CHAPTER 1

INTRODUCTION

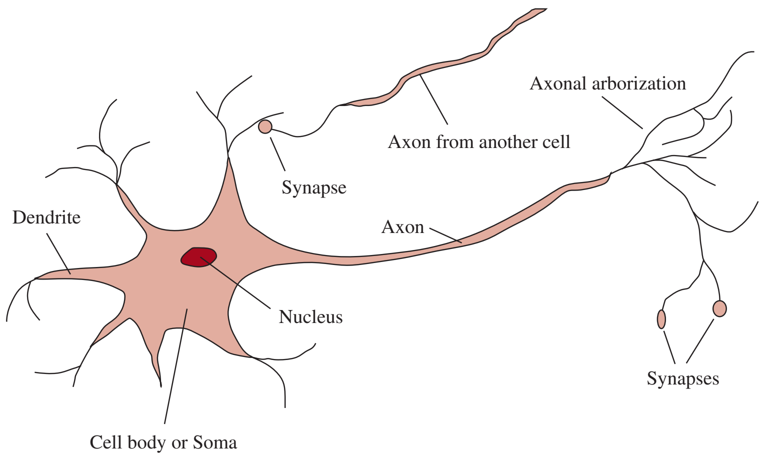


Figure 1.1 The parts of a nerve cell or neuron. Each neuron consists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. The axon stretches out for a long distance, much longer than the scale in this diagram indicates. Typically, an axon is 1 cm long (100 times the diameter of the cell body), but can reach up to 1 meter. A neuron makes connections with 10 to 100,000 other neurons at junctions called synapses. Signals are propagated from neuron to neuron by a complicated electrochemical reaction. The signals control brain activity in the short term and also enable long-term changes in the connectivity of neurons. These mechanisms are thought to form the basis for learning in the brain. Most information processing goes on in the cerebral cortex, the outer layer of the brain. The basic organizational unit appears to be a column of tissue about 0.5 mm in diameter, containing about 20,000 neurons and extending the full depth of the cortex (about 4 mm in humans).

	Supercomputer	Personal Computer	Human Brain
Computational units	10^6 GPUs + CPUs	8 CPU cores	10^6 columns
	10^{15} transistors	10^{10} transistors	10^{11} neurons
Storage units	10^{16} bytes RAM	10^{10} bytes RAM	10^{11} neurons
	10^{17} bytes disk	10^{12} bytes disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-9} sec	10^{-3} sec
Operations/sec	10^{18}	10^{10}	10^{17}

Figure 1.2 A crude comparison of a leading supercomputer, Summit (Feldman, 2017); a typical personal computer of 2019; and the human brain. Human brain power has not changed much in thousands of years, whereas supercomputers have improved from megaFLOPs in the 1960s to gigaFLOPs in the 1980s, teraFLOPs in the 1990s, petaFLOPs in 2008, and exaFLOPs in 2018 (1 exaFLOP = 10^{18} floating point operations per second).

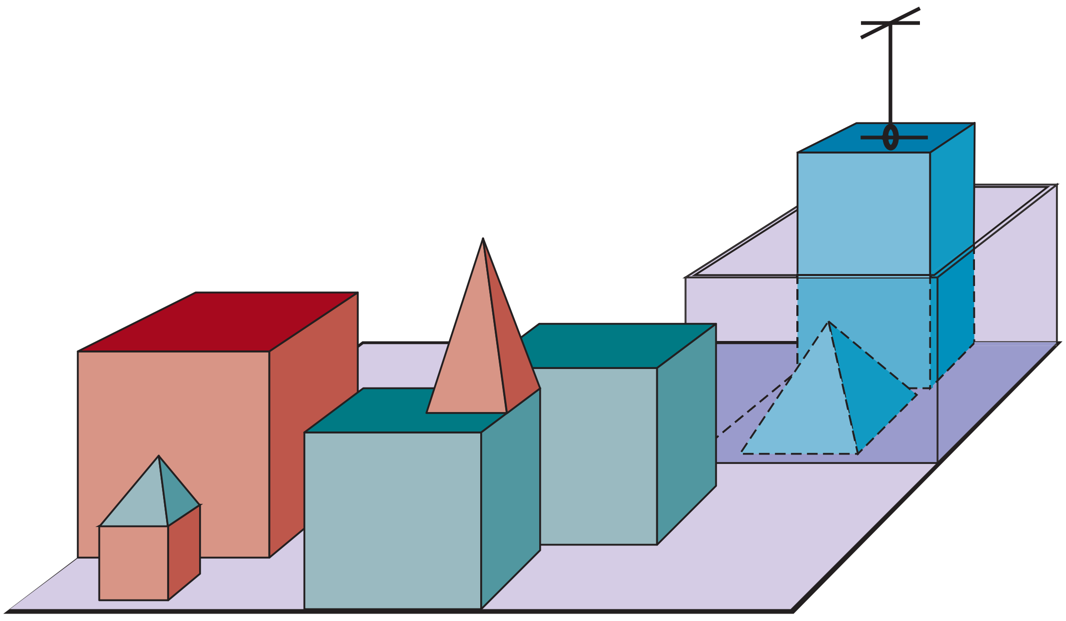


Figure 1.3 A scene from the blocks world. SHRDLU (Winograd, 1972) has just completed the command "Find a block which is taller than the one you are holding and put it in the box."

CHAPTER 2

INTELLIGENT AGENTS

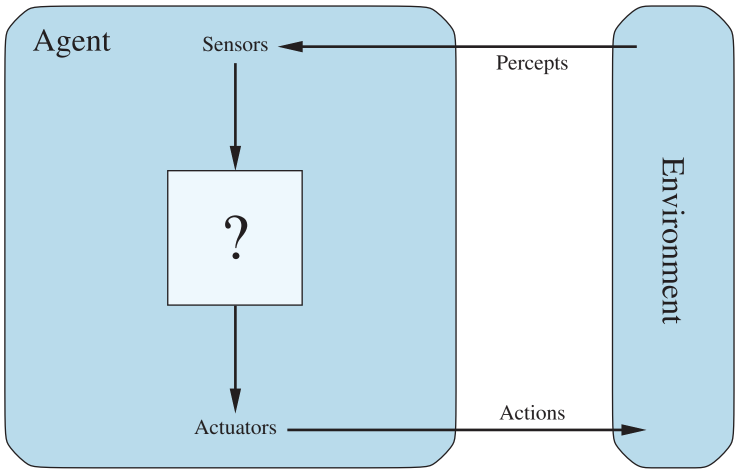


Figure 2.1 Agents interact with environments through sensors and actuators.

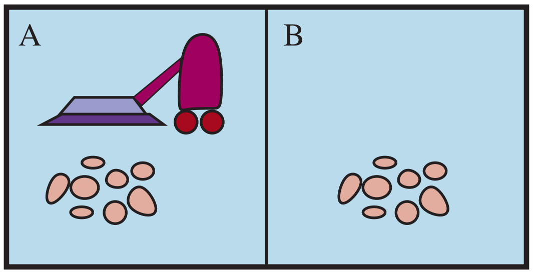


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always succeed, and so on.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean]$, $[A, Clean]$	*Right*
$[A, Clean]$, $[A, Dirty]$	*Suck*
$\vdots$	$\vdots$
$[A, Clean]$, $[A, Clean]$, $[A, Clean]$	*Right*
$[A, Clean]$, $[A, Clean]$, $[A, Dirty]$	*Suck*
$\vdots$	$\vdots$

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2. The agent cleans the current square if it is dirty, otherwise it moves to the other square. Note that the table is of unbounded size unless there is a restriction on the length of possible percept sequences.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

Figure 2.4 PEAS description of the task environment for an automated taxi driver.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Figure 2.5 Examples of agent types and their PEAS descriptions.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.

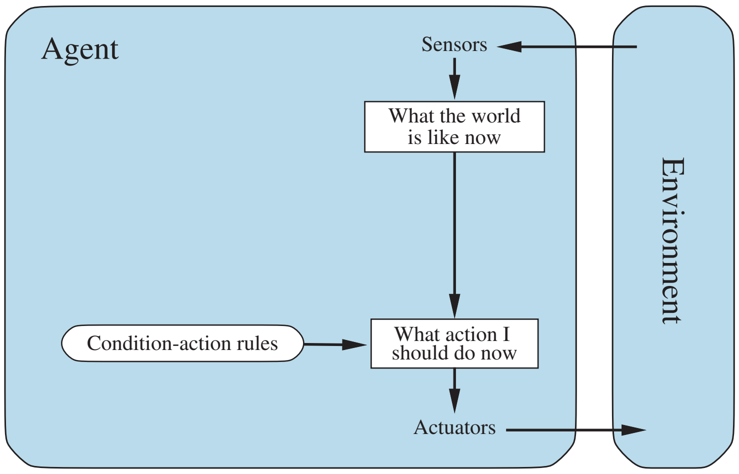


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
  persistent: rules, a set of condition–action rules

  state ← INTERPRET-INPUT(percept)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

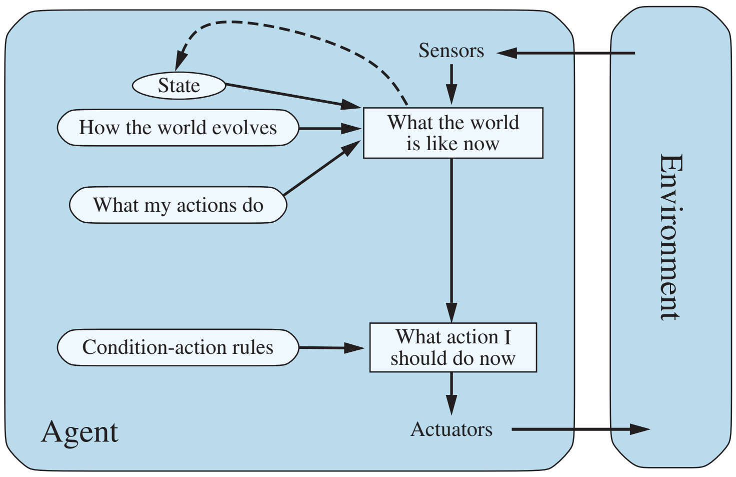


Figure 2.11 A model-based reflex agent.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
   persistent: state, the agent's current conception of the world state
               transition_model, a description of how the next state depends on
                      the current state and action
               sensor_model, a description of how the current world state is reflected
                      in the agent's percepts
               rules, a set of condition–action rules
               action, the most recent action, initially none

   state ← UPDATE-STATE(state, action, percept, transition_model, sensor_model)
   rule ← RULE-MATCH(state, rules)
   action ← rule.ACTION
   return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

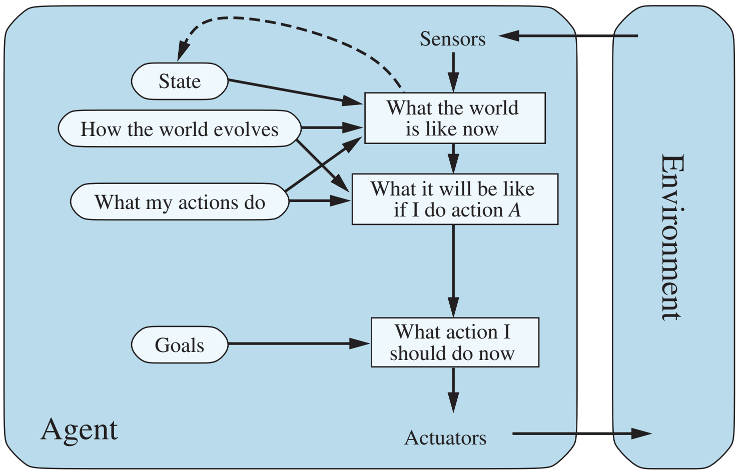

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

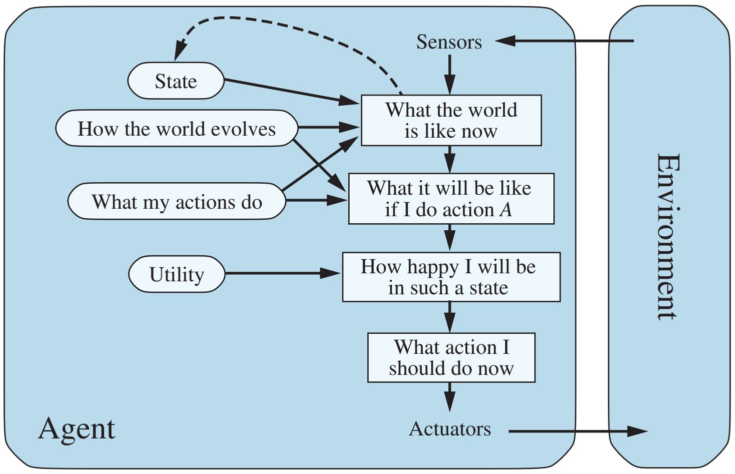

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

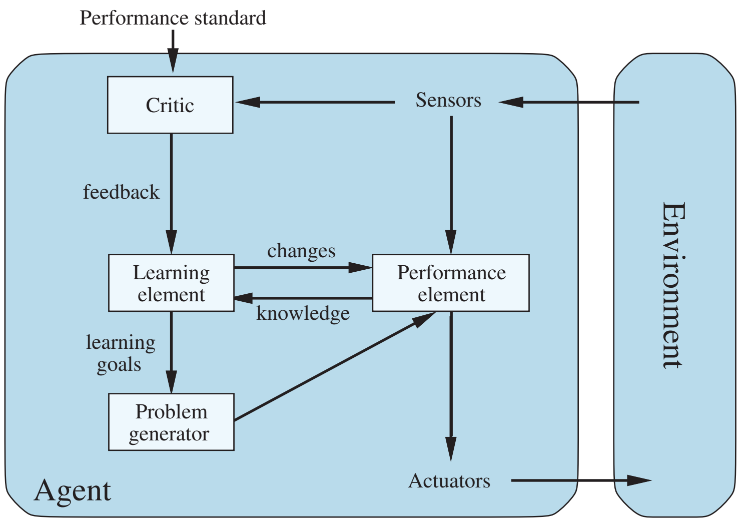


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

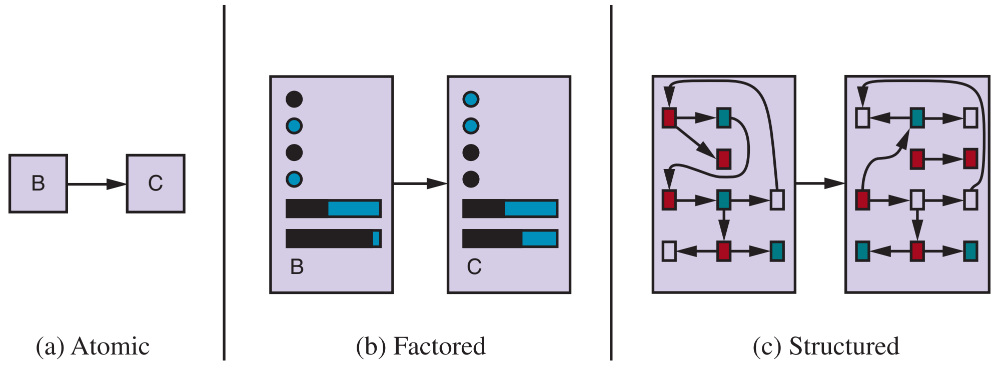


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

CHAPTER 3

SOLVING PROBLEMS BY SEARCHING

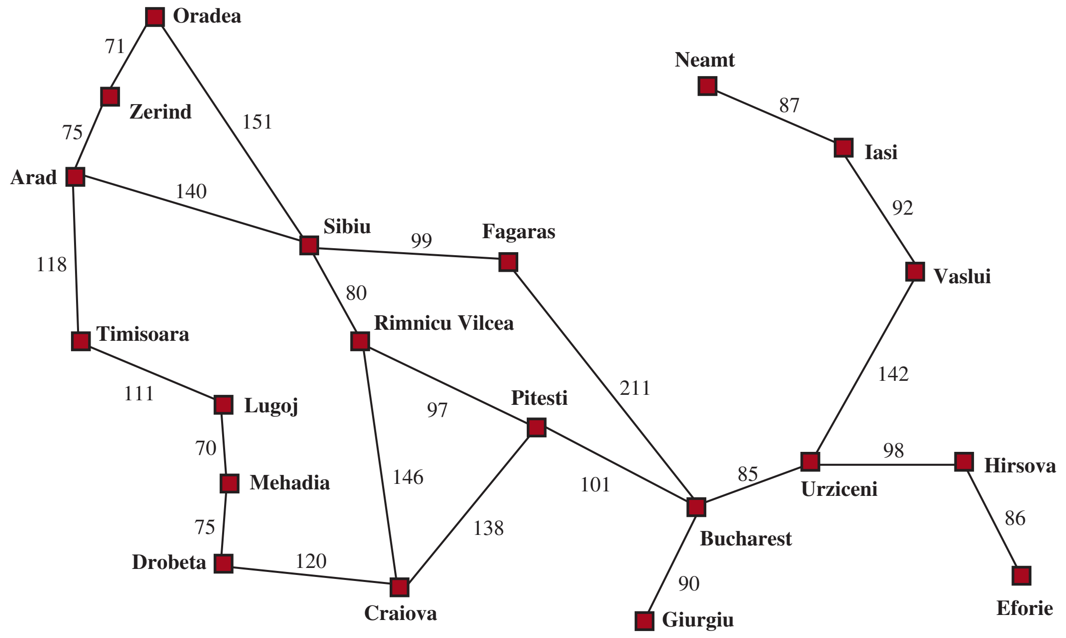


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

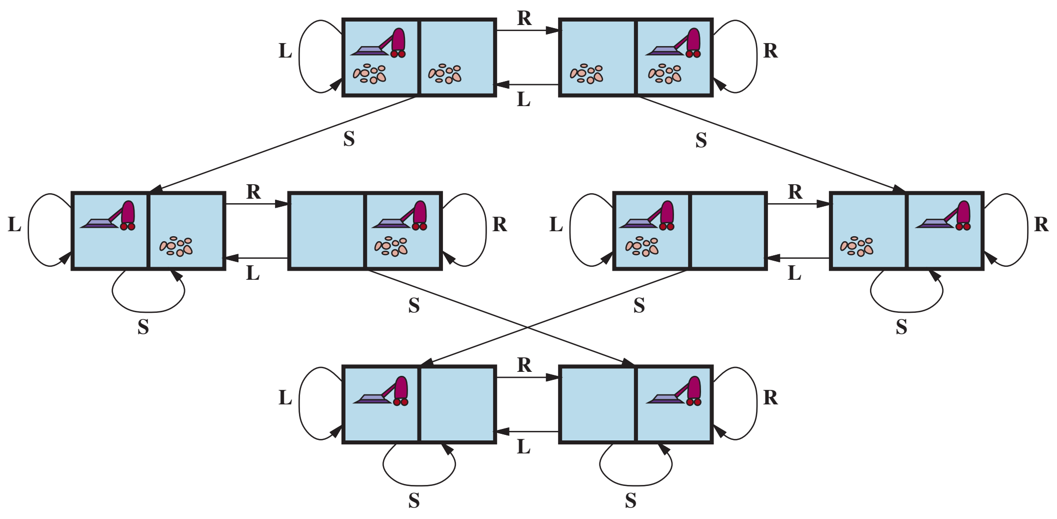


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

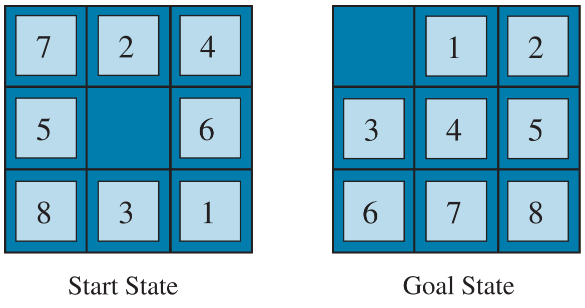


Figure 3.3 A typical instance of the 8-puzzle.

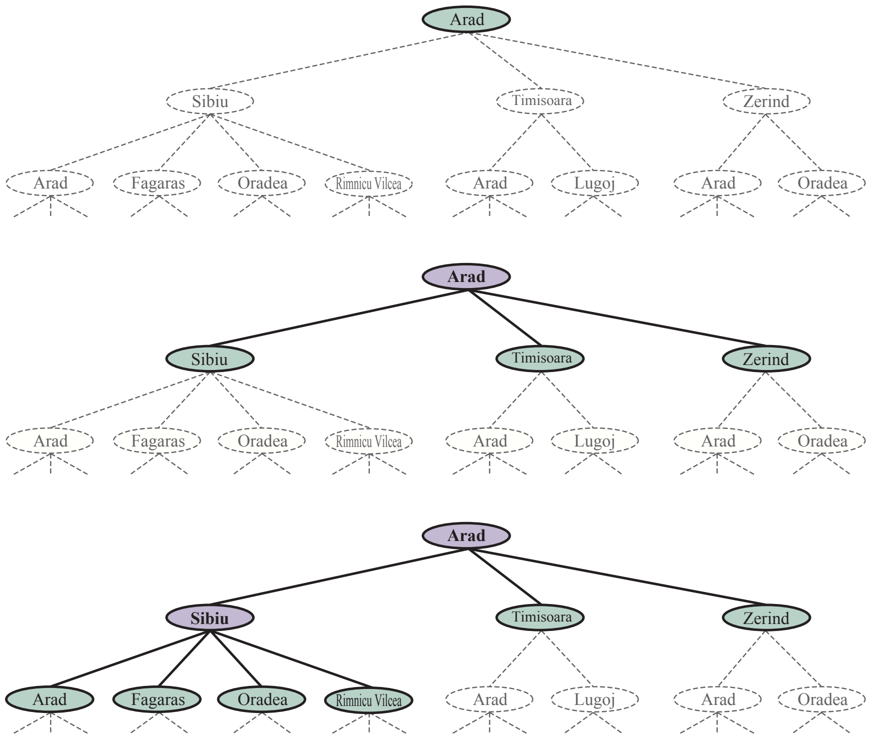


Figure 3.4 Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

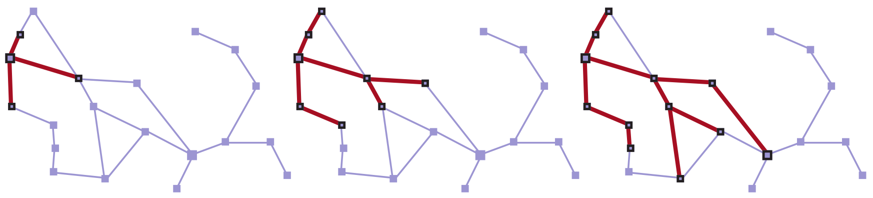


Figure 3.5 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

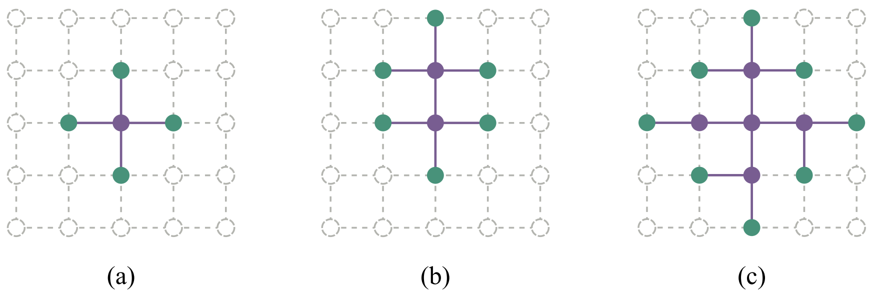


Figure 3.6 The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

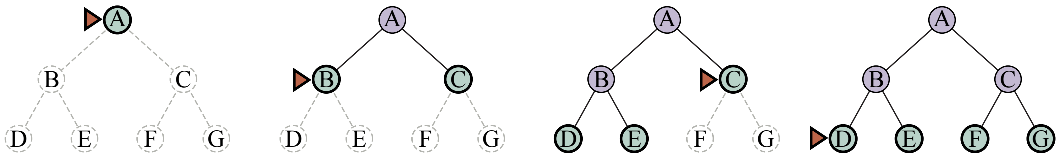


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

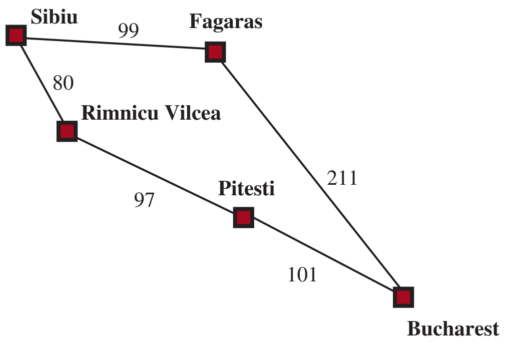


Figure 3.10 Part of the Romania state space, selected to illustrate uniform-cost search.

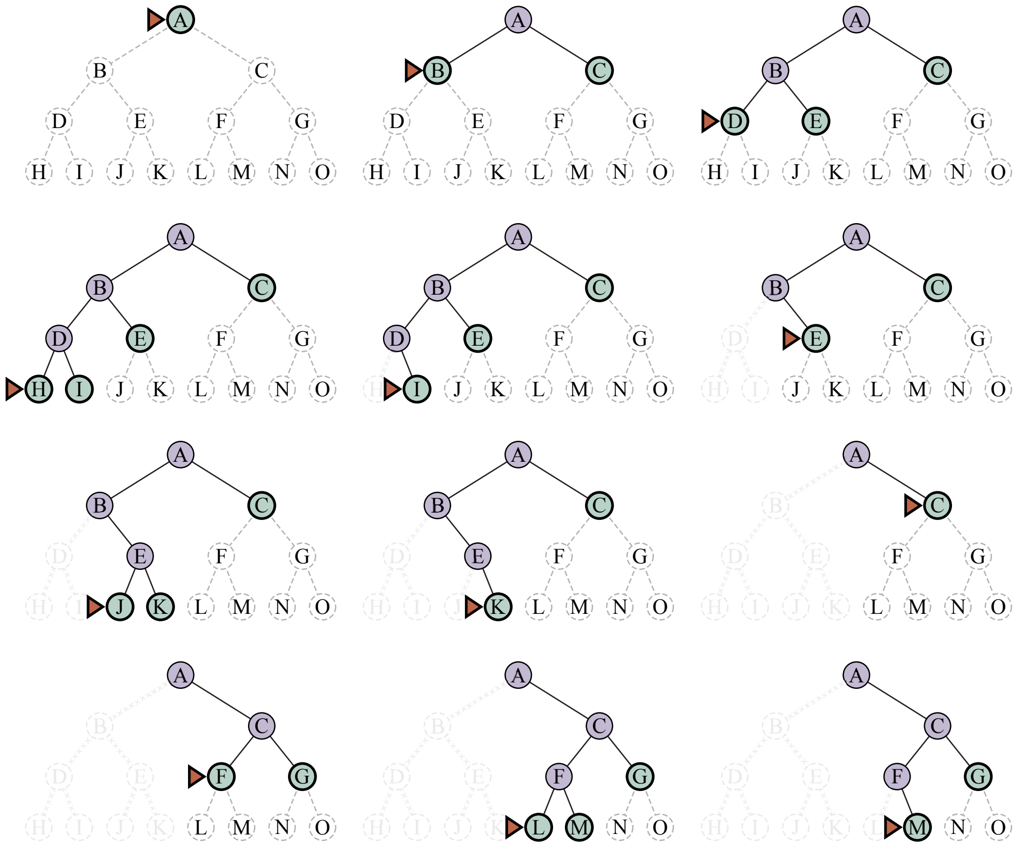


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

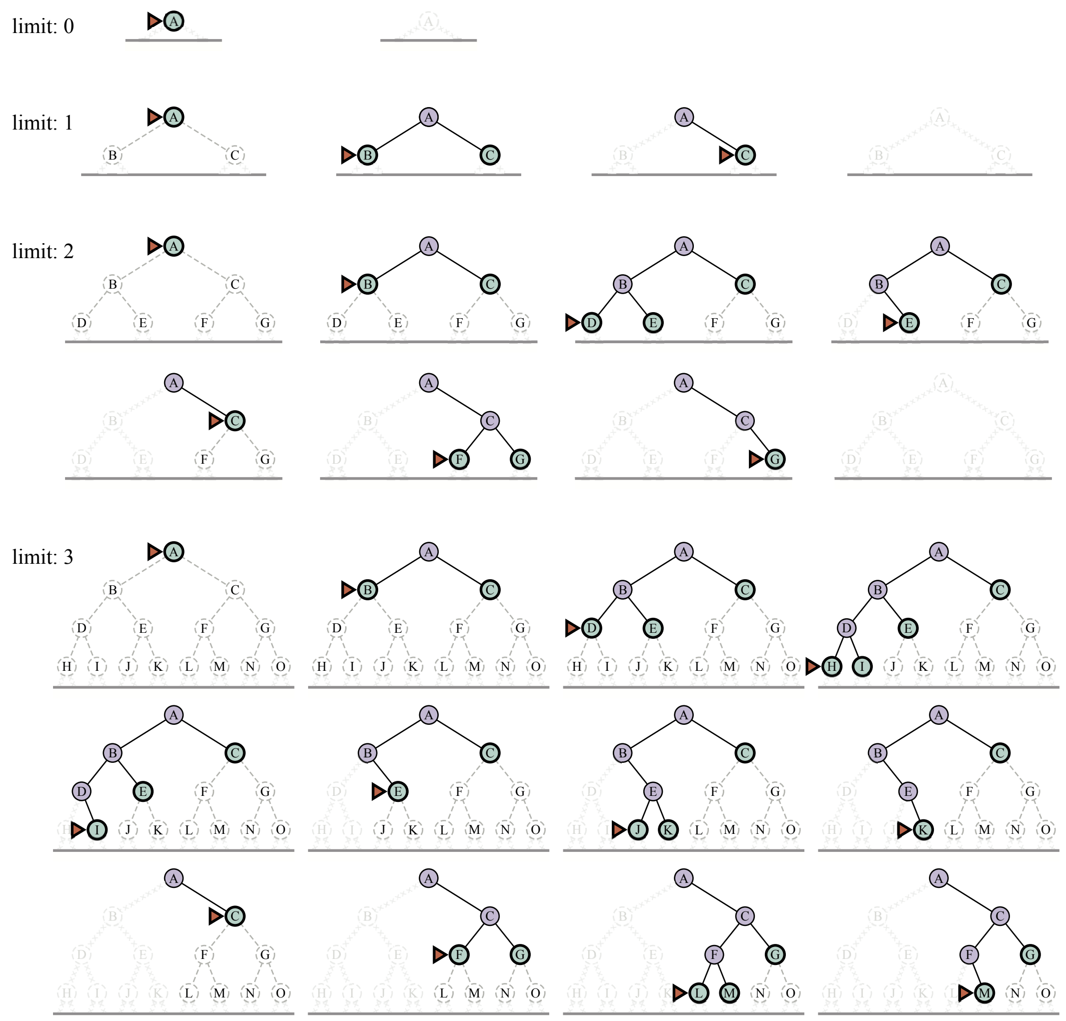


Figure 3.13 Four iterations of iterative deepening search for goal M on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

```
function BIBF-SEARCH($problem_F$, $f_F$, $problem_B$, $f_B$) returns a solution node, or failure
  $node_F$ ← NODE($problem_F$.INITIAL)                    // Node for a start state
  $node_B$ ← NODE($problem_B$.INITIAL)                    // Node for a goal state
  $frontier_F$ ← a priority queue ordered by $f_F$, with $node_F$ as an element
  $frontier_B$ ← a priority queue ordered by $f_B$, with $node_B$ as an element
  $reached_F$ ← a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B$ ← a lookup table, with one key $node_B$.STATE and value $node_B$
  solution ← failure
  while not TERMINATED(solution, $frontier_F$, $frontier_B$) do
    if $f_F$(TOP($frontier_F$)) < $f_B$(TOP($frontier_B$)) then
      solution ← PROCEED(F, $problem_F$, $frontier_F$, $reached_F$, $reached_B$, solution)
    else solution ← PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, $reached_2$, solution) returns a solution
        // Expand node on frontier; check against the other frontier in $reached_2$.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in $reached_2$ then
        $solution_2$ ← JOIN-NODES(dir, child, $reached_2$[s]))
        if PATH-COST($solution_2$) < PATH-COST(solution) then
          solution ← $solution_2$
  return solution
```

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

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

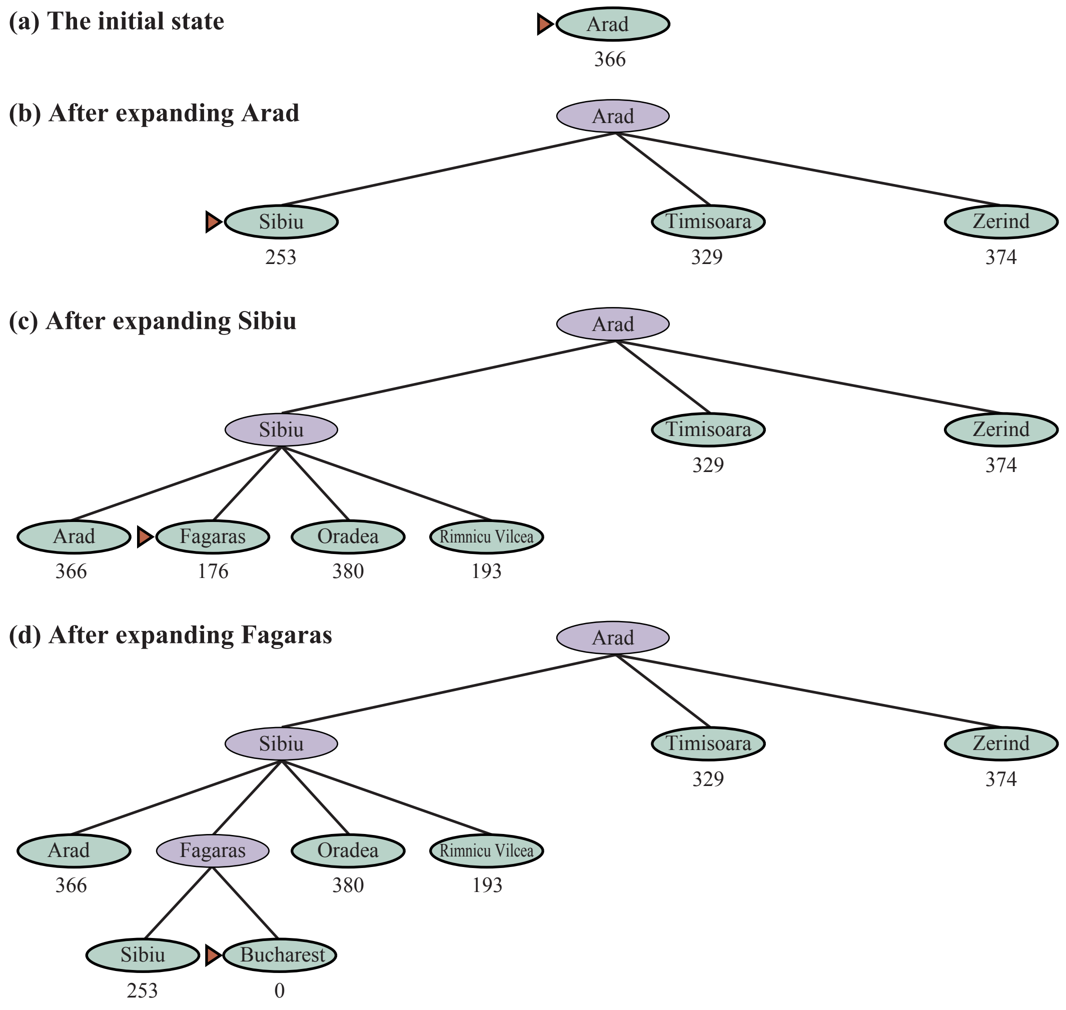


Figure 3.17 Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

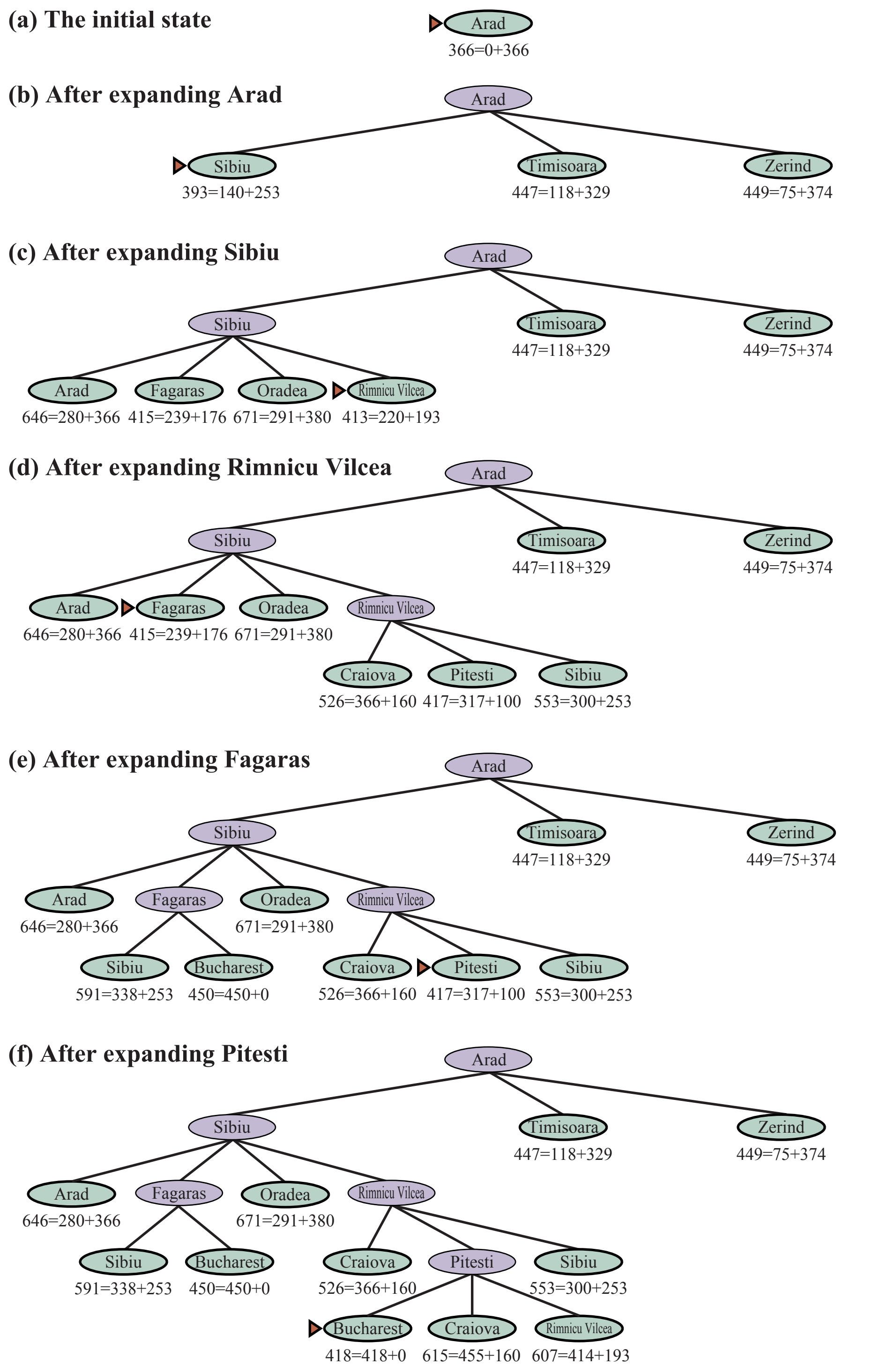


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

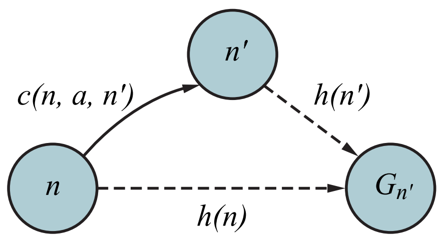


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n,a,a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

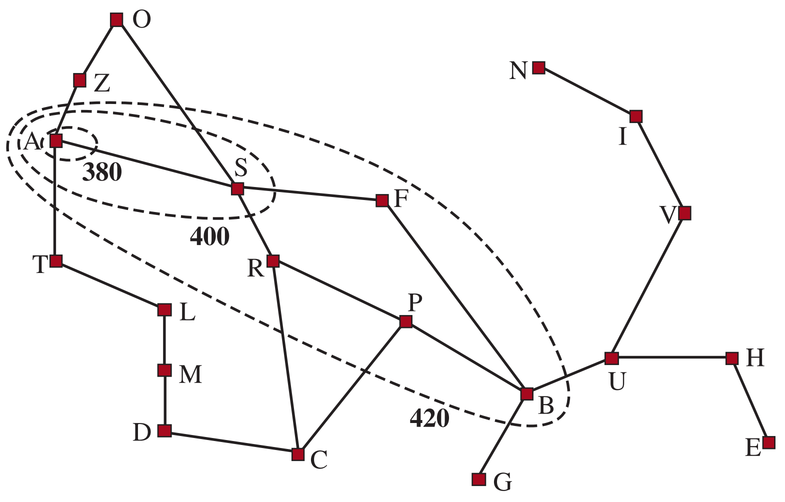


Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

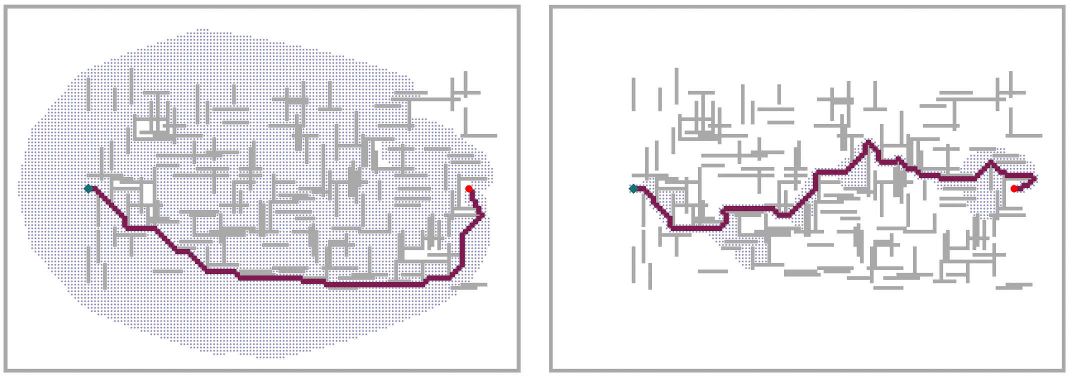


Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

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

function RBFS(problem, node, f_limit) returns a solution or failure, and a new f-cost limit
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors is empty then return failure, ∞
  for each s in successors do        // update f with value from previous search
      s.f ← max(s.PATH-COST + h(s), node.f))
  while true do
      best ← the node in successors with lowest f-value
      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, best.f
```

Figure 3.22 The algorithm for recursive best-first search.

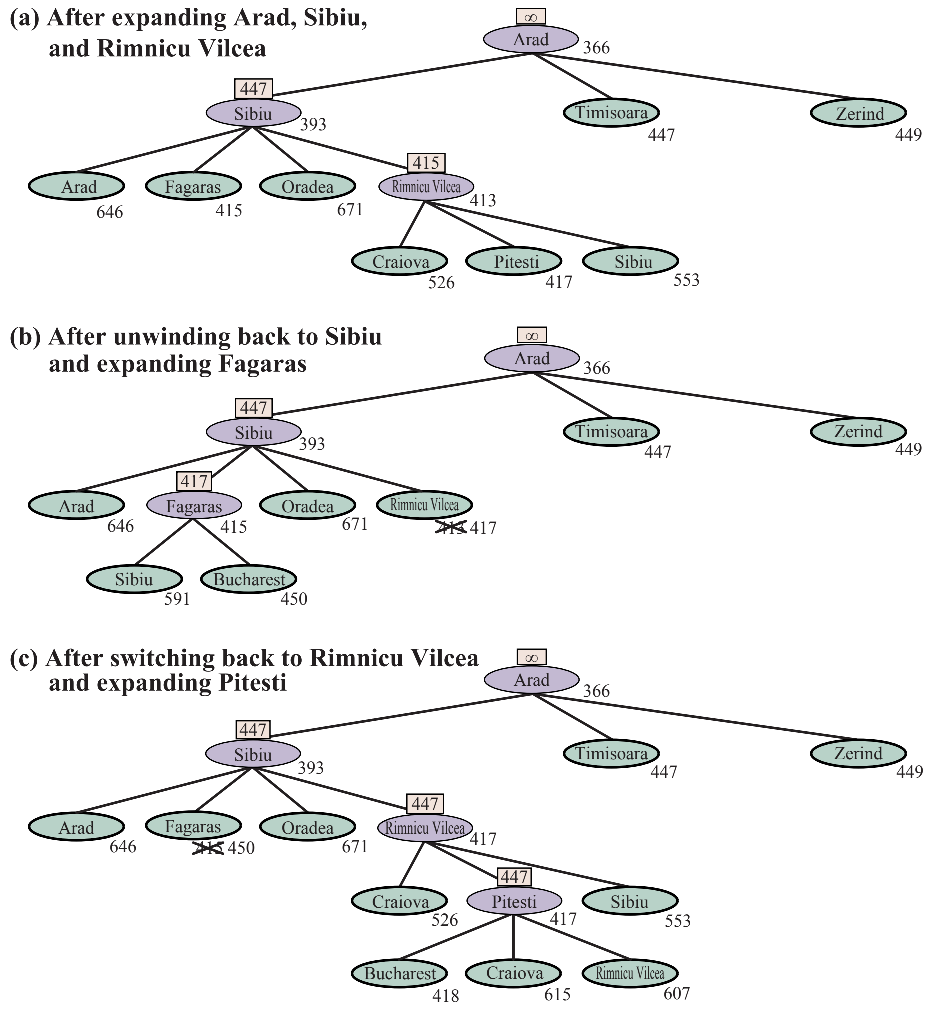


Figure 3.23 Stages in an RBFS search for the shortest route to Bucharest. The f-*limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is 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.

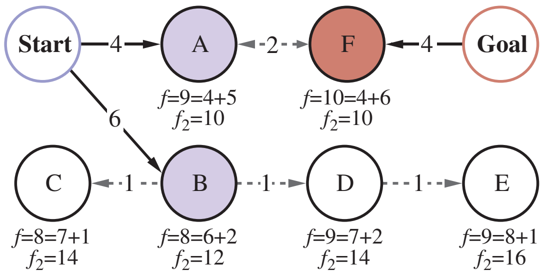


Figure 3.24 Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g+h)$ value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4+2+4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest f cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when f_2 is the evaluation function.

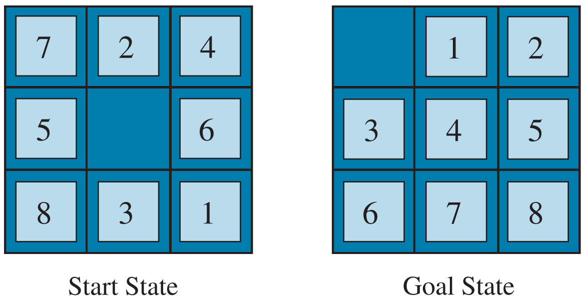


Figure 3.25 A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Figure 3.26 Comparison of the search costs and effective branching factors for 8-puzzle problems using breadth-first search, A^* with h_1 (misplaced tiles), and A^* with h_2 (Manhattan distance). Data are averaged over 100 puzzles for each solution length d from 6 to 28.

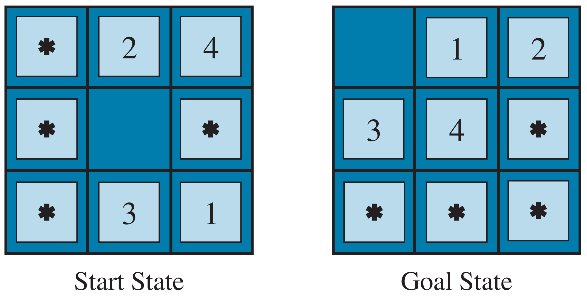
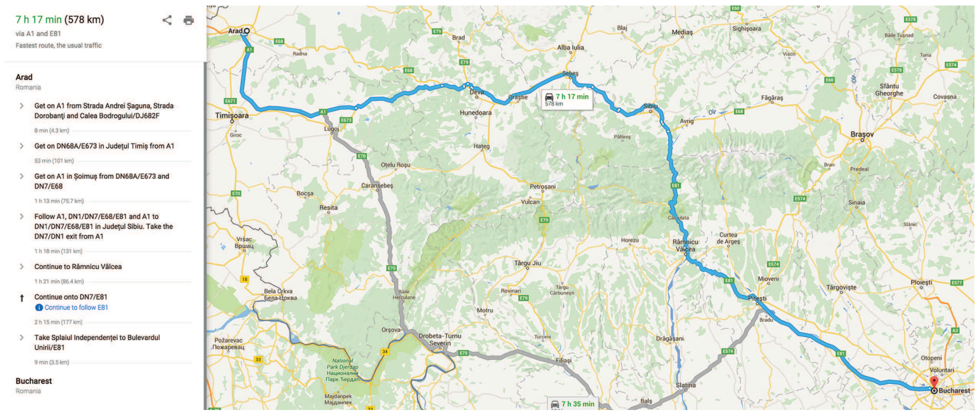


Figure 3.27 A subproblem of the 8-puzzle instance given in Figure 3.25. The task is to get tiles 1, 2, 3, 4, and the blank into their correct positions, without worrying about what happens to the other tiles.



7 h 17 min (578 km)
via A1 and E81
Fastest route, the usual traffic

Arad
Romania

› Get on A1 from Strada Andrei Şaguna, Strada Dorobanţi and Calea Bodrogului/DJ682F
8 min (4.3 km)

› Get on DN68A/E673 in Judeţul Timiş from A1
53 min (101 km)

› Get on A1 in Şoimuş from DN68A/E673 and DN7/E68
1 h 13 min (75.7 km)

› Follow A1, DN1/DN7/E68/E81 and A1 to DN1/DN7/E68/E81 in Judeţul Sibiu. Take the DN7/DN1 exit from A1
1 h 18 min (131 km)

› Continue to Râmnicu Vâlcea
1 h 21 min (86.4 km)

↑ Continue onto DN7/E81
Continue to follow E81
2 h 15 min (177 km)

› Take Splaiul Independenţei to Bulevardul Unirii/E81
9 min (3.5 km)

Bucharest
Romania

Figure 3.28 A Web service providing driving directions, computed by a search algorithm.

CHAPTER 4

SEARCH IN COMPLEX ENVIRONMENTS

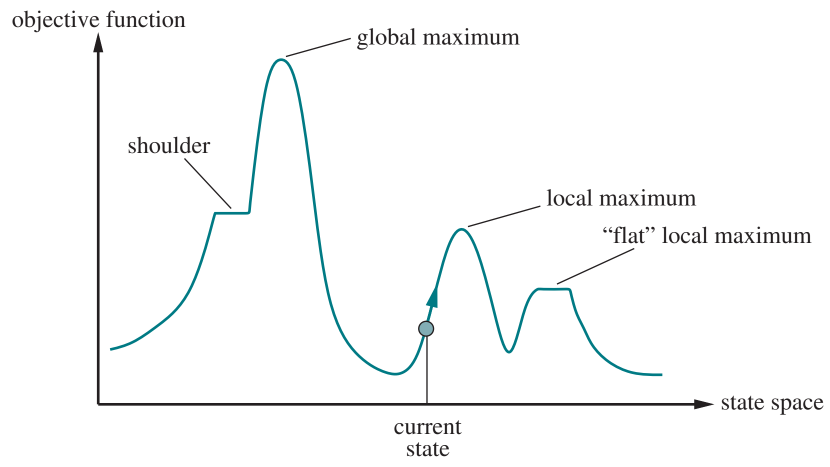


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
      neighbor ← a highest-valued successor state of current
      if VALUE(neighbor) ≤ VALUE(current) then return current
      current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

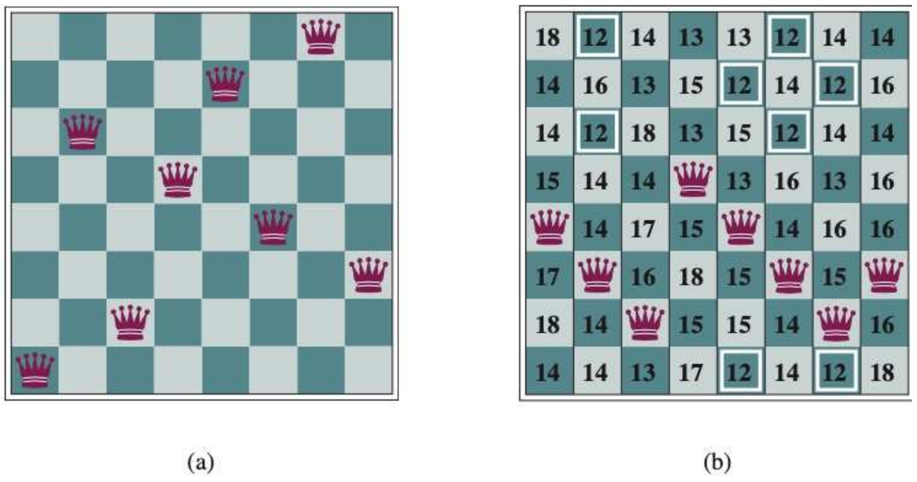


Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h = 17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h = 12$. The hill-climbing algorithm will pick one of these.

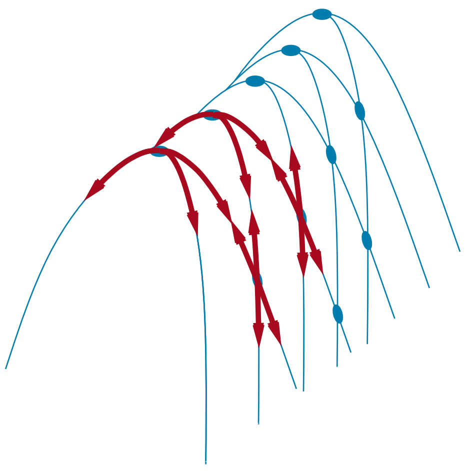


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill. Topologies like this are common in low-dimensional state spaces, such as points in a two-dimensional plane. But in state spaces with hundreds or thousands of dimensions, this intuitive picture does not hold, and there are usually at least a few dimensions that make it possible to escape from ridges and plateaus.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    current ← problem.INITIAL
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE(current) – VALUE(next)
        if ΔE > 0 then current ← next
        else current ← next only with probability e^{ΔE/T}
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the "temperature" T as a function of time.

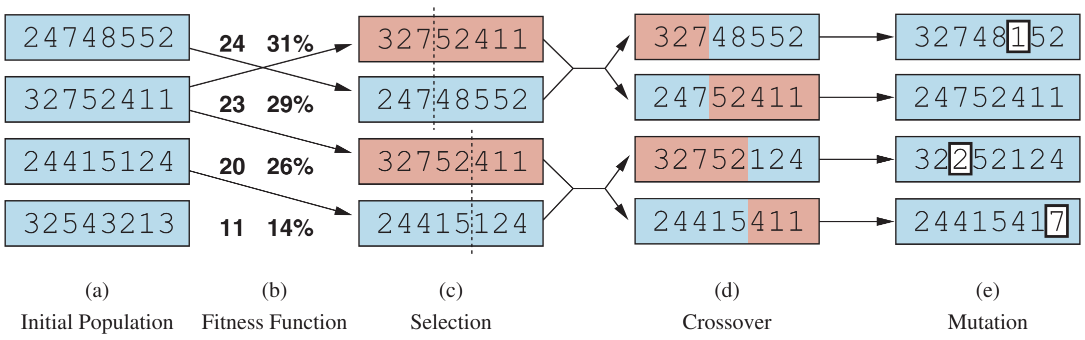


Figure 4.6 A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

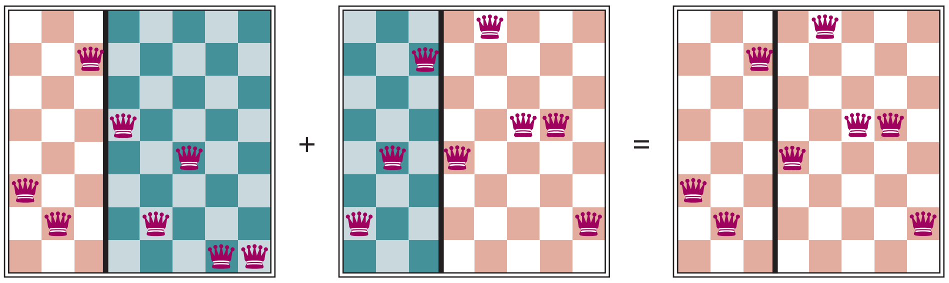


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.5(c) and the first offspring in Figure 4.5(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.5: row 1 is the bottom row, and 8 is the top row.)

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
     weights ← WEIGHTED-BY(population, fitness)
     population2 ← empty list
     for i = 1 to SIZE(population) do
        parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child ← REPRODUCE(parent1, parent2)
        if (small random probability) then child ← MUTATE(child)
        add child to population2
     population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.8 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

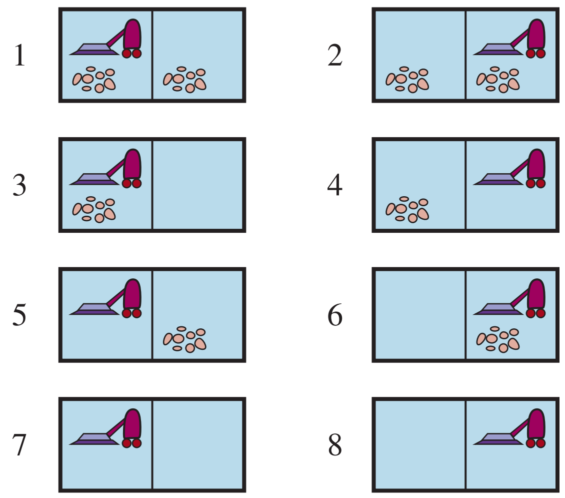


Figure 4.9 The eight possible states of the vacuum world; states 7 and 8 are goal states.

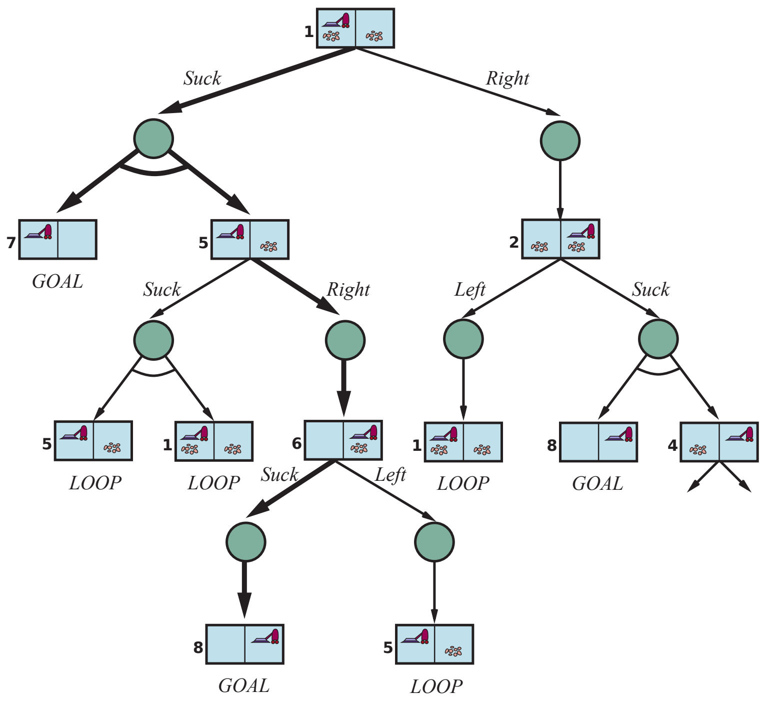


Figure 4.10 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines.

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

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

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
  for each $s_i$ in states do
     $plan_i$ ← OR-SEARCH(problem, $s_i$, path)
     if $plan_i$ = failure then return failure
  return [if $s_1$ then $plan_1$ 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. A solution is a conditional plan that considers every nondeterministic outcome and makes a plan for each one.

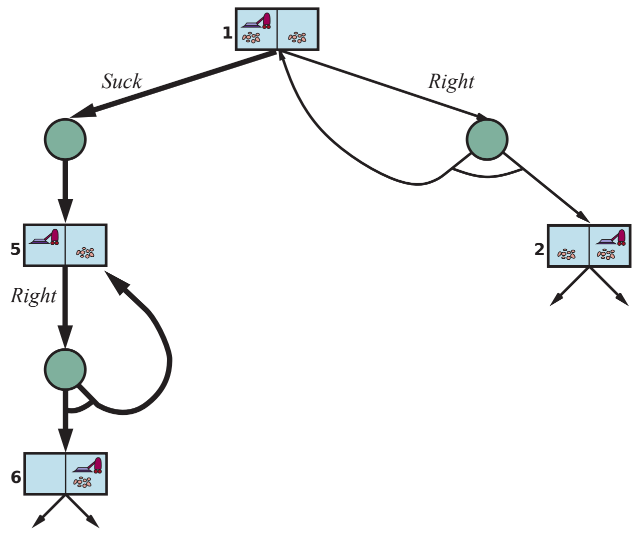


Figure 4.12 Part of the search graph for a slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably.

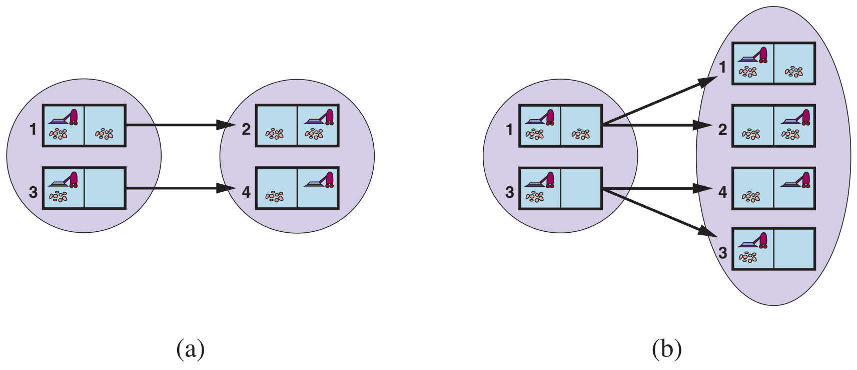


Figure 4.13 (a) Predicting the next belief state for the sensorless vacuum world with the deterministic action, *Right*. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

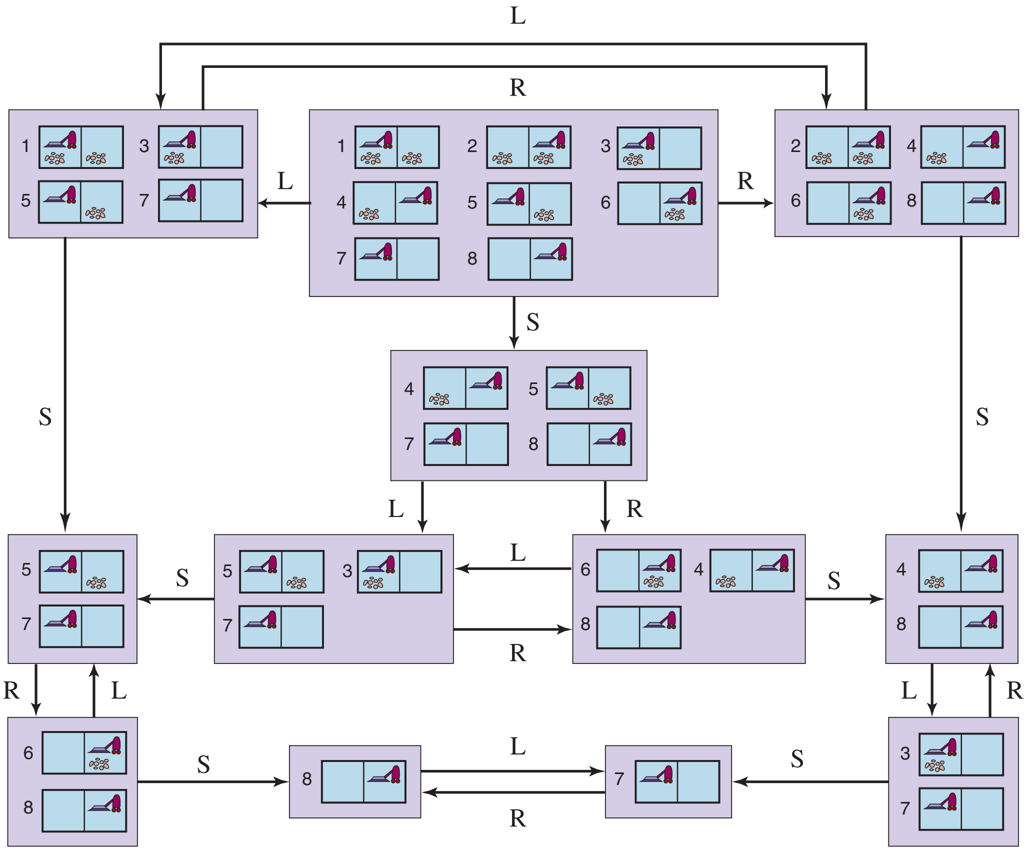


Figure 4.14 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each rectangular box corresponds to a single belief state. At any given point, the agent has a belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box.

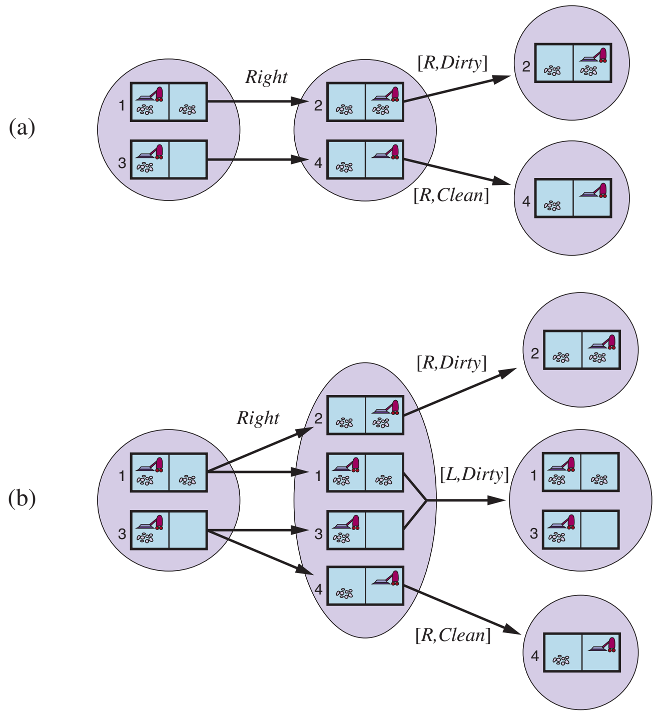


Figure 4.15 Two examples of transitions in local-sensing vacuum worlds. (a) In the deterministic world, *Right* is applied in the initial belief state, resulting in a new predicted belief state with two possible physical states; for those states, the possible percepts are $[R, Dirty]$ and $[R, Clean]$, leading to two belief states, each of which 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 $[L, Dirty]$, $[R, Dirty]$, and $[R, Clean]$, leading to three belief states as shown.

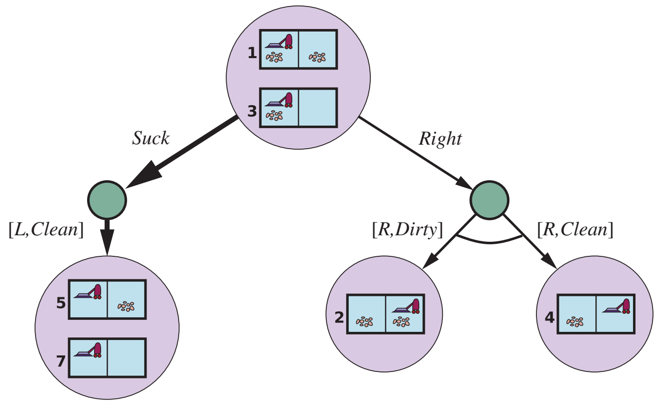


Figure 4.16 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; *Suck* is the first action in the solution.

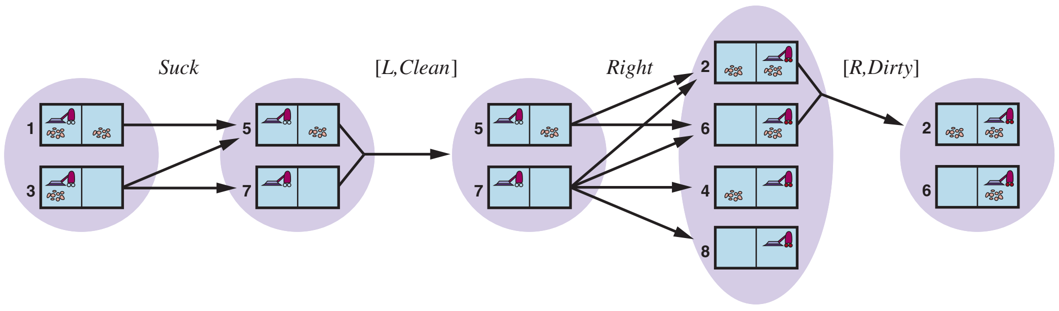
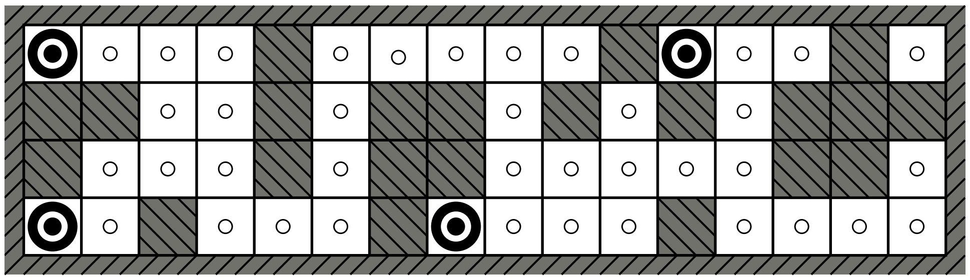
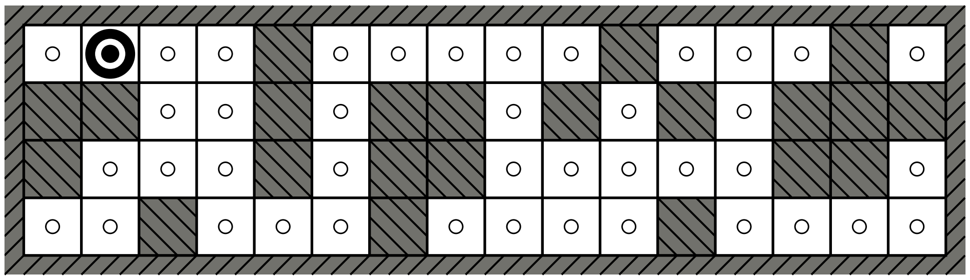


Figure 4.17 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.



(a) Possible locations of robot after $E_1 = 1011$



(b) Possible locations of robot after $E_1 = 1011$, $E_2 = 1010$

Figure 4.18 Possible positions of the robot, $\odot$, (a) after one observation, $E_1 = 1011$, and (b) after moving one square and making a second observation, $E_2 = 1010$. When sensors are noiseless and the transition model is accurate, there is only one possible location for the robot consistent with this sequence of two observations.

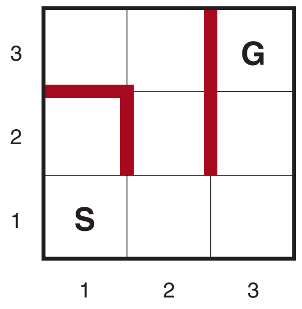


Figure 4.19 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

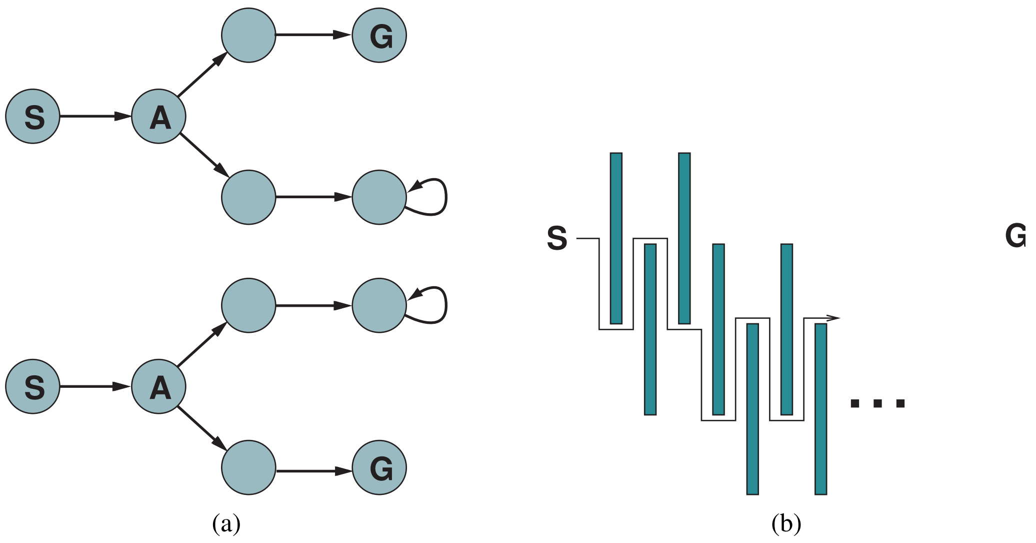


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent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

```
function ONLINE-DFS-AGENT(problem, s′) returns an action
                  s, a, the previous state and action, initially null
                  result, a table mapping (s, a) to s′, initially empty
                  untried, a table mapping s to a list of untried actions
                  unbacktracked, a table mapping s to a list of states never backtracked to

  if problem.IS-GOAL(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← problem.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
       a ← an action b such that result[s′, b] = POP(unbacktracked[s′])s′ ← null
  else a ← POP(untried[s′])
  s ← s′
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent can safely explore only in state spaces in which every action can be "undone" by some other action.

