ARTIFICIAL INTELLIGENCE

UNIT – II

<u>Syllabus</u>

Solving Problems by searching: Problem Solving Agents, Example problems, Searching for Solutions, Uninformed Search Strategies, Informed search strategies, Heuristic Functions.

Beyond Classical Search: Local Search Algorithms and Optimization Problems, Local Search in Continues Spaces, Searching with Nondeterministic Actions, Searching with partial observations, online search agents and unknown environments.

Chapter – 1

Solving Problems by searching

2.1.1. Problem Solving Agents :

Problem formulation is the process of deciding what actions and states to consider, given a goal .

- ✓ An important aspect of intelligence is goal-based problem solving.
- ✓ The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal.
- ✓ Each action changes the state and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

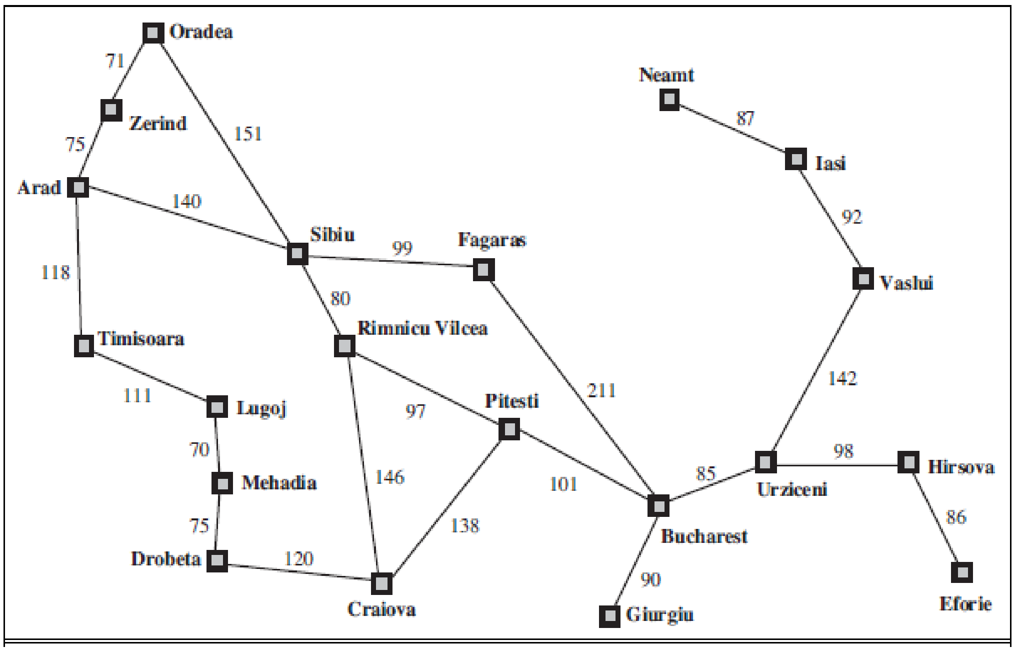
➢ **What is Search?**

- ✓ Search is the systematic examination of states to find path from the start/root state to the goal state.
- ✓ The set of possible states, together with operators defining their connectivity constitute the search space.
- ✓ The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

(i) Problem Solving Agents:

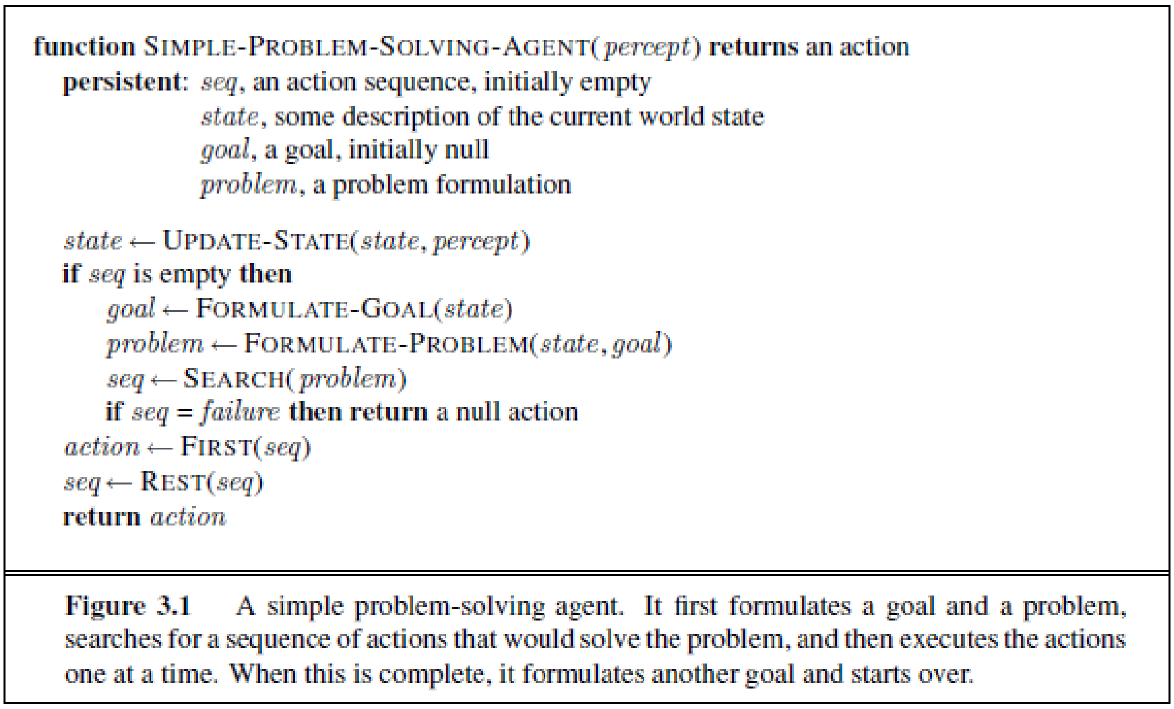
- ✓ A Problem solving agent is a goal-based agent .It decide what to do by finding sequence of actions that lead to desirable states.
- ✓ The agent can adopt a goal and aim at satisfying it.

Example : From Arad(Romania) to Bucharest



- ✓ Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
- ✓ The agent's task is to find out which sequence of actions will get to a goal state.
- ✓ Problem formulation is the process of deciding what actions and states to consider given a goal.
- ✓ An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence.
- ✓ The process of looking for sequences actions from the current state to reach the goal state is called search.
- ✓ The search algorithm takes a problem as input and returns a solution in the form of action sequence.

- ✓ Once a solution is found, the execution phase consists of carrying out the recommended action..
- ✓ Simple “formulate, search, execute” design for the agent…



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

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

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

➢ **The agent design assumes the Environment is:**

– Static : The entire process carried out without paying attention to changes that might be occurring in the environment.

– Observable : The initial state is known and the agent’s sensor detects all aspects that are relevant to the choice of action.

– Discrete : With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken.

– Deterministic : The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions.

(ii) Well-defined problems and solutions:

A well-defined problem can be described by:

i. Initial state

ii. Operator or successor function - for any state x returns s(x), the set of states reachable from x with one action

iii. State space - all states reachable from initial by any sequence of actions

iv. Path - sequence through state space

v. Path cost - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path

vi. Goal test - test to determine if at goal state

vii. The step cost of taking action ‘a’ to go from state x to state y is denoted by c(x,a,y). The step cost for Romania are shown in next figure in route distances. It is assumed that the step costs are non negative.

viii.A solution to the problem is a path from the initial state to a goal state.

ix. An optimal solution has the lowest path cost among all solutions.

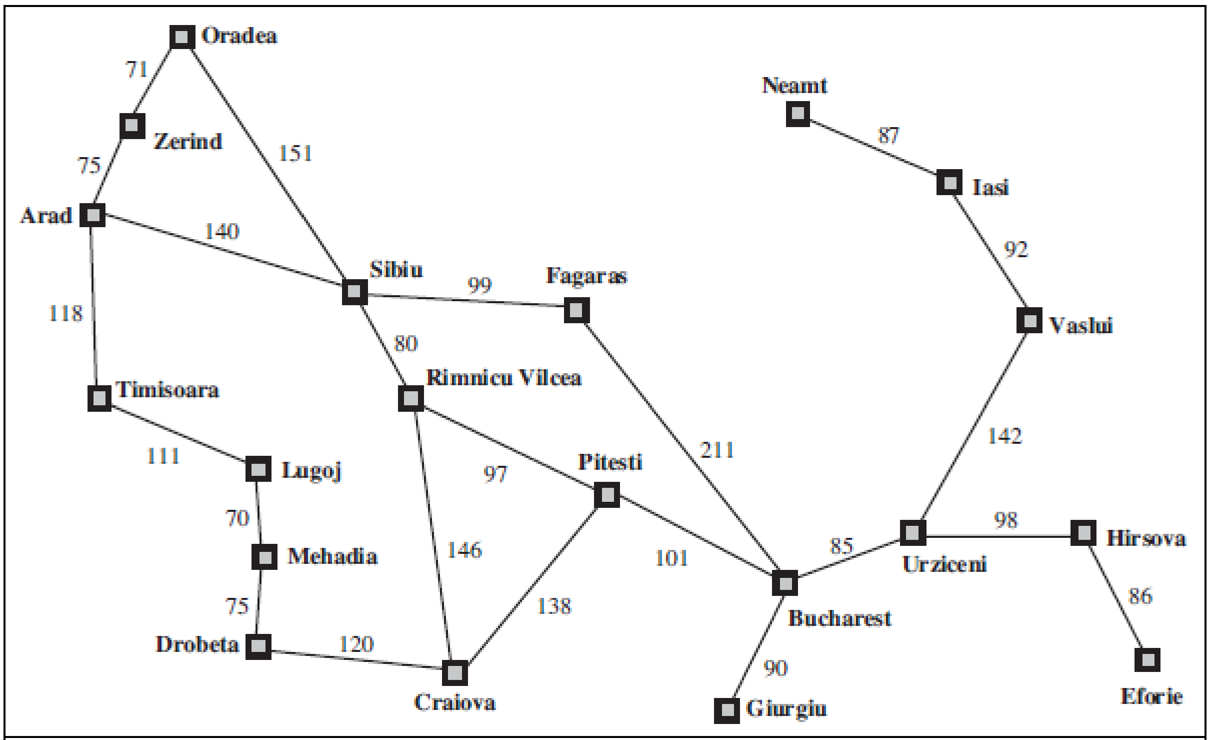


Figure 3.2 A simplified road map of part of Romania.

2.1.2. Example Problems:

- **Toy vs Real World problems…**

✓ A toy problem is intended to illustrate or exercise various problem-solving methods.

✓ It can be given a concise, exact description and hence is usable by different researchers to compare the REALWORLD performance of algorithms.

✓ A real-world problem is one whose solutions people actually PROBLEM care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

1. Toy Problems:

- **Example 1: Vaccum world**

i. **States:** The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x $2^2 = 8$ possible world states.

ii. **Initial state:** Any state can be designated as initial state.

iii. **Successor function :** This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure.

iv. **Goal Test :** This tests whether all the squares are clean.

v. **Path test :** Each step costs one ,so that the the path cost is the number of steps in the path.

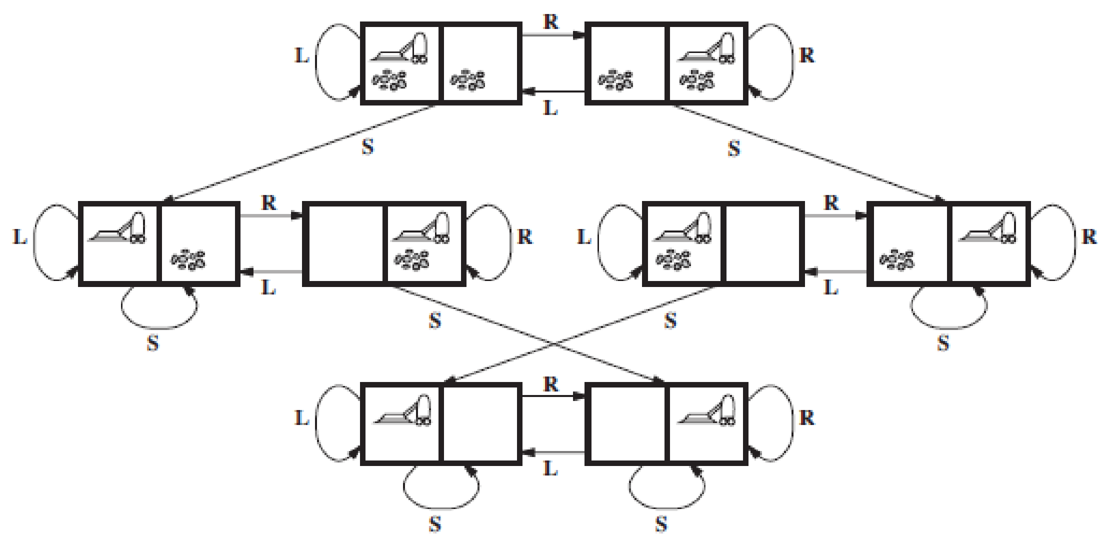
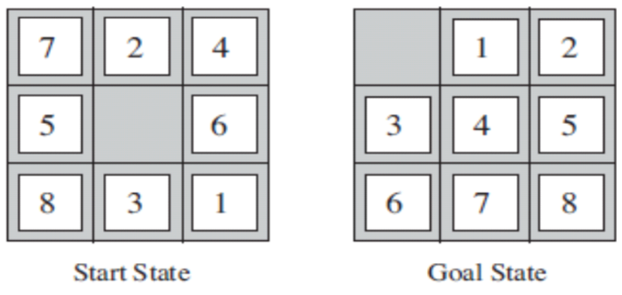


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

- ✓ **States?:** integer dirt and robot locations (ignore dirt amounts etc.)
- ✓ **Actions?:** Left, Right, Suck, NoOp
- ✓ **Goal test?:** no dirt
- ✓ **Path cost?:** 1 per action (0 for NoOp)

Example 2:The 8-puzzle:

- ✓ An 8-puzzle consists of a 3x3 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 the goal state ,as shown in Figure.



A typical instance of the 8-puzzle.

➢ **The problem formulation is as follows :**

i. **States :** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

ii. **Initial state :** Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.

iii. **Successor function :** This generates the legal states that result from trying the four actions(blank moves Left,Right,Up or down).

iv. **Goal Test :** This checks whether the state matches the goal configuration shown in figure(Other goal configurations are possible).

v. **Path cost :** Each step costs 1,so the path cost is the number of steps in the path.

- ✓ The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.
- ✓ The 8-puzzle has 9!/2 = 181,440 reachable states and is easily solved.
- ✓ The 15 puzzle (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- ✓ The 24-puzzle (on a 5 x 5 board) has around 1025 states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

i. **States?:** Integer locations of tiles (ignore intermediate positions)

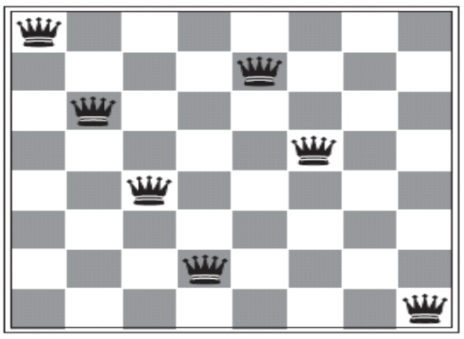
ii. **Actions?:** Move blank left, right, up, down (ignore unjamming etc.)

iii. **Goal test?:** Goal state (given).

iv. **Path cost?**: 1 per move.
[Note: optimal solution of n-Puzzle family is NP-hard]

➢ **Example 3: 8 Queen Problem:**

- ✓ The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row, column or diagonal).
- ✓ An Incremental formulation involves operators that augments the state description, starting with an empty state for 8-queens problem, this means each action adds a queen to the state.
- ✓ A complete-state formulation starts with all 8 queens on the board and move them around.
- ✓ In either case the path cost is of no interest because only the final state counts.



➢ **The first incremental formulation one might try is the following :**

i. **States :** Any arrangement of 0 to 8 queens on board is a state.

ii. **Initial state :** No queen on the board.

iii. **Successor function :** Add a queen to any empty square.

iv. **Goal Test :** 8 queens are on the board, none attacked.

- ✓ In this formulation, we have 64, 63…57 = 3 x 1014 possible sequences to investigate.
- ✓ A better formulation would prohibit placing a queen in any square that is already attacked.

- ✓ States : Arrangements of n queens (0 <= n < = 8), one per column in the left most columns ,with no queen attacking another are states.
- ✓ Successor function : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.
- ✓ This formulation reduces the 8-queen state space from 3 x10^{14} to just 2057,and solutions are easy to find.

2. Real-world Problems:

i. Route finding problem

ii. Airline Travel problem

iii. Touring problems

iv. Travelling salesperson problem

v. VLSI Layout

vi. ROBOT Navigation

vii.Automatic Assembly problem

viii.Internet searching

(i) Route-Finding Problem :

- ▪ **Route-Finding Problem** is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications.
- ▪ Some, such as Web sites and in-car systems that provide driving directions.
- ▪ Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications.
- ➢ Consider the airline travel problems that must be solved by a travel-planning Web site:

• **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 previous segments, 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.

(ii) Touring problems :

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city at least once, starting and ending in Bucharest".

(iii) Traveling Salesperson Problem (TSP) :

- 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.

(iv) VLSI layout problem :

- 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**

  → **Channel routing**.

➔ **Cell layout :**

- In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function.

→ **Channel routing :**

- Channel routing finds a specific route for each wire through the gaps between the cells.

(v) Robot Navigation :

- **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.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional.

(vi) Automatic assembly sequencing :

- **Automatic assembly sequencing** of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972).
- Progress since then has been slow but sure.
- In assembly problems, the 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.
- Another important assembly problem is **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.

2.1.3. 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.
- The diagram shows the first few steps in growing the search tree for finding a route from Arad to Bucharest.
- By **expanding** the current state , and thereby **generating** a new set of states.

