1 INTRODUCTION

2 INTELLIGENT AGENTS

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 2.3 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 2.4 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure **??**.

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

function MODEL-BASED-REFLEX-AGENT($percept$) **returns** an action
persistent: $state$, the agent's current conception of the world state
$model$, a description of how the next state depends on current state and action
$rules$, a set of condition–action rules
$action$, the most recent action, initially none

$state \leftarrow$ UPDATE-STATE($state, action, percept, model$)
$rule \leftarrow$ RULE-MATCH($state, rules$)
$action \leftarrow rule$.ACTION
return $action$

Figure 2.8 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.

3 SOLVING PROBLEMS BY SEARCHING

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

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

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

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

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

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

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

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

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure **??**, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

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

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

Figure 3.16 A recursive implementation of depth-limited tree search.

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

Figure 3.17 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

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

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

Figure 3.24 The algorithm for recursive best-first search.

4 BEYOND CLASSICAL SEARCH

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

$current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
loop do
  $neighbor \leftarrow$ a highest-valued successor of $current$
  if neighbor.VALUE $\leq$ current.VALUE **then return** $current$.STATE
  $current \leftarrow neighbor$

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

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

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

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

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

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

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

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

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

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

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

```
function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
     plan_i ← OR-SEARCH(s_i, problem, 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. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

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

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

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

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

  if GOAL-TEST(s') **then return** $stop$
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  if s is not null
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow$ an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, $result[s', b]$, H)
  $s \leftarrow s'$
  return a

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

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

5 ADVERSARIAL SEARCH

```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best 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. The notation $\operatorname{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure **??**, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

6 CONSTRAINT SATISFACTION PROBLEMS

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components $(X,\ D,\ C)$
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    $(X_i,\ X_j) \leftarrow$ REMOVE-FIRST($queue$)
    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
  $revised \leftarrow false$
  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
      $revised \leftarrow true$
  return $revised$

Figure 6.3 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 (?) because it’s the third version developed in the paper.

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

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

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter **??**. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to 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 removed from the current assignment and a new value is tried.

```
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(var, v, current, csp)
     set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. 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.

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

$n \leftarrow$ number of variables in X
$assignment \leftarrow$ an empty assignment
$root \leftarrow$ any variable in X
$X \leftarrow$ 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**
  $assignment[X_i] \leftarrow$ any consistent value from D_i
  if there is no consistent value **then return** *failure*
return $assignment$

Figure 6.11 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.

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))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action$, t))
  $t \leftarrow t + 1$
  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.

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

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

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

Figure 7.8 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**" is used here as a logical operation on its two arguments, returning *true* or *false*.

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

   clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
   new ← { }
   loop do
      for each pair of clauses C_i, C_j in clauses do
         resolvents ← PL-RESOLVE(C_i, C_j)
         if resolvents contains the empty clause then return true
         new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

Figure 7.9 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

```
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 the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
     p ← POP(agenda)
     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 agenda
  return false
```

Figure 7.12 The forward-chaining algorithm for propositional logic. The *agenda* 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 as yet unknown. 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 $Q \Rightarrow P$.

```
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 7.14 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 flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for 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
     with probability p 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 7.15 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

```
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' ≤ 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, ¬ 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, ¬ 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 A*-GRAPH-SEARCH(problem)
```

Figure 7.17 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 decide what actions to take.

```
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 7.19 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.

8 FIRST-ORDER LOGIC

9 INFERENCE IN FIRST-ORDER LOGIC

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
     return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
     return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
     new ← { }
     for each rule in KB do
        (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
        for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                      for some p'1, ..., p'n in KB
           q' ← SUBST(θ, q)
           if q' does not unify with some sentence already in KB or new then
              add q' to new
              φ ← UNIFY(q', α)
              if φ is not fail then return φ
     add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very 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.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })
---------------------------------------------------------------------
generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
     (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
     for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
        yield θ'
---------------------------------------------------------------------
generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
     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.

```
procedure APPEND(ax, y, az, continuation)

  trail ← GLOBAL-TRAIL-POINTER()
  if ax = [ ] and UNIFY(y, az) then CALL(continuation)
  RESET-TRAIL(trail)
  a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
  if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure 9.8 Pseudocode representing the result of compiling the `Append` predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL(*continuation*) continues execution with the specified continuation.

