1 INTRODUCTION
2 INTELLIGENT AGENTS

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

\textbf{Figure 2.8}

function REFLEX-VACUUM-AGENT( [location,status] ) \textbf{returns} an action
\hspace{1em} if status = \textit{Dirty} then return \textit{Suck}
\hspace{1em} else if location = A then return \textit{Right}
\hspace{1em} else if location = B then return \textit{Left}

\textbf{Figure 2.10}

function SIMPLE-REFLEX-AGENT( percept ) \textbf{returns} an action
\textbf{static}: rules, a set of condition–action rules
\hspace{1em} state \leftarrow \textsc{INTERPRET-INPUT}(percept)
\hspace{1em} rule \leftarrow \textsc{RULE-MATCH}(state,rules)
\hspace{1em} action \leftarrow \textsc{RULE-ACTION}[rule]
\hspace{1em} return action

\textbf{Figure 2.13}
function REFLEX-AGENT-WITH-STATE(\textit{percept}) returns an action

static: \textit{state}, a description of the current world state
rules, a set of condition–action rules
\textit{action}, the most recent action, initially none

\textit{state} \leftarrow \textsc{update-state}(\textit{state}, \textit{action}, \textit{percept})
\textit{rule} \leftarrow \textsc{rule-match}(\textit{state}, \textit{rules})
\textit{action} \leftarrow \textsc{rule-action}[\textit{rule}]

\textbf{return} \textit{action}

\textbf{Figure 2.16}
function _SIMPLE-PROBLEM-SOLVING-AGENT_(percept) returns an action
inputs: percept, a percept
static: 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 do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
action ← FIRST(seq)
seq ← REST(seq)
return action

Figure 3.2

function _TREE-SEARCH_(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Figure 3.9
function Tree-Search(problem, fringe) returns a solution, or failure

fringe ← INSERT(MAKE-NODE(INITIAL-STATE(problem), fringe)
loop do
  if EMPTY?(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST[problem] applied to STATE[node] succeeds
     then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function Expand(node, problem) returns a set of nodes

successors ← the empty set
for each (action, result) in SUCCESSOR-FN(problem)(STATE[node]) do
  s ← a new NODE
  STATE[s] ← result
  PARENT-NODE[s] ← node
  ACTION[s] ← action
  PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
  DEPTH[s] ← DEPTH[node] + 1
  add s to successors
return successors

Figure 3.12

function Depth-Limited-Search(problem, limit) returns a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE(problem), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
cutoff_occurred? ← false
if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
else if DEPTH[node] = limit then return cutoff
else for each successor in EXPAND(node, problem) do
  result ← RECURSIVE-DLS(successor, problem, limit)
  if result = cutoff then cutoff_occurred? ← true
  else if result ≠ failure then return result
if cutoff_occurred? then return cutoff else return failure

Figure 3.17
function \textsc{Iterative-Deepening-Search}(\textit{problem}) returns a solution, or failure

\begin{verbatim}
inputs: problem, a problem

for \textit{depth} ← 0 to \infty do
    \textit{result} ← \textsc{Depth-Limited-Search}(\textit{problem}, \textit{depth})
    if \textit{result} \neq \text{cutoff} then return \textit{result}
\end{verbatim}

Figure 3.19

function \textsc{Graph-Search}(\textit{problem}, \textit{fringe}) returns a solution, or failure

\begin{verbatim}
closed ← an empty set
\textit{fringe} ← \textsc{Insert}(\textsc{Make-Node}(\text{Initial-State}[\textit{problem}]), \textit{fringe})
loop do
    if \textsc{Empty?}(\textit{fringe}) then return failure
    \textit{node} ← \textsc{Remove-First}(\textit{fringe})
    if \textsc{Goal-Test}(\textit{problem})(\text{State}[\textit{node}]) then return \textsc{Solution}(\textit{node})
    if \text{State}[\textit{node}] is not in \textit{closed} then
        add \text{State}[\textit{node}] to \textit{closed}
        \textit{fringe} ← \textsc{Insert-All}(\textsc{Expand}(\textit{node}, \textit{problem}), \textit{fringe})
\end{verbatim}