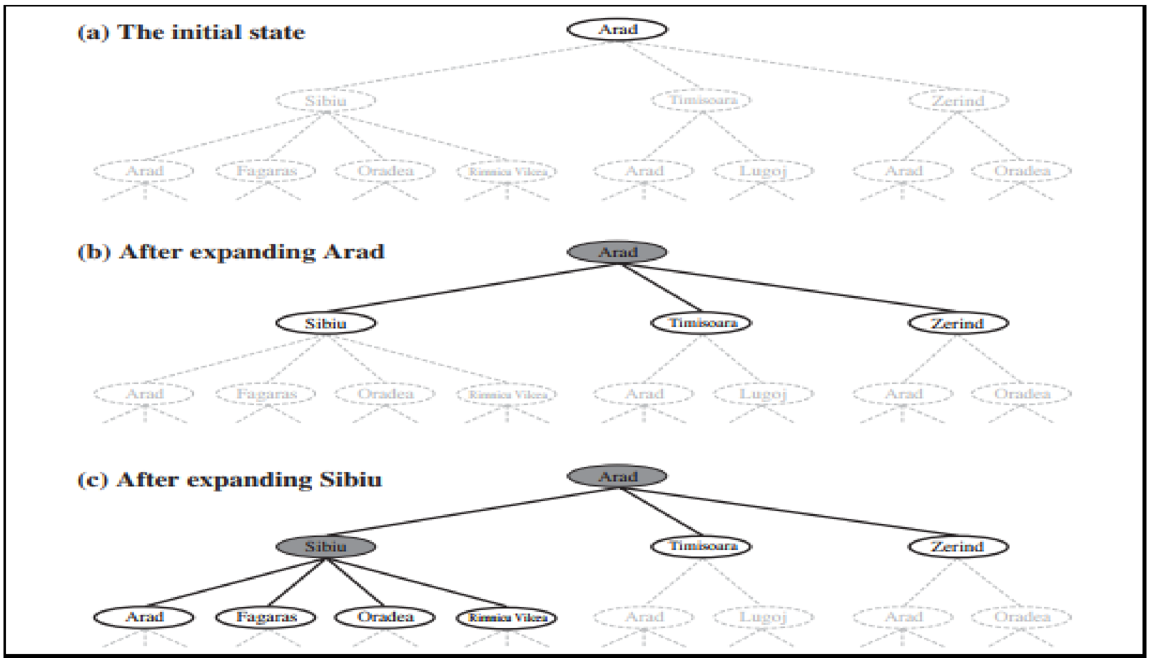


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

- 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**(Many authors call it the **open list)**

  the frontier of each tree consists of those nodes with bold outlines.

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

- Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

```
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.

- 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.

- 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.

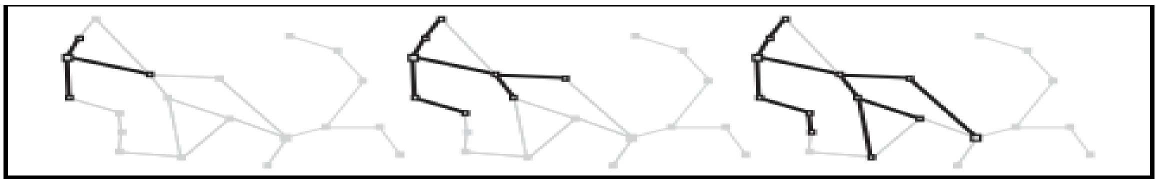


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

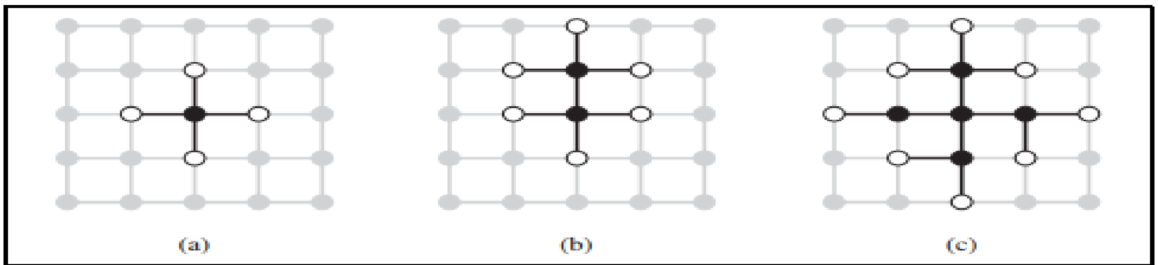


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

(i) 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.

➢ Queue :

- Now that we have nodes, we need somewhere to put them.
- 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.

- Queues are characterized by the *order* in which they store the inserted nodes.
- Three common variants are
- The first-in, first-out or **FIFO queue**, which pops the *oldest* element of the queue;
- LIFO QUEUE the last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element
- PRIORITY QUEUE of the queue; and the **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

(ii) Measuring problem-solving performance:

- Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them.
- 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?

2.1.4. Uninformed Search strategies :

- ✓ Uninformed Search Strategies have no additional information about states beyond that provided in the problem definition.

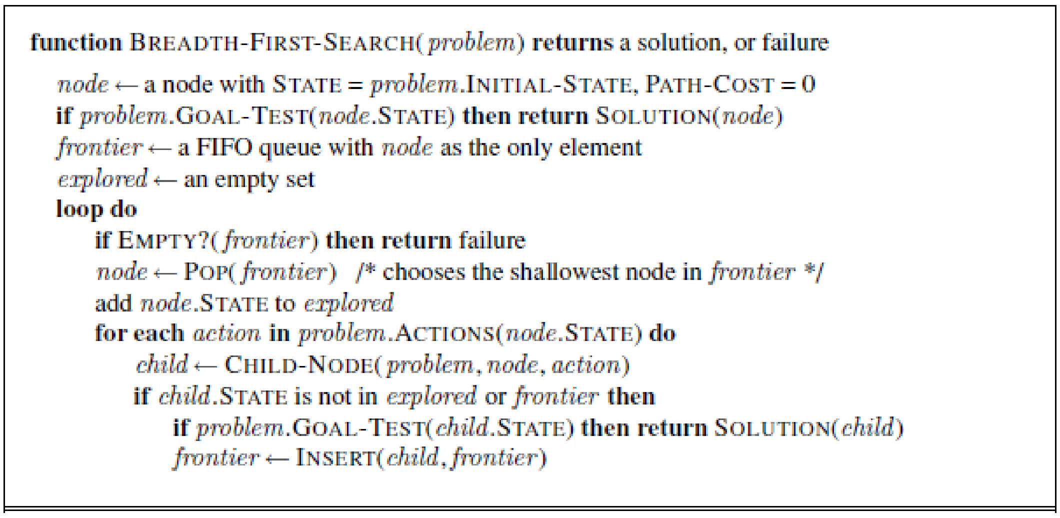
➢ **There are six uninformed search strategies as given below:**

i. Breadth-first search
ii. Uniform-cost search
iii. Depth-first search
iv. Depth-limited search
v. Iterative deepening search
vi. Bi-directional search

1. Breadth-first search(BFS):

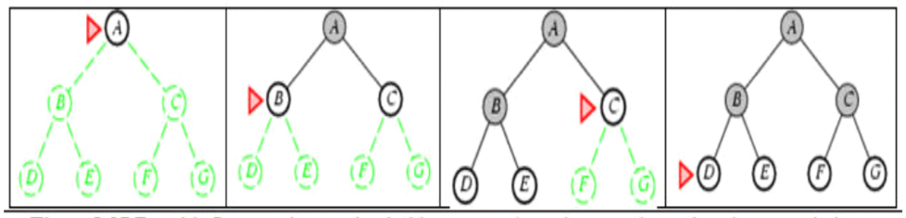
- ✓ Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- ✓ In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- ✓ Breath-First-Search is implemented by calling TREE SEARCH with an empty fringe that is a first-in-first out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- ✓ In other words, calling TREE-SEARCH (problem, FIFO-QUEUE()) results in breadth-first-search.
- ✓ The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

➢ **BFS Algorithm…**

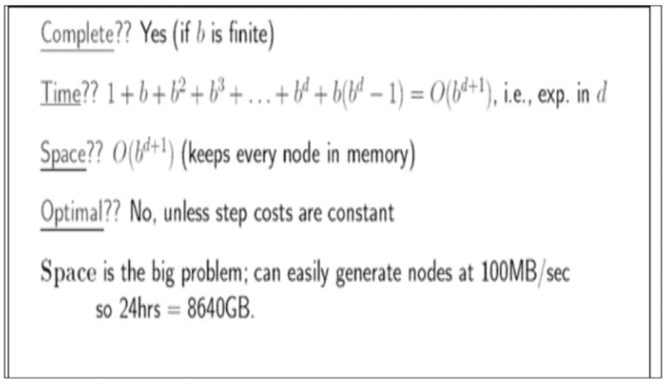


```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if 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)
```

Example for BFS Algorithm…



- **Properties of BFS…**



Complete?? Yes (if b is finite)

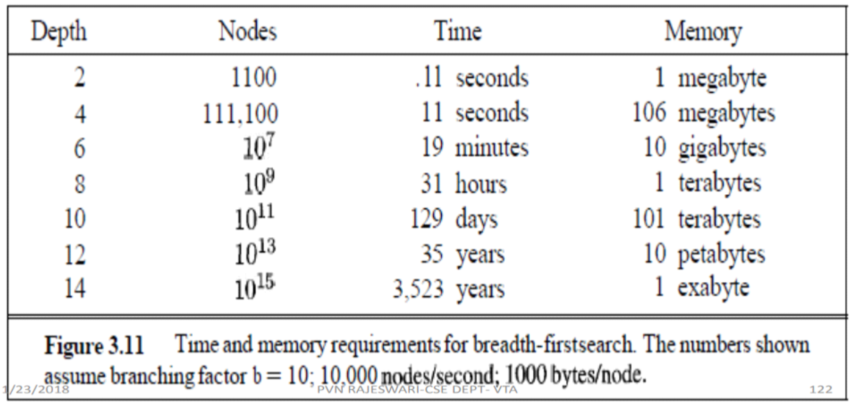
Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

- **Time and memory requirements….**



Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

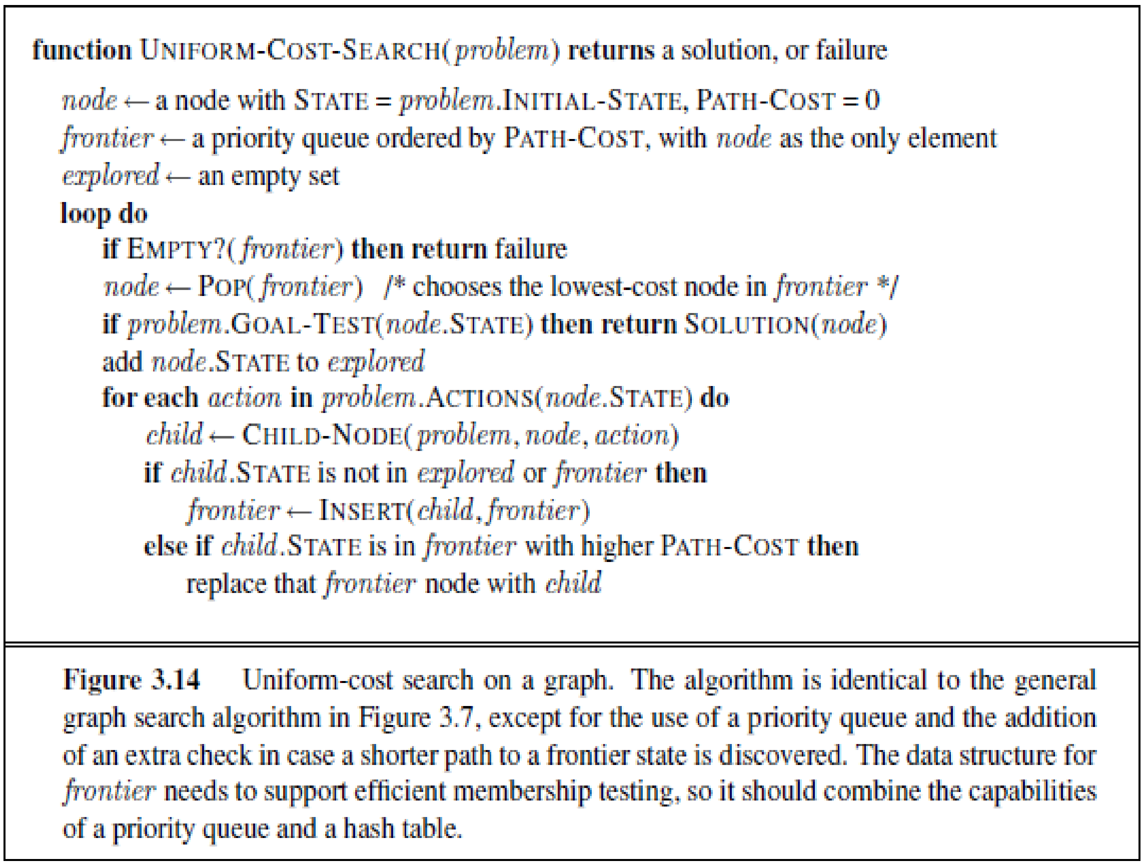
Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor b = 10; 10,000 nodes/second; 1000 bytes/node.

1/23/2018 PVN RAJESWARI-CSE DEPT- VTA 122

2. Uninform-cost Search(UCS)…

- ✓ Instead of expanding the shallowest node, uniform-cost search expands the node 'n' with the lowest path cost.
- ✓ Uniform-cost search does not care about the number of steps a path has, but only about their total cost.
- ✓ Expand least-cost unexpanded node.
- ✓ We think of this as having an evaluation function:g(n), which returns the path cost to a node n.
- ✓ fringe = queue ordered by evaluation function,lowest first.
- ✓ Equivalent to breadth-first if step costs all equal
- ✓ Complete and optimal.
- ✓ Time and space complexity are as bad as for breadth-first search.

- **UCS Algorithm:**



