function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
table, a table of actions, indexed by percept sequences, initially fully specified
  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action

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

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

Figure 2.8 The agent program for a simple reflex agent in the two-location vacuum environment. This program implements the agent function tabulated in Figure 2.3.
**Figure 2.10** A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

**Figure 2.12** A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
node ← NODE(State=problem.INITIAL)
frontier ← a priority queue ordered by f, with node as an element
reached ← a lookup table, with one entry with key problem.INITIAL and value node
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.State) then return node
    for each child in EXPAND(problem, node) do
        s ← child.State
        if s is not in reached or child.Path-Cost < reached[s].Path-Cost then
            reached[s] ← child
            add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
s ← node.State
for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.Path-Cost + problem.ACTION-COST(s, action, s')
yield NODE(State=s', Parent=node, Action=action, Path-Cost=cost)

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for yield.
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then return node
frontier ← a FIFO queue, with node as an element
reached ← {problem.INITIAL}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then return child
        if s is not in reached then
            add s to reached
            add child to frontier
    return failure

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

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

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

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

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

nodeF ← NODE(problemF.INITIAL)  // Node for a start state
nodeB ← NODE(problemB.INITIAL)  // Node for a goal state
frontierF ← a priority queue ordered by fF, with nodeF as an element
frontierB ← a priority queue ordered by fB, with nodeB as an element
reachedF ← a lookup table, with one key nodeF.STATE and value nodeF
reachedB ← a lookup table, with one key nodeB.STATE and value nodeB
solution ← failure

while not TERMINATED(solution, frontierF, frontierB) do
  if fF(TOP(frontierF)) < fB(TOP(frontierB)) then
    solution ← PROCEED(F, problemF, frontierF, reachedF, reachedB, solution)
  else solution ← PROCEED(B, problemB, frontierB, reachedB, reachedF, solution)

return solution

function PROCEED(dir, problem, frontier, reached, reached2, solution) returns a solution
  // Expand node on frontier; check against the other frontier in reached2.
  // The variable “dir” is the direction: either F for forward or B for backward.

node ← POP(frontier)
for each child in EXPAND(problem, node) do
  s ← child.STATE
  if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
    reached[s] ← child
    add child to frontier
  if s is in reached2 then
    solutions2 ← JOIN-NODES(dir, child, reached2[s])
    if PATH-COST(solution2) < PATH-COST(solution) then
      solution ← solution2

return solution

Figure 3.14 Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.
Figure 3.22 The algorithm for recursive best-first search.
**Chapter 4**

**Search in Complex Environments**

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

\[ \text{current} \leftarrow \text{problem. INITIAL} \]

**while** true **do**

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

**if** \( \text{VALUE(neighbor)} \leq \text{VALUE(current)} \) **then return** current

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

**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.

**Function** SIMULATED-ANNEALING(problem, schedule) **returns** a solution state

\[ \text{current} \leftarrow \text{problem. INITIAL} \]

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

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

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

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

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

**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. The **schedule** input determines the value of the “temperature” \( T \) as a function of time.
function GENETIC-ALGORITHM(population, fitness) returns an individual
repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
        parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child ← REPRODUCE(parent1, parent2)
        if (small random probability) then
            child ← MUTATE(child)
        add child to population2
    population ← population2
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to fitness

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

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

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

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

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        plan_i ← OR-SEARCH(problem, s_i, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic en-
vironments. A solution is a conditional plan that considers every nondeterministic outcome
and makes a plan for each one.
function ONLINE-DFS-AGENT(problem, s′) returns an action
  s, a, the previous state and action, initially null
  result, a table mapping (s, a) to s′, initially empty
  untried, a table mapping s to a list of untried actions
  unbacktracked, a table mapping s to a list of states never backtracked to

if problem.IS.GOAL(s′) then return stop
if s′ is a new state (not in untried) then untried[s′] ← problem.ACTIONS(s′)
if s is not null then
  result[s, a] ← s′
  add s to the front of unbacktracked[s′]
if untried[s′] is empty then
  if unbacktracked[s′] is empty then return stop
  a ← an action b such that result[s′, b] = P(OP(unbacktracked[s′]))s′ ← null
else a ← POP(untried[s′])
  s ← s′
return a

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

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

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

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

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.
function AC-3(csp) returns false if an inconsistency is found and true otherwise

queue ← a queue of arcs, initially all the arcs in csp

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

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i

revised ← false
for each x in D_i do
  if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j then
    delete x from D_i
    revised ← true
return revised

Figure 5.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name “AC-3” was used by the algorithm’s inventor (Mackworth, 1977) because it was the third version developed in the paper.
Figure 5.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. The functions \textsc{Select-Unassigned-Variable} and \textsc{Order-Domain-Values} implement the general-purpose heuristics discussed in Section 5.3.1. The \textsc{Inference} function can optionally impose arc-, path-, or \(k\)-consistency, as desired. If a value choice leads to failure (noticed either by \textsc{Inference} or by \textsc{Backtrack}), then value assignments (including those made by \textsc{Inference}) are retracted and a new value is tried.

