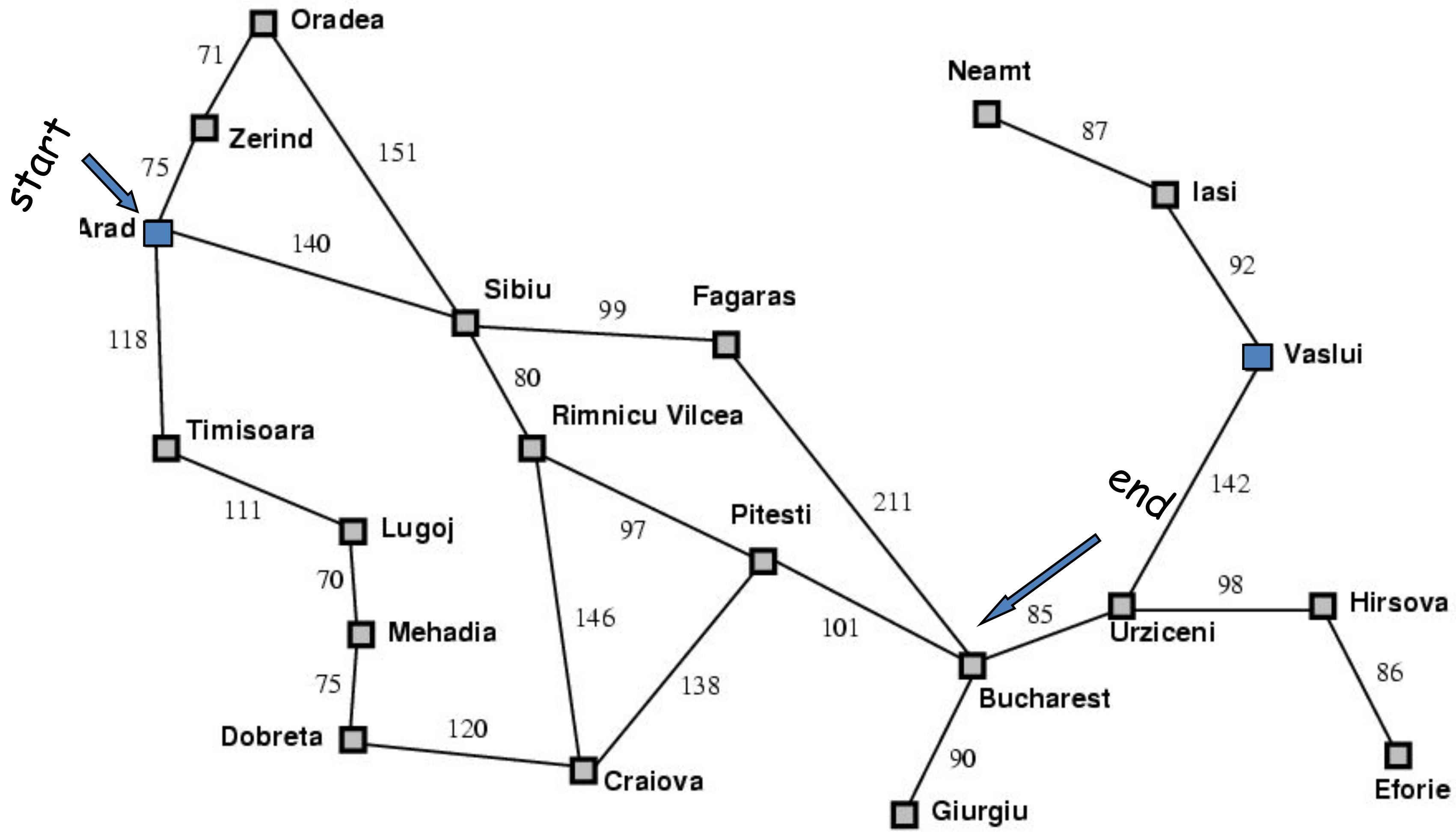
By
S. Komal Kaur,
Assistant Professor,
Vasavi College of Engineering,
Hyderabad

Chapter 3

Solving Problem by Searching

- Problem Solving Agents
- Uninformed Search
- Informed Search
- Heuristic Search
- Local Search algorithm and optimization problems.

A Simplified Road Map of Romania



Problem Solving Agents

- Goal-based agent is called as a problem-solving agent.
- Problem-solving agents use atomic representations, that is, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms.
- Intelligent agents are supposed to maximize their performance measure.

Problem Solving Agents

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
- Goal formulation, based on the current situation and the agent’s performance measure, is the first step in problem solving.
- Problem formulation is the process of deciding what actions and states to consider, given a goal.

Solving Problems by Searching

- Well-defined problems and solutions A problem can be defined formally by five components:
- PROBLEM

1. The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad).

Solving Problems by Searching

2. 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(Arad), the applicable actions are {Go(Sibiu),Go(Timisoara),Go(Zerind)}.

3. 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.

- For example, we have

  RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

Solving Problems by Searching

4. The goal test, which 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. The agent’s goal in Romania is the singleton set {In(Bucharest)}.
5. 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.

Search through a Problem Space / State Space

- Input:
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state [test]

- Output:
  - Path: start ⇒ a state satisfying goal test
  - [May require shortest path]

```
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.

Example Problem

- Toy Problem

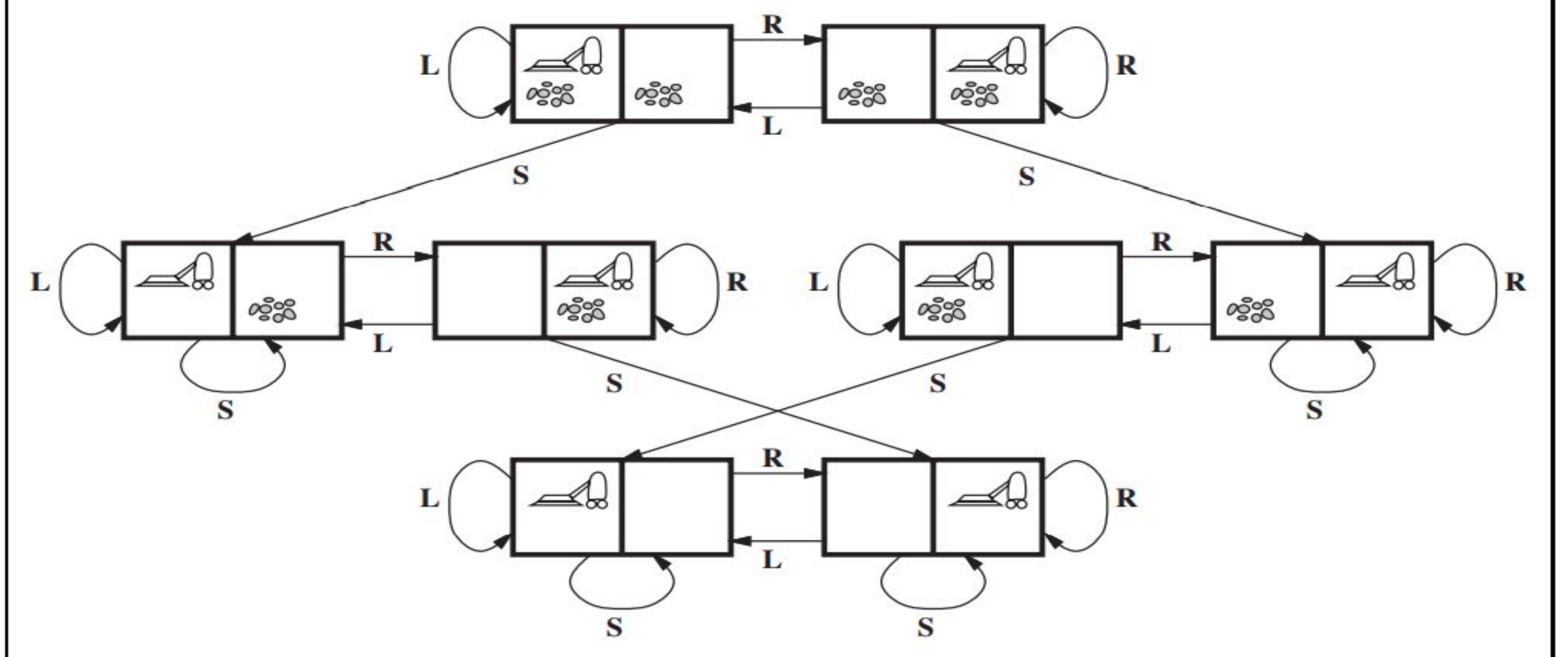


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

- This can be formulated as a problem as follows:

- **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 · 2^n states.

- **Initial state**: Any state can be designated as the initial state.

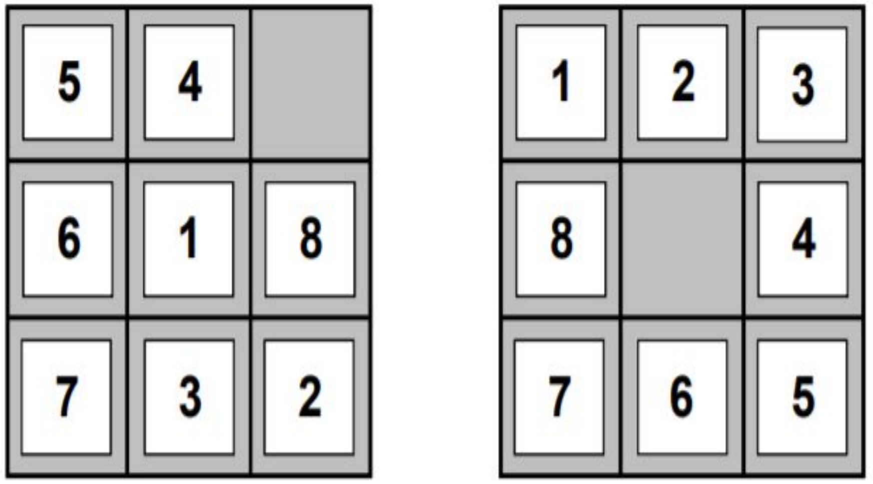
- **Actions**: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.

- **Transition model**: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure .

- **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.

Example: The 8-puzzle



Start State Goal State

states??
operators??
goal test??
path cost??

8 Puzzle Problem

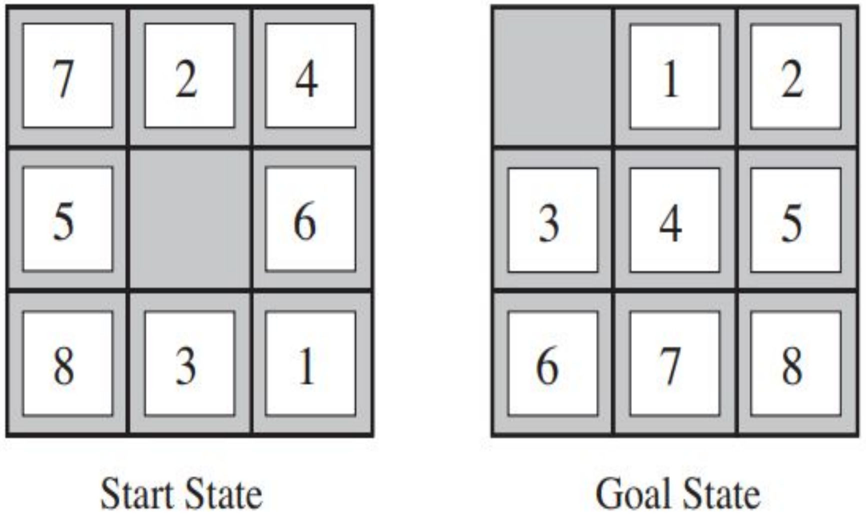
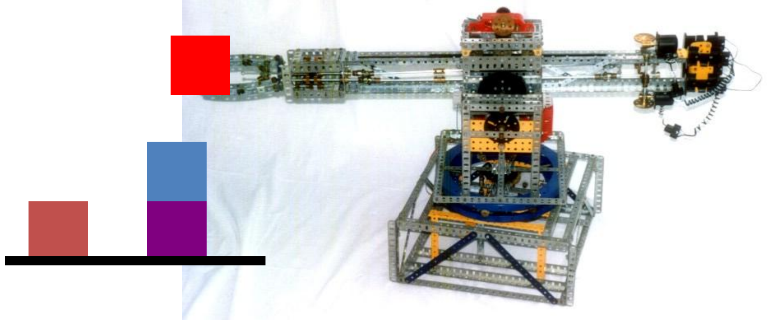


Figure 3.4 A typical instance of the 8-puzzle.

- The **8-puzzle**, an instance of which is shown in Figure , consists of a 3×3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The object is to reach a specified goal state, such as the one shown on the right of the figure.
- The standard formulation is as follows:
- States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down.
- Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure , the resulting state has the 5 and the blank switched.
- Goal test: This checks whether the state matches the goal configuration shown in Figure . (Other goal configurations are possible.)
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Ex: Blocks World



- Input:
  - Set of states

    Partially specified plans
  - Operators [and costs]

    Plan modification operators
  - Start state

    The null plan (no actions)
  - Goal state (test)

    A plan which provably achieves

    The desired world configuration

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 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

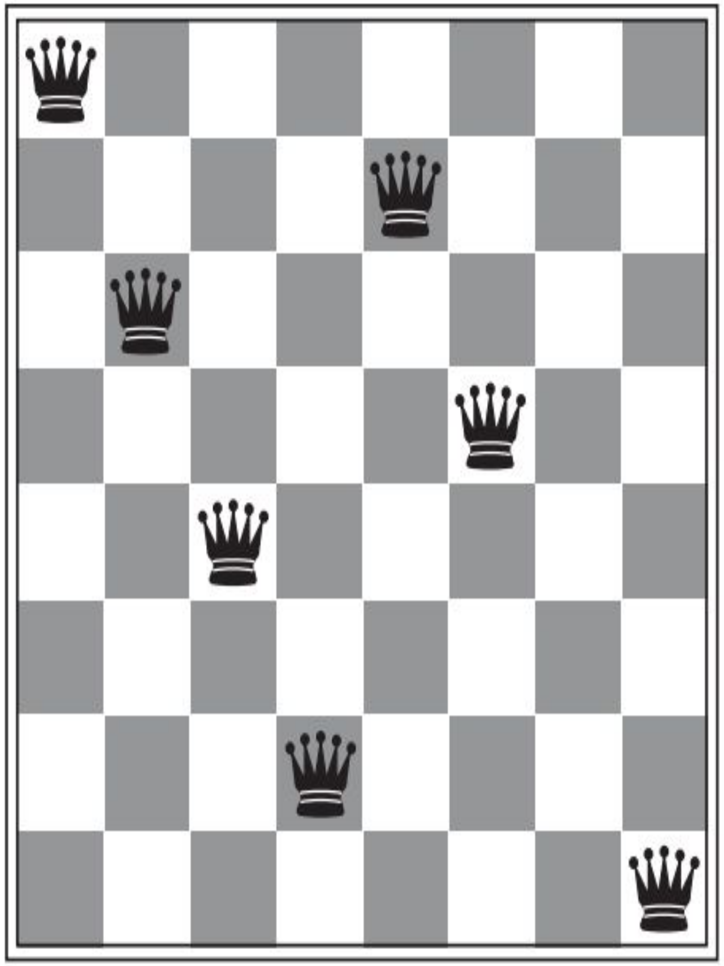


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

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.

In this formulation, we have 64 · 63 ··· 57 ≈ 1.8 × 10^14 possible sequences to investigate.

The first queen can be placed in any of the 64 squares. The next can be placed in any of the remaining 63 squares. Continuing in this fashion we get

64×63×62×61×60×59×58×57

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 shows the first few steps in growing the 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; that is, 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 set of all leaf nodes available for expansion at any given FRONTIER point is called the frontier.
- (Many authors call it the open list, which is both geographically OPEN LIST less evocative and less accurate, because other data structures are better suited than a list.)

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- The general TREE-SEARCH algorithm is shown informally.
- The eagle-eyed reader will notice one peculiar thing about the search tree shown in Figure 3.6: it includes the 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.
- Loopy paths are a special case of the more general concept of redundant paths, which exist whenever there is more than one way to get from one state to another.

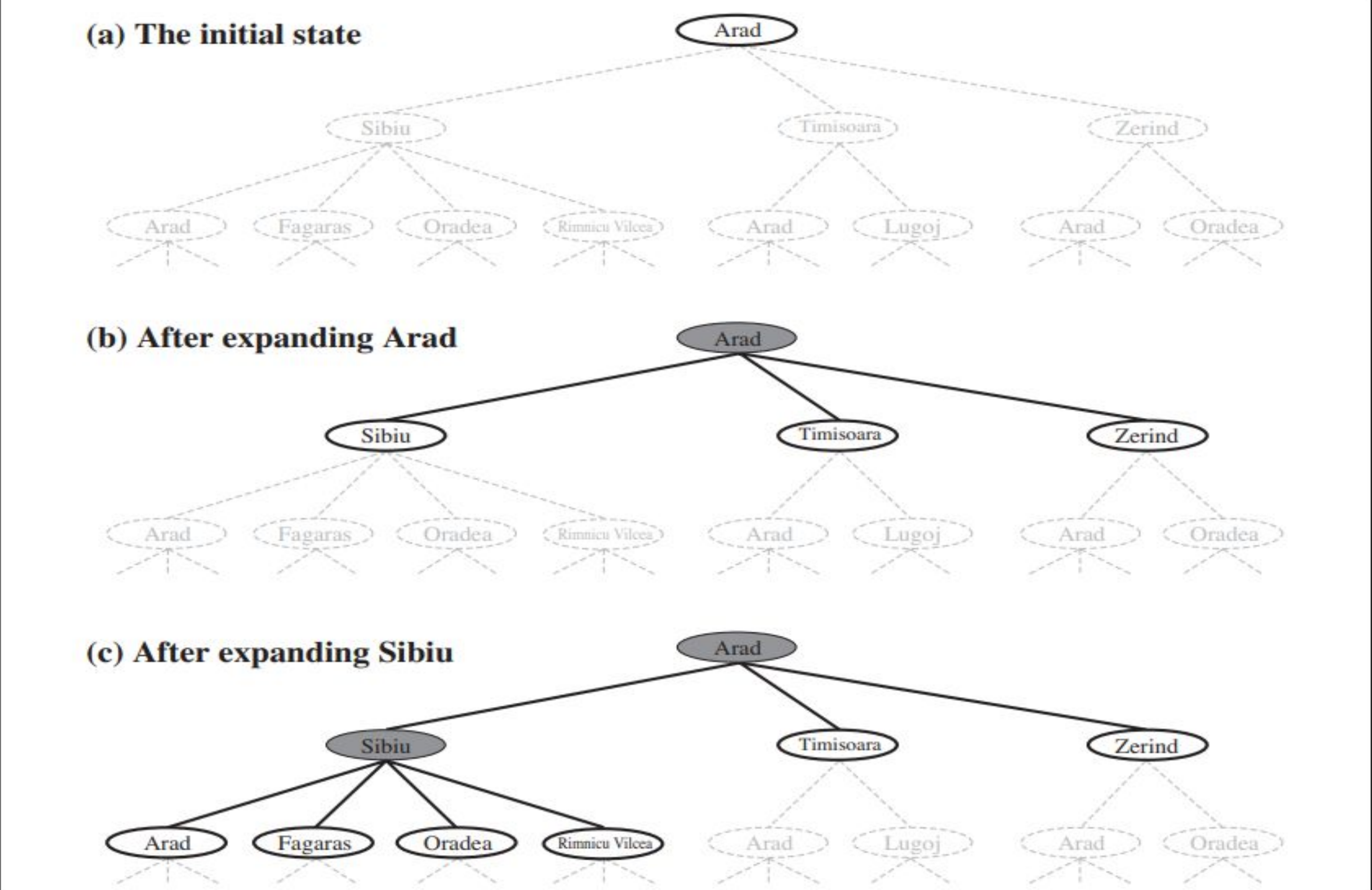


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

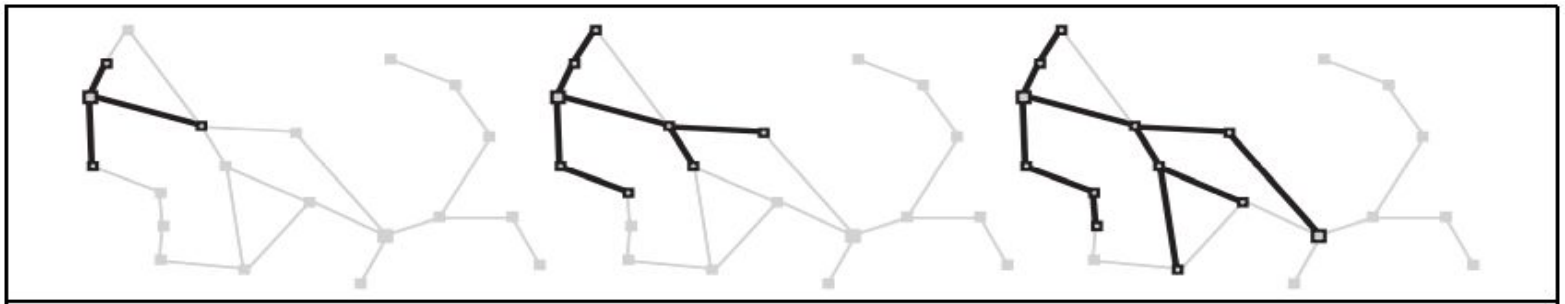


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

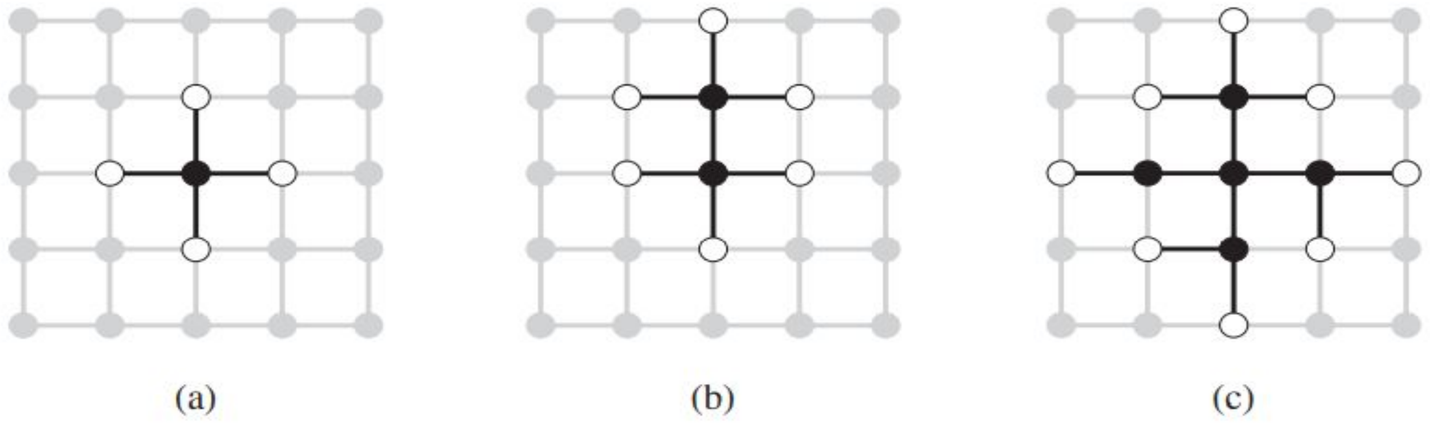


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