```
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
```

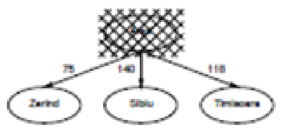
Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support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

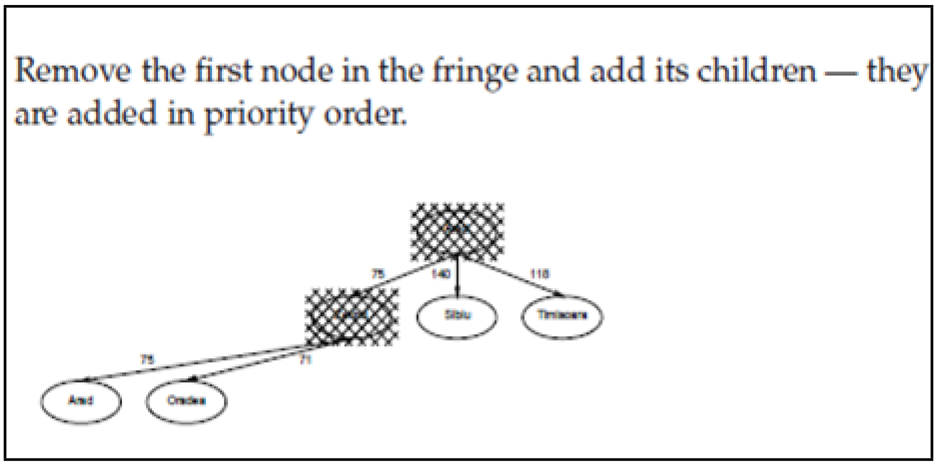
- **Process…**

- Add the node representing the initial state into the fringe.

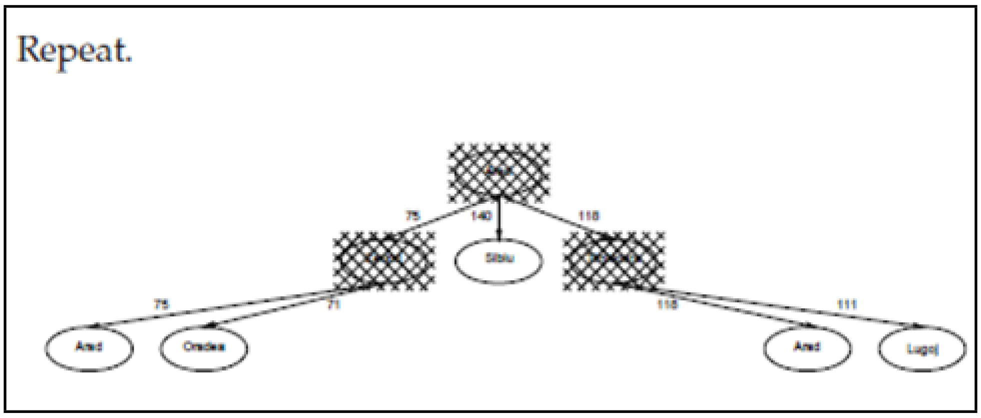
Remove the first node in the fringe and add its children

The queue is ordered with the cheapest first.



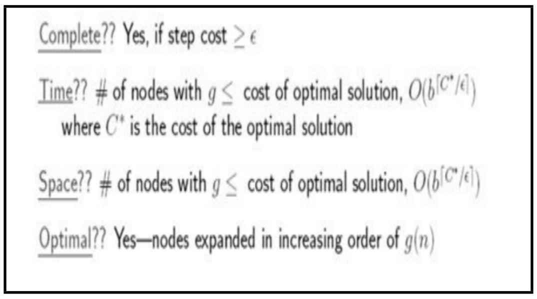


Remove the first node in the fringe and add its children — they are added in priority order.



Repeat.

➢ **Properties of UCS….**



Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

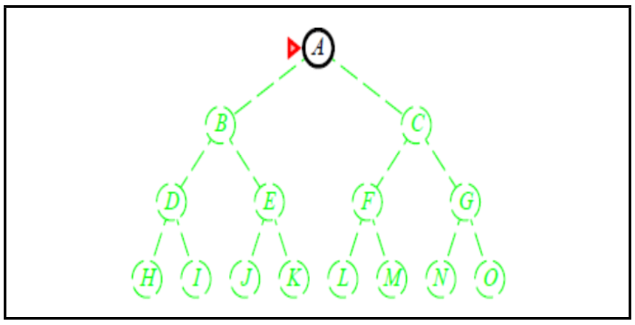
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

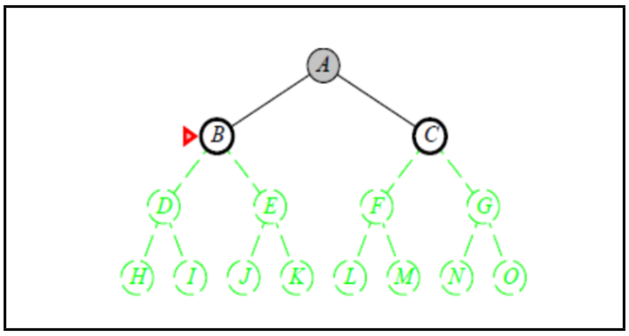
3. Depth First Search(DFS)…

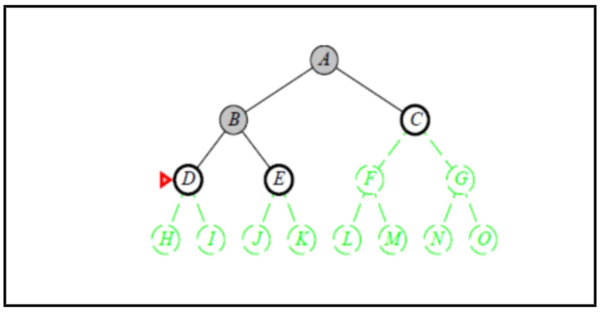
- ✓ Depth-first-search always expands the deepest node in the current fringe of the search tree.
- ✓ The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- ✓ As those nodes are expanded, they are dropped from the fringe ,so then the search "backs up" to the next shallowest node that still has unexplored successors.

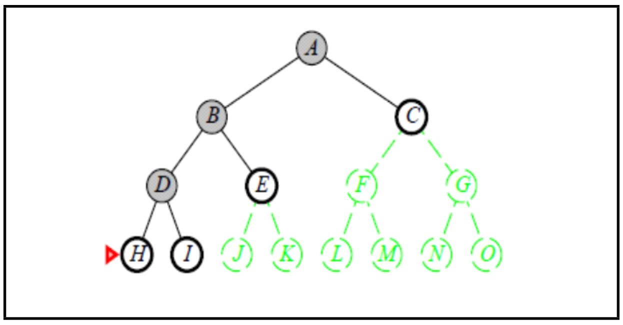
➢ **Implementation:**

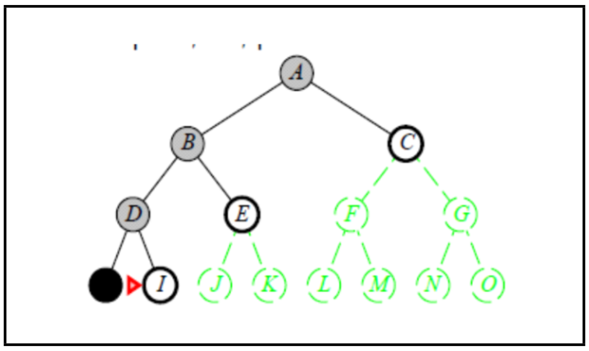
fringe = LIFO queue, i.e., put successors at front.

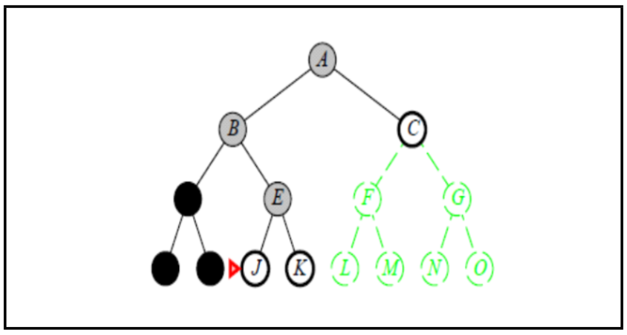


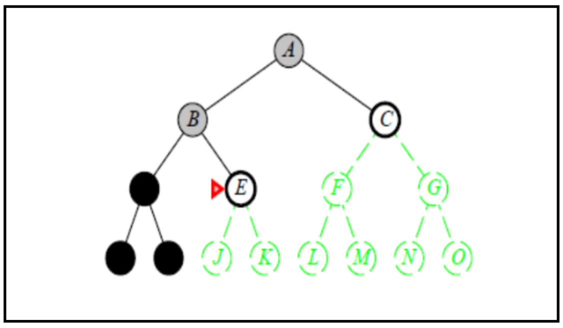












- **Properties of Depth First Search(DFS):**
  - ✓ **Complete? -** No: Fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along

    Path complete infinite spaces
  - ✓ **Time?? :** O(b^m): Terrible if m is much larger than 'd', but if solutions are dense, may be much faster than

breadth first search.

- ✓ **Space?? :** O(b^m), i.e., linear space!
- ✓ **Optimal??** : No

4. 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 DEPTH-LIMITED treated as if they have no successors. This approach is called depth limited search.
- ✓ The SEARCH depth limit solves the infinite-path problem.
- ✓ Unfortunately, it also introduces an additional source of incompleteness if we choose 'l' < d, that is, the

shallowest goal is beyond the depth limit. (This is likely when d is unknown.)

- ✓ Depth-limited search will also be non-optimal if we choose ‘l’ > d.
- ✓ Its time complexity is O(b) and its space complexity is O(b). Depth-first search can be viewed as a special case of depth-limited search with‘l’ =∞.

```
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.

5. Iterative deepening depth-first search…

- ✓ Iterative deepening search (or iterativedeepening-depth-first-search) is a general strategy often used in combination with depth-first-search,that finds the better depth limit.
- ✓ It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found.
- ✓ This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- ✓ Iterative deepening combines the benefits of depth-first and breadth-first-search.
- ✓ Like depth-first-search, its memory requirements are modest;O(bd) to be precise.

```
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.

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

➢ **Four iterations of iterative deepening search on a binary tree in shown below.**

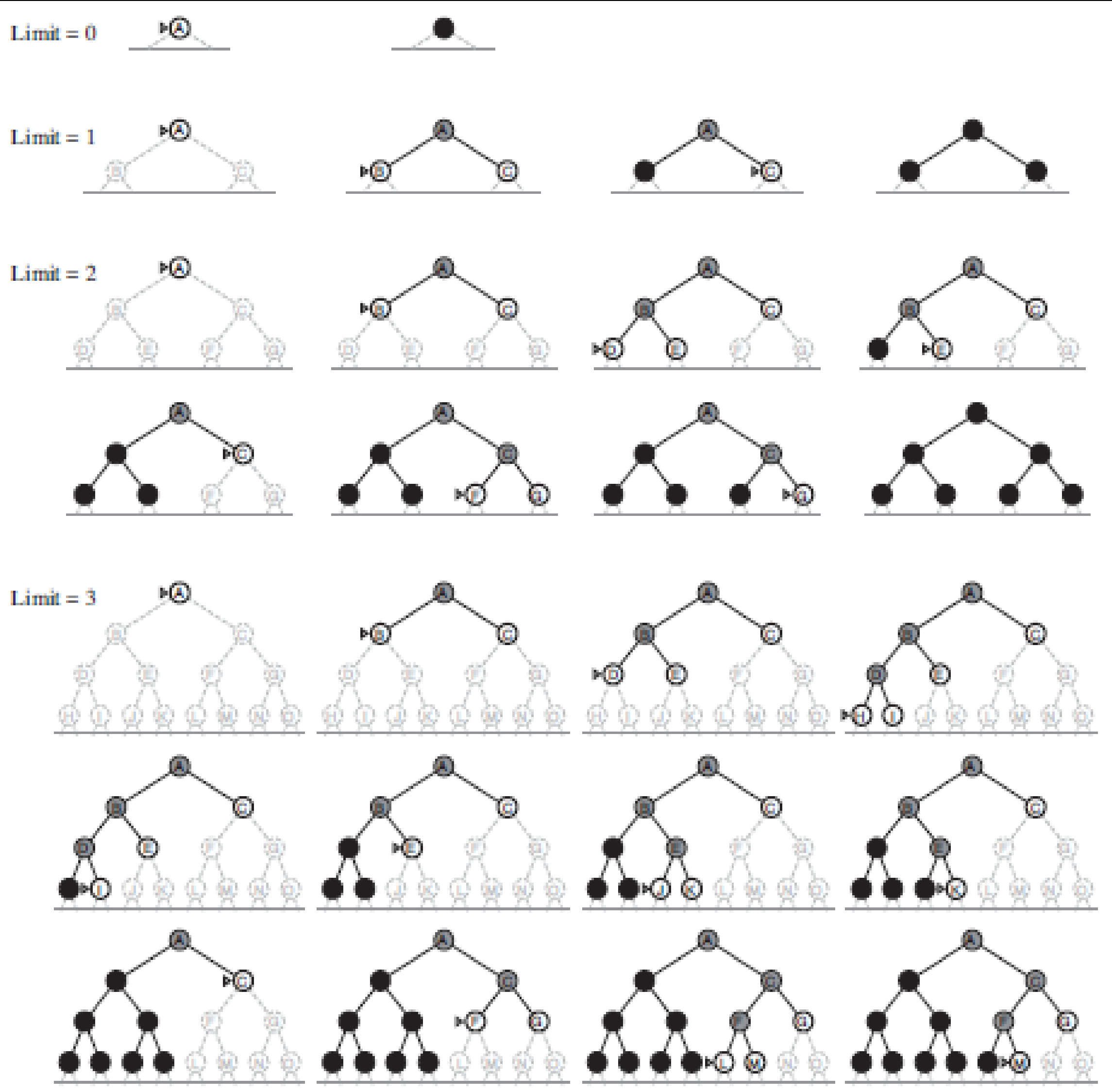
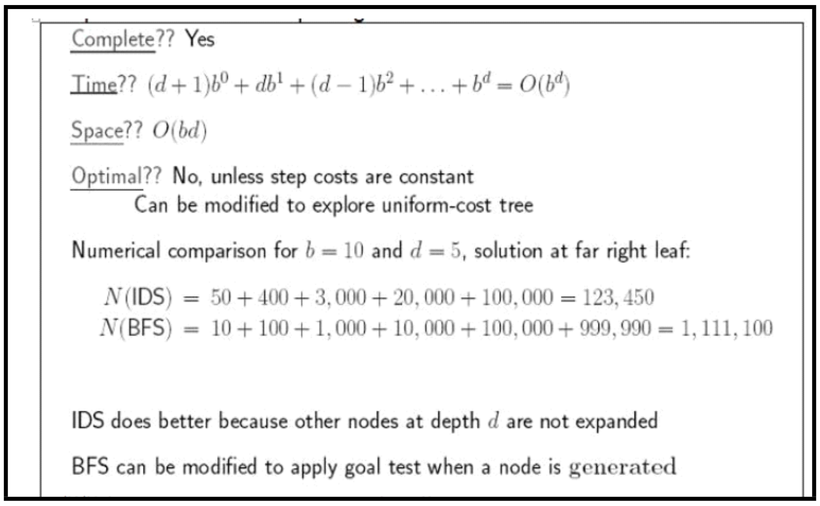


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

- **Properties of IDS….**



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

6. Bi-directional search…

- ✓ The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.
- ✓ The motivation is that $b^{d/2} + b^{d/2}$ much less than or the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

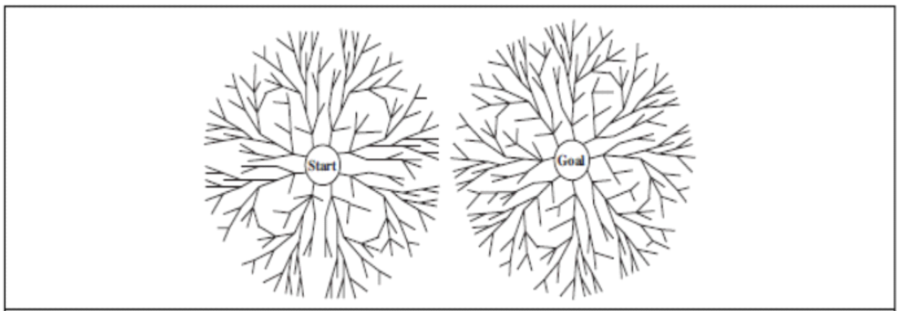
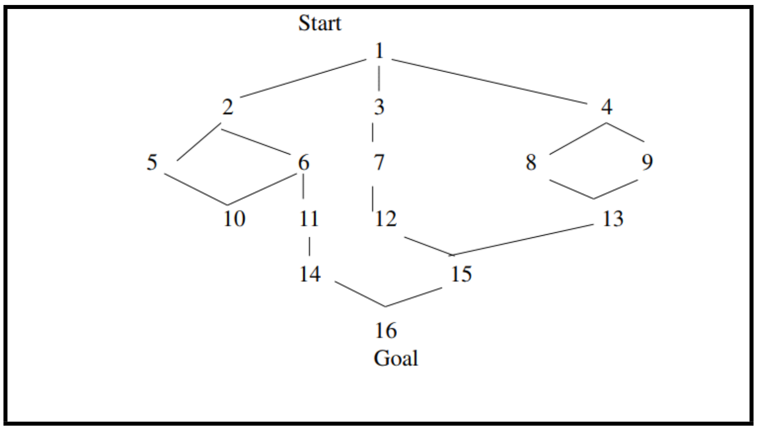
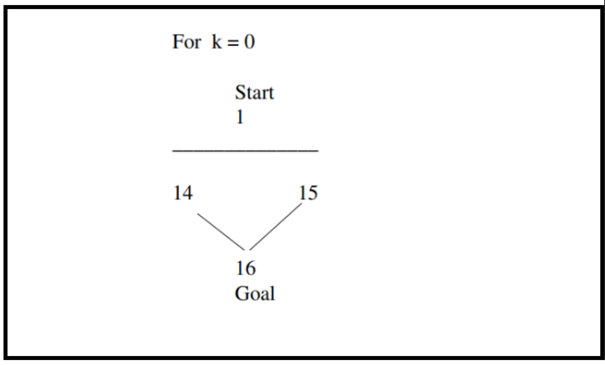
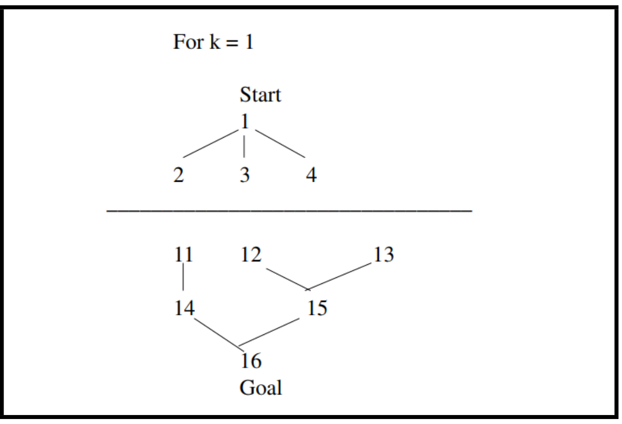


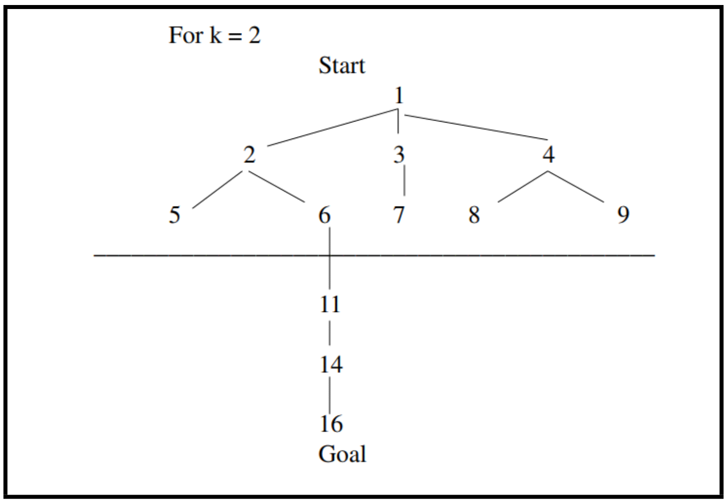
Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

- **Example Graph:**

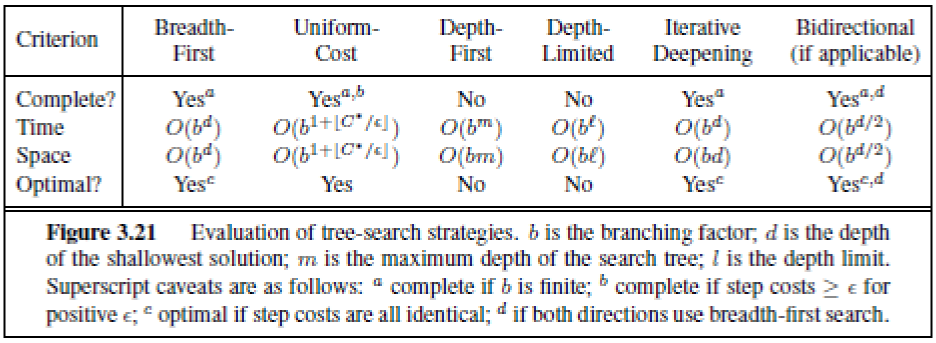








2.4.7 Comparing uninformed search strategies…



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.

2.1.5. Informed or Heuristic search strategies…

- ✓ Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself.
- ✓ It can find solutions more efficiently than uninformed strategy.
- ✓ Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function f(n).
- ✓ The node with lowest evaluation is selected for expansion, because the evaluation measures the distance to the goal.
- ✓ This can be implemented using a priority-queue, a data structure that will maintain the fringe in ascending order of fvalues.
- ✓ A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- ✓ The key component of Best-first search algorithm is a heuristic function, denoted by h(n):
  h(n) = estimated cost of the cheapest path from node n to a goal node.
- ✓ Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

➢ Heuristic Searching methods…

i. Greedy Best First search

ii. A* search

iii. Memory-bounded heuristic search(RBFS)

1. 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 a solution quickly.
- ✓ It evaluates the nodes by using the heuristic function f(n) = h(n).
- ✓ Taking the example of Route-finding problems in Romania , the goal is to reach Bucharest starting from the city Arad.
- ✓ We need to know the straight-line distances(SLD) to Bucharest from various cities as shown in Figure.
- ✓ For example, the initial state is In(Arad) ,and the straight line distance heuristic hSLD(In(Arad)) is found to be 366.

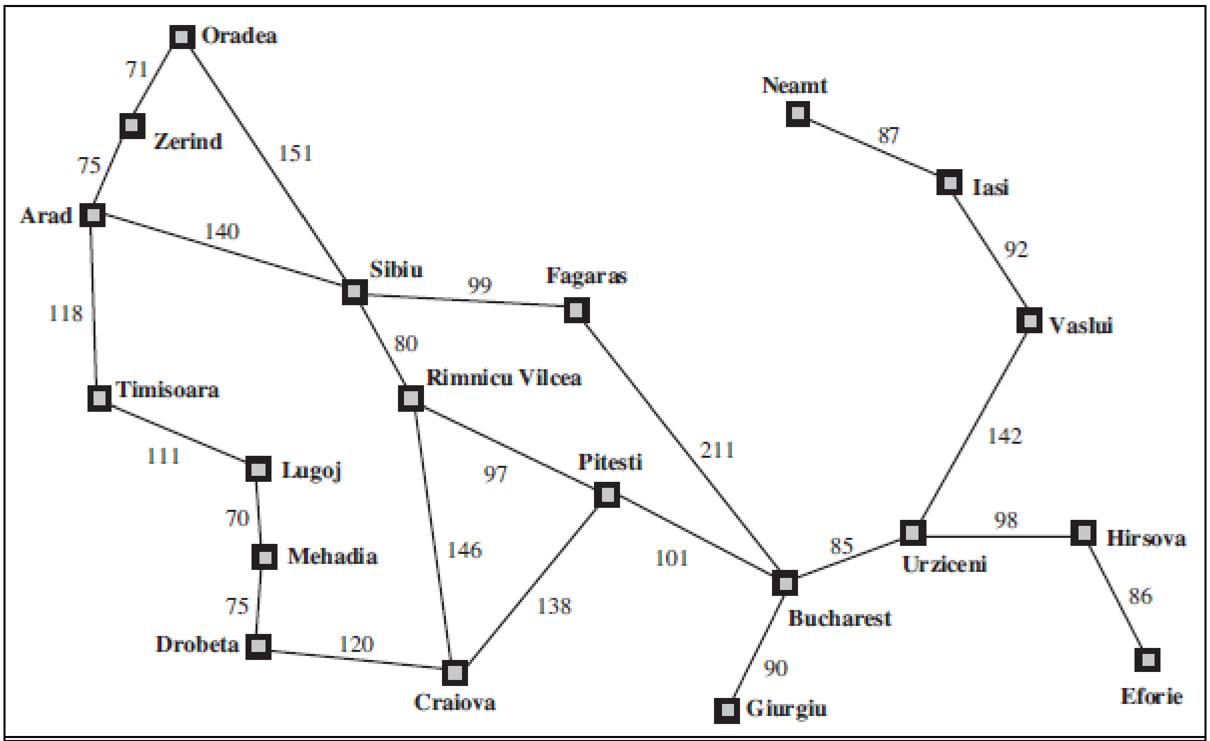
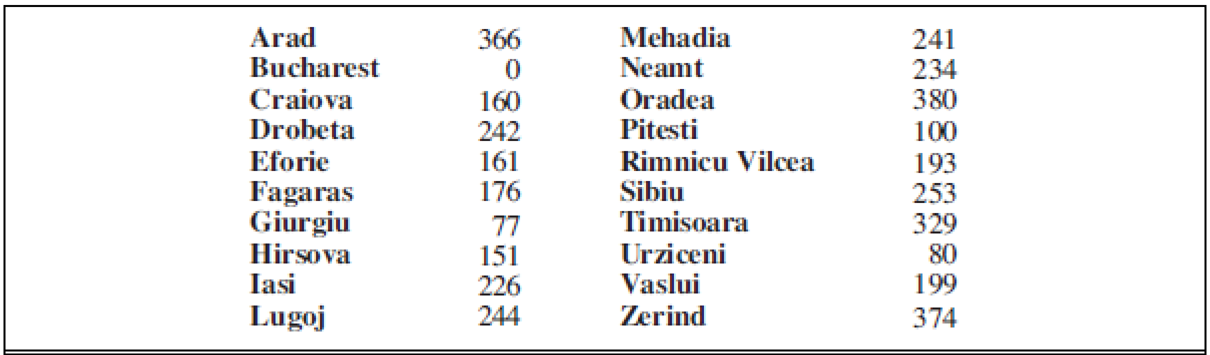


Figure 3.2 A simplified road map of part of Romania.



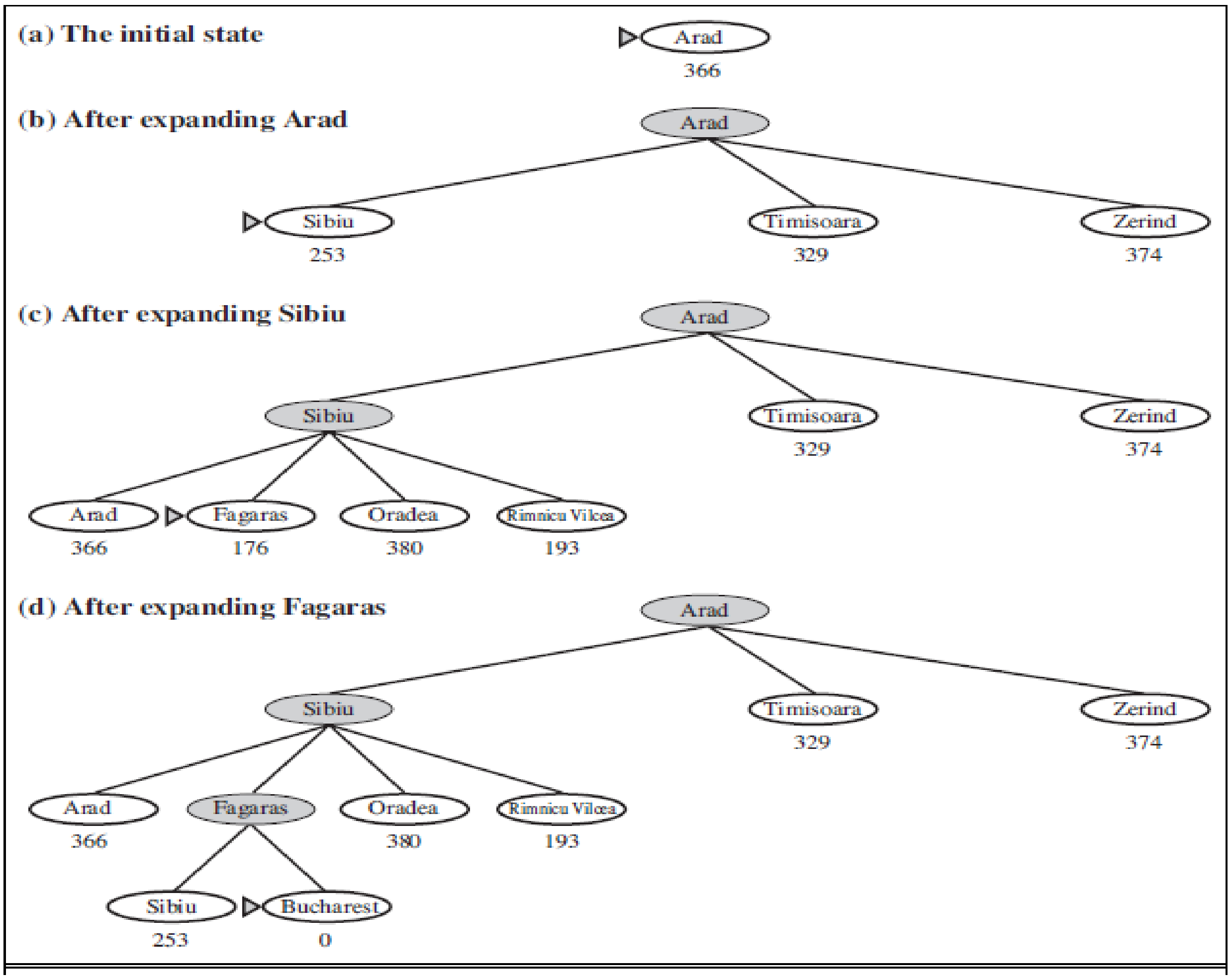
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

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

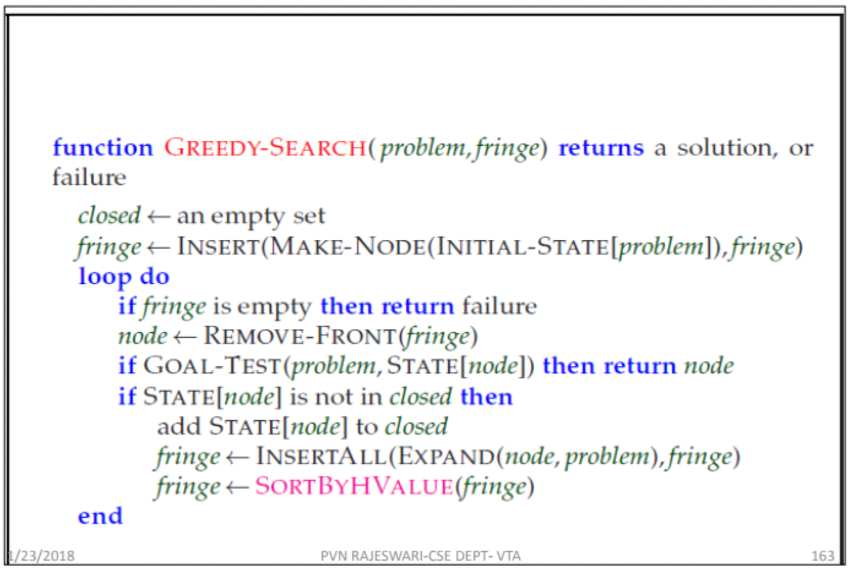
- ✓ The figure shows the progress of greedy best-first search using hSLD to find a path from Arad to Bucharest.
- ✓ The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara.

- ✓ The next node to be expanded will be Fagaras, because it is closest.
- ✓ Fagaras in turn generates Bucharest, which is the goal.

FINAL PATH : Arad → Sibiu → Fagaras → Bucharest.



➢ **Greedy search Algorithm…**