Figure 5.9 The \textsc{Min-Conflicts} local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The \textsc{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 ← number of variables in X
assignment ← an empty assignment
root ← any variable in X
X ← TOPOLOGICALSORT(X, root)

for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure

for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure

return assignment

Figure 5.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.
function $\text{MINIMAX-SEARCH}(\text{game}, \text{state})$ returns an action
  player $\leftarrow \text{game}.\text{TO-MOVE}(\text{state})$
  value, move $\leftarrow \text{MAX-VALUE}(\text{game}, \text{state})$
  return move

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

function $\text{MIN-VALUE}(\text{game}, \text{state})$ returns a (utility, move) pair
  if $\text{game}.\text{IS-TERMIONAL}(\text{state})$ then return $\text{game}.\text{UTILITY}(\text{state}, \text{player}), \text{null}$
  $v, \text{move} \leftarrow +\infty$
  for each $a$ in $\text{game}.\text{ACTIONS}(\text{state})$ do
    $v2, a2 \leftarrow \text{MAX-VALUE}(\text{game}, \text{game}.\text{RESULT}(\text{state}, a))$
    if $v2 < v$ then
      $v, \text{move} \leftarrow v2, a$
  return $v, \text{move}$

Figure 6.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions $\text{MAX-VALUE}$ 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 and the move to get there.
function \textsc{Alpha-Beta-Search}(\textit{game}, \textit{state}) \textbf{returns} an action
\begin{itemize}
  \item player $\leftarrow \textit{game}.\text{To-Move}(\textit{state})$
  \item value, move $\leftarrow \textsc{Max-Value}(\textit{game}, \textit{state}, -\infty, +\infty)$
\end{itemize}
\textbf{return} move

function \textsc{Max-Value}(\textit{game}, \textit{state}, \alpha, \beta) \textbf{returns} a (utility, move) pair
\begin{itemize}
  \item if \textit{game}.\textsc{Is-Terminal}(\textit{state}) then return \textit{game}.\text{Utility}(\textit{state}, \textit{player}), \text{null}
  \item $v \leftarrow -\infty$
  \item for each $a$ in \textit{game}.\textsc{Actions}(\textit{state}) do
    \item $v2, a2 \leftarrow \textsc{Min-Value}(\textit{game}, \textit{game}.\textsc{Result}(\textit{state}, a), \alpha, \beta)$
    \item if $v2 > v$ then
      \item $v, move \leftarrow v2, a$
      \item $\alpha \leftarrow \textsc{Max}(\alpha, v)$
    \item if $v \geq \beta$ then return $v, move$
  \end{itemize}
\textbf{return} $v, move$

function \textsc{Min-Value}(\textit{game}, \textit{state}, \alpha, \beta) \textbf{returns} a (utility, move) pair
\begin{itemize}
  \item if \textit{game}.\textsc{Is-Terminal}(\textit{state}) then return \textit{game}.\text{Utility}(\textit{state}, \textit{player}), \text{null}
  \item $v \leftarrow +\infty$
  \item for each $a$ in \textit{game}.\textsc{Actions}(\textit{state}) do
    \item $v2, a2 \leftarrow \textsc{Max-Value}(\textit{game}, \textit{game}.\textsc{Result}(\textit{state}, a), \alpha, \beta)$
    \item if $v2 < v$ then
      \item $v, move \leftarrow v2, a$
      \item $\beta \leftarrow \textsc{Min}(\beta, v)$
    \item if $v \leq \alpha$ then return $v, move$
  \end{itemize}
\textbf{return} $v, move$

\textbf{Figure 6.7} The alpha–beta search algorithm. Notice that these functions are the same as the \textsc{Minimax-Search} functions in Figure 6.3, except that we maintain bounds in the variables $\alpha$ and $\beta$, and use them to cut off search when a value is outside the bounds.

function \textsc{Monte-Carlo-Tree-Search}(\textit{state}) \textbf{returns} an action
\begin{itemize}
  \item tree $\leftarrow \text{Node}(\textit{state})$
  \item while \textsc{Is-Time-Remaining}() do
    \item leaf $\leftarrow \text{Select}(\text{tree})$
    \item child $\leftarrow \text{Expand}(\text{leaf})$
    \item result $\leftarrow \text{Simulate}(\text{child})$
    \item \textsc{Back-Propagate}(result, child)
  \end{itemize}
\textbf{return} the move in \textsc{Actions}(\textit{state}) whose node has highest number of playouts

\textbf{Figure 6.11} The Monte Carlo tree search algorithm. A game tree, \textit{tree}, is initialized, and then we repeat a cycle of \textsc{Select} / \textsc{Expand} / \textsc{Simulate} / \textsc{Back-Propagate} until we run out of time, and return the move that led to the node with the highest number of playouts.
**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
    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 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns true if a sentence holds within a model. The variable model represents a partial model—an assignment to some of the symbols. The keyword and here is an infix function symbol in the pseudocode programming language, not an operator in propositional logic; it takes two arguments and returns true or false.