```
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.

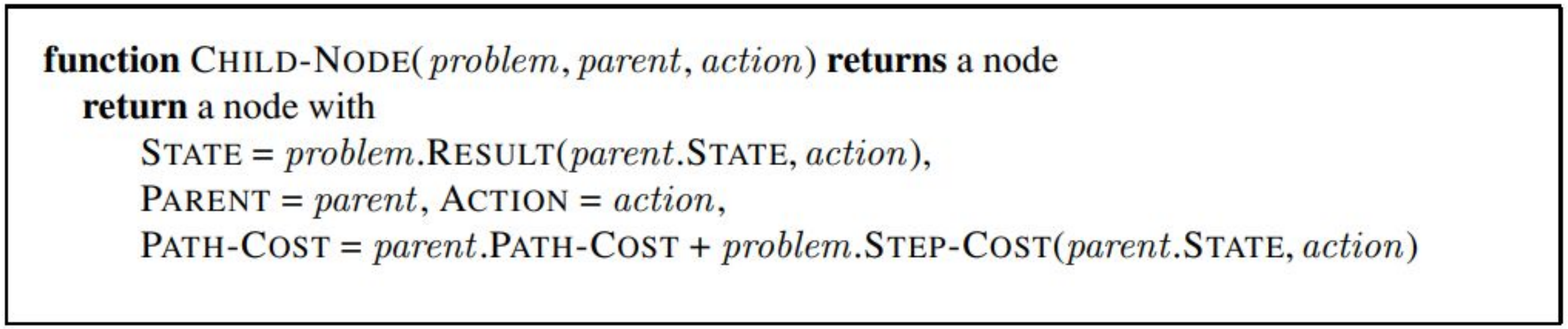
Searching For Solutions Terminology

- Search space → possible conditions and solutions.
- Initial state → state where the searching process started.
- Goal state → the ultimate aim of searching process.
- Problem space → “what to solve”
- Searching strategy →strategy for controlling the search.
- Search tree → tree representation of search space, showing possible solutions from initial state.

Infrastructure for Search Algorithm

- 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.

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)
```

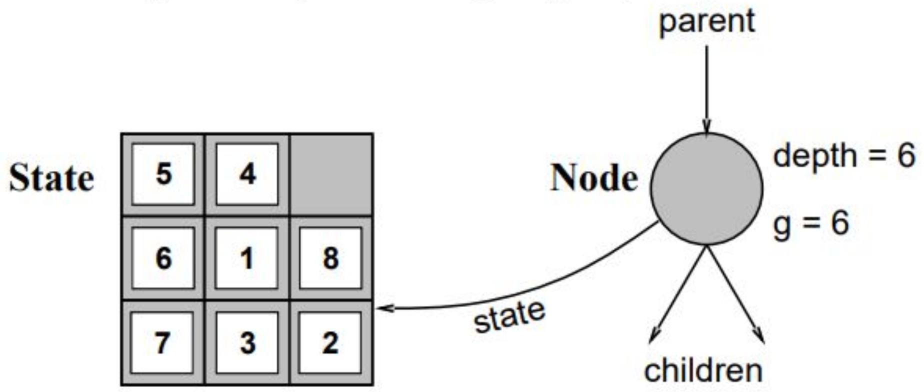
The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a queue.

The 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

Infrastructure for Search Algorithm

A *state* is a (representation of) a physical configuration
A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the OPERATORS (or SUCCESSORFN) of the problem to create the corresponding states.

Measuring Problem Solving Performance

- The output from problem-solving (searching) algorithm is either FAILURE or SOLUTION.
- Four ways:
  - **Completeness** : is guaranteed to find a solution?
  - **Optimality** : does it find optimal solution ?
  - **Time complexity** : how long?
  - **Space complexity** : how much memory?
    - Complexity : branching factor (*b*), depth (*d*), and max. depth (*m*)

Search Strategies

- Blind Search/Uninformed Search
  - Breadth first search
  - Uniform Cost Search
  - Depth first search
  - Depth limited search
  - Iterative deepening search
- Informed Search
- Constraint Satisfaction
- Adversary Search

Breadth First Search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, 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 (Figure 3.7) in which the shallowest unexpanded node is chosen for expansion.
- This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first

```
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.

