function TABLE-DRIVEN-AGENT(\textit{percept}) \textbf{returns} an action  
\textbf{persistent}: \textit{percepts}, a sequence, initially empty \linebreak \hspace{1em} \textit{table}, a table of actions, indexed by percept sequences, initially fully specified \linebreak \hspace{1em} append \textit{percept} to the end of \textit{percepts} \linebreak \hspace{1em} \textit{action} \leftarrow \text{LOOKUP}(\textit{percepts}, \textit{table}) \linebreak \hspace{1em} \textbf{return} \textit{action}  

\textbf{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([\textit{location},\textit{status}]) \textbf{returns} an action  
if \textit{status} = \textit{Dirty} then return \textit{Suck} \linebreak else if \textit{location} = A then return \textit{Right} \linebreak else if \textit{location} = B then return \textit{Left}  

\textbf{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(\textit{percept}) \textbf{returns} an action  
\textbf{persistent}: \textit{rules}, a set of condition-action rules \linebreak \hspace{1em} \textit{state} \leftarrow \text{INTERPRET-INPUT}(\textit{percept}) \linebreak \hspace{1em} \textit{rule} \leftarrow \text{RULE-MATCH}(\textit{state}, \textit{rules}) \linebreak \hspace{1em} \textit{action} \leftarrow \textit{rule}.\textit{ACTION} \linebreak \hspace{1em} \textbf{return} \textit{action}  

\textbf{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 ← UPDATE-STATE( state, action, percept, model )
rule ← RULE-MATCH( state, rules )
action ← 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.
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**
   
   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**
   
   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**
   
   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 \textsc{Iterative-Deepening-Search}(\textit{problem}) \textbf{returns} a solution, or failure
\begin{algorithmic}
  \State \textbf{for} depth = 0 \textbf{to} \infty \textbf{do}
  \State \textit{result} ← \textsc{Depth-Limited-Search}(\textit{problem}, depth)
  \If {\textit{result} $\neq$ cutoff} \textbf{return} \textit{result}\EndIf
\end{algorithmic}

\textbf{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 \textit{failure}, meaning that no solution exists.

function \textsc{Recursive-Best-First-Search}(\textit{problem}) \textbf{returns} a solution, or failure
\begin{algorithmic}
  \State \textbf{return} \textsc{RBFS}(\textit{problem}, \textsc{Make-Node}(\textit{problem}.\textsc{Initial-State}), \infty)
\end{algorithmic}

function \textsc{RBFS}(\textit{problem}, \textit{node}, \textit{f-limit}) \textbf{returns} a solution, or failure and a new \textit{f}-cost limit
\begin{algorithmic}
  \If {\textit{problem}.\textsc{Goal-Test}(\textit{node}.\textsc{State})} \textbf{return} \textsc{Solution}(\textit{node})\EndIf
  \State \textit{successors} ← []
  \For {\textit{each} \textit{action} in \textit{problem}.\textsc{Actions}(\textit{node}.\textsc{State})}
    \State add \textsc{Child-Node}(\textit{problem}, \textit{node}, \textit{action}) into \textit{successors}
  \EndFor
  \If {\textit{successors} is empty} \textbf{return} failure, \infty\EndIf
  \For {\textit{each} \textit{s} in \textit{successors}} /* update \textit{f} with value from previous search, if any */
    \State \textit{s}.\textit{f} ← \max(\textit{s}.g + \textit{s}.h, \textit{node}.f)
  \EndFor
  \Loop
    \State \textit{best} ← the lowest \textit{f}-value node in \textit{successors}
    \If {\textit{best}.\textit{f} $> \textit{f-limit}$} \textbf{return} failure, \textit{best}.f\EndIf
    \State \textit{alternative} ← the second-lowest \textit{f}-value among \textit{successors}
    \State \textit{result}, \textit{best}.f ← \textsc{RBFS}(\textit{problem}, \textit{best}, \text{min}(\textit{f-limit}, \textit{alternative}))
    \If {\textit{result} $\neq$ failure} \textbf{return} \textit{result}\EndIf
  \EndLoop
\end{algorithmic}

\textbf{Figure 3.24} The algorithm for recursive best-first search.
function **HILL-CLIMBING** (*problem*) returns a state that is a local maximum

\[
\text{current} \leftarrow \text{MAKE-NODE}(*\text{problem}.\text{INITIAL-STATE}*)
\]

**loop do**

\[
\text{neighbor} \leftarrow \text{a highest-valued successor of current}
\]

*if neighbor.VALUE \( \leq \) current.VALUE then return current.STATE

\[
\text{current} \leftarrow \text{neighbor}
\]