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 ← {}
while true 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\{ P = true \} union resolvents
        if new \subseteq clauses then return false
    clauses ← clauses union new

Figure 7.13 A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.
function PL-FC-ENTAILS?(KB, q) returns true or false
inputs: KB, the knowledge base, a set of propositional definite clauses
        q, the query, a proposition symbol

        count ← a table, where count[c] is initially the number of symbols in clause c’s premise

        inferred ← a table, where inferred[s] is initially false for all symbols

        queue ← a queue of symbols, initially symbols known to be true in KB

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

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

\[\text{clauses}\leftarrow\text{the set of clauses in the CNF representation of } s\]
\[\text{symbols}\leftarrow\text{a list of the proposition symbols in } s\]
\[\text{return }\text{DPLL}(\text{clauses, symbols, }\{\})\]

function DPLL\((\text{clauses, symbols, model})\) returns true or false

if every clause in \(\text{clauses}\) is true in \(\text{model}\) then return true
if some clause in \(\text{clauses}\) is false in \(\text{model}\) then return false
\(P, \text{value}\leftarrow\text{FIND-PURE-SYMBOL}(\text{symbols, clauses, model})\)
if \(P\) is non-null then return DPLL\((\text{clauses, symbols}\setminus\{P\}, \text{model}\cup\{P=\text{value}\})\)
\(P, \text{value}\leftarrow\text{FIND-UNIT-CLAUSE}(\text{clauses, model})\)
if \(P\) is non-null then return DPLL\((\text{clauses, symbols}\setminus\{P\}, \text{model}\cup\{P=\text{value}\})\)
\(P\leftarrow\text{FIRST}(\text{symbols}); \text{rest}\leftarrow\text{REST}(\text{symbols})\)
\(\text{return }\text{DPLL}(\text{clauses, rest, model}\cup\{P=\text{true}\})\) or \(\text{DPLL}(\text{clauses, rest, model}\cup\{P=\text{false}\})\)

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

function WALKSAT\((\text{clauses, }p, \text{max}_{\text{flips}})\) returns a satisfying model or failure
inputs: \(\text{clauses}\), a set of clauses in propositional logic
\(p\), the probability of choosing to do a “random walk” move, typically around 0.5
\(\text{max}_{\text{flips}}\), number of value flips allowed before giving up

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

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.
function HYBRID-WUMPUS-AGENT(\text{percept}) \textbf{returns} an action 

\begin{ itemize } 
  \item \textbf{inputs:} \text{percept}, a list, [stench, breeze, glitter, bump, scream] 
  \item \textbf{persistent:} \( KB \), a knowledge base, initially the atemporal “wumpus physics” 
  \hspace{1em} \( t \), a counter, initially 0, indicating time 
  \hspace{1em} \( \text{plan} \), an action sequence, initially empty 
\end{itemize} 

\text{TELL}(KB, \text{MAKE-PERCEPT-SENTENCE}(\text{percept}, t)) \text{TELL} the \( KB \) the temporal “physics” sentences for time \( t \) 

\( \text{safe} \leftarrow \{ [x,y] : \text{ASK}(KB, OK^t_{x,y}) = \text{true} \} \)

\textbf{if} \( \text{ASK}(KB, \text{Glitter}^t) = \text{true} \) \textbf{then} 

\begin{align*} 
\text{plan} & \leftarrow [\text{Grab}] + \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [\text{Climb}] 
\end{align*}

\textbf{if} \( \text{plan} \) is empty \textbf{then} 

\begin{align*} 
\text{unvisited} & \leftarrow \{ [x,y] : \text{ASK}(KB, L^t_{x,y}) = \text{false} \text{ for all } t' \leq t \} \\
\text{plan} & \leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{safe}, \text{safe}) 
\end{align*}

\textbf{if} \( \text{plan} \) is empty and \( \text{ASK}(KB, \text{HaveArrow}^t) = \text{true} \) \textbf{then} 

\begin{align*} 
\text{possible-wumpus} & \leftarrow \{ [x,y] : \text{ASK}(KB, \neg \text{W}^t_{x,y}) = \text{false} \} \\
\text{plan} & \leftarrow \text{PLAN-SHOT}(\text{current}, \text{possible-wumpus}, \text{safe}) 
\end{align*}

\textbf{if} \( \text{plan} \) is empty \textbf{then} \hspace{1em} \text{// no choice but to take a risk} 

\begin{align*} 
\text{not-unsafe} & \leftarrow \{ [x,y] : \text{ASK}(KB, \neg \text{OK}^t_{x,y}) = \text{false} \} \\
\text{plan} & \leftarrow \text{PLAN-ROUTE}(\text{current}, \text{unvisited} \cap \text{not-unsafe}, \text{safe}) 
\end{align*}

\textbf{if} \( \text{plan} \) is empty \textbf{then} 

\begin{align*} 
\text{plan} & \leftarrow \text{PLAN-ROUTE}(\text{current}, \{[1,1]\}, \text{safe}) + [\text{Climb}] \\
\text{action} & \leftarrow \text{POP}(\text{plan}) \\
\text{TELL}(KB, \text{MAKE-ACTION-SENTENCE}(\text{action}, t)) \\
\text{t} & \leftarrow t + 1 \\
\text{return} \text{ action} 
\end{align*}