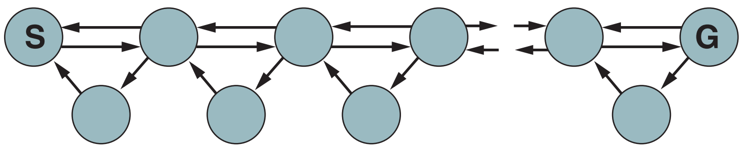


Figure 4.22 An environment in which a random walk will take exponentially many steps to find the goal.

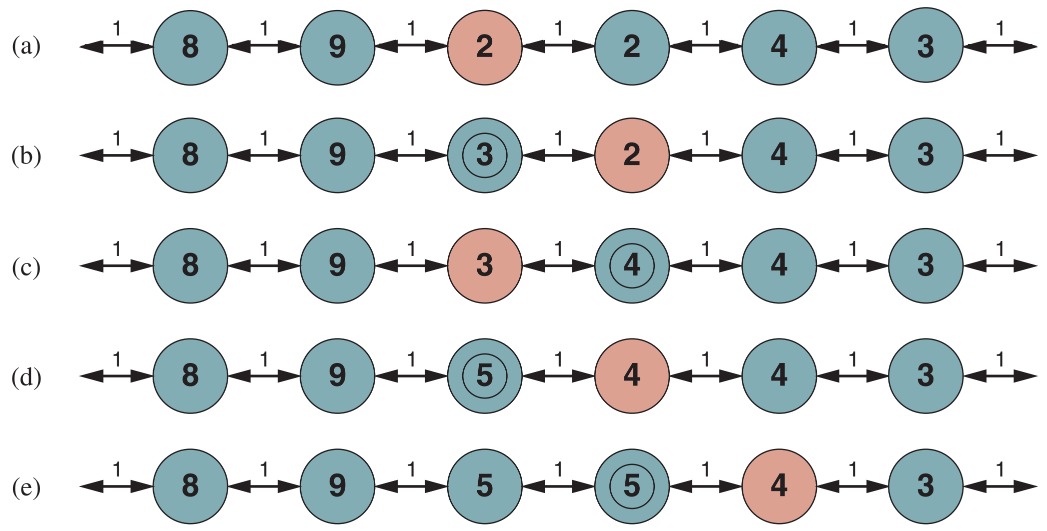


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and every link has an action cost of 1. The red state marks the location of the agent, and the updated cost estimates at each iteration have a double circle.

```
function LRTA*-AGENT(problem, s', h) returns an action
          s, a, the previous state and action, initially null
          result, a table mapping (s, a) to s', initially empty
          H, a table mapping s to a cost estimate, initially empty

  if IS-GOAL(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null then
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(problem, s, b, result[s, b], H)
  a ← argmin_{b ∈ ACTIONS(s)} LRTA*-COST(problem, s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(problem, s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return problem.ACTION-COST(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.

CHAPTER 5

CONSTRAINT SATISFACTION PROBLEMS

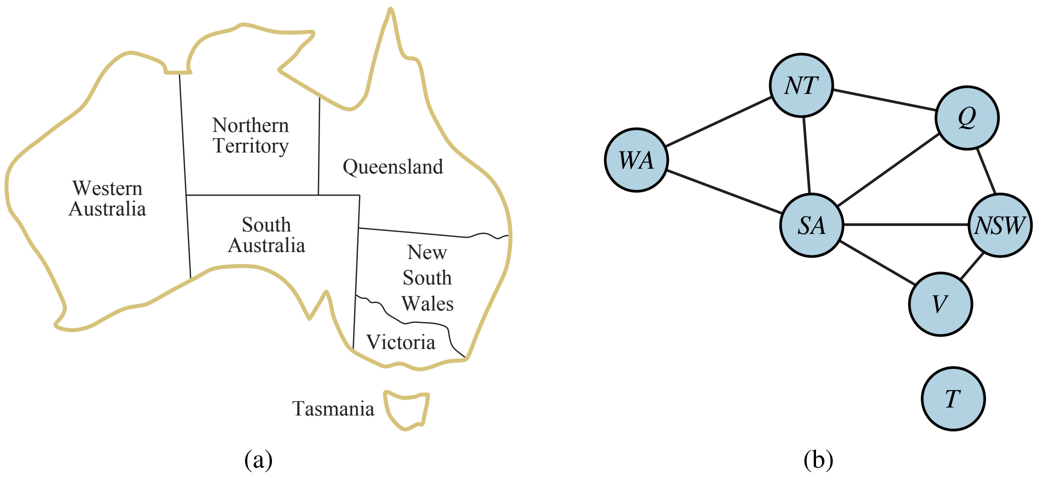
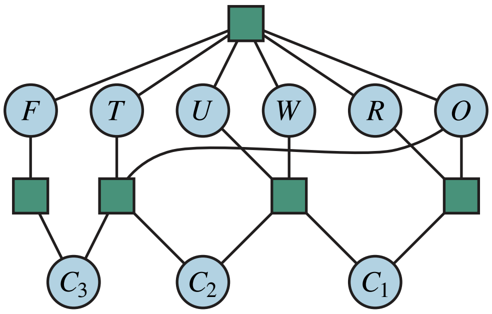


Figure 5.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$

(a)



(b)

Figure 5.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns from right to left.

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    ($X_i$, $X_j$) ← POP(queue)
    if REVISE(csp, $X_i$, $X_j$) then
      if size of $D_i$ = 0 then return false
      for each $X_k$ in $X_i$.NEIGHBORS - {$X_j$} do
        add ($X_k$, $X_i$) to queue
  return true

function REVISE(csp, $X_i$, $X_j$) returns true iff we revise the domain of $X_i$
  revised ← false
  for each x in $D_i$ do
    if no value y in $D_j$ allows (x,y) to satisfy the constraint between $X_i$ and $X_j$ then
      delete x from $D_i$
      revised ← true
  return revised
```

Figure 5.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it was the third version developed in the paper.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 5.4 (a) A Sudoku puzzle and (b) its solution.

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
       add {var = value} to assignment
       inferences ← INFERENCE(csp, var, assignment)
       if inferences ≠ failure then
         add inferences to csp
         result ← BACKTRACK(csp, assignment)
         if result ≠ failure then return result
         remove inferences from csp
       remove {var = value} from assignment
  return failure
```

Figure 5.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES implement the general-purpose heuristics discussed in Section 5.3.1. The INFERENCE function can optionally impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are retracted and a new value is tried.

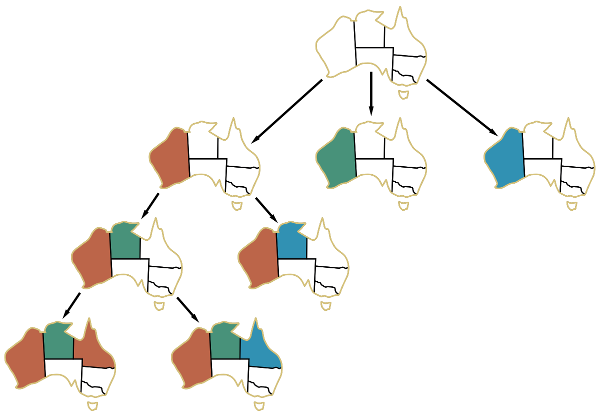


Figure 5.6 Part of the search tree for the map-coloring problem in Figure 5.1.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
After *WA*=*red*	🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
After *Q*=*green*	🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
After *V*=*blue*	🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Figure 5.7 The progress of a map-coloring search with forward checking. *WA* = *red* is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After *Q* = *green* is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After *V* = *blue* is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

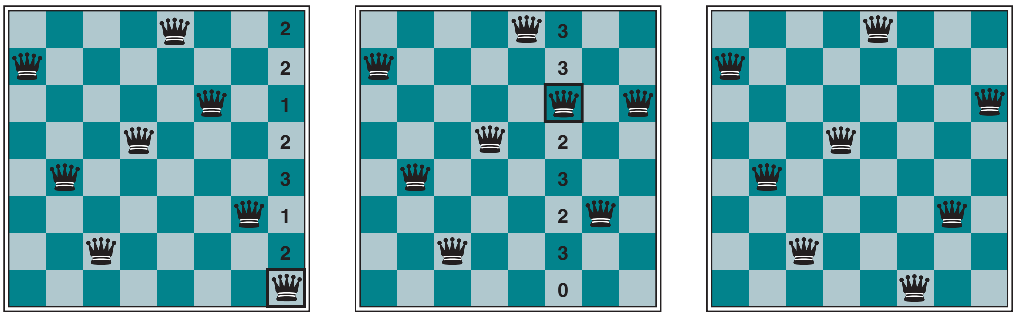


Figure 5.8 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(csp, var, v, current)
     set var = value in current
  return failure
```

Figure 5.9 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

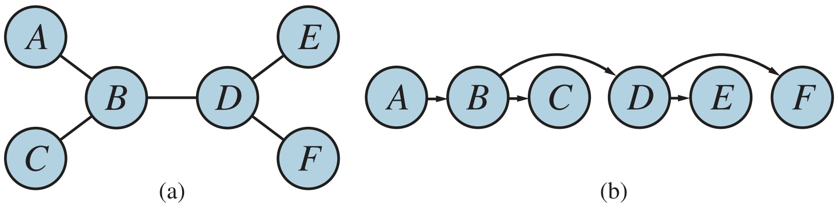


Figure 5.10 (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with A as the root. This is known as a **topological sort** of the variables.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
   inputs: csp, a CSP with components $X$, $D$, $C$

   $n$ ← number of variables in $X$
   assignment ← an empty assignment
   root ← any variable in $X$
   $X$ ← TOPOLOGICALSORT($X$, root)
   for $j = n$ down to 2 do
      MAKE-ARC-CONSISTENT(PARENT($X_j$), $X_j$)
      if it cannot be made consistent then return failure
   for $i = 1$ to $n$ do
      assignment[$X_i$] ← any consistent value from $D_i$
      if there is no consistent value then return failure
   return assignment
```

Figure 5.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

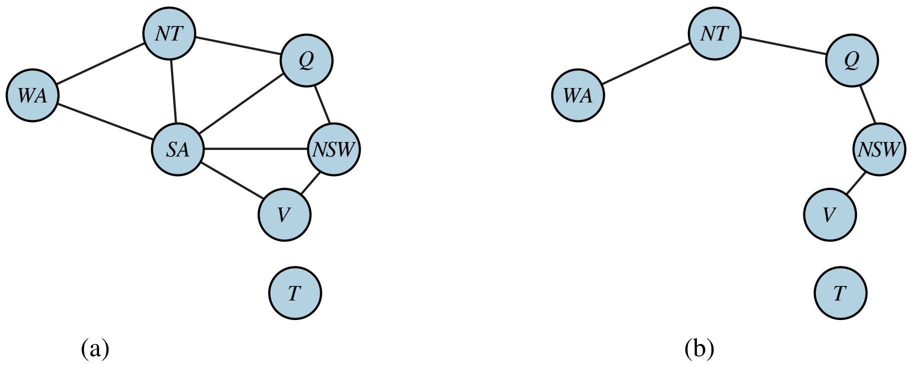


Figure 5.12 (a) The original constraint graph from Figure 5.1. (b) After the removal of *SA*, the constraint graph becomes a forest of two trees.

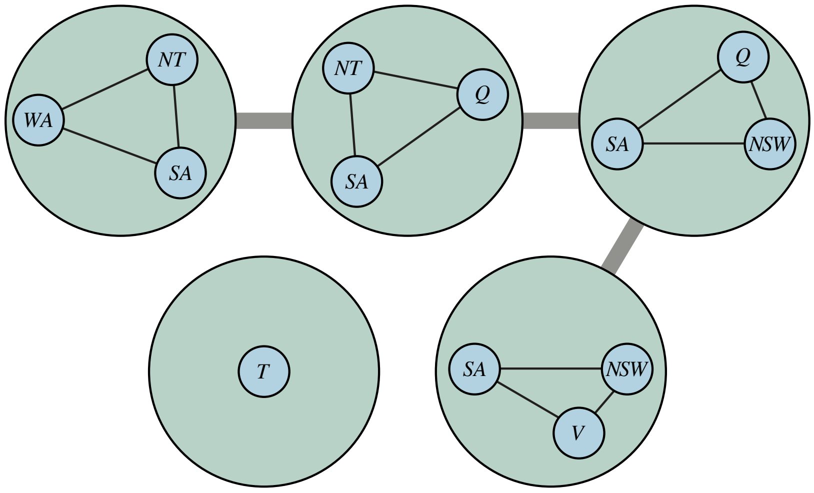


Figure 5.13 A tree decomposition of the constraint graph in Figure 5.12(a).

CHAPTER 6

ADVERSARIAL SEARCH AND GAMES

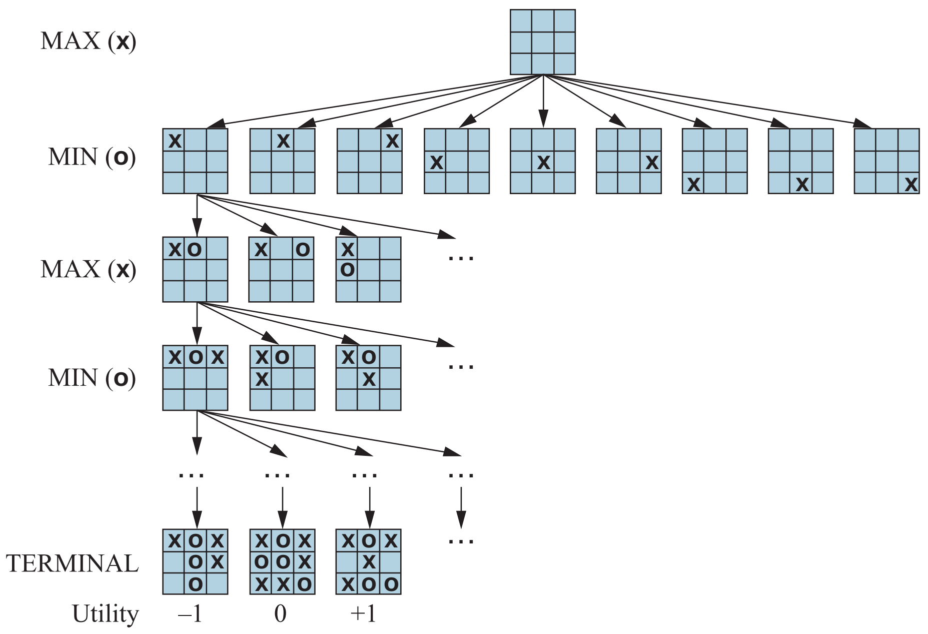


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

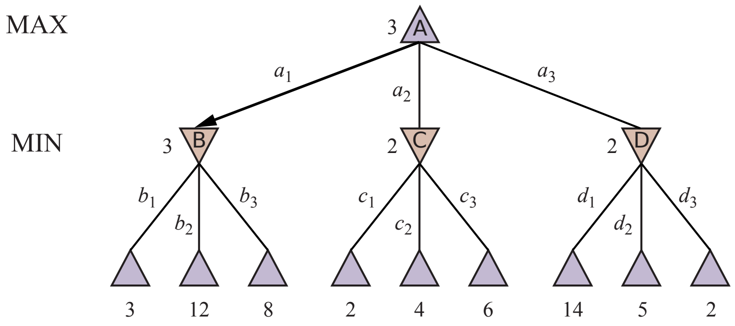


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 6.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

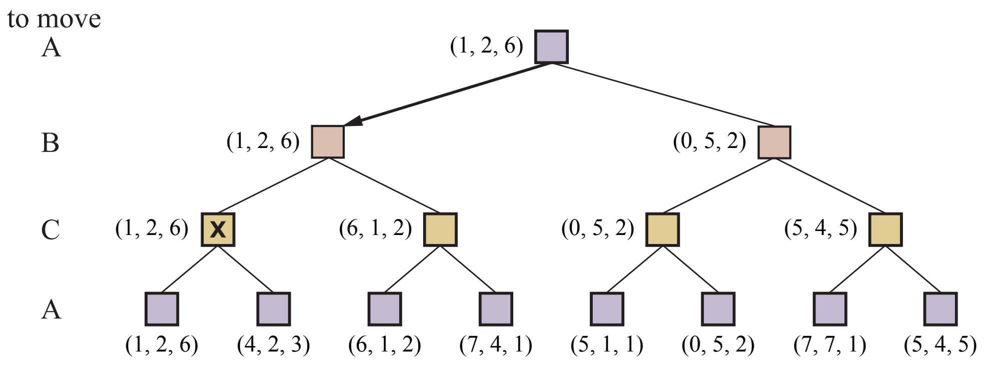


Figure 6.4 The first three ply of a game tree with three players (*A*, *B*, *C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

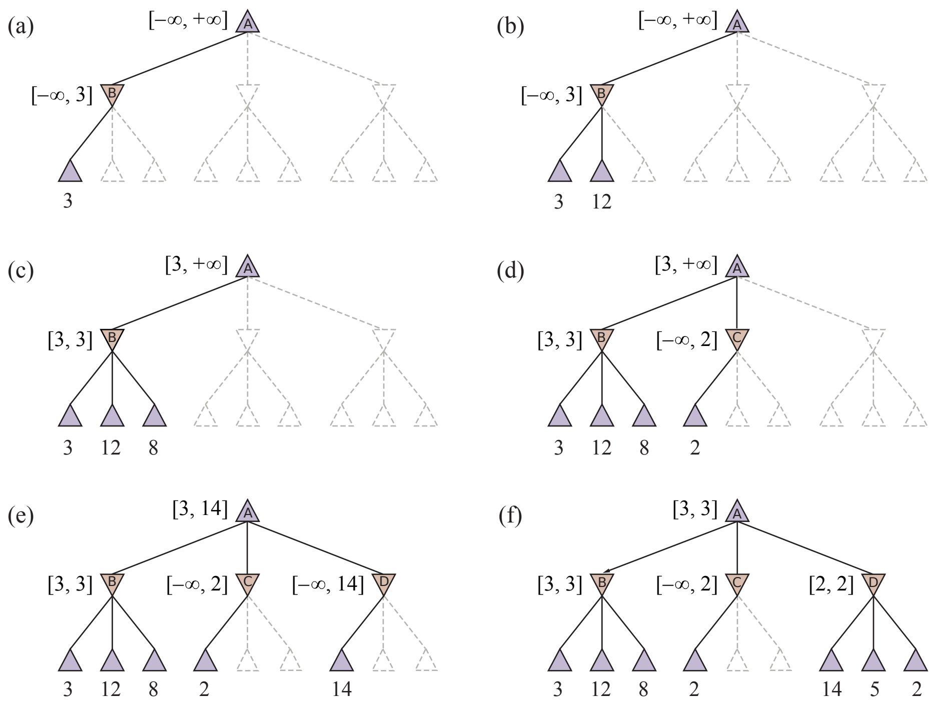


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below *B* has the value 3. Hence, *B*, which is a MIN node, has a value of *at most* 3. (b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3. (c) The third leaf below *B* has a value of 8; we have seen all *B*'s successor states, so the value of *B* is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below *C* has the value 2. Hence, *C*, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose *C*. Therefore, there is no point in looking at the other successor states of *C*. This is an example of alpha–beta pruning. (e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring *D*'s successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of *D* is worth 5, so again we need to keep exploring. The third successor is worth 2, so now *D* is worth exactly 2. MAX's decision at the root is to move to *B*, giving a value of 3.

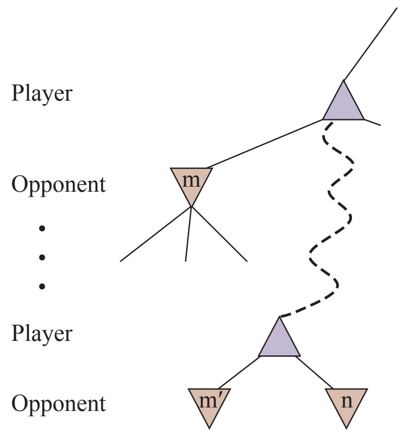
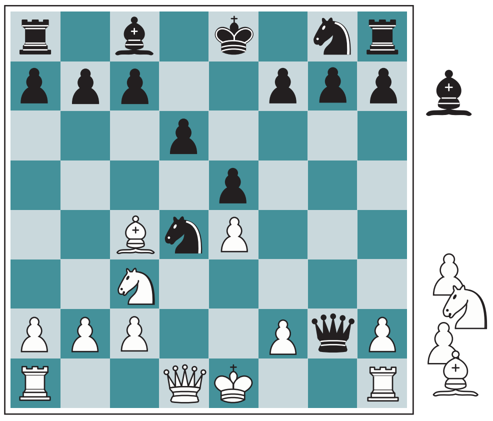


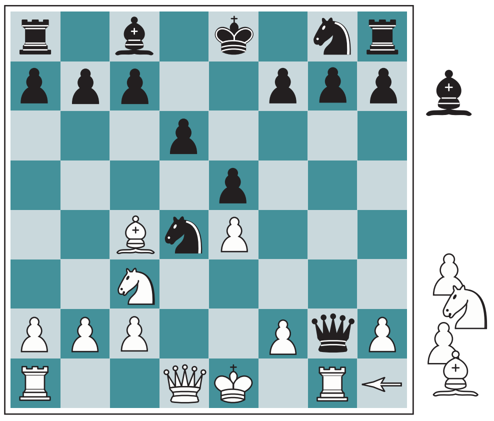
Figure 6.6 The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, $-\infty$, $+\infty$)
  return move

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow -\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 > v$ then
      v, move ← v2, a
      $\alpha \leftarrow$ MAX($\alpha$, $v$)
    if $v \geq \beta$ then return v, move
  return v, move

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  $v \leftarrow +\infty$
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), $\alpha$, $\beta$)
    if $v2 < v$ then
      v, move ← v2, a
      $\beta \leftarrow$ MIN($\beta$, $v$)
    if $v \leq \alpha$ then return v, move
  return v, move
```

Figure 6.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 6.3, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.



(a) White to move



(b) White to move

Figure 6.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

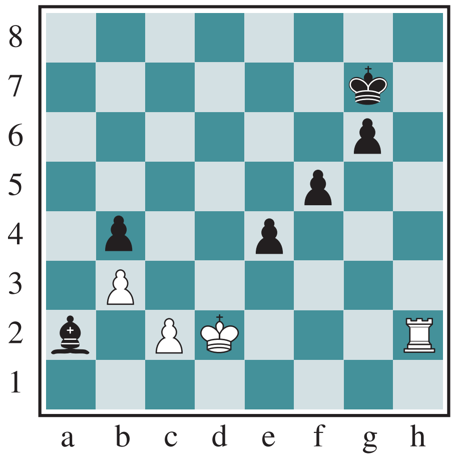


Figure 6.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, encouraging the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

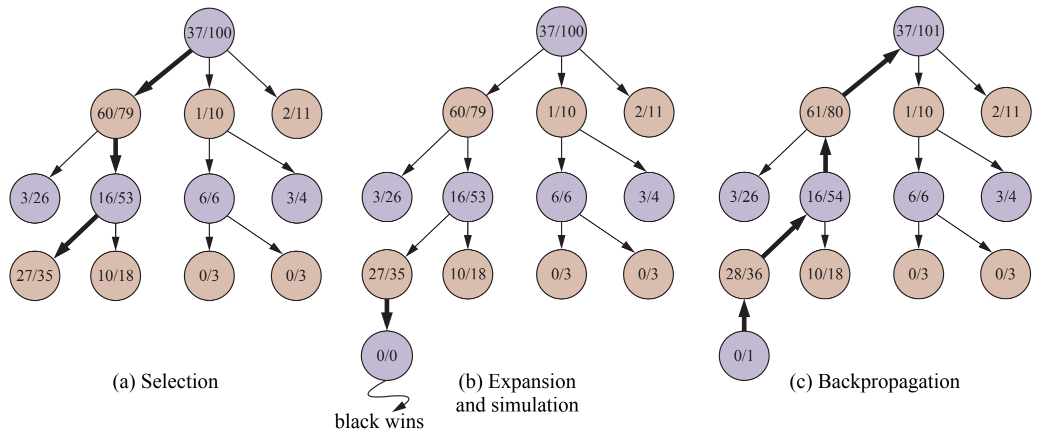


Figure 6.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 6.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

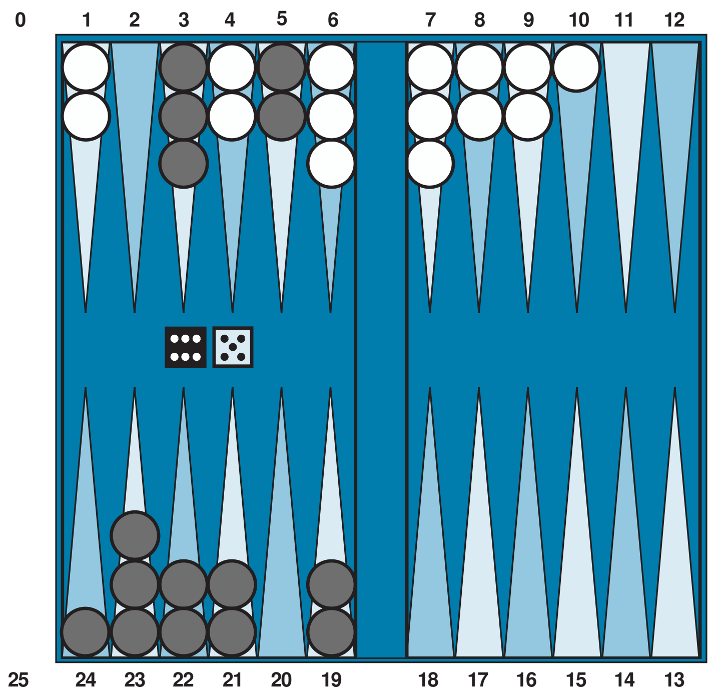


Figure 6.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

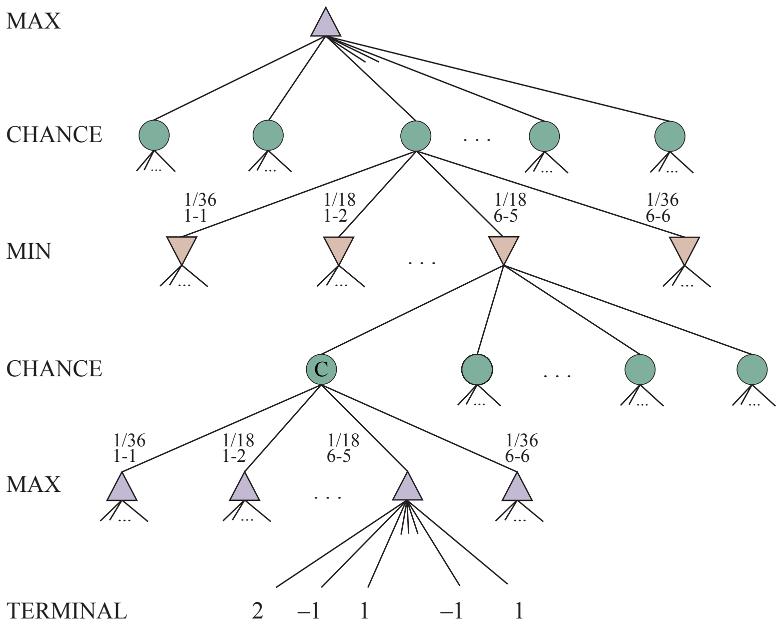


Figure 6.13 Schematic game tree for a backgammon position.

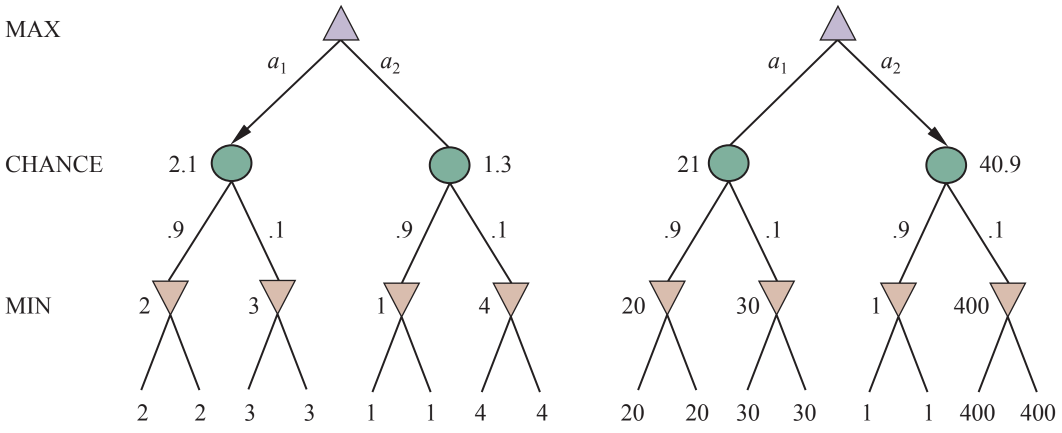


Figure 6.14 An order-preserving transformation on leaf values changes the best move.

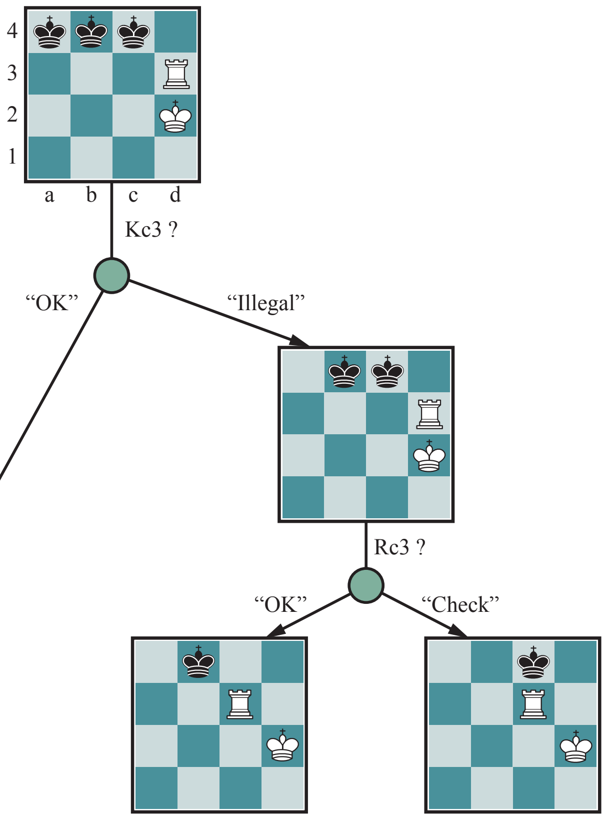


Figure 6.15 Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.

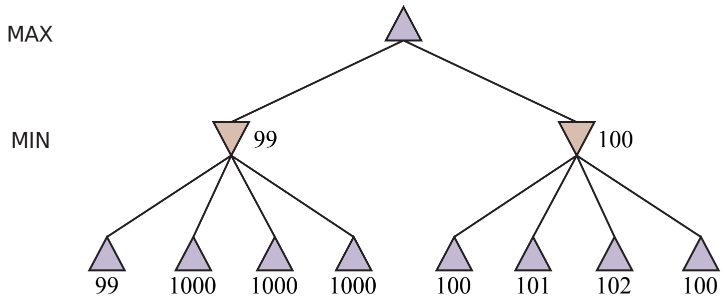


Figure 6.16 A two-ply game tree for which heuristic minimax may make an error.

CHAPTER 7

LOGICAL AGENTS

```
function KB-AGENT(percept) returns an action
   persistent: KB, a knowledge base
               t, a counter, initially 0, indicating time

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   action ← ASK(KB, MAKE-ACTION-QUERY(t))
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

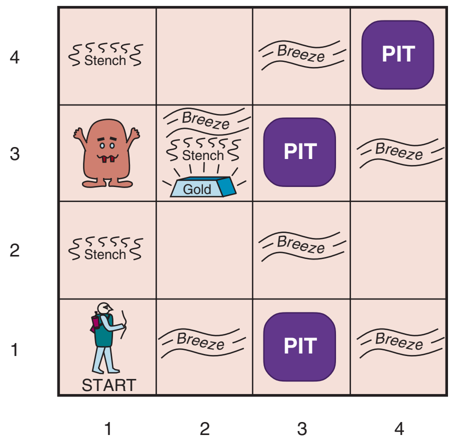


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing east (rightward).

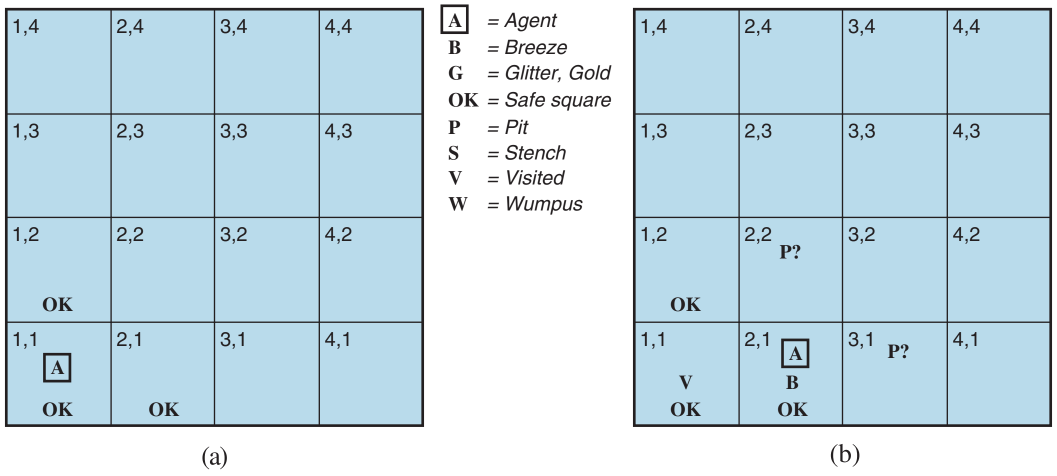


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After moving to [2,1] and perceiving [*None*, *Breeze*, *None*, *None*, *None*].

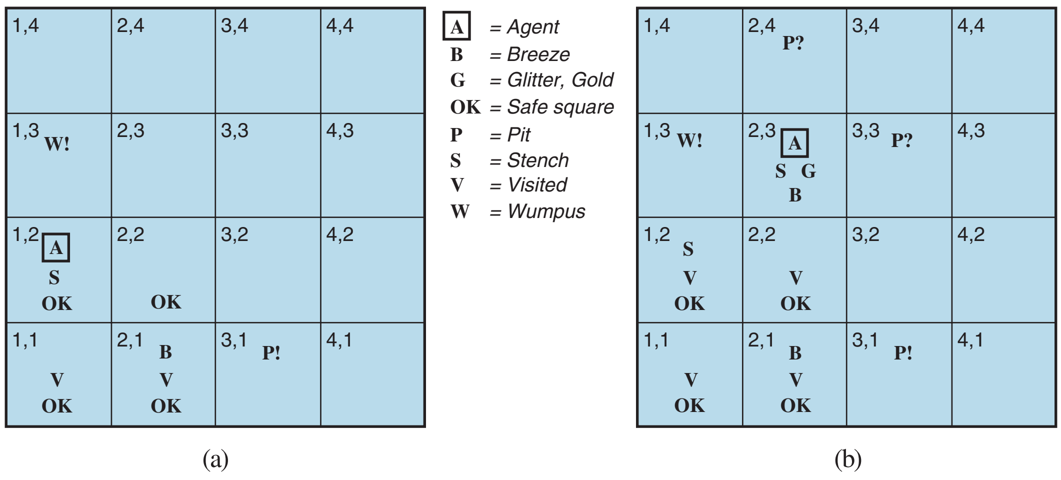


Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*, *None*, *None*, *None*, *None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

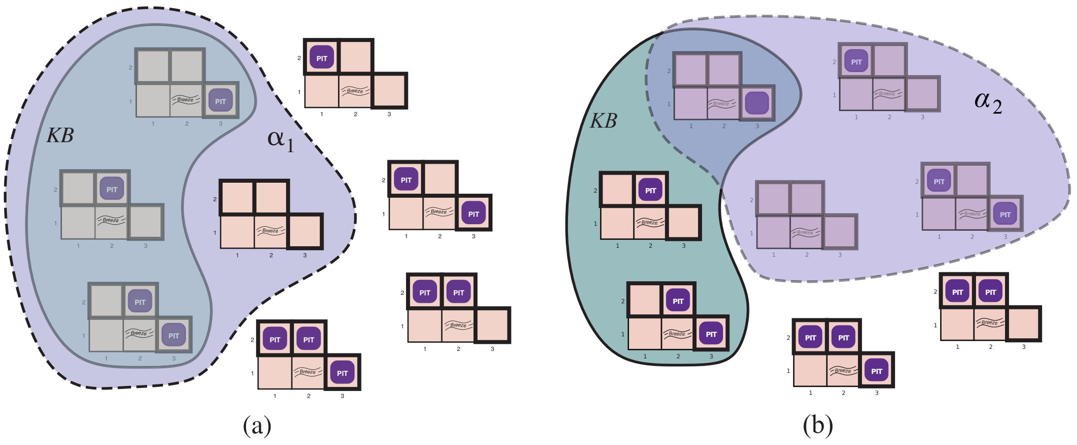


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

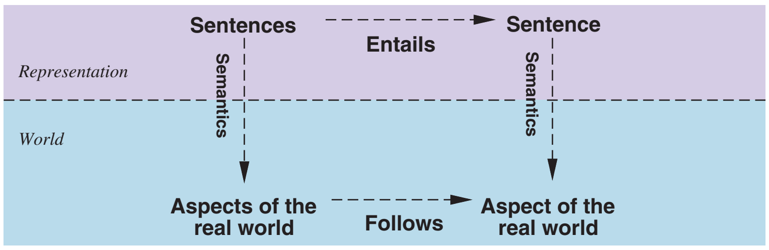


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \\
 & \mid & \neg\,\mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

P	Q	$\neg P$	$P \land Q$	$P \lor Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \lor Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \lor Q$ column to see the result: *true*.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	$\underline{\textit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	$\underline{\textit{true}}$
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	$\underline{\textit{true}}$
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```
function TT-ENTAILS?($KB, \alpha$) returns true or false
   inputs: $KB$, the knowledge base, a sentence in propositional logic
           $\alpha$, the query, a sentence in propositional logic

   symbols ← a list of the proposition symbols in $KB$ and $\alpha$
   return TT-CHECK-ALL($KB, \alpha, symbols, \{\}$)

function TT-CHECK-ALL($KB, \alpha, symbols, model$) returns true or false
   if EMPTY?(symbols) then
      if PL-TRUE?($KB, model$) then return PL-TRUE?($\alpha, model$)
      else return true        // when KB is false, always return true
   else
      $P$ ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = true\}$)
              and
              TT-CHECK-ALL($KB, \alpha, rest, model \cup \{P = false\}$))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword **and** here is an infix function symbol in the pseudocode programming language, not an operator in propositional logic; it takes two arguments and returns *true* or *false*.

$$
\begin{array}{rcll}
(\alpha \wedge \beta) & \equiv & (\beta \wedge \alpha) & \text{commutativity of } \wedge \\
(\alpha \vee \beta) & \equiv & (\beta \vee \alpha) & \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) & \equiv & (\alpha \wedge (\beta \wedge \gamma)) & \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) & \equiv & (\alpha \vee (\beta \vee \gamma)) & \text{associativity of } \vee \\
\neg(\neg \alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg \beta \Rightarrow \neg \alpha) & \text{contraposition} \\
(\alpha \Rightarrow \beta) & \equiv & (\neg \alpha \vee \beta) & \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) & \equiv & ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) & \equiv & (\neg \alpha \vee \neg \beta) & \text{De Morgan} \\
\neg(\alpha \vee \beta) & \equiv & (\neg \alpha \wedge \neg \beta) & \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) & \equiv & ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) & \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) & \equiv & ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) & \text{distributivity of } \vee \text{ over } \wedge
\end{array}
$$

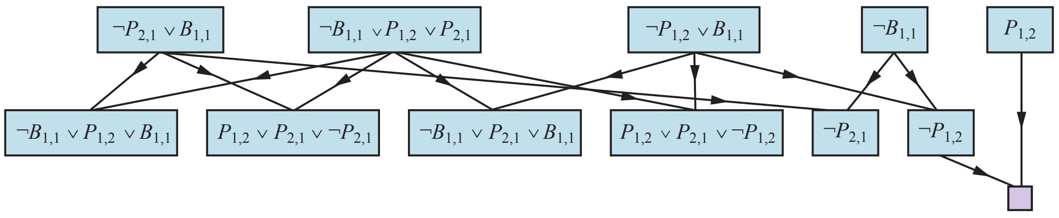
Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

$$
\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Fact} & \rightarrow & \mathit{Symbol} \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & \mathit{Fact} \mid (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}
$$

Figure 7.12 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A CNF clause such as $\neg A \vee \neg B \vee C$ can be written in definite clause form as $A \wedge B \Rightarrow C$.

```
function PL-RESOLUTION($KB$, $\alpha$) returns $true$ or $false$
  inputs: $KB$, the knowledge base, a sentence in propositional logic
          $\alpha$, the query, a sentence in propositional logic

  $clauses \leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  $new \leftarrow \{\}$
  while $true$ do
     for each pair of clauses $C_i$, $C_j$ in $clauses$ do
        $resolvents \leftarrow$ PL-RESOLVE($C_i$, $C_j$)
        if $resolvents$ contains the empty clause then return $true$
        $new \leftarrow new \cup resolvents$
     if $new \subseteq clauses$ then return $false$
     $clauses \leftarrow clauses \cup new$
```

Figure 7.13 A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.



$\neg P_{2,1} \vee B_{1,1}$
$\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$
$\neg P_{1,2} \vee B_{1,1}$
$\neg B_{1,1}$
$P_{1,2}$
$\neg B_{1,1} \vee P_{1,2} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{2,1}$
$\neg B_{1,1} \vee P_{2,1} \vee B_{1,1}$
$P_{1,2} \vee P_{2,1} \vee \neg P_{1,2}$
$\neg P_{2,1}$
$\neg P_{1,2}$

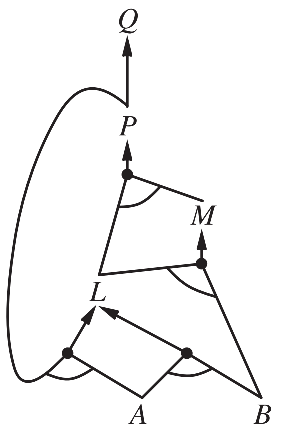
Figure 7.14 Partial application of PL-RESOLUTION to a simple inference in the wumpus world to prove the query $\neg P_{1,2}$. Each of the leftmost four clauses in the top row is paired with each of the other three, and the resolution rule is applied to yield the clauses on the bottom row. We see that the third and fourth clauses on the top row combine to yield the clause $\neg P_{1,2}$, which is then resolved with $P_{1,2}$ to yield the empty clause, meaning that the query is proven.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
     p ← POP(queue)
     if p = q then return true
     if inferred[p] = false then
        inferred[p] ← true
        for each clause c in KB where p is in c.PREMISE do
           decrement count[c]
           if count[c] = 0 then add c.CONCLUSION to queue
  return false
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *queue* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are not yet proven. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

$$P \Rightarrow Q$$
$$L \land M \Rightarrow P$$
$$B \land L \Rightarrow M$$
$$A \land P \Rightarrow L$$
$$A \land B \Rightarrow L$$
$$A$$
$$B$$

(a)



(b)

Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

```
function DPLL-SATISFIABLE?($s$) returns true or false
  inputs: $s$, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $s$
  symbols ← a list of the proposition symbols in $s$
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  $P$, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if $P$ is non-null then return DPLL(clauses, symbols – $P$, model ∪ {$P$=value})
  $P$, value ← FIND-UNIT-CLAUSE(clauses, model)
  if $P$ is non-null then return DPLL(clauses, symbols – $P$, model ∪ {$P$=value})
  $P$ ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {$P$=true}) or
         DPLL(clauses, rest, model ∪ {$P$=false})
```

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

```
function WALKSAT(clauses, $p$, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          $p$, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of value flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for each $i$ = 1 to max_flips do
     if model satisfies clauses then return model
     clause ← a randomly selected clause from clauses that is false in model
     if RANDOM(0, 1) $\leq p$ then
        flip the value in model of a randomly selected symbol from clause
     else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

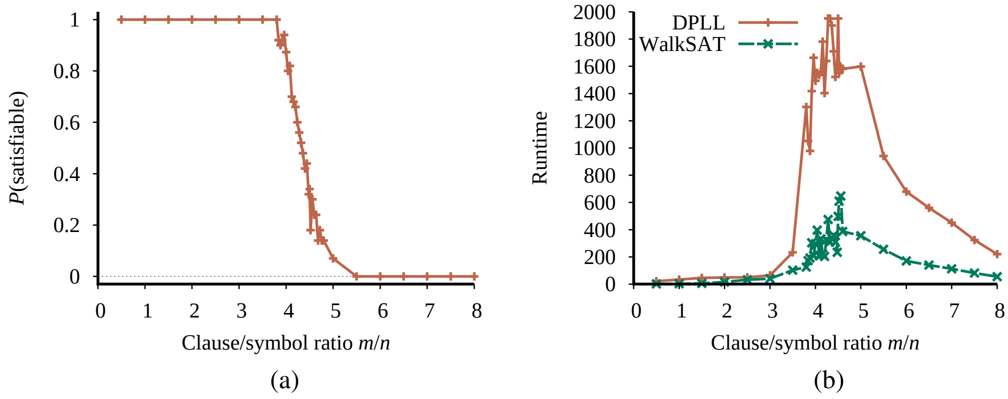


Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time (measured in number of iterations) for both DPLL and WALKSAT on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

```
function HYBRID-WUMPUS-AGENT(percept) returns an action
   inputs: percept, a list, [stench,breeze,glitter,bump,scream]
   persistent: KB, a knowledge base, initially the atemporal “wumpus physics”
               t, a counter, initially 0, indicating time
               plan, an action sequence, initially empty

   TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
   TELL the KB the temporal “physics” sentences for time t
   safe ← {[x,y] : ASK(KB, $OK^t_{x,y}$) = true}
   if ASK(KB, $Glitter^t$) = true then
      plan ← [Grab] + PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
   if plan is empty then
      unvisited ← {[x,y] : ASK(KB, $L^{t'}_{x,y}$) = false for all $t' \leq t$}
      plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
   if plan is empty and ASK(KB, $HaveArrow^t$) = true then
      possible_wumpus ← {[x,y] : ASK(KB, $\neg W_{x,y}$) = false}
      plan ← PLAN-SHOT(current, possible_wumpus, safe)
   if plan is empty then          // no choice but to take a risk
      not_unsafe ← {[x,y] : ASK(KB, $\neg OK^t_{x,y}$) = false}
      plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
   if plan is empty then
      plan ← PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
   action ← POP(plan)
   TELL(KB, MAKE-ACTION-SENTENCE(action, t))
   t ← t + 1
   return action

function PLAN-ROUTE(current,goals,allowed) returns an action sequence
   inputs: current, the agent’s current position
           goals, a set of squares; try to plan a route to one of them
           allowed, a set of squares that can form part of the route

   problem ← ROUTE-PROBLEM(current, goals,allowed)
   return SEARCH(problem)        // Any search algorithm from Chapter 3
```

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to choose actions. Each time HYBRID-WUMPUS-AGENT is called, it adds the percept to the knowledge base, and then either relies on a previously-defined plan or creates a new plan, and pops off the first step of the plan as the action to do next.

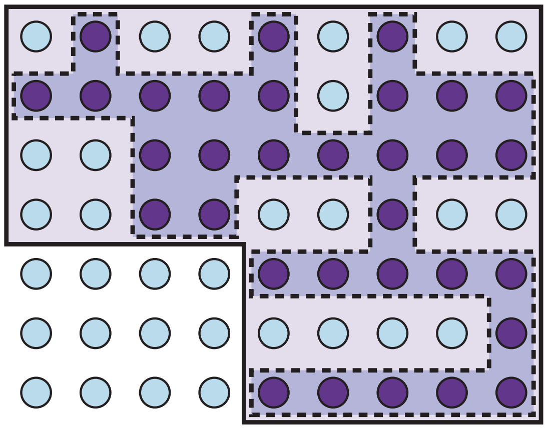


Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

```
function SATPLAN(init, transition, goal, $T_{max}$) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          $T_{max}$, an upper limit for plan length

  for $t = 0$ to $T_{max}$ do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

CHAPTER 8

FIRST-ORDER LOGIC

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

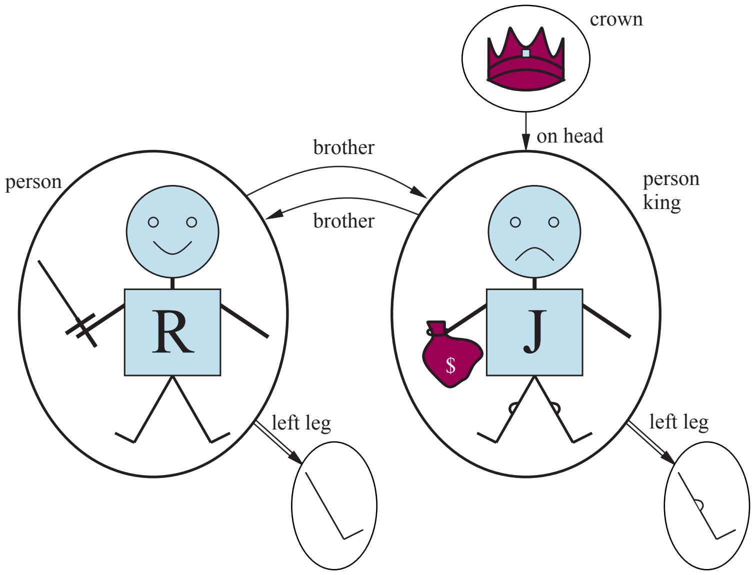


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

$$\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term},\ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \\
& \mid & \neg\ \mathit{Sentence} \\
& \mid & \mathit{Sentence} \land \mathit{Sentence} \\
& \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
& \mid & \mathit{Sentence}\ \Rightarrow\ \mathit{Sentence} \\
& \mid & \mathit{Sentence}\ \Leftrightarrow\ \mathit{Sentence} \\
& \mid & \mathit{Quantifier}\ \mathit{Variable},\ldots\ \mathit{Sentence} \\
\\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term},\ldots) \\
& \mid & \mathit{Constant} \\
& \mid & \mathit{Variable} \\
\\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1081 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

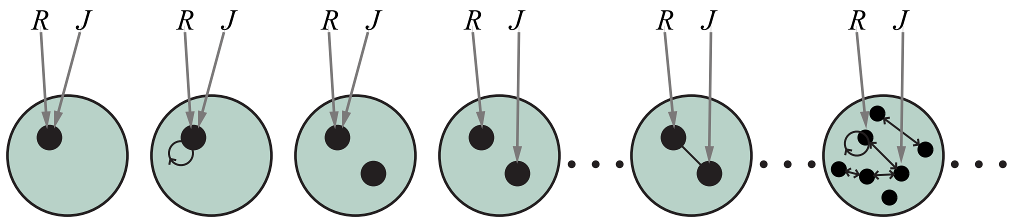


Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

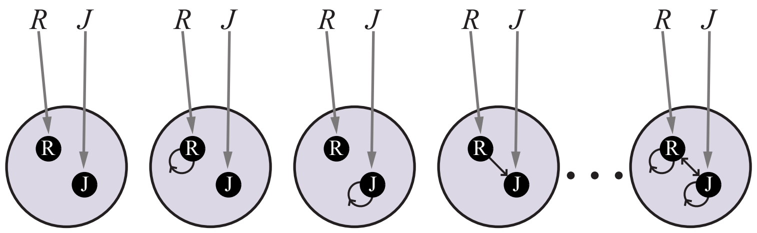


Figure 8.5 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol, under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

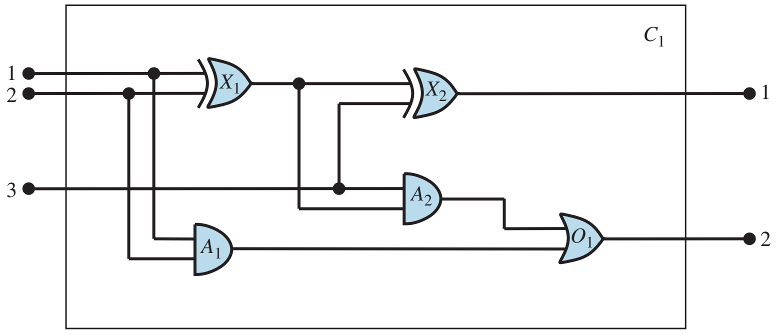


Figure 8.6 A digital circuit C_1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

CHAPTER 9

INFERENCE IN FIRST-ORDER LOGIC

```
function UNIFY($x, y, \theta$=empty) returns a substitution to make $x$ and $y$ identical, or failure
  if $\theta$ = failure then return failure
  else if $x = y$ then return $\theta$
  else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
  else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
  else if COMPOUND?($x$) and COMPOUND?($y$) then
      return UNIFY(ARGS($x$), ARGS($y$), UNIFY(OP($x$), OP($y$), $\theta$))
  else if LIST?($x$) and LIST?($y$) then
      return UNIFY(REST($x$), REST($y$), UNIFY(FIRST($x$), FIRST($y$), $\theta$))
  else return failure

function UNIFY-VAR($var, x, \theta$) returns a substitution
  if $\{var/val\} \in \theta$ for some $val$ then return UNIFY($val, x, \theta$)
  else if $\{x/val\} \in \theta$ for some $val$ then return UNIFY($var, val, \theta$)
  else if OCCUR-CHECK?($var, x$) then return failure
  else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with $\{var/val\}$ pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as $F(A,B)$, OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A,B).

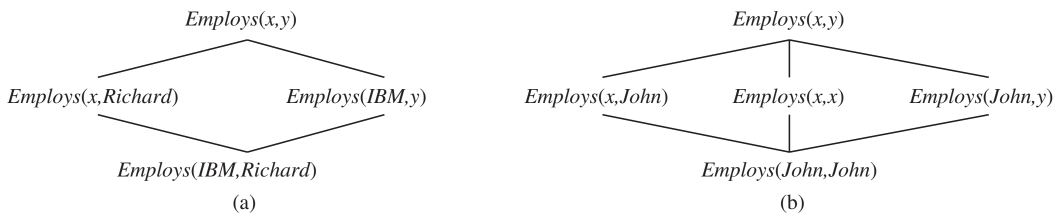


Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

```
function FOL-FC-ASK($KB, \alpha$) returns a substitution or false
  inputs: $KB$, the knowledge base, a set of first-order definite clauses
          $\alpha$, the query, an atomic sentence

  while true do
     $new \leftarrow \{\}$        // The set of new sentences inferred on each iteration
     for each rule in $KB$ do
        $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
        for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                    for some $p'_1, \ldots, p'_n$ in $KB$
           $q' \leftarrow$ SUBST($\theta, q$)
           if $q'$ does not unify with some sentence already in $KB$ or new then
              add $q'$ to new
              $\phi \leftarrow$ UNIFY($q', \alpha$)
              if $\phi$ is not failure then return $\phi$
     if $new = \{\}$ then return false
     add new to $KB$
```

Figure 9.3 A conceptually straightforward, but inefficient, forward-chaining algorithm. On each iteration, it adds to *KB* all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in *KB*. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

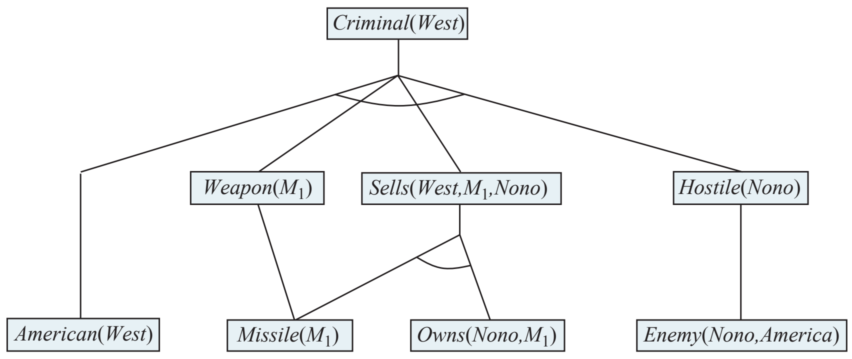
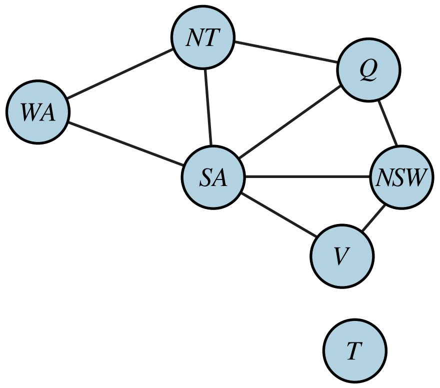


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.



$$\begin{aligned}
&\mathit{Diff}(wa,nt) \land \mathit{Diff}(wa,sa) \land \\
&\quad \mathit{Diff}(nt,q) \land \mathit{Diff}(nt,sa) \land \\
&\quad \mathit{Diff}(q,nsw) \land \mathit{Diff}(q,sa) \land \\
&\quad \mathit{Diff}(nsw,v) \land \mathit{Diff}(nsw,sa) \land \\
&\quad \mathit{Diff}(v,sa) \Rightarrow \mathit{Colorable}()
\end{aligned}$$

$$\begin{array}{ll}
\mathit{Diff}(Red,Blue) & \mathit{Diff}(Red,Green) \\
\mathit{Diff}(Green,Red) & \mathit{Diff}(Green,Blue) \\
\mathit{Diff}(Blue,Red) & \mathit{Diff}(Blue,Green)
\end{array}$$

(a) (b)

Figure 9.5 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants *Red*, *Green*, or *Blue* (which are declared *Diff*).

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

function FOL-BC-OR(KB, goal, θ) returns a substitution
  for each rule in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs ⇒ rhs) ← STANDARDIZE-VARIABLES(rule)
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

function FOL-BC-AND(KB, goals, θ) returns a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else
    first,rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

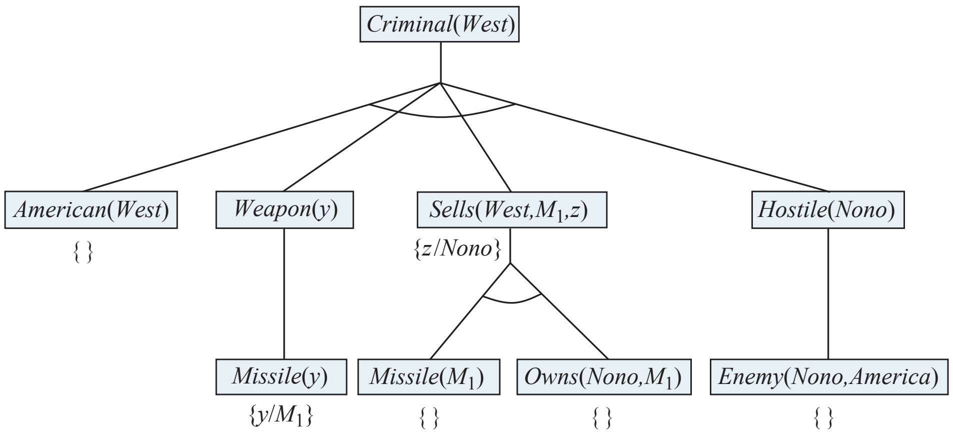


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

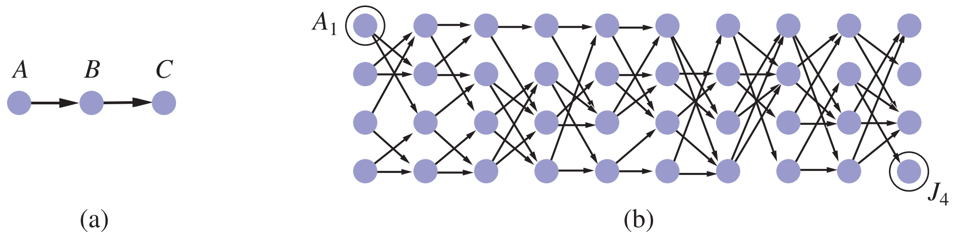


Figure 9.8 (a) Finding a path from A to C can lead Prolog into an infinite loop. (b) A graph in which each node is connected to two random successors in the next layer. Finding a path from A_1 to J_4 requires 877 inferences.