**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

\[
\text{schedule}, \text{a mapping from time to “temperature”}
\]

\[
\text{current} \leftarrow \text{MAKE-NODE}(*\text{problem}.\text{INITIAL-STATE}*)
\]

**for** \( t = 1 \) **to** \( \infty \) **do**

\[
T \leftarrow \text{schedule}(t)
\]

*if* \( T = 0 \) *then return current

\[
\text{next} \leftarrow \text{a randomly selected successor of current}
\]

\[
\Delta E \leftarrow \text{next.VALUE} - \text{current.VALUE}
\]

*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, in states do
    plan_s ← OR-SEARCH(s, problem, path)
    if plan_s = 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 | 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\[ s' \] ← ACTIONS\( (s') \)
if \( s \) is not null then
result\[ s, a \] ← \( s' \)
add \( s \) to the front of unbacktracked\[ s' \]
if untried\[ s' \] is empty then
if unbacktracked\[ s' \] is empty then return stop
else \( a \) ← an action \( b \) such that result\[ s', b \] = POP(unbacktracked\[ s' \])
else \( a \) ← POP(untried\[ s' \])
\( s ← s' \)
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 \( H[s'] \) ← \( h(s') \)
if \( s \) is not null then
result\[ s, a \] ← \( s' \)
\( H[s] ← \min \{ \text{LRTA*-COST}(s, b, result[s, b], H) \}_{b \in \text{ACTIONS}(s)} \)
\( a ← \) an action \( b \) in \( \text{ACTIONS}(s') \) that minimizes \( \text{LRTA*-COST}(s', b, \text{result}[s', b], H) \)
\( s ← s' \)
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.
function $\text{MINIMAX-DECISION}(state)$ returns an action
  return $\arg\max_{a \in \text{ACTIONS}(state)} \text{MIN-VALUE}(\text{RESULT}(state, a))$

function $\text{MAX-VALUE}(state)$ returns a utility value
  if $\text{TERMINAL-TEST}(state)$ then return $\text{UTILITY}(state)$
  $v \leftarrow -\infty$
  for each $a$ in $\text{ACTIONS}(state)$ do
    $v \leftarrow \max(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$
  return $v$

function $\text{MIN-VALUE}(state)$ returns a utility value
  if $\text{TERMINAL-TEST}(state)$ then return $\text{UTILITY}(state)$
  $v \leftarrow \infty$
  for each $a$ in $\text{ACTIONS}(state)$ do
    $v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$
  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 $\text{MAX-VALUE}$ and $\text{MIN-VALUE}$ go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\arg\max_{a \in S} f(a)$ computes the element $a$ of set $S$ that has the maximum value of $f(a)$. 
function **ALPHA-BETA-SEARCH**(state) returns an action
\[ v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]
return the action in ACTIONS(state) with value \( v \)

function **MAX-VALUE**(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return \( \text{UTILITY}(state) \)
\[ v \leftarrow -\infty \]
for each \( a \) in ACTIONS(state) do
\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta)) \]
if \( v \geq \beta \) then return \( v \)
\[ \alpha \leftarrow \text{MAX}(\alpha, v) \]
return \( v \)

function **MIN-VALUE**(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return \( \text{UTILITY}(state) \)
\[ v \leftarrow +\infty \]
for each \( a \) in ACTIONS(state) do
\[ v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta)) \]
if \( v \leq \alpha \) then return \( v \)
\[ \beta \leftarrow \text{MIN}(\beta, v) \]
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 \( \alpha \) and \( \beta \) (and the bookkeeping to pass these parameters along).
6 CONSTRAINT SATISFACTION PROBLEMS

**Function AC-3**

**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  
(Xi, Xj) ← REMOVE-FIRST(queue)  
if REVISE(csp, Xi, Xj) then  
    if size of Di = 0 then return false  
    for each Xk in Xi.NEIGHBORS - {Xj} do  
        add (Xk, Xi) to queue  
return true

**Function REVISE**

**Function** REVISE(csp, Xi, Xj) **returns** true iff we revise the domain of Xi  
revised ← false  
for each x in Di do  
    if no value y in Dj allows (x, y) to satisfy the constraint between Xi and Xj then  
        delete x from Di  
        revised ← 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 ← INERENCE\((csp, var, value)\)
    if inferences \neq failure then
      add inferences to assignment
      result ← BACKTRACK\((assignment, csp)\)
      if result \neq 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 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INERENCE can optionally be used to impose arc-, path-, or 4-consistency, as desired. If a value choice leads to failure (noticed either by INERENCE or by BACKTRACK), then value assignments (including those made by INERENCE) 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.
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))