BFS

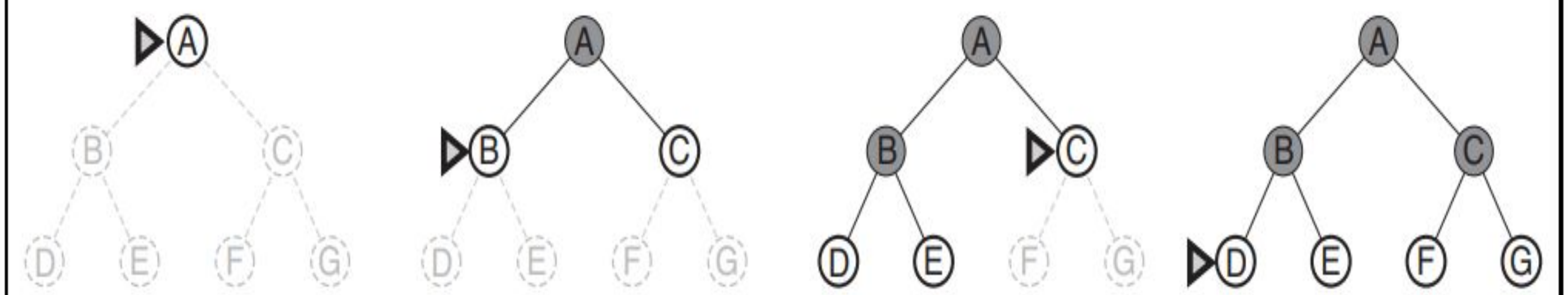


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

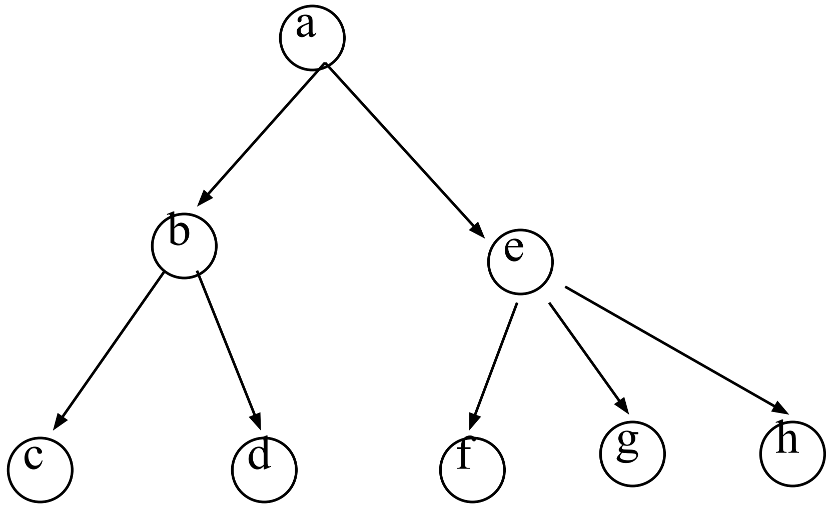
Breadth First Search

- Maintain queue of nodes to visit
- Evaluation
  - Complete?

    Yes (if branching factor is finite)
  - Time Complexity?

    $O(b^{d+1})$
  - Space Complexity?

    $O(b^{d+1})$



Uniform Cost Search

- Uniform Cost Search expands the node n with the lowest path cost g(n) by storing the frontier as a priority queue ordered by g.
- Uniform Cost Search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution.
- Enqueue nodes by **path cost**.
- g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.

UCS

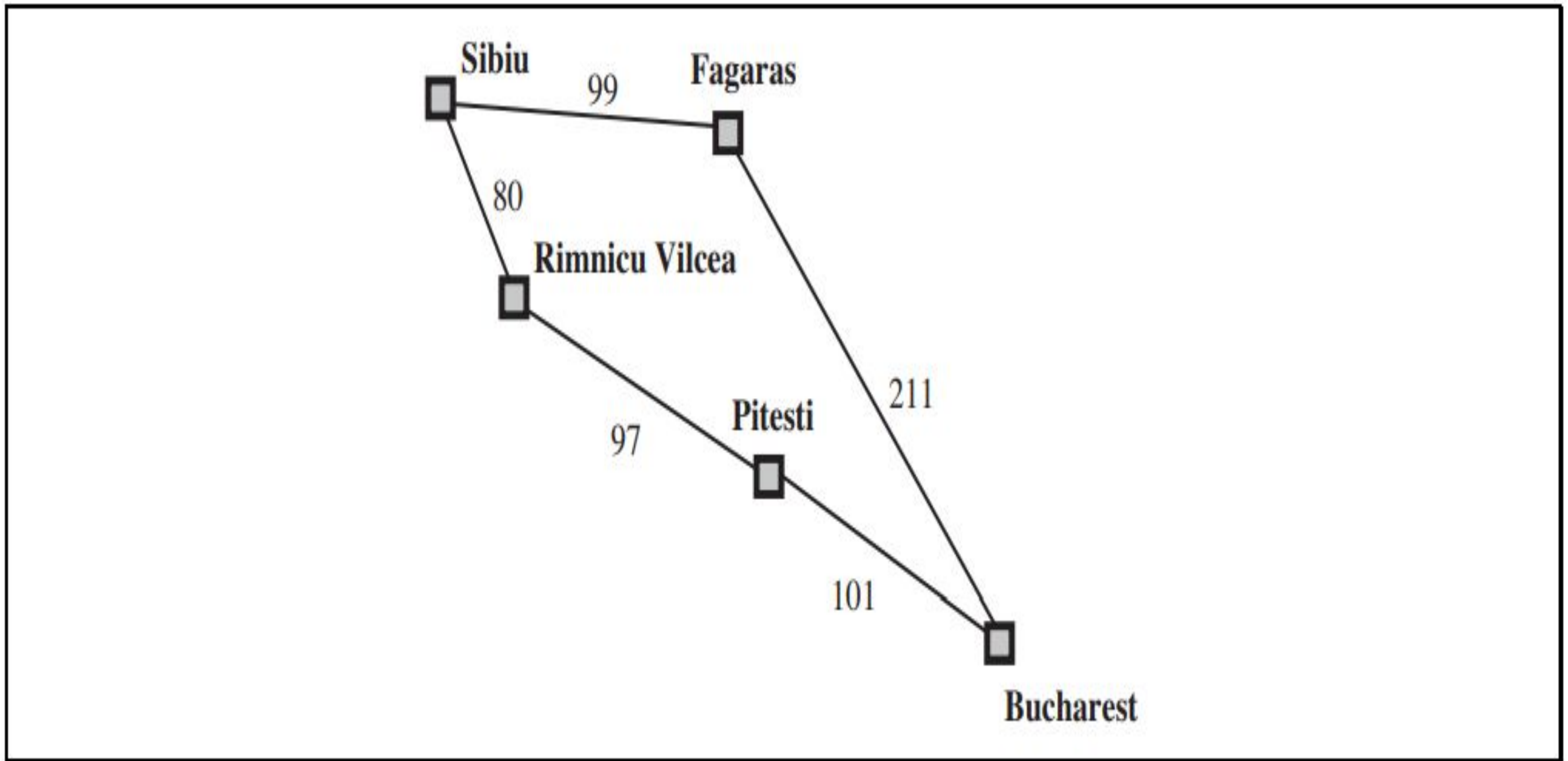


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

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

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

Depth First Search

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The progress of the search is illustrated in Figure 3.16. 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.
- The depth-first search algorithm is an instance of the graph-search algorithm in Figure 3.7; whereas breadth-first-search uses a FIFO queue, 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 the deepest unexpanded node when it was selected

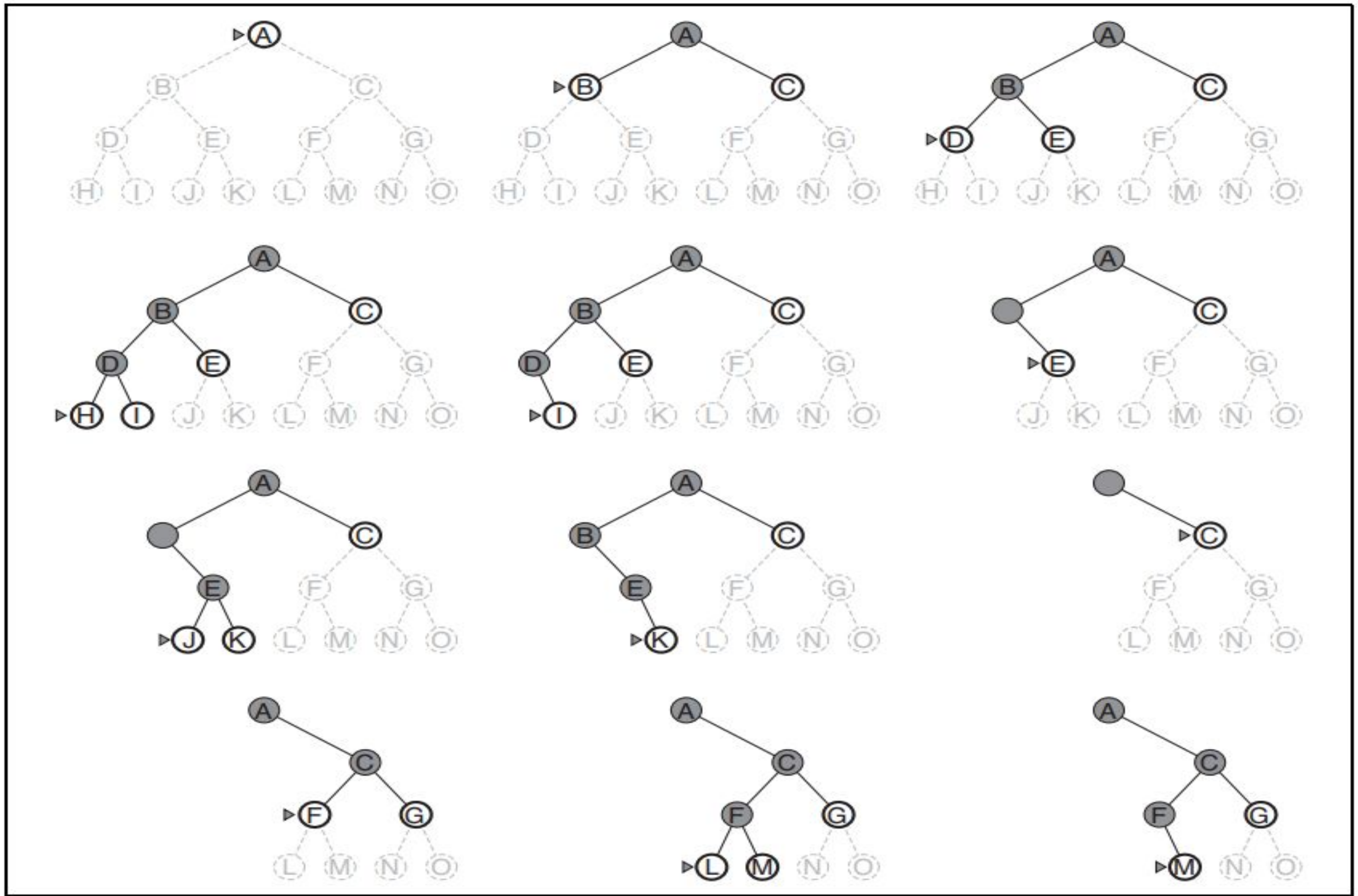


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth First Search

- Maintain stack of nodes to visit
- Evaluation
  - Complete?

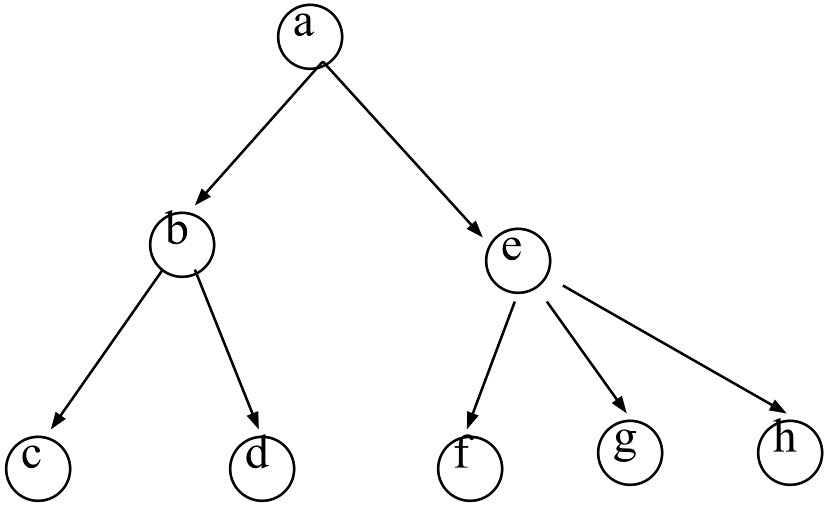
    Not for infinite spaces

  - Time Complexity?

    O(b^d)

  - Space Complexity?

    O(bd)

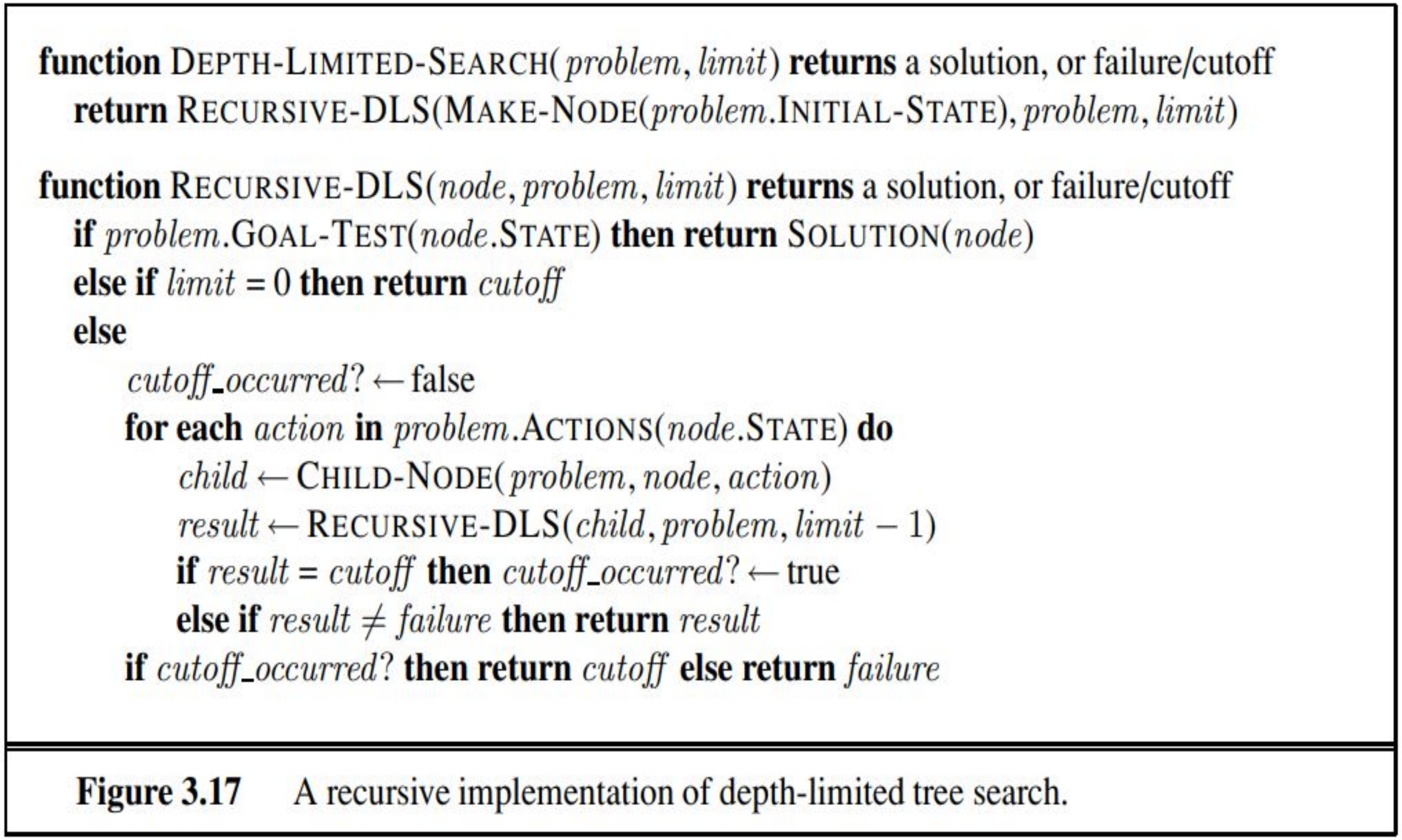


d=max depth of tree

Depth-limited search

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l.
- That is, nodes at depth l are treated as if they have no successors.
- This approach is called depth-limited search.
- The depth limit solves the infinite-path problem.