Figure 3.25
function Recursively-Best-First-Search(problem) returns a solution, or failure
RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
   if GOAL-TEST(problem)(state) then return node
   successors ← EXPAND(node, problem)
   if successors is empty then return failure, ∞
   for each s in successors do
      f[s] ← max(g(s) + h(s), f[node])
   repeat
      best ← the lowest f-value node in successors
      if f[best] > f_limit then return failure, f[best]
      alternative ← the second-lowest f-value among successors
      result, f[best] ← RBFS(problem, best, mini(f_limit, alternative))
   if result ≠ failure then return result

Figure 4.6

function Hill-Climbing(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
               neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      neighbor ← a highest-valued successor of current
      if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
      current ← neighbor

Figure 4.13
Figure 4.17

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

**inputs:** problem, a problem

schedule, a mapping from time to “temperature”

**local variables:** current, a node

next, a node

T, a “temperature” controlling the probability of downward steps

current ← MAKE-NODE(INITIAL-STATE[ problem])

for t ← 1 to ∞ do

T ← schedule[t]

if T = 0 then return current

next ← a randomly selected successor of current

ΔE ← VALUE[next] − VALUE[current]

if ΔE > 0 then current ← next

else current ← next only with probability $e^{ΔE/T}$

Figure 4.21

**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

loop for i from 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))
function ONLINE-DFS-AGENT(s') returns an action
inputs: s', a percept that identifies the current state
static: result, a table, indexed by action and state, initially empty
        unexplored, a table that lists, for each visited state, the actions not yet tried
        unbacktracked, a table that lists, for each visited 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 then unexplored[s'] ← ACTIONS(s')
if s is not null then do
    result[a, s] ← s'
    add s to the front of unbacktracked[s']
if unexplored[s'] is empty then
    if unbacktracked[s'] is empty then return stop
    else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
else a ← POP(unexplored[s'])
s ← s'
return a

Figure 4.25

function LRTA*-AGENT(s') returns an action
inputs: s', a percept that identifies the current state
static: result, a table, indexed by action and state, 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')
unless s is null
    result[a, s] ← s'
    H[s] ← \min_{b \in ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)
    a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], 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 4.29
function BACKTRACKING-Search(csp) returns a solution, or failure
return RECURSIVE-BACKTRACKING(\{\}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
if assignment is complete then return assignment
var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
  if value is consistent with assignment according to CONSTRAINTS[csp] then
    add \{var = value\} to assignment
    result ← RECURSIVE-BACKTRACKING(assignment, csp)
    if result ≠ failure then return result
    remove \{var = value\} from assignment
  end if
end for
return failure

Figure 5.4
function AC-3(csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    \((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\text{queue})\)
    if \text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j) then
        for each \(X_k\) in NEIGHBORS[\(X_i\)] do
            add \((X_k, X_i)\) to \text{queue}

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns true if we remove a value
removed \leftarrow \text{false}
for each \(x\) in \text{DOMAIN}[X_i] do
    if no value \(y\) in \text{DOMAIN}[X_j] allows \((x, y)\) to satisfy the constraint between \(X_i\) and \(X_j\) then delete \(x\) from \text{DOMAIN}[X_i]; removed \leftarrow \text{true}
return removed

Figure 5.9

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 \leftarrow \text{an initial complete assignment for csp}
for \(i = 1\) to \text{max_steps} do
    if \text{current} is a solution for csp then return \text{current}
    \(\text{var} \leftarrow \text{a randomly chosen, conflicted variable from VARIABLES[csp]}\)
    \(\text{value} \leftarrow \text{the value } v \text{ for } \text{var} \text{ that minimizes } \text{CONFLICTS(var, value, current, csp)}\)
    set \text{var} = \text{value in current}
return failure