10 CLASSICAL PLANNING

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

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

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

Figure 10.2 The simple spare tire problem.

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)$
$\quad \wedge\ Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C))$
$Goal(On(A, B) \wedge On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$
$(b \neq x) \wedge (b \neq y) \wedge (x \neq y),$
EFFECT: $On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \wedge Clear(b) \wedge Block(b) \wedge (b \neq x),$
EFFECT: $On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

Figure 10.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg\ Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg\ Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The “have cake and eat cake too” problem.

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
     if goals all non-mutex in S_t of graph then
        solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
        if solution ≠ failure then return solution
     if graph and nogoods have both leveled off then return failure
     graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

11 PLANNING AND ACTING IN THE REAL WORLD

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

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

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

Figure 11.1 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.

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

$Refinement(Navigate([a, b], [x, y]),$
  PRECOND: $a = x \ \wedge \ b = y$
  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 11.4 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
  loop do
     if 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-STATE, prefix)
     if hla is null then   /* so plan is primitive and outcome is its result */
        if outcome satisfies problem.GOAL  then return plan
     else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Figure 11.5 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*.

function ANGELIC-SEARCH($problem, hierarchy, initialPlan$) **returns** solution or $fail$

$frontier \leftarrow$ a FIFO queue with $initialPlan$ as the only element
loop do
  if EMPTY?($frontier$) **then return** $fail$
  $plan \leftarrow$ POP($frontier$) /* chooses the shallowest node in $frontier$ */
  if REACH$^+$($problem$.INITIAL-STATE, $plan$) intersects $problem$.GOAL **then**
    if $plan$ is primitive **then return** $plan$ /* REACH$^+$ is exact for primitive plans */
    $guaranteed \leftarrow$ REACH$^-$($problem$.INITIAL-STATE, $plan$) $\cap$ $problem$.GOAL
    if $guaranteed \neq \{\,\}$ and MAKING-PROGRESS($plan, initialPlan$) **then**
      $finalState \leftarrow$ any element of $guaranteed$
      return DECOMPOSE($hierarchy, problem$.INITIAL-STATE, $plan, finalState$)
    $hla \leftarrow$ some HLA in $plan$
    $prefix, suffix \leftarrow$ the action subsequences before and after hla in $plan$
    for each $sequence$ **in** REFINEMENTS($hla, outcome, hierarchy$) **do**
      $frontier \leftarrow$ INSERT(APPEND($prefix, sequence, suffix$), $frontier$)

---

function DECOMPOSE($hierarchy, s_0, plan, s_f$) **returns** a solution

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

Figure 11.8 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 $initialPlan$.

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

Figure 11.10 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. Note that each action must include the actor as an argument.

12 KNOWLEDGE REPRESENTATION

13 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 13.1 A decision-theoretic agent that selects rational actions.

14 PROBABILISTIC REASONING

```
function ENUMERATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayes net with variables {X} ∪ E ∪ Y   /* Y = hidden variables */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
     Q(x_i) ← ENUMERATE-ALL(bn.VARS, e_{x_i})
        where e_{x_i} is e extended with X = x_i
  return NORMALIZE(Q(X))
```

```
function ENUMERATE-ALL(vars, e) returns a real number
  if EMPTY?(vars) then return 1.0
  Y ← FIRST(vars)
  if Y has value y in e
     then return P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e)
     else return ∑_y P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e_y)
        where e_y is e extended with Y = y
```

Figure 14.9 The enumeration algorithm for answering queries on Bayesian networks.

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network specifying joint distribution P(X_1, ..., X_n)

  factors ← [ ]
  for each var in ORDER(bn.VARS) do
     factors ← [MAKE-FACTOR(var, e)|factors]
     if var is a hidden variable then factors ← SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Figure 14.10 The variable elimination algorithm for inference in Bayesian networks.