```
function GREEDY-SEARCH(problem, fringe) returns a solution, or
failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
            fringe ← SORTBYHVALUE(fringe)
    end
```

1/23/2018 PVN RAJESWARI-CSE DEPT- VTA 163

2. A* Search…

- ✓ A* is very efficient search strategy and it is the most widely used form of best-first search.
- ✓ Basic idea is to combine uniform cost search and greedy search.
- ✓ The evaluation function f(n) is obtained 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).

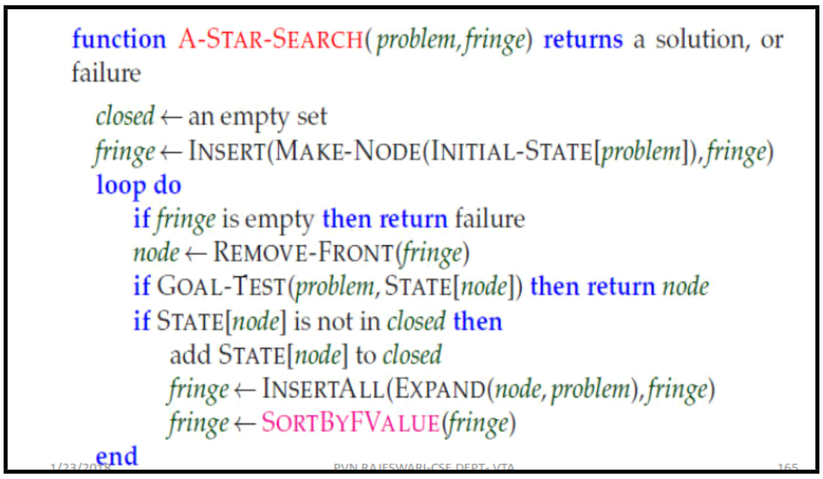
  (or)

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

  – g(n) is path cost of n;

  – h(n) is expected cost of cheapest solution from n.

- ✓ It Aims to minimise overall cost.
- ✓ **Algorithm for A* search stratergy:**