\textbf{function} \text{PLAN-ROUTE}(\text{current}, \text{goals}, \text{allowed}) \textbf{returns} an action sequence 

\begin{ itemize } 
  \item \textbf{inputs:} \text{current}, the agent’s current position 
  \hspace{1em} \text{goals}, a set of squares; try to plan a route to one of them 
  \hspace{1em} \text{allowed}, a set of squares that can form part of the route 
\end{itemize} 

\begin{align*} 
\text{problem} & \leftarrow \text{ROUTE-PROBLEM}(\text{current}, \text{goals}, \text{allowed}) \\
\text{return} \text{ SEARCH}(\text{problem}) \hspace{1em} \text{// Any search algorithm from Chapter 3} 
\end{align*}

\textbf{Figure 7.20} A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to choose actions. Each time HYBRID-WUMPUS-AGENT is called, it adds the percept to the knowledge base, and then either relies on a previously-defined plan or creates a new plan, and pops off the first step of the plan as the action to do next.
function SATPLAN(init, transition, goal, T_max) returns solution or failure
inputs: init, transition, goal, constitute a description of the problem
        T_max, an upper limit for plan length

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

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned true in the model. If no model exists, then the process is repeated with the goal moved one step later.
CHAPTER 9

INFERENACE IN FIRST-ORDER LOGIC

function UNIFY(x, y, θ=empty) returns a substitution to make x and y identical, or failure
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(ARGS(x), ARGS(y), UNIFY(OP(x), OP(y), θ))
else if LIST?(x) and LIST?(y) then
    return UNIFY(REST(x), REST(y), UNIFY(FIRST(x), FIRST(y), θ))
else return failure

function UNIFY-VAR(var, x, θ) returns a substitution
if {var/val} ∈ θ for some val then return UNIFY(val, x, θ)
else if {x/val} ∈ θ for some val 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 arguments x and y can be any expression: a constant or variable, or a compound expression such as a complex sentence or term, or a list of expressions. The argument θ is a substitution, initially the empty substitution, but with {var/val} pairs added to it as we recurse through the inputs, comparing the expressions element by element. In a compound expression such as F(A, B), OP(x) field picks out the function symbol F and ARGS(x) field picks out the argument list (A, B).
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
while true do
    new ← {} // The set of new sentences inferred on each iteration
    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 failure then return φ
        if new = {} then return false
    add new to KB

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

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

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

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

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

Init(At(C₁, SFO) \land At(C₂, JFK) \land At(P₁, SFO) \land At(P₂, JFK)
\land Cargo(C₁) \land Cargo(C₂) \land Plane(P₁) \land Plane(P₂)
\land Airport(JFK) \land Airport(SFO))

Goal(At(C₁, JFK) \land At(C₂, SFO))

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))

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))

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}))

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

Init(\text{Tire}(\text{Flat}) \land \text{Tire}(\text{Spare}) \land \text{At}(\text{Flat}, \text{Axle}) \land \text{At}(\text{Spare}, \text{Trunk}))

Goal(\text{At}(\text{Spare}, \text{Axle}))

Action(\text{Remove}(\text{obj}, \text{loc}),
    \text{PRECOND}: \text{At}(\text{obj}, \text{loc})
    \text{EFFECT}: \neg \text{At}(\text{obj}, \text{loc}) \land \text{At}(\text{obj}, \text{Ground}))

Action(\text{PutOn}(t, \text{Axle}),
    \text{PRECOND}: \text{Tire}(t) \land \text{At}(t, \text{Ground}) \land \neg \text{At}(\text{Flat}, \text{Axle}) \land \neg \text{At}(\text{Spare}, \text{Axle})
    \text{EFFECT}: \neg \text{At}(t, \text{Ground}) \land \text{At}(t, \text{Axle}))

Action(\text{LeaveOvernight},
    \text{PRECOND}: \neg \text{At}(\text{Spare}, \text{Ground}) \land \neg \text{At}(\text{Spare}, \text{Axle}) \land \neg \text{At}(\text{Spare}, \text{Trunk})
    \land \neg \text{At}(\text{Flat}, \text{Ground}) \land \neg \text{At}(\text{Flat}, \text{Axle}) \land \neg \text{At}(\text{Flat}, \text{Trunk}))

Figure 11.2 The simple spare tire problem.
**Figure 11.4** A planning problem in the blocks world: building a three-block tower. One solution is the sequence \([\text{MoveToTable}(C, A), \text{Move}(B, \text{Table}, C), \text{Move}(A, \text{Table}, B)]\).