\( t \leftarrow t + 1 \)

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(\(KB, \alpha\)) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
\(\alpha\), the query, a sentence in propositional logic

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

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

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 \(\text{model}\) 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(\(KB, \alpha\)) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
\(\alpha\), the query, a sentence in propositional logic

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

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
    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-WUMPS-A gENT (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 \cap safe, safe)
if plan is empty and ASK(KB, HaveArrow^t) = true then
   possible_wumpus ← \{ [x, y] : ASK(KB, ¬ W^t_{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 \cap 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.
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.
function \textsc{Unify}(x, y, \theta) \text{ returns} a substitution to make \( x \) and \( y \) identical

\textbf{inputs:} \( x \), a variable, constant, list, or compound expression
\quad \text{\( y \), a variable, constant, list, or compound expression}
\quad \text{\( \theta \), the substitution built up so far (optional, defaults to empty)}

\textbf{if} \( \theta = \text{failure} \) \textbf{then return} \text{failure}
\textbf{else if} \( x = y \) \textbf{then return} \( \theta \)
\textbf{else if} \( \text{VARIABLE}(x) \) \textbf{then return} \textsc{Unify-Var}(x, y, \theta)
\textbf{else if} \( \text{VARIABLE}(y) \) \textbf{then return} \textsc{Unify-Var}(y, x, \theta)
\textbf{else if} \( \text{COMPONENT}(x) \) \textbf{and} \( \text{COMPONENT}(y) \) \textbf{then}
\quad \textbf{return} \textsc{Unify}(x.\text{Args}, y.\text{Args}, \textsc{Unify}(x.\text{Op}, y.\text{Op}, \theta))
\textbf{else if} \( \text{LIST}(x) \) \textbf{and} \( \text{LIST}(y) \) \textbf{then}
\quad \textbf{return} \textsc{Unify}(x.\text{Rest}, y.\text{Rest}, \textsc{Unify}(x.\text{First}, y.\text{First}, \theta))
\textbf{else return} \text{failure}

\textbf{function} \textsc{Unify-Var}(\text{var}, x, \theta) \text{ returns} a substitution

\textbf{if} \{ \text{var}/\text{val} \} \in \theta \textbf{then return} \textsc{Unify}(\text{val}, x, \theta)
\textbf{else if} \{ x/\text{val} \} \in \theta \textbf{then return} \textsc{Unify}(\text{var}, \text{val}, \theta)
\textbf{else if} \( \text{Occur-Check}(\text{var}, x) \) \textbf{then return} \text{failure}
\textbf{else return} \text{add} \{ \text{var}/x \} \text{ to} \ \theta

\textbf{Figure 9.1} The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution \( \theta \) that is the argument to \textsc{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 \( (A, B) \).
**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.

**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

\[
\begin{align*}
\text{Init} &: \text{At}(C_1, \text{SFO}) \land \text{At}(C_2, \text{JFK}) \land \text{At}(P_1, \text{SFO}) \land \text{At}(P_2, \text{JFK}) \\
&\land \text{Cargo}(C_1) \land \text{Cargo}(C_2) \land \text{Plane}(P_1) \land \text{Plane}(P_2) \\
&\land \text{Airport}(\text{JFK}) \land \text{Airport}(\text{SFO}) \\
\text{Goal} &: \text{At}(C_1, \text{JFK}) \land \text{At}(C_2, \text{SFO}) \\
\text{Action} &: \text{Load}(c, p, a), \\
\text{PRECOND} &: \text{At}(c, a) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
\text{EFFECT} &: \neg \text{At}(c, a) \land \text{In}(c, p) \\
\text{Action} &: \text{Unload}(c, p, a), \\
\text{PRECOND} &: \text{In}(c, p) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
\text{EFFECT} &: \text{At}(c, a) \land \neg \text{In}(c, p) \\
\text{Action} &: \text{Fly}(p, \text{from}, \text{to}), \\
\text{PRECOND} &: \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}) \\
\text{EFFECT} &: \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to}) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{Init} &: \text{Tire}(\text{Flat}) \land \text{Tire}(\text{Spare}) \land \text{At}(\text{Flat, Axle}) \land \text{At}(\text{Spare, Trunk}) \\
\text{Goal} &: \text{At}(\text{Spare, Axle}) \\
\text{Action} &: \text{Remove}(\text{obj, loc}), \\
\text{PRECOND} &: \text{At}(\text{obj, loc}) \\
\text{EFFECT} &: \neg \text{At}(\text{obj, loc}) \land \text{At}(\text{obj, Ground}) \\
\text{Action} &: \text{PutOn}(t, \text{Axle}), \\
\text{PRECOND} &: \text{Tire}(t) \land \text{At}(t, \text{Ground}) \land \neg \text{At}(\text{Flat, Axle}) \\
\text{EFFECT} &: \neg \text{At}(t, \text{Ground}) \land \text{At}(t, \text{Axle}) \\
\text{Action} &: \text{LeaveOvernight}, \\
\text{PRECOND} &: \\
\text{EFFECT} &: \neg \text{At}(\text{Spare, Ground}) \land \neg \text{At}(\text{Spare, Axle}) \land \neg \text{At}(\text{Spare, Trunk}) \\
&\land \neg \text{At}(\text{Flat, Ground}) \land \neg \text{At}(\text{Flat, Axle}) \land \neg \text{At}(\text{Flat, Trunk}) \\
\end{align*}
\]