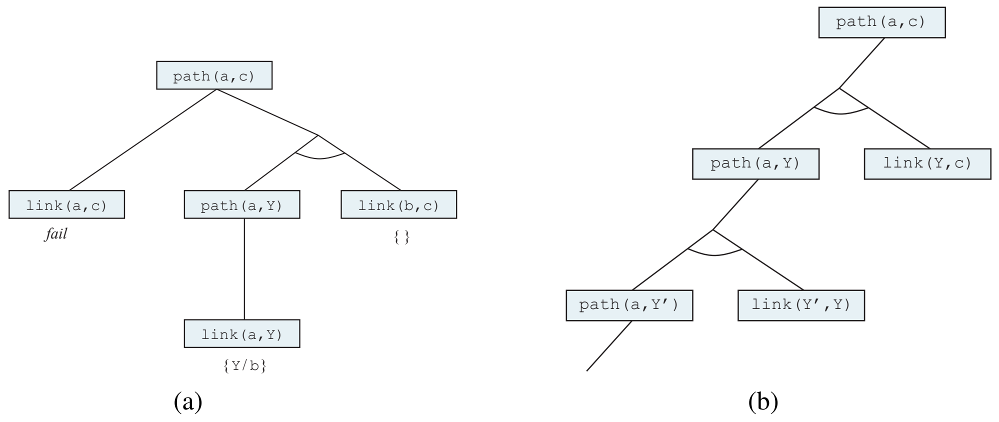
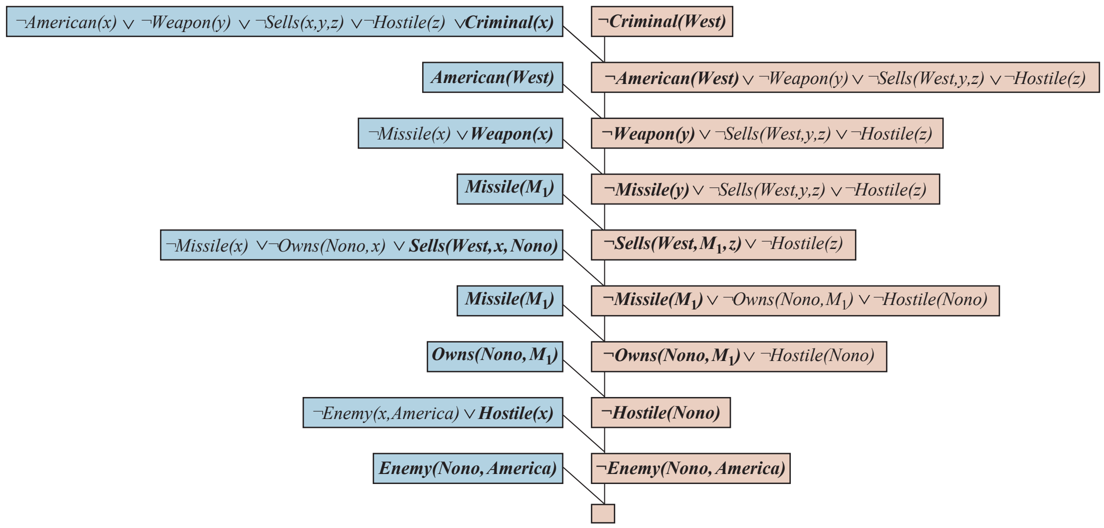


Figure 9.9 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.



¬American(x) ∨ ¬Weapon(y) ∨ ¬Sells(x,y,z) ∨ ¬Hostile(z) ∨ Criminal(x)
¬Criminal(West)
American(West)
¬American(West) ∨ ¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ Weapon(x)
¬Weapon(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(y) ∨ ¬Sells(West,y,z) ∨ ¬Hostile(z)
¬Missile(x) ∨ ¬Owns(Nono,x) ∨ Sells(West,x,Nono)
¬Sells(West,M1,z) ∨ ¬Hostile(z)
Missile(M1)
¬Missile(M1) ∨ ¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
Owns(Nono,M1)
¬Owns(Nono,M1) ∨ ¬Hostile(Nono)
¬Enemy(x,America) ∨ Hostile(x)
¬Hostile(Nono)
Enemy(Nono,America)
¬Enemy(Nono,America)

Figure 9.10 A resolution proof that West is a criminal. At each resolution step, the literals that unify are in bold and the clause with the positive literal is shaded blue.

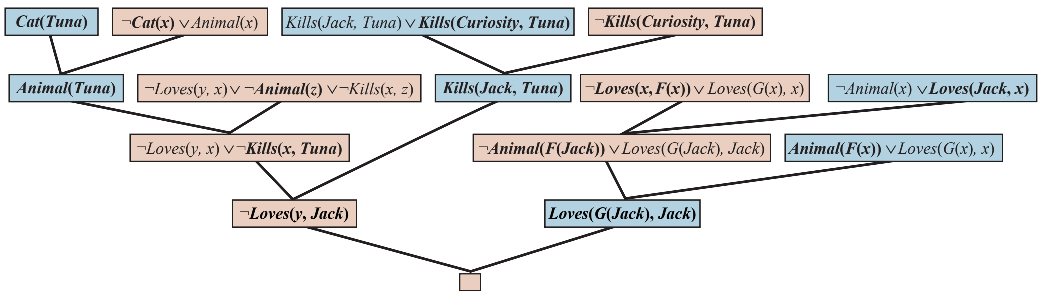

Cat(Tuna)
¬Cat(x) ∨ Animal(x)
Kills(Jack, Tuna) ∨ Kills(Curiosity, Tuna)
¬Kills(Curiosity, Tuna)
Animal(Tuna)
¬Loves(y, x) ∨ ¬Animal(z) ∨ ¬Kills(x, z)
Kills(Jack, Tuna)
¬Loves(x, F(x)) ∨ Loves(G(x), x)
¬Animal(x) ∨ Loves(Jack, x)
¬Loves(y, x) ∨ ¬Kills(x, Tuna)
¬Animal(F(Jack)) ∨ Loves(G(Jack), Jack)
Animal(F(x)) ∨ Loves(G(x), x)
¬Loves(y, Jack)
Loves(G(Jack), Jack)

Figure 9.11 A resolution proof that Curiosity killed the cat. Notice the use of factoring in the derivation of the clause $Loves(G(Jack), Jack)$. Notice also in the upper right, the unification of $Loves(x, F(x))$ and $Loves(Jack, x)$ can only succeed after the variables have been standardized apart.

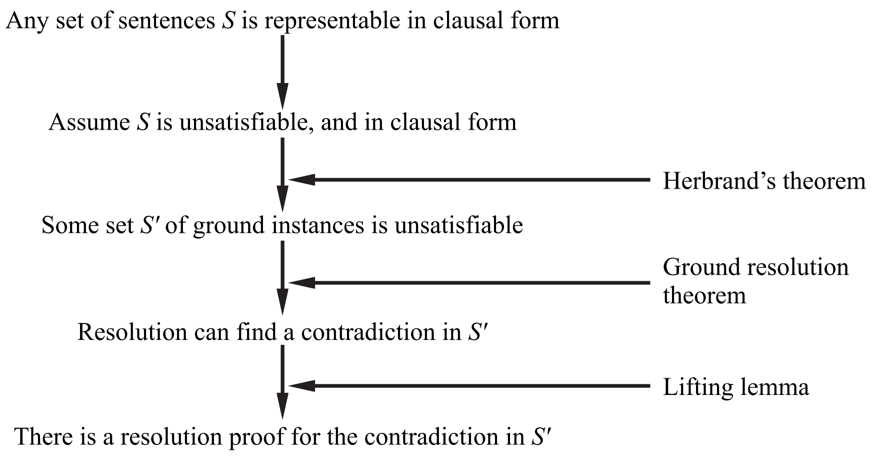

Any set of sentences S is representable in clausal form
Assume S is unsatisfiable, and in clausal form
Herbrand's theorem
Some set S′ of ground instances is unsatisfiable
Ground resolution theorem
Resolution can find a contradiction in S′
Lifting lemma
There is a resolution proof for the contradiction in S′

Figure 9.12 Structure of a completeness proof for resolution.

CHAPTER 10

KNOWLEDGE REPRESENTATION

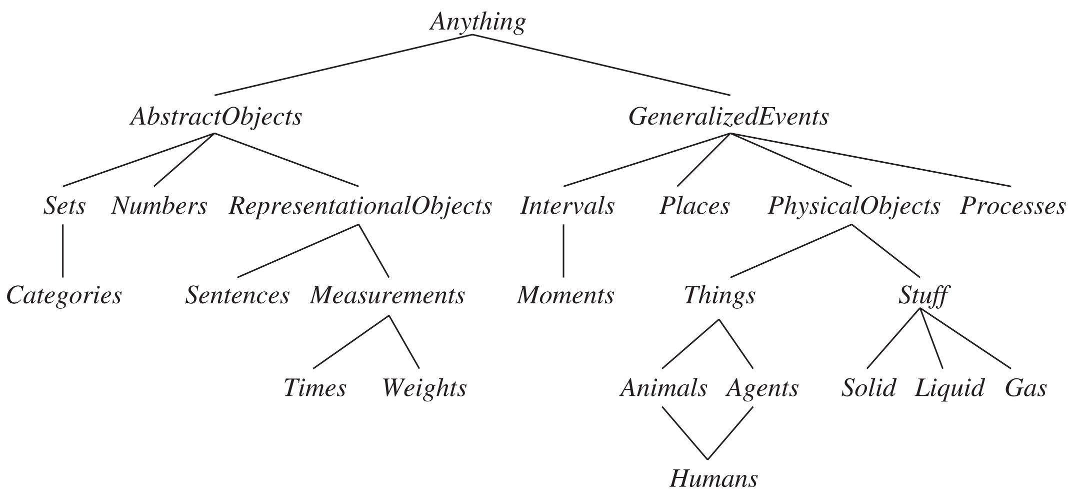


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint—a human is both an animal and an agent. We will see in Section 10.3.2 why physical objects come under generalized events.

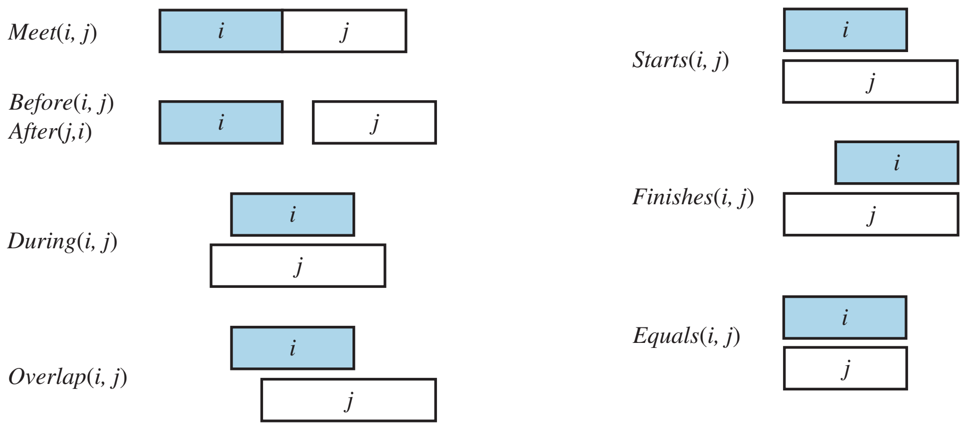


Figure 10.2 Predicates on time intervals.

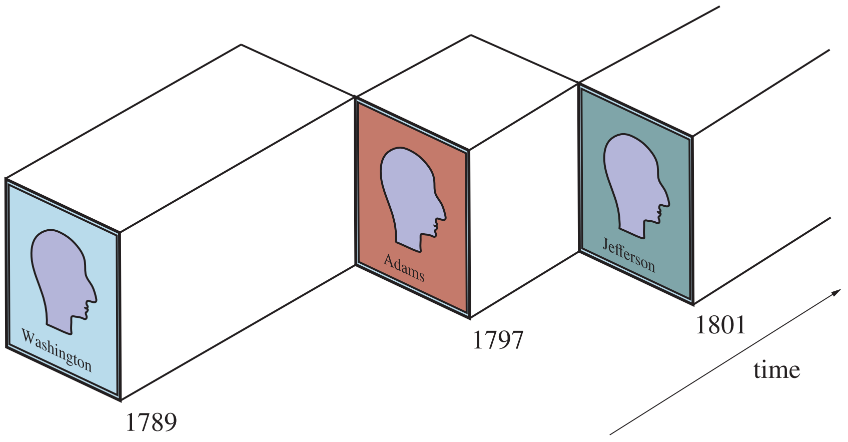


Figure 10.3 A schematic view of the object *President*(*USA*) for the early years.

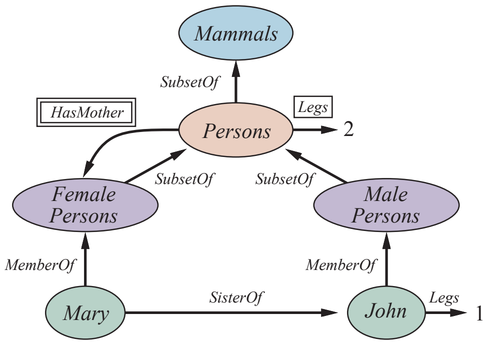


Figure 10.4 A semantic network with four objects (John, Mary, 1, and 2) and four categories. Relations are denoted by labeled links.

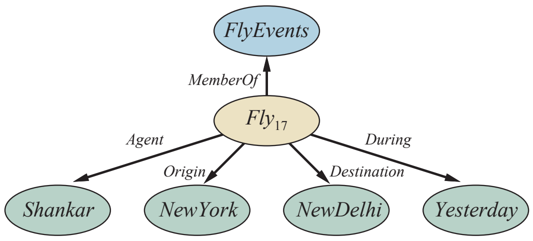


Figure 10.5 A fragment of a semantic network showing the representation of the logical assertion *Fly*(*Shankar*, *NewYork*, *NewDelhi*, *Yesterday*).

$$
\begin{array}{rcl}
\mathit{Concept} & \rightarrow & \textbf{Thing} \mid \mathit{ConceptName} \\
 & \mid & \textbf{And}(\mathit{Concept}, \ldots) \\
 & \mid & \textbf{All}(\mathit{RoleName}, \mathit{Concept}) \\
 & \mid & \textbf{AtLeast}(\mathit{Integer}, \mathit{RoleName}) \\
 & \mid & \textbf{AtMost}(\mathit{Integer}, \mathit{RoleName}) \\
 & \mid & \textbf{Fills}(\mathit{RoleName}, \mathit{IndividualName}, \ldots) \\
 & \mid & \textbf{SameAs}(\mathit{Path}, \mathit{Path}) \\
 & \mid & \textbf{OneOf}(\mathit{IndividualName}, \ldots) \\
\mathit{Path} & \rightarrow & [\mathit{RoleName}, \ldots] \\
\mathit{ConceptName} & \rightarrow & \mathit{Adult} \mid \mathit{Female} \mid \mathit{Male} \mid \ldots \\
\mathit{RoleName} & \rightarrow & \mathit{Spouse} \mid \mathit{Daughter} \mid \mathit{Son} \mid \ldots
\end{array}
$$

Figure 10.6 The syntax of descriptions in a subset of the CLASSIC language.

CHAPTER 11

AUTOMATED PLANNING

$$\begin{array}{l}
\mathit{Init}(\mathit{At}(C_1,\ \mathit{SFO}) \wedge \mathit{At}(C_2,\ \mathit{JFK}) \wedge \mathit{At}(P_1,\ \mathit{SFO}) \wedge \mathit{At}(P_2,\ \mathit{JFK}) \\
\quad \wedge\, \mathit{Cargo}(C_1) \wedge \mathit{Cargo}(C_2) \wedge \mathit{Plane}(P_1) \wedge \mathit{Plane}(P_2) \\
\quad \wedge\, \mathit{Airport}(\mathit{JFK}) \wedge \mathit{Airport}(\mathit{SFO})) \\
\mathit{Goal}(\mathit{At}(C_1,\ \mathit{JFK}) \wedge \mathit{At}(C_2,\ \mathit{SFO})) \\
\mathit{Action}(\mathit{Load}(c,\ p,\ a), \\
\quad \textsc{Precond}{:}\ \mathit{At}(c,\ a) \wedge \mathit{At}(p,\ a) \wedge \mathit{Cargo}(c) \wedge \mathit{Plane}(p) \wedge \mathit{Airport}(a) \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{At}(c,\ a) \wedge \mathit{In}(c,\ p)) \\
\mathit{Action}(\mathit{Unload}(c,\ p,\ a), \\
\quad \textsc{Precond}{:}\ \mathit{In}(c,\ p) \wedge \mathit{At}(p,\ a) \wedge \mathit{Cargo}(c) \wedge \mathit{Plane}(p) \wedge \mathit{Airport}(a) \\
\quad \textsc{Effect}{:}\ \mathit{At}(c,\ a) \wedge \neg\, \mathit{In}(c,\ p)) \\
\mathit{Action}(\mathit{Fly}(p,\ \mathit{from},\ \mathit{to}), \\
\quad \textsc{Precond}{:}\ \mathit{At}(p,\ \mathit{from}) \wedge \mathit{Plane}(p) \wedge \mathit{Airport}(\mathit{from}) \wedge \mathit{Airport}(\mathit{to}) \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{At}(p,\ \mathit{from}) \wedge \mathit{At}(p,\ \mathit{to}))
\end{array}$$

Figure 11.1 A PDDL description of an air cargo transportation planning problem.

$$\begin{array}{l}
\mathit{Init}(\mathit{Tire}(\mathit{Flat}) \wedge \mathit{Tire}(\mathit{Spare}) \wedge \mathit{At}(\mathit{Flat},\mathit{Axle}) \wedge \mathit{At}(\mathit{Spare},\mathit{Trunk})) \\
\mathit{Goal}(\mathit{At}(\mathit{Spare},\mathit{Axle})) \\
\mathit{Action}(\mathit{Remove}(\mathit{obj},\mathit{loc}), \\
\quad \textsc{Precond}{:}\ \mathit{At}(\mathit{obj},\mathit{loc}) \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{At}(\mathit{obj},\mathit{loc}) \wedge \mathit{At}(\mathit{obj},\mathit{Ground})) \\
\mathit{Action}(\mathit{PutOn}(t,\ \mathit{Axle}), \\
\quad \textsc{Precond}{:}\ \mathit{Tire}(t) \wedge \mathit{At}(t,\mathit{Ground}) \wedge \neg\, \mathit{At}(\mathit{Flat},\mathit{Axle}) \wedge \neg\, \mathit{At}(\mathit{Spare},\mathit{Axle}) \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{At}(t,\mathit{Ground}) \wedge \mathit{At}(t,\mathit{Axle})) \\
\mathit{Action}(\mathit{LeaveOvernight}, \\
\quad \textsc{Precond}{:} \\
\quad \textsc{Effect}{:}\ \neg\, \mathit{At}(\mathit{Spare},\mathit{Ground}) \wedge \neg\, \mathit{At}(\mathit{Spare},\mathit{Axle}) \wedge \neg\, \mathit{At}(\mathit{Spare},\mathit{Trunk}) \\
\qquad\qquad \wedge \neg\, \mathit{At}(\mathit{Flat},\mathit{Ground}) \wedge \neg\, \mathit{At}(\mathit{Flat},\mathit{Axle}) \wedge \neg\, \mathit{At}(\mathit{Flat},\ \mathit{Trunk}))
\end{array}$$

Figure 11.2 The simple spare tire problem.

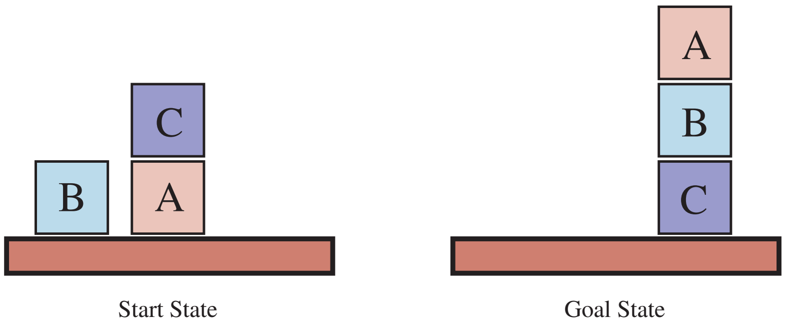


Figure 11.3 Diagram of the blocks-world problem in Figure 11.4.

$$
\begin{array}{l}
\mathit{Init}(\mathit{On}(A,\mathit{Table}) \land \mathit{On}(B,\mathit{Table}) \land \mathit{On}(C,A) \\
\quad \land\ \mathit{Block}(A) \land \mathit{Block}(B) \land \mathit{Block}(C) \land \mathit{Clear}(B) \land \mathit{Clear}(C) \land \mathit{Clear}(\mathit{Table})) \\
\mathit{Goal}(\mathit{On}(A,B) \land \mathit{On}(B,C)) \\
\mathit{Action}(\mathit{Move}(b,x,y), \\
\quad \textsc{Precond}: \mathit{On}(b,x) \land \mathit{Clear}(b) \land \mathit{Clear}(y) \land \mathit{Block}(b) \land \mathit{Block}(y) \land \\
\qquad\qquad (b \neq x) \land (b \neq y) \land (x \neq y), \\
\quad \textsc{Effect}: \mathit{On}(b,y) \land \mathit{Clear}(x) \land \neg \mathit{On}(b,x) \land \neg \mathit{Clear}(y)) \\
\mathit{Action}(\mathit{MoveToTable}(b,x), \\
\quad \textsc{Precond}: \mathit{On}(b,x) \land \mathit{Clear}(b) \land \mathit{Block}(b) \land \mathit{Block}(x), \\
\quad \textsc{Effect}: \mathit{On}(b,\mathit{Table}) \land \mathit{Clear}(x) \land \neg \mathit{On}(b,x))
\end{array}
$$

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[\mathit{MoveToTable}(C,A), \mathit{Move}(B,\mathit{Table},C), \mathit{Move}(A,\mathit{Table},B)]$.

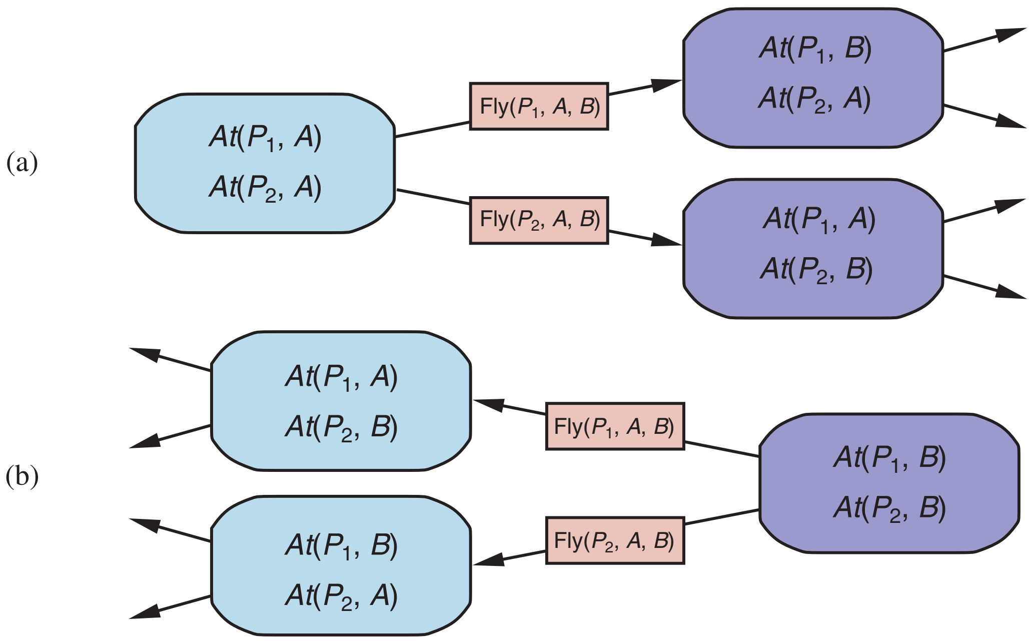


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

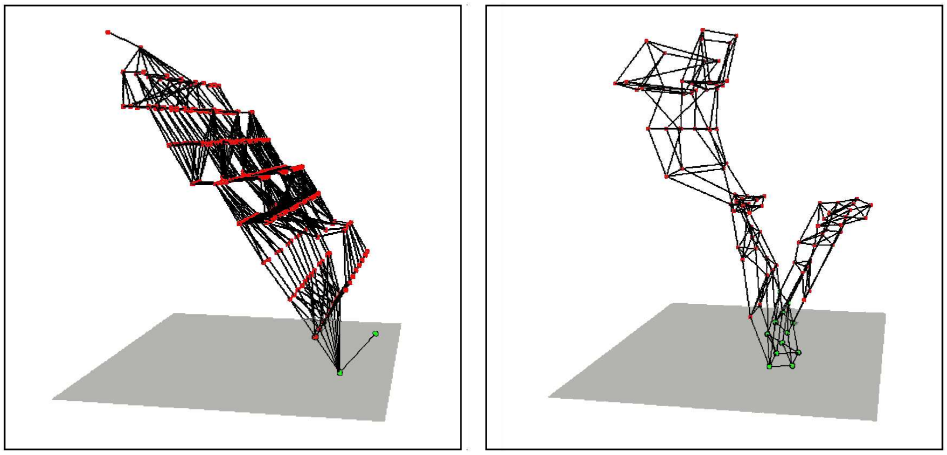


Figure 11.6 Two state spaces from planning problems with the ignore-delete-lists heuristic. The height above the bottom plane is the heuristic score of a state; states on the bottom plane are goals. There are no local minima, so search for the goal is straightforward. From Hoffmann (2005).

```
$Refinement(Go(Home, SFO),$
   STEPS: $[Drive(Home, SFOLongTermParking),$
            $Shuttle(SFOLongTermParking, SFO)]$ )
$Refinement(Go(Home, SFO),$
   STEPS: $[Taxi(Home, SFO)]$ )

$Refinement(Navigate([a,b],[x,y]),$
   PRECOND: $a = x \land b = y$
   STEPS: [] )
$Refinement(Navigate([a,b],[x,y]),$
   PRECOND: $Connected([a,b],[a-1,b])$
   STEPS: $[Left, Navigate([a-1,b],[x,y])]$ )
$Refinement(Navigate([a,b],[x,y]),$
   PRECOND: $Connected([a,b],[a+1,b])$
   STEPS: $[Right, Navigate([a+1,b],[x,y])]$ )
...
```

Figure 11.7 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution or failure

  frontier ← a FIFO queue with [Act] as the only element
  while true do
      if IS-EMPTY(frontier) then return failure
      plan ← POP(frontier)          // chooses the shallowest plan in frontier
      hla ← the first HLA in plan, or null if none
      prefix,suffix ← the action subsequences before and after hla in plan
      outcome ← RESULT(problem.INITIAL, prefix)
      if hla is null then          // so plan is primitive and outcome is its result
          if problem.IS-GOAL(outcome) then return plan
      else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
          add APPEND(prefix, sequence, suffix) to frontier
```

Figure 11.8 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

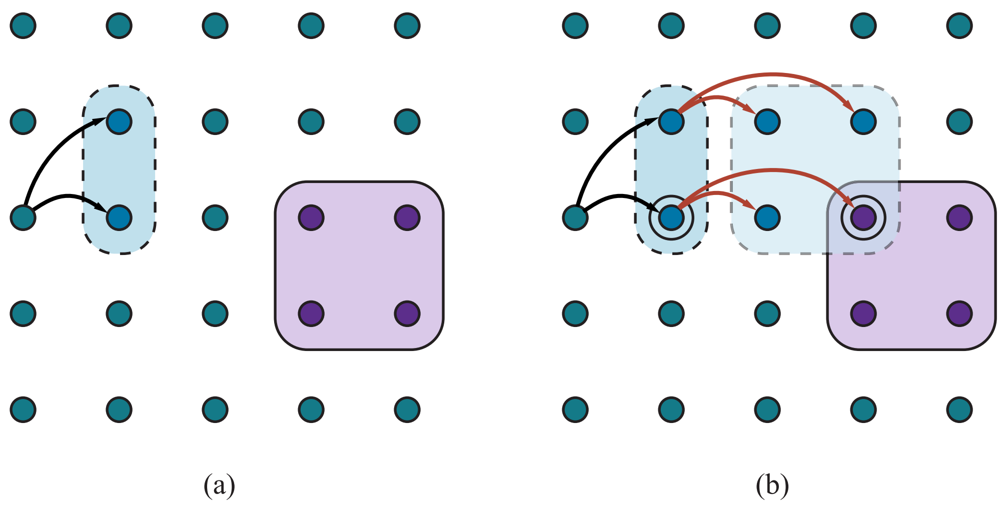


Figure 11.9 Schematic examples of reachable sets. The set of goal states is shaded in purple. Black and red arrows indicate possible implementations of h_1 and h_2, respectively. (a) The reachable set of an HLA h_1 in a state s. (b) The reachable set for the sequence $[h_1, h_2]$. Because this intersects the goal set, the sequence achieves the goal.

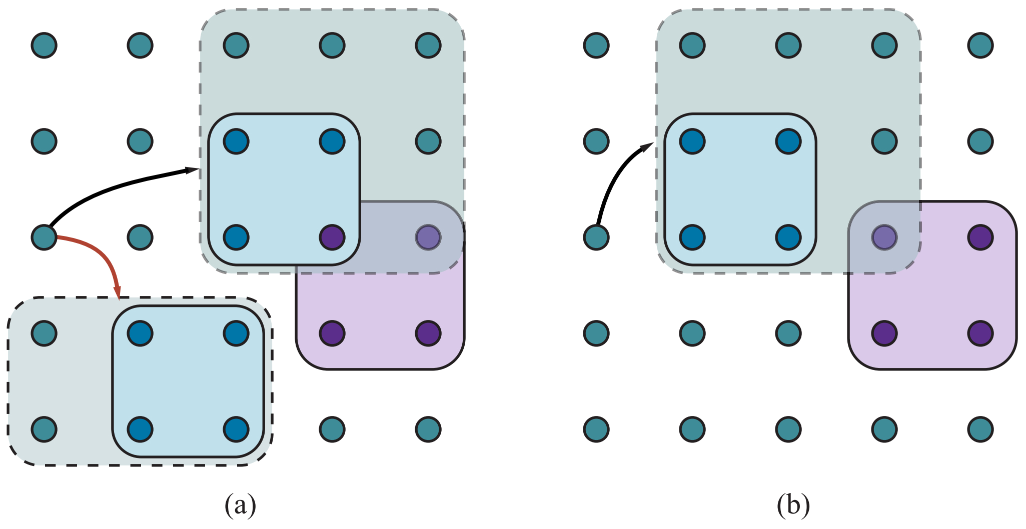


Figure 11.10 Goal achievement for high-level plans with approximate descriptions. The set of goal states is shaded in purple. For each plan, the pessimistic (solid lines, light blue) and optimistic (dashed lines, light green) reachable sets are shown. (a) The plan indicated by the black arrow definitely achieves the goal, while the plan indicated by the red arrow definitely doesn't. (b) A plan that *possibly* achieves the goal (the optimistic reachable set intersects the goal) but does not *necessarily* achieve the goal (the pessimistic reachable set does not intersect the goal). The plan would need to be refined further to determine if it really does achieve the goal.

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns a solution or fail
  frontier ← a FIFO queue with initialPlan as the only element
  while true do
     if IS-EMPTY?(frontier) then return fail
     plan ← POP(frontier)        // chooses the shallowest node in frontier
     if REACH$^+$(problem.INITIAL, plan) intersects problem.GOAL then
        if plan is primitive then return plan        // REACH$^+$ is exact for primitive plans
        guaranteed ← REACH$^-$(problem.INITIAL, plan) $\cap$ problem.GOAL
        if guaranteed $\neq$ { } and MAKING-PROGRESS(plan, initialPlan) then
           finalState ← any element of guaranteed
           return DECOMPOSE(hierarchy, problem.INITIAL, plan, finalState)
        hla ← some HLA in plan
        prefix,suffix ← the action subsequences before and after hla in plan
        outcome ← RESULT(problem.INITIAL, prefix)
        for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
           add APPEND(prefix, sequence, suffix) to frontier

function DECOMPOSE(hierarchy, $s_0$, plan, $s_f$) returns a solution
  solution ← an empty plan
  while plan is not empty do
    action ← REMOVE-LAST(plan)
    $s_i$ ← a state in REACH$^-$($s_0$, plan) such that $s_f \in$ REACH$^-$($s_i$, action)
    problem ← a problem with INITIAL = $s_i$ and GOAL = $s_f$
    solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    $s_f$ ← $s_i$
  return solution
```

Figure 11.11 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the *initialPlan*.

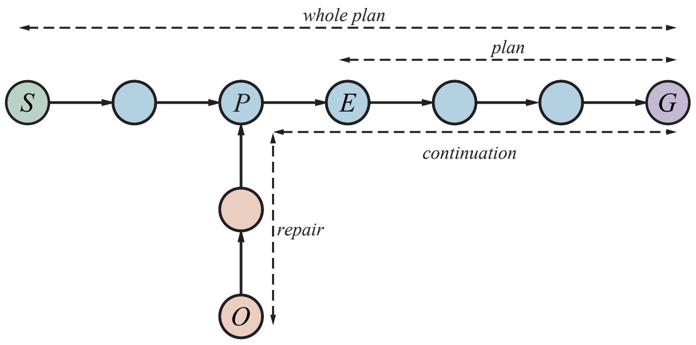


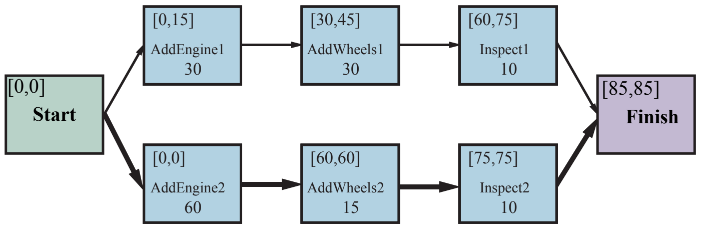
Figure 11.12 At first, the sequence "whole plan" is expected to get the agent from S to G. The agent executes steps of the plan until it expects to be in state E, but observes that it is actually in O. The agent then replans for the minimal *repair* plus *continuation* to reach G.

$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$
$\quad \{AddEngine2 \prec AddWheels2 \prec Inspect2\})$

$Resources(EngineHoists(1),\ WheelStations(1),\ Inspectors(2),\ LugNuts(500))$

$Action(AddEngine1,\ \textsc{Duration}{:}30,$
$\quad \textsc{Use}{:}EngineHoists(1))$
$Action(AddEngine2,\ \textsc{Duration}{:}60,$
$\quad \textsc{Use}{:}EngineHoists(1))$
$Action(AddWheels1,\ \textsc{Duration}{:}30,$
$\quad \textsc{Consume}{:}LugNuts(20),\ \textsc{Use}{:}WheelStations(1))$
$Action(AddWheels2,\ \textsc{Duration}{:}15,$
$\quad \textsc{Consume}{:}LugNuts(20),\ \textsc{Use}{:}WheelStations(1))$
$Action(Inspect_i,\ \textsc{Duration}{:}10,$
$\quad \textsc{Use}{:}Inspectors(1))$

Figure 11.13 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.



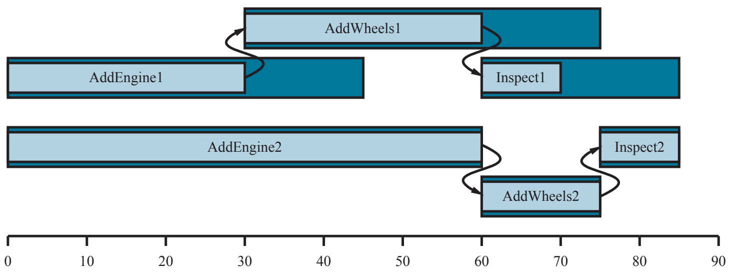


Figure 11.14 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.13. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Blue rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a blue rectangle indicates the slack.

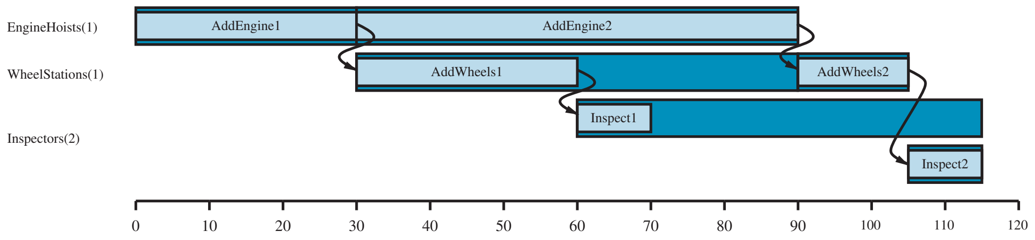


Figure 11.15 A solution to the job-shop scheduling problem from Figure 11.13, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

CHAPTER 12

QUANTIFYING UNCERTAINTY

```
function DT-AGENT(percept) returns an action
   persistent: belief_state, probabilistic beliefs about the current state of the world
               action, the agent's action

   update belief_state based on action and percept
   calculate outcome probabilities for actions,
      given action descriptions and current belief_state
   select action with highest expected utility
      given probabilities of outcomes and utility information
   return action
```

Figure 12.1 A decision-theoretic agent that selects rational actions.

Proposition	Agent 1's belief	Agent 2 bets	Agent 1 bets	Agent 1 payoffs for each outcome			
				a, b	$a, \neg b$	$\neg a, b$	$\neg a, \neg b$
a	0.4	\$4 on a	\$6 on $\neg a$	–\$6	–\$6	\$4	\$4
b	0.3	\$3 on b	\$7 on $\neg b$	–\$7	\$3	–\$7	\$3
$a \vee b$	0.8	\$2 on $\neg(a \vee b)$	\$8 on $a \vee b$	\$2	\$2	\$2	–\$8
				–\$11	–\$1	–\$1	–\$1

Figure 12.2 Because Agent 1 has inconsistent beliefs, Agent 2 is able to devise a set of three bets that guarantees a loss for Agent 1, no matter what the outcome of *a* and *b*.

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 12.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

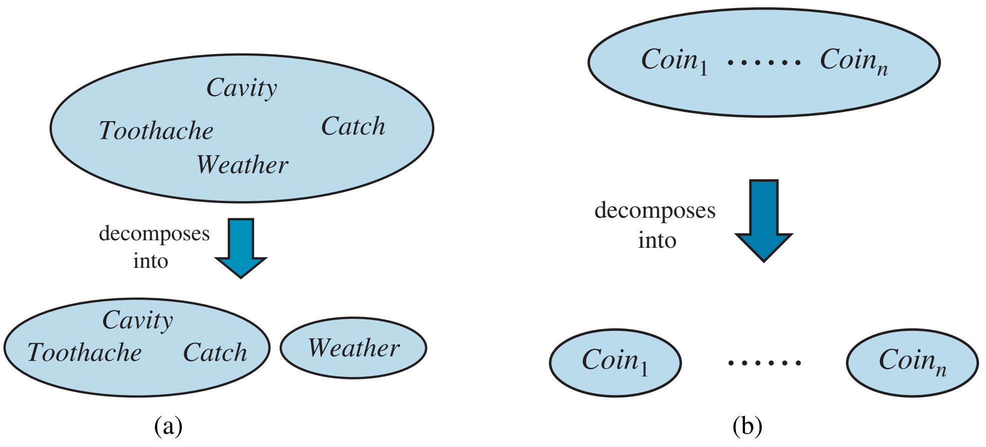


Figure 12.4 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

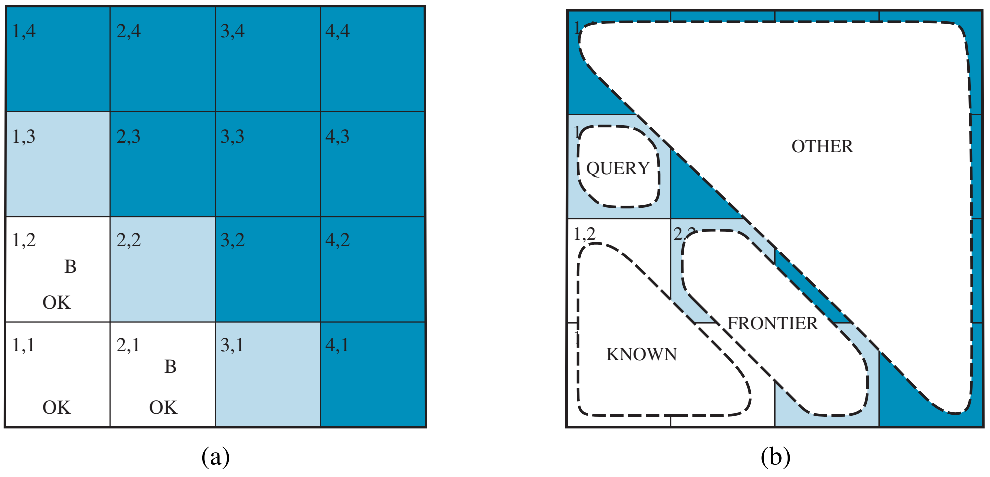


Figure 12.5 (a) After finding a breeze in both [1,2] and [2,1], the agent is stuck—there is no safe place to explore. (b) Division of the squares into *Known*, *Frontier*, and *Other*, for a query about [1,3].

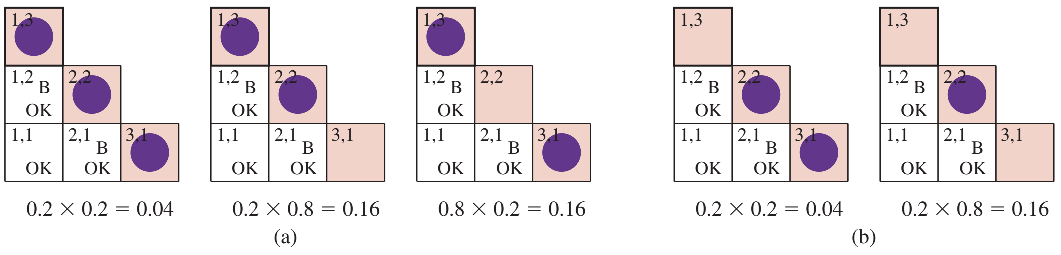


Figure 12.6 Consistent models for the frontier variables, $P_{2,2}$ and $P_{3,1}$, showing $P(frontier)$ for each model: (a) three models with $P_{1,3} = true$ showing two or three pits, and (b) two models with $P_{1,3} = false$ showing one or two pits.

CHAPTER 13

PROBABILISTIC REASONING

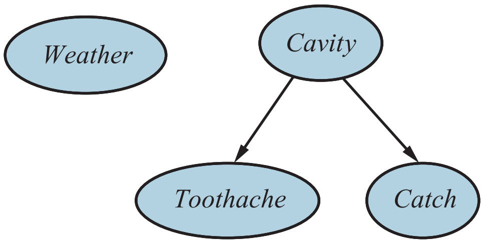


Figure 13.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

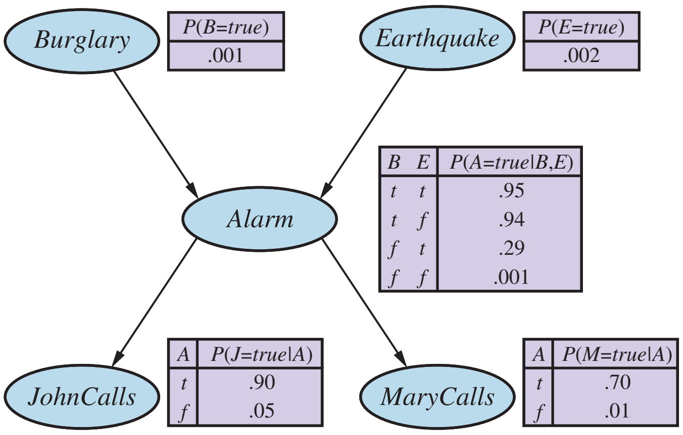


Figure 13.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters *B*, *E*, *A*, *J*, and *M* stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

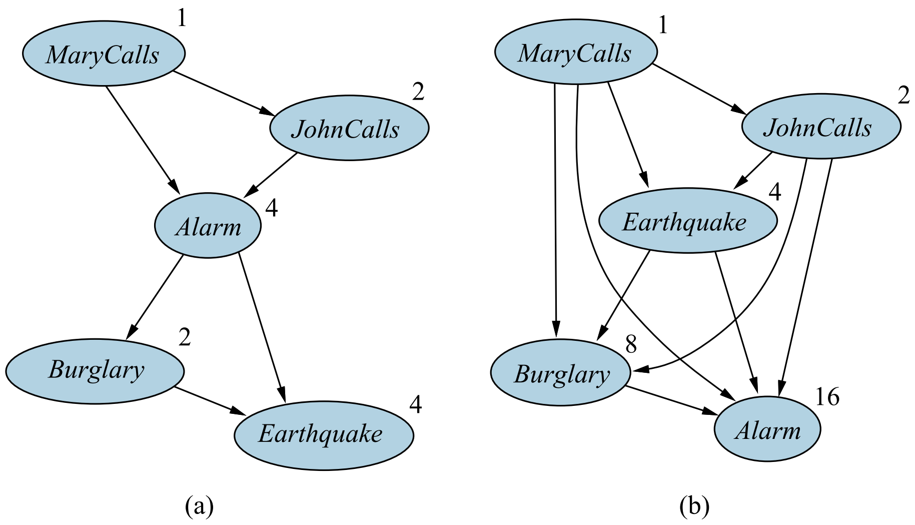


Figure 13.3 Network structure and number of parameters depends on order of introduction. (a) The structure obtained with ordering M,J,A,B,E. (b) The structure obtained with M,J,E,B,A. Each node is annotated with the number of parameters required; 13 in all for (a) and 31 for (b). In Figure 13.2, only 10 parameters were required.

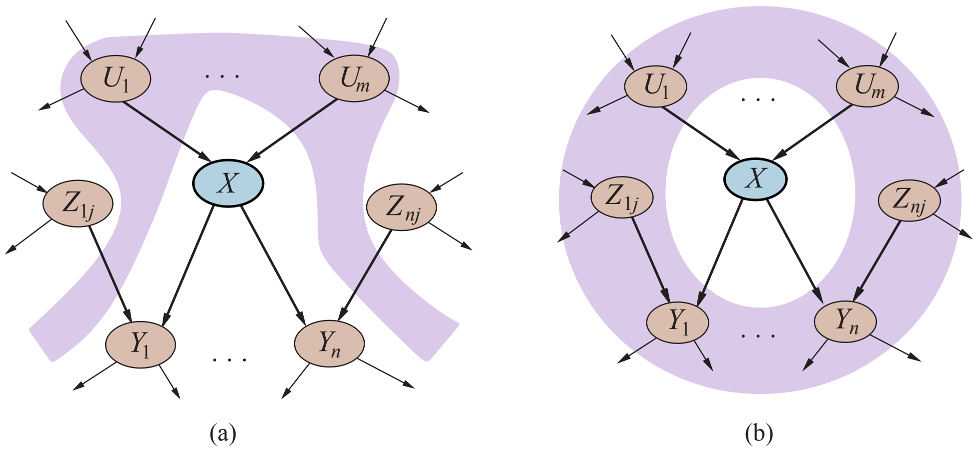


Figure 13.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the lavender area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the lavender area).

Cold	*Flu*	*Malaria*	$P(fever \mid \cdot)$	$P(\neg fever \mid \cdot)$
f	*f*	*f*	0.0	1.0
f	*f*	*t*	0.9	**0.1**
f	*t*	*f*	0.8	**0.2**
f	*t*	*t*	0.98	$0.02 = 0.2 \times 0.1$
t	*f*	*f*	0.4	**0.6**
t	*f*	*t*	0.94	$0.06 = 0.6 \times 0.1$
t	*t*	*f*	0.88	$0.12 = 0.6 \times 0.2$
t	*t*	*t*	0.988	$0.012 = 0.6 \times 0.2 \times 0.1$

Figure 13.5 A complete conditional probability table for $\mathbf{P}(Fever \mid Cold, Flu, Malaria)$, assuming a noisy-OR model with the the three q-values shown in bold.

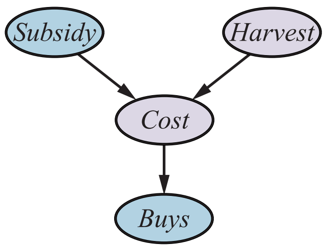
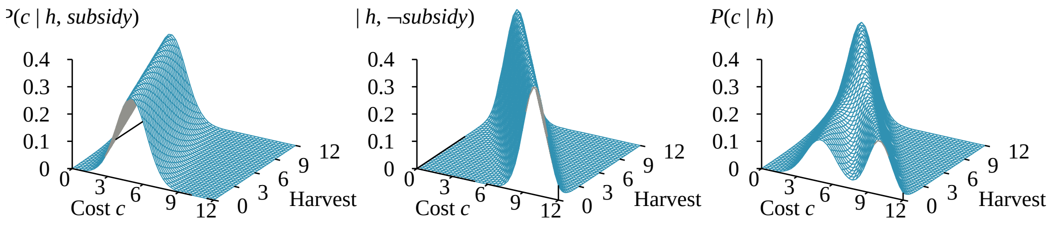


Figure 13.6 A simple network with discrete variables (*Subsidy* and *Buys*) and continuous variables (*Harvest* and *Cost*).



(a) (b) (c)

Figure 13.7 The graphs in (a) and (b) show the probability distribution over *Cost* as a function of *Harvest* size, with *Subsidy* true and false, respectively. Graph (c) shows the distribution $P(Cost \mid Harvest)$, obtained by summing over the two subsidy cases.

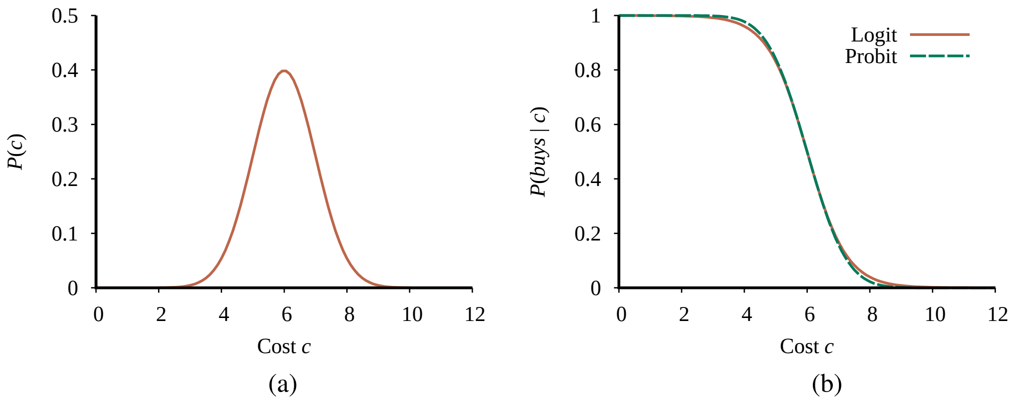


Figure 13.8 (a) A normal (Gaussian) distribution for the cost threshold, centered on $\mu = 6.0$ with standard deviation $\sigma = 1.0$. (b) Expit and probit models for the probability of *buys* given *cost*, for the parameters $\mu = 6.0$ and $\sigma = 1.0$.

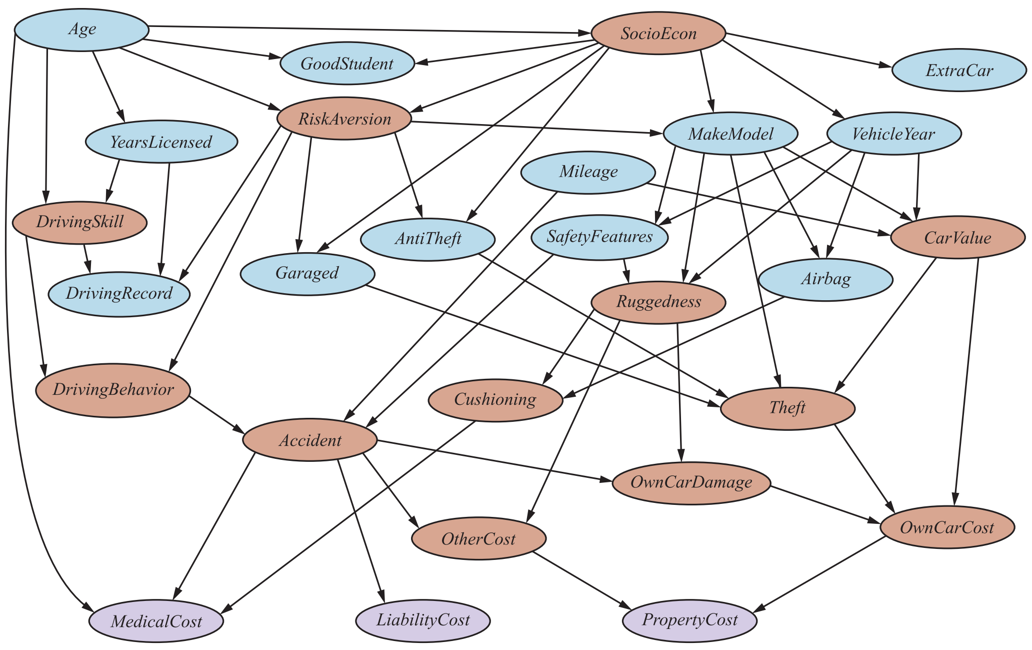


Figure 13.9 A Bayesian network for evaluating car insurance applications.

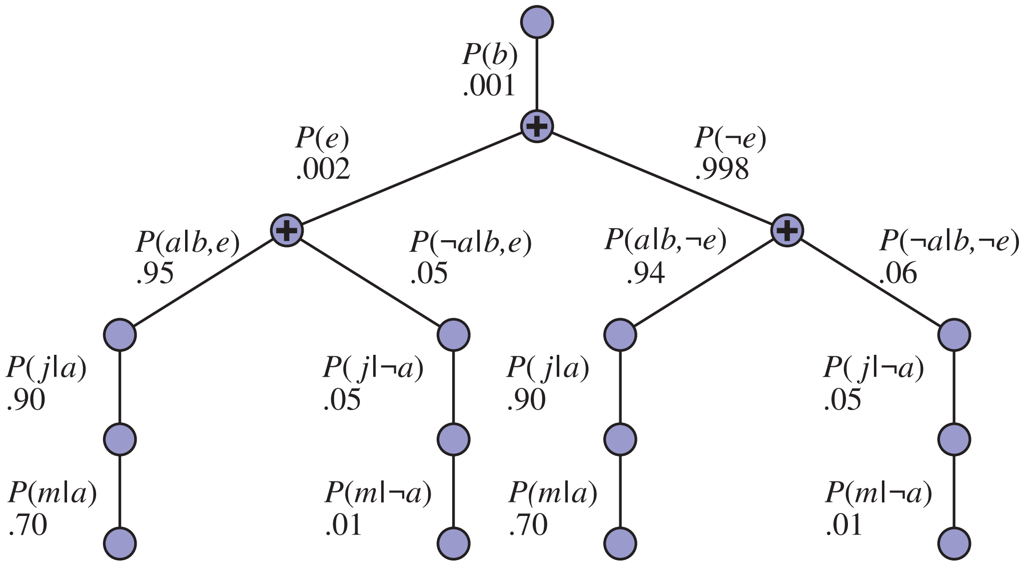


Figure 13.10 The structure of the expression shown in Equation (13.5). The evaluation proceeds top down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

```
function ENUMERATION-ASK($X$, $\mathbf{e}$, $bn$) returns a distribution over $X$
  inputs: $X$, the query variable
          $\mathbf{e}$, observed values for variables $\mathbf{E}$
          $bn$, a Bayes net with variables $vars$

  $\mathbf{Q}(X) \leftarrow$ a distribution over $X$, initially empty
  for each value $x_i$ of $X$ do
      $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL($vars$, $\mathbf{e}_{x_i}$)
          where $\mathbf{e}_{x_i}$ is $\mathbf{e}$ extended with $X = x_i$
  return NORMALIZE($\mathbf{Q}(X)$)

function ENUMERATE-ALL($vars$, $\mathbf{e}$) returns a real number
  if EMPTY?($vars$) then return 1.0
  $V \leftarrow$ FIRST($vars$)
  if $V$ is an evidence variable with value $v$ in $\mathbf{e}$
      then return $P(v \mid parents(V)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}$)
      else return $\sum_v P(v \mid parents(V)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_v$)
          where $\mathbf{e}_v$ is $\mathbf{e}$ extended with $V = v$
```

Figure 13.11 The enumeration algorithm for exact inference in Bayes nets.

X	Y	$\mathbf{f}(X,Y)$	Y	Z	$\mathbf{g}(Y,Z)$	X	Y	Z	$\mathbf{h}(X,Y,Z)$
t	t	.3	t	t	.2	t	t	t	$.3 \times .2 = .06$
t	f	.7	t	f	.8	t	t	f	$.3 \times .8 = .24$
f	t	.9	f	t	.6	t	f	t	$.7 \times .6 = .42$
f	f	.1	f	f	.4	t	f	f	$.7 \times .4 = .28$
						f	t	t	$.9 \times .2 = .18$
						f	t	f	$.9 \times .8 = .72$
						f	f	t	$.1 \times .6 = .06$
						f	f	f	$.1 \times .4 = .04$

Figure 13.12 Illustrating pointwise multiplication: $\mathbf{f}(X,Y) \times \mathbf{g}(Y,Z) = \mathbf{h}(X,Y,Z)$.

```
function ELIMINATION-ASK($X$, $\mathbf{e}$, $bn$) returns a distribution over $X$
   inputs: $X$, the query variable
           $\mathbf{e}$, observed values for variables $\mathbf{E}$
           $bn$, a Bayesian network with variables $vars$

   $factors \leftarrow [\,]$
   for each $V$ in ORDER($vars$) do
      $factors \leftarrow$ [MAKE-FACTOR($V$, $\mathbf{e}$)] + $factors$
      if $V$ is a hidden variable then $factors \leftarrow$ SUM-OUT($V$, $factors$)
   return NORMALIZE(POINTWISE-PRODUCT($factors$))
```

Figure 13.13 The variable elimination algorithm for exact inference in Bayes nets.

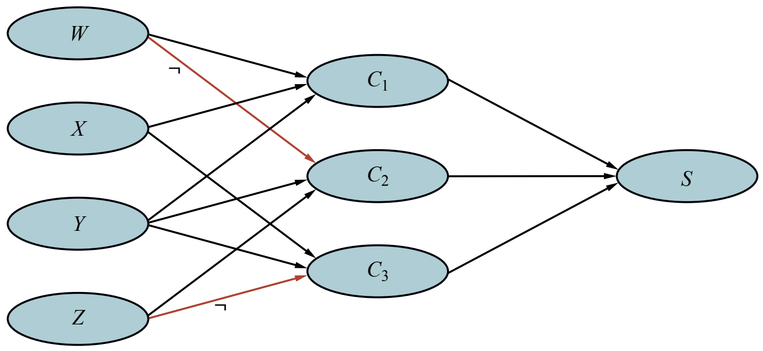
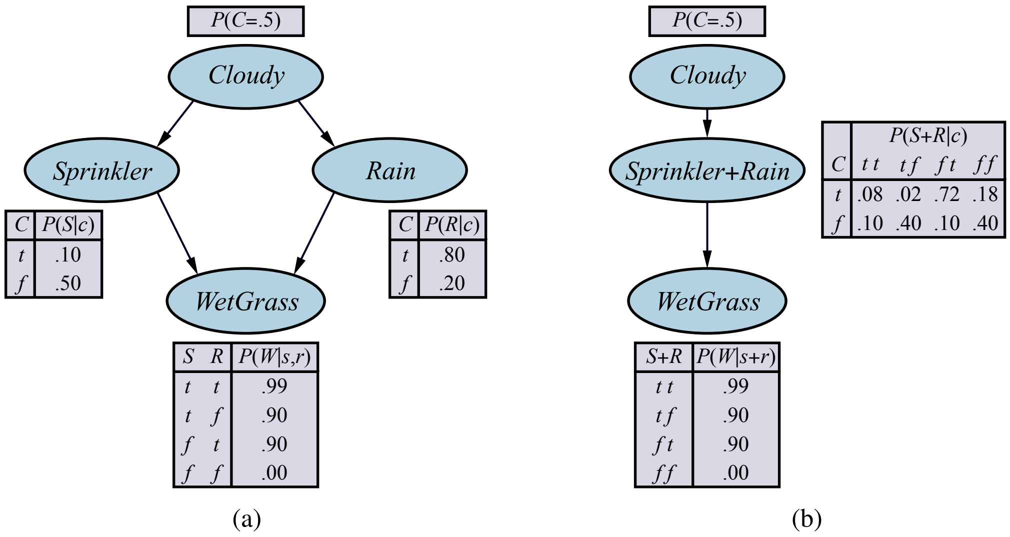


Figure 13.14 Bayes net encoding of the 3-CNF sentence

$(W \vee X \vee Y) \wedge (\neg W \vee Y \vee Z) \wedge (X \vee Y \vee \neg Z)$.



C	$P(S\|c)$
t	.10
f	.50

C	$P(R\|c)$
t	.80
f	.20

S	R	$P(W\|s,r)$
t	t	.99
t	f	.90
f	t	.90
f	f	.00

	$P(S{+}R\|c)$			
C	tt	tf	ft	ff
t	.08	.02	.72	.18
f	.10	.40	.10	.40

$S{+}R$	$P(W\|s{+}r)$
tt	.99
tf	.90
ft	.90
ff	.00

Figure 13.15 (a) A multiply connected network describing Mary's daily lawn routine: each morning, she checks the weather; if it's cloudy, she usually doesn't turn on the sprinkler; if the sprinkler is on, or if it rains during the day, the grass will be wet. Thus, *Cloudy* affects *WetGrass* via two different causal pathways. (b) A clustered equivalent of the multiply connected network.

```
function PRIOR-SAMPLE(bn) returns an event sampled from the prior specified by bn
  inputs: bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1,\ldots,X_n)$

  $\mathbf{x} \leftarrow$ an event with $n$ elements
  for each variable $X_i$ in $X_1,\ldots,X_n$ do
      $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$
```

Figure 13.16 A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.

```
function REJECTION-SAMPLING($X, \mathbf{e}, bn, N$) returns an estimate of $\mathbf{P}(X \mid \mathbf{e})$
  inputs: $X$, the query variable
          $\mathbf{e}$, observed values for variables $\mathbf{E}$
          $bn$, a Bayesian network
          $N$, the total number of samples to be generated
  local variables: $\mathbf{C}$, a vector of counts for each value of $X$, initially zero

  for $j$ = 1 to $N$ do
     $\mathbf{x} \leftarrow$ PRIOR-SAMPLE($bn$)
     if $\mathbf{x}$ is consistent with $\mathbf{e}$ then
        $\mathbf{C}[j] \leftarrow \mathbf{C}[j]+1$ where $x_j$ is the value of $X$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{C}$)
```

Figure 13.17 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

```
function LIKELIHOOD-WEIGHTING($X, \mathbf{e}, bn, N$) returns an estimate of $\mathbf{P}(X \mid \mathbf{e})$
  inputs: $X$, the query variable
          $\mathbf{e}$, observed values for variables $\mathbf{E}$
          $bn$, a Bayesian network specifying joint distribution $\mathbf{P}(X_1, \ldots, X_n)$
          $N$, the total number of samples to be generated
  local variables: $\mathbf{W}$, a vector of weighted counts for each value of $X$, initially zero

  for $j$ = 1 to $N$ do
     $\mathbf{x}, w \leftarrow$ WEIGHTED-SAMPLE($bn, \mathbf{e}$)
     $\mathbf{W}[j] \leftarrow \mathbf{W}[j] + w$ where $x_j$ is the value of $X$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{W}$)

function WEIGHTED-SAMPLE($bn, \mathbf{e}$) returns an event and a weight

  $w \leftarrow 1$; $\mathbf{x} \leftarrow$ an event with $n$ elements, with values fixed from $\mathbf{e}$
  for $i$ = 1 to $n$ do
     if $X_i$ is an evidence variable with value $x_{ij}$ in $\mathbf{e}$
        then $w \leftarrow w \times P(X_i = x_{ij} \mid parents(X_i))$
        else $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return $\mathbf{x}$, $w$
```

Figure 13.18 The likelihood-weighting algorithm for inference in Bayesian networks. In WEIGHTED-SAMPLE, each nonevidence variable is sampled according to the conditional distribution given the values already sampled for the variable's parents, while a weight is accumulated based on the likelihood for each evidence variable.

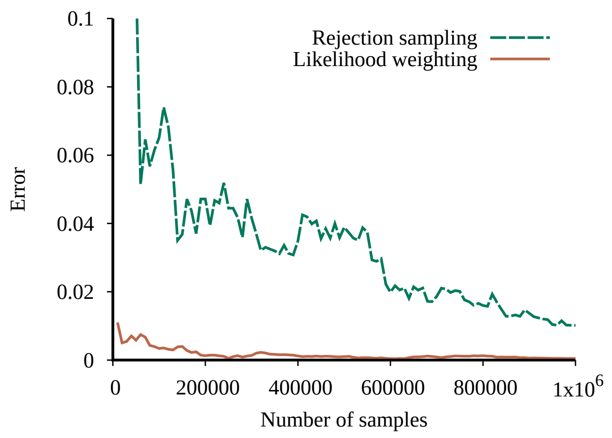


Figure 13.19 Performance of rejection sampling and likelihood weighting on the insurance network. The x-axis shows the number of samples generated and the y-axis shows the maximum absolute error in any of the probability values for a query on *PropertyCost*.

```
function GIBBS-ASK($X$, $\mathbf{e}$, $bn$, $N$) returns an estimate of $\mathbf{P}(X \mid \mathbf{e})$
  local variables: $\mathbf{C}$, a vector of counts for each value of $X$, initially zero
                   $\mathbf{Z}$, the nonevidence variables in $bn$
                   $\mathbf{x}$, the current state of the network, initialized from $\mathbf{e}$

  initialize $\mathbf{x}$ with random values for the variables in $\mathbf{Z}$
  for $k = 1$ to $N$ do
      choose any variable $Z_i$ from $\mathbf{Z}$ according to any distribution $\rho(i)$
      set the value of $Z_i$ in $\mathbf{x}$ by sampling from $\mathbf{P}(Z_i \mid mb(Z_i))$
      $\mathbf{C}[j] \leftarrow \mathbf{C}[j] + 1$ where $x_j$ is the value of $X$ in $\mathbf{x}$
  return NORMALIZE($\mathbf{C}$)
```

Figure 13.20 The Gibbs sampling algorithm for approximate inference in Bayes nets; this version chooses variables at random, but cycling through the variables but also works.