```
function A-STAR-SEARCH(problem,fringe) returns a solution, or
failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem),fringe)
          fringe ← SORTBYFVALUE(fringe)
  end
```

1/23/2018 PVN RAJESWARI-CSE DEPT- VTA 165

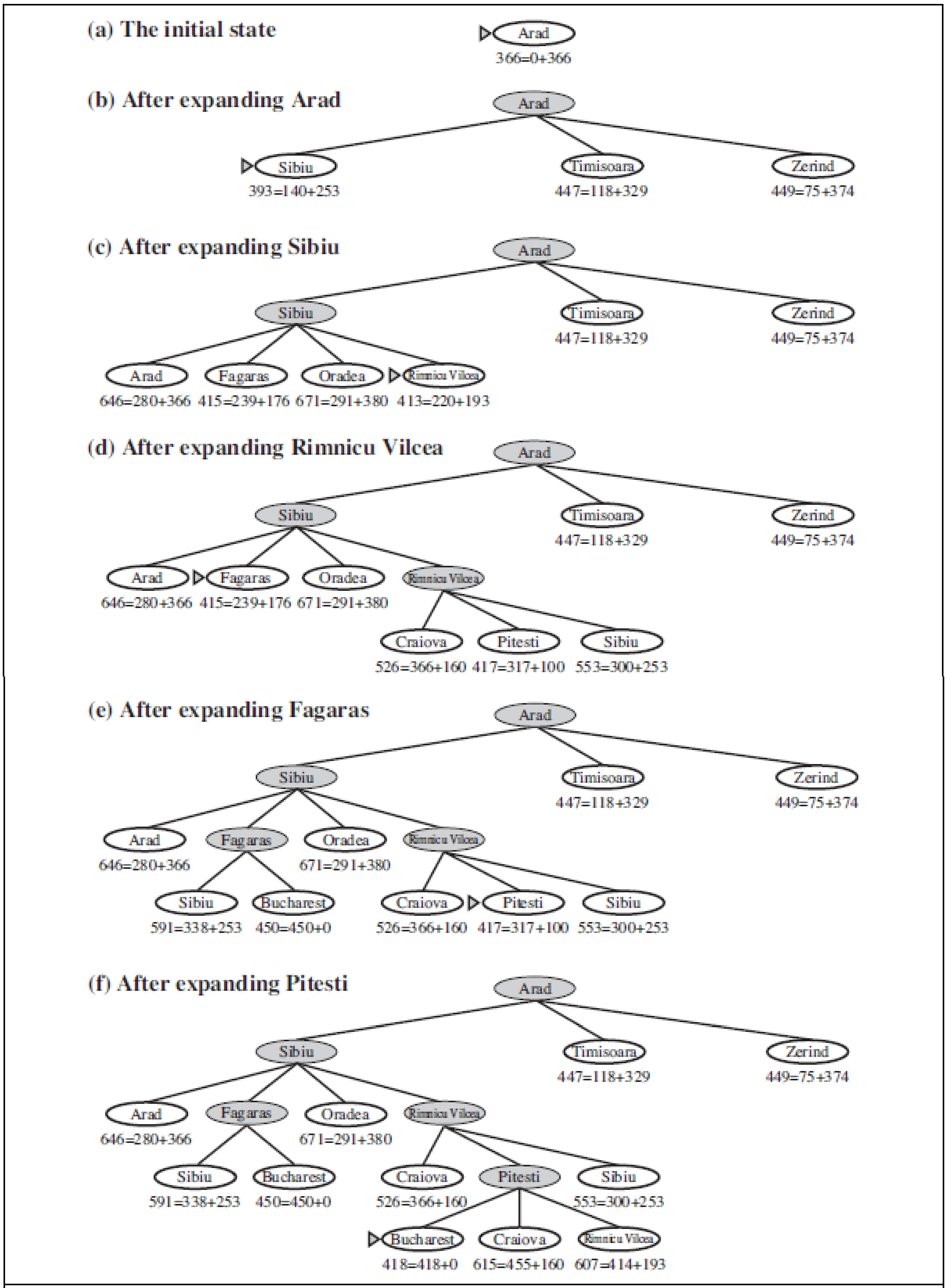


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

FINAL PATH: Arad →Sibiu →Rimnicu→ Pitesti →Bucharest.

➢ Optimality of A*…

- ✓ A⍰ search is both complete and optimal.
- ✓ The algorithm is identical to UNIFORM-COST-SEARCH except that A⍰ uses g + h instead of g.
- ✓ A* is optimal in precise sense—it is guaranteed to find a minimum cost path to the goal.

There are a set of conditions under which A* will find such a path:

1. Each node in the graph has a finite number of children.
2. All arcs have a cost greater than some positive .
3. For all nodes in the graph h(n) always underestimates the true distance to the goal.

➢ Conditions for the Optimality…

1. 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.

2. 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.

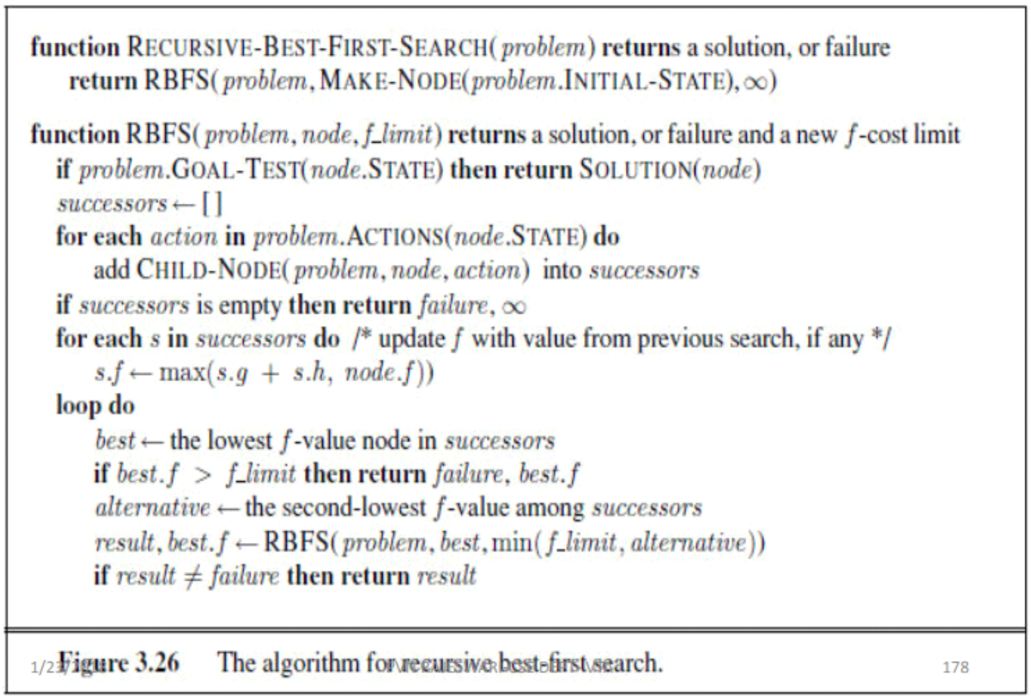
➢ Problems in A*…

- ✓ Computation time, A⍰'s main drawback.
- ✓ Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A⍰ usually runs out of space long before it runs out of time.
- ✓ For this reason, A⍰ is not practical for many largescale problems.
- ✓ There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

3. Memory-bounded Heuristic search…

- ✓ 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.
- ✓ 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.
- ✓ 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.

➢ RBFS Algorithm…

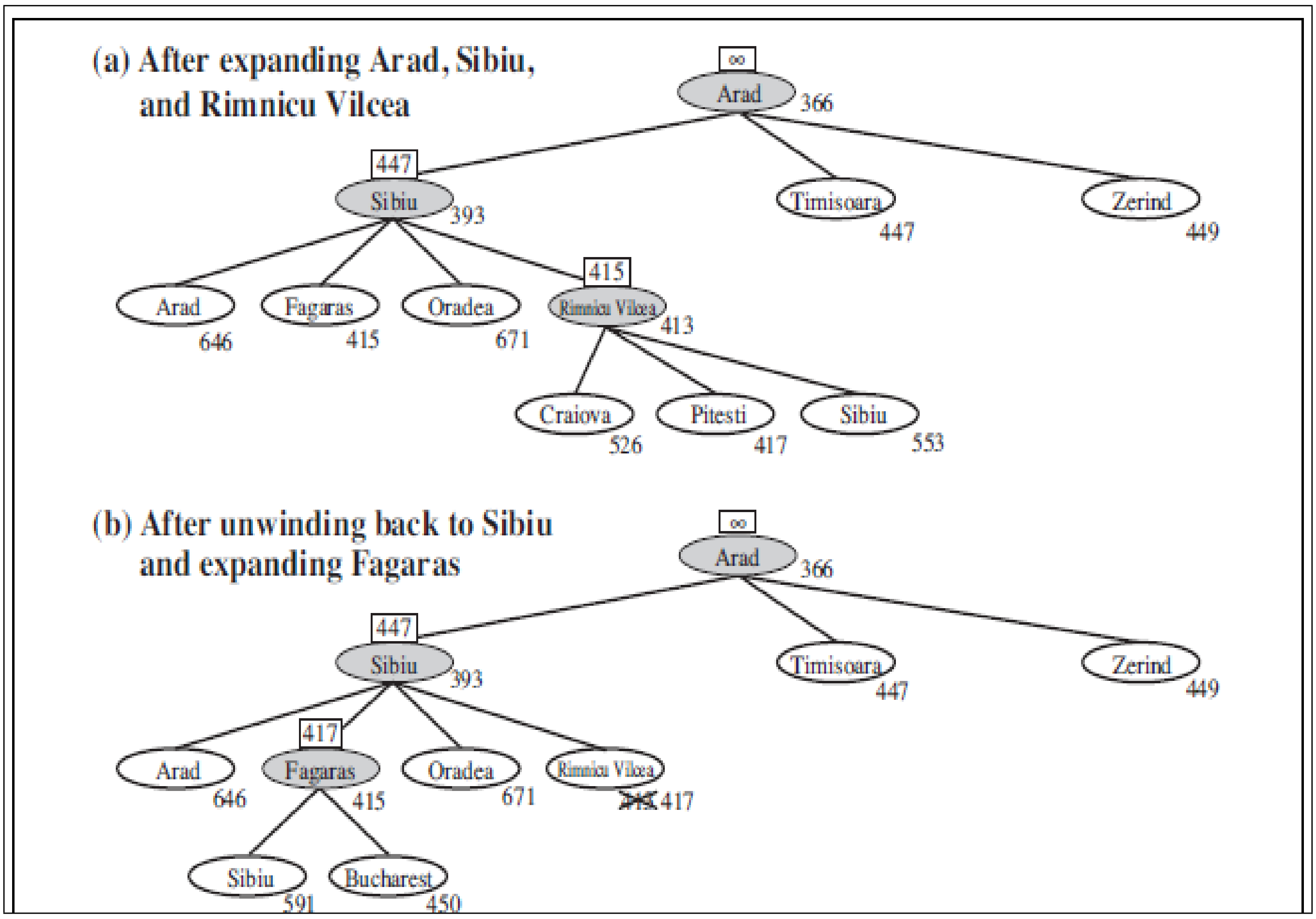


```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

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
```

1/2 Figure 3.26 The algorithm for recursive best-first search. 178

➢ Stages in RBFS Algorithm…



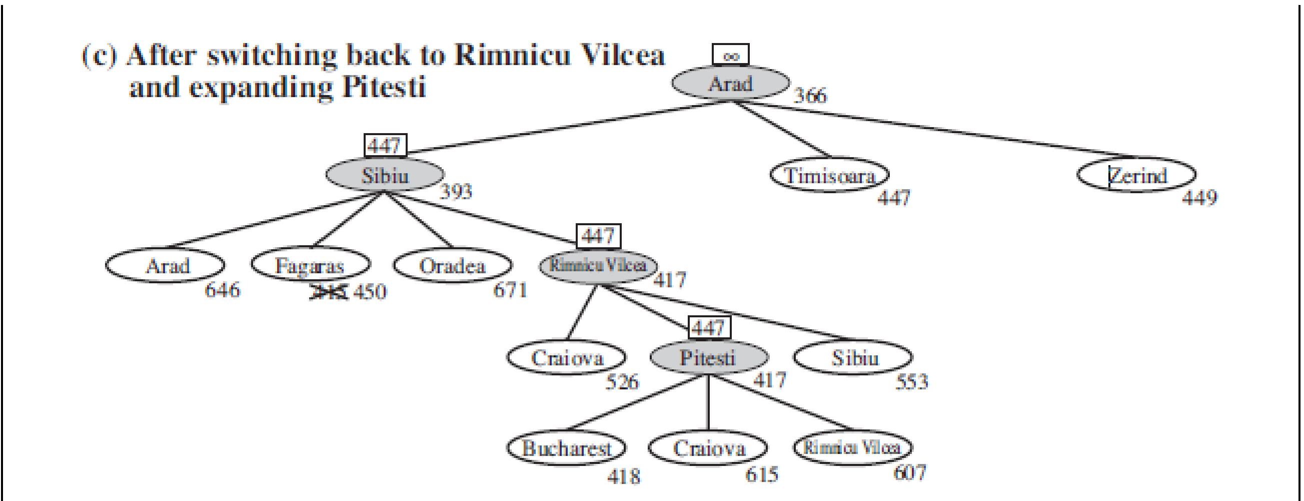


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The *f*-limit value for each recursive call is shown on top of each current node, and every node is labeled with its *f*-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is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

- ✓ RBFS is a bit more efficient than IDA* Still excessive node generation (mind changes).
- ✓ Like A*, optimal if h(n) is admissible. Space complexity is O(bd).
- ✓ IDA* retains only one single number (the current f-cost limit).
- ✓ Time complexity difficult to characterize Depends on accuracy if h(n) and how often best path changes.
- ✓ IDA* and RBFS suffer from too little memory.

2.1.6. HEURISTIC FUNCTIONS :

- The 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.

This means that an exhaustive tree search to depth 22 would look at about

$$3^{22} \approx 3.1 \times 10^{10} \text{ states.}$$

- A graph search would cut this down by a factor of about 170,000 because only 9!/2 = 181, 440 distinct states are reachable.

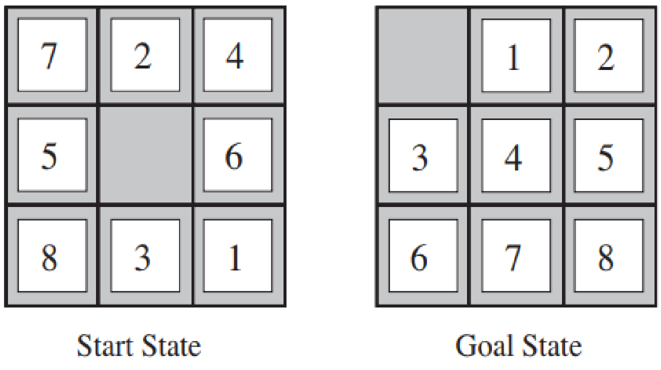


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

- The corresponding number for the 15-puzzle is roughly , so the next order of business is to find a good heuristic function.
- If we want to find the shortest solutions by using A⍰, we need a heuristic function that never overestimates the number of steps to the goal.
- There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:
- **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**.

(i) 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,

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

- For example, if A⍰ finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

(ii) Generating admissible heuristics from relaxed problems :

- We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better.
- How might one have come up with h2? Is it possible for a computer to invent such a heuristic mechanically.

  h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for *simplified* versions of the puzzle.
- 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 cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem***".
- Furthermore, because the derived heuristic is an exact cost for the relaxed 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.

(iii) Generating admissible heuristics from subproblems : (Pattern databases)

- Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem.
- For example, a subproblem of the 8-puzzle instance is

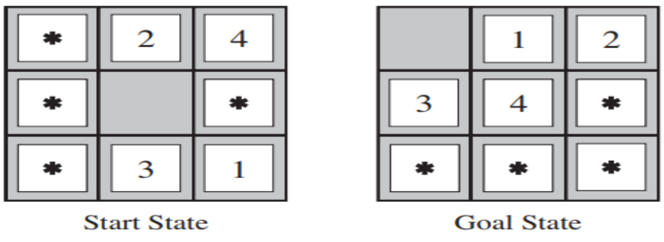


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions.
- Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem.
- It turns out to be more accurate than Manhattan distance in some cases.
- 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.
- The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered .
- Each database yields an admissible heuristic, and these heuristics can be combined, and by taking the maximum value.

  A combined heuristic of this kind is much more accurate than the Manhattan distance .
- One might wonder whether the heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap.
- Would this still give an admissible heuristic? The answer is no.
- So,we record not the total cost of solving the 1-2-3-4 subproblem, but just the number of moves involving 1-2-3-4. Then it is easy to see that the sum of the two costs is still a lower bound on the cost of solving the entire problem.
- This is the idea behind **disjoint pattern databases**.

  With such databases, it is possible to solve random 15-puzzles in a few milliseconds .

(iv) Learning heuristics 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 (with luck) predict solution costs for other states that arise during search.

4 BEYOND CLASSICAL SEARCH

In which we relax the simplifying assumptions of the previous chapter, thereby getting closer to the real world.

Chapter 3 addressed a single category of problems: observable, deterministic, known environments where the solution is a sequence of actions. In this chapter, we look at what happens when these assumptions are relaxed. We begin with a fairly simple case: Sections 4.1 and 4.2 cover algorithms that perform purely **local search** in the state space, evaluating and modifying one or more current states rather than systematically exploring paths from an initial state. These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics **(simulated annealing)** and evolutionary biology **(genetic algorithms).**

Then, in Sections 4.3-4.4, we examine what happens when we relax the assumptions of determinism and observability. The key idea is that if an agent cannot predict exactly whit percept it **will** receive, then it will need to consider what to do under each **contingency that** its percepts may reveal. With partial observability, the agent will also need to keep track of the states it might be in.

Finally, Section 4.5 investigates **online search,** in which the agent is faced with a state space that is initially unknown and must be explored.

4.1 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each paint along the path. When a goal is found, *the path* to that goal also constitutes a *solution* to the problem. In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 71). what matters is the final configuration of queens, not the order in which they are added. The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing. and portfolio management.

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **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.

LOCAL SEARCH

CURRENT NODE

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.** Many optimization problems do not fit the "standard" search model introduced in Chapter 3. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

OPTIMIZATION PROBLEM

OBJECTIVE FUNCTION

To understand local search, we find it useful to consider the **state-space landscape** (as 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.** {You can convert from one to the other **just** by insetting a minus sign.) 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.

STATE-SPACE LANDSCAPE

GLOBAL MINIMUM

GLOBAL MAXIMUM

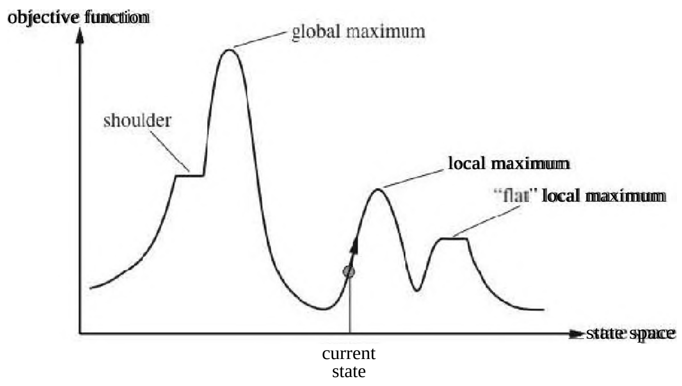


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

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

  current 4— 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 tech nique. 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.

4.1.1 Hill-climbing search

HILL CLIMBING

STEEPEST ASCENT

The **hill-climbing** search algorithm **(steepest-ascent** version) is shown in Figure 4.2. 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. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia

To illustrate hill climbing, we will use the **8-queens problem** introduced on page 71. Local search algorithms typically use a **complete-state formulation,** where each state has S 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 x 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.

GREEDY LOCAL SEARCH

Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. For example, from the state in Figure 4.3(a), it takes just five steps to reach the state in Figure 4.3(b), which has *h*= 1 and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:

LOCAL MAXIMUM

- 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. Figure 4.1 illustrates the problem schematically. More

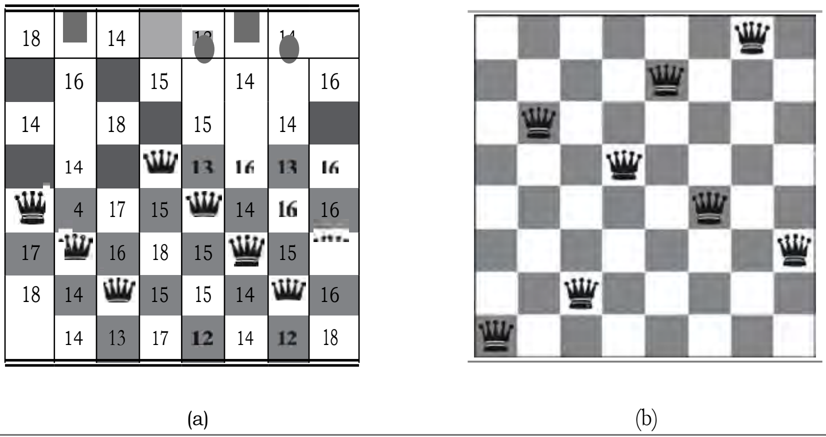


Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

concretely, the state in Figure 4.3(b) is a local maximum (i.e., a local minimum for the cost *h);* every move of a single queen makes the situation worse.

RIDGE

- Ridges: a ridge is shown in Figure 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate

PLATEAU

SHOULDER

- Plateaux: a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. (See Figure 4.1.) A hill-climbing search might get lost on the plateau.

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state. steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck—not bad for a state space with $8^8 \approx 17$ million states.

The algorithm in Figure 4.2 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going—to allow a

SIDEWAYS MOVE

sideways move in the hope that the plateau is really a shoulder, as shown in Figure 4.1? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder, One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

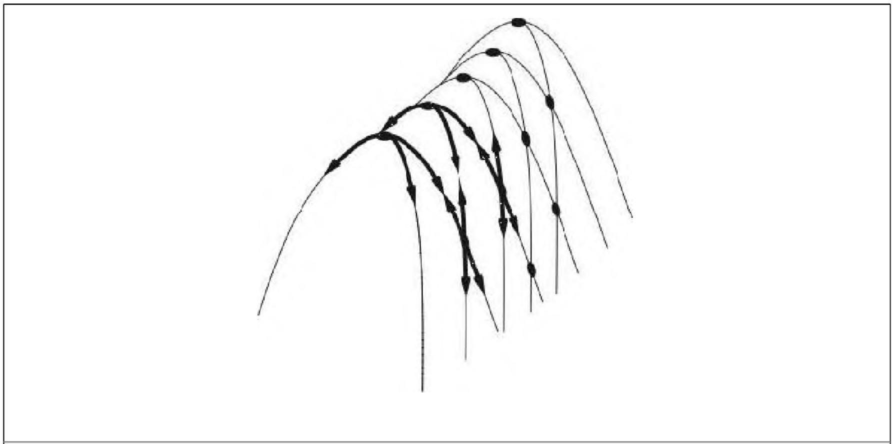


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

STOCHASTIC 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 rinds 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.

FIRST-CHOICE HILL CLIMBING

RANDOM-RESTART 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. It is trivially complete with probability approaching 1, because it will eventually generate *a* goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is 1/p. For 8-queens instances with no sideways moves allowed, p 0.14, so we need roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1-p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, $1/0.94 \approx 1.06$ iterations are needed on average and (1 x 21) + (0.06/0.94) x 64 ti 25 steps. For 8-queens, then, random-restart hill climbing is very effective indeed. Even for three million queens, the approach can find solutions in under a minute.[2]

Generating a *random* state from an implicitly specified state space can be a hard problem in itself.

[2] Luby *et at* (1993) prove that it is best, in some cases, to restart a randomized search algorithm after a particular, fixed amount of time and that this can be *much* more efficient than letting each search continue indefinitely. Disallowing or limiting the number of sideways moves is an example of this idea.

The success of hill climbing depends very much on the shape of the state-space landscape: if there arc few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly. On the other hand, many real problems have a landscape that looks more like a widely scattered family of balding porcupines on a flat floor, with miniature porcupines living on the tip of each porcupine needle, *ad infinitum.* NP-hard problems typically have an exponential number of local maxima to get stuck on Despite this, a reasonably good local maximum can often be found after a small number of restarts.

4.1.2 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 hall 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).

SIMULATEC ANNEALING

GRADIENT DESCENT

The innermost loop of the simulated-annealing algorithm (Figure 4,5) is quite similar to hill climbing. Instead of picking the *best* move, however, it picks a *random* move If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with sonic probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the *schedule* lowers T slowly enough, the algorithm **will** find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks In Exercise 44, you are asked to compare its performance to that of random-restart hill climbing on the 8-queens puzzle_

4.1.3 Local beam search

LOCAL BEAN SEARCH

Keeping just one node in memory might seem to be **an** extreme reaction to the problem of memory limitations. The **local beam search** algorithm[-] keeps track of k states rather than

[-] Local beam search is an adaptation of beam search, which is a path-based algorithm.

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

  current 4— MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to co do
      T — schedule(t)
      if T = 0 then return current
      next ⟵ a randomly selected successor of current
        E 4— next .VALUE — current.VALUE
      if ΔE > 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.

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.

At first sight, a local beam search with *k* states might seem to be nothing more than running *k* random restarts in parallel instead of in sequence. **In** fact, the two algorithms are quite different. 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.* In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

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 In 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).

STOCHASTIC BEAM SEARCH

4.1.4 Genetic algorithms

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. The analogy to natural selection is the same as in stochastic beam search, except that now we arc dealing with sexual rather than asexual reproduction.

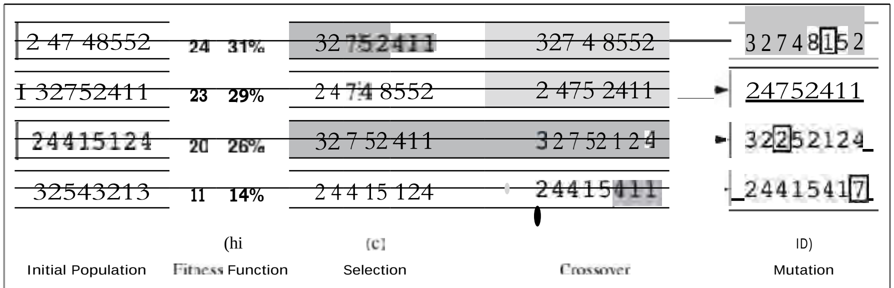


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

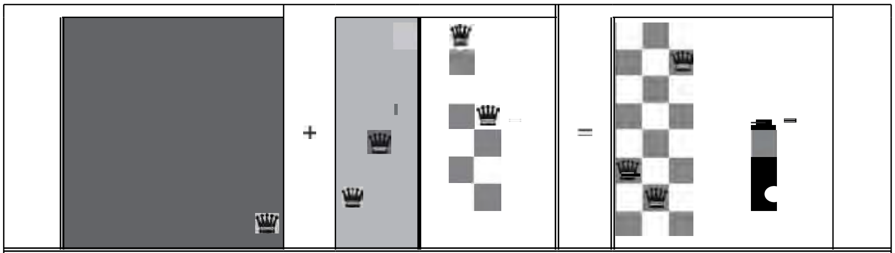


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

POPULATION

INDIVIDUAL

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 Os and Is. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We demonstrate later that the two encodings behave differently.) Figure 4.6(a) shows a population of four 8-digit strings representing 8-queens states.

FITNESS FUNCTION

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.** A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all.[4] For each pair to be mated, a **crossover** point is chosen randomly from the positions in the string. In Figure 4.6, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.[5]

CROSSOVER

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. The 8-queens states involved in this reproduction step are shown in Figure 4.7. The example shows that when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when must individuals are quite similar.

MUTATION

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. Figure 4.8 describes an algorithm that implements all these steps.

Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can he shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.

SCHEMA

The theory of genetic algorithms explains how this works using the idea of a **schema**, which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema. It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema within the population will grow over time. Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemata correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemata may represent components of the antenna, such as reflectors and deflectors. A good

INSTANCE

[4] There are many variants of this selection rule. The method of caning, in which all individuals below a given threshold are discarded, can he shown to converge faster than the random version (Baum *et at,* 1995).

[5] It is here that the encoding matters. If a 24-bit encoding is used instead of it digits, then the crossover point has a 2/3 chance of being in the middle of a digit, which results in an essentially arbitrary mutation of that digit.