Figure 10.2 The simple spare tire problem.
Init(On(A, Table) \land On(B, Table) \land On(C, A) \\
\land Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C))
Goal(On(A, B) \land On(B, C))
Action(Move\(b, x, y\)),
\hspace{1em}\text{PRECOND: } On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land \\
(b \neq x) \land (b \neq y) \land (x \neq y),
\hspace{1em}\text{EFFECT: } On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))
Action(MoveToTable\(b, x\)),
\hspace{1em}\text{PRECOND: } On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),
\hspace{1em}\text{EFFECT: } On(b, Table) \land Clear(x) \land \neg On(b, x))

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

Init(Have(Cake))
Goal(Have(Cake) \land Eaten(Cake))
Action(Eat(Cake))
\hspace{1em}\text{PRECOND: } Have(Cake)
\hspace{1em}\text{EFFECT: } \neg Have(Cake) \land Eaten(Cake))
Action(Bake(Cake))
\hspace{1em}\text{PRECOND: } \neg Have(Cake)
\hspace{1em}\text{EFFECT: } Have(Cake))

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

function \textsc{GraphPlan}(\text{problem}) \text{ returns } \text{solution or failure}
\hspace{1em}graph \leftarrow \textsc{Initial-Planning-Graph}(\text{problem})
\hspace{1em}goals \leftarrow \textsc{Conjuncts}(\text{problem}.\text{Goal})
\hspace{1em}nogoods \leftarrow \text{an empty hash table}
\hspace{1em}\text{for } tl = 0 \text{ to } \infty \text{ do}
\hspace{2em}if goals all non-mutex in } S_t \text{ of } graph \text{ then}
\hspace{3em}solution \leftarrow \textsc{Extract-Solution}(graph, goals, \text{NUMLEVELS}(graph), nogoods)
\hspace{3em}if solution \neq failure \text{ then return } solution
\hspace{2em}if graph and nogoods have both leveled off \text{ then return } failure
\hspace{1em}graph \leftarrow \textsc{Expand-Graph}(graph, \text{problem})

Figure 10.9 The \textsc{GraphPlan} algorithm. \textsc{GraphPlan} calls \textsc{Expand-Graph} to add a level until either a solution is found by \textsc{Extract-Solution}, or no solution is possible.
PLANNING AND ACTING IN THE REAL WORLD

\[ \text{Jobs}\{\text{AddEngine1} \prec \text{AddWheels1} \prec \text{Inspect1}\}, \]
\[ \{\text{AddEngine2} \prec \text{AddWheels2} \prec \text{Inspect2}\}\]

\[ \text{Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))}\]

\[ \text{Action(AddEngine1, Duration:30, Use: EngineHoists(1))}\]
\[ \text{Action(AddEngine2, Duration:60, Use: EngineHoists(1))}\]
\[ \text{Action(AddWheels1, Duration:30, Consume:LugNuts(20), Use: WheelStations(1))}\]
\[ \text{Action(AddWheels2, Duration:15, Consume:LugNuts(20), Use: WheelStations(1))}\]
\[ \text{Action(Inspect, Duration:10, Use: Inspectors(1))}\]

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  \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 11.4 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)
   end loop

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 \textsc{Angelicsearch}(problem, hierarchy, initialPlan) \textbf{returns} solution or fail

frontier \leftarrow \text{a FIFO queue with } initialPlan \text{ as the only element}