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

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

Figure 14.12 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, **e**, bn, N) **returns** an estimate of $\mathbf{P}(X|\mathbf{e})$
  inputs: X, the query variable
    e, observed values for variables **E**
    bn, a Bayesian network
    N, the total number of samples to be generated
  local variables: **N**, a vector of counts for each value of X, initially zero

  for j = 1 to N **do**
    x ← PRIOR-SAMPLE(bn)
    if x is consistent with **e then**
      N[x] ← **N**[x]+1 where x is the value of X in **x**
  return NORMALIZE(**N**)

Figure 14.13 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

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

  for j = 1 to N do
     x, w ← WEIGHTED-SAMPLE(bn, e)
     W[x] ← W[x] + w where x is the value of X in x
  return NORMALIZE(W)
```

```
function WEIGHTED-SAMPLE(bn, e) returns an event and a weight

  w ← 1; x ← an event with n elements initialized from e
  foreach variable X_i in X_1, ..., X_n do
     if X_i is an evidence variable with value x_i in e
        then w ← w × P(X_i = x_i | parents(X_i))
        else x[i] ← a random sample from P(X_i | parents(X_i))
  return x, w
```

Figure 14.14 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.

```
function GIBBS-ASK(X, e, bn, N) returns an estimate of P(X|e)
  local variables: N, a vector of counts for each value of X, initially zero
                   Z, the nonevidence variables in bn
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Z
  for j = 1 to N do
     for each Z_i in Z do
        set the value of Z_i in x by sampling from P(Z_i|mb(Z_i))
        N[x] ← N[x] + 1 where x is the value of X in x
  return NORMALIZE(N)
```

Figure 14.15 The Gibbs sampling algorithm for approximate inference in Bayesian networks; this version cycles through the variables, but choosing variables at random also works.

15 PROBABILISTIC REASONING OVER TIME

```
function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
  inputs: ev, a vector of evidence values for steps 1, ..., t
          prior, the prior distribution on the initial state, P(X_0)
  local variables: fv, a vector of forward messages for steps 0, ..., t
                   b, a representation of the backward message, initially all 1s
                   sv, a vector of smoothed estimates for steps 1, ..., t

  fv[0] ← prior
  for i = 1 to t do
     fv[i] ← FORWARD(fv[i − 1], ev[i])
  for i = t downto 1 do
     sv[i] ← NORMALIZE(fv[i] × b)
     b ← BACKWARD(b, ev[i])
  return sv
```

Figure 15.4 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 (**??**) and (**??**), respectively.

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 **T**
    d, the length of the lag for smoothing
  persistent: t, the current time, initially 1
    f, the forward message $\mathbf{P}(X_t|e_{1:t})$, initially hmm.PRIOR
    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 $\mathbf{P}(e_t|X_t)$
  if $t > d$ **then**
    $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}$, e_t)
    remove e_{t-d-1} from the beginning of $e_{t-d:t}$
    $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d}|X_{t-d})$
    $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
  else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
  $t \leftarrow t+1$
  if $t > d$ **then return** NORMALIZE($\mathbf{f} \times \mathbf{B1}$) **else return** null

Figure 15.6 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 (**??**).

function PARTICLE-FILTERING(**e**, N, dbn) **returns** a set of samples for the next time step
  inputs: **e**, the new incoming evidence
    N, the number of samples to be maintained
    dbn, a DBN with prior $\mathbf{P}(\mathbf{X}_0)$, transition model $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0)$, sensor model $\mathbf{P}(\mathbf{E}_1|\mathbf{X}_1)$
  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 15.17 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.

16 MAKING SIMPLE DECISIONS

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

  integrate $percept$ into D
  $j \leftarrow$ the value that maximizes $VPI(E_j) \ / \ Cost(E_j)$
  if $VPI(E_j) \ > \ Cost(E_j)$
    return REQUEST(E_j)
  else return the best action from D

Figure 16.9 Design of a simple 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.

17 MAKING COMPLEX DECISIONS

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

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (**??**).

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

  repeat
     U ← POLICY-EVALUATION(π, U, mdp)
     unchanged? ← true
     for each state s in S do
        if max_{a ∈ A(s)} Σ_{s′} P(s′ | s, a) U[s′] > Σ_{s′} P(s′ | s, π[s]) U[s′] then do
           π[s] ← argmax_{a ∈ A(s)} Σ_{s′} P(s′ | s, a) U[s′]
           unchanged? ← false
  until unchanged?
  return π
```