```
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 4— empty set
      for a; = 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 4— MUTATE(child)
         add child to new_population
      population 4— new_population
   until some individual is fit enough, or enough time has elapsed
   return the best individual in population, according to FITNESS-FM
```

```
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.

component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their æsthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

4.2 LOCAL SEARCH IN CONTINUOUS SPACES

In Chapter 2, we explained the distinction between discrete and continuous environments, pointing out that most real-world environments are continuous, Yet none of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors. This section provides a *very brief* introduction to sonic. local search techniques for finding optimal solutions in continuous spaces. The literature on this topic is vast; many of the basic techniques

EVOLUTION AND SEARCH

The theory of **evolution** was developed in Charles Darwin's *On the Origin of Species by Means of Natural Selection* (1859) and independently by Alfred Russel Wallace (1858). The central idea is simple: variations occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness.

Darwin's theory was developed with no knowledge of how the traits of organisms can be inherited and modified. The probabilistic laws governing these processes were first identified by Gregor Mendel (1866), a monk who experimented with sweet peas. Much later, Watson and Crick (1953) identified the structure of the DNA molecule and its alphabet, AGTC (adenine, guanine, thymine, cytosine). In the standard model, variation occurs both by point mutations in the letter sequence and by "crossover" (in which the DNA of an offspring is generated by combining long sections of DNA from each parent).

The analogy to local search algorithms has already *been* described; the principal difference between stochastic beam search and evolution is the use of *sexual* reproduction, wherein successors are generated from *multiple* organisms rather than just one. The actual mechanisms of evolution are, however, far richer than most genetic algorithms allow. For example, mutations can involve reversals, duplications, and movement of large chunks of DNA; some viruses borrow DNA from one organism and insert it in another; and there are transposable genes that do nothing but copy themselves many thousands of times within the genome. There are even genes that poison cells from potential mates that do not carry the gene, thereby increasing their own chances of replication. Most important is the fact that the *genes themselves encode the mechanisms* whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

Darwinian evolution may appear inefficient, having generated blindly some $10^{[illegible]}$ or so organisms without improving its search heuristics one iota. Fifty years before Darwin, however, the otherwise great French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits *acquired by adaptation during an organism's lifetime* would be passed on to its offspring. Such a process would be effective but does not seem to occur in nature. Much later, James Baldwin (1896) proposed a superficially similar theory: that behavior learned during an organism's lifetime could accelerate the rate of evolution_ ike Lamarck's, Baldwin's theory is entirely consistent with Darwinian evolution because it relies on selection pressures operating on individuals that have found local optima among the set of possible behaviors allowed by their genetic makeup. Computer simulations confirm that the "Baldwin effect" is real, once "ordinary" evolution has created organisms whose internal performance measure correlates with actual fitness.

originated in the 17th century, after the development of calculus by Newton and Leibniz.[6] We find uses for these techniques at several places in the book, including the chapters on learning, vision, and robotics.

We begin with an example. Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map (Figure 3.2) to its nearest airport is minimized. The state space is then defined by the coordinates of the airports: (x_1, y_1), (x_2, y_2), and (x_3, y_3). This is a *six-dimensional* space; we also say

VARIABLE

that states are defined by six variables. (In general, states are defined by an n-dimensional vector of variables, x.) Moving around in this space corresponds to moving one or more of the airports on the map. The objective function $f(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state once we compute the closest cities. Let C_i be the set of cities whose closest airport (in the current state) is airport i. Then, *in the neighborhood of the current state,* where the C_is remain constant, we have

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2 \qquad (4.1)$$

This expression is correct *locally,* but not globally because the sets C_i are (discontinuous) functions of the state.

DISCRETIZATION

One way to avoid continuous problems is simply to discretize the neighborhood of each state. For example, we can move only one airport at a time in either the x or y direction by a fixed amount $\pm\delta$. With 6 variables, this gives 12 possible successors for each state. We can then apply any of the local search algorithms described previously. We could also apply stochastic hill climbing and simulated annealing directly, without discretizing the space. These algorithms choose successors randomly, which can be done by generating random vectors of length δ.

GRADIENT

Many methods attempt to use the **gradient** of the landscape to find a maximum. The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope. For our problem, we have

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3}\right)$$

In some cases, we can find a maximum by solving the equation $\nabla f = 0$. (This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.) In many cases, however, this equation cannot be solved in closed form. For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state This means we can compute the gradient *locally* (but not *globally);* for example,

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c) . \qquad (4.2)$$

Given a locally correct expression for the gradient, we can perform steepest-ascent hill climb-

[6] A basic knowledge of multivariate calculus and vector arithmetic is useful for reading this section.

ing by updating the current state according to the formula

$$\mathrm{x} \quad \asymp \quad \alpha\nabla f(\mathbf{x}) ,$$

STEP SIZE

where a is a small constant often called the step size. In other cases, the objective function might not be available in a differentiable form at all—for example, the value of a particular set of airport locations might be determined by running some large-scale economic simulation package. In those cases, we can calculate a so-called empirical gradient by evaluating the response to small increments and decrements in each coordinate. Empirical gradient search is the same as steepest-ascent hill climbing in a discretized version of the state space.

EMPIRICAL GRADIENT

Hidden beneath the phrase "a is a small constant" lies a huge variety of methods for adjusting a. The basic problem is that, if a is too small, too many steps are needed; if a is too large, the search could overshoot the maximum. The technique of line search tries to overcome this dilemma by extending the current gradient direction—usually by repeatedly doubling a—until f starts to decrease again. The point at which this occurs becomes the new current state. There are several schools of thought about how the new direction should be chosen at this point.

LINE SEARCH

NEWTOI –RAPHSON

For many problems, the most effective algorithm is the venerable **Newton–Raphson** method. This is a general technique for finding roots of functions—that is, solving equations of the form $g(x) = 0$. It works by computing a new estimate for the root x according to Newton's formula

$$x \qquad x - g(x)/g\ \ (x) \ .$$

To rind a maximum or minimum *of f,* we need to find x such that the *gradient* is zero (i.e. $\nabla f(\mathbf{x}) = 0$). Thus, *g(x)* in Newton's formula becomes V f(x), and the update equation can be written in matrix-vector form as

$$\mathrm{x} \quad \asymp \quad - \mathbf{H} \quad (\mathbf{x})\nabla f\ (x)\ \mathrm{ı}$$

HESSIAN

where H**i** (x) is the Hessian matrix of second derivatives, whose elements H_{ij} are given by $\partial\ \ f/\partial x_i \partial x_j$. For our airport example, we can see from Equation (4.2) that Hy (x) is particularly simple: the off-diagonal elements are zero and the diagonal elements for airport i are just twice the number of cities in C_i. A moment's calculation shows that one step of the update moves airport directly to the centroid of which is the minimum of the local expression for f from Equation (4.1). For high-dimensional problems, however, computing the n entries of the Hessian and inverting it may be expensive, so many approximate versions of the Newton-Raphson method have been developed.

Local search methods suffer from local maxima, ridges, and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

CONSTRAINED OPTIMIZATION

A final topic with which a passing acquaintance is useful is constrained optimization. An optimization problem is constrained if solutions must satisfy some hard constraints on the values of the variables. For example, in our airport-siting problem, we might constrain sites

˜ In general, the Newton–Raphson update can be seen as fitting a quadratic surface to f at x and then moving directly to the minimum of that surface—which is also the minimum of f if f is quadratic.

to be inside Romania and on dry land (rather than in the middle of lakes). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of **linear programming** problems, in which constraints must be linear inequalities forming a **convex set** and the objective function is also linear. The time complexity of linear programming is polynomial in the number of variables.

LINEAR PROGRAMMING

CONVEX SET

Linear programming is probably the most widely studied and broadly useful class of optimization problems. It is a special case of the more general problem of **convex optimization**, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region. Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables. Several important problems in machine learning and control theory can be formulated as convex optimization problems (see Chapter 20).

CONVEX OPTIMIZATION

4.3 SEARCHING WITH NONDETERMINISTIC ACTIONS

In Chapter 3, we assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action, although of course they tell the agent the initial state.

When the environment is either partially observable or nondeterministic (or both), percepts become useful. In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals. When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred. In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts. So the solution to a problem is not a sequence but a **contingency plan** (also known as a **strategy**) that specifies what to do depending on what percepts are received. In this section, we examine the case of nondeterminism, deferring partial observability to Section 4.4.

CONTINGENCY PLAN

STRATEGY

4.3.1 The erratic vacuum world

As an example, we use the vacuum world, first introduced in Chapter 2 and defined as a search problem in Section 3.2.1. Recall that the state space has eight states, as shown in Figure 4.9. There are three actions—*Left*, *Right*, and *Suck*—and the goal is to clean up all the dirt (states 7 and 8). If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms in Chapter 3 and the solution is an action sequence. For example, if the initial state is 1, then the action sequence [*Suck*, *Right*, *Suck*] will reach a goal state, 8.

A set of points S is convex if the line joining any two points in S is also contained in S. A convex function is one for which the space "above" it forms a convex set; by definition, convex functions have no local (as opposed to global) minima.

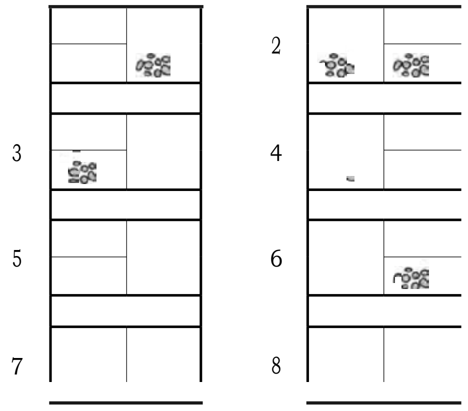


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

ERRATIC VACUUM WORLD

Now suppose that we introduce nondeterminism in the form of a powerful but erratic vacuum cleaner. In the erratic vacuum world, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square. too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.[2]

To provide a precise formulation of this problem, we need to generalize the notion of a transition model from Chapter 3. Instead of defining the transition model by a RESULT function that returns a single state, we use a RESULTS function that returns a *set* of possible outcome states. For example, in the erratic vacuum world, the *Suck* action in state 1 leads to a state in the set $\{5, 7\}$—the dirt in the right-hand square may or may not be vacuumed up.

We also need to generalize the notion of a solution to the problem. For example, if we start in state 1, there is no single *sequence* of actions that solves the problem. Instead, we need a contingency plan such as the following:

$$[Stick, \text{ if } State = 5 \text{ then } [Right, Suck] \text{ else } \quad . \qquad (4.3)$$

Thus, solutions for nondeterministic problems can contain nested if—then—else statements; this means that they are *trees* rather than sequences_ This allows the selection of actions based on contingencies arising during execution. Many problems in the real, physical world are contingency problems because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

[2] We assume that most readers face similar problems and can sympathize with our agent. We apologize to owners of modem, efficient home appliances who cannot take advantage of this pedagogical device.

4.3.2 AND—OR search trees

The next question is how to find contingent solutions to nondeterministic problems. As in Chapter 3, we begin by constructing search trees, but here the trees have a different character. In a deterministic environment, the only branching is introduced by the agent's own choices in each state. We call these nodes **OR nodes.** In the vacuum world, for example, at an OR node the agent chooses *Left or Right or Suck.* In a nondeterministic environment, branching is also introduced by the *environment's* choice of outcome for each action. We call these nodes **AND** nodes. For example, the *Suck* action in state l leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 *and* for state *7.* These two kinds of nodes alternate, leading to an AND–OR **tree** as illustrated in Figure 4.10.