\textbf{loop} do
  \textbf{if} \text{EMPTY?}(frontier) \textbf{then} return fail
  plan \leftarrow \text{POP}(frontier) /* chooses the shallowest node in frontier */
  \textbf{if} \text{REACH}^+(\text{problem.}\text{initial-state}, \text{plan}) \text{ intersects } \text{problem}.\text{goal} \textbf{then}
    \textbf{if} plan \text{ is primitive} \text{ then} return plan /* \text{REACH}^+ \text{ is exact for primitive plans */}
    \text{guaranteed} \leftarrow \text{REACH}^-(\text{problem.}\text{initial-state}, \text{plan}) \cap \text{problem.}\text{goal}.
    \textbf{if} \text{guaranteed} \neq \emptyset \textbf{ and } \text{\textsc{Making-progress}}(plan, initialPlan) \textbf{then}
      \text{finalState} \leftarrow \text{any element of guaranteed}
      \textbf{return} \text{\textsc{Decompose}}(hierarchy, \text{problem.}\text{initial-state}, \text{plan}, \text{finalState})
    \text{hla} \leftarrow \text{some HLA in plan}
    prefix, suffix \leftarrow \text{the action subsequences before and after } hla \text{ in plan}
    \textbf{for each} sequence \text{ in } \text{\textsc{Refinements}}(hla, outcome, hierarchy) \textbf{do}
      \text{frontier} \leftarrow \text{\textsc{Insert}}(\text{\textsc{Append}}(prefix, sequence, suffix), frontier)

function \textsc{Decompose}(hierarchy, s_i, plan, s_f) \textbf{returns} a solution

solution \leftarrow \text{an empty plan}

\textbf{while} plan \text{ is not empty} \textbf{do}
  \text{action} \leftarrow \text{\textsc{Remove-last}}(plan)
  s_i \leftarrow \text{a state in } \text{REACH}^-(s_0, plan) \text{ such that } s_f \in \text{REACH}^-(s_i, action)
  \text{problem} \leftarrow \text{a problem with } \text{\textsc{Initial-state}} = s_i \text{ and } \text{\textsc{Goal}} = s_f
  \text{solution} \leftarrow \text{\textsc{Append}}(\text{\textsc{Angelicsearch}}(\text{problem, hierarchy, action}), \text{solution})
  s_f \leftarrow s_i

\textbf{return} solution

\textbf{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 \text{\textsc{Making-progress}} \text{ checks to make sure that we aren’t stuck in an infinite regression of refinements. At top level, call } \text{\textsc{Angelicsearch}} \text{ with } [\text{Act}] \text{ as the } \text{\textit{initialPlan}}.

\textit{Actors(A, B)}

\text{Init}(\text{At}(A, \text{LeftBaseline}) \land \text{At}(B, \text{RightNet}) \land
  \text{Approaching}(\text{Ball, RightBaseline})) \land \text{Partner}(A, B) \land \text{Partner}(B, A)

\text{Goal}(\text{Returned}(\text{Ball}) \land (\text{At}(a, \text{RightNet}) \lor \text{At}(a, \text{LeftNet}))

\text{Action}(\text{Hit}(\text{actor, Ball}),
  \text{\textbf{PRECOND:}} \text{Approaching}(\text{Ball, loc}) \land \text{At}(\text{actor, loc})
  \text{\textbf{EFFECT:}} \text{Returned}(\text{Ball}))

\text{Action}(\text{Go}(\text{actor, to}),
  \text{\textbf{PRECOND:}} \text{At}(\text{actor, loc}) \land \text{to} \neq \text{loc},
  \text{\textbf{EFFECT:}} \text{At}(\text{actor, to}) \land \neg \text{At}(\text{actor, loc}))

\textbf{Figure 11.10} The doubles tennis problem. Two actors \textit{A} and \textit{B} are playing together and can be in one of four locations: \textit{LeftBaseline}, \textit{RightBaseline}, \textit{LeftNet}, and \textit{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
### Quantifying Uncertainty

**Function** DT-AGENT( percept ) returns an action

- **Persistent:** belief<sub>state</sub>, probabilistic beliefs about the current state of the world
- **Action:** the agent’s action

Update belief<sub>state</sub> based on action and percept

Calculate outcome probabilities for actions, given action descriptions and current belief<sub>state</sub>

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.
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 \( \{ X \} \cup E \cup Y \) / * \( Y \) = hidden variables */

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

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

function **ENUMERATE-ALL**(\( \text{vars}, e \)) returns a real number
if \( \text{EMPTY}(\text{vars}) \) then return 1.0
\( Y \leftarrow \text{FIRST}(\text{vars}) \)
if \( Y \) has value \( y \) in \( e \)
then return \( P(y \mid \text{parents}(Y)) \times \text{ENUMERATE-ALL}(\text{REST}(\text{vars}), e) \)
else return \( \sum_y P(y \mid \text{parents}(Y)) \times \text{ENUMERATE-ALL}(\text{REST}(\text{vars}), e_y) \)
where \( e_y \) is \( e \) extended with \( Y = y \)