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

```
function POMDP-VALUE-ITERATION(pomdp, ε) returns a utility function
  inputs: pomdp, a POMDP with states S, actions A(s), transition model P(s′ | s, a),
             sensor model P(e | s), rewards R(s), discount γ
          ε, the maximum error allowed in the utility of any state
  local variables: U, U′, sets of plans p with associated utility vectors α_p

  U′ ← a set containing just the empty plan [ ], with α_[ ](s) = R(s)
  repeat
     U ← U′
     U′ ← 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 (??)
     U′ ← REMOVE-DOMINATED-PLANS(U′)
  until MAX-DIFFERENCE(U, U′) < ε(1 − γ)/γ
  return U
```

Figure 17.9 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.

18 LEARNING FROM EXAMPLES

function DECISION-TREE-LEARNING($examples, attributes, parent\_examples$) **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 ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
    tree ← a new decision tree with root test A
    for each value v_k of A do
        exs ← {e : e ∈ examples and e.A = v_k}
        subtree ← DECISION-TREE-LEARNING(exs, attributes − A, examples)
        add a branch to tree with label (A = v_k) and subtree subtree
    return tree
```

Figure 18.4 The decision-tree learning algorithm. The function IMPORTANCE is described in Section **??**. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

```
function CROSS-VALIDATION-WRAPPER(Learner, k, examples) returns a hypothesis

  local variables: errT, an array, indexed by size, storing training-set error rates
                   errV, an array, indexed by size, storing validation-set error rates
  for size = 1 to ∞ do
     errT[size], errV[size] ← CROSS-VALIDATION(Learner, size, k, examples)
     if errT has converged then do
        best_size ← the value of size with minimum errV[size]
        return Learner(best_size, examples)
```

```
function CROSS-VALIDATION(Learner, size, k, examples) returns two values:
        average training set error rate, average validation set error rate

  fold_errT ← 0; fold_errV ← 0
  for fold = 1 to k do
     training_set, validation_set ← PARTITION(examples, fold, k)
     h ← Learner(size, training_set)
     fold_errT ← fold_errT + ERROR-RATE(h, training_set)
     fold_errV ← fold_errV +ERROR-RATE(h, validation_set)
  return fold_errT/k, fold_errV/k
```

Figure 18.7 An algorithm to select the model that has the lowest error rate on validation data by building models of increasing complexity, and choosing the one with best empirical error rate on validation data. Here $errT$ means error rate on the training data, and $errV$ means error rate on the validation data. $Learner(size, examples)$ returns a hypothesis whose complexity is set by the parameter $size$, and which is trained on the $examples$. PARTITION(*examples*, *fold*, *k*) splits *examples* into two subsets: a validation set of size N/k and a training set with all the other examples. The split is different for each value of *fold*.

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

  if examples is empty then return the trivial decision list No
  t ← 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 ← Yes else o ← No
  return a decision list with initial test t and outcome o and remaining tests given by
      DECISION-LIST-LEARNING(examples − examples_t)
```

Figure 18.10 An algorithm for learning decision lists.

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights w_{i,j}, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
     for each weight w_{i,j} in network do
        w_{i,j} ← a small random number
     for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
           a_i ← x_i
        for ℓ = 2 to L do
           for each node j in layer ℓ do
              in_j ← Σ_i w_{i,j} a_i
              a_j ← g(in_j)
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
           Δ[j] ← g′(in_j) × (y_j − a_j)
        for ℓ = L − 1 to 1 do
           for each node i in layer ℓ do
              Δ[i] ← g′(in_i) Σ_j w_{i,j} Δ[j]
        /* Update every weight in network using deltas */
        for each weight w_{i,j} in network do
           w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
  until some stopping criterion is satisfied
  return network
```

Figure 18.23 The back-propagation algorithm for learning in multilayer networks.