DLS



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

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

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

Iterative Deepening DF 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. The algorithm is shown in Figure 3.18.
- Iterative deepening combines the benefits of depth-first and breadth-first search.
- Like depth-first search, its memory requirements are modest: O(bd) to be precise.
- Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.
- Figure shows four iterations of ITERATIVE-DEEPENING-SEARCH on a binary search tree, where the solution is found on the fourth iteration

IDDFS

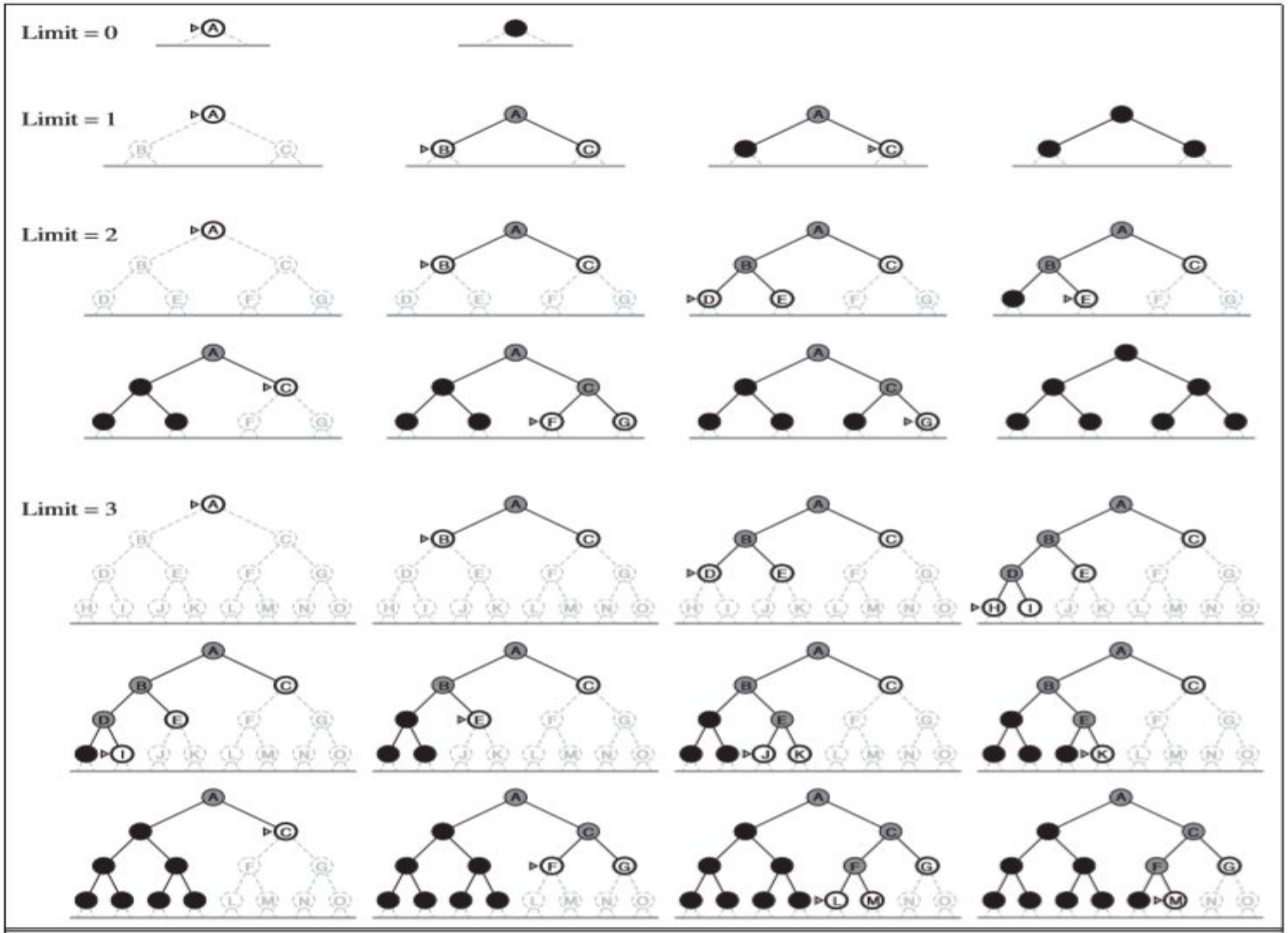
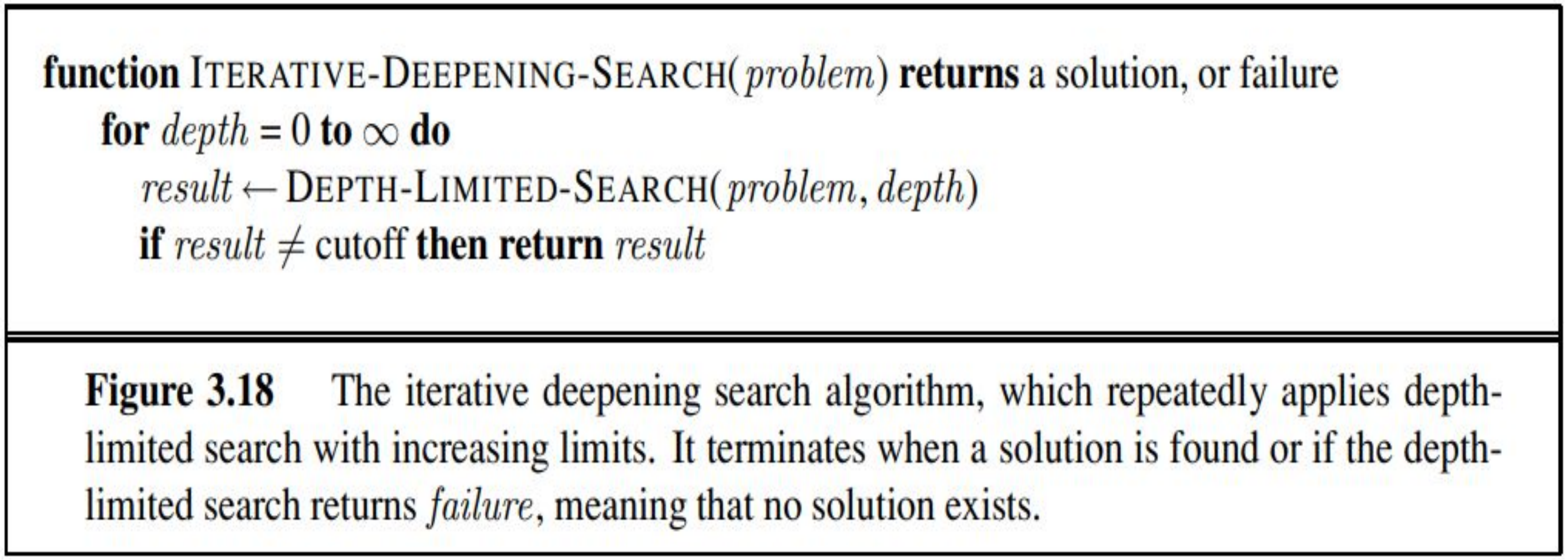


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

IDDFS



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

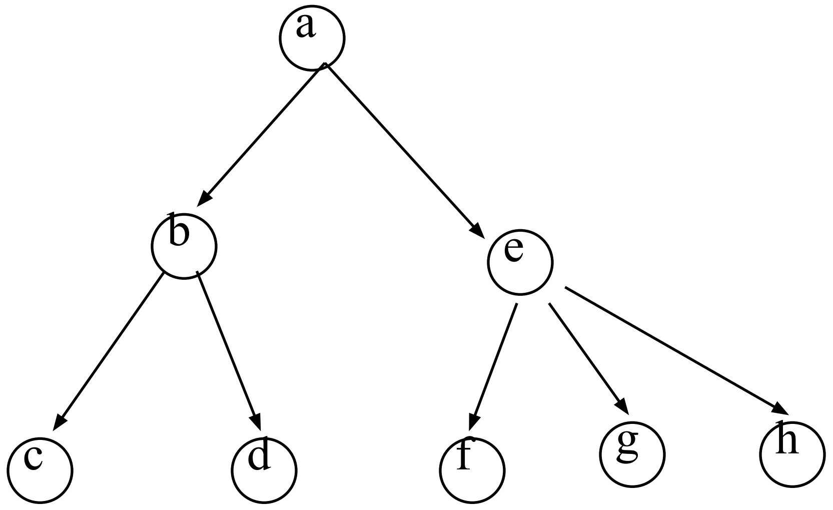
Iterative Deepening Search

- DFS with limit; incrementally grow limit
- Evaluation
  - Complete?

    Yes
  - Time Complexity?

    O(b^d)
  - Space Complexity?

    O(d)

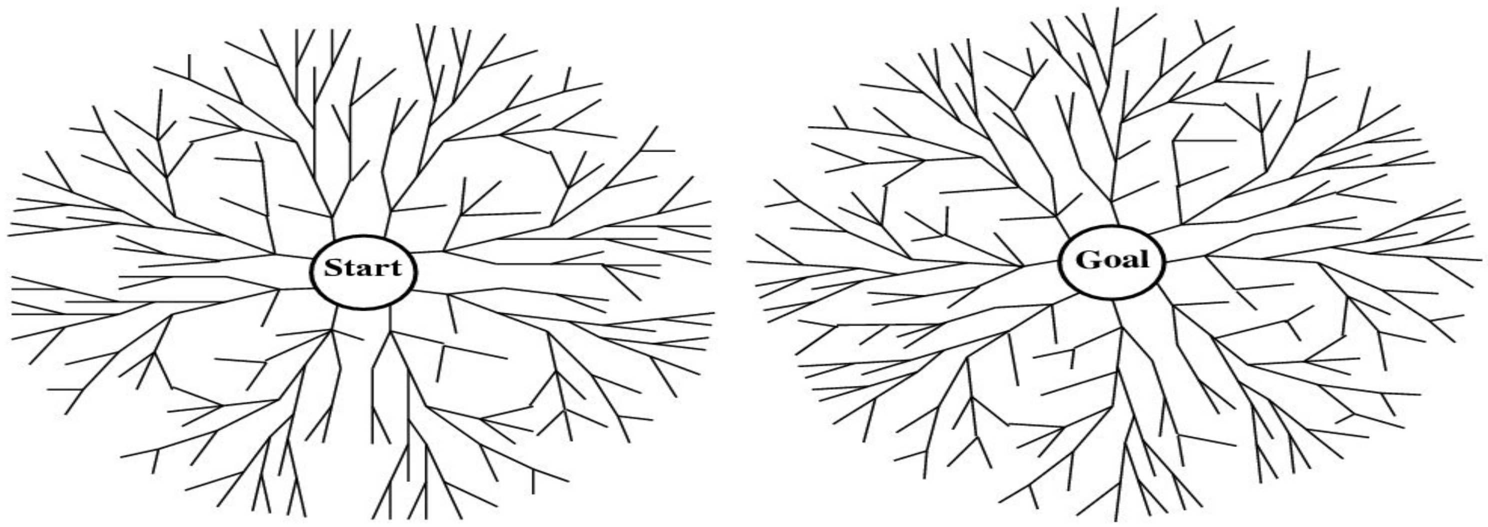


Bidirectional Search

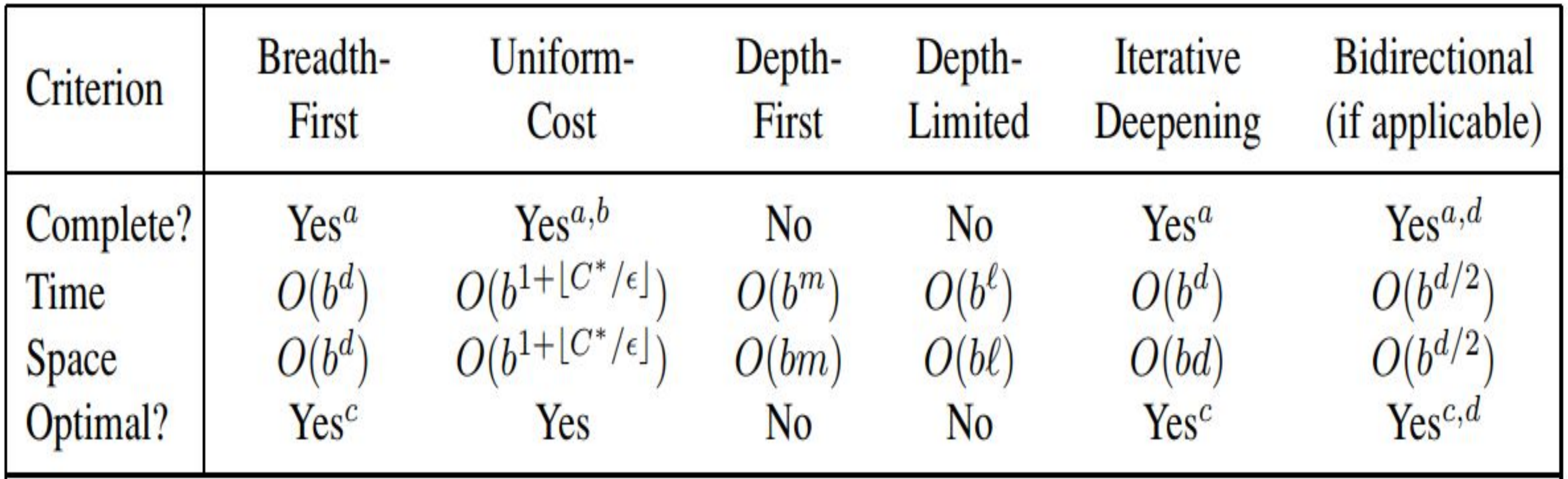
The idea behind bidirectional search is to run two simultaneous searches

—one forward from the initial state and the other backward from the goal.