**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 $P(X_1, \ldots, X_n)$

$x \leftarrow$ an event with $n$ elements

foreach variable $X_i$ in $X_1, \ldots, X_n$ do
  $x[i] \leftarrow$ a random sample from $P(X_i | \text{parents}(X_i))$
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 $P(X|e)$
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
  $x \leftarrow$ PRIOR-SAMPLE($bn$)
  if $x$ is consistent with $e$ then
    $N[x] \leftarrow N[x]+1$ 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 \( P(X_1, \ldots, X_n) \)
- \( N \), 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
  \( x, w \leftarrow\) WEIGHTED-SAMPLE\((bn, e)\)
  \( W[x] \leftarrow W[x] + w \) where \( x \) is the value of \( X \) in \( x \)

return NORMALIZE\((W)\)

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

\( w \leftarrow 1; x \leftarrow \) an event with \( n \) elements initialized from \( e \)

foreach variable \( X_i \) in \( X_1, \ldots, X_n \) do
  if \( X_i \) is an evidence variable with value \( x_i \) in \( e \)
    then \( w \leftarrow w \times P(X_i = x_i | \text{parents}(X_i)) \)
  else \( x[i] \leftarrow \) a random sample from \( P(X_i | \text{parents}(X_i)) \)

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 \( P(Z_i | mb(Z_i)) \)
  \( N[x] \leftarrow N[x] + 1 \) 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.
PROBABILISTIC REASONING OVER TIME

function \textsc{Forward-Backward}(\textit{ev, prior}) returns a vector of probability distributions
inputs: \textit{ev}, a vector of evidence values for steps 1, \ldots, \textit{t}
\textit{prior}, the prior distribution on the initial state, \textit{P(X_0)}
local variables: \textit{fv}, a vector of forward messages for steps 0, \ldots, \textit{t}
\textit{b}, a representation of the backward message, initially all 1s
\textit{sv}, a vector of smoothed estimates for steps 1, \ldots, \textit{t}

\textit{fv}[0] \leftarrow \textit{prior}
for \textit{i} = 1 to \textit{t} do
\hspace{1em} \textit{fv}[i] \leftarrow \textsc{Forward}(\textit{fv}[i-1], \textit{ev}[i])
for \textit{i} = \textit{t} downto 1 do
\hspace{1em} \textit{sv}[i] \leftarrow \textsc{Normalize}(\textit{fv}[i] \times \textit{b})
\hspace{1em} \textit{b} \leftarrow \textsc{Backward}(\textit{b}, \textit{ev}[i])
return \textit{sv}

Figure 15.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The \textsc{Forward} and \textsc{Backward} operators are defined by Equations (??) and (??), respectively.
function FIXED-LAG-SMOOTHING(eₜ, hmm, d) returns a distribution over Xₜ₋₉
inputs: eₜ, the current evidence for time step t
hmm, a hidden Markov model with S × S transition matrix T
d, the length of the lag for smoothing
persistent: t, the current time, initially 1
f, the forward message P(Xₜ|e₁₉), initially hmm.prior
B, the d-step backward transformation matrix, initially the identity matrix
eₜ₋ₙ₋₁, double-ended list of evidence from t – d to t, initially empty
local variables: Oₜ₋ₙ₋₁, Oₜ, diagonal matrices containing the sensor model information
add eₜ to the end of eₜ₋ₙ₋₁
Oₜ ← diagonal matrix containing P(eₜ|Xₜ)
if t > d then
  f ← FORWARD(f, eₜ)
  remove eₜ₋ₙ₋₁ from the beginning of eₜ₋ₙ₋₁
  Oₜ₋ₙ₋₁ ← diagonal matrix containing P(eₜ₋ₙ₋₁|Xₜ₋ₙ₋₁)
  B ← Oₜ₋ₙ₋₁ T⁻¹ BTOₜ
else B ← BTOₜ
t ← t + 1
if t > d then return NORMALIZE(f × 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(f × B1) is just α f × 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 P(X₀), transition model P(X₁|X₀), sensor model P(E₁|X₁)
persistent: S, a vector of samples of size N, initially generated from P(X₀)
local variables: W, a vector of weights of size N
for i = 1 to N do
  S[i] ← sample from P(X₁ | X₀ = S[i]) /* step 1 */
  W[i] ← P(e | X₁ = S[i]) /* step 2 */
S ← 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.
function VALUE-ITERATION(mdp, \( \epsilon \)) returns a utility function