```
function ADABOOST(examples, L, K) returns a weighted-majority hypothesis
  inputs: examples, set of N labeled examples (x_1, y_1), ..., (x_N, y_N)
          L, a learning algorithm
          K, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially 1/N
                   h, a vector of K hypotheses
                   z, a vector of K hypothesis weights

  for k = 1 to K do
     h[k] ← L(examples, w)
     error ← 0
     for j = 1 to N do
        if h[k](x_j) ≠ y_j then error ← error + w[j]
     for j = 1 to N do
        if h[k](x_j) = y_j then w[j] ← w[j] · error/(1 − error)
     w ← NORMALIZE(w)
     z[k] ← log (1 − error)/error
  return WEIGHTED-MAJORITY(h, z)
```

Figure 18.33 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**.

19 KNOWLEDGE IN LEARNING

```
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 19.2 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 ← the set of all hypotheses
  for each example e in examples do
     if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  return V
-----------------------------------------------------------------
function VERSION-SPACE-UPDATE(V, e) returns an updated version space

  V ← {h ∈ V : h is consistent with e}
```

Figure 19.3 The version space learning algorithm. It finds a subset of V that is consistent with all the $examples$.

```
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 19.8 An algorithm for finding a minimal consistent determination.

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 19.12 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.

20 LEARNING PROBABILISTIC MODELS

21 REINFORCEMENT LEARNING

function PASSIVE-ADP-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \;/\; N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL? **then** $s, a \leftarrow$ null **else** $s, a \leftarrow s', \pi[s']$
  return a

Figure 21.2 A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page **??**.

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

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence, as described in the text.

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, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

Figure 21.8 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 because the Q-value of a state can be related directly to those of its neighbors.

22 NATURAL LANGUAGE PROCESSING

```
function HITS(query) returns pages with hub and authority numbers

  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY ← Σ_i INLINK_i(p).HUB
      p.HUB ← Σ_i OUTLINK_i(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

Figure 22.1 The HITS algorithm for computing hubs and authorities with respect to a query. RELEVANT-PAGES fetches the pages that match the query, and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page's score by the sum of the squares of all pages' scores (separately for both the authority and hubs scores).

23 NATURAL LANGUAGE FOR COMMUNICATION

```
function CYK-PARSE(words, grammar) returns P, a table of probabilities
  N ← LENGTH(words)
  M ← the number of nonterminal symbols in grammar
  P ← an array of size [M, N, N], initially all 0
  /* Insert lexical rules for each word */
  for i = 1 to N do
    for each rule of form (X → words_i [p]) do
      P[X, i, 1] ← p
  /* Combine first and second parts of right-hand sides of rules, from short to long */
  for length = 2 to N do
    for start = 1 to N − length + 1 do
      for len1 = 1 to N − 1 do
        len2 ← length − len1
        for each rule of the form (X → Y Z [p]) do
          P[X, start, length] ← MAX(P[X, start, length],
                                P[Y, start, len1] × P[Z, start + len1, len2] × p)
  return P
```

Figure 23.4 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable derivation for the whole sequence and for each subsequence. It returns the whole table, P, in which an entry $P[X, start, len]$ is the probability of the most probable X of length len starting at position $start$. If there is no X of that size at that location, the probability is 0.

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

Figure 23.5 Annotated tree for the sentence "Her eyes were glazed as if she didn't hear or even see him." from the Penn Treebank. Note that in this grammar there is a distinction between an object noun phrase (*NP*) and a subject noun phrase (*NP-SBJ*). Note also 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 VPs, [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].

24 PERCEPTION

25 ROBOTICS

```
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X'|X, v, ω), P(z|z*), m) returns
a set of samples for the next time step
  inputs: a, robot velocities v and ω
          z, range scan z_1, ..., z_M
          P(X'|X, v, ω), motion model
          P(z|z*), range sensor noise model
          m, 2D map of the environment
  persistent: S, a vector of samples of size N
  local variables: W, a vector of weights of size N
                   S', a temporary vector of particles of size N
                   W', a vector of weights of size N

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

Figure 25.9 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

26 PHILOSOPHICAL FOUNDATIONS

27 AI: THE PRESENT AND FUTURE

28 MATHEMATICAL BACKGROUND

29 NOTES ON LANGUAGES AND ALGORITHMS

generator POWERS-OF-2() **yields** ints
  $i \leftarrow 1$
  while $true$ **do**
    yield i
    $i \leftarrow 2 \times i$

---

for p **in** POWERS-OF-2() **do**
  PRINT(p)

Figure 29.1 Example of a generator function and its invocation within a loop.