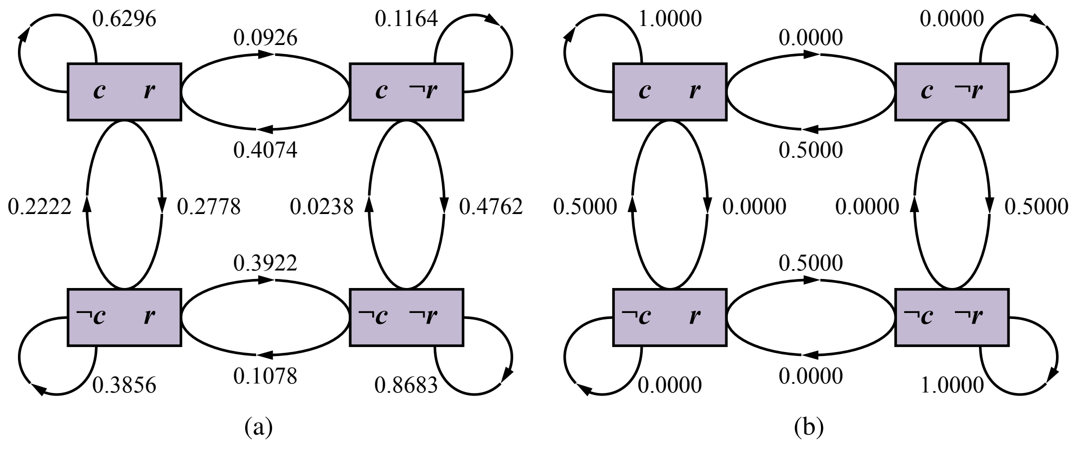


Figure 13.21 (a) The states and transition probabilities of the Markov chain for the query $\mathbf{P}(Rain \mid Sprinkler = true, WetGrass = true)$. Note the self-loops: the state stays the same when *either* variable is chosen and then resamples the same value it already has. (b) The transition probabilities when the CPT for *Rain* constrains it to have the same value as *Cloudy*.

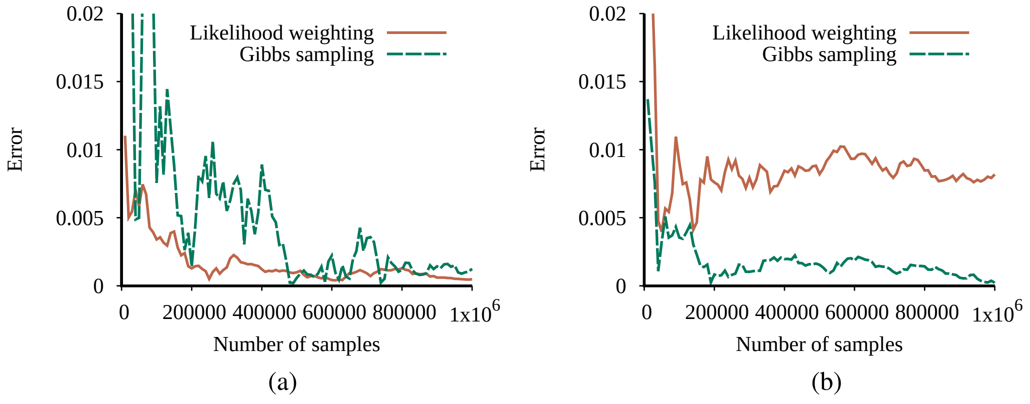


Figure 13.22 Performance of Gibbs sampling compared to likelihood weighting on the car insurance network: (a) for the standard query on *PropertyCost*, and (b) for the case where the output variables are observed and *Age* is the query variable.

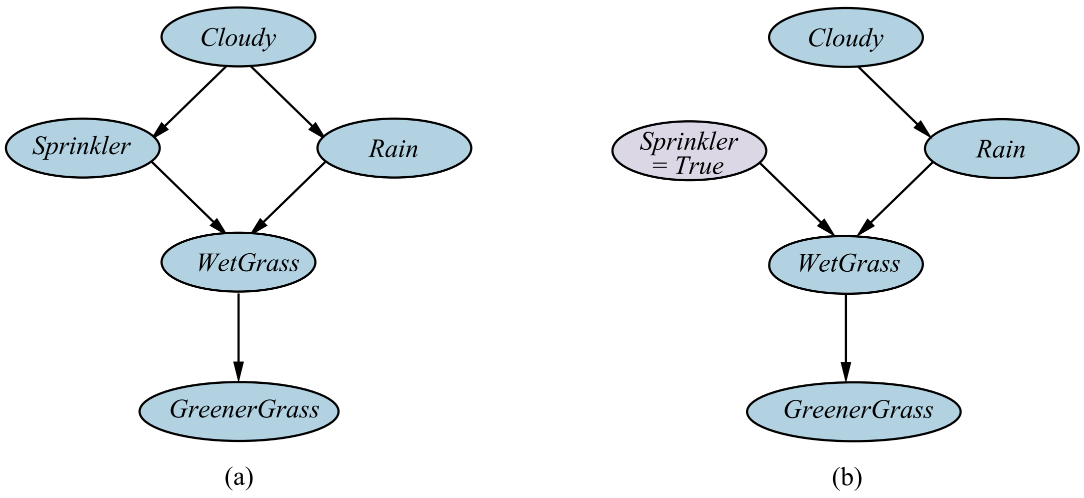


Figure 13.23 (a) A causal Bayesian network representing cause–effect relations among five variables. (b) The network after performing the action "turn *Sprinkler* on."

CHAPTER 14

PROBABILISTIC REASONING OVER TIME

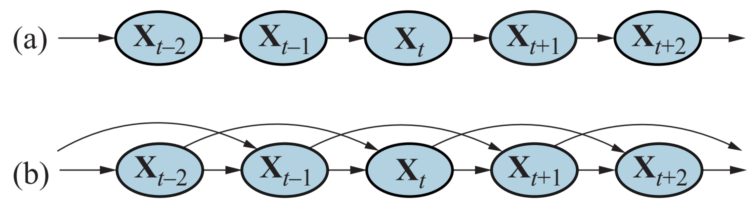


Figure 14.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

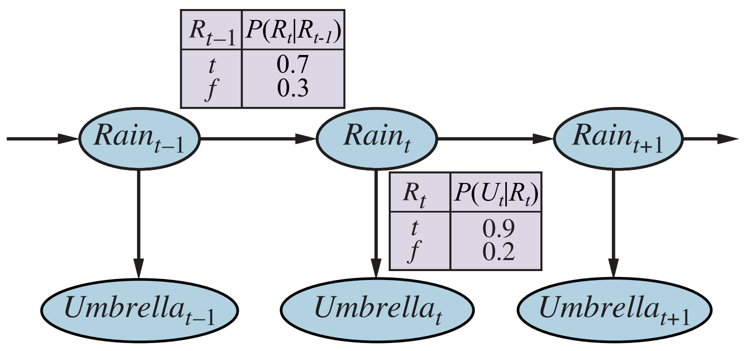


Figure 14.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $\mathbf{P}(Rain_t \,|\, Rain_{t-1})$ and the sensor model is $\mathbf{P}(Umbrella_t \,|\, Rain_t)$.

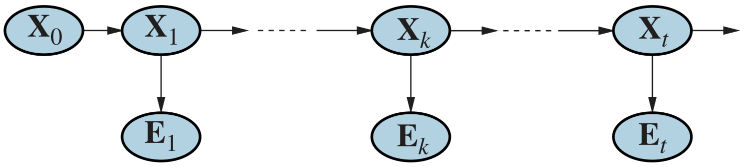


Figure 14.3 Smoothing computes $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.

```
function FORWARD-BACKWARD($\mathbf{ev}$, $prior$) returns a vector of probability distributions
  inputs: $\mathbf{ev}$, a vector of evidence values for steps $1,\ldots,t$
          $prior$, the prior distribution on the initial state, $\mathbf{P}(\mathbf{X}_0)$
  local variables: $\mathbf{fv}$, a vector of forward messages for steps $0,\ldots,t$
                   $\mathbf{b}$, a representation of the backward message, initially all 1s
                   $\mathbf{sv}$, a vector of smoothed estimates for steps $1,\ldots,t$

  $\mathbf{fv}[0] \leftarrow prior$
  for $i = 1$ to $t$ do
      $\mathbf{fv}[i] \leftarrow$ FORWARD($\mathbf{fv}[i-1], \mathbf{ev}[i]$)
  for $i = t$ down to 1 do
      $\mathbf{sv}[i] \leftarrow$ NORMALIZE($\mathbf{fv}[i] \times \mathbf{b}$)
      $\mathbf{b} \leftarrow$ BACKWARD($\mathbf{b}, \mathbf{ev}[i]$)
  return $\mathbf{sv}$
```

Figure 14.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (14.5) and (14.9), respectively.

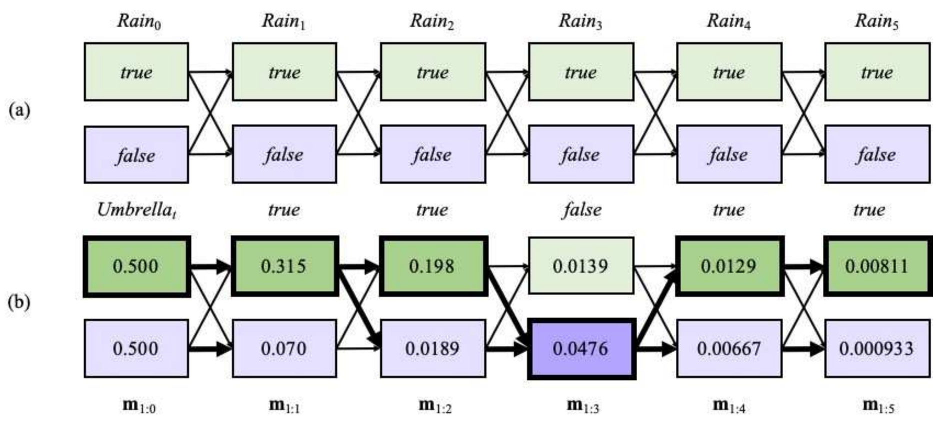
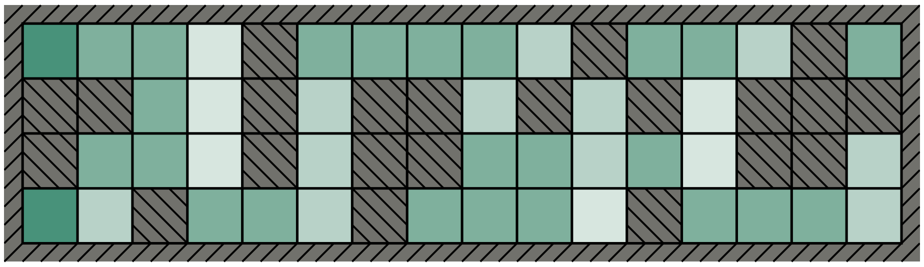


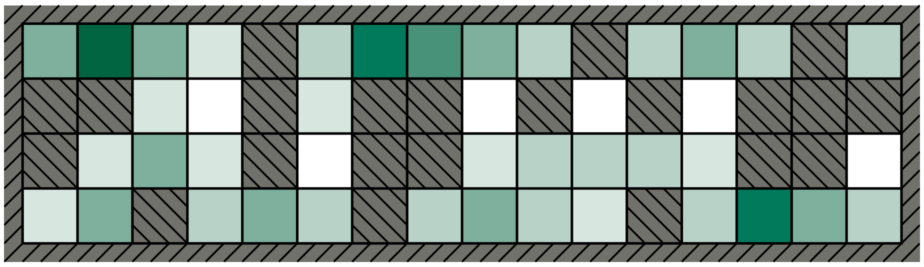
Figure 14.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$, where the evidence starts at time 1. For each t, we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time t. Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence, shown by the bold outlines and darker shading.

```
function FIXED-LAG-SMOOTHING($e_t$, hmm, $d$) returns a distribution over $\mathbf{X}_{t-d}$
   inputs: $e_t$, the current evidence for time step $t$
           hmm, a hidden Markov model with $S \times S$ transition matrix $\mathbf{T}$
           $d$, the length of the lag for smoothing
   persistent: $t$, the current time, initially 1
               $\mathbf{f}$, the forward message $\mathbf{P}(X_t \mid e_{1:t})$, initially hmm.PRIOR
               $\mathbf{B}$, the $d$-step backward transformation matrix, initially the identity matrix
               $e_{t-d:t}$, double-ended list of evidence from $t-d$ to $t$, initially empty
   local variables: $\mathbf{O}_{t-d}, \mathbf{O}_t$, diagonal matrices containing the sensor model information

   add $e_t$ to the end of $e_{t-d:t}$
   $\mathbf{O}_t \leftarrow$ diagonal matrix containing $\mathbf{P}(e_t \mid X_t)$
   if $t > d$ then
      $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}, e_{t-d}$)
      remove $e_{t-d-1}$ from the beginning of $e_{t-d:t}$
      $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d} \mid X_{t-d})$
      $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
   else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
   $t \leftarrow t+1$
   if $t > d+1$ then return NORMALIZE($\mathbf{f} \times \mathbf{B1}$) else return null
```

Figure 14.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step. Notice that the final output NORMALIZE($\mathbf{f} \times \mathbf{B1}$) is just $\alpha \mathbf{f} \times \mathbf{b}$, by Equation (14.14).

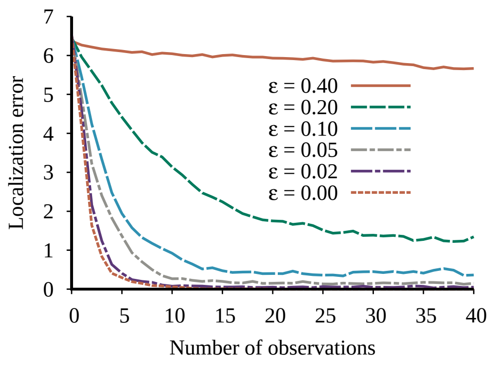


(a) Posterior distribution over robot location after $E_1 = 1011$

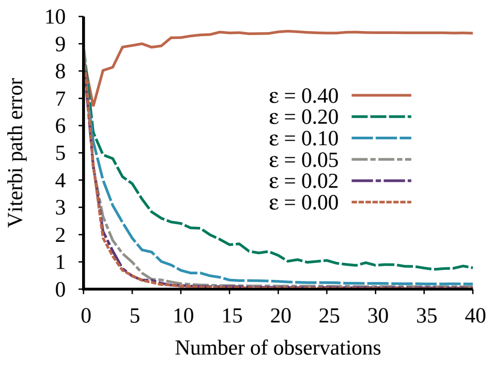


(b) Posterior distribution over robot location after $E_1 = 1011$, $E_2 = 1010$

Figure 14.7 Posterior distribution over robot location: (a) after one observation $E_1 = 1011$ (i.e., obstacles to the north, south, and west); (b) after a random move to an adjacent location and a second observation $E_2 = 1010$ (i.e., obstacles to the north and south). The darkness of each square corresponds to the probability that the robot is at that location. The sensor error rate for each bit is $\epsilon = 0.2$.



(a)



(b)

Figure 14.8 Performance of HMM localization as a function of the length of the observation sequence for various different values of the sensor error probability ϵ; data averaged over 400 runs. (a) The localization error, defined as the Manhattan distance from the true location. (b) The Viterbi path error, defined as the average Manhattan distance of states on the Viterbi path from corresponding states on the true path.

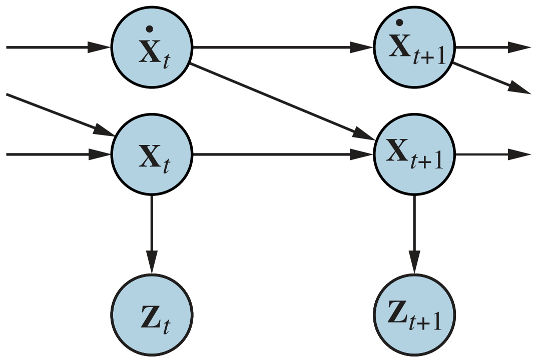


Figure 14.9 Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{\mathbf{X}}_t$, and position measurement $\mathbf{Z}_t$.

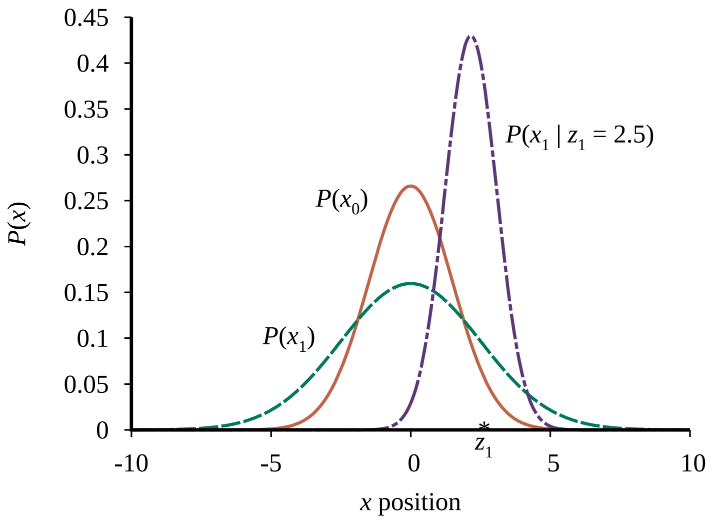


Figure 14.10 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.5$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$ (marked on the x-axis). Notice how the prediction $P(x_1)$ is flattened out, relative to $P(x_0)$, by the transition noise. Notice also that the mean of the posterior distribution $P(x_1 \mid z_1)$ is slightly to the left of the observation z_1 because the mean is a weighted average of the prediction and the observation.

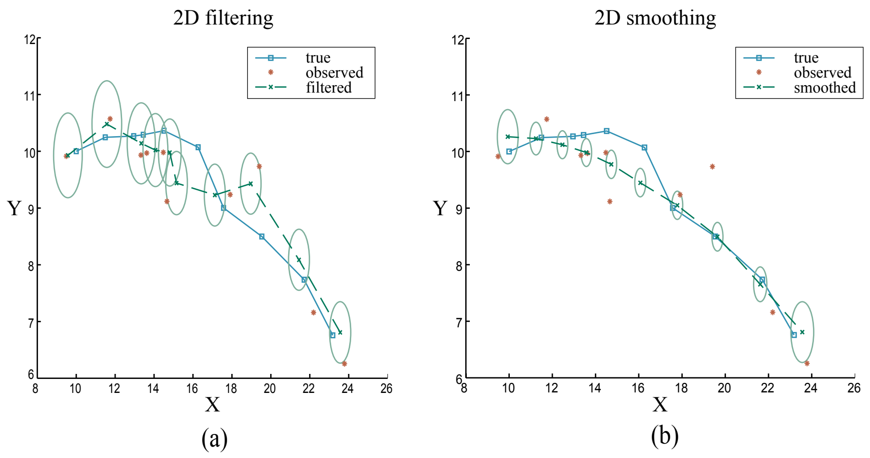


Figure 14.11 (a) Results of Kalman filtering for an object moving on the X–Y plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing for the same observation sequence.

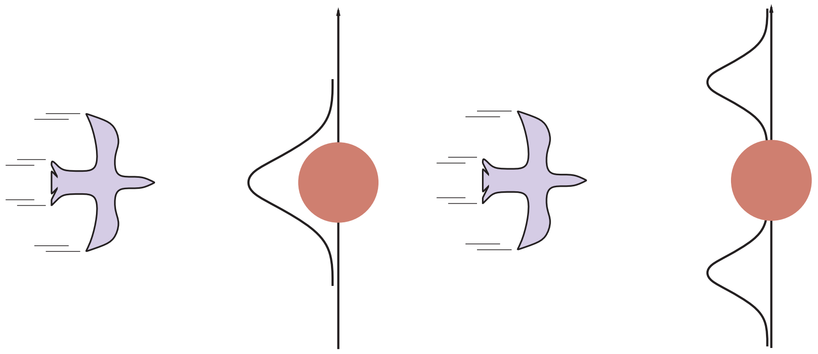


Figure 14.12 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird's evasive action, predicting that it will fly to one side or the other.

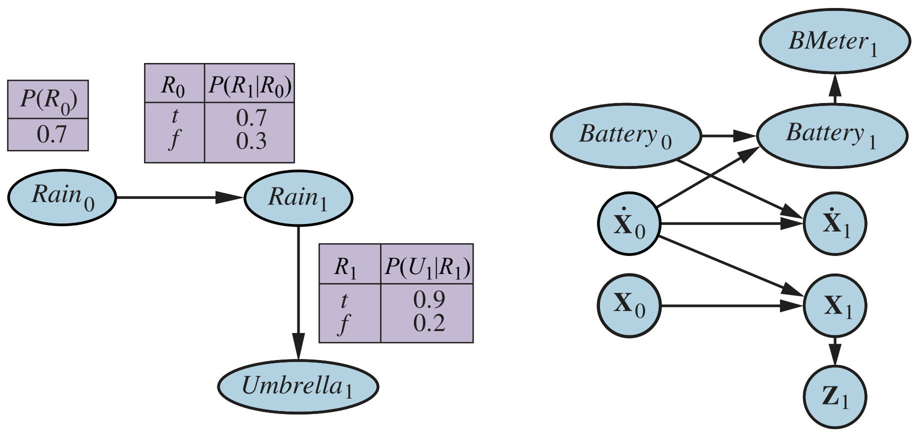


Figure 14.13 Left: Specification of the prior, transition model, and sensor model for the umbrella DBN. Subsequent slices are copies of slice 1. Right: A simple DBN for robot motion in the X–Y plane.

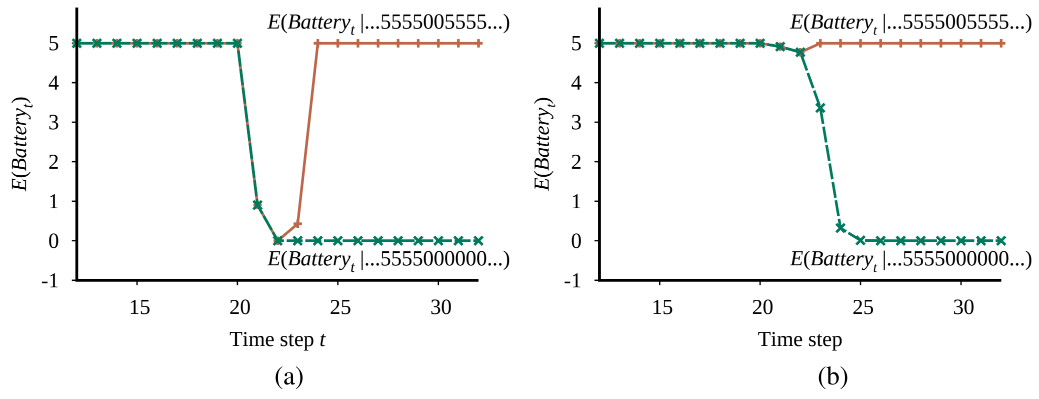


Figure 14.14 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t = 21$ and $t = 22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t = 21$ onwards. (b) The same experiment run with the transient failure model. The transient failure is handled well, but the persistent failure results in excessive pessimism about the battery charge.

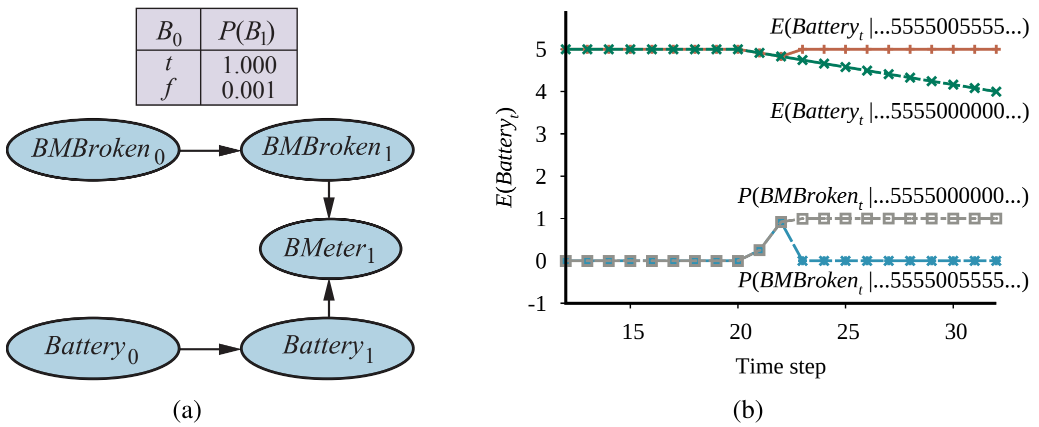
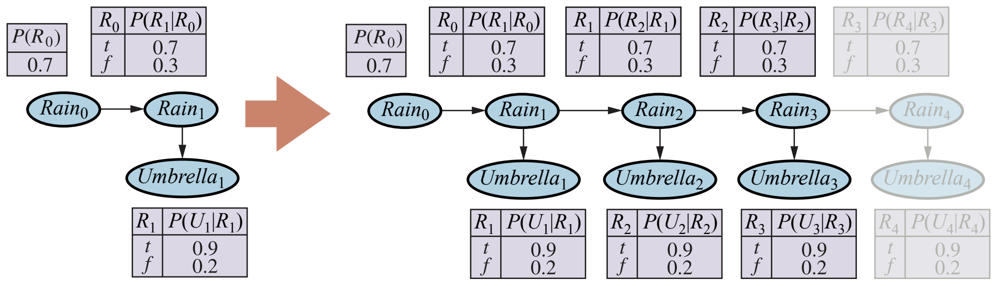


Figure 14.15 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability trajectories for *BMBroken* given the two observation sequences.



$P(R_0)$ 0.7
R_0 | $P(R_1|R_0)$: t 0.7, f 0.3
$Rain_0$
$Rain_1$
$Umbrella_1$
R_1 | $P(U_1|R_1)$: t 0.9, f 0.2
R_1 | $P(R_2|R_1)$: t 0.7, f 0.3
R_2 | $P(R_3|R_2)$: t 0.7, f 0.3
R_3 | $P(R_4|R_3)$: t 0.7, f 0.3
$Rain_2$
$Rain_3$
$Rain_4$
$Umbrella_2$
$Umbrella_3$
$Umbrella_4$
R_2 | $P(U_2|R_2)$: t 0.9, f 0.2
R_3 | $P(U_3|R_3)$: t 0.9, f 0.2
R_4 | $P(U_4|R_4)$: t 0.9, f 0.2

Figure 14.16 Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence $Umbrella_{1:3}$. Further slices have no effect on inferences within the observation period.

```
function PARTICLE-FILTERING($\mathbf{e}, N, dbn$) returns a set of samples for the next time step
  inputs: $\mathbf{e}$, the new incoming evidence
          $N$, the number of samples to be maintained
          $dbn$, a DBN defined by $\mathbf{P}(\mathbf{X}_0)$, $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0)$, and $\mathbf{P}(\mathbf{E}_1 \mid \mathbf{X}_1)$
  persistent: $S$, a vector of samples of size $N$, initially generated from $\mathbf{P}(\mathbf{X}_0)$
  local variables: $W$, a vector of weights of size $N$

  for $i$ = 1 to $N$ do
      $S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0 = S[i])$          // step 1
      $W[i] \leftarrow \mathbf{P}(\mathbf{e} \mid \mathbf{X}_1 = S[i])$                          // step 2
  $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT($N, S, W$)                    // step 3
  return $S$
```

Figure 14.17 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.

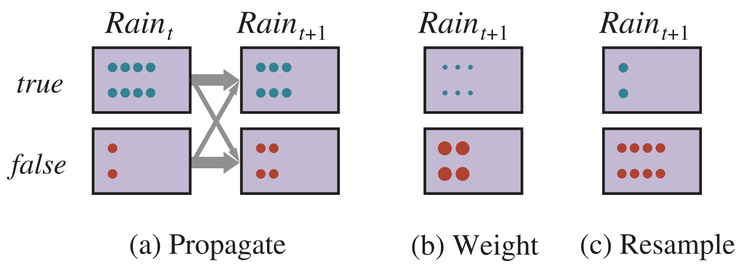


Figure 14.18 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t, 8 samples indicate *rain* and 2 indicate $\neg$*rain*. Each is propagated forward by sampling the next state through the transition model. At time $t+1$, 6 samples indicate *rain* and 4 indicate $\neg$*rain*. (b) $\neg$*umbrella* is observed at $t+1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 2 samples that indicate *rain* and 8 that indicate $\neg$*rain*.

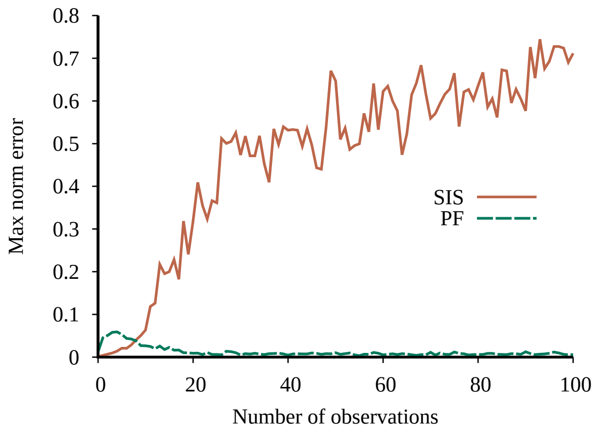


Figure 14.19 Max norm error in the grid-world location estimate (compared to exact inference) for likelihood weighting (sequential importance sampling) with 100,000 samples and particle filtering with 1,000 samples; data averaged over 50 runs.

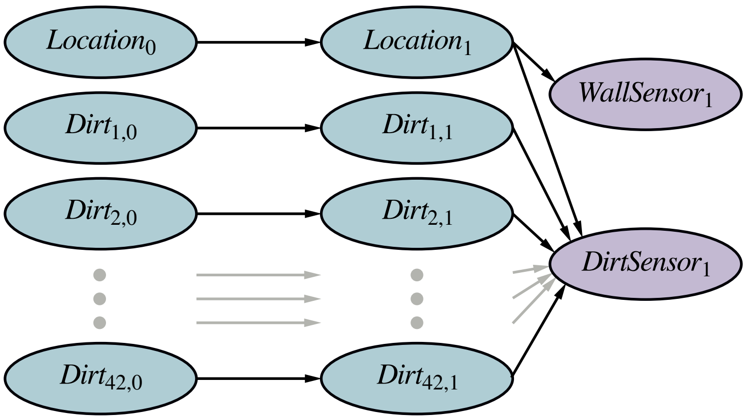


Figure 14.20 A dynamic Bayes net for simultaneous localization and mapping in the stochastic-dirt vacuum world. Dirty squares persist with probability p, and clean squares become dirty with probability $1-p$. The local dirt sensor is 90% accurate, for the square in which the robot is currently located.

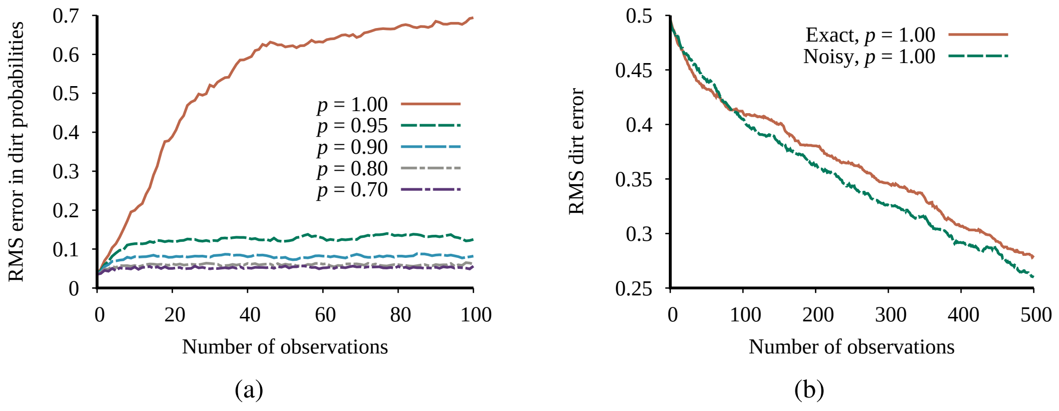


Figure 14.21 (a) Performance of the standard particle filtering algorithm with 1,000 particles, showing RMS error in marginal dirt probabilities compared to exact inference for different values of the dirt persistence p. (b) Performance of Rao-Blackwellized particle filtering (100 particles) compared to ground truth, for both exact location sensing and noisy wall sensing and with deterministic dirt. Data averaged over 20 runs.

CHAPTER 15
MAKING SIMPLE DECISIONS

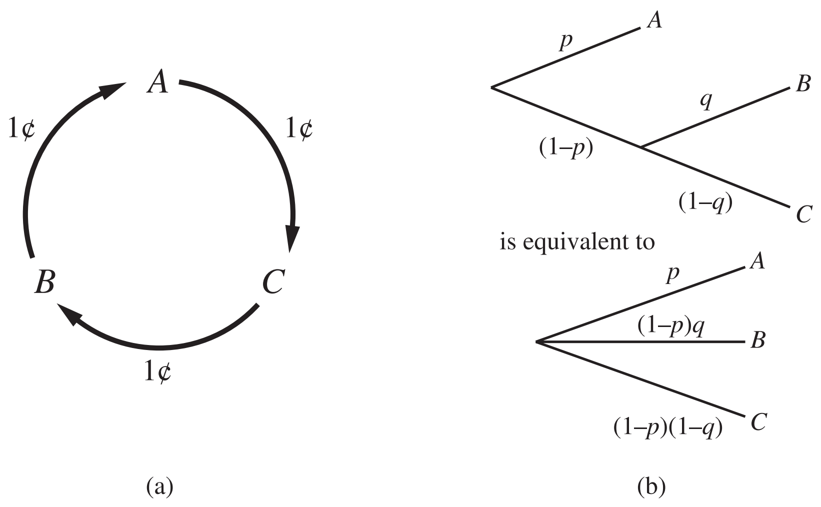


Figure 15.1 (a) Nontransitive preferences $A \succ B \succ C \succ A$ can result in irrational behavior: a cycle of exchanges each costing one cent. (b) The decomposability axiom.

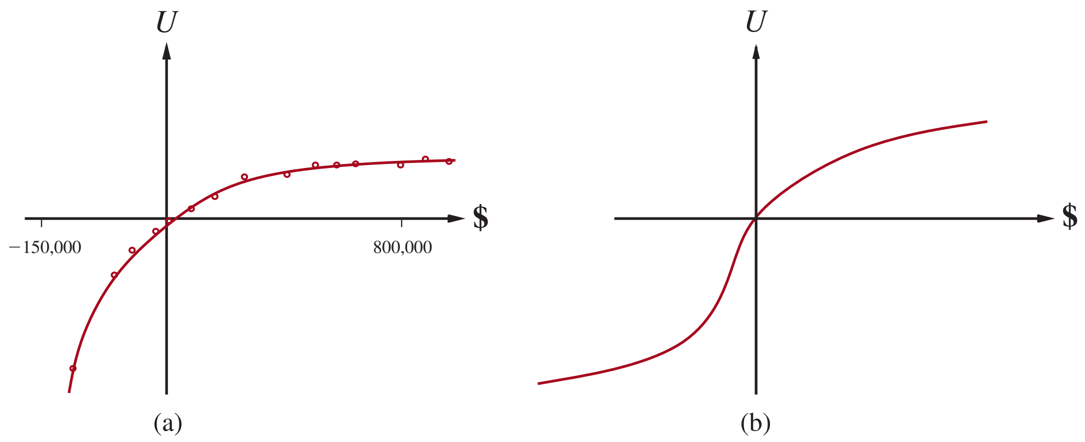


Figure 15.2 The utility of money. (a) Empirical data for Mr. Beard over a limited range. (b) A typical curve for the full range.

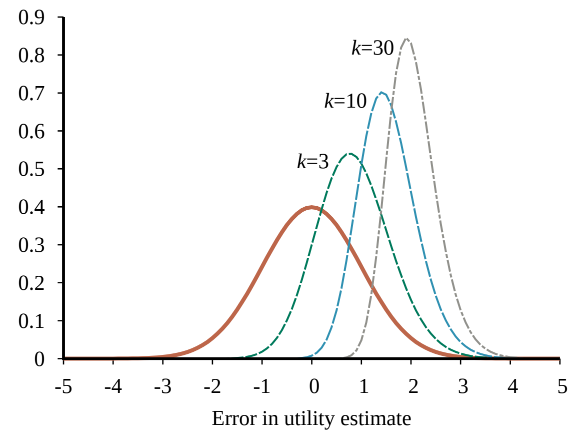


Figure 15.3 Unjustified optimism caused by choosing the best of k options: we assume that each option has a true utility of 0 but a utility estimate that is distributed according to a unit normal (brown curve). The other curves show the distributions of the maximum of k estimates for $k = 3$, 10, and 30.

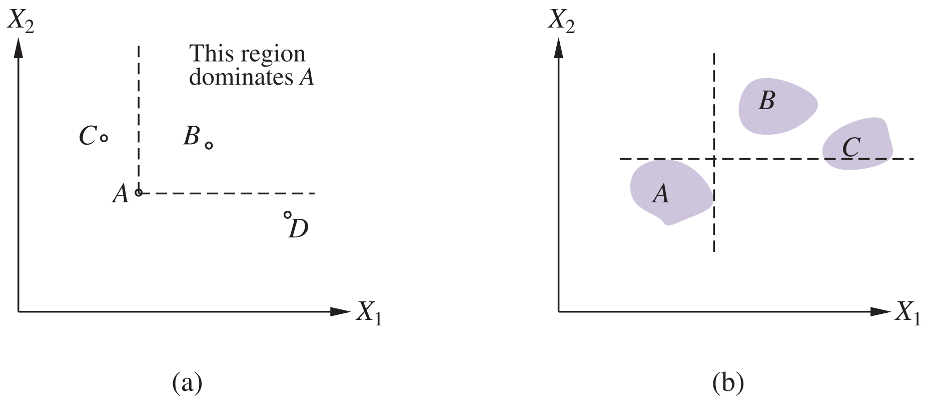


Figure 15.4 Strict dominance. (a) Deterministic: Option A is strictly dominated by B but not by C or D. (b) Uncertain: A is strictly dominated by B but not by C.

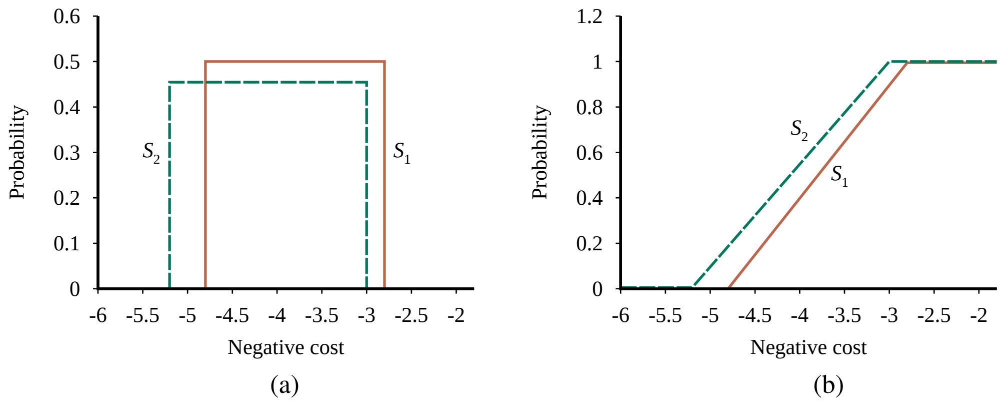


Figure 15.5 Stochastic dominance. (a) S_1 stochastically dominates S_2 on frugality (negative cost). (b) Cumulative distributions for the frugality of S_1 and S_2.

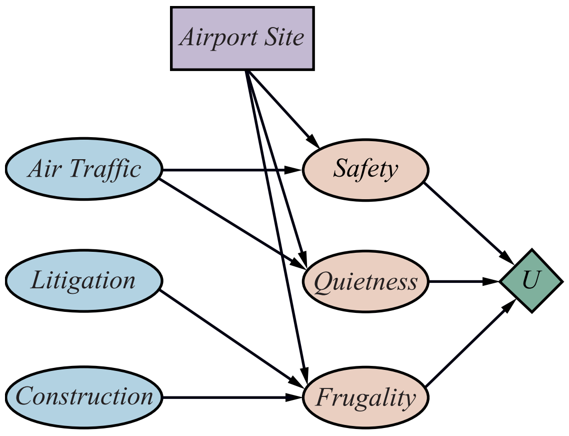


Figure 15.6 A decision network for the airport-siting problem.

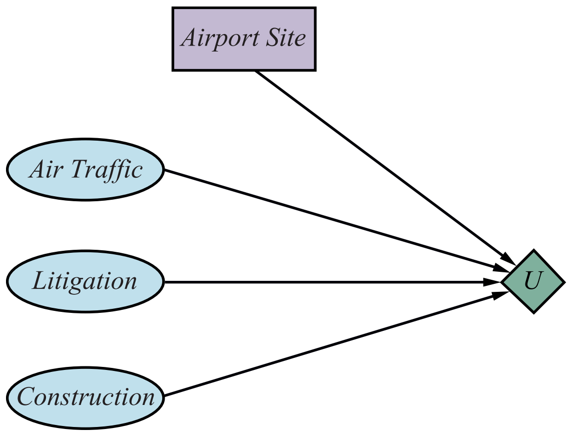


Figure 15.7 A simplified representation of the airport-siting problem. Chance nodes corresponding to outcome states have been factored out.

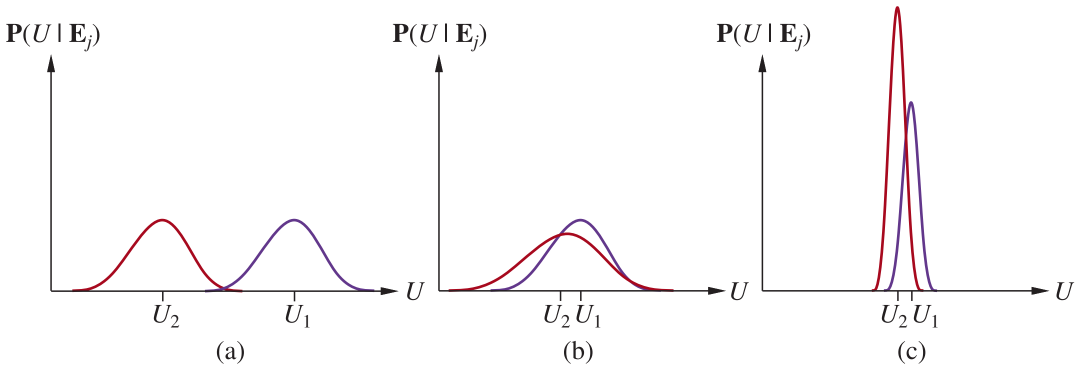
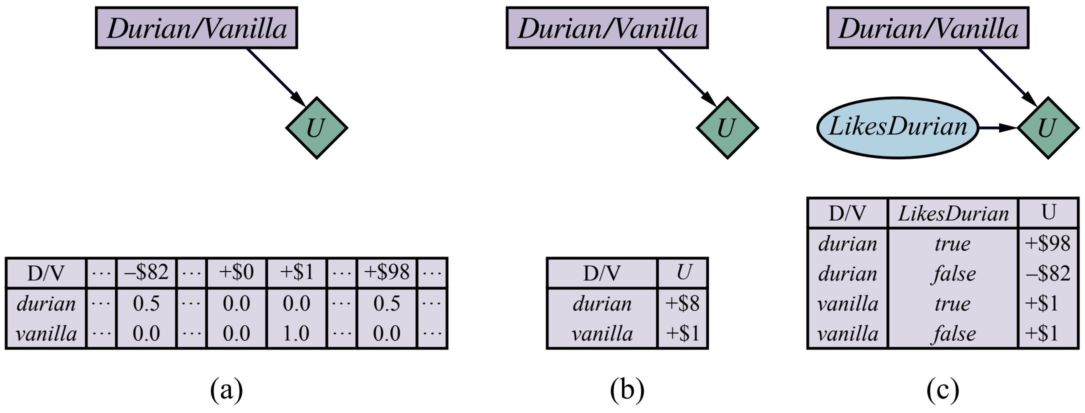


Figure 15.8 Three generic cases for the value of information. In (a), a_1 will almost certainly remain superior to a_2, so the information is not needed. In (b), the choice is unclear and the information is crucial. In (c), the choice is unclear, but because it makes little difference, the information is less valuable. (Note: The fact that U_2 has a high peak in (c) means that its expected value is known with higher certainty than U_1.)

```
function INFORMATION-GATHERING-AGENT(percept) returns an action
  persistent: D, a decision network

  integrate percept into D
  j ← the value that maximizes $VPI(E_j)$ / $C(E_j)$
  if $VPI(E_j) > C(E_j)$
      then return $Request(E_j)$
  else return the best action from D
```

Figure 15.9 Design of a simple, myopic information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.



D/V	···	–\$82	···	+\$0	+\$1	···	+\$98	···
durian	···	0.5	···	0.0	0.0	···	0.5	···
vanilla	···	0.0	···	0.0	1.0	···	0.0	···

D/V	*U*
durian	+\$8
vanilla	+\$1

D/V	*LikesDurian*	U
durian	*true*	+\$98
durian	*false*	–\$82
vanilla	*true*	+\$1
vanilla	*false*	+\$1

Figure 15.10 (a) A decision network for the ice cream choice with an uncertain utility function. (b) The network with the expected utility of each action. (c) Moving the uncertainty from the utility function into a new random variable.

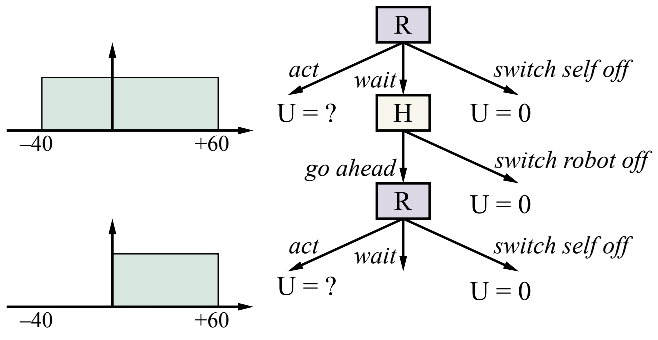
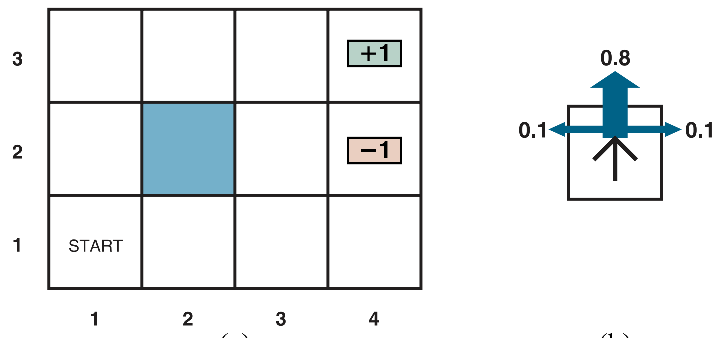


Figure 15.11 The off-switch game. R, the robot, can choose to act now, with a highly uncertain payoff; to switch itself off; or to defer to H, the human. H can switch R off or let it go ahead. R now has the same choice again. Acting still has an uncertain payoff, but now R knows the payoff is nonnegative.

CHAPTER 16

MAKING COMPLEX DECISIONS



(a) (b)

Figure 16.1 (a) A simple, stochastic 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. Transitions into the two terminal states have reward +1 and –1, respectively, and all other transitions have a reward of –0.04.

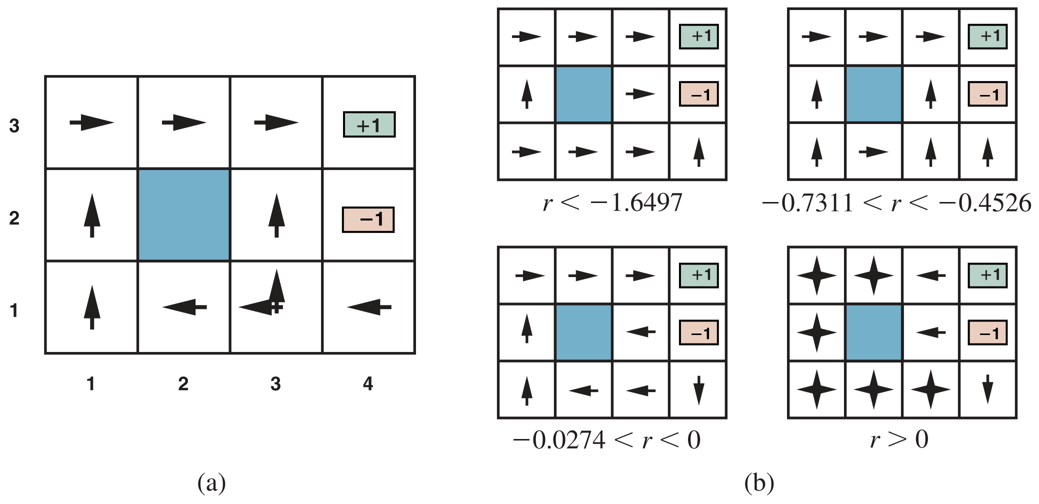


Figure 16.2 (a) The optimal policies for the stochastic environment with $r = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. (b) Optimal policies for four different ranges of r.

3	0.8516	0.9078	0.9578	+1
2	0.8016		0.7003	−1
1	0.7453	0.6953	0.6514	0.4279
	1	2	3	4

Figure 16.3 The utilities of the states in the 4×3 world with $\gamma = 1$ and $r = -0.04$ for transitions to nonterminal states.

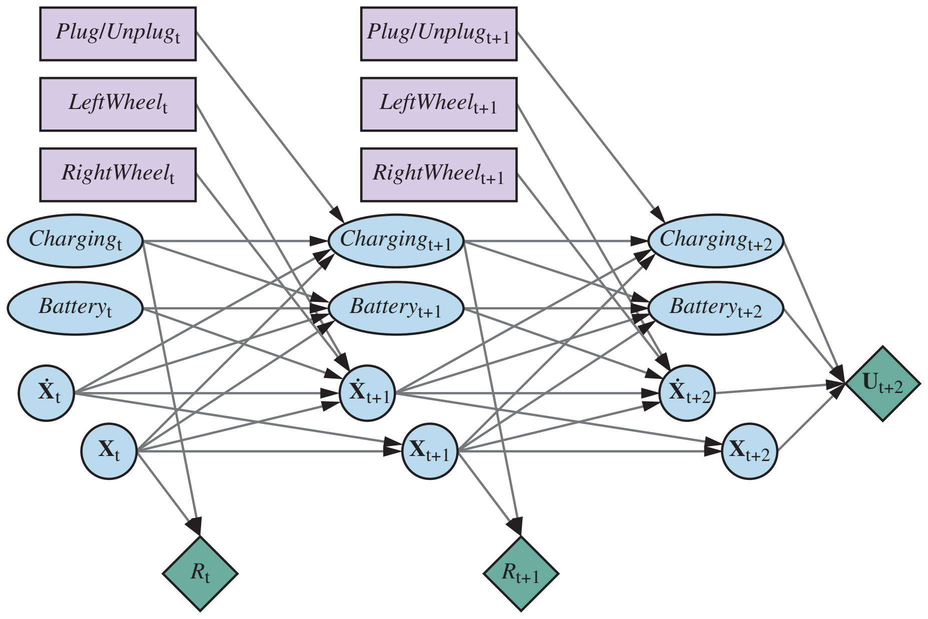
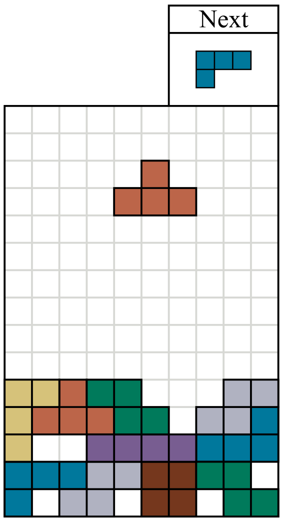
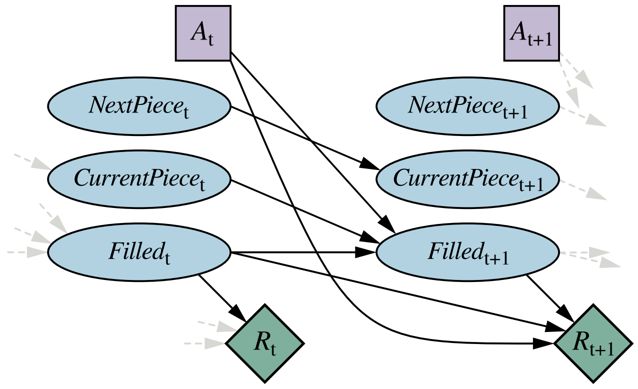


Figure 16.4 A dynamic decision network for a mobile robot with state variables for battery level, charging status, location, and velocity, and action variables for the left and right wheel motors and for charging.



(a)



(b)

Figure 16.5 (a) The game of Tetris. The T-shaped piece at the top center can be dropped in any orientation and in any horizontal position. If a row is completed, that row disappears and the rows above it move down, and the agent receives one point. The next piece (here, the L-shaped piece at top right) becomes the current piece, and a new next piece appears, chosen at random from the seven piece types. The game ends if the board fills up to the top. (b) The DDN for the Tetris MDP.

```
function VALUE-ITERATION($mdp, \epsilon$) returns a utility function
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s,a)$,
              rewards $R(s,a,s')$, discount $\gamma$
            $\epsilon$, the maximum error allowed in the utility of any state
  local variables: $U$, $U'$, vectors of utilities for states in $S$, initially zero
                   $\delta$, the maximum relative change in the utility of any state

  repeat
      $U \leftarrow U'$; $\delta \leftarrow 0$
      for each state $s$ in $S$ do
          $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE($mdp, s, a, U$)
          if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return $U$
```

Figure 16.6 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (16.12).

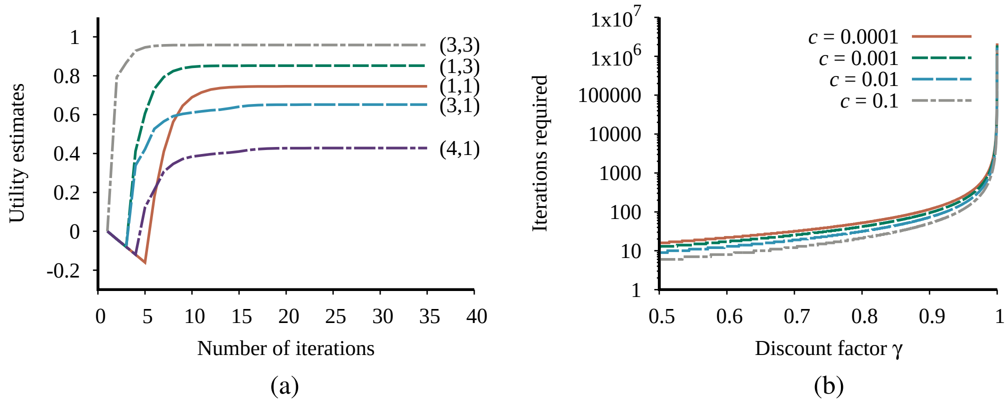


Figure 16.7 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

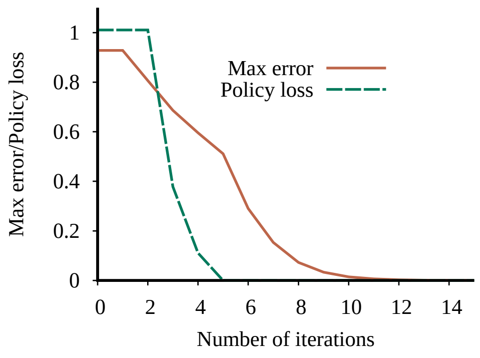


Figure 16.8 The maximum error $\|U_i - U\|$ of the utility estimates and the policy loss $\|U^{\pi_i} - U\|$, as a function of the number of iterations of value iteration on the 4×3 world.

```
function POLICY-ITERATION($mdp$) returns a policy
  inputs: $mdp$, an MDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: $U$, a vector of utilities for states in $S$, initially zero
                   $\pi$, a policy vector indexed by state, initially random

  repeat
      $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
      $unchanged? \leftarrow$ true
      for each state $s$ in $S$ do
          $a^* \leftarrow \operatorname{argmax}_{a \in A(s)}$ Q-VALUE($mdp, s, a, U$)
          if Q-VALUE($mdp, s, a^*, U$) $>$ Q-VALUE($mdp, s, \pi[s], U$) then
              $\pi[s] \leftarrow a^*$; $unchanged? \leftarrow$ false
  until $unchanged?$
  return $\pi$
```

Figure 16.9 The policy iteration algorithm for calculating an optimal policy.

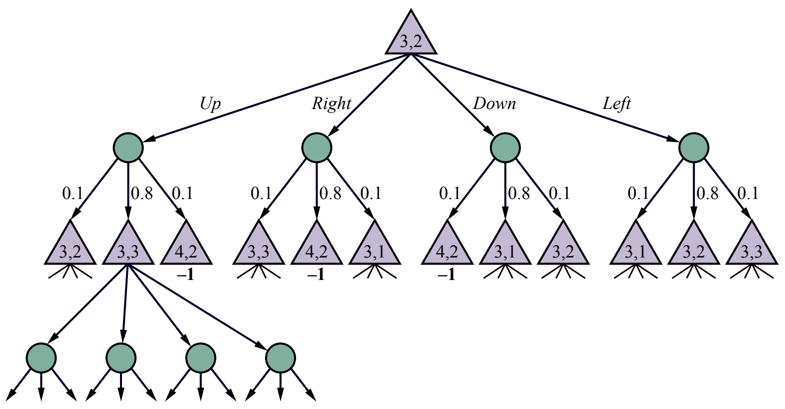


Figure 16.10 Part of an expectimax tree for the 4×3 MDP rooted at (3,2). The triangular nodes are max modes and the circular nodes are chance nodes.

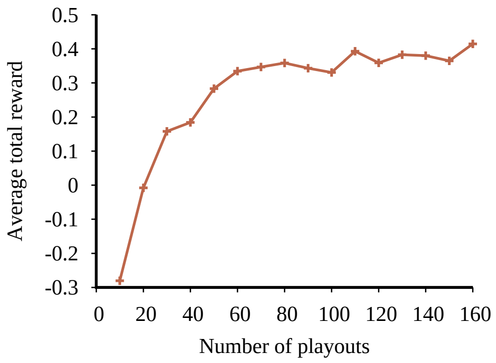


Figure 16.11 Performance of UCT as a function of the number of playouts per move for the 4×3 world using a random playout policy, averaged over 1000 runs per data point.

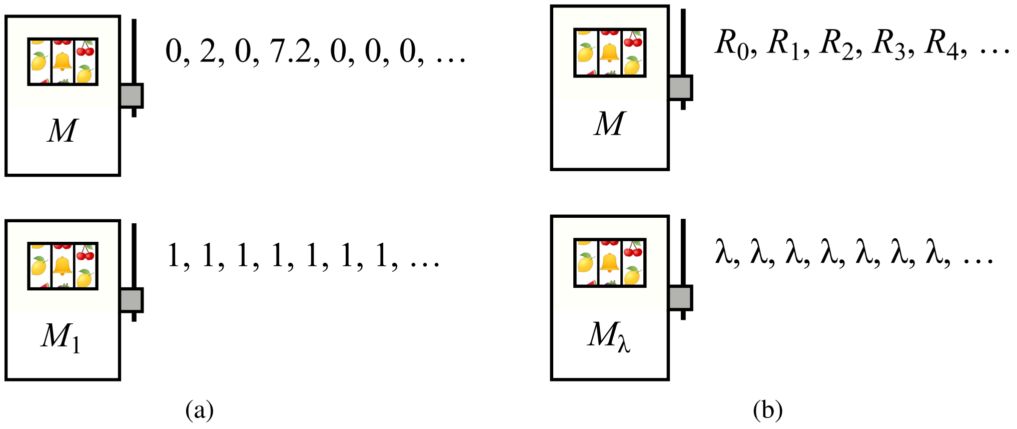


Figure 16.12 (a) A simple deterministic bandit problem with two arms. The arms can be pulled in any order, and each yields the sequence of rewards shown. (b) A more general case of the bandit in (a), where the first arm gives an arbitrary sequence of rewards and the second arm gives a fixed reward λ.

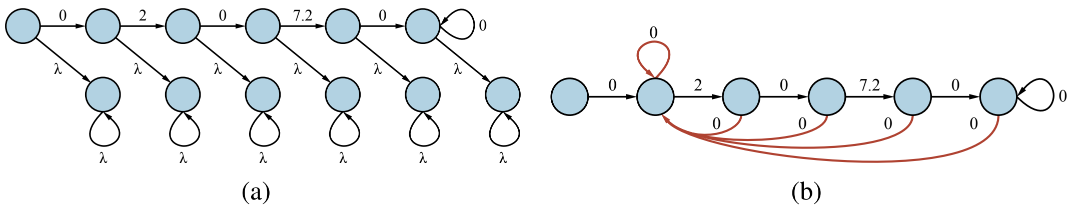
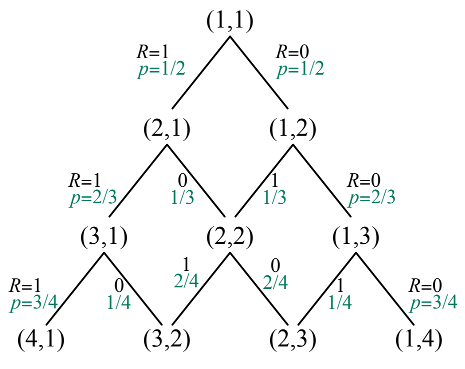
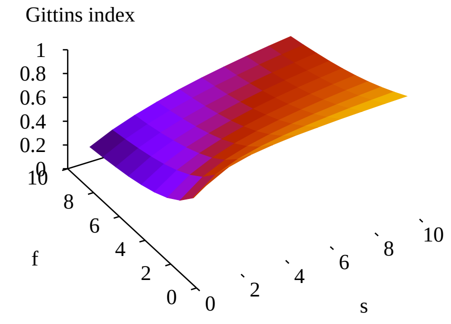


Figure 16.13 (a) The reward sequence $M = 0, 2, 0, 7.2, 0, 0, 0, \ldots$ augmented with a choice to switch permanently to a constant arm M_λ at each point. (b) An MDP whose optimal value is exactly equivalent to the optimal value for (a), at the point where the optimal policy is indifferent between M and M_λ.

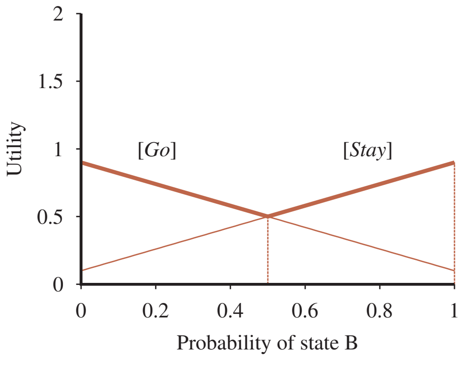


(a)

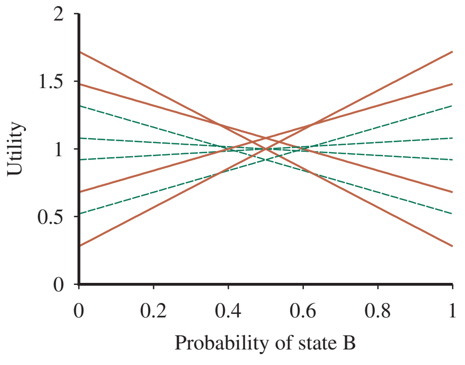


(b)

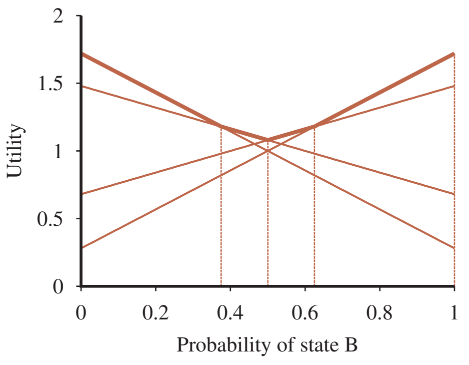
Figure 16.14 (a) States, rewards, and transition probabilities for the Bernoulli bandit. (b) Gittins indices for the states of the Bernoulli bandit process.



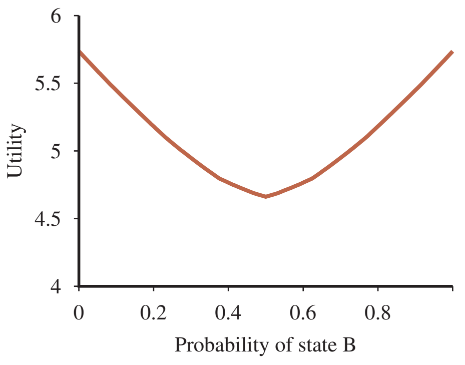
(a)



(b)



(c)



(d)

Figure 16.15 (a) Utility of two one-step plans as a function of the initial belief state $b(B)$ for the two-state world, with the corresponding utility function shown in bold. (b) Utilities for 8 distinct two-step plans. (c) Utilities for four undominated two-step plans. (d) Utility function for optimal eight-step plans.

```
function POMDP-VALUE-ITERATION($pomdp, \epsilon$) returns a utility function
   inputs: $pomdp$, a POMDP with states $S$, actions $A(s)$, transition model $P(s' \mid s, a)$,
               sensor model $P(e \mid s)$, rewards $R(s, a, s')$, discount $\gamma$
            $\epsilon$, the maximum error allowed in the utility of any state
   local variables: $U$, $U'$, sets of plans $p$ with associated utility vectors $\alpha_p$

   $U' \leftarrow$ a set containing all one-step plans $[a]$, with $\alpha_{[a]}(s) = \sum_{s'} P(s' \mid s, a)\, R(s, a, s')$
   repeat
       $U \leftarrow U'$
       $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept,
           a plan in $U$ with utility vectors computed according to Equation (16.18)
       $U' \leftarrow$ REMOVE-DOMINATED-PLANS($U'$)
   until MAX-DIFFERENCE($U, U'$) $\leq \epsilon(1-\gamma)/\gamma$
   return $U$
```

Figure 16.16 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

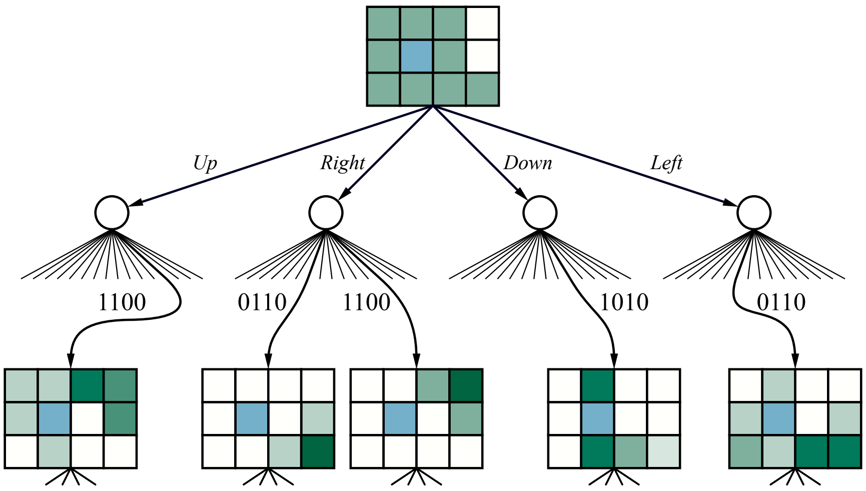


Figure 16.17 Part of an expectimax tree for the 4×3 POMDP with a uniform initial belief state. The belief states are depicted with shading proportional to the probability of being in each location.