**inputs:** mdp, an MDP with states \( S \), actions \( A(s) \), transition model \( P(s' | s, a) \),
rewards \( R(s) \), 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 change in the utility of any state in an iteration

repeat

\[ U \leftarrow U'; \delta \leftarrow 0 \]

for each state \( s \) in \( S \) do

\[ U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U'[s'] \]

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 \( \max_{a \in A(s)} \sum_{s'} P(s' | s, a) \ U[s'] > \sum_{s'} P(s' | s, \pi[s]) \ U[s'] \) then do
            \( \pi[s] \leftarrow \arg\max_{a \in A(s)} \sum_{s'} P(s' | s, a) \ U[s'] \)
            unchanged? ← false
        until unchanged?
    return π

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

function POMDP-VALUE-ITERATION(pomdp, \( \epsilon \)) 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 \( \gamma \)
\( \epsilon \), 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 just the empty plan [], with \( \alpha_p[[]] = R(s) \)
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') < \( \epsilon (1 - \gamma) / \gamma \)
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.
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
    $A \leftarrow \text{argmax}_{a \in \text{attributes}} \text{IMPORTANCE}(a, \text{examples})$
    tree $\leftarrow$ a new decision tree with root test $A$
    for each value $v_k$ of $A$ do
        $exs \leftarrow \{e : e \in \text{examples} \text{ and } e.A = v_k\}$
        subtree $\leftarrow$ DECISION-TREE-LEARNING(exs, attributes $\setminus A$, 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 \infty 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 \( \mathbf{x} \) and output vector \( \mathbf{y} \)

network, a multilayer network with \( L \) layers, weights \( w_{i,j} \), activation function \( g \)

local variables: \( \Delta \), a vector of errors, indexed by network node

repeat
  for each weight \( w_{i,j} \) in network do
    \( w_{i,j} \leftarrow \) a small random number
  for each example \((\mathbf{x}, \mathbf{y})\) in examples do
    /* Propagate the inputs forward to compute the outputs */
    for each node \( i \) in the input layer do
      \( a_i \leftarrow x_i \)
    for \( \ell = 2 \) to \( L \) do
      for each node \( j \) in layer \( \ell \) do
        \( \text{in}_j \leftarrow \sum_i w_{i,j} a_i \)
        \( a_j \leftarrow g(\text{in}_j) \)
    /* Propagate deltas backward from output layer to input layer */
    for each node \( j \) in the output layer do
      \( \Delta[j] \leftarrow g'(\text{in}_j) \times (y_j - a_j) \)
    for \( \ell = L - 1 \) to \( 1 \) do
      for each node \( i \) in layer \( \ell \) do
        \( \Delta[i] \leftarrow g'(\text{in}_i) \sum_j w_{i,j} \Delta[j] \)
    /* Update every weight in network using deltas */
    for each weight \( w_{i,j} \) in network do
      \( w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j] \)
  until some stopping criterion is satisfied

return network

Figure 18.23 The back-propagation algorithm for learning in multilayer networks.
function \textsc{AdaBoost}(\textit{examples}, \textit{L}, \textit{K}) returns a weighted-majority hypothesis

\textbf{inputs:} \textit{examples}, set of \textit{N} labeled examples \((x_1, y_1), \ldots, (x_N, y_N)\)
\textit{L}, a learning algorithm
\(\textit{K}\), the number of hypotheses in the ensemble

\textbf{local variables:} \textit{w}, a vector of \(\textit{N}\) example weights, initially \(1/\textit{N}\)
\textit{h}, a vector of \textit{K} hypotheses
\textit{z}, a vector of \textit{K} hypothesis weights

\textbf{for} \(k = 1 \text{ to } K \) \textbf{do}
\hspace{1em} \textit{h}[k] \leftarrow \textit{L}(\textit{examples}, \textit{w})
\hspace{1em} \textit{error} \leftarrow 0
\hspace{1em} \textbf{for} \(j = 1 \text{ to } \textit{N} \) \textbf{do}
\hspace{2em} \textbf{if} \textit{h}[k](x_j) \neq y_j \textbf{ then} \textit{error} \leftarrow \textit{error} + \textit{w}[j]
\hspace{1em} \textbf{for} \(j = 1 \text{ to } \textit{N} \) \textbf{do}
\hspace{2em} \textbf{if} \textit{h}[k](x_j) = y_j \textbf{ then} \textit{w}[j] \leftarrow \textit{w}[j] \cdot \textit{error}/(1 - \textit{error})
\hspace{1em} \textit{w} \leftarrow \textsc{Normalize}(\textit{w})
\hspace{1em} \textit{z}[k] \leftarrow \log (1 - \textit{error})/\textit{error}
\hspace{1em} \textbf{return} \textsc{Weighted-Majority}(\textit{h}, \textit{z})