---

**Refinement** (Go(Home, SFO)),

**STEPS:** [Drive(Home, SFO|LongTermParking), Shuttle(SFO|LongTermParking, 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])]}

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

frontier ← a FIFO queue with [Act] as the only element

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

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

frontier ← a FIFO queue with initialPlan as the only element

while true do
  if IS-EMPTY?(frontier) then return fail
  plan ← POP(frontier)  // chooses the shallowest node in frontier
  if REACH^+(problem.INITIAL, plan) intersects problem.GOAL then
    if plan is primitive then return plan  // REACH^+ is exact for primitive plans
    guaranteed ← REACH^−(problem.INITIAL, plan) \ problem.GOAL
    if guaranteed ≠ {} and MAKING-PROGRESS(plan, initialPlan) then
      finalState ← any element of guaranteed
      return DECOMPOSE(hierarchy, problem.INITIAL, plan, finalState)
  hla ← some HLA in plan
  prefix, suffix ← the action subsequences before and after hla in plan
  outcome ← RESULT(problem.INITIAL, prefix)
  for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
    add APPEND(prefix, sequence, suffix) to frontier

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

solution ← an empty plan

while plan is not empty do
  action ← REMOVE-LAST(plan)
  s_i ← a state in REACH^−(s_0, plan) such that s_f ∈ REACH^−(s_i, action)
  problem ← a problem with INITIAL = s_i and GOAL = s_f
  solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
  s_f ← s_i

return solution

Figure 11.11 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don’t. The predicate MAKING-PROGRESS checks to make sure that we aren’t stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with [Act] as the initialPlan.
Jobs(\{(\text{AddEngine1} \prec \text{AddWheels1} \prec \text{Inspect1})\},
\{(\text{AddEngine2} \prec \text{AddWheels2} \prec \text{Inspect2})\})

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

\begin{align*}
\text{Action(AddEngine1, DURATION:30,} \\
& \hspace{1cm} \text{USE:EngineHoists(1))} \\
\text{Action(AddEngine2, DURATION:60,} \\
& \hspace{1cm} \text{USE:EngineHoists(1))} \\
\text{Action(AddWheels1, DURATION:30,} \\
& \hspace{1cm} \text{CONSUME:LugNuts(20), USE:WheelStations(1))} \\
\text{Action(AddWheels2, DURATION:15,} \\
& \hspace{1cm} \text{CONSUME:LugNuts(20), USE:WheelStations(1))} \\
\text{Action(Inspect_i, DURATION:10,} \\
& \hspace{1cm} \text{USE:Inspectors(1))}
\end{align*}

\textbf{Figure 11.13} A job-shop scheduling problem for assembling two cars, with resource constraints. The notation } A \prec B \text{ means that action } A \text{ must precede action } B.\)
function DT-AGENT(percept) returns an action

persistent: belief state, probabilistic beliefs about the current state of the world
action, the agent’s action

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

Figure 12.1 A decision-theoretic agent that selects rational actions.
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 vars
Q(X) ← a distribution over X, initially empty
for each value x of X do
  Q(x) ← **ENUMERATE-ALL**(vars, e, x)
  where e is e extended with X = x
return **NORMALIZE**(Q(X))

function **ENUMERATE-ALL**(vars, e) returns a real number
  if **EMPTY**(vars) then return 1.0
  V ← **FIRST**(vars)
  if V is an evidence variable with value v in e
    then return **P**(v | parents(V)) × **ENUMERATE-ALL**(REST(vars), e)
  else return ∑v **P**(v | parents(V)) × **ENUMERATE-ALL**(REST(vars), e)
    where e is e extended with V = v

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

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 with variables vars
factors ← []
for each V in ORDER(vars) do
  factors ← [**MAKE-FACTOR**(V, e)] + factors
  if V is a hidden variable then factors ← **SUM-OUT**(V, factors)
return **NORMALIZE**(**POINTWISE-PRODUCT**(factors))

Figure 13.13 The variable elimination algorithm for exact inference in Bayes nets.
function P \textsc{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
for each 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 13.16 A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable’s parents.

function \textsc{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: C, a vector of counts for each value of X, initially zero

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

Figure 13.17 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.
function LIKELIHOOD-WEIGHTING($X, e, bn, N$) returns an estimate of $P(X \mid e)$
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($bn, e$)
    $W[j] \leftarrow W[j] + w$ where $x_j$ 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, with values fixed from $e$
for $i = 1$ to $n$ do
    if $X_i$ is an evidence variable with value $x_{ij}$ in $e$
        then $w \leftarrow w \times P(X_i = x_{ij} \mid \text{parents}(X_i))$
    else $x[i] \leftarrow$ a random sample from $P(X_i \mid \text{parents}(X_i))$
return $x$, $w$

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

function GIBBS-ASK($X, e, bn, N$) returns an estimate of $P(X \mid e)$
local variables: $C$, a vector of counts for each value of $X$, initially zero
$Z$, the nonevidence variables in $bn$
$x$, the current state of the network, initialized from $e$
initialize $x$ with random values for the variables in $Z$
for $k = 1$ to $N$ do
    choose any variable $Z_i$ from $Z$ according to any distribution $\rho(i)$
    set the value of $Z_i$ in $x$ by sampling from $P(Z_i \mid \text{mb}(Z_i))$
    $C[j] \leftarrow C[j] + 1$ where $x_j$ is the value of $X$ in $x$
return NORMALIZE($C$)

Figure 13.20 The Gibbs sampling algorithm for approximate inference in Bayes nets; this version chooses variables at random, but cycling through the variables but also works.
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 down to 1 do
    sv[i] ← NORMALIZE(fv[i] × b)
    b ← BACKWARD(b, ev[i])
return sv