—hoping that the two searches meet in the middle.



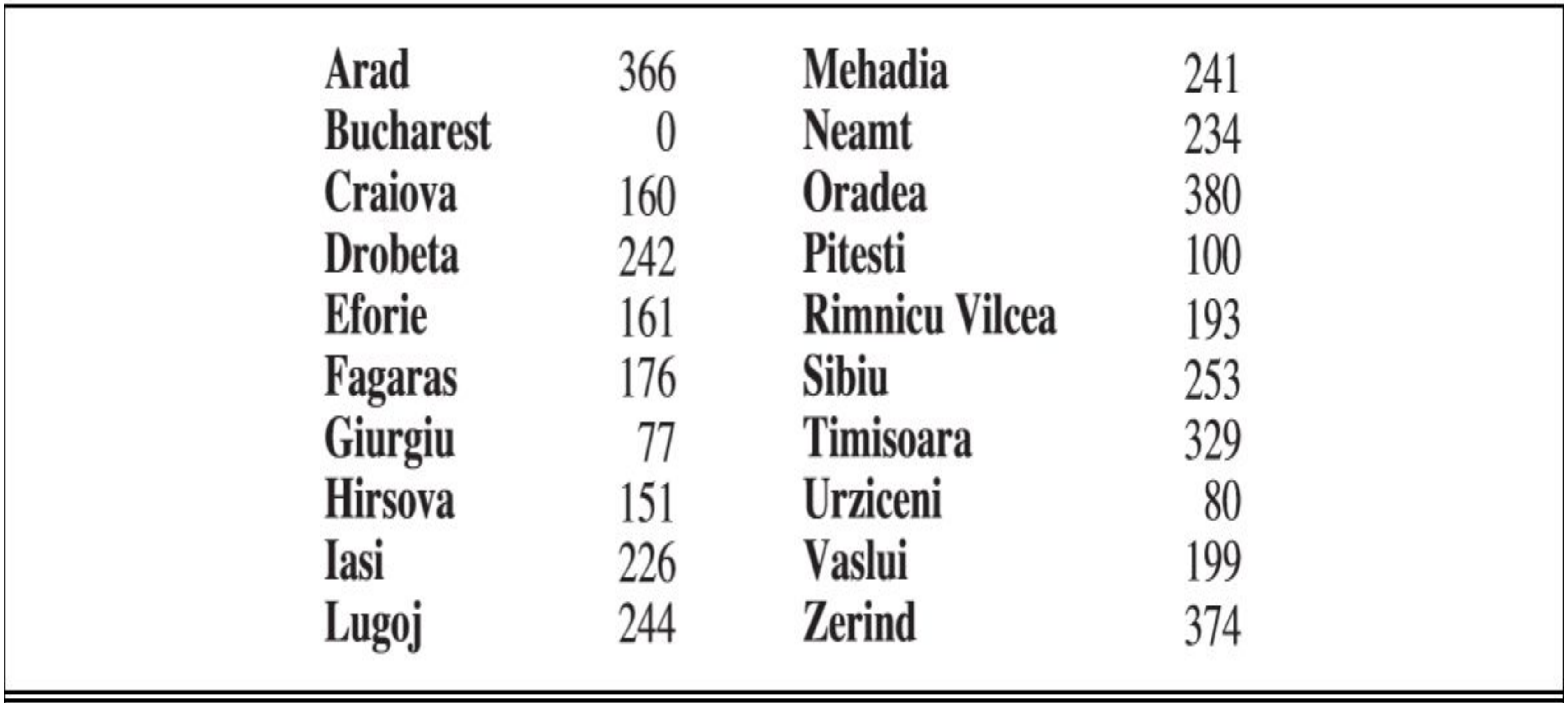
Comparing uninformed search strategies



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.

SLD



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.

Search Strategies v2

- Blind Search
- Informed Search
  - Best-first search
  - A* search
  - Iterative deepening A* search
  - Local search
- Constraint Satisfaction
- Adversary Search

Informed Search

- 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.
- The general approach we consider is called 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 constructed as a cost estimate, so the node with the lowest evaluation is expanded first.
- 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.

Best-first Search

- Generalization of breadth first search
- Priority queue of nodes to be explored
- Cost function f(n) applied to each node

```
Add initial state to priority queue
While queue not empty
    Node = head(queue)
        If goal?(node) then return node
        Add children of node to queue
```

Best First Search

- Breadth first = best first
  - With f(n) = depth(n)

- Dijkstra’s Algorithm = best first
  - With f(n) = g(n), i.e. the sum of edge costs from start to n
  - Space bound (stores all generated nodes)

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).
- Let us see how this works for route-finding problems in Romania; we use the straight line distance heuristic, which we will call hSLD .
- If the goal is Bucharest, we need to STRAIGHT-LINE DISTANCE know the straight-line distances to Bucharest, which are shown in Figure .
- For example, hSLD (In(Arad)) = 366.

Greedy Best First Search

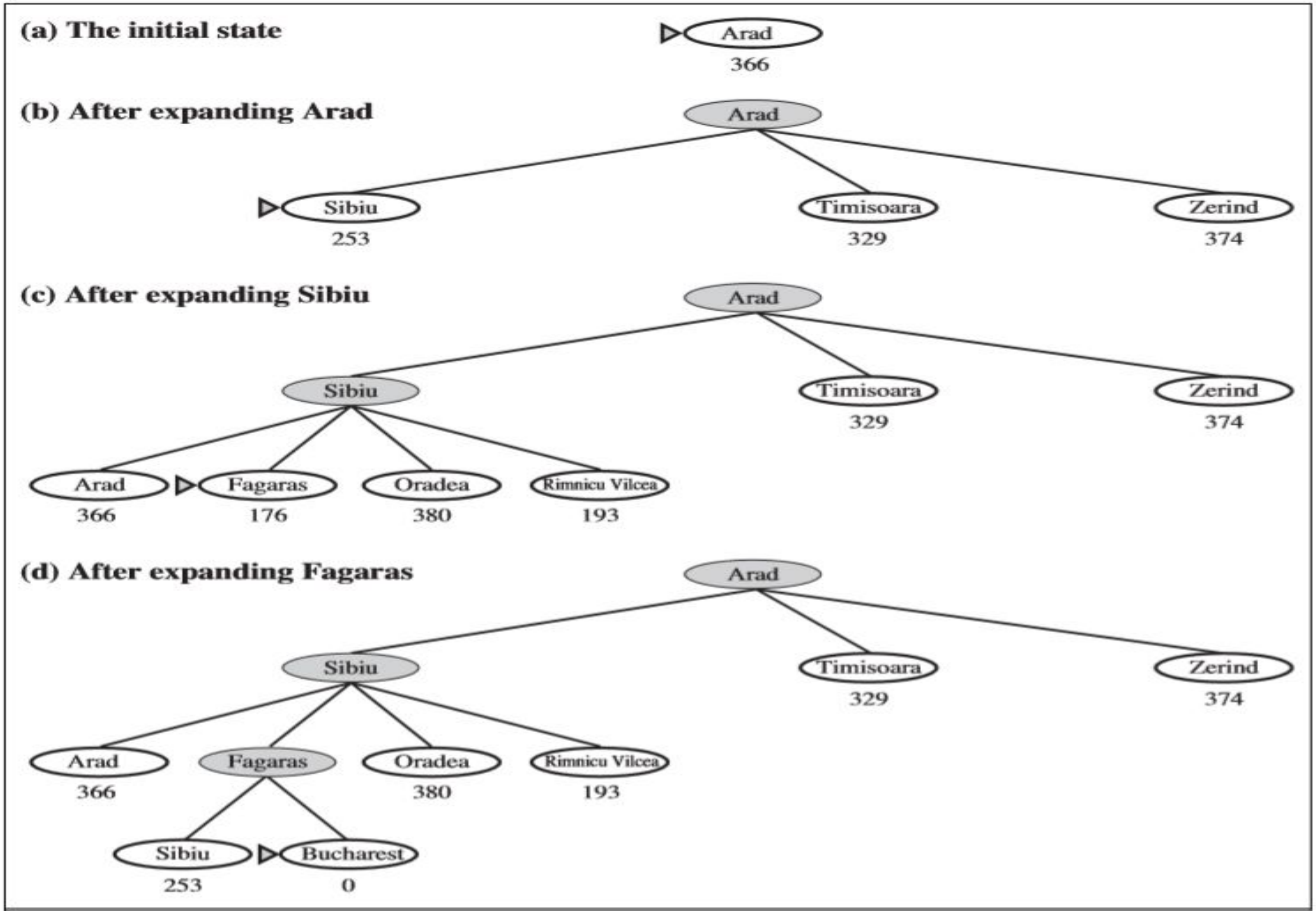


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* search: Minimizing the total estimated solution cost

- The most widely known form of best-first search is called A∗ (pronounced “A-star search”).
- 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.
- Thus, 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).

A* Search

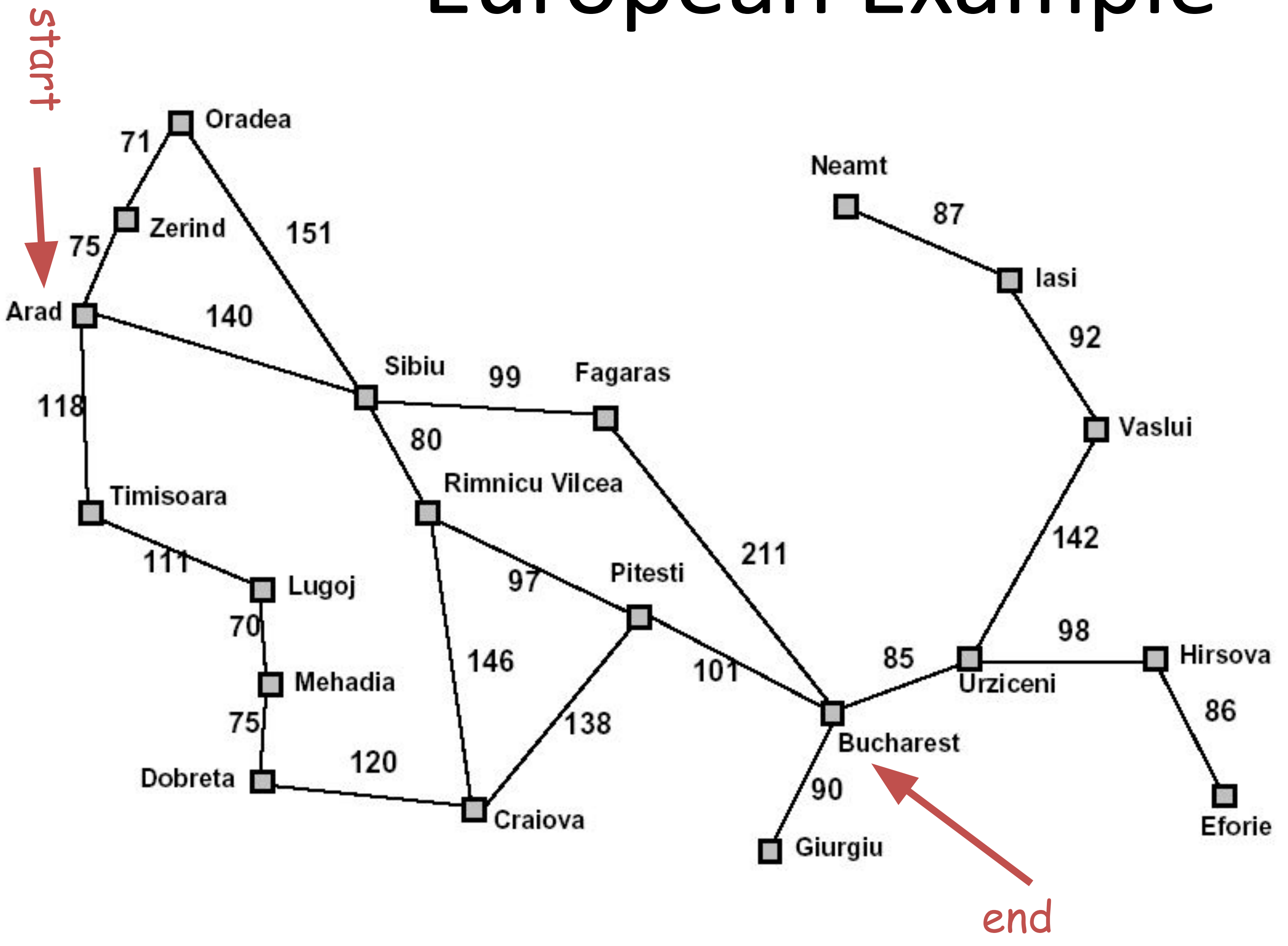
- Hart, Nilsson & Rafael 1968
  - Best first search with f(n) = g(n) + h(n)
  - Where g(n) = sum of costs from start to n
  - h(n) = estimate of lowest cost path n → goal

    h(goal) = 0
  - If h(n) is admissible (and monotonic) then A* is optimal

Underestimates cost of any solution which can reached from node

f values increase from node to descendants (triangle inequality)

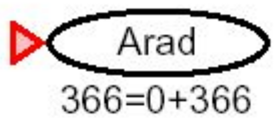
European Example



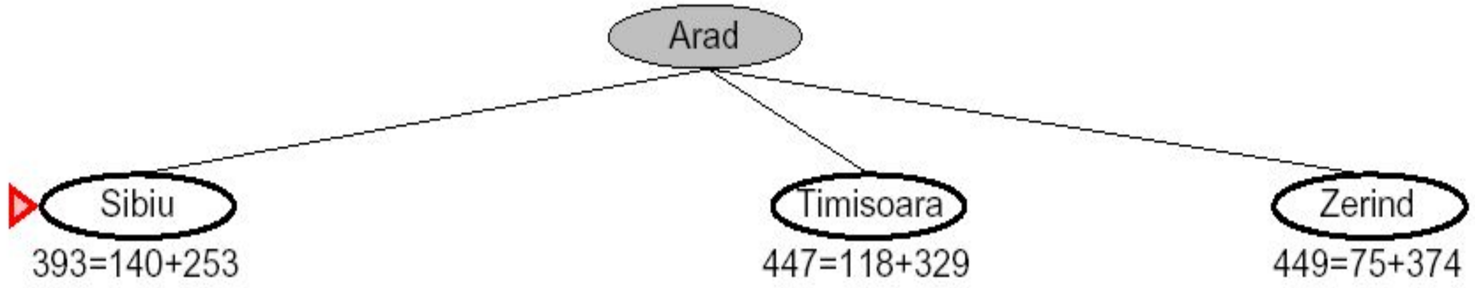
Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