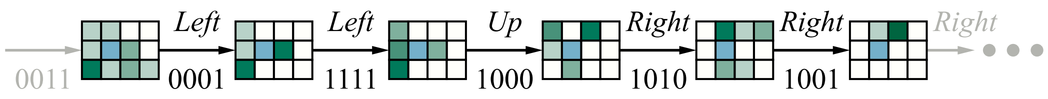


Figure 16.18 A sequence of percepts, belief states, and actions in the 4×3 POMDP with a wall-sensing error of $\epsilon = 0.2$. Notice how the early *Left* moves are safe—they are very unlikely to fall into $(4,2)$—and coerce the agent's location into a small number of possible locations. After moving *Up*, the agent thinks it is probably in $(3,3)$, but possibly in $(1,3)$. Fortunately, moving *Right* is a good idea in both cases, so it moves *Right*, finds out that it had been in $(1,3)$ and is now in $(2,3)$, and then continues moving *Right* and reaches the goal.

CHAPTER 17

MULTIAGENT DECISION MAKING

$Actors(A, B)$
$Init(At(A, LeftBaseline) \land At(B, RightNet) \land$
$\quad Approaching(Ball, RightBaseline) \land Partner(A, B) \land Partner(B, A)$
$Goal(Returned(Ball) \land (At(x, RightNet) \lor At(x, LeftNet))$
$Action(Hit(actor, Ball),$
$\quad$ PRECOND:$Approaching(Ball, loc) \land At(actor, loc)$
$\quad$ EFFECT:$Returned(Ball))$
$Action(Go(actor, to),$
$\quad$ PRECOND:$At(actor, loc) \land to \neq loc,$
$\quad$ EFFECT:$At(actor, to) \land \neg At(actor, loc))$

Figure 17.1 The doubles tennis problem. Two actors, A and B, are playing together and can be in one of four locations: *LeftBaseline*, *RightBaseline*, *LeftNet*, and *RightNet*. The ball can be returned only if a player is in the right place. The *NoOp* action is a dummy, which has no effect. Note that each action must include the actor as an argument.

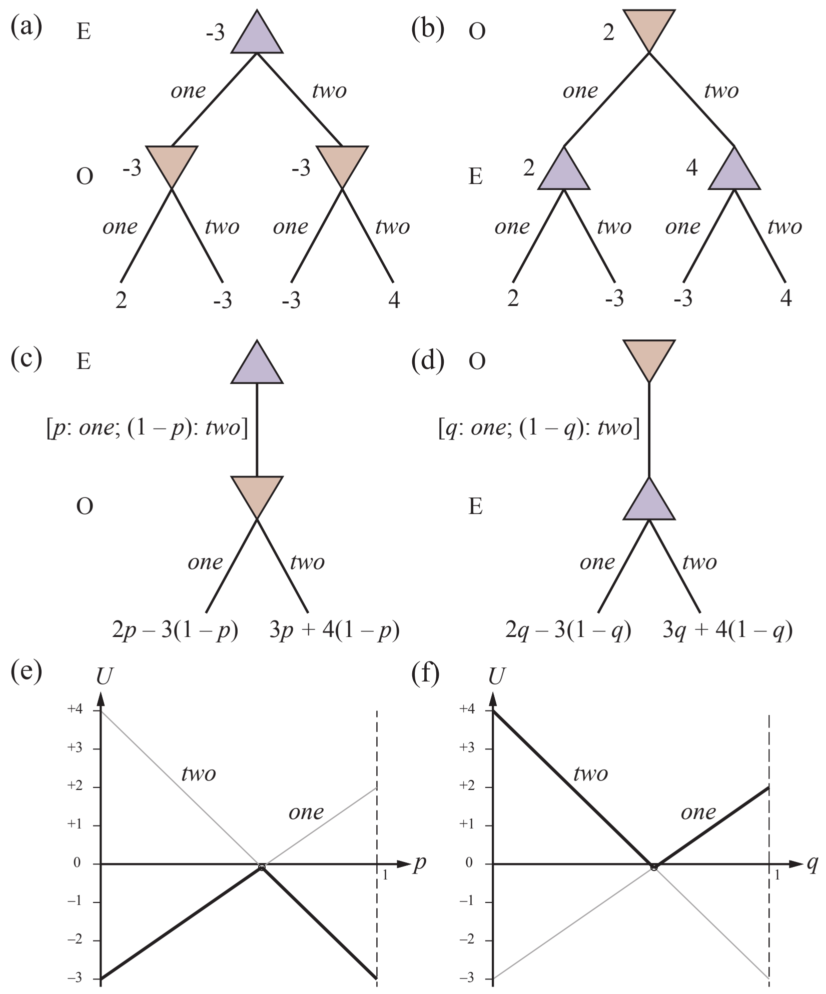


Figure 17.2 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Parameterized game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the “better” of the two actions, so the value of the first player’s mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

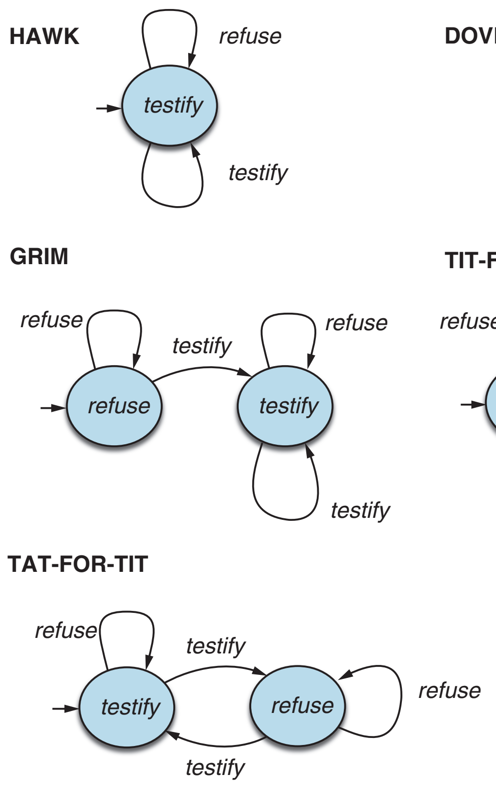


Figure 17.3 Some common, colorfully named finite-state machine strategies for the infinitely repeated prisoner's dilemma.

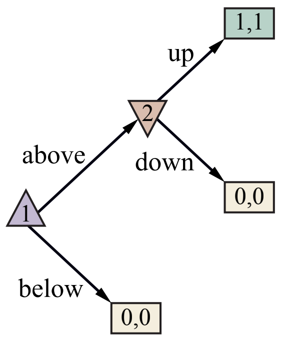


Figure 17.4 An extensive-form game with a counterintuitive Nash equilibrium.

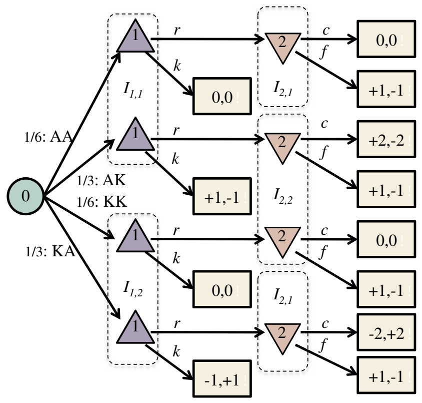


Figure 17.5 Extensive form of a simplified version of poker with two players and only four cards. The moves are r (raise), f (fold), c (call), and k (check).

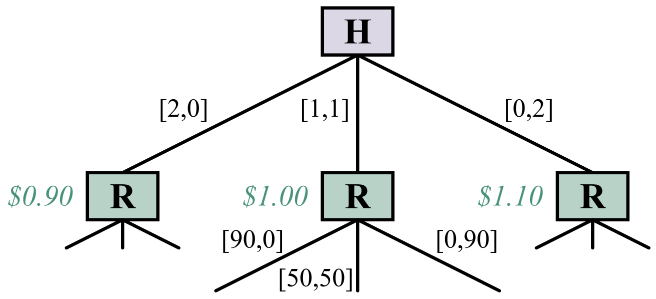


Figure 17.6 The paperclip game. Each branch is labeled $[p,s]$ denoting the number of paperclips and staples manufactured on that branch. Harriet the human can choose to make two paperclips, two staples, or one of each. (The values in green italics are the values for Harriet if the game ended there, assuming $\theta = 0.45$.) Robbie the robot then has a choice to make 90 paperclips, 90 staples, or 50 of each.

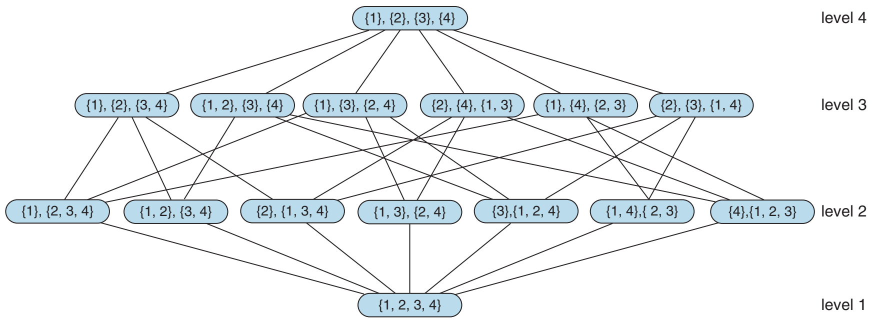


Figure 17.7 The coalition structure graph for $N = \{1,2,3,4\}$. Level 1 has coalition structures containing a single coalition; level 2 has coalition structures containing two coalitions, and so on.

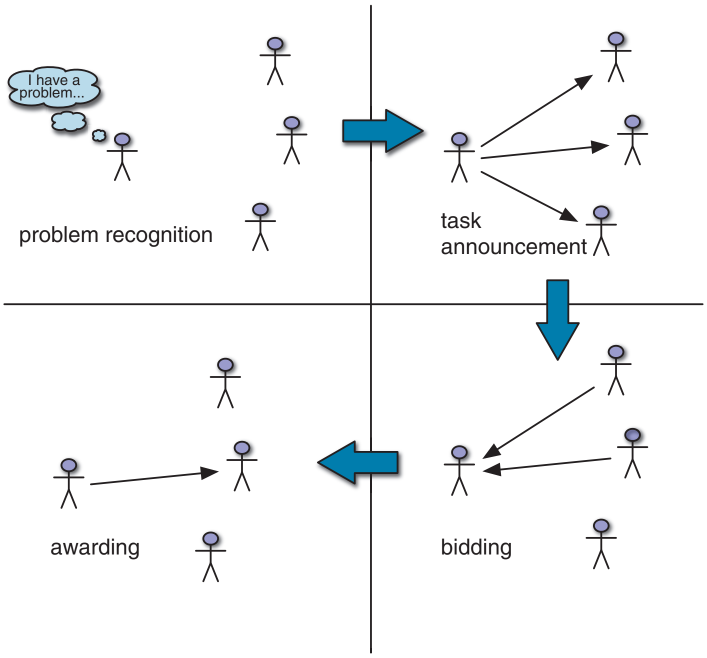


Figure 17.8 The contract net task allocation protocol.

CHAPTER 18

PROBABILISTIC PROGRAMMING

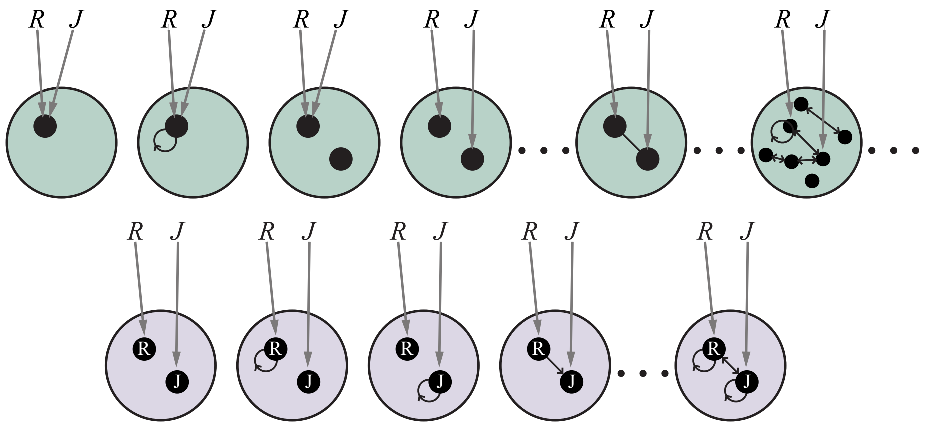


Figure 18.1 Top: Some members of the set of all possible worlds for a language with two constant symbols, R and J, and one binary relation symbol, under the standard semantics for first-order logic. Bottom: the possible worlds under database semantics. The interpretation of the constant symbols is fixed, and there is a distinct object for each constant symbol.

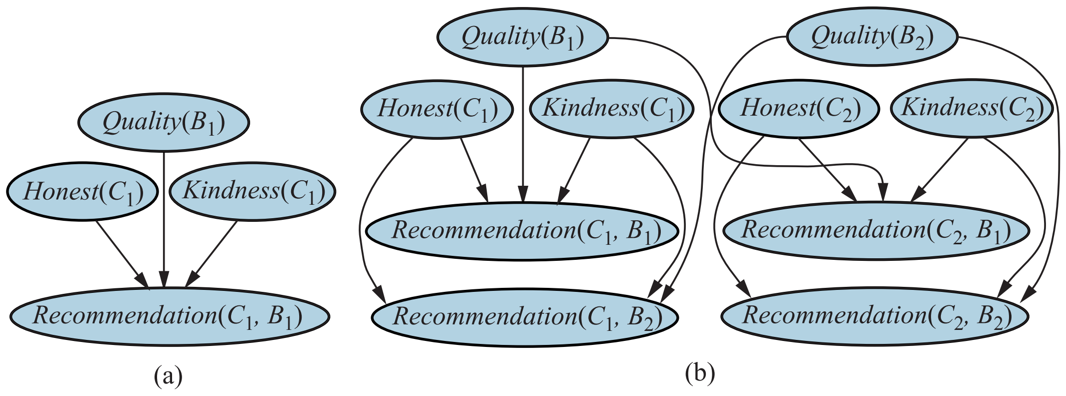

Quality(B_1)
Honest(C_1)
Kindness(C_1)
Recommendation(C_1, B_1)
(a)
Quality(B_1)
Quality(B_2)
Honest(C_1)
Kindness(C_1)
Honest(C_2)
Kindness(C_2)
Recommendation(C_1, B_1)
Recommendation(C_2, B_1)
Recommendation(C_1, B_2)
Recommendation(C_2, B_2)
(b)

Figure 18.2 (a) Bayes net for a single customer C_1 recommending a single book B_1. *Honest*(C_1) is Boolean, while the other variables have integer values from 1 to 5. (b) Bayes net with two customers and two books.

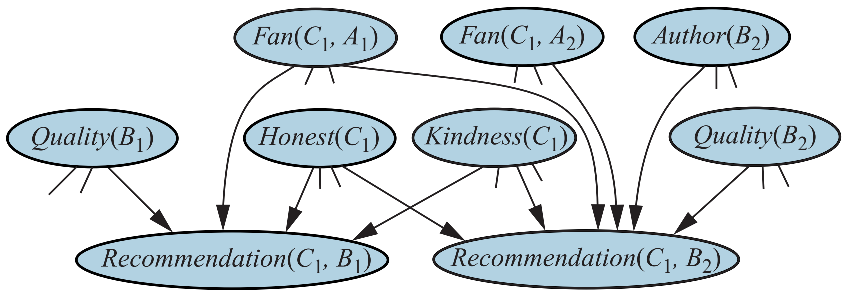


Figure 18.3 Fragment of the equivalent Bayes net for the book recommendation RPM when $Author(B_2)$ is unknown.

Variable	**Value**	**Probability**
$\#Customer$	2	0.3333
$\#Book$	3	0.3333
$Honest_{\langle Customer,,1\rangle}$	*true*	0.99
$Honest_{\langle Customer,,2\rangle}$	*false*	0.01
$Kindness_{\langle Customer,,1\rangle}$	4	0.3
$Kindness_{\langle Customer,,2\rangle}$	1	0.1
$Quality_{\langle Book,,1\rangle}$	1	0.05
$Quality_{\langle Book,,2\rangle}$	3	0.4
$Quality_{\langle Book,,3\rangle}$	5	0.15
$\#LoginID_{\langle Owner,\langle Customer,,1\rangle\rangle}$	1	1.0
$\#LoginID_{\langle Owner,\langle Customer,,2\rangle\rangle}$	2	0.25
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,1\rangle}$	2	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,2\rangle}$	4	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,1\rangle\rangle,1\rangle,\langle Book,,3\rangle}$	5	0.5
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,1\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,2\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,1\rangle,\langle Book,,3\rangle}$	1	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,1\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,2\rangle}$	5	0.4
$Recommendation_{\langle LoginID,\langle Owner,\langle Customer,,2\rangle\rangle,2\rangle,\langle Book,,3\rangle}$	1	0.4

Figure 18.4 One particular world for the book recommendation OUPM. The number variables and basic random variables are shown in topological order, along with their chosen values and the probabilities for those values.

type Researcher, Paper, Citation
random String Name(Researcher)
random String Title(Paper)
random Paper PubCited(Citation)
random String Text(Citation)
random Boolean Professor(Researcher)
origin Researcher Author(Paper)

$\#Researcher \sim OM(3,1)$
$Name(r) \sim NamePrior()$
$Professor(r) \sim Boolean(0.2)$
$\#Paper(Author = r) \sim \textbf{if } Professor(r) \textbf{ then } OM(1.5, 0.5) \textbf{ else } OM(1, 0.5)$
$Title(p) \sim PaperTitlePrior()$
$CitedPaper(c) \sim UniformChoice(\{Paper\ p\})$
$Text(c) \sim HMMGrammar(Name(Author(CitedPaper(c))), Title(CitedPaper(c)))$

Figure 18.5 An OUPM for citation information extraction. For simplicity the model assumes one author per paper and omits details of the grammar and error models.

$$
\begin{aligned}
&\#SeismicEvents \sim Poisson(T * \lambda_e)\\
&Time(e) \sim UniformReal(0,T)\\
&EarthQuake(e) \sim Boolean(0.999)\\
&Location(e) \sim \textbf{if } Earthquake(e) \textbf{ then } SpatialPrior() \textbf{ else } UniformEarth()\\
&Depth(e) \sim \textbf{if } Earthquake(e) \textbf{ then } UniformReal(0,700) \textbf{ else } Exactly(0)\\
&Magnitude(e) \sim Exponential(log(10))\\
&Detected(e,p,s) \sim Logistic(weights(s,p), Magnitude(e),\ Depth(e),\ Dist(e,s))\\
&\#Detections(site = s) \sim Poisson(T * \lambda_f(s))\\
&\#Detections(event{=}e,\ phase{=}p,\ station{=}s) = \textbf{if } Detected(e,p,s) \textbf{ then } 1 \textbf{ else } 0\\
&OnsetTime(a,s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0,T)\\
&\quad \textbf{else} = Time(event(a)) + GeoTT(Dist(event(a),s), Depth(event(a)), phase(a))\\
&\qquad\qquad + Laplace(\mu_t(s), \sigma_t(s))\\
&Amplitude(a,s) \textbf{ if } (event(a) = null) \textbf{ then } \sim NoiseAmpModel(s)\\
&\quad \textbf{else} = AmpModel(Magnitude(event(a)), Dist(event(a),s), Depth(event(a)), phase(a))\\
&Azimuth(a,s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0,\ 360)\\
&\quad \textbf{else} = GeoAzimuth(Location(event(a)), Depth(event(a)), phase(a), Site(s))\\
&\qquad\qquad + Laplace(0, \sigma_a(s))\\
&Slowness(a,s) \textbf{ if } (event(a) = null) \textbf{ then } \sim UniformReal(0,20)\\
&\quad \textbf{else} = GeoSlowness(Location(event(a)), Depth(event(a)), phase(a), Site(s))\\
&\qquad\qquad + Laplace(0, \sigma_s(s))\\
&ObservedPhase(a,s) \sim CategoricalPhaseModel(phase(a))
\end{aligned}
$$

Figure 18.6 A simplified version of the NET-VISA model (see text).

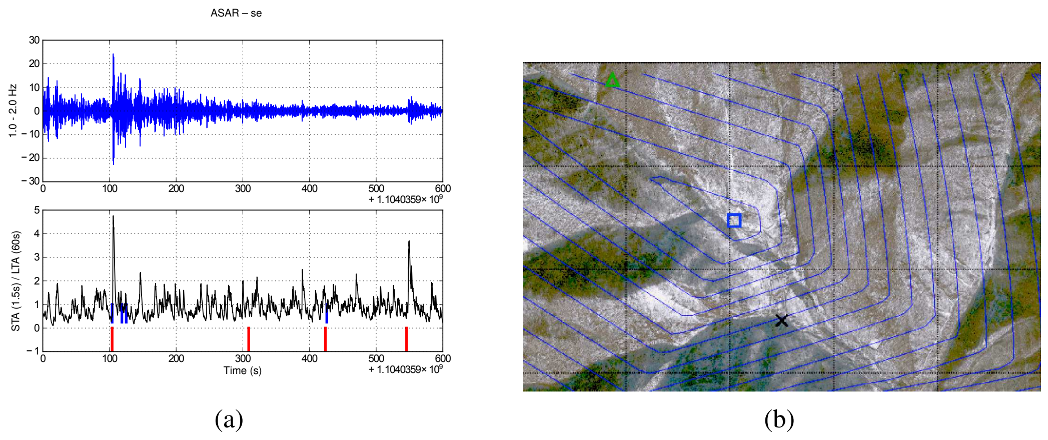


Figure 18.7 (a) Top: Example of seismic waveform recorded at Alice Springs, Australia. Bottom: the waveform after processing to detect the arrival times of seismic waves. Blue lines are the automatically detected arrivals; red lines are the true arrivals. (b) Location estimates for the DPRK nuclear test of February 12, 2013: UN CTBTO Late Event Bulletin (green triangle at top left); NET-VISA (blue square in center). The entrance to the underground test facility (small "x") is 0.75km from NET-VISA's estimate. Contours show NET-VISA's posterior location distribution. Courtesy of CTBTO Preparatory Commission.

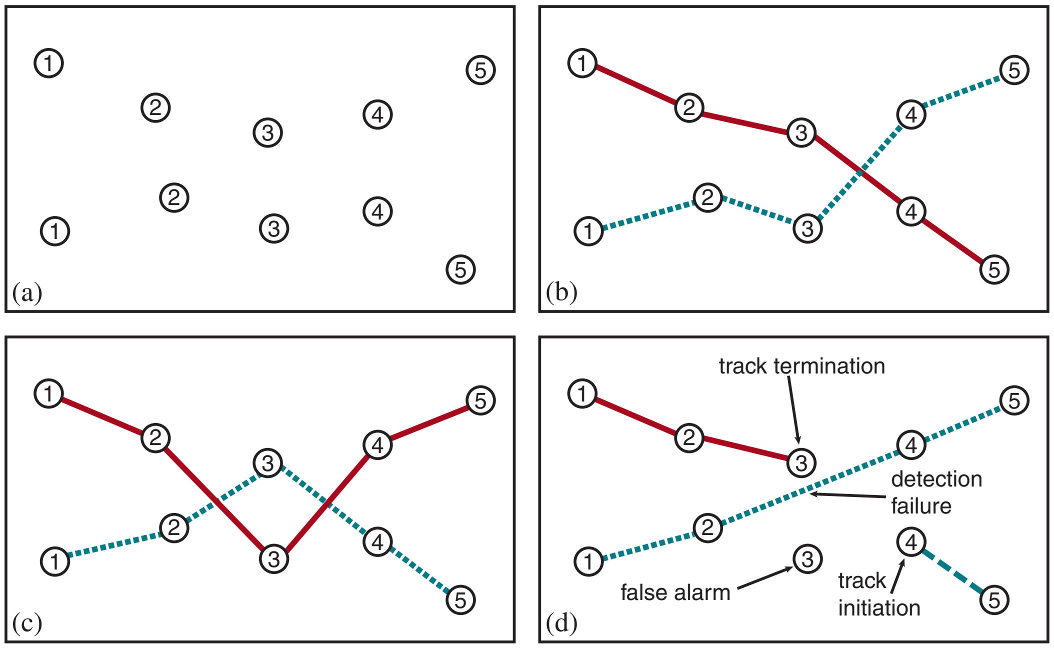


Figure 18.8 (a) Observations made of object locations in 2D space over five time steps. Each observation blip is labeled with the time step but does not identify the object that produced it. (b–c) Possible hypotheses about the underlying object tracks. (d) A hypothesis for the case in which false alarms, detection failures, and track initiation/termination are possible.

```
$\#Aircraft(EntryTime = t) \sim Poisson(\lambda_a)$
$Exits(a,t) \sim$ **if** $InFlight(a,t)$ **then** $Boolean(\alpha_e)$
$InFlight(a,t) = (t = EntryTime(a)) \vee (InFlight(a,t-1) \wedge \neg Exits(a,t-1))$
$X(a,t) \sim$ **if** $t = EntryTime(a)$ **then** $InitX()$
    **else if** $InFlight(a,t)$ **then** $\mathcal{N}(\mathbf{F}X(a,t-1), \Sigma_x)$
$\#Blip(Source = a, Time = t) \sim$ **if** $InFlight(a,t)$ **then** $Bernoulli(DetectionProb(X(a,t)))$
$\#Blip(Time = t) \sim Poisson(\lambda_f)$
$Z(b) \sim$ **if** $Source(b) = null$ **then** $UniformZ(R)$ **else** $\mathcal{N}(\mathbf{H}X(Source(b), Time(b)), \Sigma_z)$
```

Figure 18.9 An OUPM for radar tracking of multiple targets with false alarms, detection failure, and entry and exit of aircraft. The rate at which new aircraft enter the scene is λ_a, while the probability per time step that an aircraft exits the scene is α_e. False alarm blips (i.e., ones not produced by an aircraft) appear uniformly in space at a rate of λ_f per time step. The probability that an aircraft is detected (i.e., produces a blip) depends on its current position.

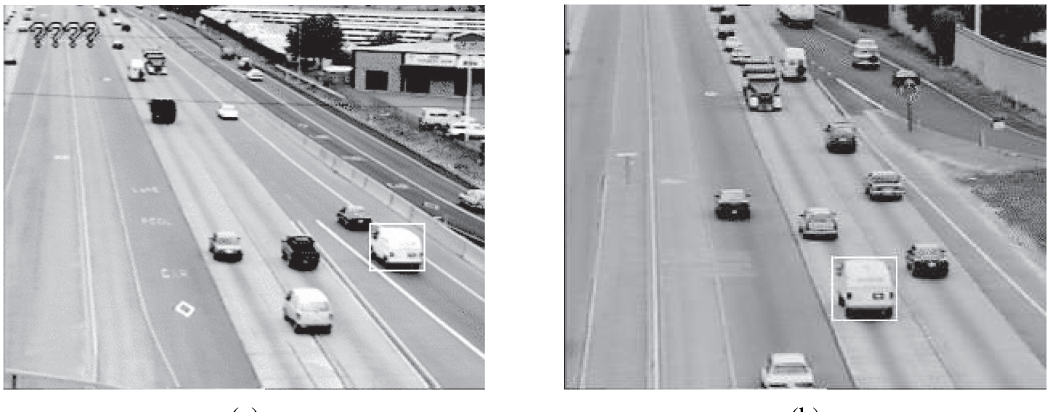

(a) (b)

Figure 18.10 Images from (a) upstream and (b) downstream surveillance cameras roughly two miles apart on Highway 99 in Sacramento, California. The boxed vehicle has been identified at both cameras.

```
function GENERATE-IMAGE() returns an image with some letters
   $letters \leftarrow$ GENERATE-LETTERS(10)
   return RENDER-NOISY-IMAGE($letters$, 32, 128)

function GENERATE-LETTERS($\lambda$) returns a vector of letters
   $n \sim Poisson(\lambda)$
   $letters \leftarrow [\,]$
   for $i$ = 1 to $n$ do
      $letters[i] \sim UniformChoice(\{a,b,c,\cdots\})$
   return $letters$

function RENDER-NOISY-IMAGE($letters, width, height$) returns a noisy image of the letters
   $clean\_image \leftarrow$ RENDER($letters, width, height, text\_top = 10, text\_left = 10$)
   $noisy\_image \leftarrow [\,]$
   $noise\_variance \sim UniformReal(0.1,\ 1)$
   for $row$ = 1 to $width$ do
      for $col$ = 1 to $height$ do
         $noisy\_image[row,col] \sim \mathcal{N}(clean\_image[row,col], noise\_variance)$
   return $noisy\_image$
```

Figure 18.11 Generative program for an open-universe probability model for optical character recognition. The generative program produces degraded images containing sequences of letters by generating each sequence, rendering it into a 2D image, and incorporating additive noise at each pixel.

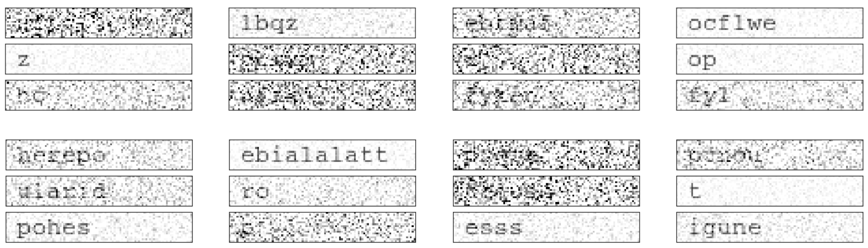


Figure 18.12 The top panel shows twelve degraded images produced by executing the generative program from Figure 18.11. The number of letters, their identities, the amount of additive noise, and the specific pixel-wise noise are all part of the domain of the probability model. The bottom panel shows twelve degraded images produced by executing the generative program from Figure 18.15. The Markov model for letters typically yields sequences of letters that are easier to pronounce.

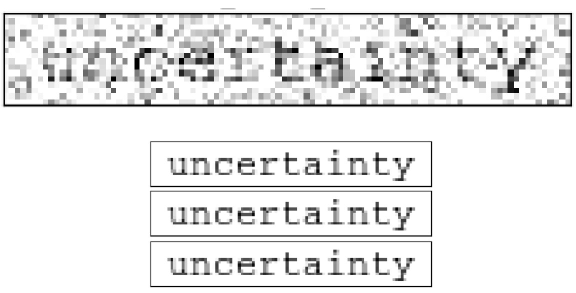


Figure 18.13 Noisy input image (top) and inference results (bottom) produced by three runs, each of 25 MCMC iterations, with the model from Figure 18.11. Note that the inference process correctly identifies the sequence of letters.

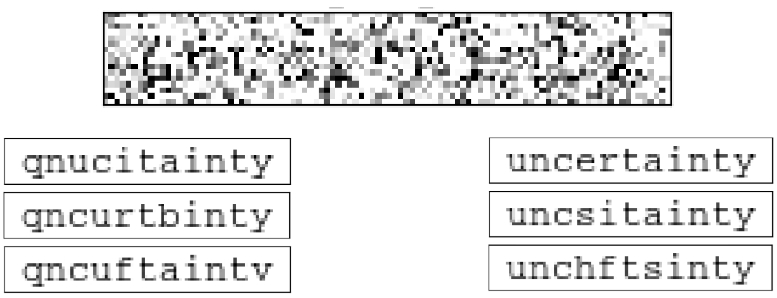


Figure 18.14 Top: extremely noisy input image. Bottom left: with three inference results from 25 MCMC iterations with the independent-letter model from Figure 18.11. Bottom right: three inference results with the letter bigram model from Figure 18.15. Both models exhibit ambiguity in the results, but the latter model's results reflect prior knowledge of plausible letter sequences.

```
function GENERATE-MARKOV-LETTERS($\lambda$) returns a vector of letters
  $n \sim Poisson(\lambda)$
  $letters \leftarrow [\,]$
  $letter\_probs \leftarrow$ MARKOV-INITIAL()
  for $i = 1$ to $n$ do
      $letters[i] \sim Categorical(letter\_probs)$
      $letter\_probs \leftarrow$ MARKOV-TRANSITION($letters[i]$)
  return $letters$
```

Figure 18.15 Generative program for an improved optical character recognition model that generates letters according to a letter bigram model whose pairwise letter frequencies are estimated from a list of English words.

CHAPTER 19

LEARNING FROM EXAMPLES

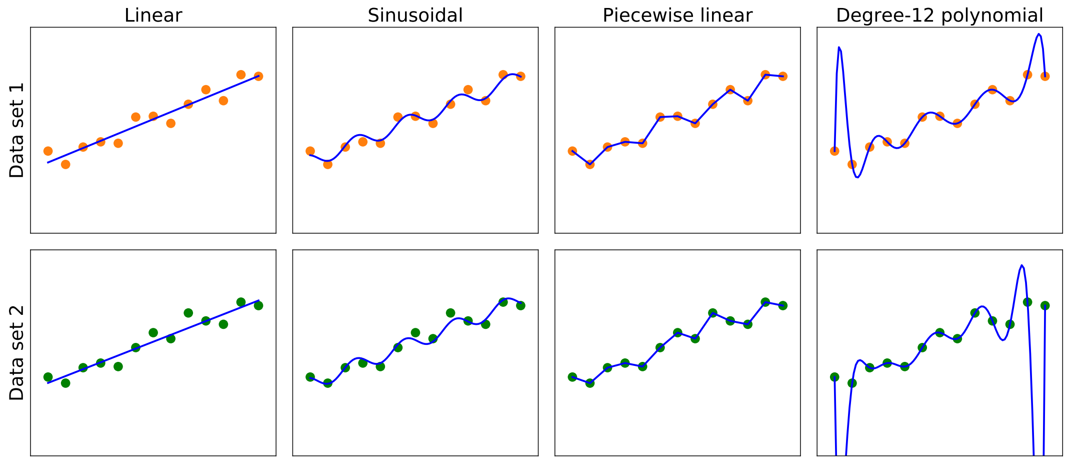


Figure 19.1 Finding hypotheses to fit data. **Top row**: four plots of best-fit functions from four different hypothesis spaces trained on data set 1. **Bottom row**: the same four functions, but trained on a slightly different data set (sampled from the same $f(x)$ function).

Example	Input Attributes										Output
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y_1 = Yes$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y_2 = No$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y_3 = Yes$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = Yes$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y_5 = No$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = Yes$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = No$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = Yes$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = No$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = No$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = No$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = Yes$

Figure 19.2 Examples for the restaurant domain.

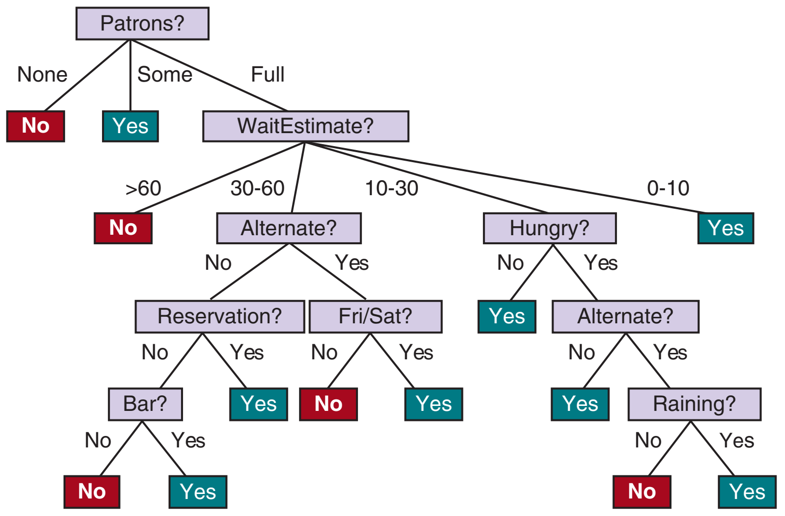


Figure 19.3 A decision tree for deciding whether to wait for a table.

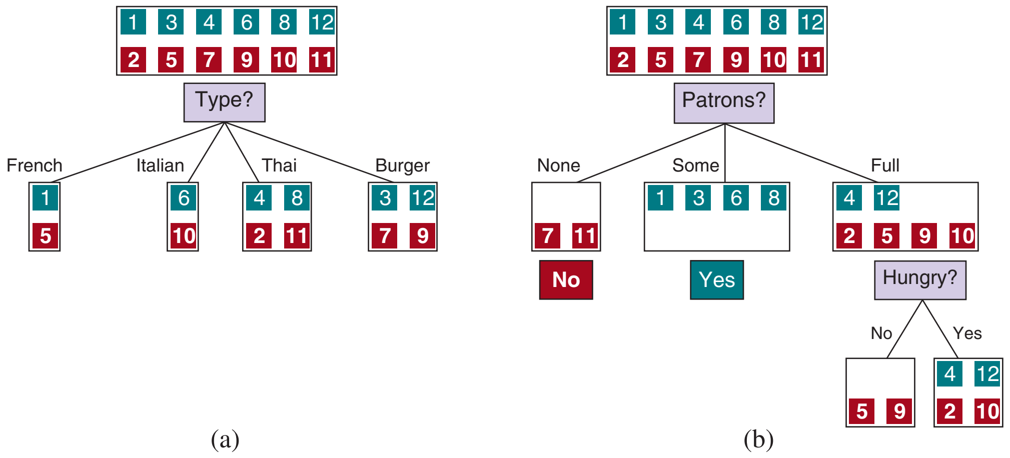


Figure 19.4 Splitting the examples by testing on attributes. At each node we show the positive (light boxes) and negative (dark boxes) examples remaining. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

```
function LEARN-DECISION-TREE(examples, attributes, parent_examples) returns a tree
  if examples is empty then return PLURALITY-VALUE(parent_examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
     $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE(a, examples)
     tree ← a new decision tree with root test A
     for each value v of A do
        $exs \leftarrow \{e : e \in examples$ and $e.A = v\}$
        subtree ← LEARN-DECISION-TREE(exs, attributes − A, examples)
        add a branch to tree with label $(A = v)$ and subtree subtree
     return tree
```

Figure 19.5 The decision tree learning algorithm. The function IMPORTANCE is described in Section 19.3.3. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

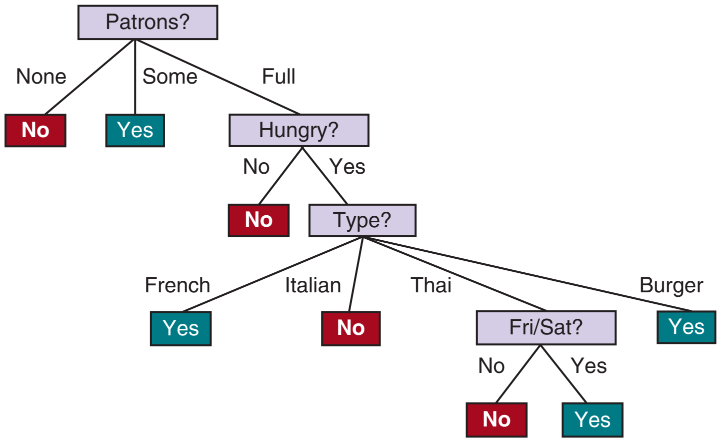


Figure 19.6 The decision tree induced from the 12-example training set.

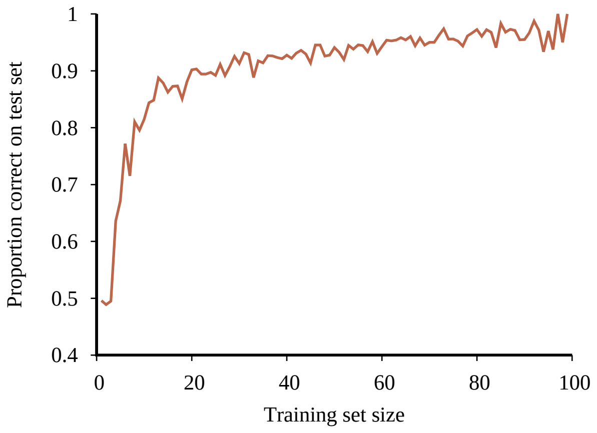


Figure 19.7 A learning curve for the decision tree learning algorithm on 100 randomly generated examples in the restaurant domain. Each data point is the average of 20 trials.

```
function MODEL-SELECTION($Learner, examples, k$) returns a (hypothesis, error rate) pair

  $err \leftarrow$ an array, indexed by $size$, storing validation-set error rates
  $training\_set$, $test\_set \leftarrow$ a partition of $examples$ into two sets
  for $size$ = 1 to $\infty$ do
      $err[size] \leftarrow$ CROSS-VALIDATION($Learner, size, training\_set, k$)
      if $err$ is starting to increase significantly then
          $best\_size \leftarrow$ the value of $size$ with minimum $err[size]$
          $h \leftarrow Learner(best\_size, training\_set)$
          return $h$, ERROR-RATE($h$, $test\_set$)

function CROSS-VALIDATION($Learner, size, examples, k$) returns error rate

  $N \leftarrow$ the number of $examples$
  $errs \leftarrow 0$
  for $i$ = 1 to $k$ do
     $validation\_set \leftarrow examples[(i - 1) \times N/k : i \times N/k]$
     $training\_set \leftarrow examples - validation\_set$
     $h \leftarrow Learner(size, training\_set)$
     $errs \leftarrow errs$ + ERROR-RATE($h, validation\_set$)
  return $errs / k$        // average error rate on validation sets, across k-fold cross-validation
```

Figure 19.8 An algorithm to select the model that has the lowest validation error. It builds models of increasing complexity, and choosing the one with best empirical error rate, *err*, on the validation data set. $Learner(size, examples)$ returns a hypothesis whose complexity is set by the parameter *size*, and which is trained on *examples*. In CROSS-VALIDATION, each iteration of the **for** loop selects a different slice of the *examples* as the validation set, and keeps the other examples as the training set. It then returns the average validation set error over all the folds. Once we have determined which value of the *size* parameter is best, MODEL-SELECTION returns the model (i.e., learner/hypothesis) of that size, trained on all the training examples, along with its error rate on the held-out test examples.

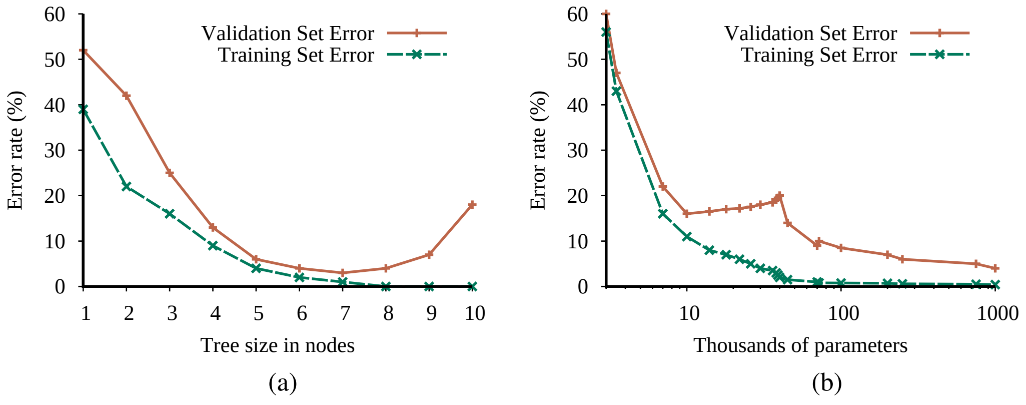


Figure 19.9 Error rates on training data (lower, green line) and validation data (upper, orange line) for models of different complexity on two different problems. MODEL-SELECTION picks the hyperparameter value with the lowest validation-set error. In (a) the model class is decision trees and the hyperparameter is the number of nodes. The data is from a version of the restaurant problem. The optimal size is 7. In (b) the model class is convolutional neural networks (see Section 22.3) and the hyperparameter is the number of regular parameters in the network. The data is the MNIST data set of images of digits; the task is to identify each digit. The optimal number of parameters is 1,000,000 (note the log scale).

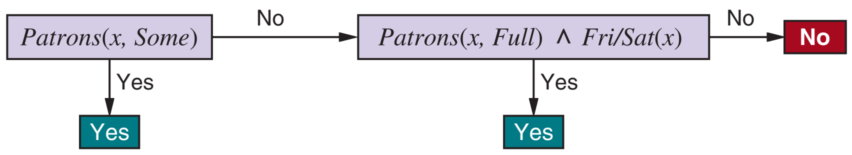


Figure 19.10 A decision list for the restaurant problem.

```
function DECISION-LIST-LEARNING(examples) returns a decision list, or failure

  if examples is empty then return the trivial decision list No
  $t \leftarrow$ a test that matches a nonempty subset $examples_t$ of examples
      such that the members of $examples_t$ are all positive or all negative
  if there is no such t then return failure
  if the examples in $examples_t$ are positive then $o \leftarrow Yes$ else $o \leftarrow No$
  return a decision list with initial test t and outcome o and remaining tests given by
       DECISION-LIST-LEARNING($examples - examples_t$)
```

Figure 19.11 An algorithm for learning decision lists.

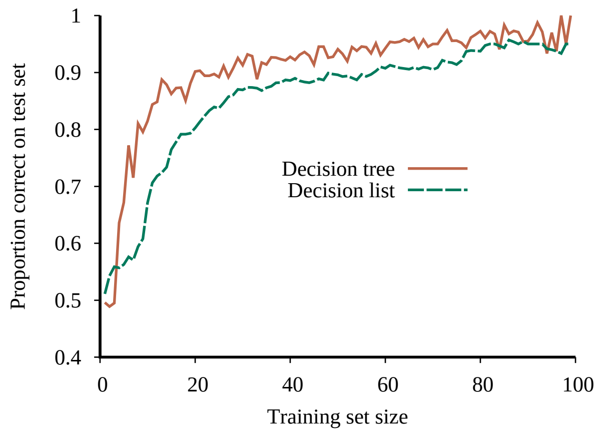


Figure 19.12 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for LEARN-DECISION-TREE is shown for comparison; decision trees do slightly better on this particular problem.

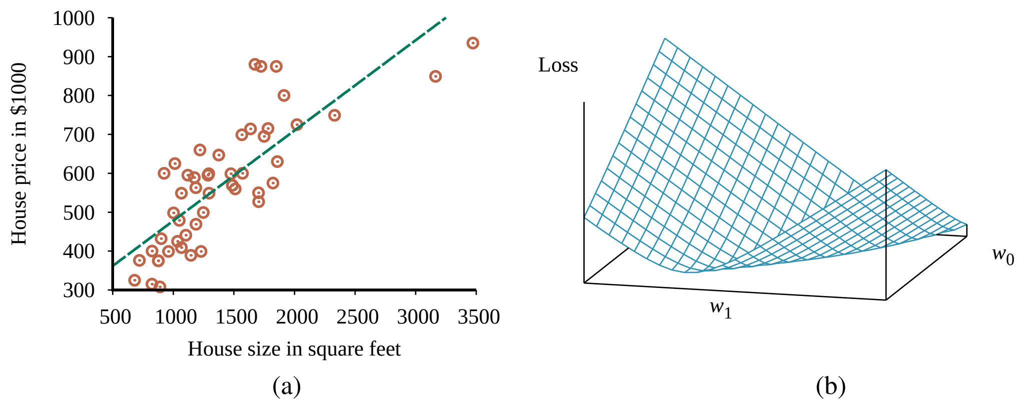


Figure 19.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (y_j - w_1 x_j + w_0)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

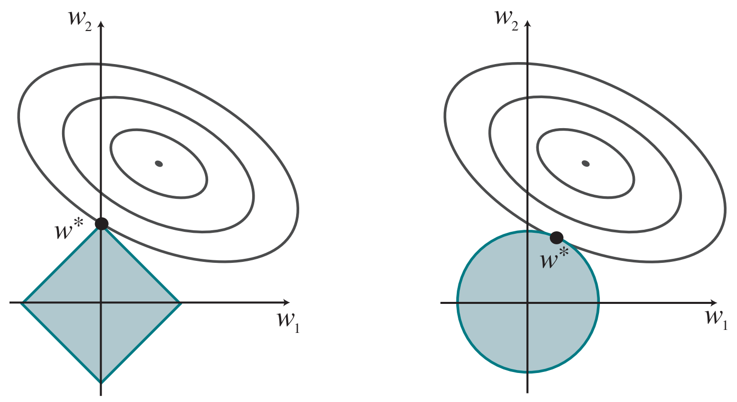


Figure 19.14 Why L_1 regularization tends to produce a sparse model. Left: With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

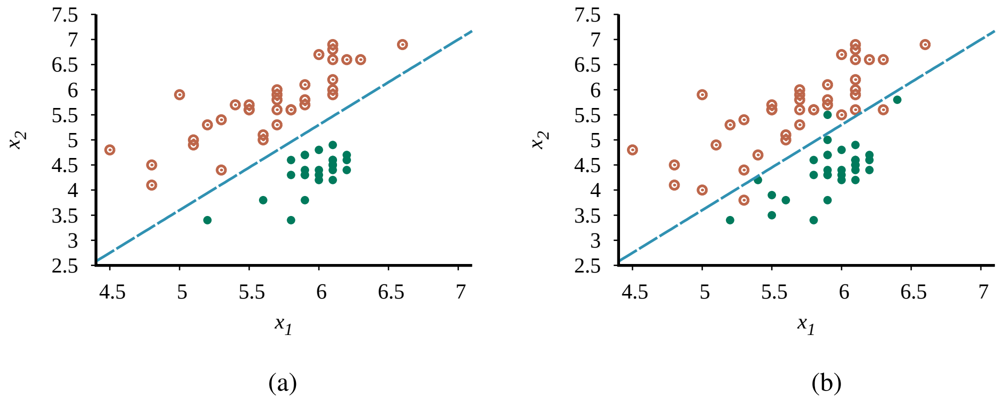


Figure 19.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (open orange circles) and nuclear explosions (green circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

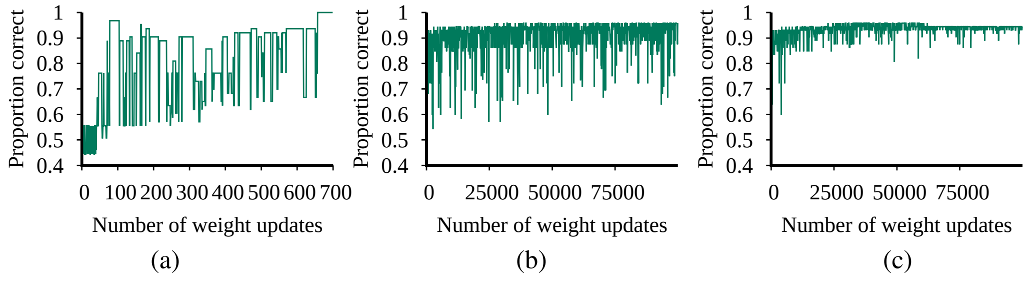


Figure 19.16 (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 19.15(a). (b) The same plot for the noisy, nonseparable data in Figure 19.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000+t)$.

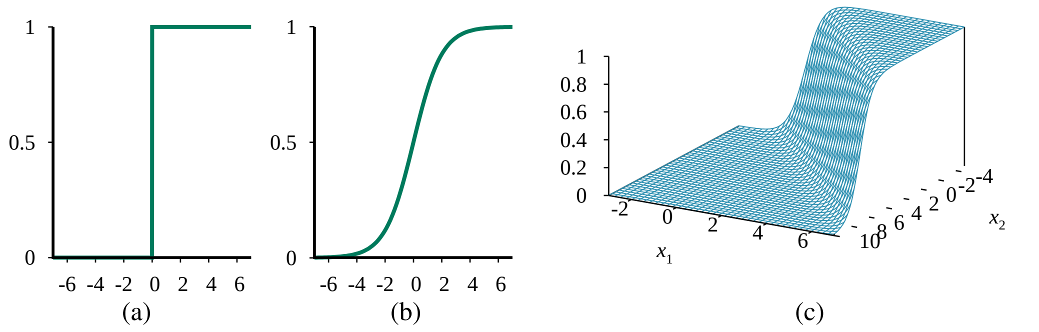


Figure 19.17 (a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is nondifferentiable at $z=0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 19.15(b).

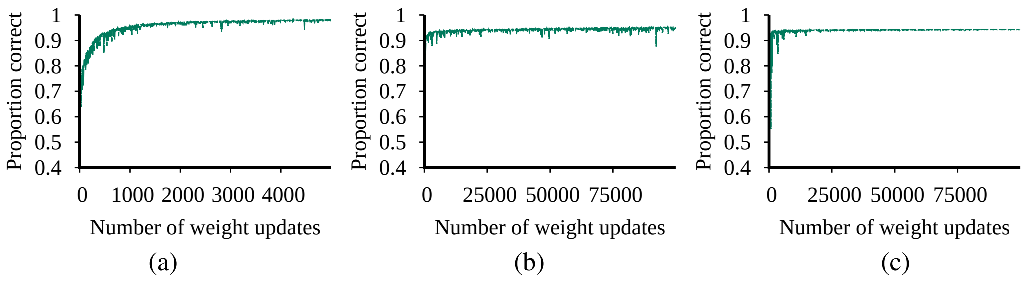


Figure 19.18 Repeat of the experiments in Figure 19.16 using logistic regression. The plot in (a) covers 5000 iterations rather than 700, while the plots in (b) and (c) use the same scale as before.

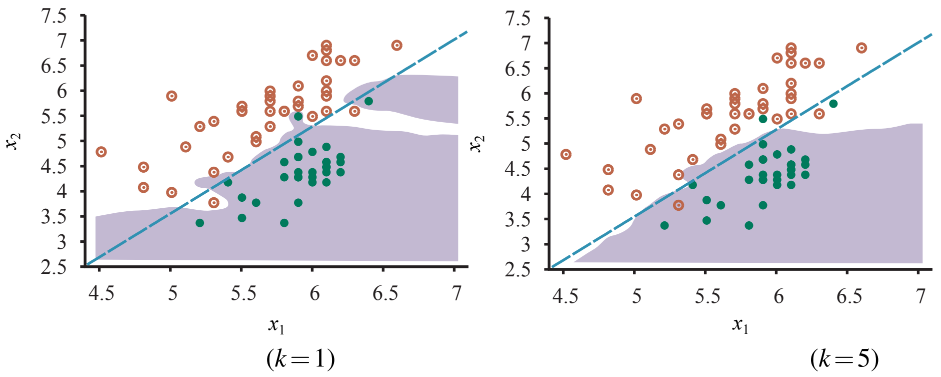


Figure 19.19 (a) A k-nearest-neighbors model showing the extent of the explosion class for the data in Figure 19.15, with $k=1$. Overfitting is apparent. (b) With $k=5$, the overfitting problem goes away for this data set.

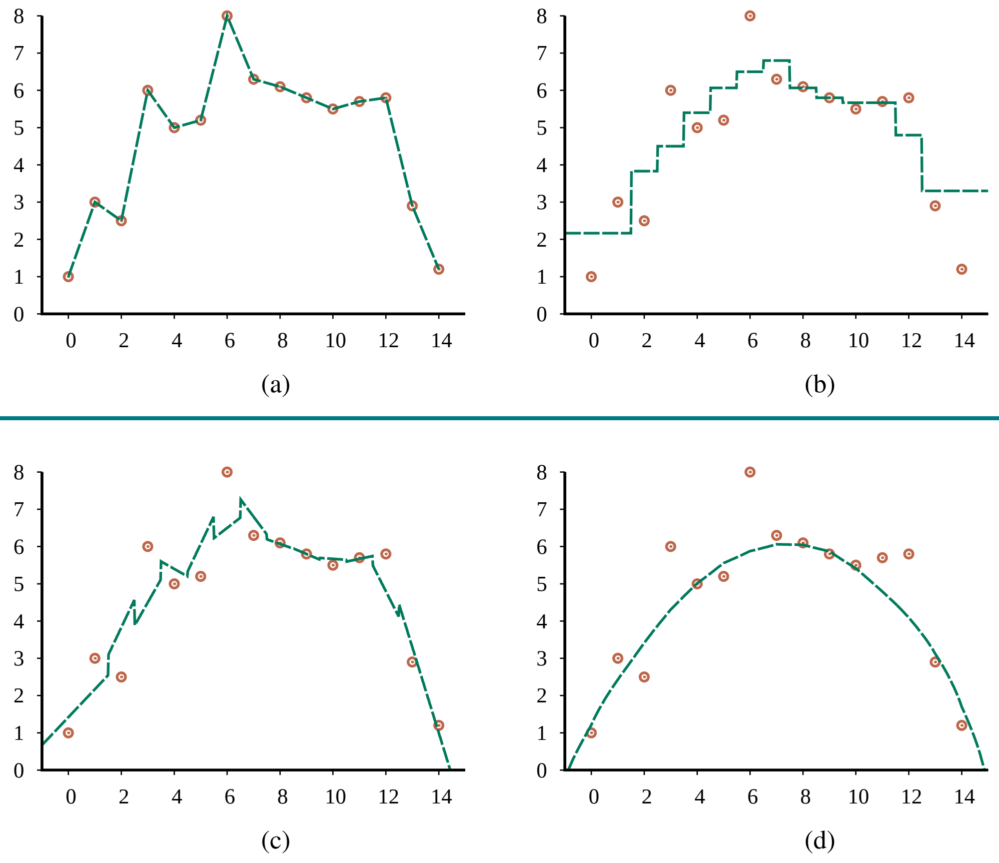
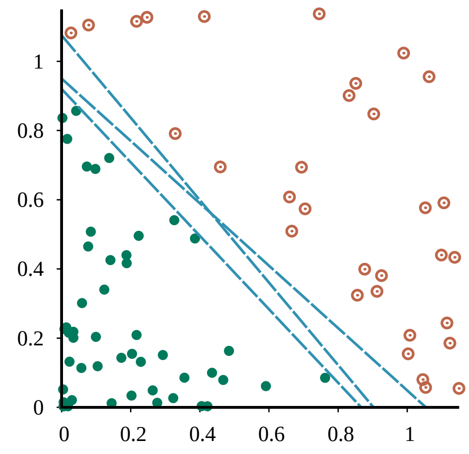
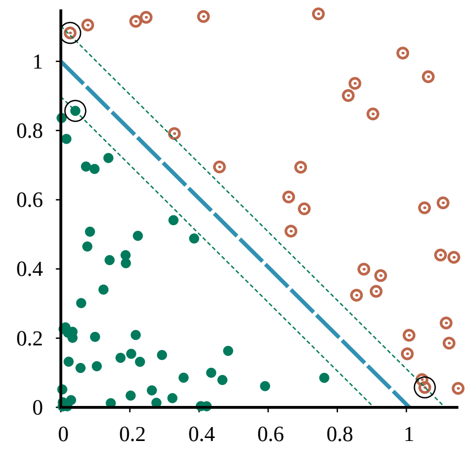


Figure 19.20 Nonparametric regression models: (a) connect the dots, (b) 3-nearest neighbors average, (c) 3-nearest-neighbors linear regression, (d) locally weighted regression with a quadratic kernel of width 10.



(a)



(b)

Figure 19.21 Support vector machine classification: (a) Two classes of points (orange open and green filled circles) and three candidate linear separators. (b) The maximum margin separator (heavy line), is at the midpoint of the **margin** (area between dashed lines). The **support vectors** (points with large black circles) are the examples closest to the separator; here there are three.

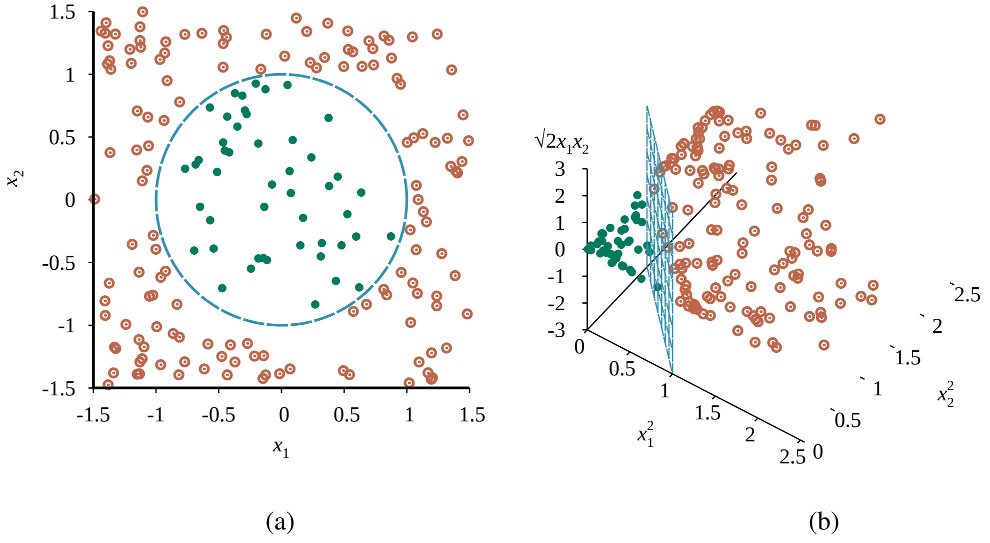


Figure 19.22 (a) A two-dimensional training set with positive examples as green filled circles and negative examples as orange open circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions. Figure 19.21(b) gives a closeup of the separator in (b).

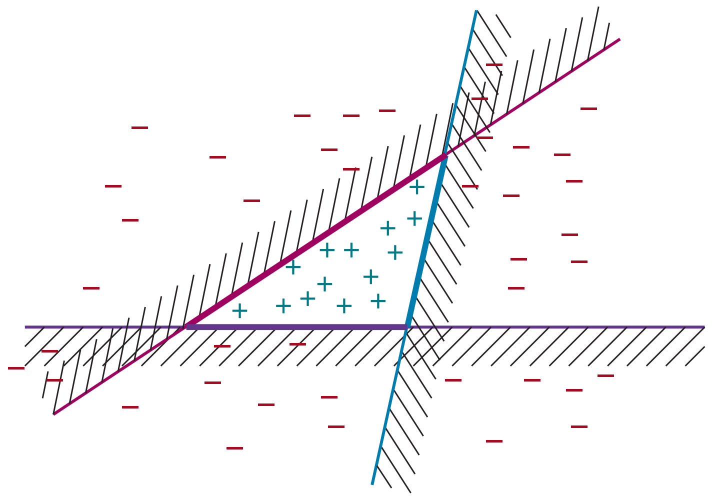


Figure 19.23 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the unshaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

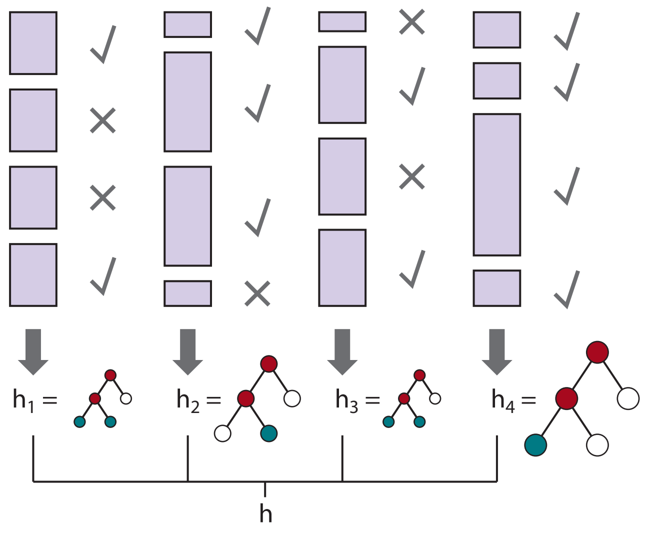
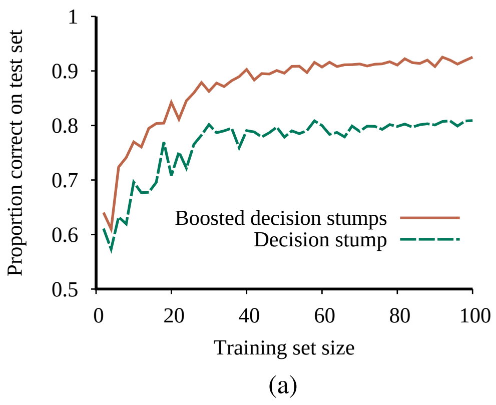


Figure 19.24 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The checks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

```
function AdaBoost($examples, L, K$) returns a hypothesis
  inputs: $examples$, set of $N$ labeled examples $(x_1,y_1),\ldots,(x_N,y_N)$
          $L$, a learning algorithm
          $K$, the number of hypotheses in the ensemble
  local variables: $\mathbf{w}$, a vector of $N$ example weights, initially all $1/N$
                   $\mathbf{h}$, a vector of $K$ hypotheses
                   $\mathbf{z}$, a vector of $K$ hypothesis weights

  $\epsilon \leftarrow$ a small positive number, used to avoid division by zero
  for $k$ = 1 to $K$ do
      $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$
      $error \leftarrow 0$
      for $j$ = 1 to $N$ do        // Compute the total error for $\mathbf{h}[k]$
          if $\mathbf{h}[k](x_j) \neq y_j$ then $error \leftarrow error + \mathbf{w}[j]$
      if $error > 1/2$ then break from loop
      $error \leftarrow \min(error,\ 1 - \epsilon)$
      for $j$ = 1 to $N$ do        // Give more weight to the examples $\mathbf{h}[k]$ got wrong
          if $\mathbf{h}[k](x_j) = y_j$ then $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
      $\mathbf{w} \leftarrow$ Normalize($\mathbf{w}$)
      $\mathbf{z}[k] \leftarrow \frac{1}{2} \log((1 - error)/error)$        // Give more weight to accurate $\mathbf{h}[k]$
  return $Function(x) : \sum \mathbf{z}_i\ \mathbf{h}_i(x)$
```

Figure 19.25 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**. For regression problems, or for binary classification with two classes -1 and 1, this is $\sum_k \mathbf{h}[k]\mathbf{z}[k]$.