\textbf{Figure 18.33} The \textsc{AdaBoost} variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function \textsc{Weighted-Majority} generates a hypothesis that returns the output value with the highest vote from the hypotheses in \textit{h}, with votes weighted by \textit{z}.
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 generalized as needed.
### function `VERSION-SPACE-LEARNING(examples)` returns a version space

**local variables:** $V$, the version space: the set of all hypotheses

1. $V \leftarrow$ the set of all hypotheses
2. **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

1. $V \leftarrow \{ h \in V : h \text{ is consistent with } e \}$

---

**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**

1. **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**

1. **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 \textsc{foil}(\textit{examples}, \textit{target}) \textbf{returns} a set of Horn clauses

\hspace{1em} \textbf{inputs:} \textit{examples}, set of examples

\hspace{1.5em} \textit{target}, a literal for the goal predicate

\hspace{1em} \textbf{local variables:} \textit{clauses}, set of clauses, initially empty

\hspace{1em} \textbf{while} \textit{examples} contains positive examples \textbf{do}

\hspace{2.5em} \textit{clause} $\leftarrow$ \textsc{new-clause}(\textit{examples}, \textit{target})

\hspace{2.5em} remove positive examples covered by \textit{clause} from \textit{examples}

\hspace{2.5em} add \textit{clause} to \textit{clauses}

\hspace{1em} \textbf{return} \textit{clauses}

---

function \textsc{new-clause}(\textit{examples}, \textit{target}) \textbf{returns} a Horn clause

\hspace{1em} \textbf{local variables:} \textit{clause}, a clause with \textit{target} as head and an empty body

\hspace{2.5em} \textit{l}, a literal to be added to the clause

\hspace{2.5em} \textit{extended-examples}, a set of examples with values for new variables

\hspace{1em} \textit{extended-examples} $\leftarrow$ \textit{examples}

\hspace{1em} \textbf{while} \textit{extended-examples} contains negative examples \textbf{do}

\hspace{2.5em} \textit{l} $\leftarrow$ \textsc{choose-literal}(\textsc{new-literals}(\textit{clause}), \textit{extended-examples})

\hspace{2.5em} append \textit{l} to the body of \textit{clause}

\hspace{2.5em} \textit{extended-examples} $\leftarrow$ set of examples created by applying \textsc{extend-example} to each example in \textit{extended-examples}

\hspace{1em} \textbf{return} \textit{clause}

---

function \textsc{extend-example}(\textit{example}, \textit{literal}) \textbf{returns} a set of examples

\hspace{1em} \textbf{if} \textit{example} satisfies \textit{literal} \textbf{then} return the set of examples created by extending \textit{example} with each possible constant value for each new variable in \textit{literal}

\hspace{1em} \textbf{else} return the empty set

---

\textbf{Figure 19.12}  Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. \textsc{new-literals} and \textsc{choose-literal} are explained in the text.
20 LEARNING PROBABILISTIC MODELS
function PASSIVE-ADP-AGENT(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 $R$, discount $\gamma$
    $U$, 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
        $P(t | s, a) \leftarrow N_{s'|sa}[t, s, a] / N_{sa}[s, a]$
        $U \leftarrow POLICY-EVALUATION(\pi, U, mdp)$
    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
Ns, 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’] ← r’
if s is not null then
  increment Ns[s]
  U[s] ← U[s] + α(Ns[s])(r + γ U[s’] − U[s])
if s’ TERMINAL? then s, a, r ← null else s, a, r ← s’, π[s’], r’
return a

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function α(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
Ns, 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] ← r’
if s is not null then
  increment Nsa[s, a]
  Q[s, a] ← Q[s, a] + α(Nsa[s, a])(r + γ max a’ Q[s’, a’] − Q[s, a])
  s, a, r ← s’, argmax a’ f(Q[s’, a’], Nsa[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.
### 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).
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 $P[X, \text{start}, \text{len}]$ is the probability of the most probable $X$ of length $\text{len}$ starting at position $\text{start}$. If there is no $X$ of that size at that location, the probability is 0.
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
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.
PHILOSOPHICAL FOUNDATIONS
27 AI: THE PRESENT AND FUTURE
MATHEMATICAL BACKGROUND
Figure 29.1  Example of a generator function and its invocation within a loop.

```plaintext
generator POWERS-OF-2() yields ints
  i ← 1
  while true do
    yield i
    i ← 2 × i
  end

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