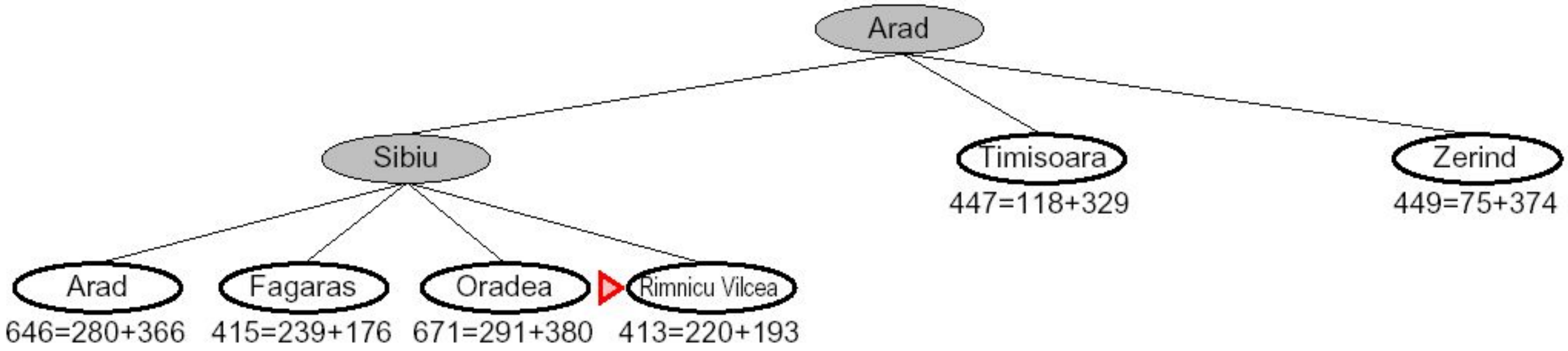
A* Example



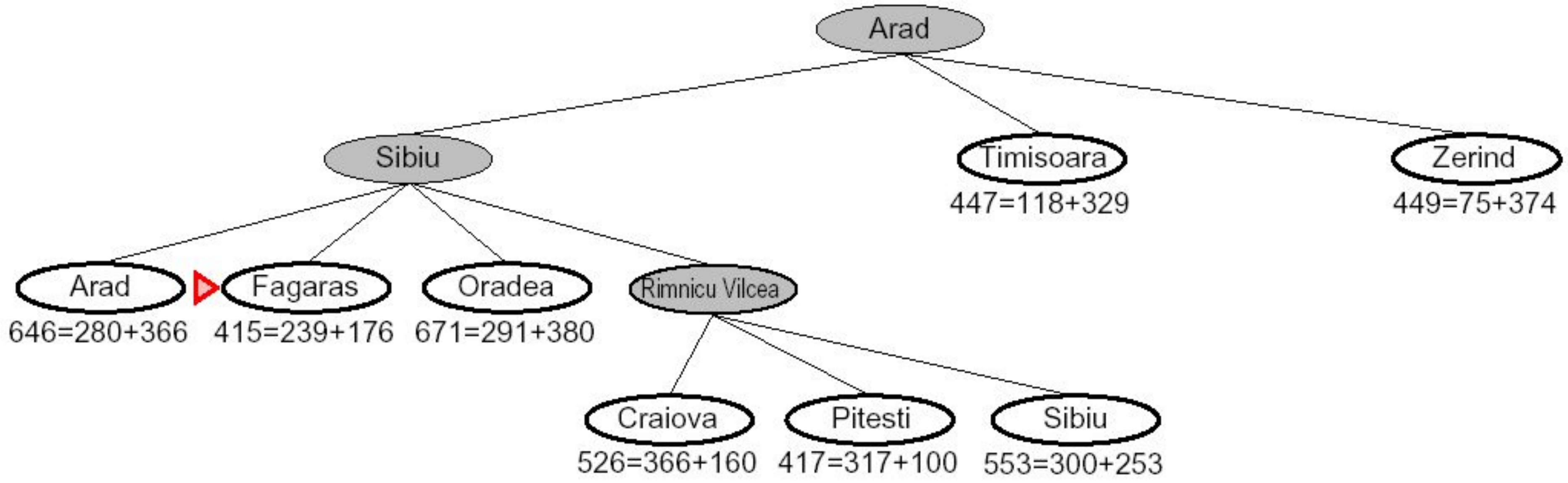
A* Example


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

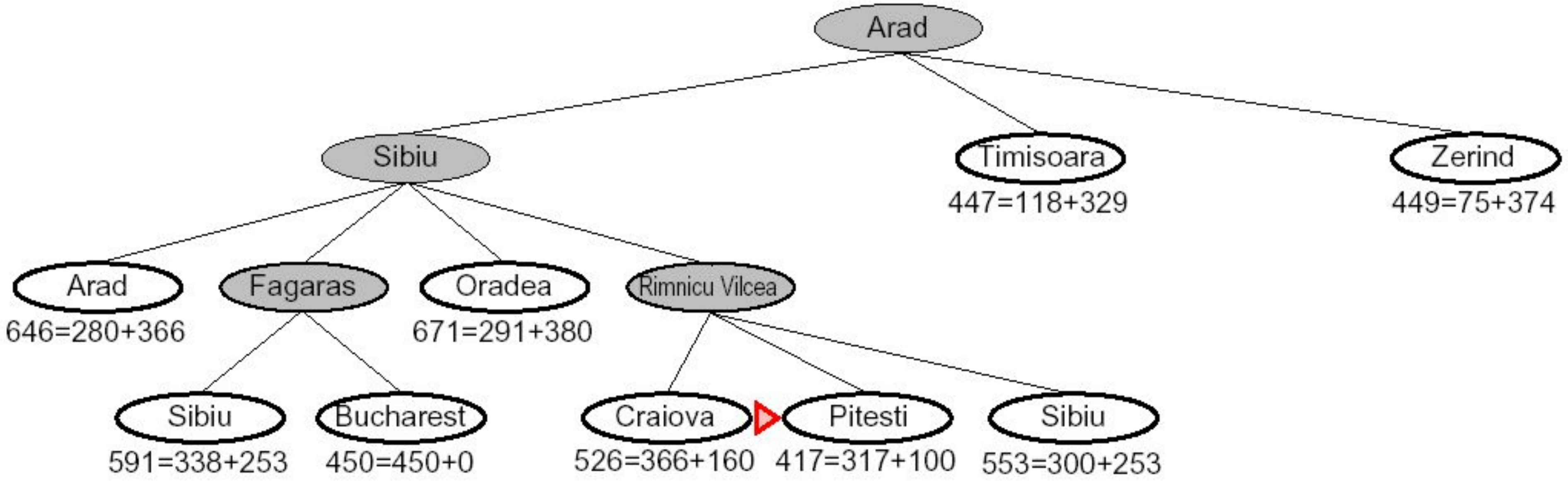
A* Example


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

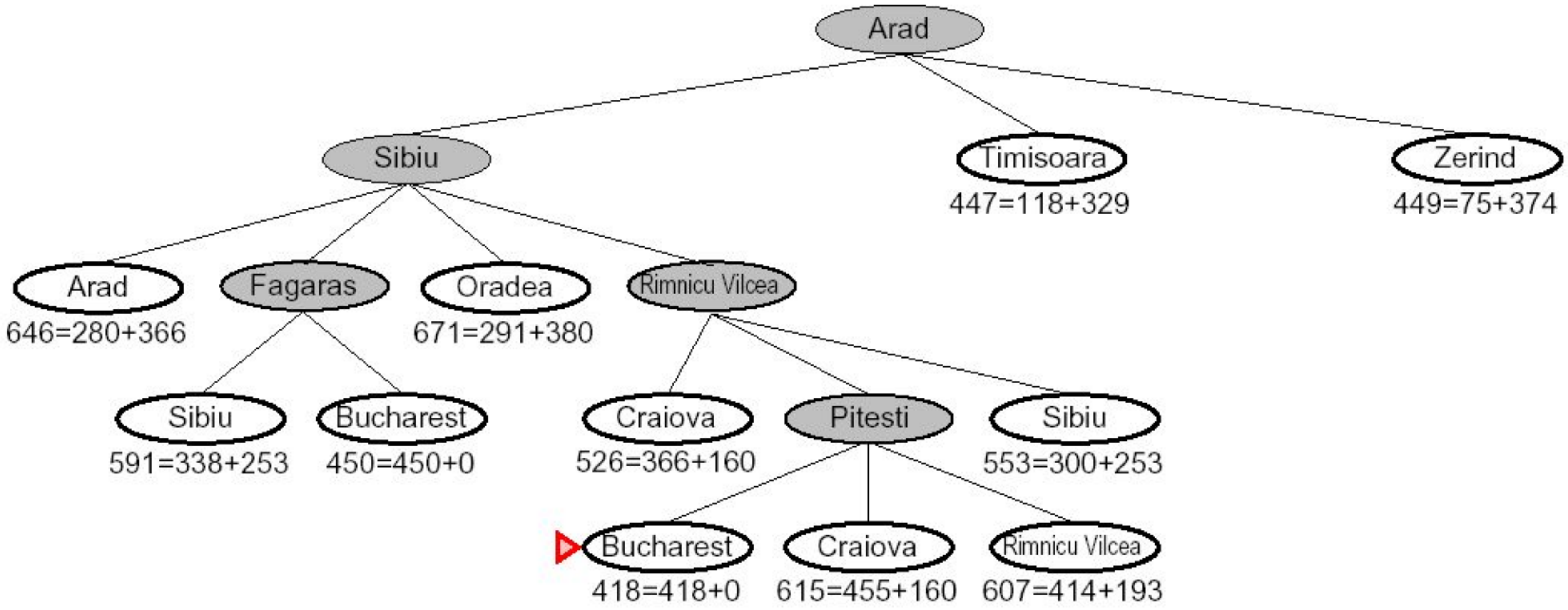
A* Example


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Example


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

A* Example


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

Conditions for optimality: Admissibility and consistency

- 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.
- Because 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. An obvious example of an admissible heuristic is the straight-line distance hSLD that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

Consistency

- A second, slightly stronger condition called consistency (or sometimes monotonicity) 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 :

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

Optimality of A*

- As we mentioned earlier, A∗ has the following properties: 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.
- The first step is to establish the following: 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

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

  $= g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$.
- The next step is to prove that whenever A∗ selects a node n for expansion, the optimal path to that node has been found.

Contours

- The fact that f-costs are non-decreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map. Figure 3.25 shows an example.
- Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on.
- Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

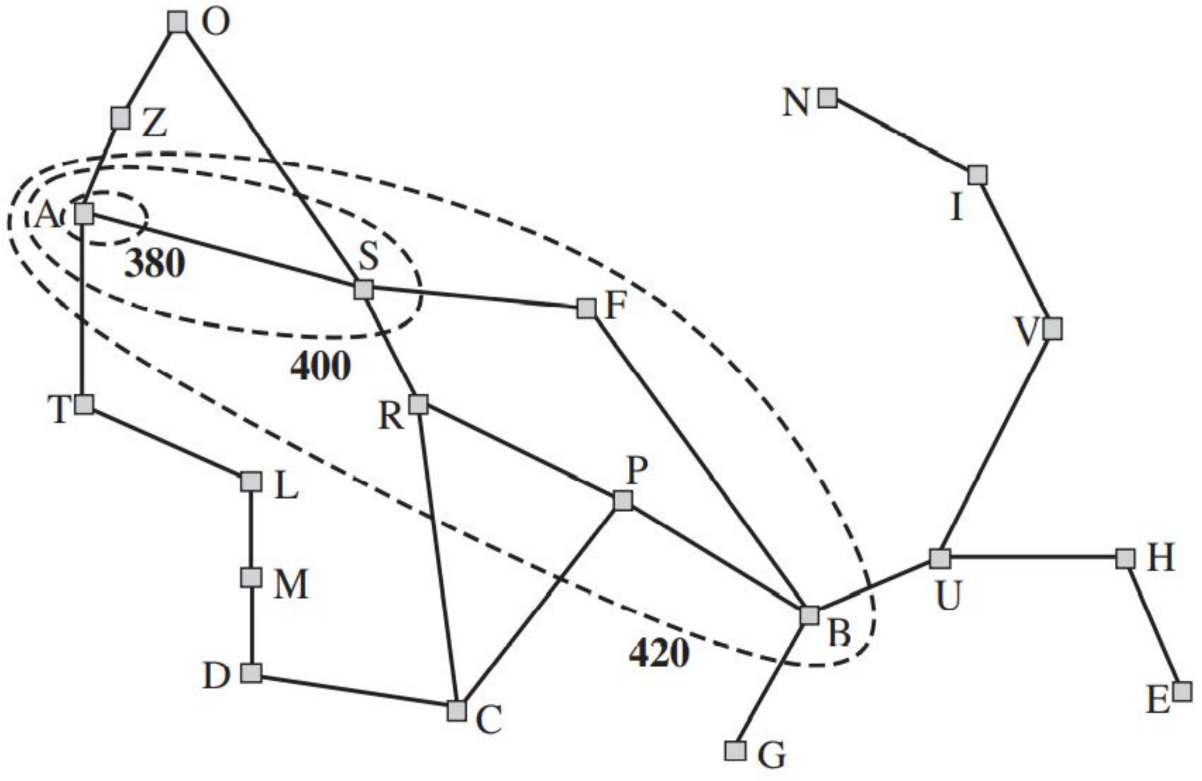


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

Heuristic Function

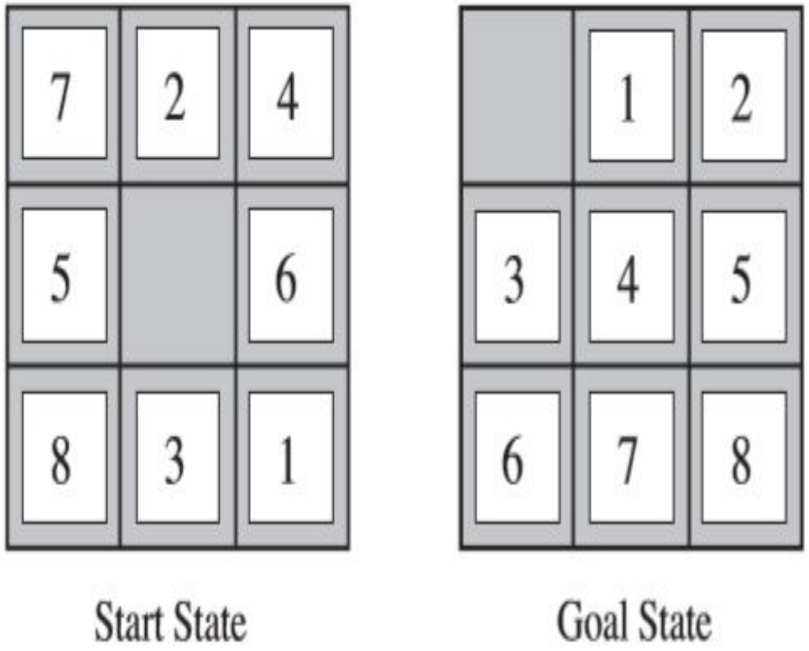


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

Heuristic Function

- h1 = the number of misplaced tiles. 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

Local search Algorithm and optimization Problems

- Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained.
- Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

Local search Algorithm and optimization Problems

- To understand local search, we find it useful to consider the state-space landscape (in Figure 4.1).
- A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum.
- if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum.
- Local search algorithms explore this landscape.
- A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum

Hill Climbing

- The hill-climbing search algorithm (steepest-ascent version) is shown in Figure.
- It is simply a loop that continually moves in the direction of increasing value—that is, uphill.
- It terminates when it reaches a “peak” where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.