Figure 14.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (14.5) and (14.9), respectively.
function \textsc{Fixed-Lag-Smoothing}(e_t, hmm, d) returns a distribution over $X_{t-d}$
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
\hspace{1em} $f$, the forward message $P(X_t | e_{1:t})$, initially $hmm.prior$
\hspace{1em} $B$, the $d$-step backward transformation matrix, initially the identity matrix
\hspace{1em} $e_{t-d:t}$, double-ended list of evidence from $t-d$ to $t$, initially empty
local variables: $O_{t-d}, O_t$, diagonal matrices containing the sensor model information

add $e_t$ to the end of $e_{t-d:t}$
$O_t \leftarrow$ diagonal matrix containing $P(e_t | X_t)$
if $t > d$ then
\hspace{1em} $f \leftarrow \textsc{Forward}(f, e_{t-d})$
\hspace{1em} remove $e_{t-d-1}$ from the beginning of $e_{t-d:t}$
\hspace{1em} $O_{t-d} \leftarrow$ diagonal matrix containing $P(e_{t-d} | X_{t-d})$
\hspace{1em} $B \leftarrow O_{t-d}^{-1} T^{-1} \text{BTO}_t$
else $B \leftarrow \text{BTO}_t$
$t \leftarrow t + 1$
if $t > d + 1$ then return $\textsc{Normalize}(f \times B1)$ else return null

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

function \textsc{Particle-Filtering}(e, N, dbn) returns a set of samples for the next time step
inputs: $e$, the new incoming evidence
\hspace{1em} $N$, the number of samples to be maintained
\hspace{1em} $dbn$, a DBN defined by $P(X_0)$, $P(X_1 | X_0)$, and $P(E_1 | X_1)$
persistent: $S$, a vector of samples of size $N$, initially generated from $P(X_0)$
local variables: $W$, a vector of weights of size $N$

for $i = 1$ to $N$
\hspace{1em} $S[i] \leftarrow$ sample from $P(X_1 | X_0 = S[i])$ \hfill \text{\# step 1}
\hspace{1em} $W[i] \leftarrow P(e | X_1 = S[i])$ \hfill \text{\# step 2}
$S \leftarrow \textsc{Weighted-Sample-With-Replacement}(N, S, W)$ \hfill \text{\# step 3}
return $S$

Figure 14.17 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in $\textsc{Prior-Sample}$. The $\textsc{Weighted-Sample-With-Replacement}$ operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.
function INFORMATION-GATHERING-AGENT(percept) returns an action
  persistent: $D$, a decision network

  integrate percept into $D$
  $j \leftarrow$ the value that maximizes $\text{VPI}(E_j) / \text{C}(E_j)$
  if $\text{VPI}(E_j) > \text{C}(E_j)$
    then return Request($E_j$)
  else return the best action from $D$

Figure 15.9 Design of a simple, myopic information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.
function \textsc{Value-Iteration}(mdp, ε) returns a utility function

inputs: mdp, an MDP with states $S$, actions $A(s)$, transition model $P(s' | s, a)$, rewards $R(s, a, s')$, discount $\gamma$

$\epsilon$, the maximum error allowed in the utility of any state

local variables: $U$, $U'$, vectors of utilities for states in $S$, initially zero

$\delta$, the maximum relative change in the utility of any state

repeat

$U \leftarrow U'$; $\delta \leftarrow 0$

for each state $s$ in $S$ do

$U'[s] \leftarrow \max_{a \in A(s)} Q\text{-VALUE}(mdp, s, a, U)$

if $|U'[s] - U[s]| > \delta$ then

$\delta \leftarrow |U'[s] - U[s]|$

until $\delta \leq \epsilon(1 - \gamma)/\gamma$

return $U$

\textbf{Figure 16.6} The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (16.2).
function POLICY-ITERATION\((mdp)\) returns a policy
inputs: \(mdp\), an MDP with states \(S\), actions \(A(s)\), transition model \(P(s' \mid s, a)\)
local variables: \(U\), a vector of utilities for states in \(S\), initially zero
\(\pi\), a policy vector indexed by state, initially random

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

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

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

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

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

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

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

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

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

Figure 17.1 The doubles tennis problem. Two actors, $A$ and $B$, are playing together and can be in one of four locations: $\text{LeftBaseline}$, $\text{RightBaseline}$, $\text{LeftNet}$, and $\text{RightNet}$. The ball can be returned only if a player is in the right place. The $\text{NoOp}$ action is a dummy, which has no effect. Note that each action must include the actor as an argument.
type Researcher, Paper, Citation
random String Name(Researcher)
random String Title(Paper)
random Paper PubCited(Citation)
random String Text(Citation)
random Boolean Professor(Researcher)
origin Researcher Author(Paper)

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

Figure 18.5 An OUPM for citation information extraction. For simplicity the model assumes one author per paper and omits details of the grammar and error models.
Chapter 18  Probabilistic Programming