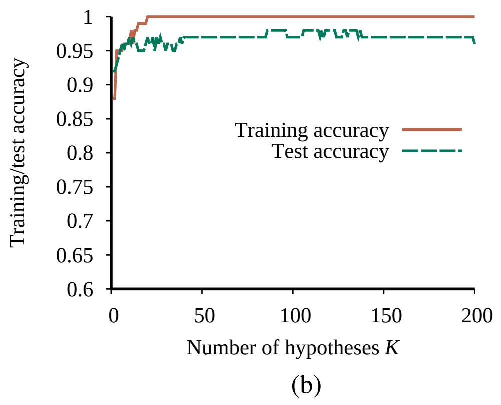


Figure 19.26 (a) Graph showing the performance of boosted decision stumps with $K=5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

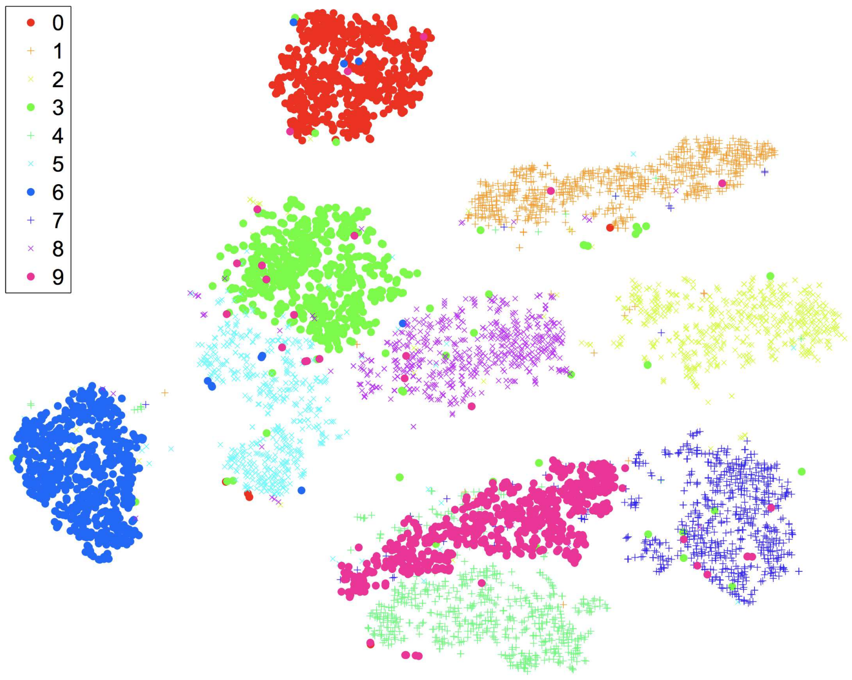


Figure 19.27 A two-dimensional t-SNE map of the MNIST data set, a collection of 60,000 images of handwritten digits, each 28×28 pixels and thus 784 dimensions. You can clearly see clusters for the ten digits, with a few confusions in each cluster; for example the top cluster is for the digit 0, but within the bounds of the cluster are a few data points representing the digits 3 and 6. The t-SNE algorithm finds a representation that accentuates the differences between clusters.

Tests for Features and Data

(1) Feature expectations are captured in a schema. (2) All features are beneficial. (3) No feature's cost is too much. (4) Features adhere to meta-level requirements. (5) The data pipeline has appropriate privacy controls. (6) New features can be added quickly. (7) All input feature code is tested.

Tests for Model Development

(1) Every model specification undergoes a code review. (2) Every model is checked in to a repository. (3) Offline proxy metrics correlate with actual metrics (4) All hyperparameters have been tuned. (5) The impact of model staleness is known. (6) A simpler model is not better. (7) Model quality is sufficient on all important data slices. The model has been tested for considerations of inclusion.

Tests for Machine Learning Infrastructure

(1) Training is reproducible. (2) Model specification code is unit tested. (3) The full ML pipeline is integration tested. (4) Model quality is validated before attempting to serve it. (5) The model allows debugging by observing the step-by-step computation of training or inference on a single example. (6) Models are tested via a canary process before they enter production serving environments. (7) Models can be quickly and safely rolled back to a previous serving version.

Monitoring Tests for Machine Learning

(1) Dependency changes result in notification. (2) Data invariants hold in training and serving inputs. (3) Training and serving features compute the same values. (4) Models are not too stale. (5) The model is numerically stable. (6) The model has not experienced regressions in training speed, serving latency, throughput, or RAM usage. (7) The model has not experienced a regression in prediction quality on served data.

Figure 19.28 A set of criteria to see how well you are doing at deploying your machine learning model with sufficient tests. Abridged from Breck *et al.* (2016), who also provide a scoring metric.

CHAPTER 20

KNOWLEDGE IN LEARNING

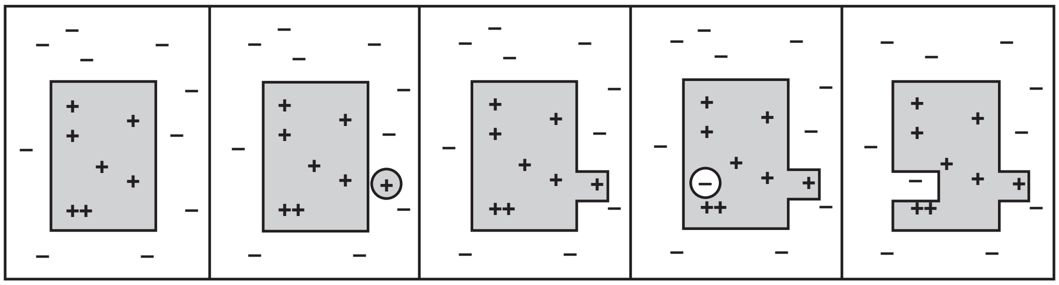


Figure 20.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

```
function CURRENT-BEST-LEARNING(examples, h) returns a hypothesis or fail

  if examples is empty then
     return h
  e ← FIRST(examples)
  if e is consistent with h then
     return CURRENT-BEST-LEARNING(REST(examples), h)
  else if e is a false positive for h then
     for each h' in  specializations of h consistent with examples seen so far do
        h'' ← CURRENT-BEST-LEARNING(REST(examples), h')
        if h'' ≠ fail then return h''
  else if e is a false negative for h then
     for each h' in generalizations of h consistent with examples seen so far do
        h'' ← CURRENT-BEST-LEARNING(REST(examples), h')
        if h'' ≠ fail then return h''
  return fail
```

Figure 20.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or gneralized as needed.

```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: $V$, the version space: the set of all hypotheses

  $V \leftarrow$ the set of all hypotheses
  for each example $e$ in examples do
      if $V$ is not empty then $V \leftarrow$ VERSION-SPACE-UPDATE($V, e$)
  return $V$
```

```
function VERSION-SPACE-UPDATE($V, e$) returns an updated version space
  $V \leftarrow \{h \in V : h$ is consistent with $e\}$
```

Figure 20.3 The version space learning algorithm. It finds a subset of V that is consistent with all the *examples*.

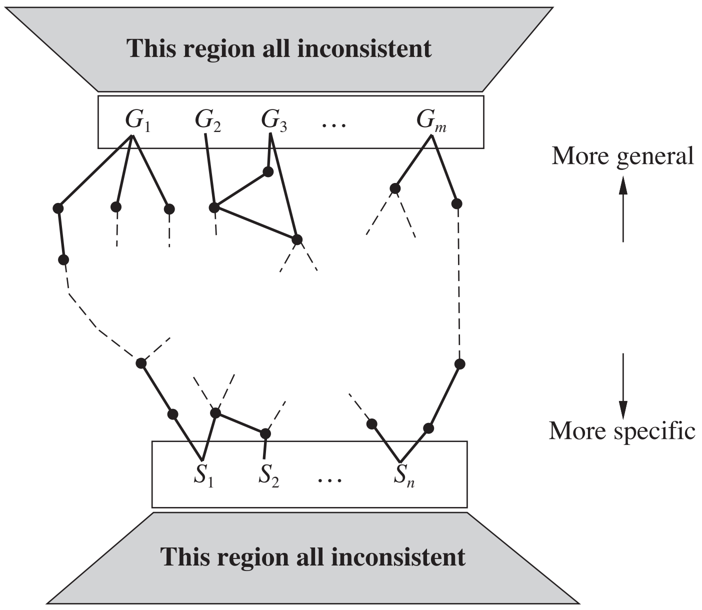


Figure 20.4 The version space contains all hypotheses consistent with the examples.

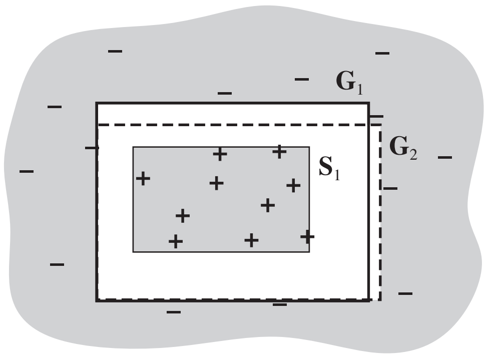


Figure 20.5 The extensions of the members of *G* and *S*. No known examples lie in between the two sets of boundaries.

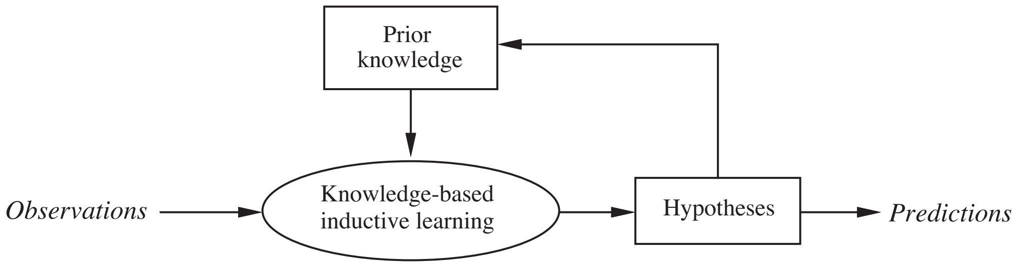


Figure 20.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

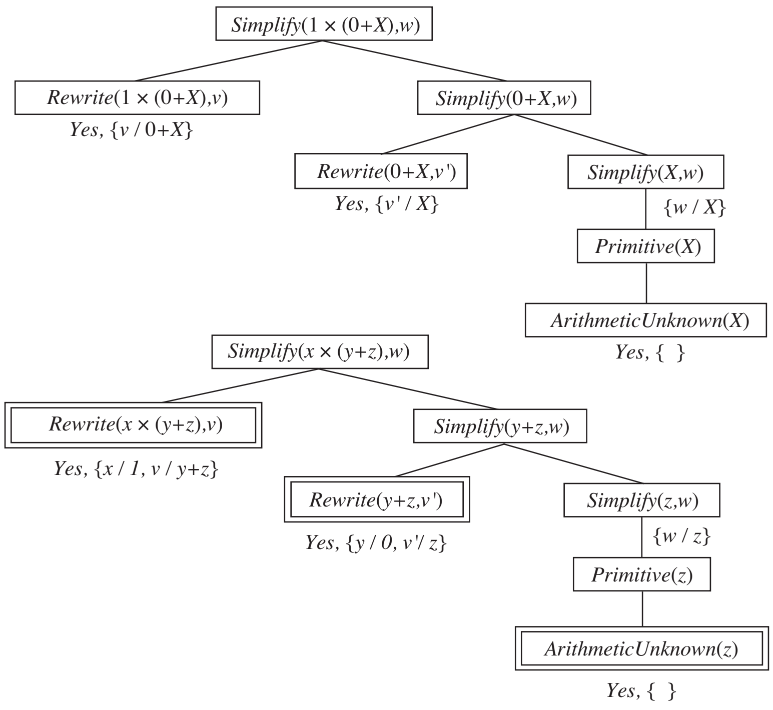


Figure 20.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$\mathit{ArithmeticUnknown}(z) \Rightarrow \mathit{Simplify}(1 \times (0+z), z) .$$

The second tree shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

```
function MINIMAL-CONSISTENT-DET($E, A$) returns a set of attributes
  inputs: $E$, a set of examples
          $A$, a set of attributes, of size $n$

  for $i = 0$ to $n$ do
    for each subset $A_i$ of $A$ of size $i$ do
      if CONSISTENT-DET?($A_i, E$) then return $A_i$
```

```
function CONSISTENT-DET?($A, E$) returns a truth value
  inputs: $A$, a set of attributes
          $E$, a set of examples
  local variables: $H$, a hash table

  for each example $e$ in $E$ do
    if some example in $H$ has the same values as $e$ for the attributes $A$
      but a different classification then return false
    store the class of $e$ in $H$, indexed by the values for attributes $A$ of the example $e$
  return true
```

Figure 20.8 An algorithm for finding a minimal consistent determination.

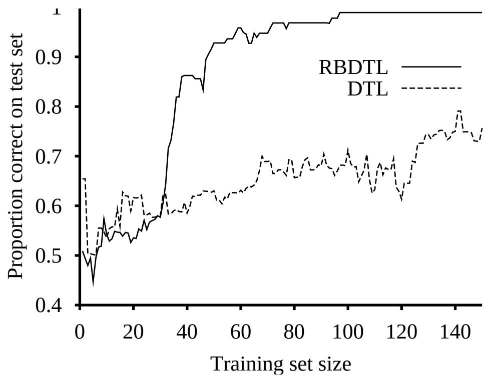
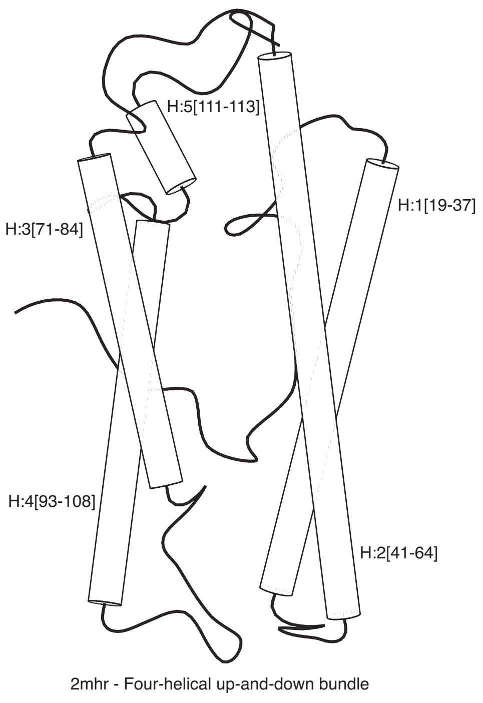
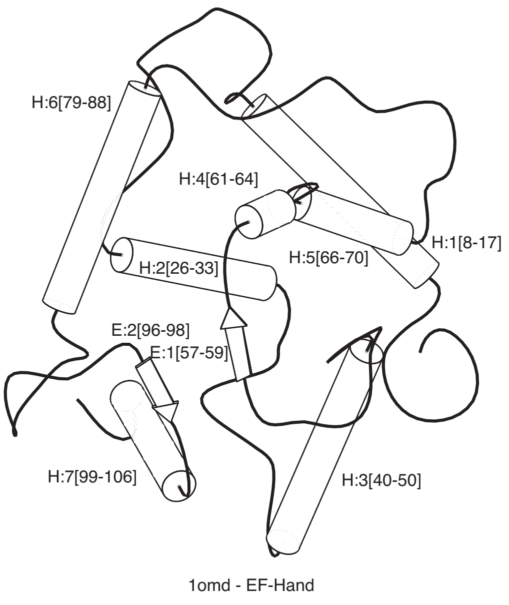


Figure 20.9 A performance comparison between DECISION-TREE-LEARNING and RBDTL on randomly generated data for a target function that depends on only 5 of 16 attributes.



(a)



(b)

Figure 20.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as $\mathit{TotalLength}(D2mhr, 118) \wedge \mathit{NumberHelices}(D2mhr, 6) \wedge \ldots$. From these descriptions and from classifications such as $\mathit{Fold}(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, D2mhr)$, the ILP system PROGOL (Muggleton, 1995) learned the following rule:

$$
\begin{aligned}
&\mathit{Fold}(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow \\
&\quad \mathit{Helix}(p, h_1) \wedge \mathit{Length}(h_1, \text{HIGH}) \wedge \mathit{Position}(p, h_1, n) \\
&\quad \wedge (1 \leq n \leq 3) \wedge \mathit{Adjacent}(p, h_1, h_2) \wedge \mathit{Helix}(p, h_2)\ .
\end{aligned}
$$

This kind of rule could not be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters. The rule can be translated into English as " Protein p has fold class "Four-helical up-and-down-bundle" if it contains a long helix h_1 at a secondary structure position between 1 and 3 and h_1 is next to a second helix."

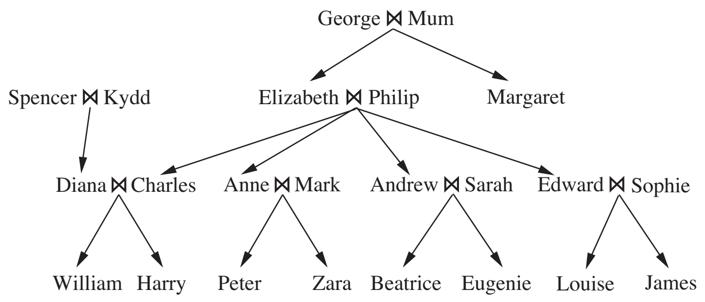


Figure 20.11 A typical family tree.

```
**function** FOIL(*examples*, *target*) **returns** a set of Horn clauses
  **inputs**: *examples*, set of examples
          *target*, a literal for the goal predicate
  **local variables**: *clauses*, set of clauses, initially empty

  **while** *examples* contains positive examples **do**
     $clause \leftarrow$ NEW-CLAUSE(*examples*, *target*)
     remove positive examples covered by *clause* from *examples*
     add *clause* to *clauses*
  **return** *clauses*
```

```
**function** NEW-CLAUSE(*examples*, *target*) **returns** a Horn clause
  **local variables**: *clause*, a clause with *target* as head and an empty body
                    *l*, a literal to be added to the clause
                    *extended_examples*, a set of examples with values for new variables

  $extended\_examples \leftarrow examples$
  **while** *extended_examples* contains negative examples **do**
     $l \leftarrow$ CHOOSE-LITERAL(NEW-LITERALS(*clause*), *extended_examples*)
     append *l* to the body of *clause*
     $extended\_examples \leftarrow$ set of examples created by applying EXTEND-EXAMPLE
        to each example in *extended_examples*
  **return** *clause*

**function** EXTEND-EXAMPLE(*example*, *literal*) **returns** a set of examples
  **if** *example* satisfies *literal*
     **then return** the set of examples created by extending *example* with
        each possible constant value for each new variable in *literal*
  **else return** the empty set
```

Figure 20.12 Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERALS and CHOOSE-LITERAL are explained in the text.

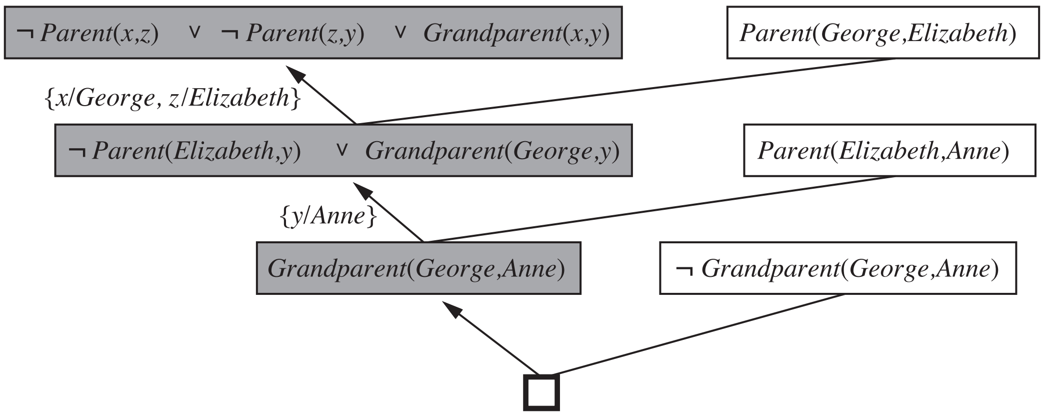

¬ Parent(x,z) ∨ ¬ Parent(z,y) ∨ Grandparent(x,y)
Parent(George,Elizabeth)
{x/George, z/Elizabeth}
¬ Parent(Elizabeth,y) ∨ Grandparent(George,y)
Parent(Elizabeth,Anne)
{y/Anne}
Grandparent(George,Anne)
¬ Grandparent(George,Anne)

Figure 20.13 Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications* (including negated *Classifications*).

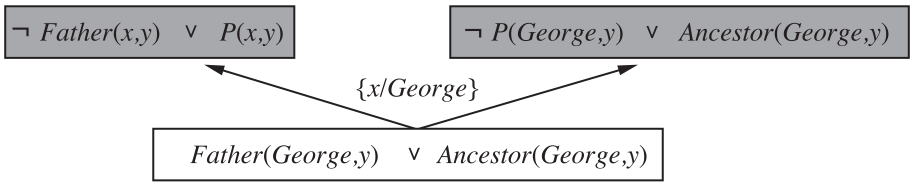

¬ Father(x,y) ∨ P(x,y)
¬ P(George,y) ∨ Ancestor(George,y)
{x/George}
Father(George,y) ∨ Ancestor(George,y)

Figure 20.14 An inverse resolution step that generates a new predicate *P*.

CHAPTER 21

LEARNING PROBABILISTIC MODELS

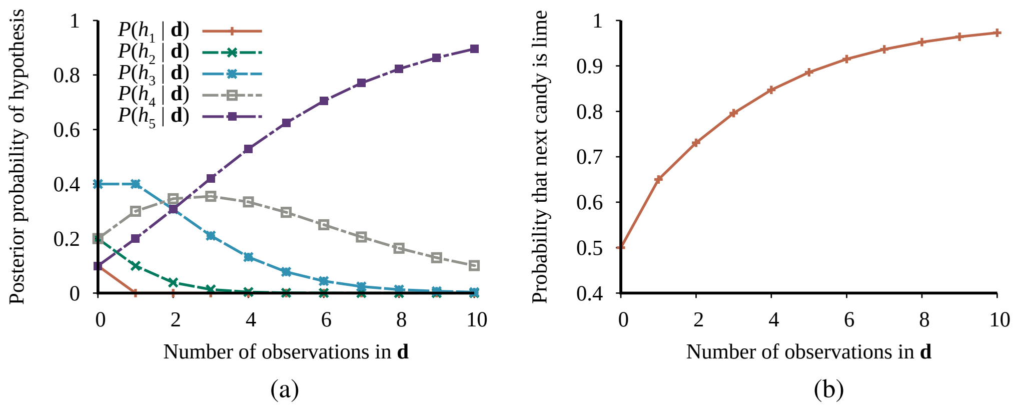


Figure 21.1 (a) Posterior probabilities $P(h_i \mid d_1, \ldots, d_N)$ from Equation (21.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(D_{N+1} = lime \mid d_1, \ldots, d_N)$ from Equation (21.2).

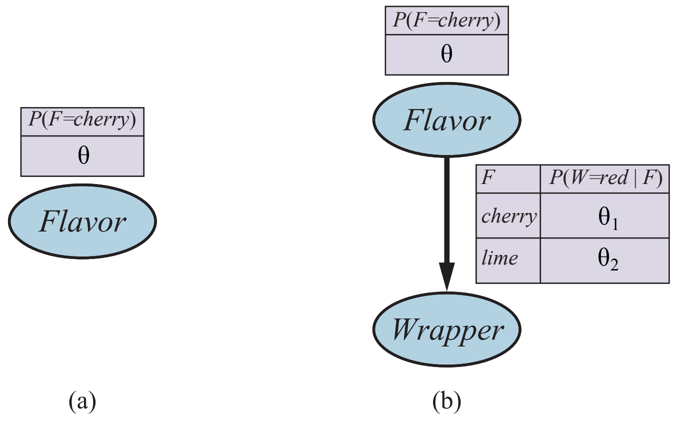


Figure 21.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherry and lime. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

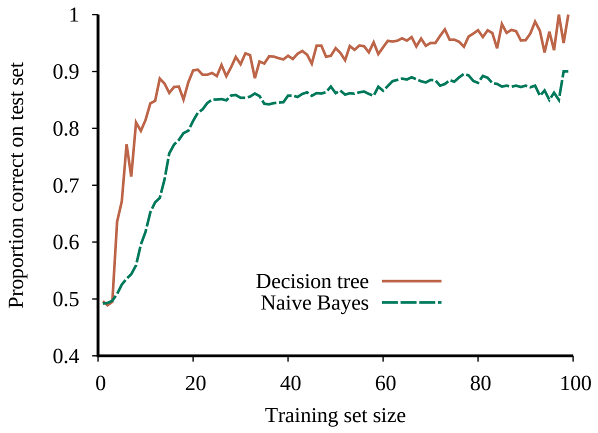
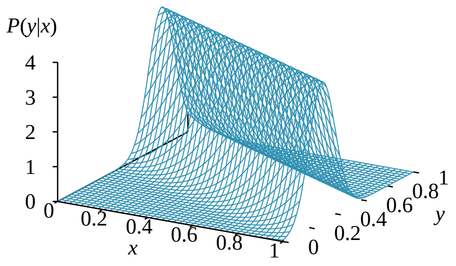
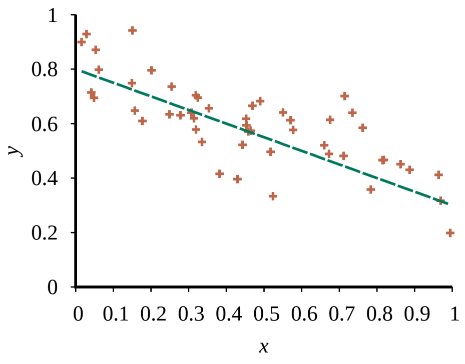


Figure 21.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 19; the learning curve for decision tree learning is shown for comparison.



(a)



(b)

Figure 21.4 (a) A linear–Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model and the best-fit line.

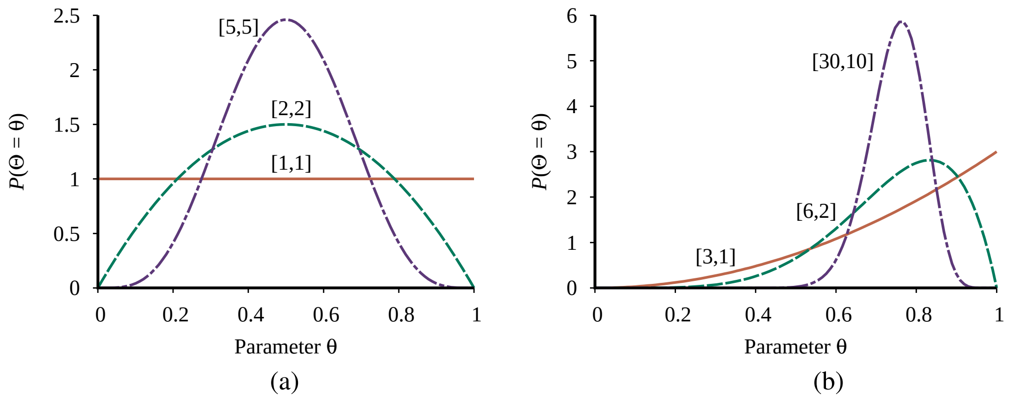


Figure 21.5 Examples of the *Beta*(a,b) distribution for different values of (a,b).

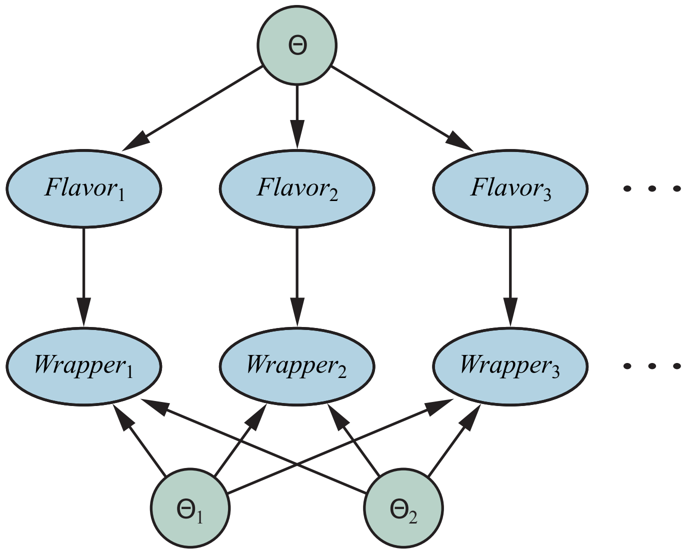


Figure 21.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in $Flavor_i$ and $Wrapper_i$.

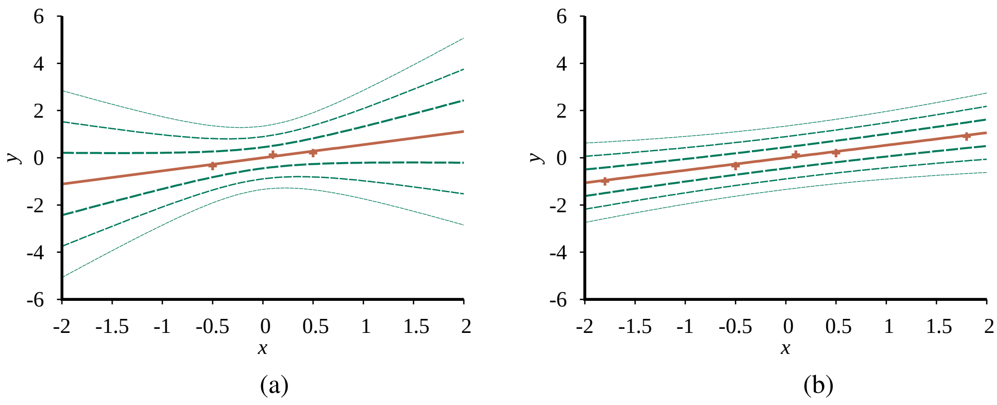


Figure 21.7 Bayesian linear regression with a model constrained to pass through the origin and fixed noise variance $\sigma^2 = 0.2$. Contours at ± 1, ± 2, and ± 3 standard deviations are shown for the predictive density. (a) With three data points near the origin, the slope is quite uncertain, with $\sigma_N^2 \approx 0.3861$. Notice how the uncertainty increases with distance from the observed data points. (b) With two additional data points further away, the slope θ is very tightly constrained, with $\sigma_N^2 \approx 0.0286$. The remaining variance in the predictive density is almost entirely due to the fixed noise σ^2.

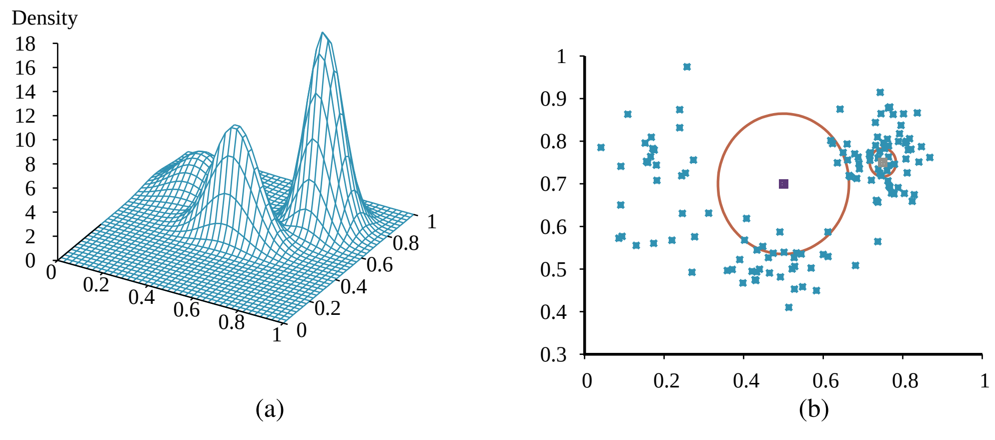


Figure 21.8 (a) A 3D plot of the mixture of Gaussians from Figure 21.12(a). (b) A 128-point sample of points from the mixture, together with two query points (small orange squares) and their 10-nearest-neighborhoods (large circle and smaller circle to the right).

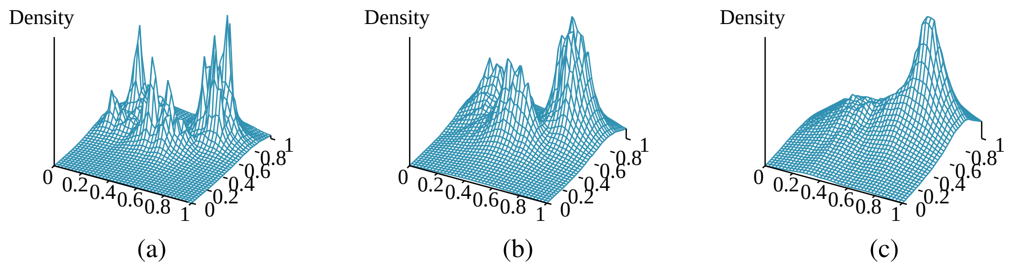


Figure 21.9 Density estimation using k-nearest-neighbors, applied to the data in Figure 21.8(b), for $k=3$, 10, and 40 respectively. $k = 3$ is too spiky, 40 is too smooth, and 10 is just about right. The best value for k can be chosen by cross-validation.

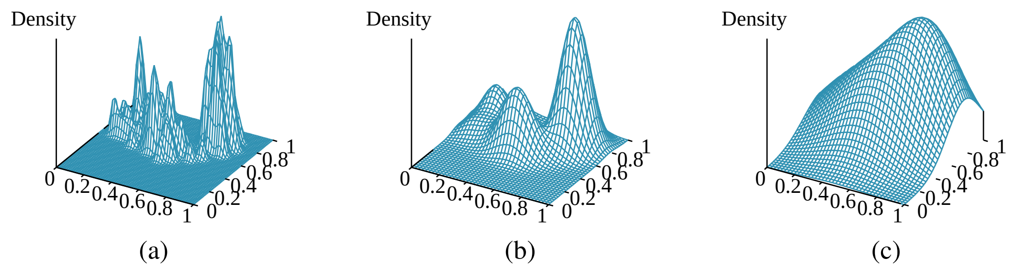


Figure 21.10 Density estimation using kernels for the data in Figure 21.8(b), using Gaussian kernels with $w=0.02$, 0.07, and 0.20 respectively. $w=0.07$ is about right.

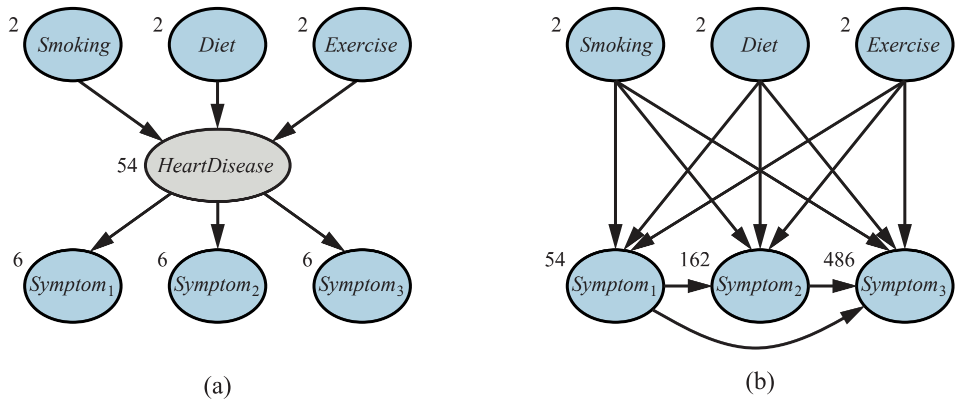
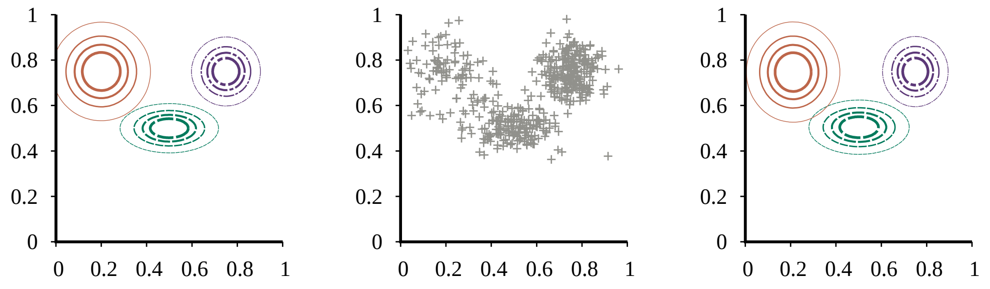
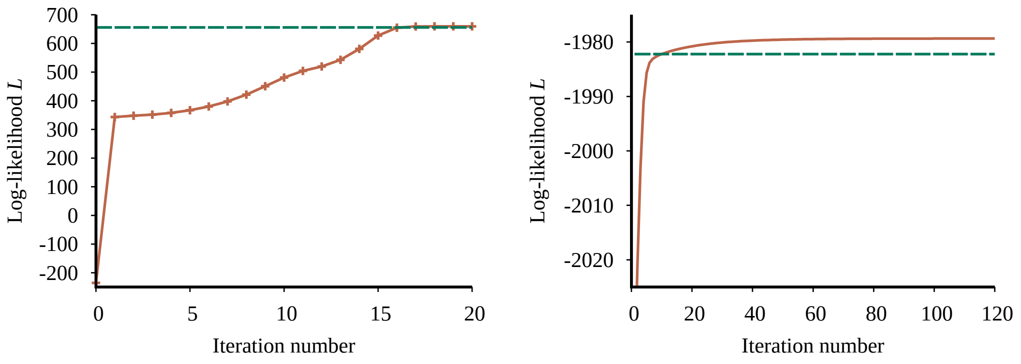


Figure 21.11 (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.



(a) (b) (c)

Figure 21.12 (a) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. (b) 500 data points sampled from the model in (a). (c) The model reconstructed by EM from the data in (b).



(a) (b)

Figure 21.13 Graphs showing the log likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 21.12. (b) Graph for the Bayesian network in Figure 21.14(a).

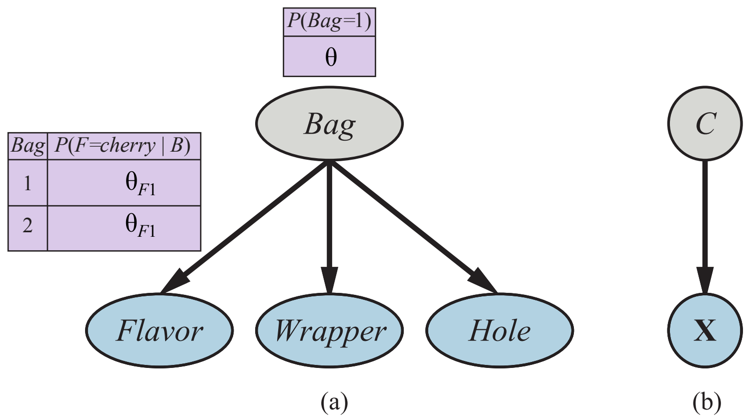


Figure 21.14 (a) A mixture model for candy. The proportions of different flavors, wrappers, and presence of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables **X** depend on the component C.

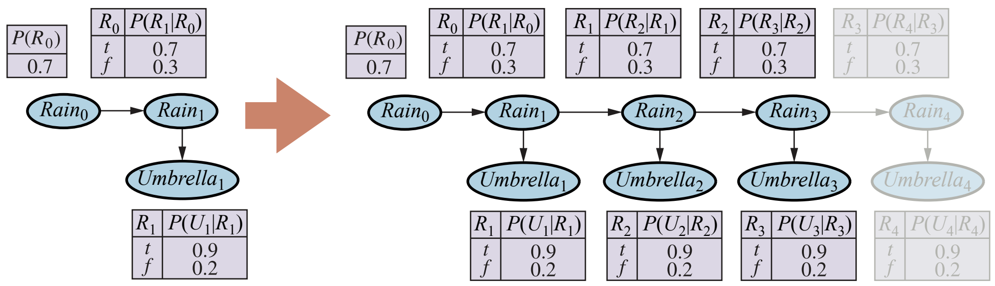

P(R0) 0.7
R0 | P(R1|R0): t 0.7, f 0.3
Rain0
Rain1
Umbrella1
R1 | P(U1|R1): t 0.9, f 0.2
R1 | P(R2|R1): t 0.7, f 0.3
R2 | P(R3|R2): t 0.7, f 0.3
R3 | P(R4|R3): t 0.7, f 0.3
Rain2
Rain3
Rain4
Umbrella2
Umbrella3
Umbrella4
R2 | P(U2|R2): t 0.9, f 0.2
R3 | P(U3|R3): t 0.9, f 0.2
R4 | P(U4|R4): t 0.9, f 0.2

Figure 21.15 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 14.16).

CHAPTER 22

DEEP LEARNING

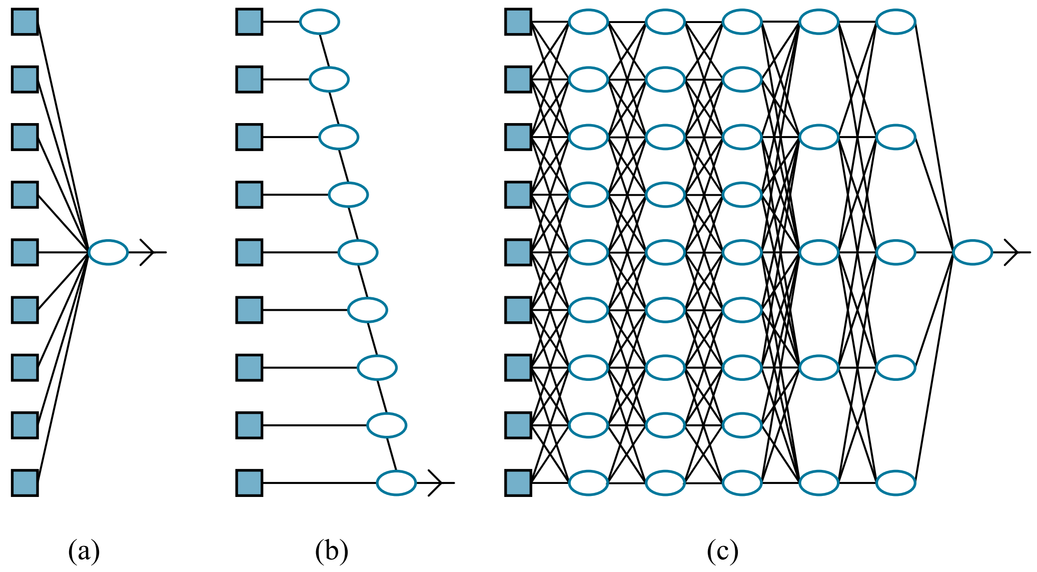


Figure 22.1 (a) A shallow model, such as linear regression, has short computation paths between inputs and output. (b) A decision list network (page 692) has some long paths for some possible input values, but most paths are short. (c) A deep learning network has longer computation paths, allowing each variable to interact with all the others.

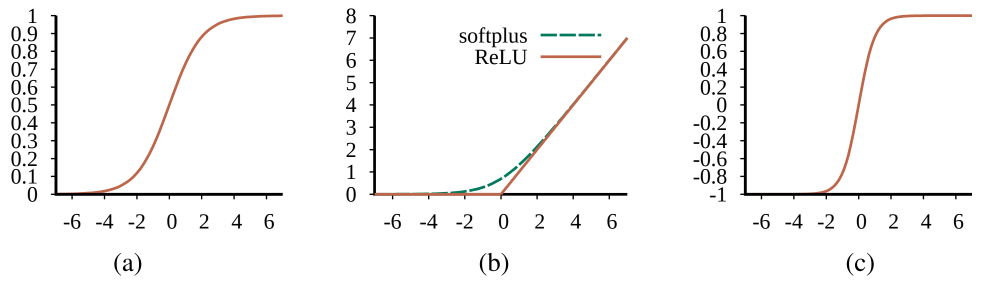


Figure 22.2 Activation functions commonly used in deep learning systems: (a) the logistic or sigmoid function; (b) the ReLU function and the softplus function; (c) the tanh function.

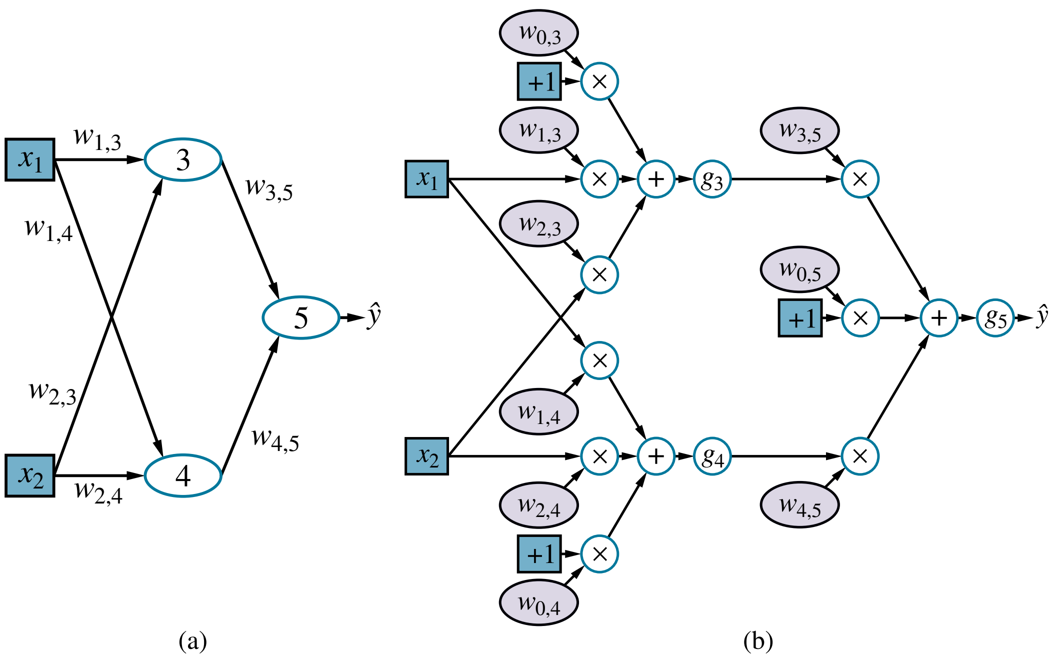


Figure 22.3 (a) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights. (b) The network in (a) unpacked into its full computation graph.

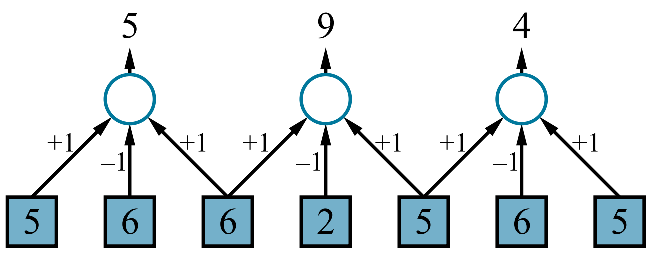


Figure 22.4 An example of a one-dimensional convolution operation with a kernel of size $l=3$ and a stride $s=2$. The peak response is centered on the darker (lower intensity) input pixel. The results would usually be fed through a nonlinear activation function (not shown) before going to the next hidden layer.

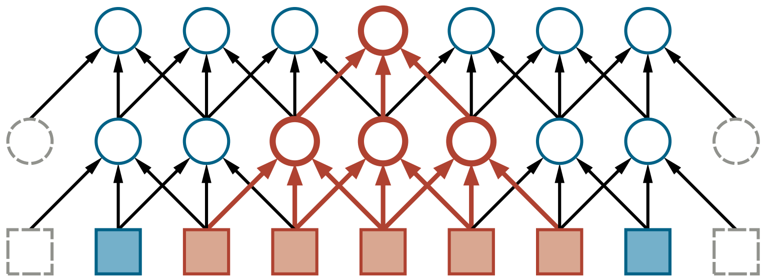
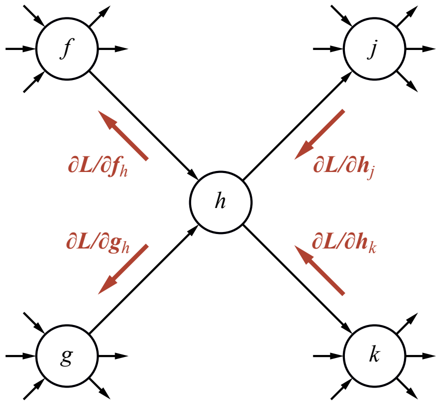


Figure 22.5 The first two layers of a CNN for a 1D image with a kernel size $l = 3$ and a stride $s = 1$. Padding is added at the left and right ends in order to keep the hidden layers the same size as the input. Shown in red is the receptive field of a unit in the second hidden layer. Generally speaking, the deeper the unit, the larger the receptive field.



f
j
$\partial L/\partial f_h$
$\partial L/\partial h_j$
h
$\partial L/\partial g_h$
$\partial L/\partial h_k$
g
k

Figure 22.6 Illustration of the back-propagation of gradient information in an arbitrary computation graph. The forward computation of the output of the network proceeds from left to right, while the back-propagation of gradients proceeds from right to left.

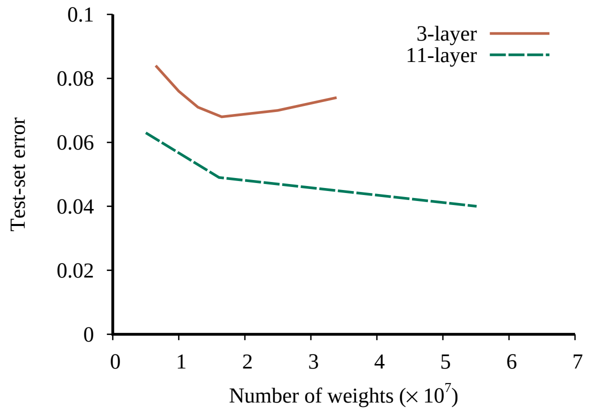


Figure 22.7 Test-set error as a function of layer width (as measured by total number of weights) for three-layer and eleven-layer convolutional networks. The data come from early versions of Google's system for transcribing addresses in photos taken by Street View cars (Goodfellow *et al.*, 2014).

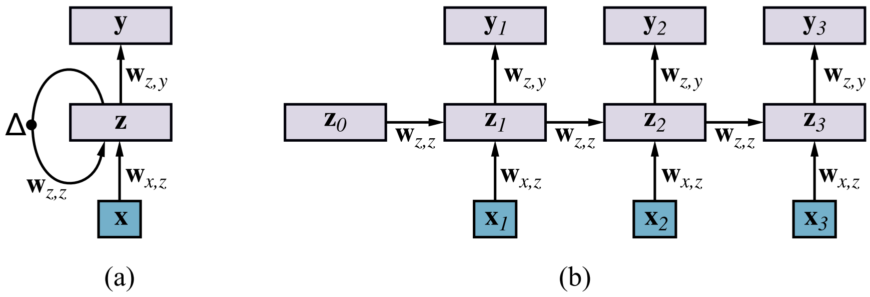


Figure 22.8 (a) Schematic diagram of a basic RNN where the hidden layer **z** has recurrent connections; the Δ symbol indicates a delay. (b) The same network unrolled over three time steps to create a feedforward network. Note that the weights are shared across all time steps.

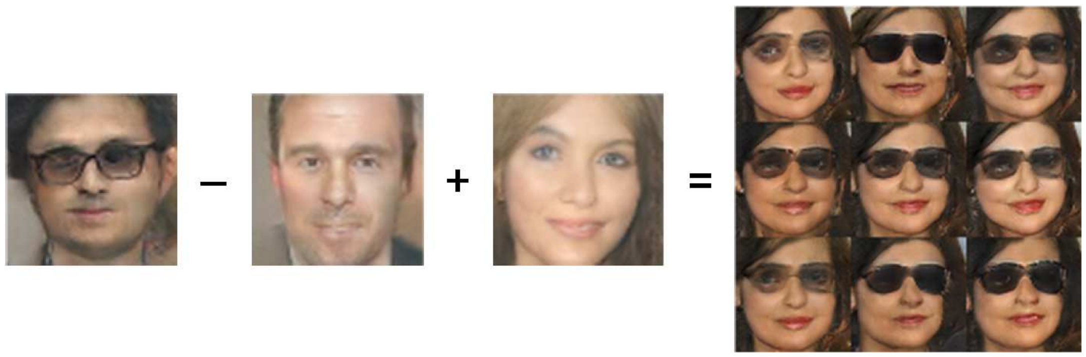


Figure 22.9 A demonstration of how a generative model has learned to use different directions in **z** space to represent different aspects of faces. We can actually perform arithmetic in **z** space. The images here are all generated from the learned model and show what happens when we decode different points in **z** space. We start with the coordinates for the concept of "man with glasses," subtract off the coordinates for "man," add the coordinates for "woman," and obtain the coordinates for "woman with glasses." Images reproduced with permission from (Radford *et al.*, 2015).

CHAPTER 23

REINFORCEMENT LEARNING

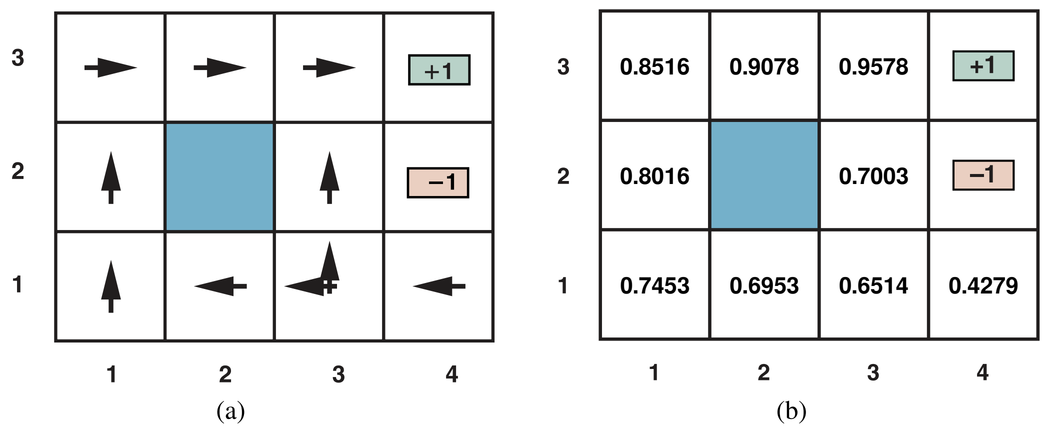


Figure 23.1 (a) The optimal policies for the stochastic environment with $R(s,a,s') = -0.04$ for transitions between nonterminal states. There are two policies because in state (3,1) both *Left* and *Up* are optimal. We saw this before in Figure 16.2. (b) The utilities of the states in the 4×3 world, given policy π.

```
function PASSIVE-ADP-LEARNER(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r$
  persistent: $\pi$, a fixed policy
              mdp, an MDP with model $P$, rewards $R$, actions $A$, discount $\gamma$
              $U$, a table of utilities for states, initially empty
              $N_{s'|s,a}$, a table of outcome count vectors indexed by state and action, initially zero
              $s$, $a$, the previous state and action, initially null

  if $s'$ is new then $U[s'] \leftarrow 0$
  if $s$ is not null then
    increment $N_{s'|s,a}[s,a][s']$
    $R[s, a, s'] \leftarrow r$
    add $a$ to $A[s]$
    $\mathbf{P}(\cdot \mid s,a) \leftarrow$ NORMALIZE($N_{s'|s,a}[s, a]$)
    $U \leftarrow$ POLICYEVALUATION($\pi, U, mdp$)
    $s, a \leftarrow s', \pi[s']$
    return $a$
```

Figure 23.2 A passive reinforcement learning agent based on adaptive dynamic programming. The agent chooses a value for γ and then incrementally computes the P and R values of the MDP. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 567.

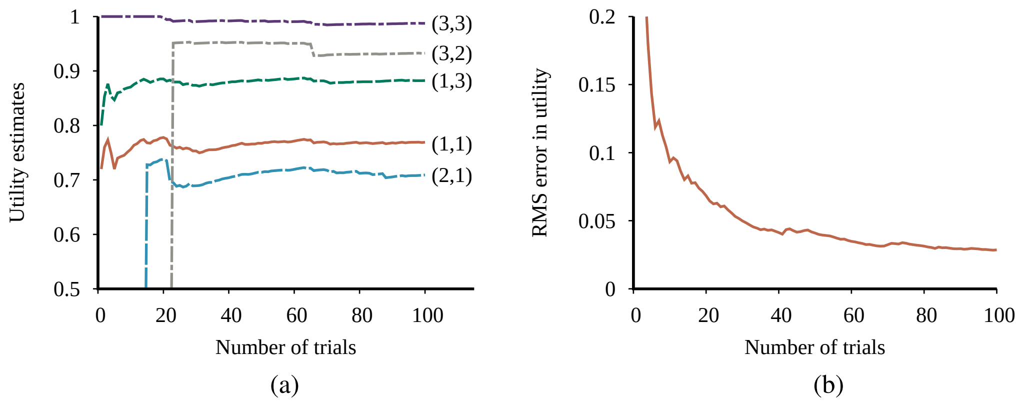


Figure 23.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 23.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice that it takes 14 and 23 trials respectively before the rarely visited states (2,1) and (3,2) "discover" that they connect to the +1 exit state at (4,3). (b) The root-mean-square error (see Appendix A) in the estimate for $U(1,1)$, averaged over 50 runs of 100 trials each.

```
function PASSIVE-TD-LEARNER(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r$
  persistent: $\pi$, a fixed policy
              $s$, the previous state, initially null
              $U$, a table of utilities for states, initially empty
              $N_s$, a table of frequencies for states, initially zero

  if $s'$ is new then $U[s'] \leftarrow 0$
  if $s$ is not null then
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s]) \times (r + \gamma U[s'] - U[s])$
  $s \leftarrow s'$
  return $\pi[s']$
```

Figure 23.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence.

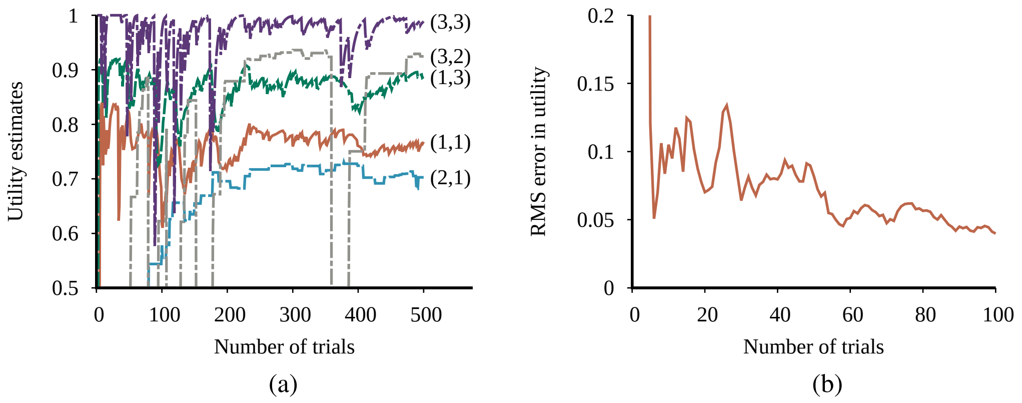
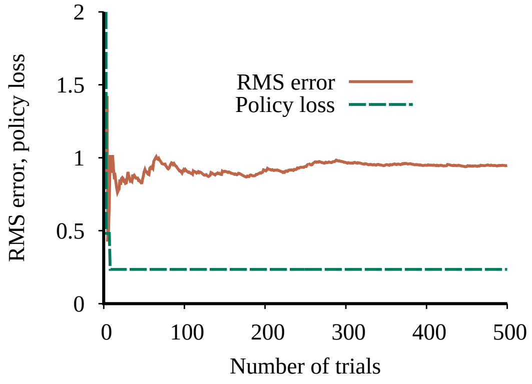
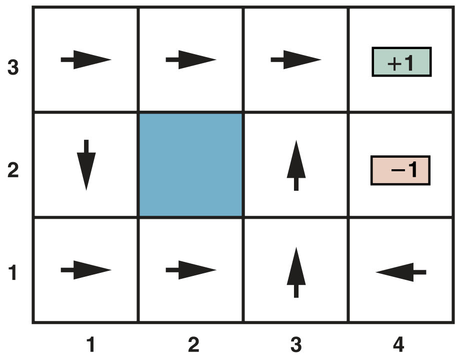


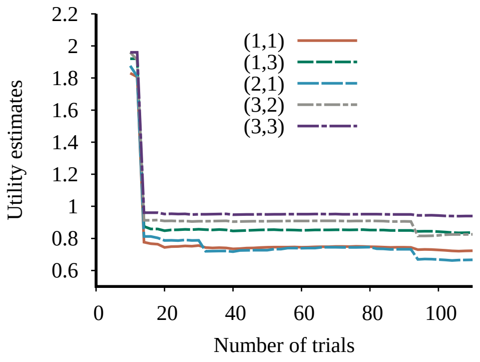
Figure 23.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials, for a single run of 500 trials. Compare with the run of 100 trials in Figure 23.3(a). (b) The root-mean-square error in the estimate for $U(1,1)$, averaged over 50 runs of 100 trials each.

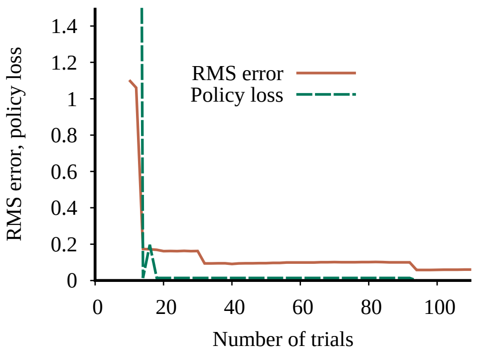




(a) (b)

Figure 23.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) The root mean square (RMS) error averaged across all nine nonterminal squares and the policy loss in (1,1). We see that the policy converges quickly, after just eight trials, to a suboptimal policy with a loss of 0.235. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials. Notice the *Down* action in (1,2).



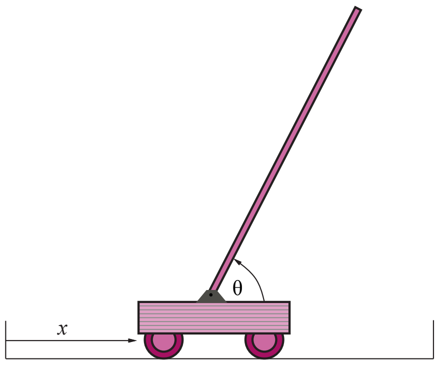


(a) (b)

Figure 23.7 Performance of the exploratory ADP agent using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state $s'$ and reward signal $r$
  persistent: $Q$, a table of action values indexed by state and action, initially zero
              $N_{sa}$, a table of frequencies for state–action pairs, initially zero
              $s$, $a$, the previous state and action, initially null

  if $s$ is not null then
     increment $N_{sa}[s,a]$
     $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a'])$
  return $a$
```

Figure 23.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s,a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model.



(a)

(b)

Figure 23.9 (a) Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes the cart's position x and velocity $\dot{x}$, as well as the pole's angle θ and rate of change of angle $\dot{\theta}$. (b) Six superimposed time-lapse images of a single autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy-search algorithm (Ng *et al.*, 2003). A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

CHAPTER 24

NATURAL LANGUAGE PROCESSING

Tag	**Word**	**Description**	**Tag**	**Word**	**Description**
CC	*and*	Coordinating conjunction	PRP$	*your*	Possessive pronoun
CD	*three*	Cardinal number	RB	*quickly*	Adverb
DT	*the*	Determiner	RBR	*quicker*	Adverb, comparative
EX	*there*	Existential there	RBS	*quickest*	Adverb, superlative
FW	*per se*	Foreign word	RP	*off*	Particle
IN	*of*	Preposition	SYM	+	Symbol
JJ	*purple*	Adjective	TO	*to*	to
JJR	*better*	Adjective, comparative	UH	*eureka*	Interjection
JJS	*best*	Adjective, superlative	VB	*talk*	Verb, base form
LS	*1*	List item marker	VBD	*talked*	Verb, past tense
MD	*should*	Modal	VBG	*talking*	Verb, gerund
NN	*kitten*	Noun, singular or mass	VBN	*talked*	Verb, past participle
NNS	*kittens*	Noun, plural	VBP	*talk*	Verb, non-3rd-sing
NNP	*Ali*	Proper noun, singular	VBZ	*talks*	Verb, 3rd-sing
NNPS	*Fords*	Proper noun, plural	WDT	*which*	Wh-determiner
PDT	*all*	Predeterminer	WP	*who*	Wh-pronoun
POS	*’s*	Possessive ending	WP$	*whose*	Possessive wh-pronoun
PRP	*you*	Personal pronoun	WRB	*where*	Wh-adverb
$	$	Dollar sign	#	#	Pound sign
“	‘	Left quote	”	’	Right quote
(	[	Left parenthesis	)	]	Right parenthesis
,	,	Comma	.	!	Sentence end
:	;	Mid-sentence punctuation			

Figure 24.1 Part-of-speech tags (with an example word for each tag) for the Penn Treebank corpus (Marcus *et al.*, 1993). Here “3rd-sing” is an abbreviation for “third person singular present tense.”

S	→	*NP VP*	[0.90]	I + feel a breeze
	|	*S Conj S*	[0.10]	I feel a breeze + and + It stinks
NP	→	*Pronoun*	[0.25]	I
	|	*Name*	[0.10]	Ali
	|	*Noun*	[0.10]	pits
	|	*Article Noun*	[0.25]	the + wumpus
	|	*Article Adjs Noun*	[0.05]	the + smelly dead + wumpus
	|	*Digit Digit*	[0.05]	3 4
	|	*NP PP*	[0.10]	the wumpus + in 1 3
	|	*NP RelClause*	[0.05]	the wumpus + that is smelly
	|	*NP Conj NP*	[0.05]	the wumpus + and + I
VP	→	*Verb*	[0.40]	stinks
	|	*VP NP*	[0.35]	feel + a breeze
	|	*VP Adjective*	[0.05]	smells + dead
	|	*VP PP*	[0.10]	is + in 1 3
	|	*VP Adverb*	[0.10]	go + ahead
Adjs	→	*Adjective*	[0.80]	smelly
	|	*Adjective Adjs*	[0.20]	smelly + dead
PP	→	*Prep NP*	[1.00]	to + the east
RelClause	→	*RelPro VP*	[1.00]	that + is smelly

Figure 24.2 The grammar for $\mathcal{E}_0$, with example phrases for each rule. The syntactic categories are sentence (*S*), noun phrase (*NP*), verb phrase (*VP*), list of adjectives (*Adjs*), prepositional phrase (*PP*), and relative clause (*RelClause*).

Noun	→	**stench** [0.05] | **breeze** [0.10] | **wumpus** [0.15] | **pits** [0.05] | ...
Verb	→	**is** [0.10] | **feel** [0.10] | **smells** [0.10] | **stinks** [0.05] | ...
Adjective	→	**right** [0.10] | **dead** [0.05] | **smelly** [0.02] | **breezy** [0.02]...
Adverb	→	**here** [0.05] | **ahead** [0.05] | **nearby** [0.02] | ...
Pronoun	→	**me** [0.10] | **you** [0.03] | **I** [0.10] | **it** [0.10] | ...
RelPro	→	**that** [0.40] | **which** [0.15] | **who** [0.20] | **whom** [0.02] | ...
Name	→	**Ali** [0.01] | **Bo** [0.01] | **Boston** [0.01] | ...
Article	→	**the** [0.40] | **a** [0.30] | **an** [0.10] | **every** [0.05] | ...
Prep	→	**to** [0.20] | **in** [0.10] | **on** [0.05] | **near** [0.10] | ...
Conj	→	**and** [0.50] | **or** [0.10] | **but** [0.20] | **yet** [0.02] | ...
Digit	→	**0** [0.20] | **1** [0.20] | **2** [0.20] | **3** [0.20] | **4** [0.20] | ...

Figure 24.3 The lexicon for $\mathcal{E}_0$. *RelPro* is short for relative pronoun, *Prep* for preposition, and *Conj* for conjunction. The sum of the probabilities for each category is 1.

List of items	Rule
S	
$NP\ VP$	$S \rightarrow NP\ VP$
$NP\ VP\ Adjective$	$VP \rightarrow VP\ Adjective$
$NP\ Verb\ Adjective$	$VP \rightarrow Verb$
$NP\ Verb$ **dead**	$Adjective \rightarrow$ **dead**
NP **is dead**	$Verb \rightarrow$ **is**
$Article\ Noun$ **is dead**	$NP \rightarrow Article\ Noun$
$Article$ **wumpus is dead**	$Noun \rightarrow$ **wumpus**
the wumpus is dead	$Article \rightarrow$ **the**

Figure 24.4 Parsing the string "The wumpus is dead" as a sentence, according to the grammar $\mathscr{E}_0$. Viewed as a top-down parse, we start with S, and on each step match one nonterminal X with a rule of the form $(X \rightarrow Y \ldots)$ and replace X in the list of items with $Y \ldots$; for example replacing S with the sequence $NP\ VP$. Viewed as a bottom-up parse, we start with the words "the wumpus is dead", and on each step match a string of tokens such as $(Y \ldots)$ against a rule of the form $(X \rightarrow Y \ldots)$ and replace the tokens with X; for example replacing "the" with *Article* or *Article Noun* with *NP*.