OR NODE

AND NODE

AND–OR TREE

A solution for an AND–OR search problem is a subtree that (1) has a goal node at every leaf, (2) specifies one action at each of its OR nodes, and (3) includes every outcome branch at each of its AND nodes. The solution is shown in bold lines in the figure; it corresponds to the plan given in Equation (4.3). (The plan uses if—then—else notation 10 handle the AND branches, but when there are more than two branches at a node, it might be better to use a **case**

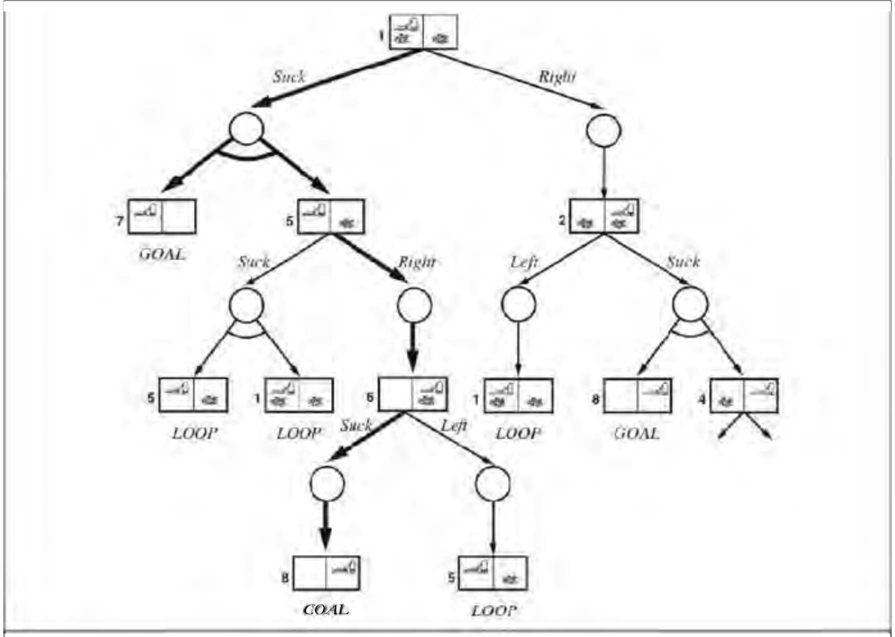


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches_ The solution found is shown in bold lines.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
   OR-SEARCH(problem.INITIAL-STATE, problem, [ ])

function OR-SEARCH(state, problem, path) returns a conditional plan., or failure
   if problem.GOAL-TEST(state) then return the empty plan
   if state is on path. then return failure
   for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state   path])
      if plan ≠ failure then return [action | plan]
   return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
   for each a, in states do
      $plan_i$ — OR-SEARCH($s_i$, problem, path)
      if $plan_i$ = failure then return failure
   return [if $s_1$ then plan., else if $s_2$ then $plan_2$ else . . . if $s_{n-1}$ then $plan_{n-1}$ else $plan_n$]
```

Figure **4.11** An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list L)

construct) Modifying the basic problem-solving agent shown in Figure **3.1** to execute contingent solutions of this kind is straightforward. One may also consider a somewhat different agent design, in which the agent can act *before* it has found a guaranteed plan and deals with some contingencies only as they arise during execution. This type of **interleaving** of search and execution is also useful for exploration problems (see Section 4.5) and for game playing (see Chapter 5).

INTERLEAVING

Figure **4.11** gives a recursive, depth-first algorithm for AND—OR graph search. One key aspect of the algorithm is the way in which it deals with cycles, which often arise in nondeterministic problems (e.g., if an action sometimes has no effect or if an unintended effect can be corrected). If the current state is identical to a state on the path from the root ; then it returns with failure. This doesn't mean that there is *no* solution from the current state; it simply means that if there *is* a noncyclic solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded. With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state. Notice that the algorithm does not check whether the current state is a repetition of a state on some *other* path from the root, which is important for efficiency. Exercise 4.5 investigates this issue.

AND—OR graphs can also be explored by breadth-first or best-first methods. The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence, but the notion of admissibility carries over and there is an analog of the A* algorithm for finding optimal solutions. Pointers are given in the bibliographical notes at the end of the chapter.

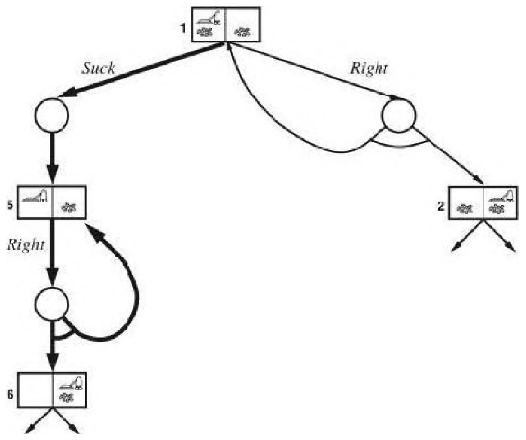


Figure 4.12 Part of the search graph for the slippery vacuum world, where we Ewe shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

4.3.3 Try, try again

Consider the slippery vacuum world, which is identical to the ordinary (non-erratic) vacuum world except that movement actions sometimes fail, leaving the agent in the same location. For example, moving *Right in state* 1 leads to the state set {1, 2}. Figure 4.12 shows part of the search graph; clearly, there are no longer any acyclic solutions from state **I,** and

CYCLE SOLUTION

LABEL

AND-OR-GRAPH-SEARCH would return with failure. There is, however, a cyclic **solution,** which is to keep trying *Right* until it works. We cart express this solution by adding a **label** to denote some portion of the plan and using that label later instead of repeating the plan itself. Thus, our cyclic solution is

$$[Suck, Ll : Right, \textbf{if } State = 5 \textbf{ then } L_1 \textbf{ else } Suck] .$$

(A better syntax for the looping part of this plan would be "while *State* = 5 **do** *Right*.") In general a cyclic plan may be considered a solution provided that every leaf is a goal state and that a leaf is reachable from every point in the plan. The modifications needed to AND-OR-GRAPH-SEARCH are covered in Exercise 4.6. The key realization is that a loop in the state space back to a state *L* translates to a loop in the plan back to the point where the subplan for state *L* is executed.

Given the definition of a cyclic solution, an agent executing such a solution **will** eventually reach the goal *provided that each outcome of a nondeterministic action eventually occurs.* Is this condition reasonable? It depends on the reason for the nondeterminism. If the action rolls a die, then it's reasonable to suppose that eventually **a six** will be rolled. If the action is to insert a hotel card key into the door lock, but it doesn't work the first time, then perhaps it will eventually work, or perhaps one has the wrong key (or the wrong room!). After seven or eight tries, most people will assume the problem is with the key and will go back to the front desk to get a new one. One way to understand this decision is to say that the initial problem formulation (observable, nondeterministic) is abandoned in favor of a different formulation (partially observable, deterministic) where the failure is attributed to an unobservable property of the key. We have more to say on this issue in Chapter 13.

4.4 SEARCHING WITH PARTIAL OBSERVATIONS

We now turn to the problem of partial observability, where the agent's percepts do not suffice to pin down the exact state. As noted at the beginning of the previous section, if the agent is in one of several possible states, then an action may lead to one of several possible outcomes—even *if the environment is deterministic.* The key concept required for solving partially observable problems is the **belief state.** representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point. We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as nondeterministic actions.

BELIEF STATE

4.4.1 Searching with no observation

When the agent's percepts provide *no information at all,* we have what is called a **sensor. less problem or** sometimes a **conformant problem.** At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable. Moreover, sensorless agents can be surprisingly useful, primarily because they *don't* rely on sensors working properly. In manufacturing systems. for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all. The high cost of sensing is another reason to avoid it: for example, doctors often prescribe a broad-spectrum antibiotic rather than using the contingent plan of doing an expensive blood test. then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.

SENSORLESS

CONFORMANT

We can make a sensorless version of the vacuum world. Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of dirt. In that case, its initial state could be any element of the set {1, 2, 3, 4, 5,6, 7, 8}. Now, consider what happens if it tries the action *Right.* This will cause it to be in one of the states {2, 4, 6, 8}—the agent now has more information! Furthermore, the action sequence [*Right*,*Suck*] will always end up in one of the states {4, 8}. Finally, the sequence [*Right*,*Suck*,*Left*,*Suck*] is guaranteed to reach the goal state *7* no matter what the start state. We say that the agent can **coerce** the world into state 7.

COERCION

To solve sensorless problems, we search in the space of belief states rather than physical states. [1] Notice that in belief-state space, the problem is *fully observable* because the agent

In a fully observable environment, each belief state contains one physical state. Thus, we can view the algorithms in Chapter 3 as searching in a belief-state space of singleton belief states.

always knows its own belief state. Furthermore, the solution (if any) is always a sequence of actions. This is because, as in the ordinary problems of Chapter 3, the percepts received after each action are completely predictable—they're always empty! So there are no contingencies to plan for. This is true *even if the environment is nondeterminstic.*

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by ACTIONS$_P$, RESULT$_P$, GOAL-TEST$_P$, and STEP-COST p. Then we can define the corresponding sensorless problem as follows:

- Belief states: The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2N states, although many may be unreachable from the initial state.
- **Initial state:** Typically the set of all states in P, although in some cases the agent will have more knowledge than this.
- Actions: This is slightly tricky. Suppose the agent is in belief state $b = \{s_1,$ s2}, but ACTIONS$_P(s_1)$ ACTIONS$_P(s_2)$; then the agent is unsure of which actions are legal. If we assume that illegal actions have no effect on the environment, then it is safe to take the *anion* of all the actions in any of the physical states in the current belief state b:

  $$ACTIONS(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s) .$$

  On the other hand, **if an illegal** action might be the end of the world, it is safer to allow only the *intersection,* that is, the **set** of actions legal in *all* the states. For the vacuum world, every state has **the** same legal actions, so both methods give the same result.
- Transition model: The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

  $$= \text{RESULT}(b, a) = \quad : .s^r = \text{RESULT}_P(s, a) \text{ and } s\, E\, b\} . \qquad (4.4)$$

  With deterministic actions, b' is never larger than b. With nondeterminism, we have

  $$= \text{RESULT}(b, a) = \quad : \quad E\ \text{RESULTS}_P(s, a) \text{ and } s\ E\ b\}$$
  $$= \bigcup_{s \in b} \text{RESULTS}_P(s, a) ,$$

  which may be larger than b, as shown in Figure 4.13. The process of generating the new belief state after the action is called the **prediction** step; the notation $b' =$ PREDICT$_P(b, a)$ will come in handy.

PREDICTION

- **Goal test:** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if *all* the physical states in it satisfy GCAL-TESTp. The agent may *accidentally* achieve the goal earlier, but it won't *know* that it has done so.
- **Path cost: This is** also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values. (This gives rise to a new class of problems, which we explore in Exercise 4.9.) For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

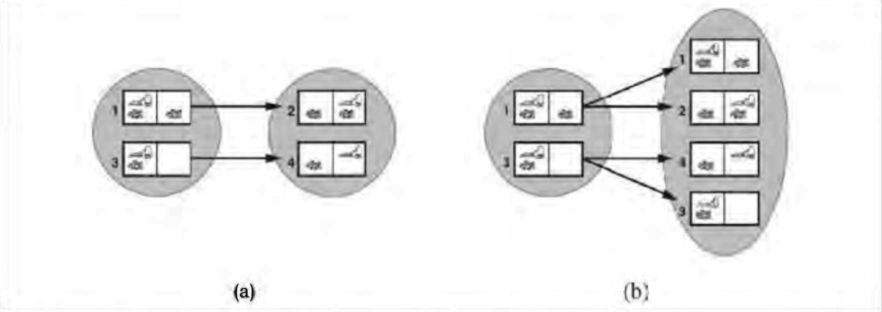


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Figure 4.14 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of $2^8 = 256$ possible belief states.

The preceding definitions enable the automatic construction of the belief-state problem formulation from the definition of the underlying physical problem. Once this is done, we can apply any of the search algorithms of Chapter 3. In fact, we can do a little bit more than that. In "ordinary" graph search, newly generated states are tested to see if they are identical to existing states. This works for belief states, too; for example, in Figure 4.14, the action sequence [*Suck*,*Left*,*Suck*] starting at the initial state reaches the same belief state as [*Right*,*Left*,*Suck*], namely, {5, 7}. Now, consider the belief state reached by [*Left*], namely, {1, 3, 5, 7}. Obviously, this is not identical to {5, 7}, but it is a *superset*. It is *easy to* prove (Exercise 4.8) that if an action sequence is a solution for a belief state *b*, it is also a solution for any subset of *b*. Hence, we can discard a path reaching { 1, 3, 5, 7} if {5, 7} has already been generated. Conversely, if {1, 3, 5, 7} has already been generated and found to be solvable, then any *subset*, such as {5, 7}, is guaranteed to he solvable, This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Even with this improvement, however, sensorless problem-solving as we have described it is seldom feasible in practice. The difficulty is not so much the vastness of the belief-state space—even though it is exponentially larger than the underlying physical state space; in most cases the branching factor and solution length in the belief-state space and physical state space are not so different. The real difficulty lies with the size of each belief state. For example, the initial belief state for the 10 x 10 vacuum world contains 100 x 2^{100} or around 10^{32} physical states—far too many if we use the atomic representation, which is an explicit List of states.

One solution is to represent the belief state by some more compact description. *In* English, we could say the agent knows "Nothing" in the initial state; after moving *Left,* we could say, "Not in the rightmost column," and so on. Chapter 7 explains how to do this in a formal representation scheme. Another approach is to avoid the standard search algorithms, which treat belief states as black boxes just like any other problem state. Instead, we can look

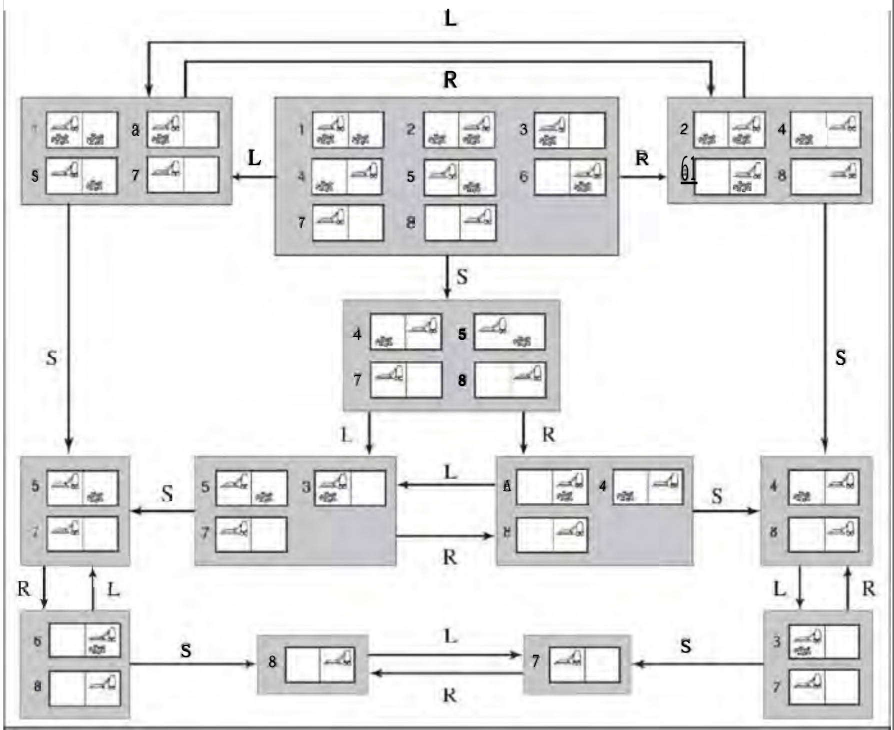


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief stale. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled links. Self-loops are omitted for clarity.

INCREMENTAL BELIEF-STATE SEARCH

inside the belief states and develop **incremental belief-state search** algorithms that build up the solution one physical state at a time. For example, in the sensorless vacuum world, the initial belief state is {1,2,3,4,5.6,7.8}, and we have to find an action sequence that works in **all** 8 states. We can do this by first finding a solution that works for state I; then we check if it works for state 2; if not, go back and find a different solution for state 1, and so on. Just as an AND—OR search has to find a solution for every branch at an **Alen** node, this algorithm has to find a solution for every state in the belief state; the difference is that AND—OR search can find a different solution for each branch, whereas an incremental belief-state search has to **find** *one* solution that works for *all* the states.

The main advantage of the incremental approach is that it is typically able to detect failure quickly—when a belief state is unsolvable, it is usually the case that a small subset of the belief state, consisting of the first few states examined, is also unsolvable. In some cases, this leads to a speedup proportional to the size of the belief states, which may themselves be as large as the physical state space itself.

Even the most efficient solution algorithm is not of much use when no solutions exist. Many things just cannot be done without sensing. For example, the sensorless 8-puzzle is impossible. On the other hand, a little bit of sensing can go a long way. For example, every 8-puzzle instance is solvable if just one square is visible—the solution involves moving each tile in turn into the visible square and then keeping track of its location.

4.4.2 Searching with observations

Fora general partially observable problem. we have to specify how the environment generates percepts for the agent. For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares. The formal problem specification includes a PERCEPTS) function that returns the percept received in a given state. (If sensing is nondeterministic, then we use a PERCEPTS function that returns a set of possible percepts.) For example. in the local-sensing vacuum world, the PERCEPT in state 1 is [A, Dirty]. Fully observable problems are a special case in which PERCEPT$(s) = s$ for every state s, while sensorless problems are a special case in which PERCEPT(s) = *mitt.*

When observations are partial, it **will** usually be the ease that several states could have produced any given percept. For example, the percept [A, *Dirty]* is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world **will** be 1, 3}. The ACTIONS, STEP-COST, and GOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated. We can think of transitions from one belief state to the next for a particular action as occurring in three stages, as shown in Figure 4.15:

- The **prediction** stage is the same as for sensorless problems: given the action a in belief state *b*, the predicted belief state is $b =$ PREDICT(b, a).
- The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state:

$$\text{POSSIBLE-PERCEPTS}(b) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in$$

- The **update** stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_o *is* just the set of states in b that could have produced the percept:

$$b_o = \text{UPDATE}(b, o = \{s : \text{o=PERCEPT(s) and } s \text{ E } \hat{b}\} .$$

Notice that each updated belief stale b_o can be no larger than the predicted belief stale b; observations can only help reduce uncertainty compared to the sensorless case. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a *partition* of the original predicted belief state.

** Here, and throughout the book, the "hat" in b means an estimated or predicted value for 5.

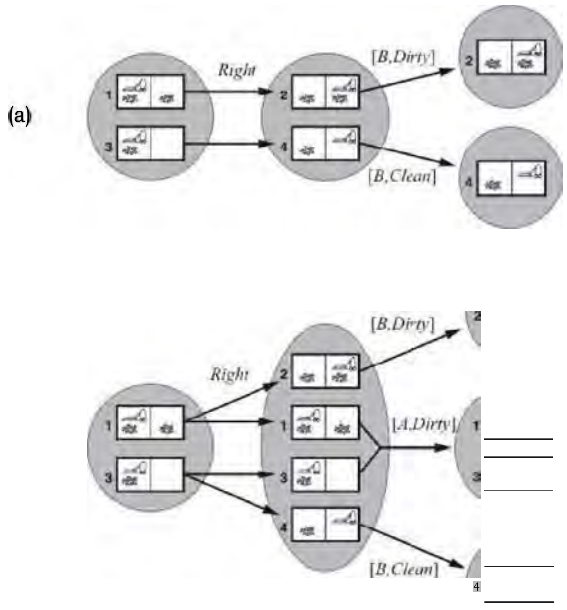


Figure 4.15 Two example of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are *[13. Dirty]* and *[B, Clean* leading to two belief states, each of winch is a singleton. (b) In the slippery world, *Right* is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are IA, *Dirty], [B, Dirty],* and *[B, Clean],* leading to three belief states as shown.

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$\text{RESULTS}(b, \quad = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, \text{a.}), o) \text{ and } o \; E \; \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\} \qquad (4.5)$$

Again, the nondeterminism in the partially observable problem comes from the inability to predict exactly which percept will be received after acting; underlying nondeterminism in the physical environment may *contribute* to this inability by enlarging the belief state at the prediction stage, leading to more percepts at the observation stage.

4.4.3 Solving partially observable problems

The preceding section showed how to derive the RESULTS function for a nondeterministic belief-state problem from an underlying physical problem and the PERCEPT function. Given

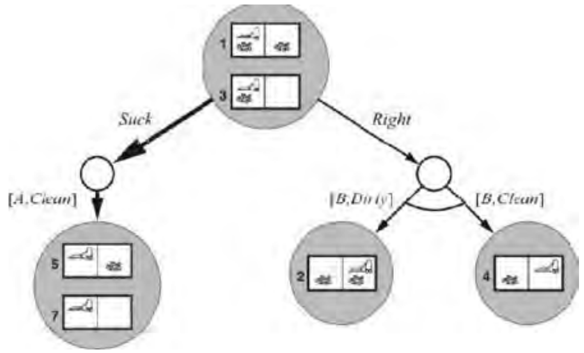


Figure 4.16 The first level of the AND—OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first step of the solution.

such a formulation, the AND—OR search algorithm of Figure 4.11 can be applied directly to derive a solution. Figure 4.16 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept *[A, Dirty].* The solution is the conditional plan

[*Suck*, *Right*, **if** *Bstate* = {6} **then** *Suck* **else** [ ]] .

Notice that, because we supplied a belief-state problem to the AND—OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state. This is as it should be: in a partially observable environment the agent won't be able to execute a solution that requires testing the actual state.

As in the case of standard search algorithms applied to sensorless problems, the AND—OR search algorithm treats belief states as black boxes, just like any other states. One can improve on this by checking for previously generated belief states that are subsets or supersets of the current state, just as for sensorless problems. One can also derive incremental search algorithms, analogous to those described for sensorless problems, that provide substantial speedups over the black-box approach.

4.4.4 An agent for partially observable environments

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent in Figure 3.1: the agent formulates a problem, calls a search algorithm (such as AND-OR-GRAPH-SEARCH) to solve it, and executes the solution. There are two main differences. First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if—then—else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly. Second, the agent will need to maintain its belief state as it performs actions and receives percepts. This process resembles the prediction–observation–update process in Equation (4.5) but is actually simpler because the percept is given by the environment rather than calculated by the

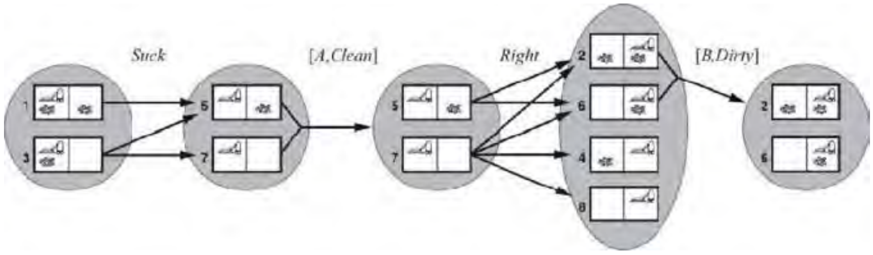


Figure 4.17 **Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum** world with local sensing.

agent. Given an initial belief *state* b, an action a, and a percept a, the new belief state is:

$$b = \text{UPDATE}(\text{PREDICT}(b, a), o) . \qquad (4.6)$$

Figure 4.17 shows the belief state being maintained in the *kindergarten* vacuum world with local **sensing, wherein any square may become dirty at any lime** unless the agent is actively cleaning it at that moment.

In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system. MONITORING This function goes under various names, including **monitoring, filtering** and state estima- FILTERING tion. Equation (4.6) is called a **recursive** state estimator because it computes the new belief STATE ESTIMATION state from the previous one rather than by examining the entire percept sequence. If the agent RECURSIVE is not to "fall behind," the computation has to happen as fast as percepts are coming in. As the environment becomes more complex, the exact update computation becomes infeasible and the agent will have to compute an approximate belief state, perhaps focusing on **the** implications of the percept for the aspects of the environment that are of current interest. Most work on this problem has been done for stochastic, continuous-state environments with the tools of probability theory, as explained in Chapter 15. Here we will show an example in a discrete environment with detrministic sensors and nondeterministic actions.

LOCALIZATION The example concerns a robot with the task of **localization:** working out where it is, given a map of the world and a sequence of percepts and actions. Our robot is placed in the mare-like environment of Figure 4.18. **The robot is** equipped with four sonar sensors that tell whether there is an obstacle—the outer wall or a black square in the figure—in each of the four compass directions. We assume that the sensors give perfectly correct data, and that the robot has a correct map of the enviomment. But unfortunately the robot's navigational system is broken, so when it executes a *Move* action, it moves randomly to one of the adjacent squares. The robot's task is to determine its current location.

Suppose the robot has just been switched on, so it does not know where it is. Thus its initial belief state b consists of the set of all locations. The the robot receives the percept

The usual apologies to those who are unfamiliar with the effect of small children on the environment.

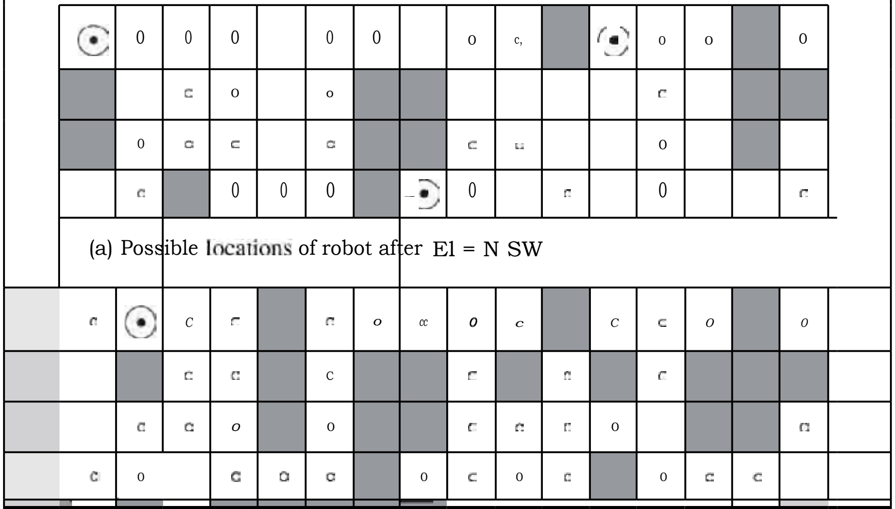

(a) Possible locations of robot after E1 = N SW
(b) Possible locations of robot After E1 = NSW, E 2 = N S

Figure 4.18 Possible positions of the robot, *O,* (a) after one observation El = *NSW* and (b) after a second observation E2 = *NS.* When sensors are noiseless and the transition model is accurate, there are no other possible locations for the robot consistent with this sequence of two observations.

NSW, meaning there are obstacles to the north, west, and south, and does an update using the equation b_o = UPDATE(b), yielding the 4 locations shown in Figure 4.18(a). You can inspect the maze to see that those are the only four locations that yield the percept *NWS*.

Next the robot executes a *Move* action, but the result is nondeterministic. The new belief state, b_e = PREDICT{b_o, *Move),* contains all the locations that are one step away from the locations in h_0. When the second percept, *NS,* arrives, the robot does UPDATE(b_a, *NS)* and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b). That's the only location that could be the result of

UPDATE(PREDICT(UPDATE(*b*, *NSW),* *Move*), *NS)*

With nondetermnistic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information. Sometimes the percepts don't help much for localization: If there were one or more long cast-west corridors, then a robot could receive a long sequence of *NS* percepts, but never know where in the corridor(s) it was.

4.5 ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

°FUME SEARCH

ONLINE SEARCH

So far we have concentrated on agents that use offline search algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, an **online search** agent **interleaves** computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic or semidynamic domains—domains where there is a penalty for sitting around and computing too long. Online search is also helpful in nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that *might* happen but probably won't. Of course, there is a tradeoff: the more an agent plans ahead, the less often it will find itself up the creek without a paddle.

EXPLORATION PROBLEM

Online search is a *necessary* idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an **exploration problem** and must use its actions as experiments in order to learn enough to make deliberation worthwhile.

The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to *B.* Methods for escaping from labyrinths—required knowledge for aspiring heroes of antiquity—are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however. Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

4.5.1 Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment (Chapter 17 relaxes these assumptions), but we stipulate that the agent knows only the following:

- ACTIONS(s), which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$—note that this cannot be used until the agent knows that s' is the outcome; and
- GOAL-TEST(s).

Note in particular that the agent *cannot* determine RESULT(s, a) except by actually being in s and doing a. For example, in the maze problem shown in Figure 4.19, the agent does not know that going *Up* from (1,1) leads to (1,2); nor, having done that, does it know that going *Down* will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

[*] The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received rather than waiting for the entire input data set to become available.

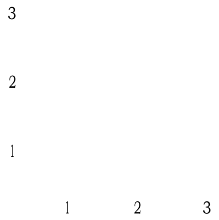


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

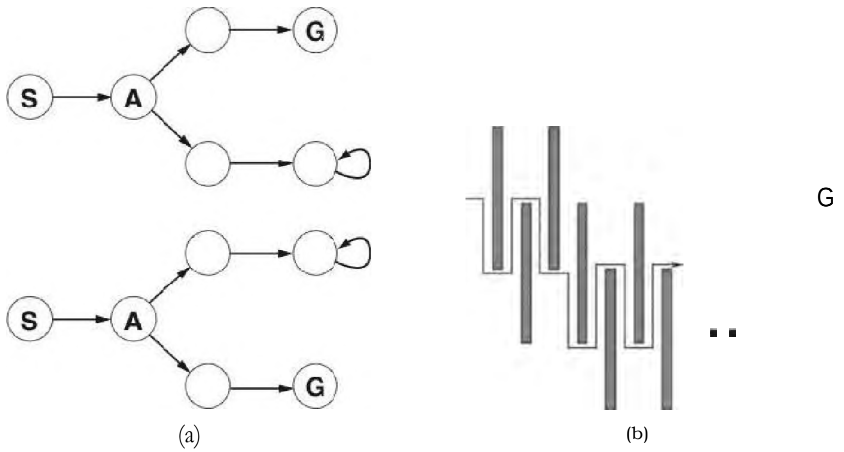


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Finally, the agent might have access to an admissible heuristic function $h(s)$ that estimates the distance from the current state to a goal state. For example, in Figure 4.19, the agent might know the location of the goal and be able to use the Manhattan-distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow *if it knew the search space in advance—that is, the* actual shortest path (or shortest complete exploration). In the language of online algorithms, this is called the competitive **ratio;** we would like it to be as small as possible.

COMPETITIVE RATIC

IRREVERSIBLE

DEW END

Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some eases. For example, if some actions arc **irreversible**-i.e., they lead to a state from which no action leads back to the previous state—the online search might accidentally reach a **dead-end state** from which no goal state is reachable. Perhaps the term "accidentally" is unconvincing—after all, there might be an algorithm that happens not to take the dead-end path as it explores_ Our claim, to be more precise, is that *no algorithm can avoid dead ends in al! state spaces.* Consider the two dead-end state spaces in Figure 4.20(a). To an online search algorithm that has visited states *S* and A, the two state spaces look *identical,* so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an **adversary argument—we** can imagine an adversary constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses.

ADVERSARY ARGUMENT

Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we simply assume that the state space is **safely explorable—that** is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

SAFELY EXPLORABLE

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure 4.20(h) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

4.5.2 Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have *seen* previously. For example, offline algorithms such as A^5 can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a *local* order. Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure 4.21. This agent stores its map in a table, RESULT[*s*, a], that records the state resulting from executing action a in state *a.* Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. In depth-first search, this means going back to the state from which the agent most recently entered the current state. To achieve that, the algorithm keeps a table that