Hill Climbing

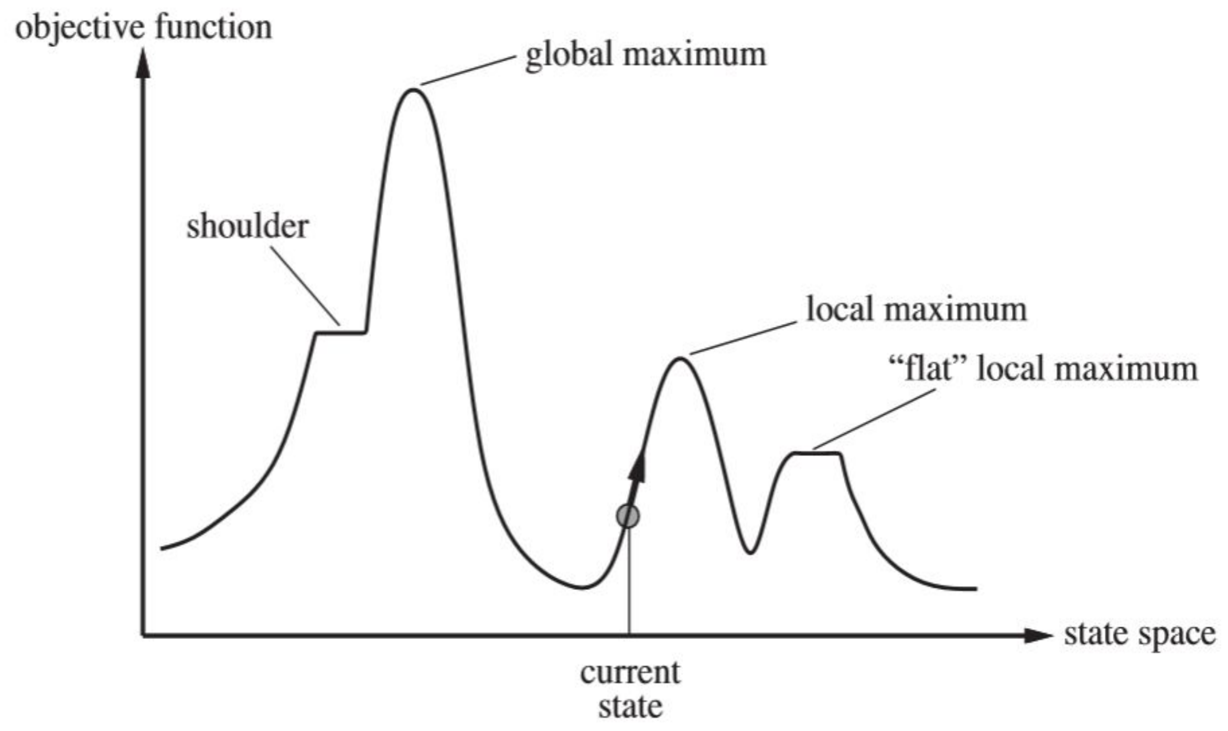
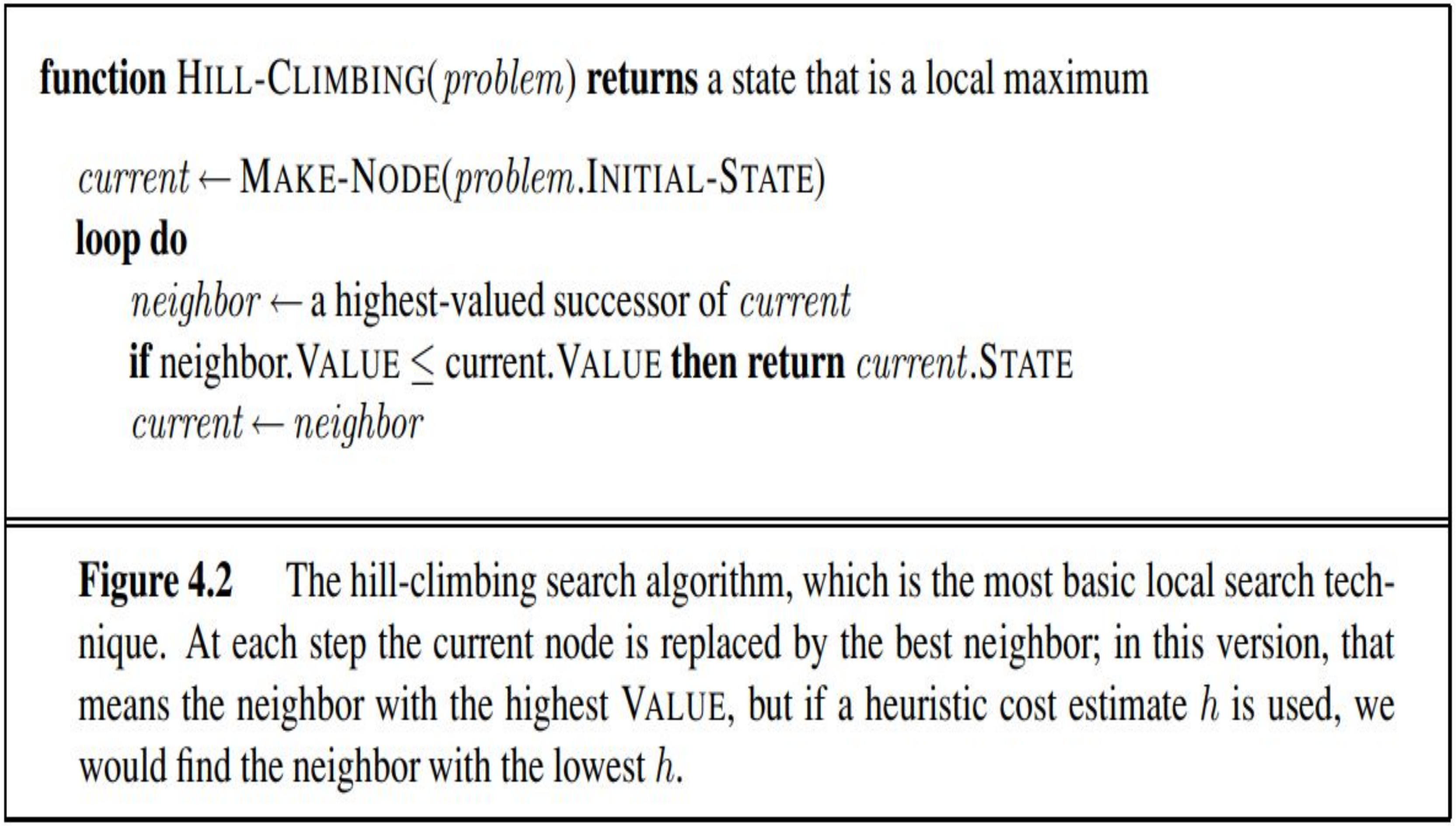