```
function CYK-PARSE($words$, $grammar$) returns a table of parse trees
  inputs: $words$, a list of words
          $grammar$, a structure with LEXICALRULES and GRAMMARRULES
  $T \leftarrow$ a table       // $T[X, i, k]$ is most probable $X$ tree spanning $words_{i:k}$
  $P \leftarrow$ a table, initially all 0       // $P[X, i, k]$ is probability of tree $T[X, i, k]$
  // Insert lexical categories for each word.
  for $i = 1$ to LEN($words$) do
    for each $(X, p)$ in $grammar$.LEXICALRULES($words_i$) do
      $P[X, i, i] \leftarrow p$
      $T[X, i, i] \leftarrow$ TREE($X$, $words_i$)
  // Construct $X_{i:k}$ from $Y_{i:j} + Z_{j+1:k}$, shortest spans first.
  for each $(i, j, k)$ in SUBSPANS(LEN($words$)) do
    for each $(X, Y, Z, p)$ in $grammar$.GRAMMARRULES do
      $PYZ \leftarrow P[Y, i, j] \times P[Z, j+1, k] \times p$
      if $PYZ > P[X, i, k]$ do
        $P[X, i, k] \leftarrow PYZ$
        $T[X, i, k] \leftarrow$ TREE($X$, $T[Y, i, j]$, $T[Z, j+1, k]$)
  return $T$

function SUBSPANS(N) yields $(i, j, k)$ tuples
  for $length = 2$ to $N$ do
    for $i = 1$ to $N + 1 - length$ do
      $k \leftarrow i + length - 1$
      for $j = i$ to $k - 1$ do
        yield $(i, j, k)$
```

Figure 24.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable parse tree for the sequence and its subsequences. The table $P[X,i,k]$ gives the probability of the most probable tree of category X spanning $words_{i:k}$. The output table $T[X, i, k]$ contains the most probable tree of category X spanning positions i to k inclusive. The function SUBSPANS returns all tuples (i,j,k) covering a span of $words_{i:k}$, with $i \leq j < k$, listing the tuples by increasing length of the $i:k$ span, so that when we go to combine two shorter spans into a longer one, the shorter spans are already in the table. LEXICALRULES($word$) returns a collection of (X, p) pairs, one for each rule of the form $X \rightarrow word$ [p], and GRAMMARRULES gives (X,Y,Z,p) tuples, one for each grammar rule of the form $X \rightarrow Y\ Z\ [p]$.

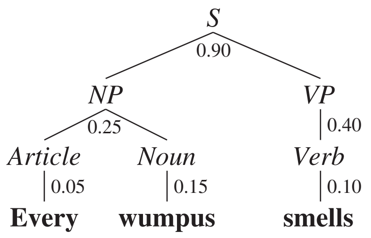


Figure 24.6 Parse tree for the sentence "Every wumpus smells" according to the grammar $\mathcal{E}_0$. Each interior node of the tree is labeled with its probability. The probability of the tree as a whole is $0.9 \times 0.25 \times 0.05 \times 0.15 \times 0.40 \times 0.10 = 0.0000675$. The tree can also be written in linear form as [S [NP [$Article$ **every**] [$Noun$ **wumpus**]][VP [$Verb$ **smells**]]].

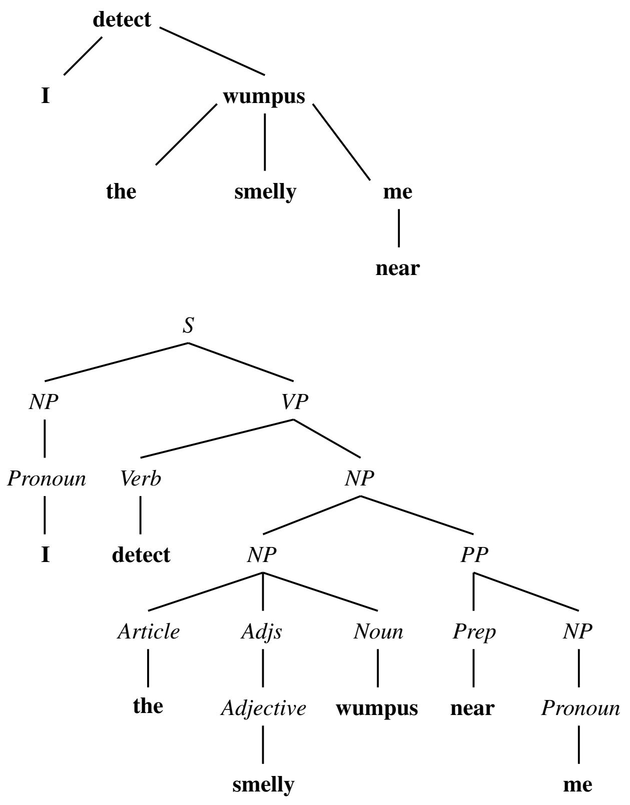


Figure 24.7 A dependency-style parse (top) and the corresponding phrase structure parse (bottom) for the sentence *I detect the smelly wumpus near me.*

```
[ [S [NP-2 Her eyes]
     [VP were
          [VP glazed
               [NP *-2]
               [SBAR-ADV as if
                      [S [NP she]
                          [VP did n't
                             [VP [VP hear [NP *-1]]
                                  or
                                  [VP [ADVP even] see [NP *-1]]
                                  [NP-1 him]]]]]]]]
  .]
```

Figure 24.8 Annotated tree for the sentence "Her eyes were glazed as if she didn't hear or even see him." from the Penn Treebank. Note a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase "hear or even see him" as consisting of two constituent *VP*s, [*VP* **hear** [*NP* *-1]] and [*VP* [*ADVP* **even**] **see** [*NP* *-1]], both of which have a missing object, denoted *-1, which refers to the *NP* labeled elsewhere in the tree as [*NP*-1 **him**]. Similarly, the [*NP* *-2] refers to the [*NP*-2 **Her eyes**].

$$
\begin{array}{rcl}
S(v) & \rightarrow & NP(Sbj,pn,n)\ VP(pn,v) \mid \ldots \\
NP(c,pn,n) & \rightarrow & Pronoun(c,pn,n) \mid Noun(c,pn,n) \mid \ldots \\
VP(pn,v) & \rightarrow & Verb(pn,v)\ NP(Obj,pn,n) \mid \ldots \\
PP(head) & \rightarrow & Prep(head)\ NP(Obj,pn,h) \\
Pronoun(Sbj,1S,\mathbf{I}) & \rightarrow & \mathbf{I} \\
Pronoun(Sbj,1P,\mathbf{we}) & \rightarrow & \mathbf{we} \\
Pronoun(Obj,1S,\mathbf{me}) & \rightarrow & \mathbf{me} \\
Pronoun(Obj,3P,\mathbf{them}) & \rightarrow & \mathbf{them} \\
 & & \\
Verb(3S,\mathbf{see}) & \rightarrow & \mathbf{see}
\end{array}
$$

Figure 24.9 Part of an augmented grammar that handles case agreement, subject–verb agreement, and head words. Capitalized names are constants: *Sbj*, and *Obj* for subjective and objective case; *1S* for first person singular; *1P* and *3P* for first and third person plural. As usual, lowercase names are variables. For simplicity, the probabilities have been omitted.

$$
\begin{aligned}
&Exp(op(x_1,x_2)) \rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\\
&Exp(x) \rightarrow \textbf{(}\ Exp(x)\ \textbf{)}\\
&Exp(x) \rightarrow Number(x)\\
&Number(x) \rightarrow Digit(x)\\
&Number(10 \times x_1 + x_2) \rightarrow Number(x_1)\ Digit(x_2)\\
&Operator(+) \rightarrow \textbf{+}\\
&Operator(-) \rightarrow \textbf{-}\\
&Operator(\times) \rightarrow \times\\
&Operator(\div) \rightarrow \div\\
&Digit(0) \rightarrow \textbf{0}\\
&Digit(1) \rightarrow \textbf{1}\\
&\ldots
\end{aligned}
$$

Figure 24.10 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent.

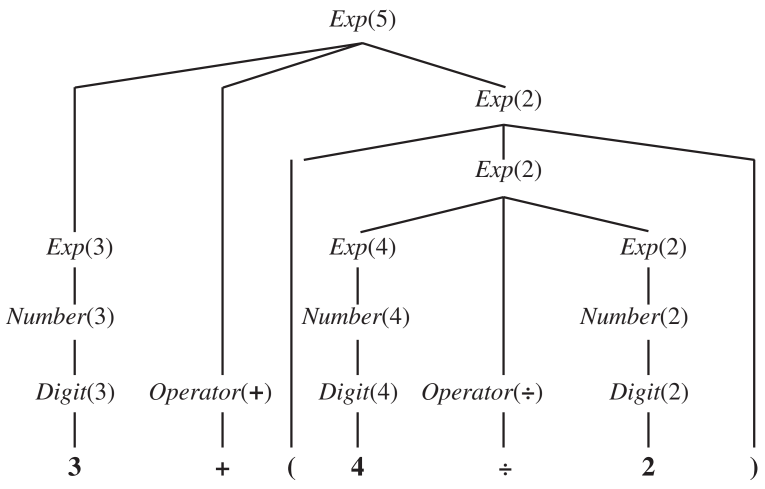
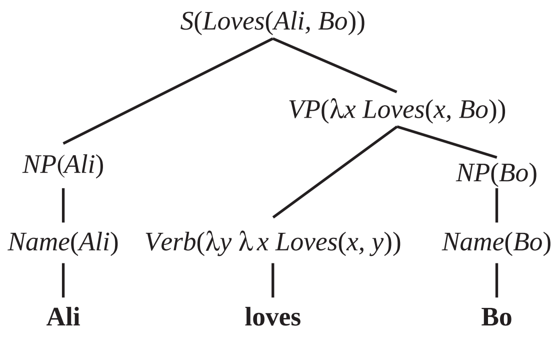


Figure 24.11 Parse tree with semantic interpretations for the string "$3+(4\div2)$".

$$S(pred(n)) \rightarrow NP(n)\ VP(pred)$$
$$VP(pred(n)) \rightarrow Verb(pred)\ NP(n)$$
$$NP(n) \rightarrow Name(n)$$

$$Name(Ali) \rightarrow \textbf{Ali}$$
$$Name(Bo) \rightarrow \textbf{Bo}$$
$$Verb(\lambda y\ \lambda x\ Loves(x,y)) \rightarrow \textbf{loves}$$

(a)



(b)

Figure 24.12 (a) A grammar that can derive a parse tree and semantic interpretation for "Ali loves Bo" (and three other sentences). Each category is augmented with a single argument representing the semantics. (b) A parse tree with semantic interpretations for the string "Ali loves Bo."

CHAPTER 25

DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

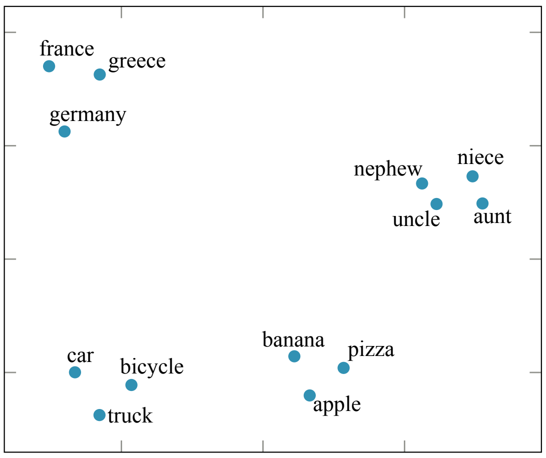


Figure 25.1 Word embedding vectors computed by the GloVe algorithm trained on 6 billion words of text. 100-dimensional word vectors are projected down onto two dimensions in this visualization. Similar words appear near each other.

A	**B**	**C**	$\mathbf{D} = \mathbf{C} + (\mathbf{B} - \mathbf{A})$	**Relationship**
Athens	Greece	Oslo	Norway	*Capital*
Astana	Kazakhstan	Harare	Zimbabwe	*Capital*
Angola	kwanza	Iran	rial	*Currency*
copper	Cu	gold	Au	*Atomic Symbol*
Microsoft	Windows	Google	Android	*Operating System*
New York	New York Times	Baltimore	Baltimore Sun	*Newspaper*
Berlusconi	Silvio	Obama	Barack	*First name*
Switzerland	Swiss	Cambodia	Cambodian	*Nationality*
Einstein	scientist	Picasso	painter	*Occupation*
brother	sister	grandson	granddaughter	*Family Relation*
Chicago	Illinois	Stockton	California	*State*
possibly	impossibly	ethical	unethical	*Negative*
mouse	mice	dollar	dollars	*Plural*
easy	easiest	lucky	luckiest	*Superlative*
walking	walked	swimming	swam	*Past tense*

Figure 25.2 A word embedding model can sometimes answer the question "**A** is to **B** as **C** is to [what]?" with vector arithmetic: given the word embedding vectors for the words **A**, **B**, and **C**, compute the vector $\mathbf{D} = \mathbf{C} + (\mathbf{B} - \mathbf{A})$ and look up the word that is closest to **D**. (The answers in column **D** were computed automatically by the model. The descriptions in the "Relationship" column were added by hand.) Adapted from Mikolov *et al.* (2013, 2014).

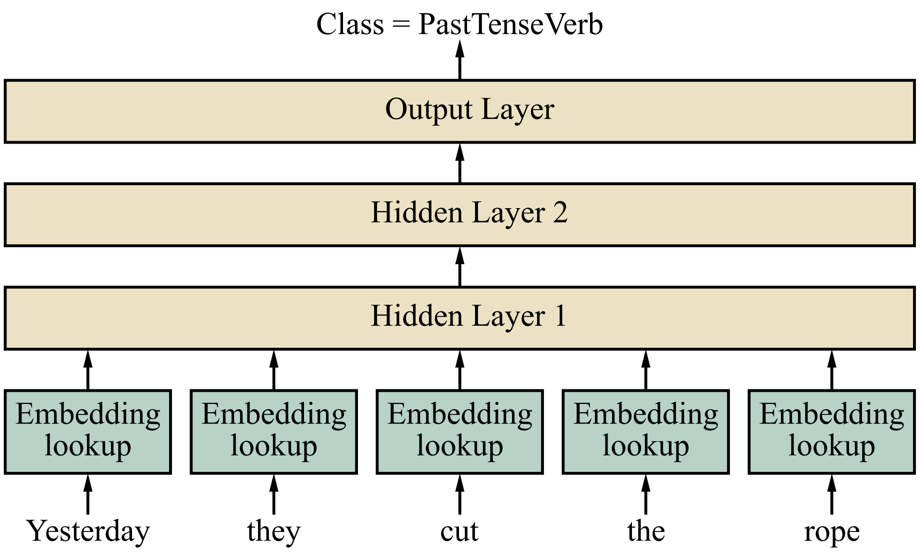


Figure 25.3 Feedforward part-of-speech tagging model. This model takes a 5-word window as input and predicts the tag of the word in the middle—here, *cut*. The model is able to account for word position because each of the 5 input embeddings is multiplied by a different part of the first hidden layer. The parameter values for the word embeddings and for the three layers are all learned simultaneously during training.

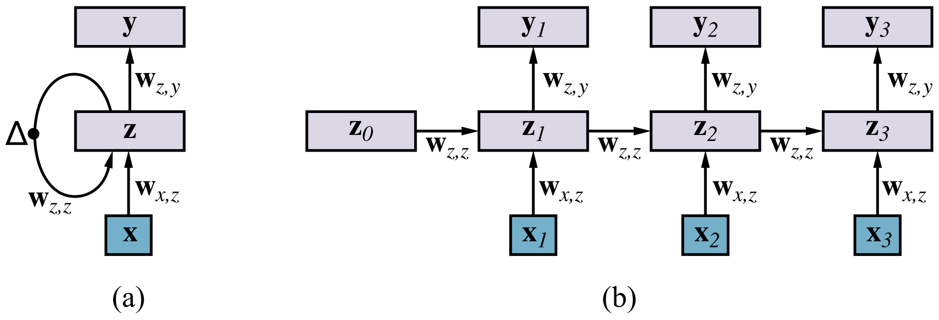


Figure 25.4 (a) Schematic diagram of an RNN where the hidden layer **z** has recurrent connections; the Δ symbol indicates a delay. Each input **x** is the word embedding vector of the next word in the sentence. Each output **y** is the output for that time step. (b) The same network unrolled over three timesteps to create a feedforward network. Note that the weights are shared across all timesteps.

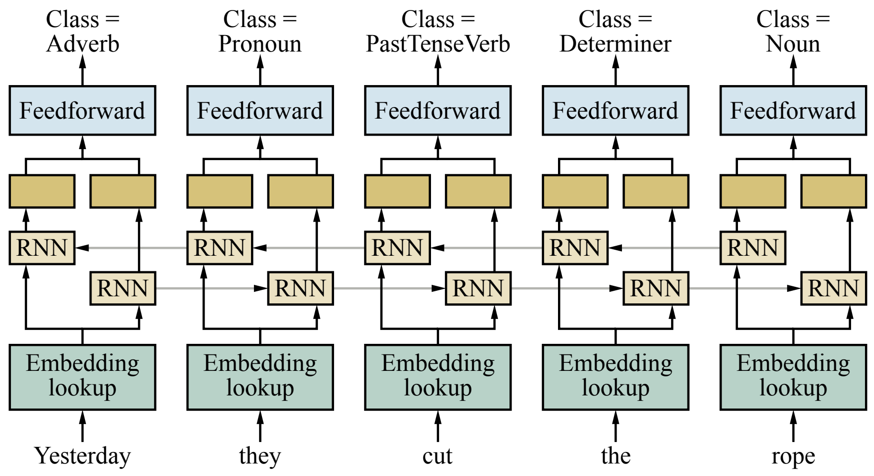


Figure 25.5 A bidirectional RNN network for POS tagging.

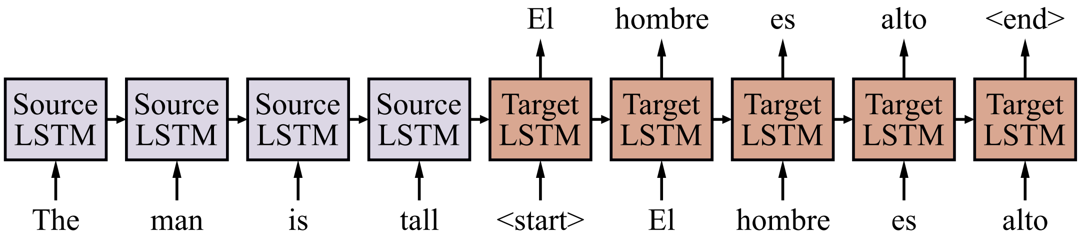


Figure 25.6 Basic sequence-to-sequence model. Each block represents one LSTM timestep. (For simplicity, the embedding and output layers are not shown.) On successive steps we feed the network the words of the source sentence "The man is tall," followed by the `<start>` tag to indicate that the network should start producing the target sentence. The final hidden state at the end of the source sentence is used as the hidden state for the start of the target sentence. After that, each target sentence word at time t is used as input at time $t+1$, until the network produces the `<end>` tag to indicate that sentence generation is finished.

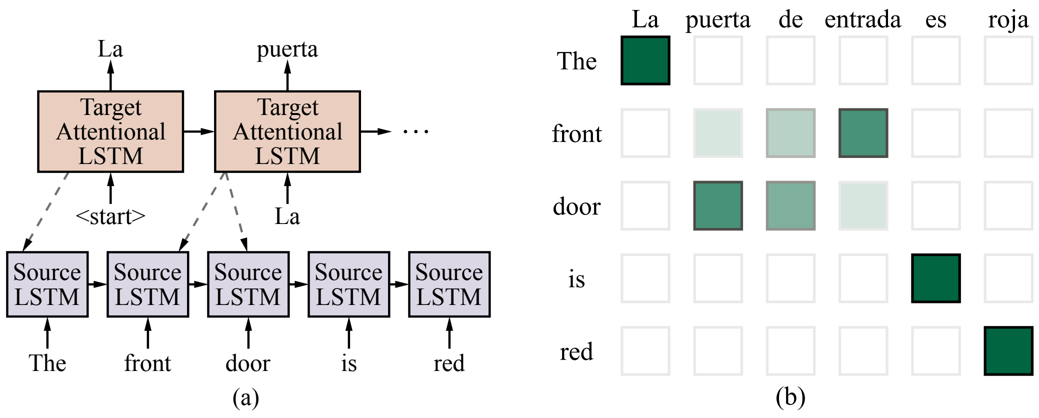
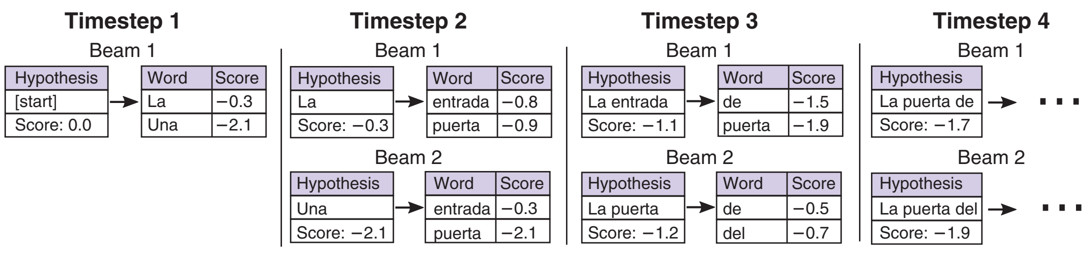


Figure 25.7 (a) Attentional sequence-to-sequence model for English-to-Spanish translation. The dashed lines represent attention. (b) Example of attention probability matrix for a bilingual sentence pair, with darker boxes representing higher values of a_{ij}. The attention probabilities sum to one over each column.



Timestep 1, Beam 1

Hypothesis
[start]
Score: 0.0

Word	Score
La	−0.3
Una	−2.1

Timestep 2, Beam 1

Hypothesis
La
Score: −0.3

Word	Score
entrada	−0.8
puerta	−0.9

Timestep 2, Beam 2

Hypothesis
Una
Score: −2.1

Word	Score
entrada	−0.3
puerta	−2.1

Timestep 3, Beam 1

Hypothesis
La entrada
Score: −1.1

Word	Score
de	−1.5
puerta	−1.9

Timestep 3, Beam 2

Hypothesis
La puerta
Score: −1.2

Word	Score
de	−0.5
del	−0.7

Timestep 4, Beam 1

Hypothesis
La puerta de
Score: −1.7

Timestep 4, Beam 2

Hypothesis
La puerta del
Score: −1.9

Figure 25.8 Beam search with beam size of $b = 2$. The score of each word is the log-probability generated by the target RNN softmax, and the score of each hypothesis is the sum of the word scores. At timestep 3, the highest scoring hypothesis *La entrada* can only generate low-probability continuations, so it "falls off the beam."

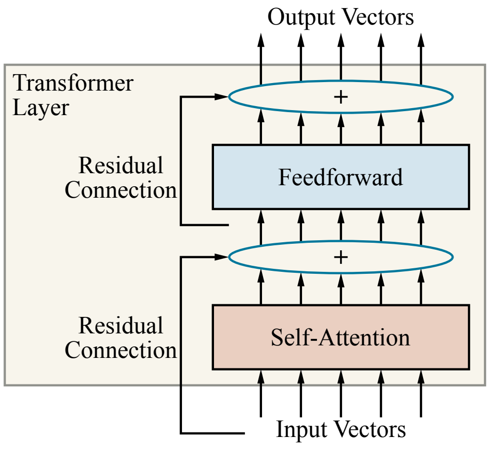


Figure 25.9 A single-layer transformer consists of self-attention, a feedforward network, and residual connections.

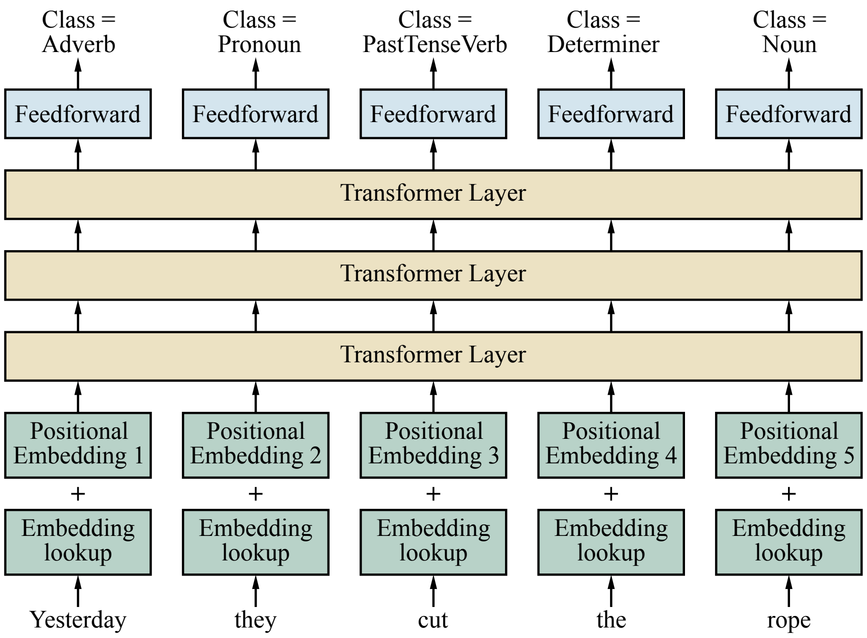


Figure 25.10 Using the transformer architecture for POS tagging.

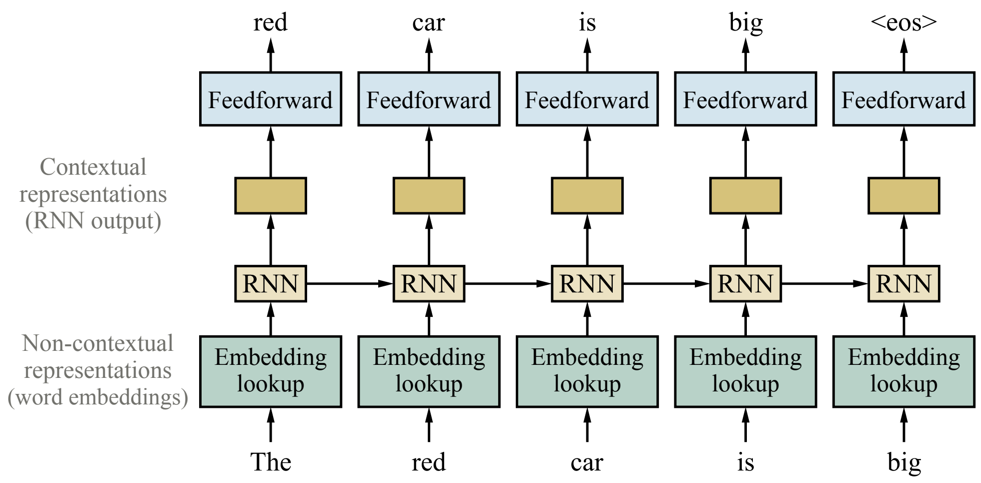

red
car
is
big
<eos>
Feedforward
Contextual representations (RNN output)
RNN
Non-contextual representations (word embeddings)
Embedding lookup
The
red
car
is
big

Figure 25.11 Training contextual representations using a left-to-right language model.

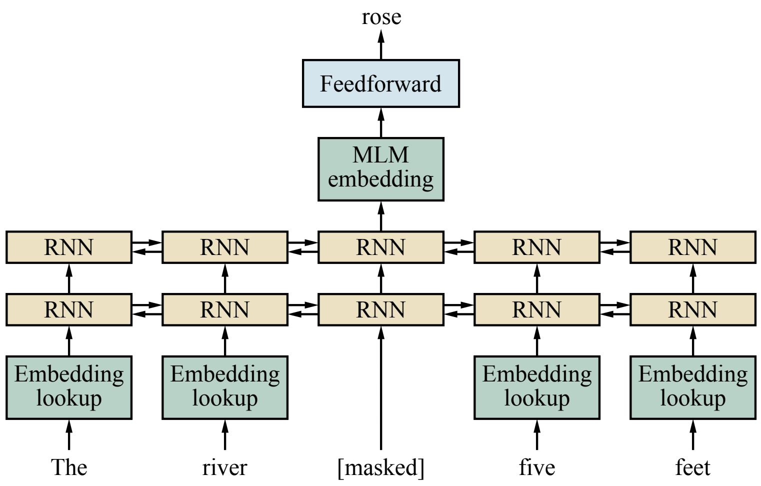


Figure 25.12 Masked language modeling: pretrain a bidirectional model—for example, a multilayer RNN—by masking input words and predicting only those masked words.

1. **What will best separate a mixture of iron filings and black pepper?**
   (a) magnet (b) filter paper (c) triple beam balance (d) voltmeter
2. **Which form of energy is produced when a rubber band vibrates?**
   (a) chemical (b) light (c) electrical (d) sound
3. **Because copper is a metal, it is**
   (a) liquid at room temperature (b) nonreactive with other substances
   (c) a poor conductor of electricity (d) a good conductor of heat
4. **Which process in an apple tree primarily results from cell division?**
   (a) growth (b) photosynthesis (c) gas exchange (d) waste removal

Figure 25.13 Questions from an 8th grade science exam that the ARISTO system can answer correctly using an ensemble of methods, with the most influential being a ROBERTA language model. Answering these questions requires knowledge about natural language, the structure of multiple-choice tests, commonsense, and science.

It is a truth universally acknowledged that the earth is not the center of the universe. There are those who assert there is. I do not accept them, but others I consider to be of the same opinion. The truth is, however, that if there are other than the center, and if there are any other living things in the universe and if they are not human, then we do not yet have our answers. We have to go on. This page gives a simplified, simplified answer to the problem. We don't have all the answers. The truth is, however, that the truth is out there.

When Gregor Samsa woke up one morning, he did not notice anything strange. "When my wife is looking at me, I feel like she is looking at a piece of art," he said. "I think she is admiring something I have created." The idea is that by looking at your own life, you learn something important and become a better person. It is a theory that emerged from psychologist Daniel Goleman's work, in which he asked "How do you know you're not a loser?"

Alice was beginning to get very tired of sitting with her sister on the bank. She sat up, yawned, and said, with a loud little scream, "I hope you don't mind if I keep on doing what I should like to do, and if someone asks me which of us will do more, don't tell them that I won't do much, my dear sister."

All happy families are alike; each happy family is like a garden of paradise. The only difference between happy families and unhappy families, is that the unhappy family doesn't have any flowers or trees.

Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Please fill out the following details. Thank you... Thank you for your interest in this interview. Please wait...

Figure 25.14 Example completion texts generated by the GPT-2 language model, given the prompts in **bold**. Most of the texts are quite fluent English, at least locally. The final example demonstrates that sometimes the model just breaks down.

CHAPTER 26

ROBOTICS

(a)

(b)

Figure 26.1 (a) An industrial robotic arm with a custom end-effector. Image credit: Macor/123RF. (b) A Kinova® JACO® Assistive Robot arm mounted on a wheelchair. Kinova and JACO are trademarks of Kinova, Inc.

Figure 26.2 (a) NASA's Curiosity rover taking a selfie on Mars. Image courtesy of NASA. (b) A Skydio drone accompanying a family on a bike ride. Image courtesy of Skydio.

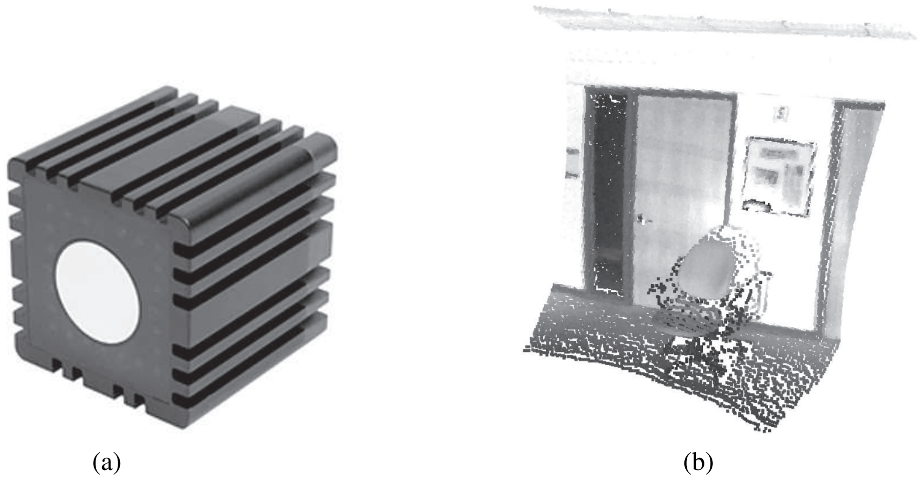


Figure 26.3 (a) Time-of-flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot's vicinity. Image courtesy of Willow Garage, LLC.

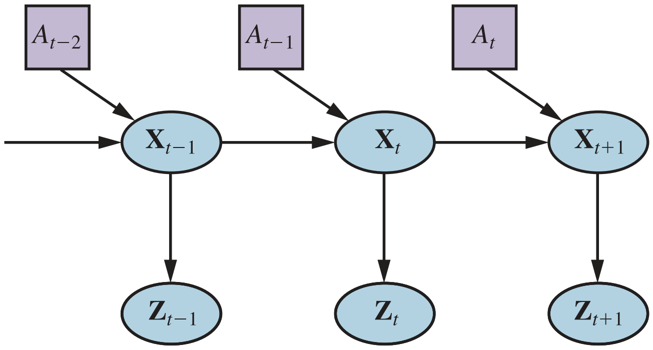


Figure 26.4 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic decision network.

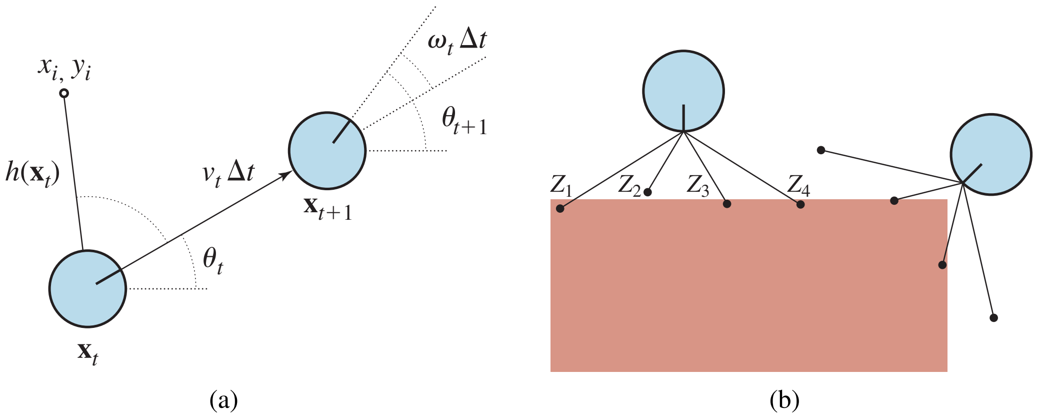


Figure 26.5 (a) A simplified kinematic model of a mobile robot. The robot is shown as a circle with an interior radius line marking the forward direction. The state $\mathbf{x}_t$ consists of the (x_t, y_t) position (shown implicitly) and the orientation θ_t. The new state $\mathbf{x}_{t+1}$ is obtained by an update in position of $v_t\Delta_t$ and in orientation of $\omega_t\Delta_t$. Also shown is a landmark at (x_i, y_i) observed at time t. (b) The range-scan sensor model. Two possible robot poses are shown for a given range scan (z_1, z_2, z_3, z_4). It is much more likely that the pose on the left generated the range scan than the pose on the right.

```
function MONTE-CARLO-LOCALIZATION$a$, $z$, $N$, $P(X'|X,\ v,\ \omega)$, $P(z|z^*)$, $map$
  returns a set of samples, $S$, for the next time step
  inputs: $a$, robot velocities $v$ and $\omega$
          $z$, a vector of $M$ range scan data points
          $P(X'|X,\ v,\ \omega)$, motion model
          $P(z|z^*)$, a range sensor noise model
          $map$, a 2D map of the environment
  persistent: $S$, a vector of $N$ samples
  local variables: $W$, a vector of $N$ weights
                   $S'$, a temporary vector of $N$ samples

  if $S$ is empty then
     for $i = 1$ to $N$ do        // initialization phase
        $S[i] \leftarrow$ sample from $P(X_0)$
  for $i = 1$ to $N$ do           // update cycle
     $S'[i] \leftarrow$ sample from $P(X'|X = S[i], v, \omega)$
     $W[i] \leftarrow 1$
     for $j = 1$ to $M$ do
        $z^* \leftarrow$ RAYCAST($j, X = S'[i]$, $map$)
        $W[i] \leftarrow W[i]\ \cdot\ P(z_j|\ z^*)$
     $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT($N$, $S'$, $W$)
  return $S$
```

Figure 26.6 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

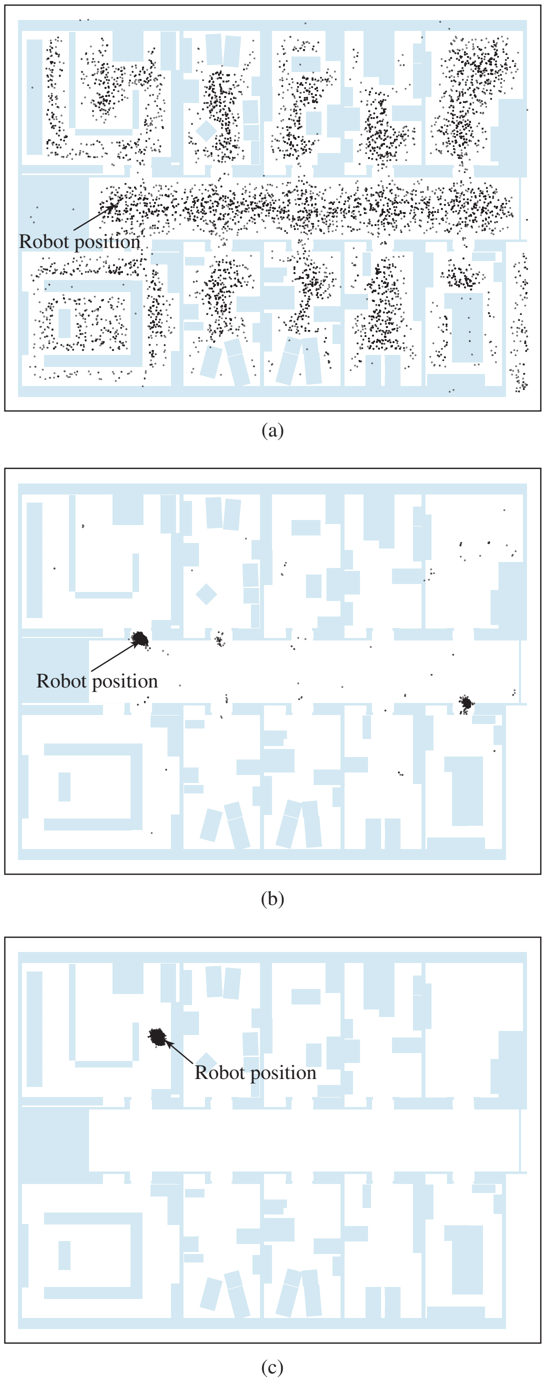


Figure 26.7 Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.

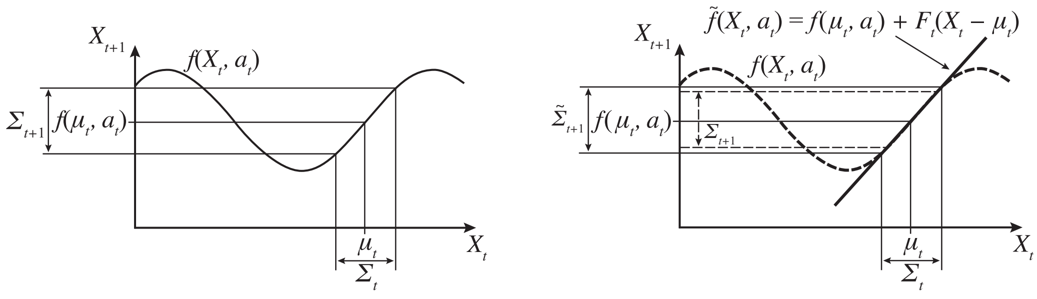


Figure 26.8 One-dimensional illustration of a linearized motion model: (a) The function f, and the projection of a mean μ_t and a covariance interval (based on Σ_t) into time $t+1$. (b) The linearized version is the tangent of f at μ_t. The projection of the mean μ_t is correct. However, the projected covariance $\tilde{\Sigma}_{t+1}$ differs from Σ_{t+1}.

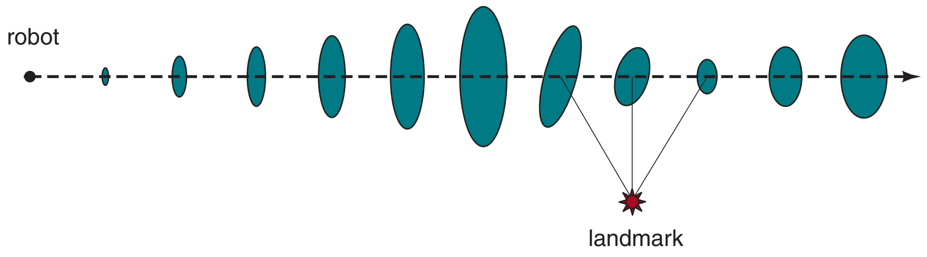


Figure 26.9 Localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty in its location estimate increases, as illustrated by the error ellipses. When it observes a landmark with known position, the uncertainty is reduced.

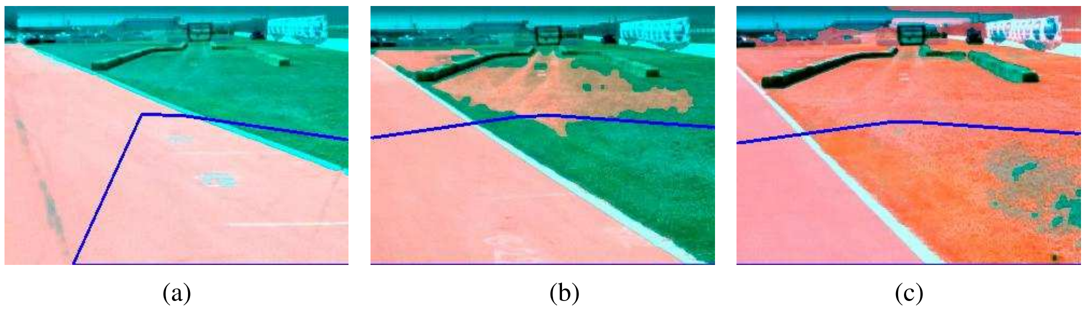


Figure 26.10 Sequence of "drivable surface" classifications using adaptive vision. (a) Only the road is classified as drivable (pink area). The V-shaped blue line shows where the vehicle is heading. (b) The vehicle is commanded to drive off the road, and the classifier is beginning to classify some of the grass as drivable. (c) The vehicle has updated its model of drivable surfaces to correspond to grass as well as road. Courtesy of Sebastian Thrun.

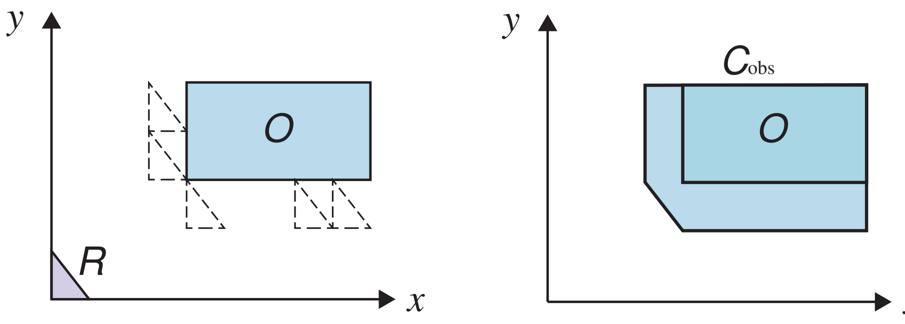


Figure 26.11 A simple triangular robot that can translate, and needs to avoid a rectangular obstacle. On the left is the workspace, on the right is the configuration space.

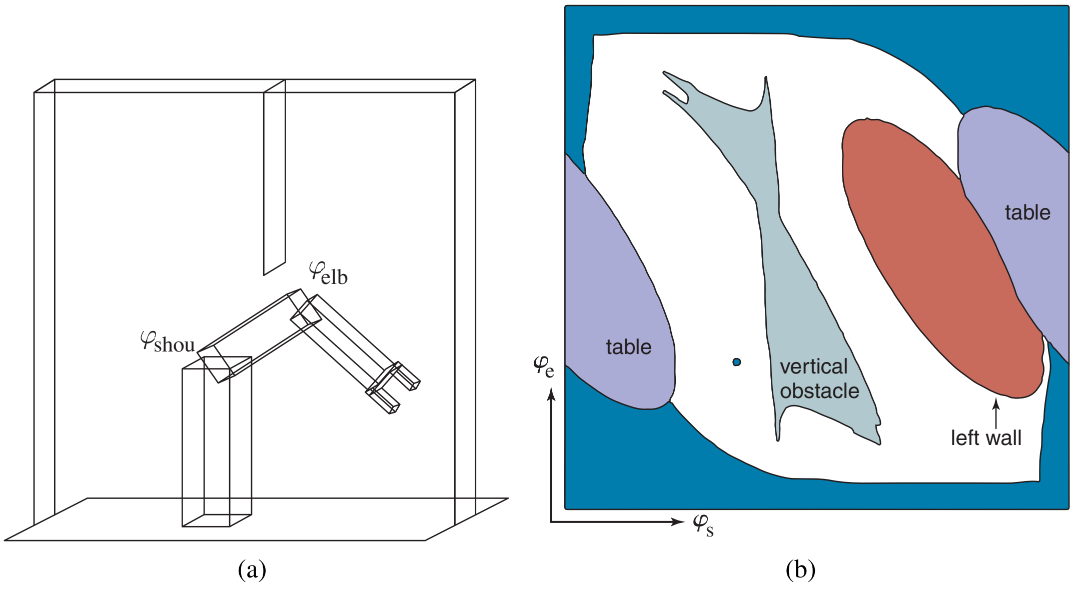


Figure 26.12 (a) Workspace representation of a robot arm with two degrees of freedom. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

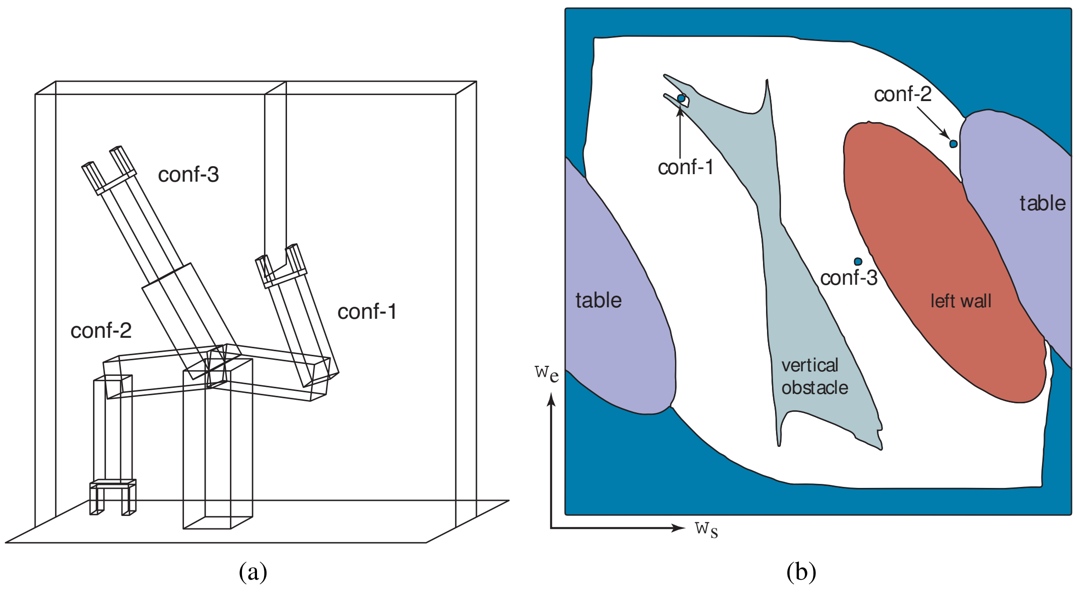


Figure 26.13 Three robot configurations, shown in workspace and configuration space.

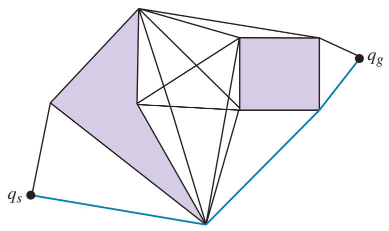


Figure 26.14 A visibility graph. Lines connect every pair of vertices that can "see" each other—lines that don't go through an obstacle. The shortest path must lie upon these lines.

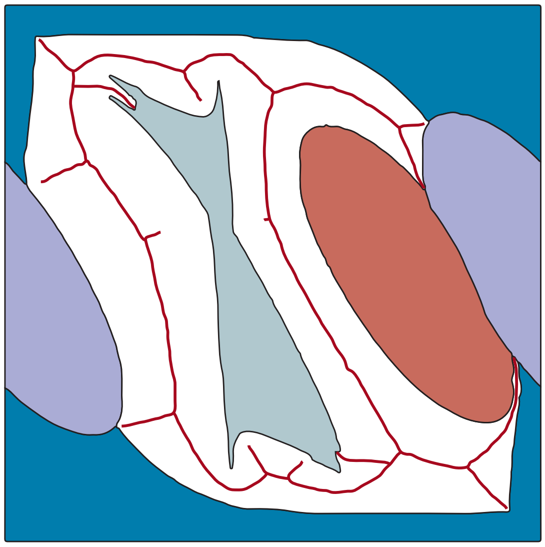


Figure 26.15 A Voronoi diagram showing the set of points (black lines) equidistant to two or more obstacles in configuration space.

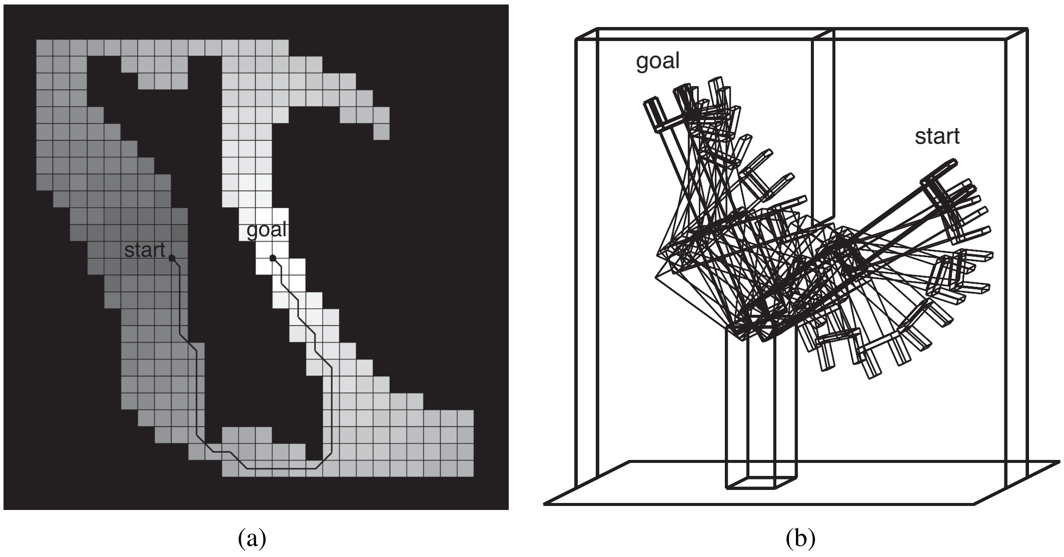


Figure 26.16 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

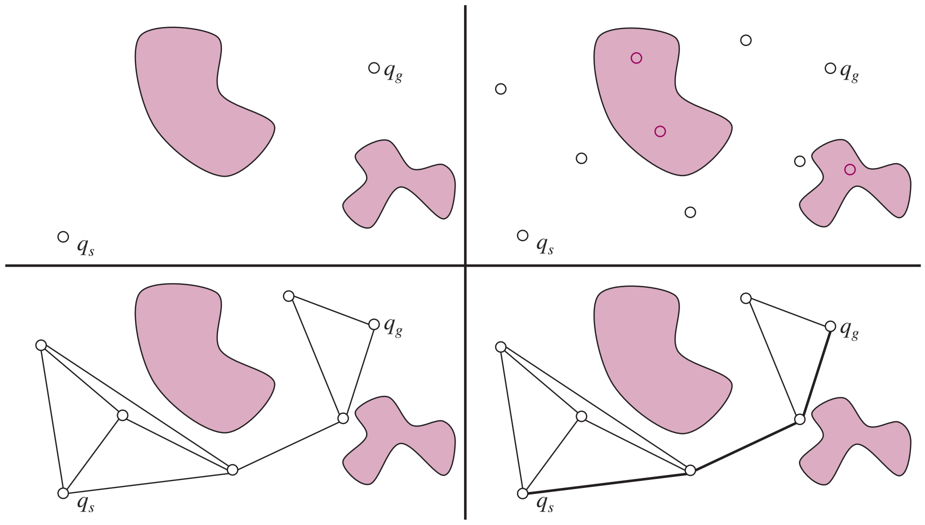


Figure 26.17 The probabilistic roadmap (PRM) algorithm. **Top left**: the start and goal configurations. **Top right**: sample M collision-free milestones (here $M = 5$). **Bottom left**: connect each milestone to its k nearest neighbors (here $k = 3$). **Bottom right**: find the shortest path from the start to the goal on the resulting graph.

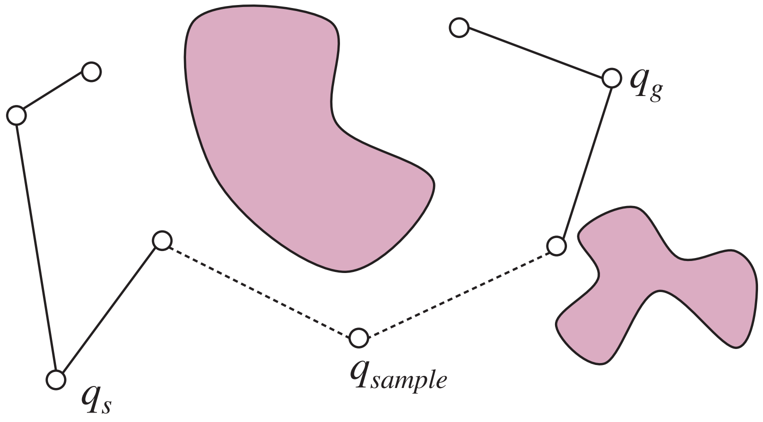


Figure 26.18 The bidirectional RRT algorithm constructs two trees (one from the start, the other from the goal) by incrementally connecting each sample to the closest node in each tree, if the connection is possible. When a sample connects to both trees, that means we have found a solution path.

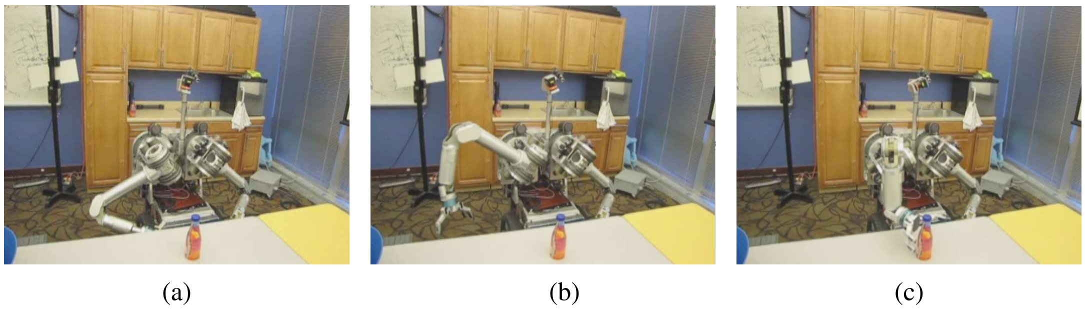


Figure 26.19 Snapshots of a trajectory produced by an RRT and post-processed with short-cutting. Courtesy of Anca Dragan.

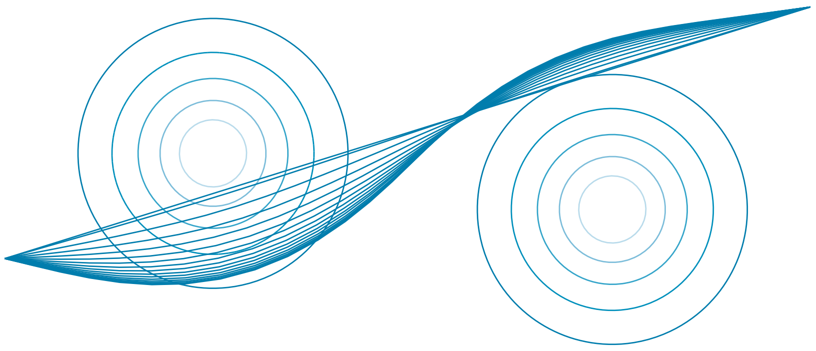


Figure 26.20 Trajectory optimization for motion planning. Two point-obstacles with circular bands of decreasing cost around them. The optimizer starts with the straight line trajectory, and lets the obstacles bend the line away from collisions, finding the minimum path through the cost field.

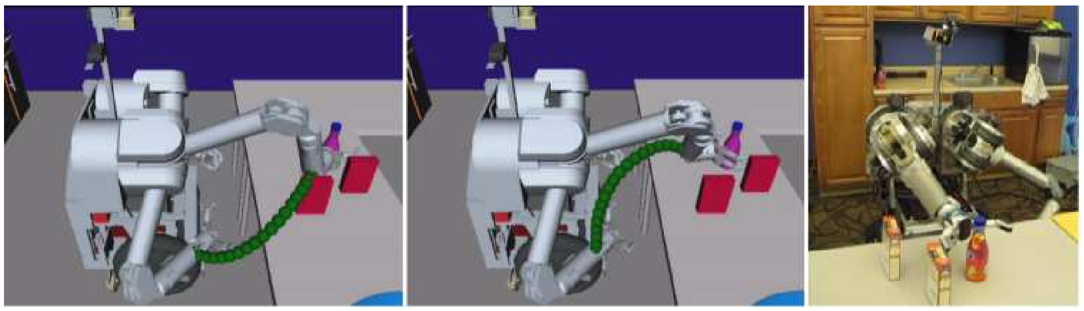


Figure 26.21 The task of reaching to grasp a bottle solved with a trajectory optimizer. Left: the initial trajectory, plotted for the end effector. Middle: the final trajectory after optimization. Right: the goal configuration. Courtesy of Anca Dragan. See Ratliff *et al.* (2009).

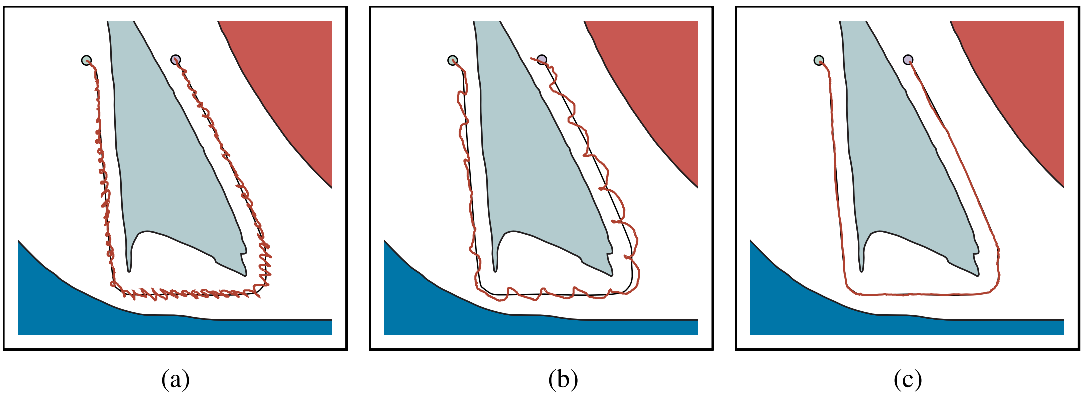


Figure 26.22 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the smooth line path, but in (a) and (b) deviates substantially from the path.

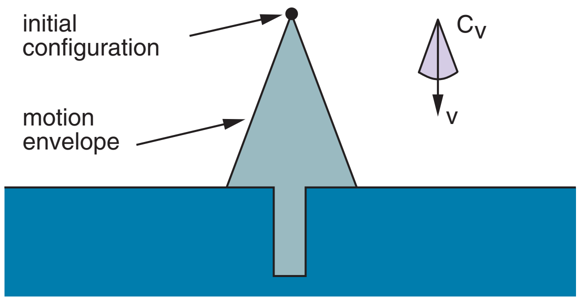


Figure 26.23 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v, but with uncertainty the actual velocity could be anywhere in C_v, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

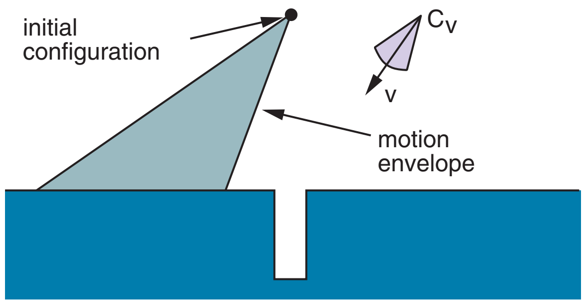


Figure 26.24 The first motion command and the resulting envelope of possible robot motions. No matter what actual motion ensues, we know the final configuration will be to the left of the hole.

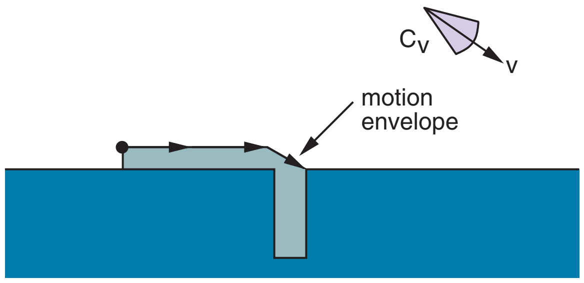


Figure 26.25 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the hole.

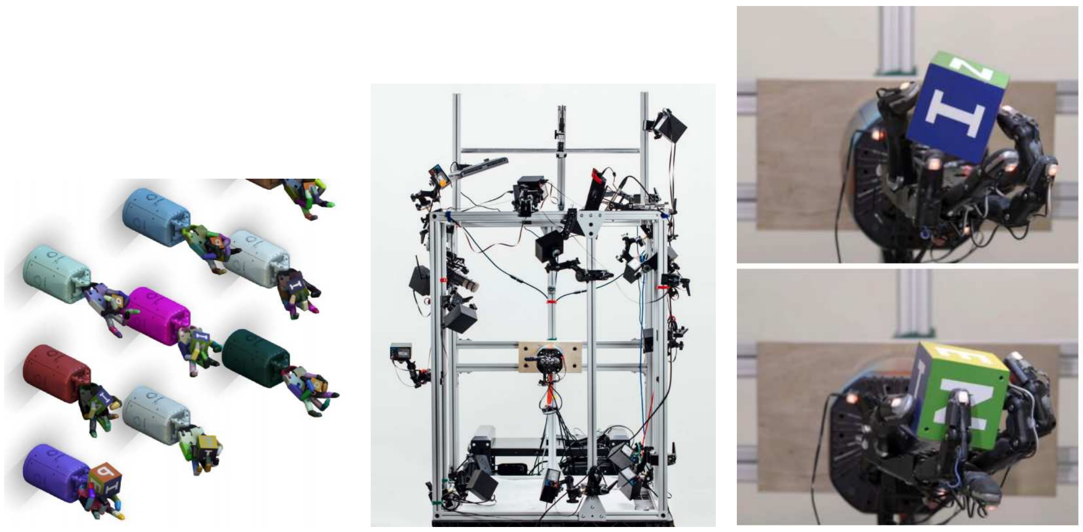


Figure 26.26 Training a robust policy. (a) Multiple simulations are run of a robot hand manipulating objects, with different randomized parameters for physics and lighting. Courtesy of Wojciech Zaremba. (b) The real-world environment, with a single robot hand in the center of a cage, surrounded by cameras and range finders. (c) Simulation and real-world training yields multiple different policies for grasping objects; here a pinch grasp and a quadpod grasp. Courtesy of OpenAI. See Andrychowicz *et al.* (2018a).

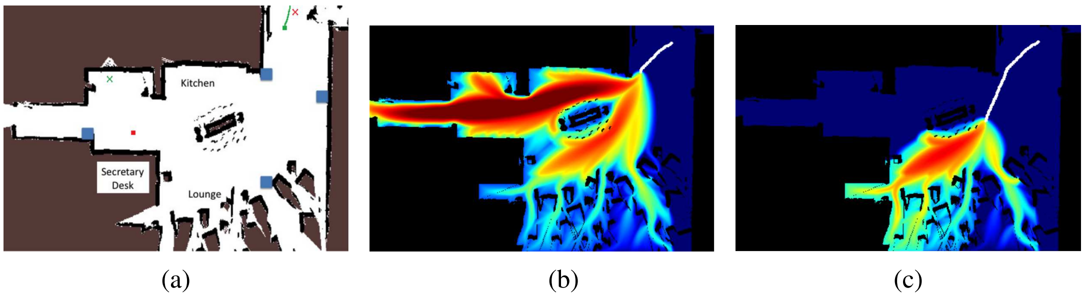


Figure 26.27 Making predictions by assuming that people are noisily rational given their goal: the robot uses the past actions to update a belief over what goal the person is heading to, and then uses the belief to make predictions about future actions. (a) The map of a room. (b) Predictions after seeing a small part of the person's trajectory (white path); (c) Predictions after seeing more human actions: the robot now knows that the person is not heading to the hallway on the left, because the path taken so far would be a poor path if that were the person's goal. Images courtesy of Brian D. Ziebart. See Ziebart *et al.* (2009).

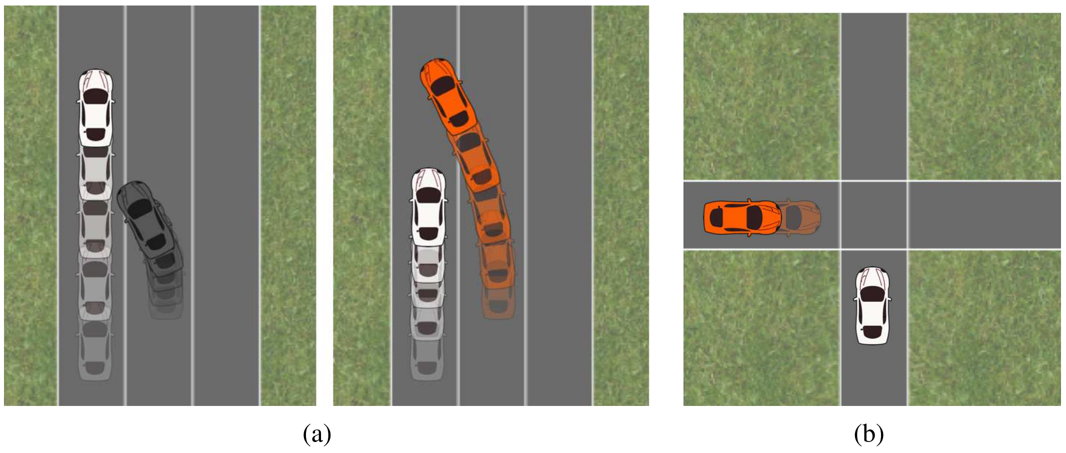


Figure 26.28 (a) Left: An autonomous car (middle lane) predicts that the human driver (left lane) wants to keep going forward, and plans a trajectory that slows down and merges behind. Right: The car accounts for the influence its actions can have on human actions, and realizes it can merge in front and rely on the human driver to slow down. (b) That same algorithm produces an unusual strategy at an intersection: the car realizes that it can make it more likely for the person (bottom) to proceed faster through the intersection by starting to inch backwards. Images courtesy of Anca Dragan. See Sadigh *et al.* (2016).

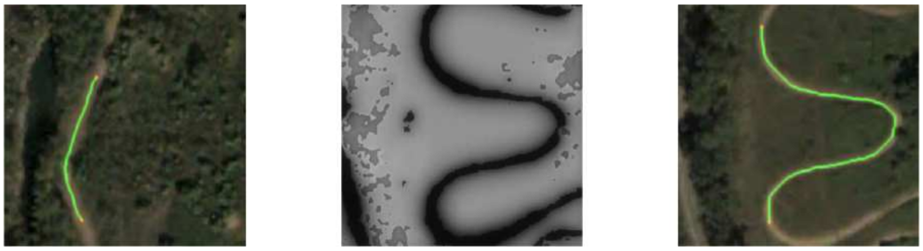


Figure 26.29 Left: A mobile robot is shown a demonstration that stays on the dirt road. Middle: The robot infers the desired cost function, and uses it in a new scene, knowing to put lower cost on the road there. Right: The robot plans a path for the new scene that also stays on the road, reproducing the preferences behind the demonstration. Images courtesy of Nathan Ratliff and James A. Bagnell. See Ratliff *et al.* (2006).

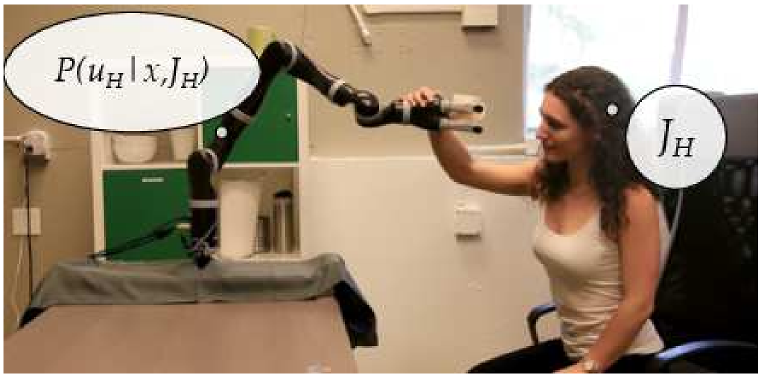
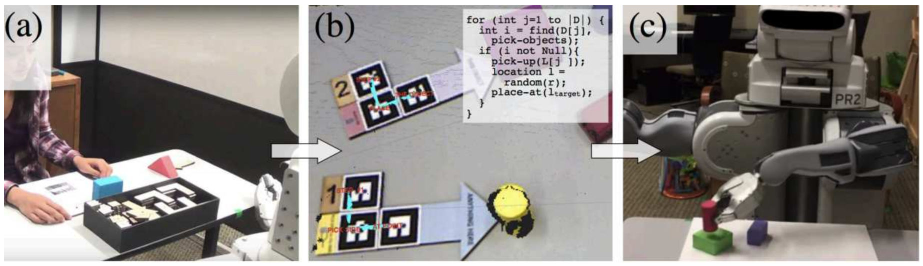


Figure 26.30 A human teacher pushes the robot down to teach it to stay closer to the table. The robot appropriately updates its understanding of the desired cost function and starts optimizing it. Courtesy of Anca Dragan. See Bajcsy *et al.* (2017).