```
function ONLINE-DFS-AGENT($s'$) returns an action
  inputs: $s'$, a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              $s$, a, the previous state and action, initially null

  if GOAL-TEST($s'$) then return stop
  if a' is a new state (not in untried) then untried[$s'$]     ACTIONS($s$ )
  if s is not null then
      result[$s$, a] ← $s'$
      add $s$ to the front of unbacktracked[$s'$]
  if untried[$s'$] is empty then
      if unbacktracked[$s'$] is empty then return stop
      else a 4— an action $b$ such that result[$s'$  $b$] = POP(unbacktracked[$s'$])
  else $a$ ← POP(untried[$s'$])
  $s$ ←
  return a
```

Figure 4.21 **An online** search agent that uses depth-first exploration. The agent is applicable only in stale spaces in which every action can be "undone" by some other action.

lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure 4.19. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice. For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS -AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

4.5.3 Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is *already* an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

RANDOM WALK

Instead of random restarts, one might consider using a **random walk to explore the** environment. A random walk simply selects at random one of the available actions from the

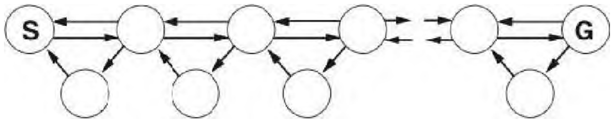


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

current state; preference cars be given to actions that have nut yet been tried. It is easy to prove that a random walk will *eventually* find a goal or complete its exploration, provided that the space is finite." On the other hand, the process can be very stow. Figure 4.22 shows an environment in which a random walk will take exponentially many steps to find the goal because, at each step, backward progress is twice as likely as forward progress. The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.

Augmenting hill climbing with *memory* rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.23 shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor a' is the cost to get to s' plus the estimated cost to get to a goal from there—that is, $c(s, a, s') + H(s')$. In the example, there are two actions, with estimated costs 1 + 9 and 1 + 2, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from **a** goal, so its H should be updated accordingly, as shown in Figure 4.23(b). Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

LRTA'

An agent implementing this scheme, which is called learning real-time A (**LRTA***), is shown in Figure 4.24. Like ONLINE-DFS-AGENT, it builds a map of the environment in the *result* table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state *a* are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$. This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

OPTIMISM UNDER LICERTAINTY

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states **in** $O(n$] steps in the worst case,

[14] Random walks am complete on infinite one-dimensional and two-dimensional grids. Oa a three-dimensional grid, the probability that the walk ever returns to the starting point is only about 0.3405 (Hughes, 1995).

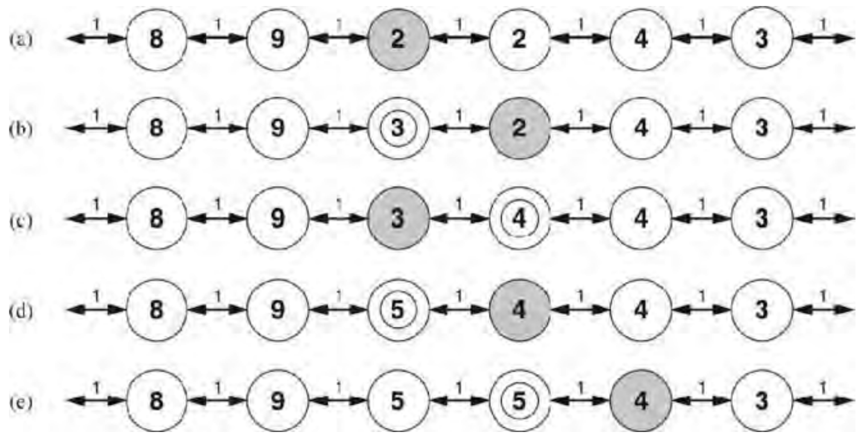


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
  inputs: .s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              a, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s']    h(s')
  if s is not null
     result[s , a] ⁴— a'
                  min    LRTA*-COST(s, b, result[s,  H)
           ∈ACTIONS(s)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s , b, result's', b], H)
   ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h (s)
  else return c(s, a, .s')      H[s']
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

but often does much better. The LRTA* agent is just one of a large family of online agents that one can define by spccifying the action selection rule and the update rule in different ways_ We discuss this family, developed originally for stochastic environments, in Chapter 21.

4.5.4 Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA*. In Chapter 21, we show that these updates eventually converge to *exact* values for every state. provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the lowest-cost successor—that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment of Figure 4.19, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,1) to (1,2), the agent still has no idea that the *Down* action goes back to (1,1) or that the *Up* action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that *Up* increases the y-coordinate unless there is a wall in the way, that *Dawn* reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far we have hidden the information inside the black box called the RESULT function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent. These are covered in Chapter 18.

4.6 SUMMARY

This chapter has examined search algorithms for problems beyond the "classical" case of finding the shortest path to a goal in an observable, deterministic, discrete environment

- *Local search* methods such as **hill climbing** operate on complete-state formulations, keeping only a small number of nodes in memory. Several stochastic algorithms have been developed, including **simulated annealing,** which returns optimal solutions when given an appropriate cooling schedule.
- Many local search methods apply also to problems in continuous spaces. **Linear pro. gramming** and **convex optimization** problem; obey certain restrictions on the shape of the state space and the nature of the objective function, and admit polynomial-time algorithms that are often extremely efficient in practice_
- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover, which combines pairs of states from the population.