#SeismicEvents ~ Poisson(T \ast \lambda_e)
Time(e) ~ UniformReal(0, T)
EarthQuake(e) ~ Boolean(0.999)
Location(e) ~ if Earthquake(e) then SpatialPrior() else UniformEarth() 
Depth(e) ~ if Earthquake(e) then Uniform(0, 700) else Exactly(0) 
Magnitude(e) ~ Exponential(log(10))
Detected(e, p, s) ~ Logistic(weights(s, p), Magnitude(e), Depth(e), Dist(e, s))

#SeismicEvents ~ Poisson(T \ast \lambda_e)

#Detecteds(site = s) ~ Poisson(T \ast \lambda_f(s))

#Detecteds(event = e, phase = p, station = s) = if Detected(e, p, s) then 1 else 0

OnsetTime(a, s) if (event(a) = null) then ~ UniformReal(0, T)
else = Time(event(a)) + GeoTT(Dist(event(a), s), Depth(event(a)), phase(a))
+ Laplace(\mu_s(s), \sigma_s(s))

Amplitude(a, s) if (event(a) = null) then ~ NoiseAmpModel(s)
else = AmpModel(Magnitude(event(a)), Dist(event(a), s), Depth(event(a)), phase(a))

Azimuth(a, s) if (event(a) = null) then ~ UniformReal(0, 360)
else = GeoAzimuth(Location(event(a)), Depth(event(a)), phase(a), Site(s))
+ Laplace(0, \sigma_a(s))

Slowness(a, s) if (event(a) = null) then ~ UniformReal(0, 20)
else = GeoSlowness(Location(event(a)), Depth(event(a)), phase(a), Site(s))
+ Laplace(0, \sigma_s(s))

ObservedPhase(a, s) ~ CategoricalPhaseModel(phase(a))

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

#Aircraft(EntryTime = t) ~ Poisson(\lambda_a)
Exits(a, t) ~ if InFlight(a, t) then Boolean(\alpha_e)
InFlight(a, t) = (t = EntryTime(a)) \lor (InFlight(a, t - 1) \land \neg Exits(a, t - 1))
X(a, t) ~ if t = EntryTime(a) then InitX()
else if InFlight(a, t) then \mathcal{N}(FX(a, t - 1), \Sigma_e)

#Blip(Source = a, Time = t) ~ if InFlight(a, t) then Bernoulli(DetectionProb(X(a, t)))
#Blip(Time = t) ~ Poisson(\lambda_f)
Z(b) ~ if Source(b) = null then UniformZ(R) else \mathcal{N}(HX(Source(b), Time(b)), \Sigma_z)

Figure 18.9 An OUPM for radar tracking of multiple targets with false alarms, detection failure, and entry and exit of aircraft. The rate at which new aircraft enter the scene is \lambda_a, while the probability per time step that an aircraft exits the scene is \alpha_e. False alarm blips (i.e., ones not produced by an aircraft) appear uniformly in space at a rate of \lambda_f per time step. The probability that an aircraft is detected (i.e., produces a blip) depends on its current position.
**Figure 18.11** Generative program for an open-universe probability model for optical character recognition. The generative program produces degraded images containing sequences of letters by generating each sequence, rendering it into a 2D image, and incorporating additive noise at each pixel.

**Figure 18.15** Generative program for an improved optical character recognition model that generates letters according to a letter bigram model whose pairwise letter frequencies are estimated from a list of English words.
function LEARN-DECISION-TREE(examples, attributes, parent_examples) returns a tree

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

Figure 19.5 The decision tree learning algorithm. The function IMPORTANCE is described in Section 19.3.3. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.
function MODEL-SELECTION(Learner, examples, k) returns a (hypothesis, error rate) pair

err ← an array, indexed by size, storing validation-set error rates
training_set, test_set ← a partition of examples into two sets

for size = 1 to ∞ do
    err[size] ← CROSS-VALIDATION(Learner, size, training_set, k)
    if err is starting to increase significantly then
        best_size ← the value of size with minimum err[size]
        h ← Learner(best_size, training_set)
        return h, ERROR-RATE(h, test_set)

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

N ← the number of examples
errs ← 0

for i = 1 to k do
    validation_set ← examples[(i − 1) × N/k, i × N/k]
    training_set ← examples − validation_set
    h ← Learner(size, training_set)
    errs ← errs + ERROR-RATE(h, validation_set)

return errs / k  // average error rate on validation sets, across k-fold cross-validation

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

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, of examples
such that the members of examples, are all positive or all negative

if there is no such t then return failure

if the examples in examples, 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 19.11 An algorithm for learning decision lists.
Figure 19.25 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in \( h \), with votes weighted by \( z \). For regression problems, or for binary classification with two classes -1 and 1, this is \( \sum_k h[k]z[k] \).
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''
else
    return fail

Figure 20.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or generalized 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 20.3 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 20.8 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 ← 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 ← examples
while extended_examples contains negative examples do
    l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended_examples)
    append l to the body of clause
    extended_examples ← set of examples created by applying EXTEND-EXAMPLE to each example in extended_examples

return clause

function EXTEND-EXAMPLE(example, literal) returns a set of examples

if example satisfies literal
    then return the set of examples created by extending example with each possible constant value for each new variable in literal
else return the empty set