```
for (int j=1 to |D|) {
  int i = find(D[j],
    pick-objects);
  if (i not Null){
    pick-up(L[j ]);
    location l =
      random(r);
    place-at(l_target);
  }
}
```

Figure 26.31 A programming interface that involves placing specially designed blocks in the robot's workspace to select objects and specify high-level actions. Images courtesy of Maya Cakmak. See Sefidgar *et al.* (2017).

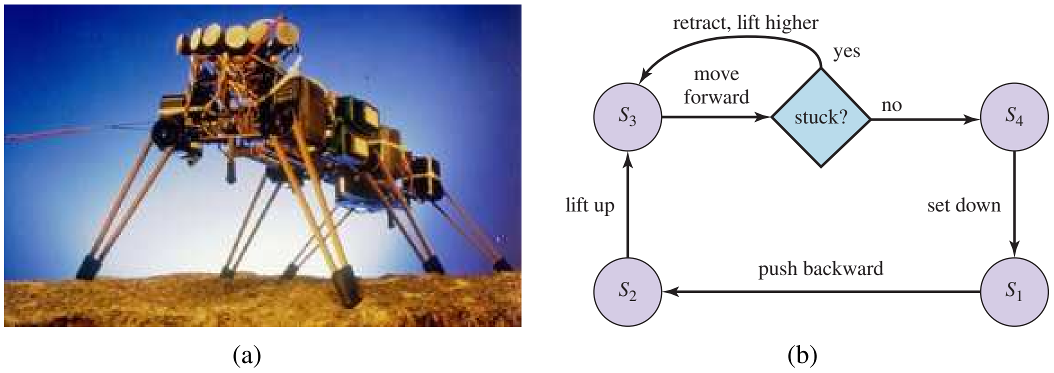
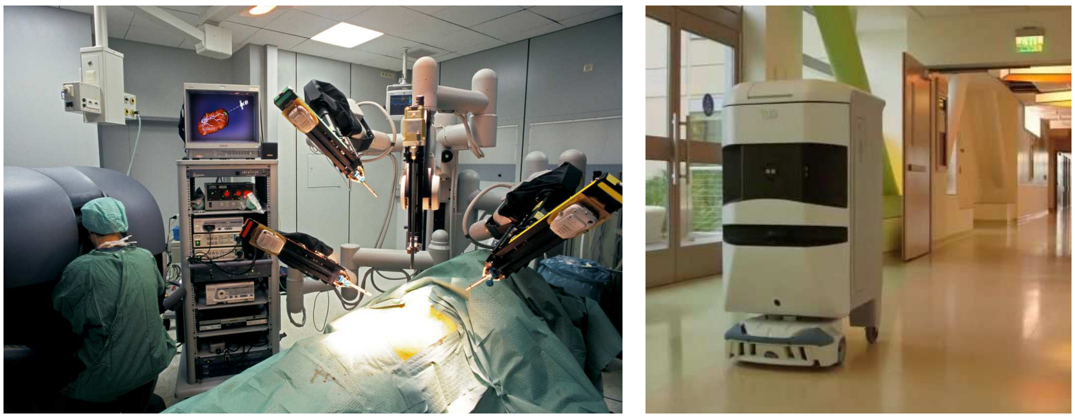


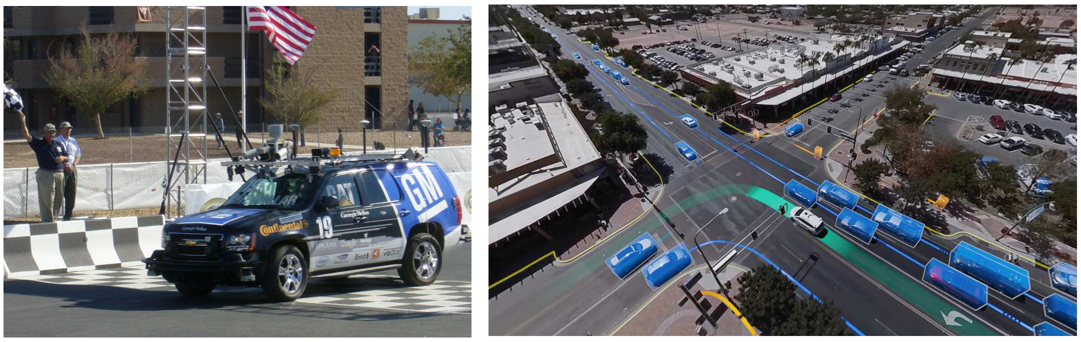
Figure 26.32 (a) Genghis, a hexapod robot. (Image courtesy of Rodney A. Brooks.) (b) An augmented finite state machine (AFSM) that controls one leg. The AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

Figure 26.33 (a) A patient with a brain–machine interface controlling a robot arm to grab a drink. Image courtesy of Brown University. (b) Roomba, the robot vacuum cleaner. Photo by HANDOUT/KRT/Newscom.



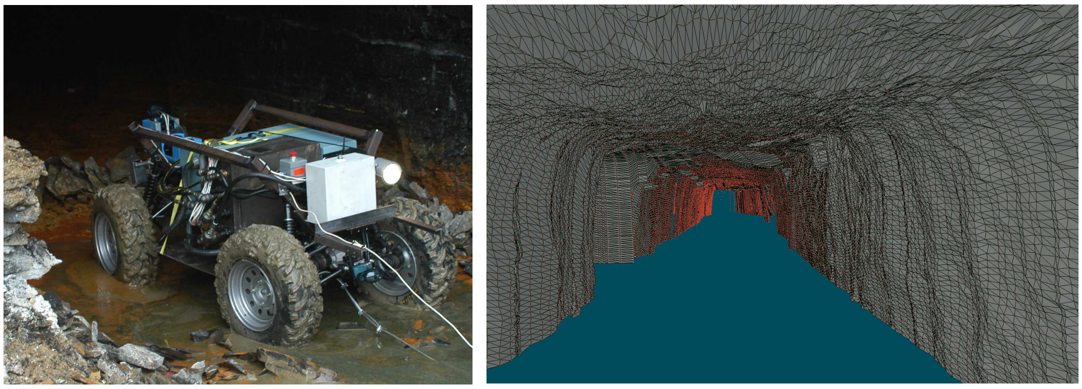
(a) (b)

Figure 26.34 (a) Surgical robot in the operating room. Photo by Patrick Landmann/Science Source. (b) Hospital delivery robot. Photo by Wired.



(a) (b)

Figure 26.35 (a) Autonomous car BOSS which won the DARPA Urban Challenge. Photo by Tangi Quemener/AFP/Getty Images/Newscom. Courtesy of Sebastian Thrun. (b) Aerial view showing the perception and predictions of the Waymo autonomous car (white vehicle with green track). Other vehicles (blue boxes) and pedestrians (orange boxes) are shown with anticipated trajectories. Road/sidewalk boundaries are in yellow. Photo courtesy of Waymo.



(a) (b)

Figure 26.36 (a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot. Courtesy of Sebastian Thrun.

CHAPTER 27

COMPUTER VISION

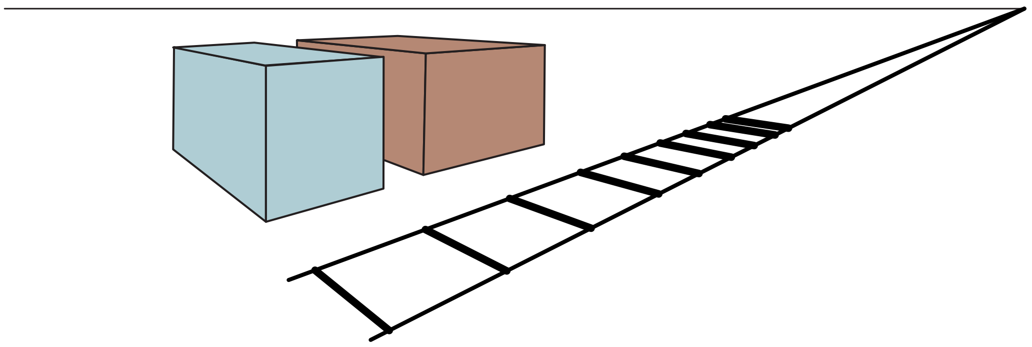


Figure 27.1 Geometry in the scene appears distorted in images. Parallel lines appear to meet, like the railway tracks in a desolate town. Buildings that have right angles in the real world scene have distorted angles in the image.

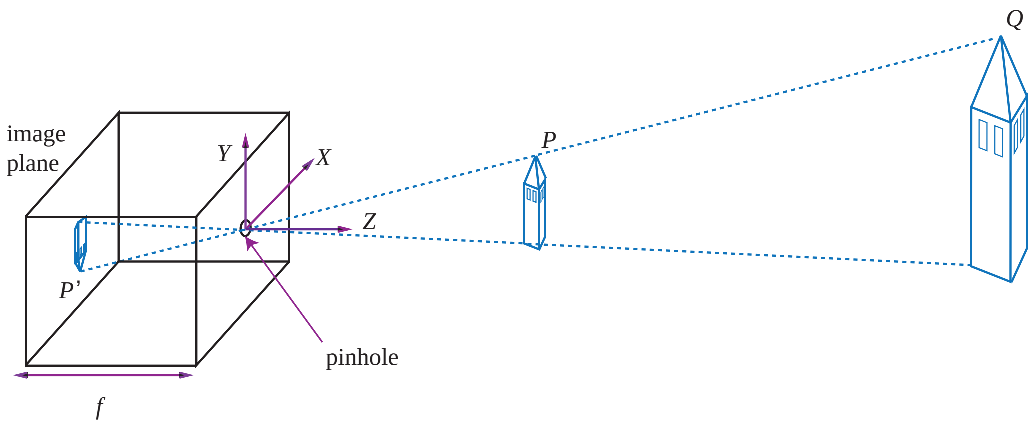


Figure 27.2 Each light sensitive element at the back of a pinhole camera receives light that passes through the pinhole from a small range of directions. If the pinhole is small enough, the result is a focused image behind the pinhole. The process of projection means that large, distant objects look the same as smaller, nearby objects—the point P' in the image plane could have come from a nearby toy tower at point P or from a distant real tower at point Q.

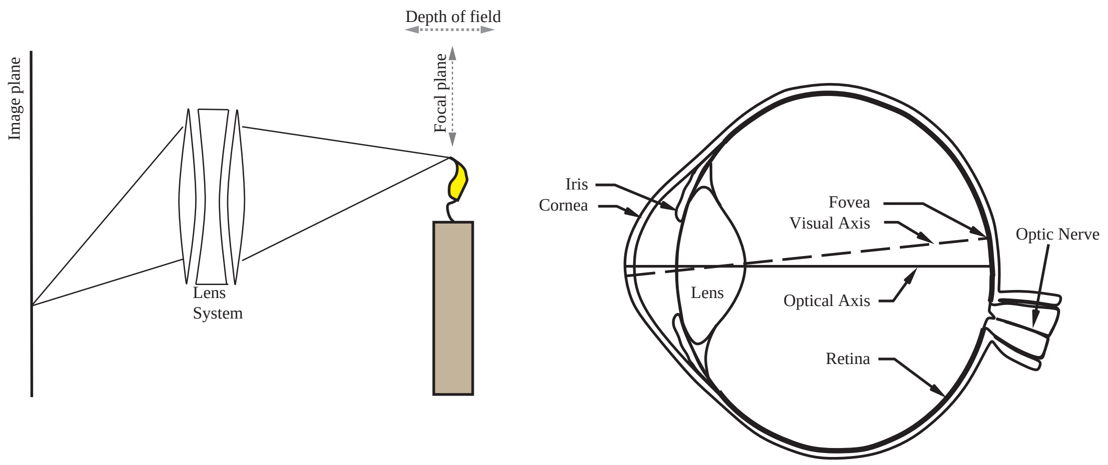


Figure 27.3 Lenses collect the light leaving a point in the scene (here, the tip of the candle flame) in a range of directions, and steer all the light to arrive at a single point on the image plane. Points in the scene near the focal plane—within the depth of field—will be focused properly. In cameras, elements of the lens system move to change the focal plane, whereas in the eye, the shape of the lens is changed by specialized muscles.

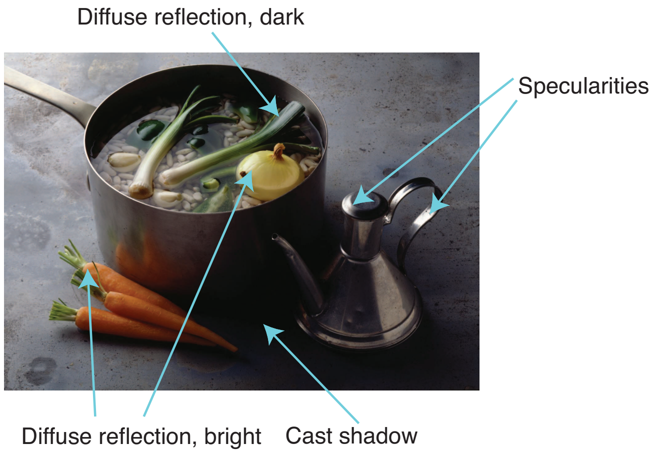


Figure 27.4 This photograph illustrates a variety of illumination effects. There are specularities on the stainless steel cruet. The onions and carrots are bright diffuse surfaces because they face the light direction. The shadows appear at surface points that cannot see the light source at all. Inside the pot are some dark diffuse surfaces where the light strikes at a tangential angle. (There are also some shadows inside the pot.) Photo by Ryman Cabannes/Image Professionals GmbH/Alamy Stock Photo.

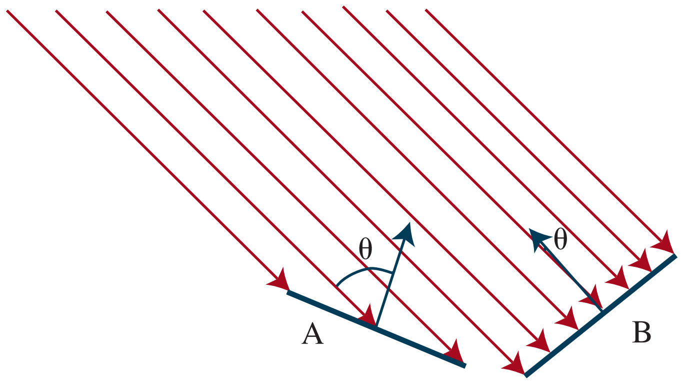


Figure 27.5 Two surface patches are illuminated by a distant point source, whose rays are shown as light arrows. Patch A is tilted away from the source (θ is close to 90°) and collects less energy, because it cuts fewer light rays per unit surface area. Patch B, facing the source (θ is close to 0°), collects more energy.

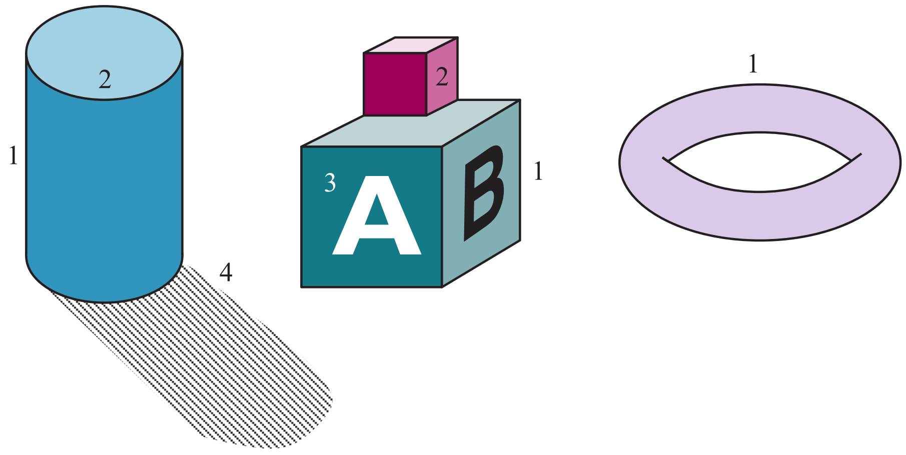


Figure 27.6 Different kinds of edges: (1) depth discontinuities; (2) surface orientation discontinuities; (3) reflectance discontinuities; (4) illumination discontinuities (shadows).

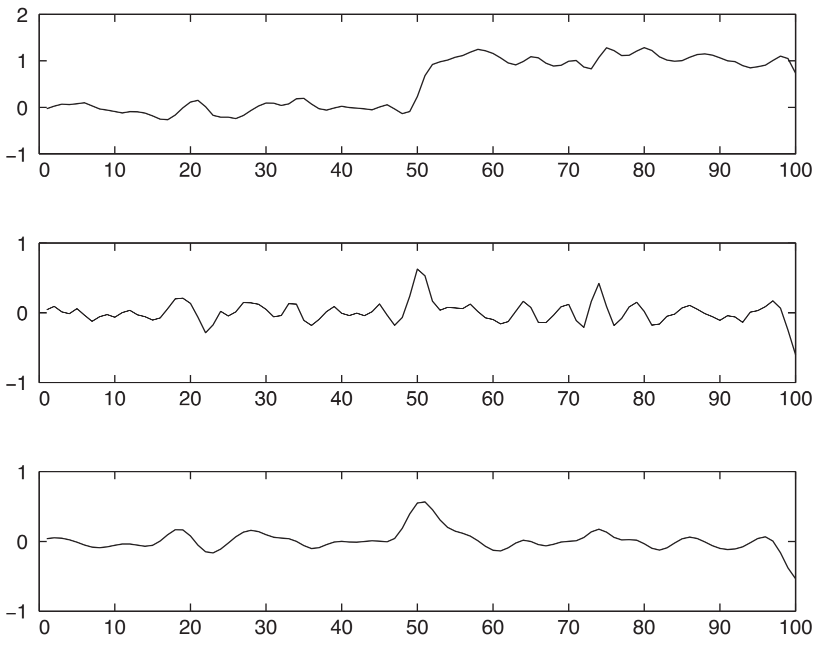


Figure 27.7 Top: Intensity profile $I(x)$ along a one-dimensional section across a step edge. Middle: The derivative of intensity, $I'(x)$. Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity. The noisy candidate edge at $x = 75$ has disappeared.

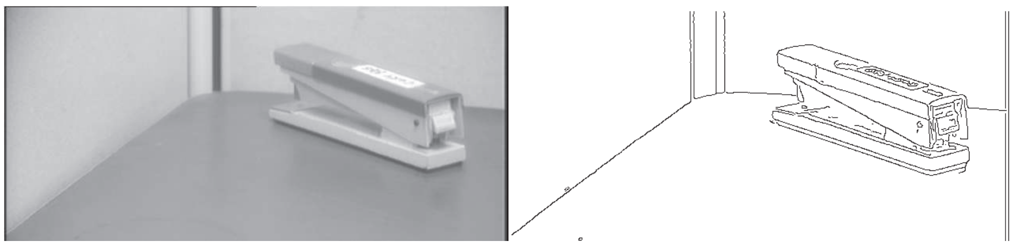


Figure 27.8 (a) Photograph of a stapler. (b) Edges computed from (a).

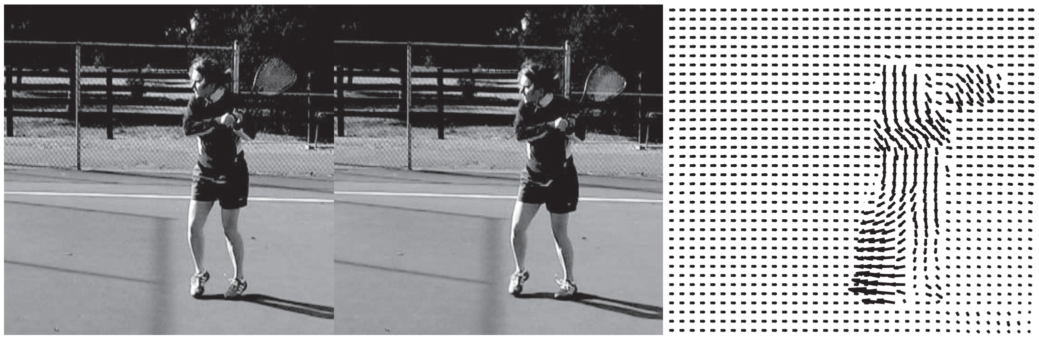


Figure 27.9 Two frames of a video sequence and the optical flow field corresponding to the displacement from one frame to the other. Note how the movement of the tennis racket and the front leg is captured by the directions of the arrows. (Images courtesy of Thomas Brox.)

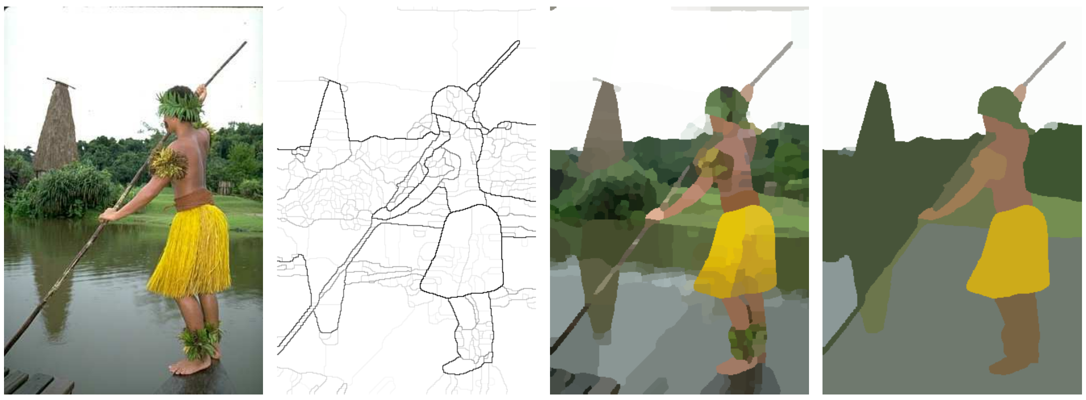


Figure 27.10 (a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the image, resulting in fewer regions. (Images courtesy of Pablo Arbelaez, Michael Maire, Charless Fowlkes and Jitendra Malik.)

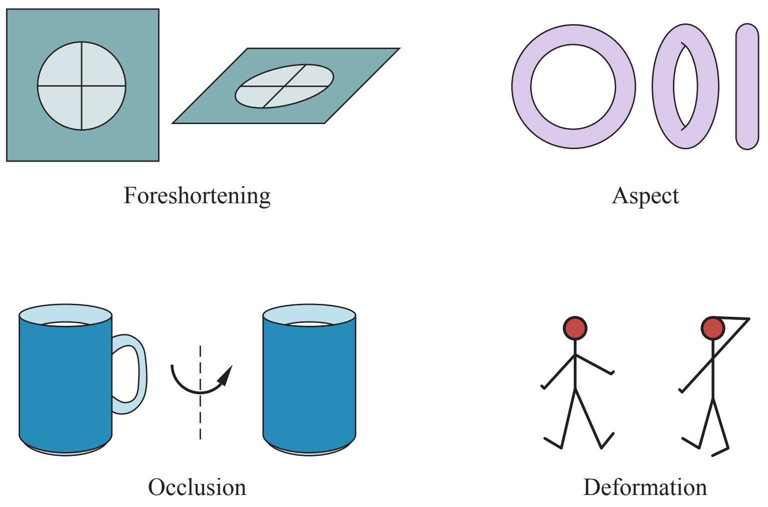


Figure 27.11 Important sources of appearance variation that can make different images of the same object look different. First, elements can foreshorten, like the circular patch on the top left. This patch is viewed at a glancing angle, and so is elliptical in the image. Second, objects viewed from different directions can change shape quite dramatically, a phenomenon known as aspect. On the top right are three different aspects of a doughnut. Occlusion causes the handle of the mug on the bottom left to disappear when the mug is rotated. In this case, because the body and handle belong to the same mug, we have self-occlusion. Finally, on the bottom right, some objects can deform dramatically.

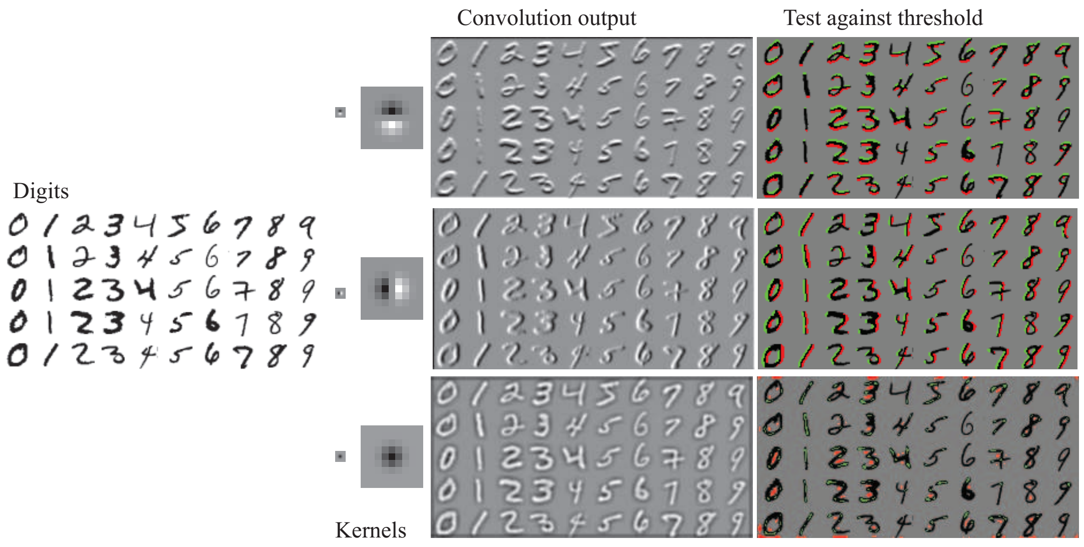


Figure 27.12 On the far left, some images from the MNIST data set. Three kernels appear on the center left. They are shown at actual size (tiny blocks) and magnified to reveal their content: mid-grey is zero, light is positive, and dark is negative. Center right shows the results of applying these kernels to the images. Right shows pixels where the response is bigger than a threshold (green) or smaller than a threshold (red). You should notice that this gives (from top to bottom): a horizontal bar detector; a vertical bar detector; and (harder to note) a line ending detector. These detectors pay attention to the contrast of the bar, so (for example) a horizontal bar that is light on top and dark below produces a positive (green) response, and one that is dark on top and light below gets a negative (red) response. These detectors are moderately effective, but not perfect.

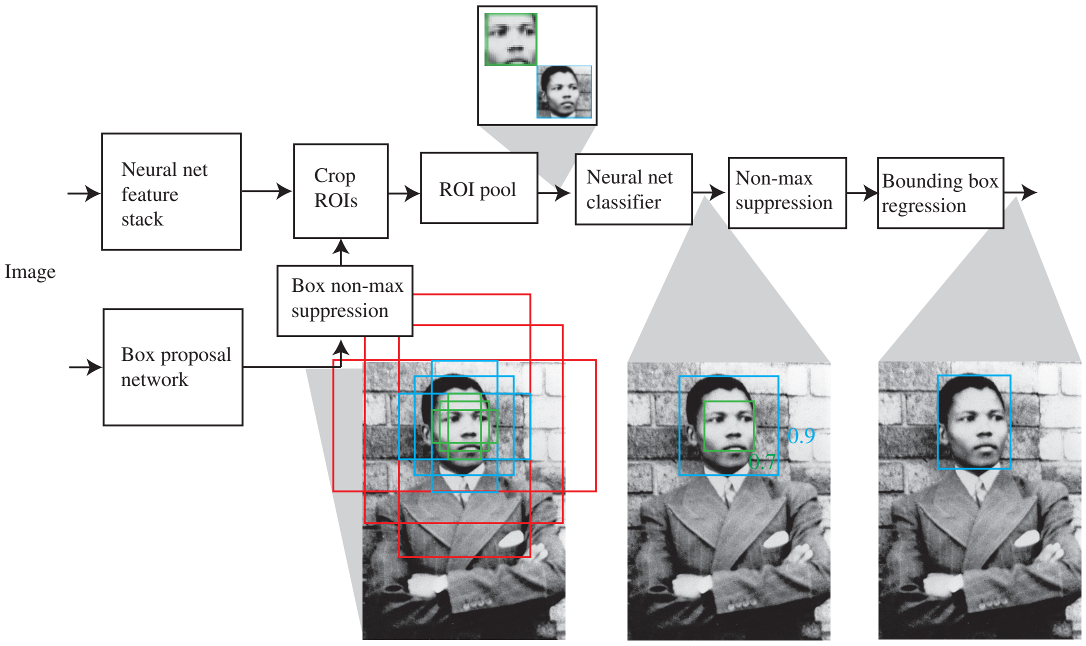


Figure 27.13 Faster RCNN uses two networks. A picture of a young Nelson Mandela is fed into the object detector. One network computes "objectness" scores of candidate image boxes, called "anchor boxes," centered at a grid point. There are nine anchor boxes (three scales, three aspect ratios) at each grid point. For the example image, an inner green box and an outer blue box have passed the objectness test. The second network is a feature stack that computes a representation of the image suitable for classification. The boxes with highest objectness score are cut from the feature map, standardized in size with ROI pooling, and passed to a classifier. The blue box has a higher score than the green box and overlaps it, so the green box is rejected by non-maximum suppression. Finally, bounding box regression the blue box so that it fits the face. This means that the relatively coarse sampling of locations, scales, and aspect ratios does not weaken accuracy. Photo by Sipa/Shutterstock.

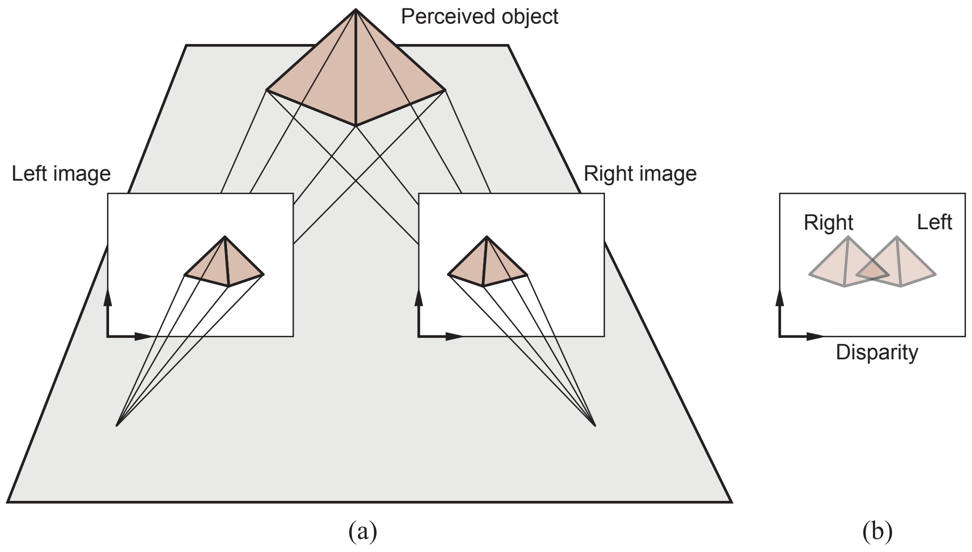


Figure 27.14 Translating a camera parallel to the image plane causes image features to move in the camera plane. The disparity in positions that results is a cue to depth. If we superimpose left and right images, as in (b), we see the disparity.

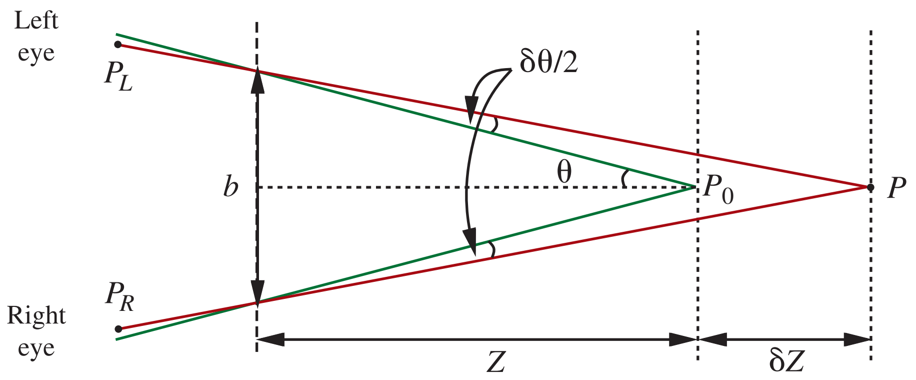


Figure 27.15 The relation between disparity and depth in stereopsis. The centers of projection of the two eyes are distance b apart, and the optical axes intersect at the fixation point P_0. The point P in the scene projects to points P_L and P_R in the two eyes. In angular terms, the disparity between these is $\delta\theta$ (the diagram shows two angles of $\delta\theta/2$).

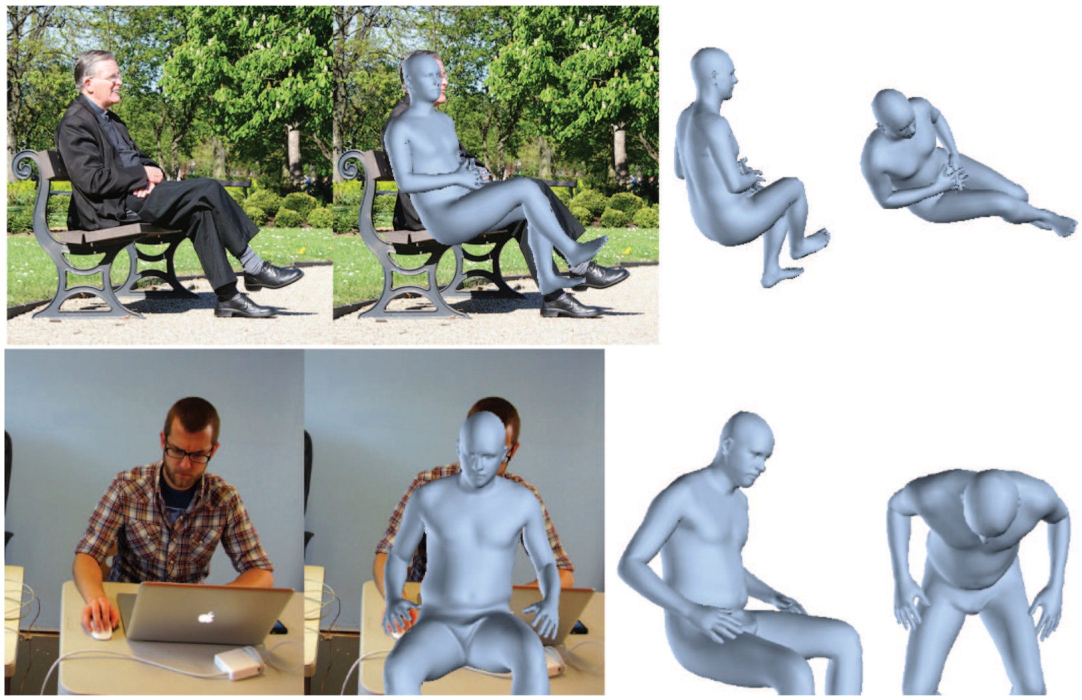


Figure 27.16 Reconstructing humans from a single image is now practical. Each row shows a reconstruction of 3D body shape obtained using a single image. These reconstructions are possible because methods can estimate the location of joints, the joint angles in 3D, the shape of the body, and the pose of the body with respect to an image. Each row shows the following: **far left** a picture; **center left** the picture with the reconstructed body superimposed; **center right** another view of the reconstructed body; and **far right** yet another view of the reconstructed body. The different views of the body make it much harder to conceal errors in reconstruction. Figure courtesy of Angjoo Kanazawa, produced by a system described in Kanazawa *et al*. (2018a).

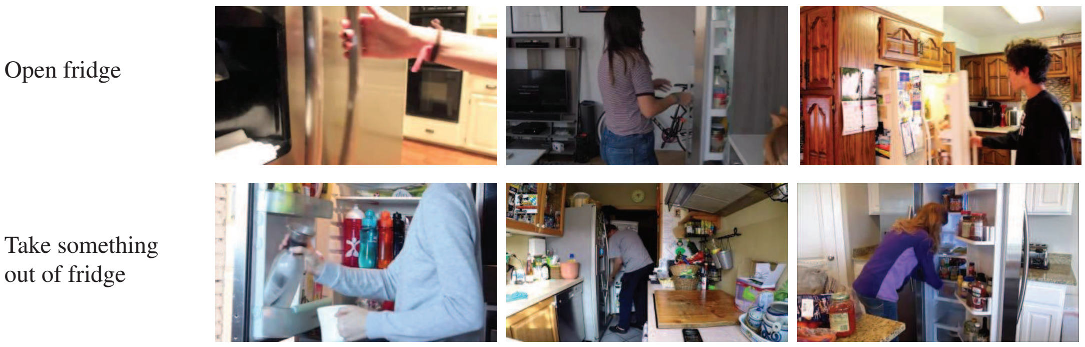


Figure 27.17 The same action can look very different; and different actions can look similar. These examples show actions taken from a data set of natural behaviors; the labels are chosen by the curators of the data set, rather than predicted by an algorithm. **Top:** examples of the label "opening fridge," some shown in closeup and some from afar. **Bottom:** examples of the label "take something out of fridge." Notice how in both rows the subject's hand is close to the fridge door—telling the difference between the cases requires quite subtle judgment about where the hand is and where the door is. Figure courtesy of David Fouhey, taken from a data set described in Fouhey *et al.* (2018).

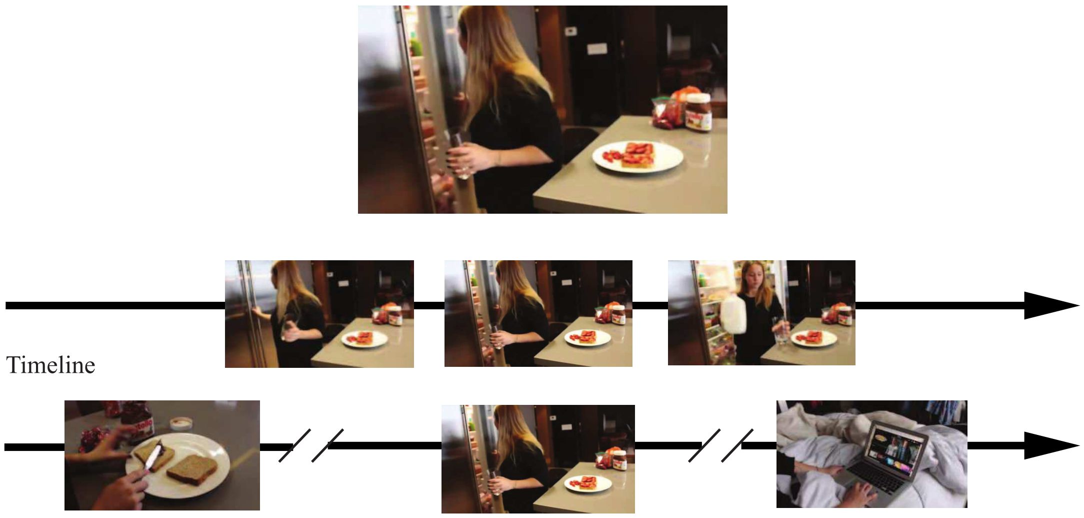


Figure 27.18 What you call an action depends on the time scale. The single frame at the **top** is best described as opening the fridge (you don't gaze at the contents when you close a fridge). But if you look at a short clip of video (indicated by the frames in the **center** row), the action is best described as getting milk from the fridge. If you look at a long clip (the frames in the **bottom** row), the action is best described as fixing a snack. Notice that this illustrates one way in which behavior composes: getting milk from the fridge is sometimes part of fixing a snack, and opening the fridge is usually part of getting milk from the fridge. Figure courtesy of David Fouhey, taken from a data set described in Fouhey *et al.* (2018).

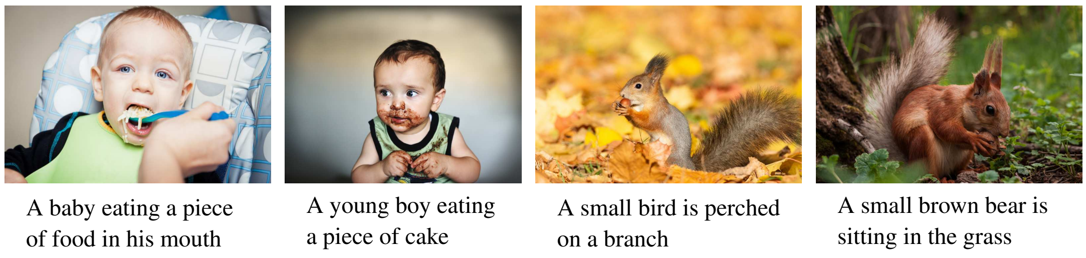

A baby eating a piece of food in his mouth
A young boy eating a piece of cake
A small bird is perched on a branch
A small brown bear is sitting in the grass

Figure 27.19 Automated image captioning systems produce some good results and some failures. The two captions at left describe the respective images well, although "eating . . . in his mouth" is a disfluency that is fairly typical of the recurrent neural network language models used by early captioning systems. For the two captions on the right, the captioning system seems not to know about squirrels, and so guesses the animal from context; it also fails to recognize that the two squirrels are eating. Image credits: geraine/Shutterstock; ESB Professional/Shutterstock; BushAlex/Shutterstock; Maria.Tem/Shutterstock. The images shown are similar but not identical to the original images from which the captions were generated. For the original images see Aneja *et al.* (2018).

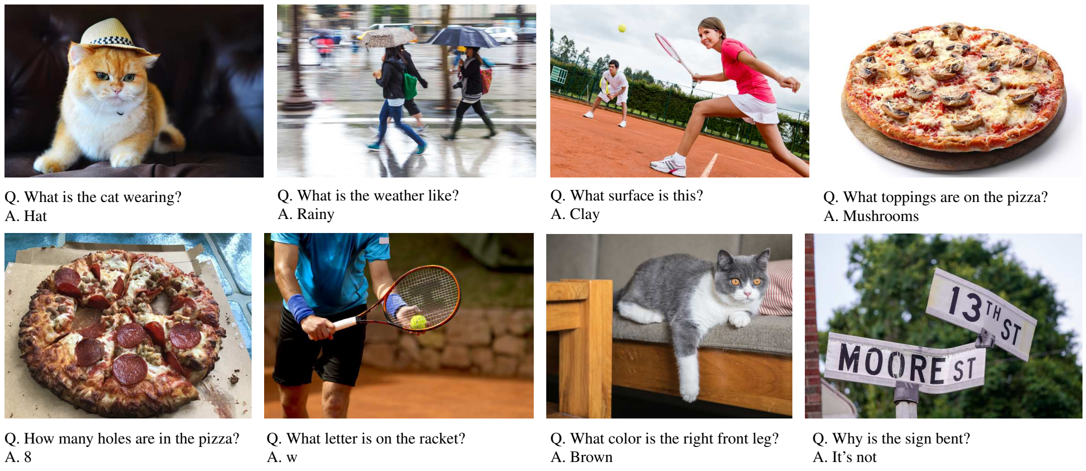

Q. What is the cat wearing?
A. Hat
Q. What is the weather like?
A. Rainy
Q. What surface is this?
A. Clay
Q. What toppings are on the pizza?
A. Mushrooms
Q. How many holes are in the pizza?
A. 8
Q. What letter is on the racket?
A. w
Q. What color is the right front leg?
A. Brown
Q. Why is the sign bent?
A. It's not
13TH ST
MOORE ST

Figure 27.20 Visual question-answering systems produce answers (typically chosen from a multiple-choice set) to natural-language questions about images. **Top:** the system is producing quite sensible answers to rather difficult questions about the image. **Bottom:** less satisfactory answers. For example, the system is guessing about the number of holes in a pizza, because it doesn't understand what counts as a hole, and it has real difficulty counting. Similarly, the system selects brown for the cat's leg because the background is brown and it can't localize the leg properly. Image credits: (Top) Tobyanna/Shutterstock; 679411/Shutterstock; ESB Professional/Shutterstock; Africa Studio/Shutterstock; (Bottom) Stuart Russell; Maxisport/Shutterstock; Chendongshan/Shutterstock; Scott Biales DitchTheMap/Shutterstock. The images shown are similar but not identical to the original images to which the question-answering system was applied. For the original images see Goyal *et al.* (2017).

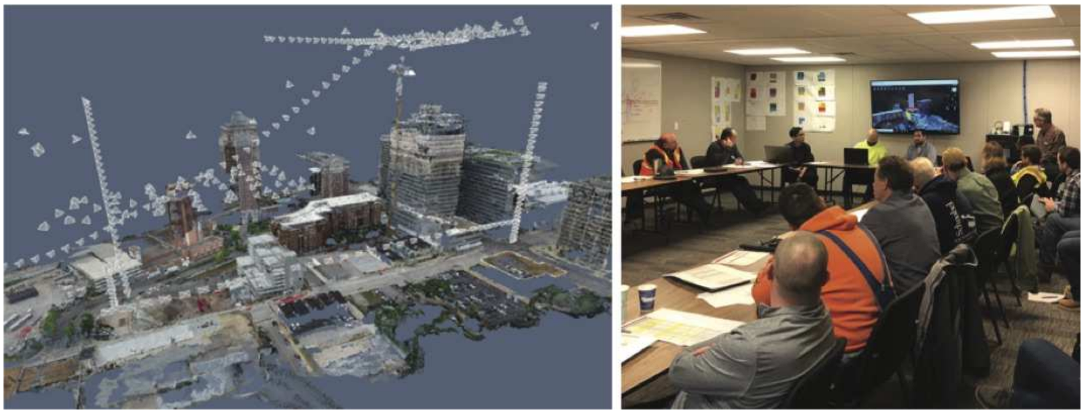


Figure 27.21 3D models of construction sites are produced from images by structure-from-motion and multiview stereo algorithms. They help construction companies to coordinate work on large buildings by comparing a 3D model of the actual construction to date with the building plans. **Left**: A visualization of a geometric model captured by drones. The reconstructed 3D points are rendered in color, so the result looks like progress to date (note the partially completed building with crane). The small pyramids show the pose of a drone when it captured an image, to allow visualization of the flight path. **Right**: These systems are actually used by construction teams; this team views the model of the as-built site, and compares it with building plans as part of the coordination meeting. Figure courtesy of Derek Hoiem, Mani Golparvar-Fard and Reconstruct, produced by a commercial system described in a blog post at `medium.com/reconstruct-inc`.

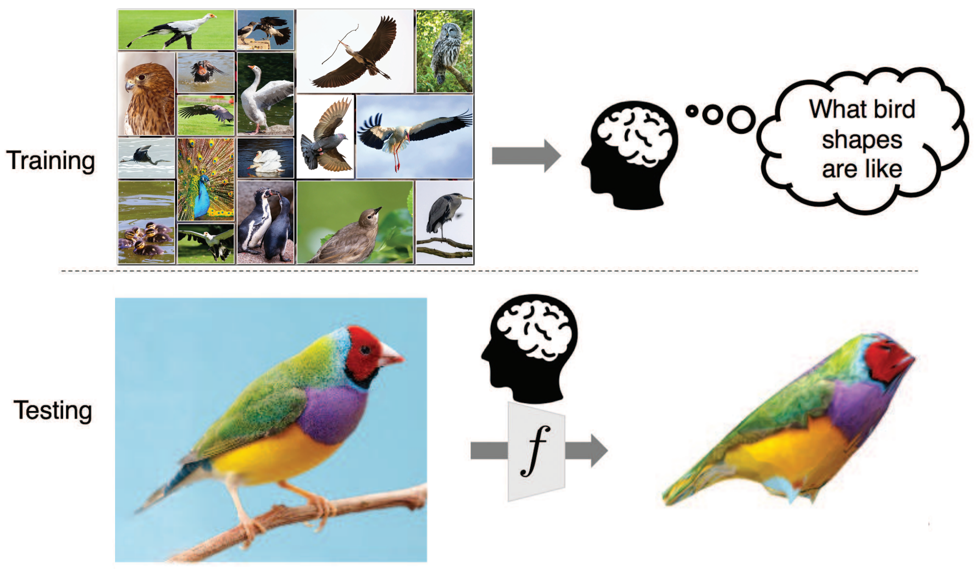


Figure 27.22 If you have seen many pictures of some category—say, birds (**top**)—you can use them to produce a 3D reconstruction from a single new view (**bottom**). You need to be sure that all objects have a fairly similar geometry (so a picture of an ostrich won't help if you're looking at a sparrow), but classification methods can sort this out. From the many images you can estimate how texture values in the image are distributed across the object, and thus complete the texture for parts of the bird you haven't seen yet (**bottom**). Figure courtesy of Angjoo Kanazawa, produced by a system described in Kanazawa *et al.* (2018b). Top photo credit: Satori/123RF; Bottom left credit: Four Oaks/Shutterstock.

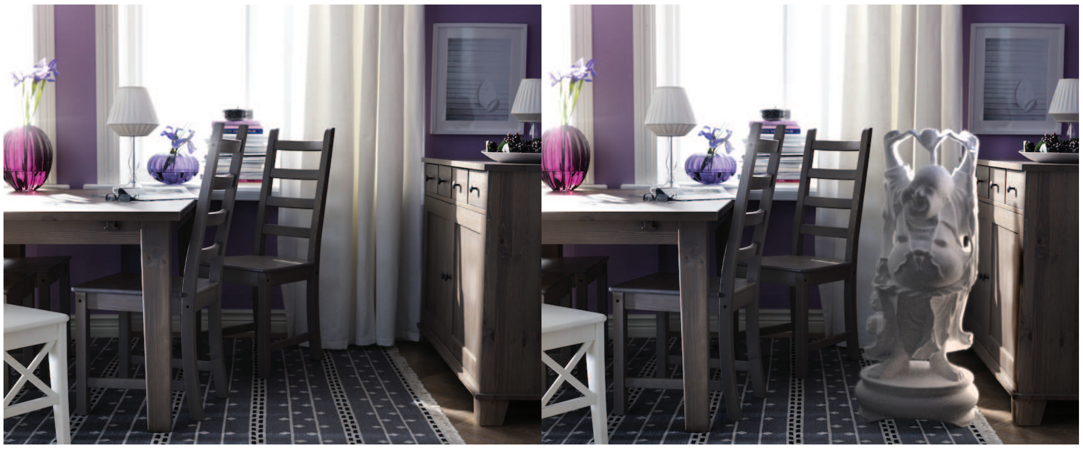


Figure 27.23 On the **left**, an image of a real scene. On the **right**, a computer graphics object has been inserted into the scene. You can see that the light appears to be coming from the right direction, and that the object seems to cast appropriate shadows. The generated image is convincing even if there are small errors in the lighting and shadows, because people are not expert at identifying these errors. Figure courtesy of Kevin Karsch, produced by a system described in Karsch *et al.* (2011).

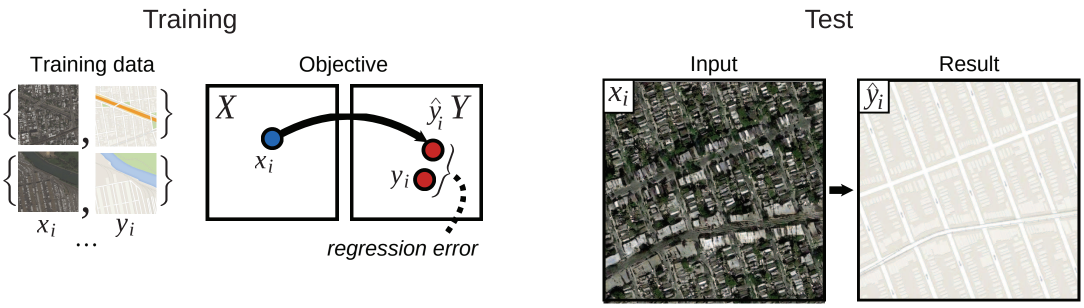


Figure 27.24 Paired image translation where the input consists of aerial images and the corresponding map tiles, and the goal is to train a network to produce a map tile from an aerial image. (The system can also learn to generate aerial images from map tiles.) The network is trained by comparing $\hat{y}_i$ (the output for example x_i of type X) to the right output y_i of type Y. Then at test time, the network must make new images of type Y from new inputs of type X. Figure courtesy of Phillip Isola, Jun-Yan Zhu and Alexei A. Efros, produced by a system described in Isola *et al.* (2017). Map data © 2019 Google.

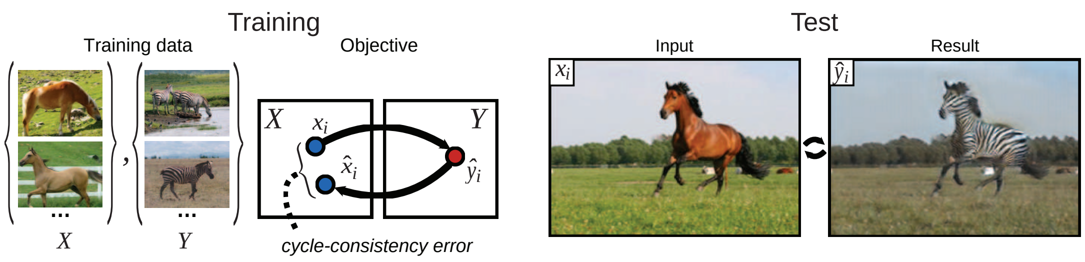


Figure 27.25 Unpaired image translation: given two populations of images (here type X is horses and type Y is zebras), but no corresponding pairs, learn to translate a horse into a zebra. The method trains two predictors: one that maps type X to type Y, and another that maps type Y to type X. If the first network maps a horse x_i to a zebra $\hat{y}_i$, the second network should map $\hat{y}_i$ back to the original x_i. The difference between x_i and $\hat{x}_i$ is what trains the two networks. The cycle from Y to X and back must be closed. Such networks can successfully impose rich transformations on images. Figure courtesy of Alexei A. Efros; see Zhu *et al.* (2017). Running horse photo by Justyna Furmanczyk Gibaszek/Shutterstock.

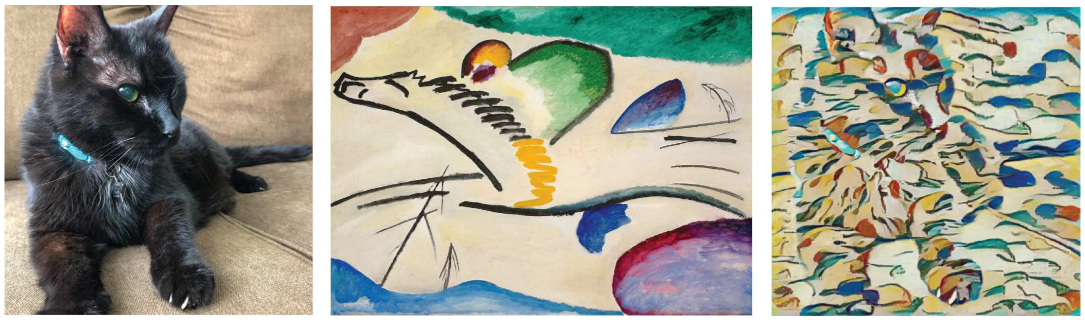
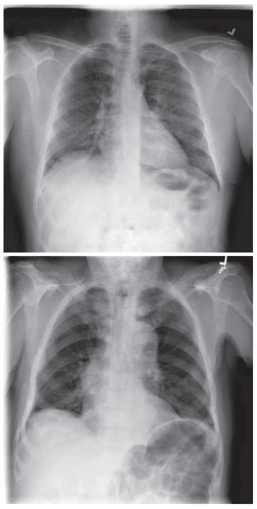


Figure 27.26 Style transfer: The *content* of a photo of a cat is combined with the *style* of an abstract painting to yield a new image of the cat rendered in the abstract style (right). The painting is Wassily Kandinsky's *Lyrisches* or *The Lyrical* (public domain); the cat is Cosmo.



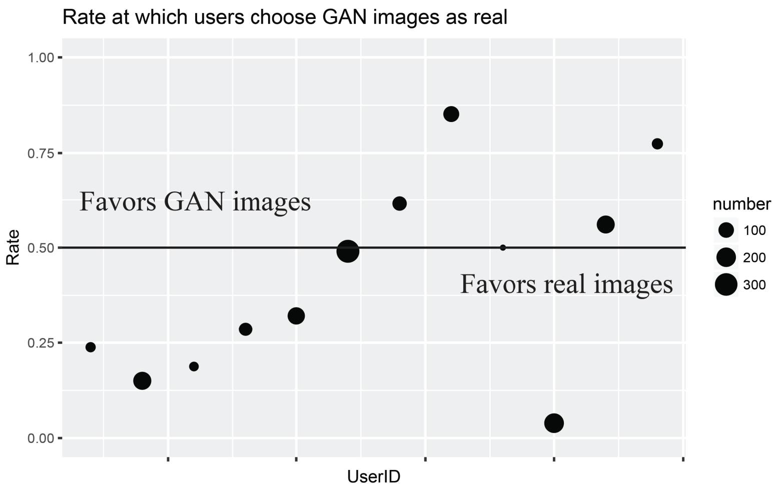


Figure 27.27 GAN generated images of lung X-rays. On the left, a pair consisting of a real X-ray and a GAN-generated X-ray. On the right, results of a test asking radiologists, given a pair of X-rays as seen on the left, to tell which is the real X-ray. On average, they chose correctly 61% of the time, somewhat better than chance. But they differed in their accuracy—the chart on the right shows the error rate for 12 different radiologists; one of them had an error rate near 0% and another had 80% errors. The size of each dot indicates the number of images each radiologist viewed. Figure courtesy of Alex Schwing, produced by a system described in Deshpande *et al*. (2019).

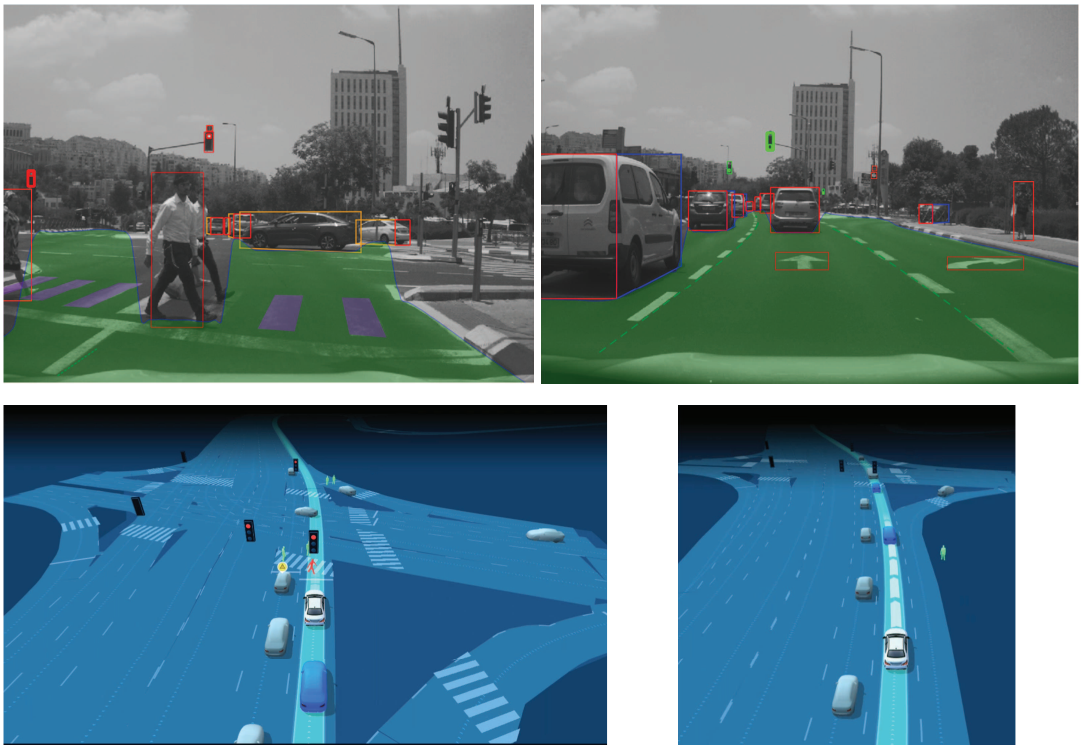
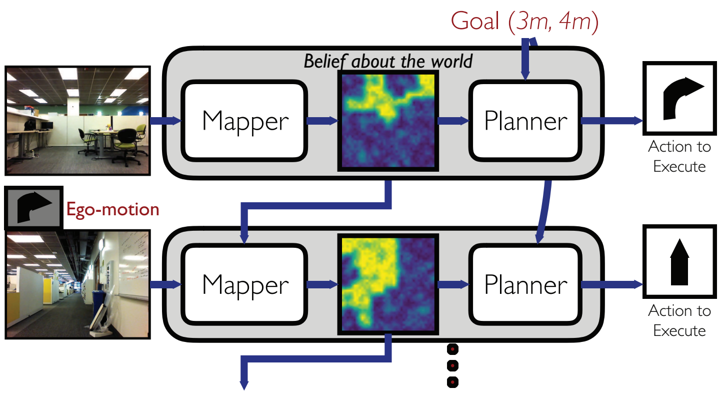


Figure 27.28 Mobileye's camera-based sensing for autonomous vehicles. **Top row**: Two images from a front-facing camera, taken a few seconds apart. The green area is the free space—the area to which the vehicle could physically move in the immediate future. Objects are displayed with 3D bounding boxes defining their sides (red for the rear, blue for the right side, yellow for the left side, and green for the front). Objects include vehicles, pedestrians, the inner edge of the self-lane marks (necessary for lateral control), other painted road and crosswalk marks, traffic signs, and traffic lights. Not shown are animals, poles and cones, sidewalks, railings, and general objects (e.g., a couch that fell from the back of a truck). Each object is then marked with a 3D position and velocity. **Bottom row**: A full physical model of the environment, rendered from the detected objects. (Images show Mobileye's vision-only system results). Images courtesy of Mobileye.



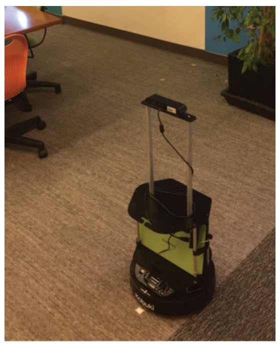


Figure 27.29 Navigation is tackled by decomposition into two problems: mapping and planning. With each successive time step, information from sensors is used to incrementally build an uncertain model of the world. This model along with the goal specification is passed to a planner that outputs the next action that the robot should take in order to achieve the goal. Models of the world can be purely geometric (as in classical SLAM), or semantic (as obtained via learning), or even topological (based on landmarks). The actual robot appears on the right. Figures courtesy of Saurabh Gupta.

CHAPTER 28

PHILOSOPHY, ETHICS, AND SAFETY OF AI

CHAPTER 29

THE FUTURE OF AI