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill Climbing



```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
      neighbor ← a highest-valued successor of current
      if neighbor.VALUE ≤ current.VALUE then return current.STATE
      current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill Climbing

- To illustrate hill climbing, we will use the 8-queens problem.
- Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column.
- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.3(a) shows a state with $h = 17$.
- The figure also shows the values of all its successors, with the best successors having $h = 12$.
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Example

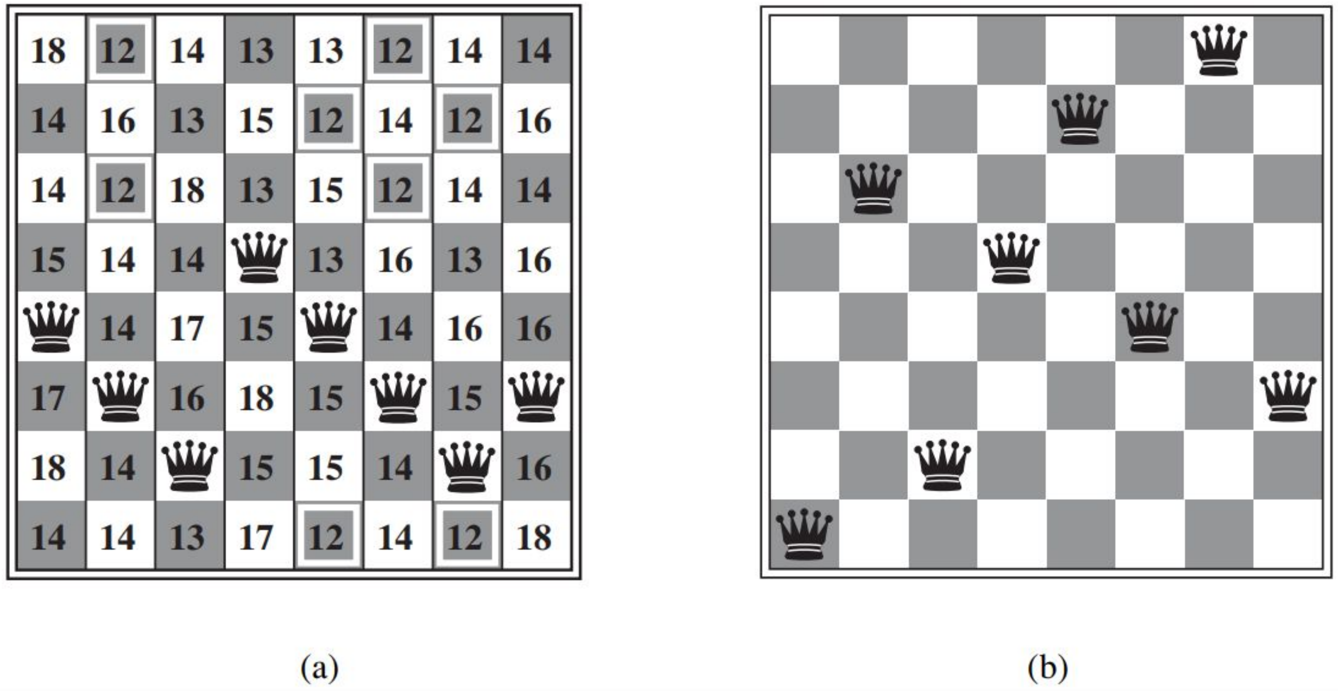


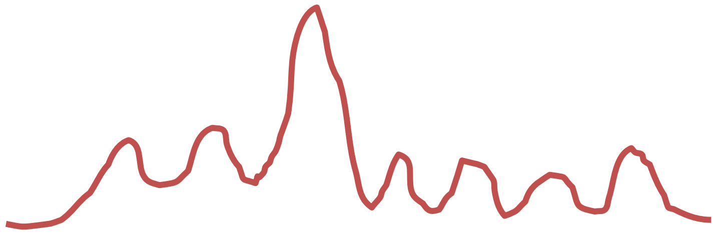
Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill Climbing

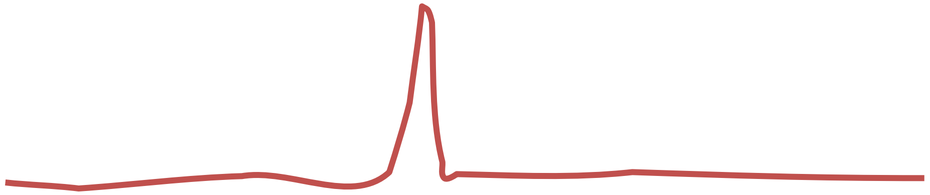
"Gradient ascent"

- Idea
  - Always choose best child; no backtracking
- Problems?

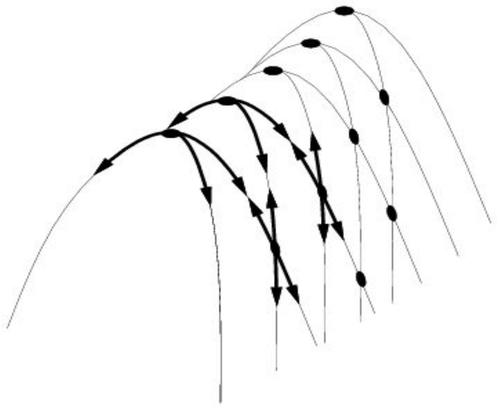
Local maxima



Plateaus



Diagonal ridges



Hill Climbing

- Unfortunately, hill climbing often gets stuck for the following reasons:
- Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- Ridges: a ridge is shown in Figure. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- Plateau: a plateau is a flat area of the state-space landscape.
- It can be a flat local SHOULDER maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.
- A hill-climbing search might get lost on the plateau.

Hill Climbing

- Many variants of hill climbing have been invented. Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- This is a good strategy when a state has many (e.g., thousands) of successors

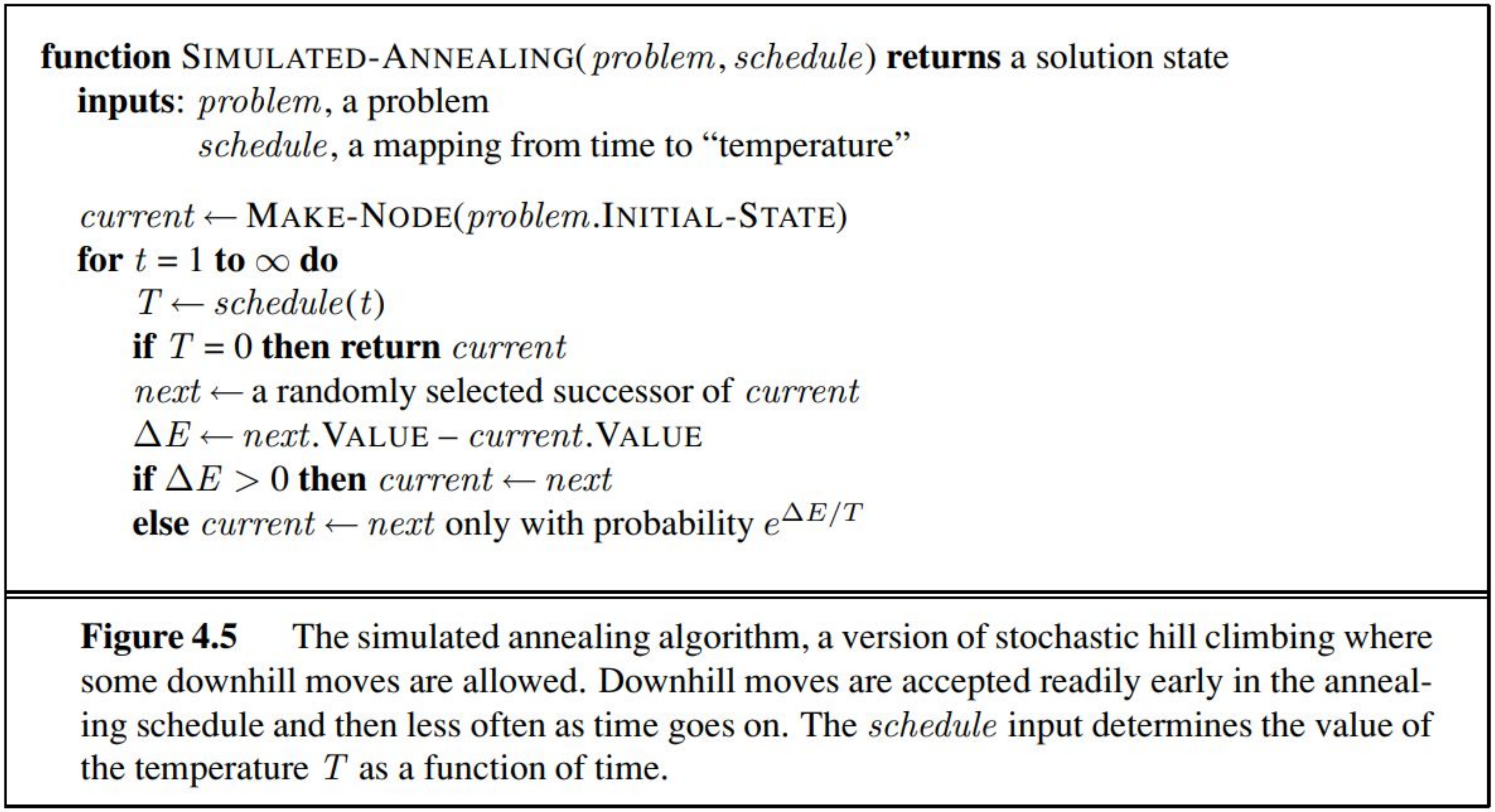
Hill Climbing

- The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima.
- Random-restart hill climbing adopts the well-known adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found

Simulated annealing

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.
- Simulated annealing is such an algorithm. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.
- To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum.
- If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum.
- The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

Simulated Annealing



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to “temperature”

   current ← MAKE-NODE(problem.INITIAL-STATE)
   for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      $\Delta E$ ← next.VALUE – current.VALUE
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Local beam search

- The local beam search algorithm keeps track of k states rather than just one.
- It begins with k randomly generated states.
- At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts.
- Otherwise, it selects the k best successors from the complete list and repeats.
- In a random-restart search, each search process runs independently of the others.
- In a local beam search, useful information is passed among the parallel search threads.

Local beam search

- In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing.
- A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem.
- Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.
- Stochastic beam search bears some resemblance to the process of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

GENETIC ALGORITHM

- A genetic algorithm (or GA)is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.
- Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.

GENETIC ALGORITHM

- The production of the next generation of states is shown in Figure 4.6(b)–(e). In (b), each state is rated by the objective function, or (in GA terminology) the fitness function.
- In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b).

GENETIC ALGORITHM

- In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.
- Finally, in (e), each location is subject to random mutation with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

GENETIC ALGORITHM

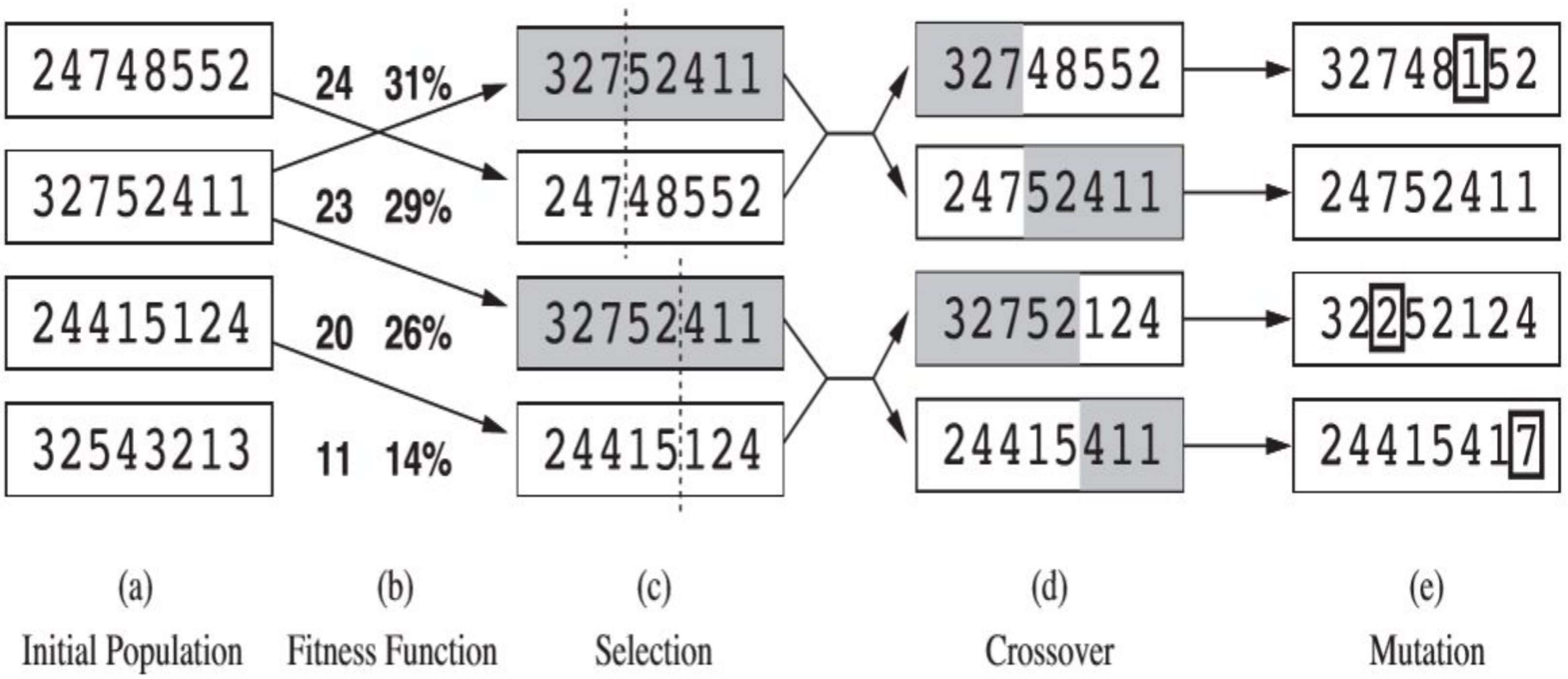


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

GENETIC ALGORITHM

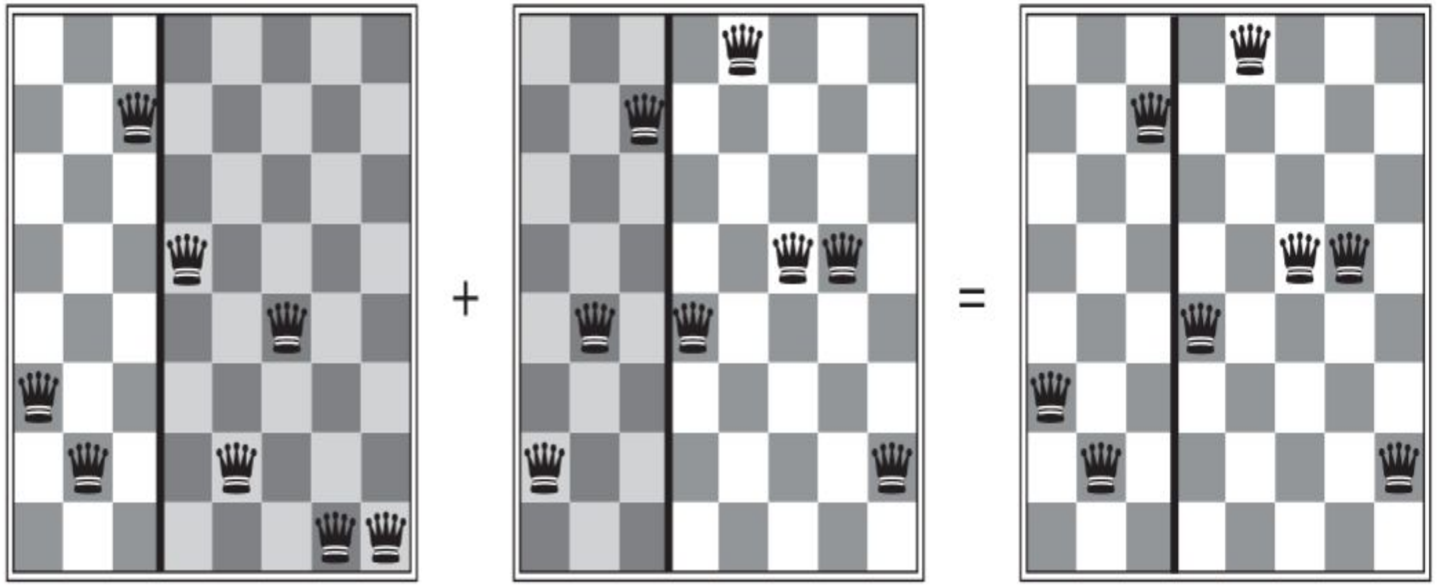
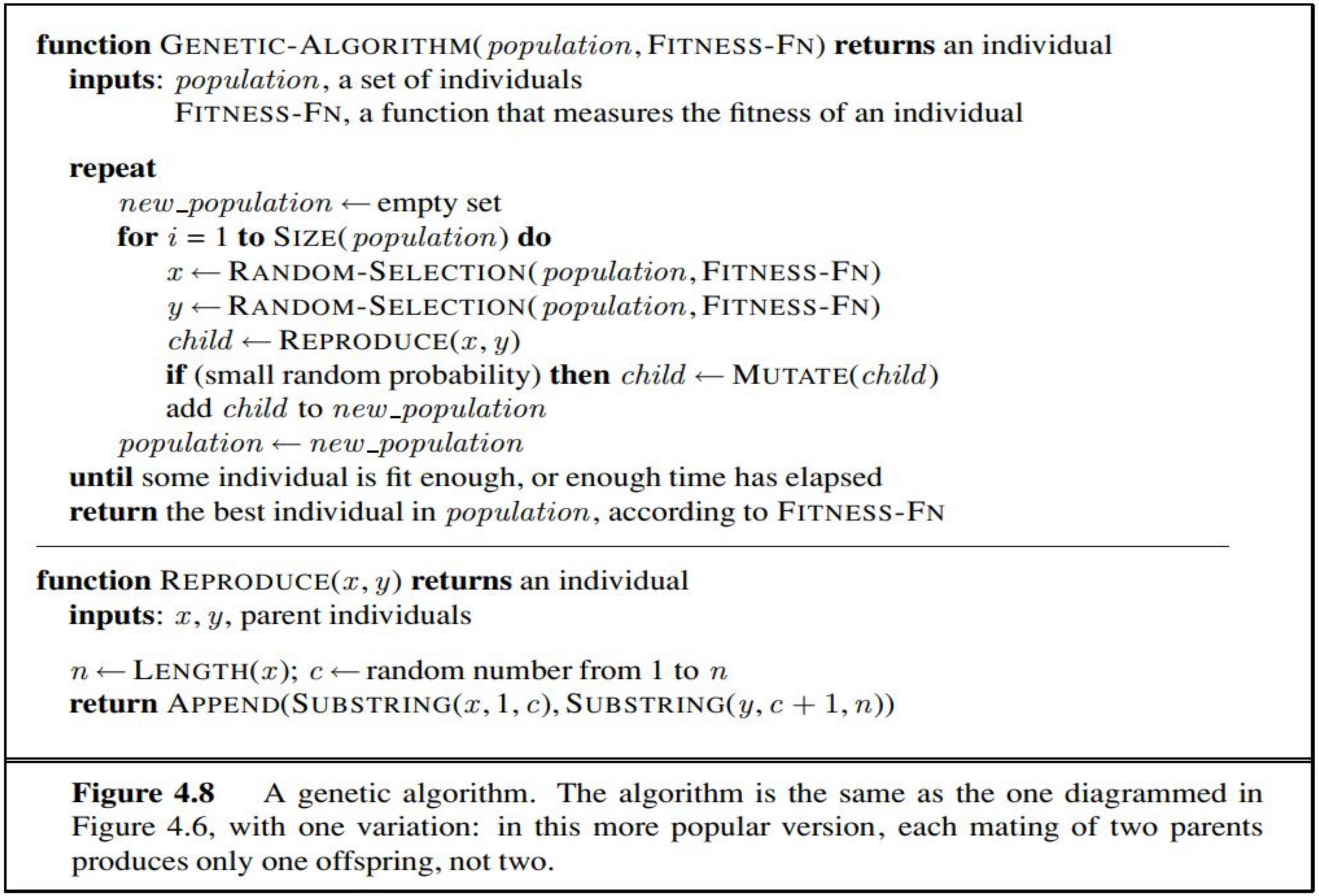


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

Genetic Algorithm



```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for $i$ = 1 to SIZE(population) do
        $x$ ← RANDOM-SELECTION(population, FITNESS-FN)
        $y$ ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE($x$, $y$)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE($x$, $y$) returns an individual
  inputs: $x$, $y$, parent individuals

  $n$ ← LENGTH($x$); $c$ ← random number from 1 to $n$
  return APPEND(SUBSTRING($x$, 1, $c$), SUBSTRING($y$, $c+1$, $n$))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.