Figure 20.12 Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERALS and CHOOSE-LITERAL are explained in the text.
function Passive-ADP-Learner(percept) returns an action

inputs: percept, a percept indicating the current state $s'$ and reward signal $r$

persistent: $\pi$, a fixed policy

$mdp$, an MDP with model $P$, rewards $R$, actions $A$, discount $\gamma$

$U$, a table of utilities for states, initially empty

$N_{s'|s,a}$, a table of outcome count vectors indexed by state and action, initially zero

$s, a$, the previous state and action, initially null

if $s'$ is new then $U[s'] \leftarrow 0$

if $s$ is not null then

increment $N_{s'|s,a}[s,a][s']$

$R[s, a, s'] \leftarrow r$

add $a$ to $A[s]$

$P(\cdot | s,a) \leftarrow \text{NORMALIZE}(N_{s'|s,a}[s,a])$

$U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$

$s, a \leftarrow s', \pi[s']$

return $a$

Figure 23.2 A passive reinforcement learning agent based on adaptive dynamic programming. The agent chooses a value for $\gamma$ and then incrementally computes the $P$ and $R$ values of the MDP. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 567.
**function** PASSIVE-TD-LEARNER(*percept*) **returns** an action
  
  **inputs:** *percept*, a percept indicating the current state $s'$ and reward signal $r$
  
  **persistent:** $\pi$, a fixed policy
  
  $s$, the previous state, initially null
  
  $U$, a table of utilities for states, initially empty
  
  $N_s$, a table of frequencies for states, initially zero

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

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

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

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

**Figure 23.8** An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function $f$ as the exploratory ADP agent, but avoids having to learn the transition model.
function CYK-PARSE(words, grammar) returns a table of parse trees
inputs: words, a list of words
       grammar, a structure with LEXICALRULES and GRAMMARRULES

T ← a table  // T[X, i, k] is most probable X tree spanning words_{i:k}
P ← a table, initially all 0  // P[X, i, k] is probability of tree T[X, i, k]

// Insert lexical categories for each word.
for i = 1 to LEN(words) do
    for each (X, p) in grammar.LEXICALRULES(words_i) do
        P[X, i, i] ← p
        T[X, i, i] ← TREE(X, words_i)

    // Construct X_{i,k} from Y_{i,j} + Z_{j+1,k}, shortest spans first.
    for each (i, j, k) in SUBSPANS(LEN(words)) do
        for each (X, Y, Z, p) in grammar.GRAMMARRULES do
            PYZ ← P[Y, i, j] × P[Z, j + 1, k] × p
            if PYZ > P[X, i, k] do
                P[X, i, k] ← PYZ
                T[X, i, k] ← TREE(X, T[Y, i, j], T[Z, j + 1, k])

return T

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

Figure 24.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable parse tree for the sequence and its subsequences. The table P[X, i, k] gives the probability of the most probable tree of category X spanning words_{i:k}. The output table T[X, i, k] contains the most probable tree of category X spanning positions i to k inclusive. The function SUBSPANS returns all tuples (i,j,k) covering a span of words_{i:k}, with i ≤ j < k, listing the tuples by increasing length of the i : k span, so that when we go to combine two shorter spans into a longer one, the shorter spans are already in the table. LEXICALRULES(word) returns a collection of (X, p) pairs, one for each rule of the form X → word [p], and GRAMMARRULES gives (X, Y, Z, p) tuples, one for each grammar rule of the form X → Y Z [p].
Figure 24.8 Annotated tree for the sentence “Her eyes were glazed as if she didn’t hear or even see him.” from the Penn Treebank. Note a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase “hear or even see him” as consisting of two constituent 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]. Similarly, the [NP *-2] refers to the [NP-2 Her eyes].
It is a truth universally acknowledged that the earth is not the center of the universe. There are those who assert there is. I do not accept them, but others I consider to be of the same opinion. The truth is, however, that if there are other than the center, and if there are any other living things in the universe and if they are not human, then we do not yet have our answers. We have to go on. This page gives a simplified, simplified answer to the problem. We don’t have all the answers. The truth is, however, that the truth is out there.

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

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

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

Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Tell me a story. Please fill out the following details. Thank you...

Figure 25.14 Example completion texts generated by the GPT-2 language model, given the prompts in bold. Most of the texts are quite fluent English, at least locally. The final example demonstrates that sometimes the model just breaks down.
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X′|X, v, ω), P(z|z∗), map)
returns a set of samples, S, for the next time step
inputs: a, robot velocities v and ω
        z, a vector of M range scan data points
        P(X′|X, v, ω), motion model
        P(z|z∗), a range sensor noise model
        map, a 2D map of the environment
persistent: S, a vector of N samples
local variables: W, a vector of N weights
                S′, a temporary vector of N samples

if S is empty then
    for i = 1 to N do  // initialization phase
        S[i] ← sample from P(X0)
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], map)
    W[i] ← W[i] · P(zj|z∗)
    S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S′, W)
return S

Figure 26.6 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.
CHAPTER 28

PHILOSOPHY, ETHICS, AND SAFETY OF AI
CHAPTER 29

THE FUTURE OF AI
APPENDIX A

MATHEMATICAL BACKGROUND
NOTES ON LANGUAGES AND ALGORITHMS