Figure 5.11
6 ADVERSARIAL SEARCH

function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game

v ← MAX-VALUE(state)
return the action in SUCCESSORS(state) with value v

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

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

Figure 6.4
function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game

v ← MAX-VALUE(state, −∞, +∞)
return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
inputs: state, current state in game
α, the value of the best alternative for MAX along the path to state
β, the value of the best alternative for MIN along the path to state

if TERMINAL-TEST(state) then return UTILITY(state)
v ← −∞
for a, s in SUCCESSORS(state) do
  v ← MAX(v, MIN-VALUE(s, α, β))
  if v ≥ β then return v
  α ← MAX(α, v)
return v

function MIN-VALUE(state, α, β) returns a utility value
inputs: state, current state in game
α, the value of the best alternative for MAX along the path to state
β, the value of the best alternative for MIN along the path to state

if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for a, s in SUCCESSORS(state) do
  v ← MIN(v, MAX-VALUE(s, α, β))
  if v ≤ α then return v
  β ← MIN(β, v)
return v

Figure 6.9
7

LOGICAL AGENTS

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

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

Figure 7.2

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
else do
    P ← FIRST(symbols); rest ← REST(symbols)
    return TT-CHECK-ALL(KB, α, rest, EXTEND(P, true, model) and
                        TT-CHECK-ALL(KB, α, rest, EXTEND(P, false, model))

Figure 7.12
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

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

Figure 7.15

function PL-FC-ENTAILS?(KB, q) returns true or false
inputs: KB, the knowledge base, a set of propositional Horn clauses
        q, the query, a proposition symbol
local variables: count, a table, indexed by clause, initially the number of premises
                inferred, a table, indexed by symbol, each entry initially false
                agenda, a list of symbols, initially the symbols known to be true in KB

while agenda is not empty do
    p ← POP(agenda)
    unless inferred[p] ← true
    for each Horn clause c in whose premise p appears do
        decrement count[c]
        if count[c] = 0 then do
            if HEAD[c] = q then return true
            PUSH(HEAD[c], agenda)
        end do
end loop
return false

Figure 7.18
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, EXTEND(P, value, model))
P, value ← FIND-UNIT-CLAUSE(clauses, model)
if P is non-null then return DPLL(clauses, symbols – P, EXTEND(P, value, model))
P ← FIRST(symbols); rest ← REST(symbols)
return DPLL(clauses, rest, EXTEND(P, true, model)) or DPLL(clauses, rest, EXTEND(P, false, model))

Figure 7.21

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 7.23
function PL-WUMPUS-AGENT( percept ) returns an action
inputs: percept, a list, [stench,breeze,glitter]

static: KB, a knowledge base, initially containing the “physics” of the wumpus world
x, y, orientation, the agent’s position (initially 1,1) and orientation (initially right)
visited, an array indicating which squares have been visited, initially false
action, the agent’s most recent action, initially null
plan, an action sequence, initially empty

update x,y,orientation, visited based on action
if stench then TELL(KB, S_{x,y}) else TELL(KB, ¬S_{x,y})
if breeze then TELL(KB, B_{x,y}) else TELL(KB, ¬B_{x,y})
if glitter then action ← grab
else if plan is nonempty then action ← POP(plan)
else if for some fringe square [i,j], ASK(KB, (¬P_{i,j} ∧ ¬W_{i,j})) is true or
    for some fringe square [i,j], ASK(KB, (P_{i,j} ∨ W_{i,j})) is false then do
    plan ← A*-GRAPH-SEARCH(Route-Problem([x,y], orientation,[i,j],visited))
    action ← POP(plan)
else action ← a randomly chosen move
return action

Figure 7.26
8 FIRST-ORDER LOGIC
function \textsc{Unify}(x, y, \theta) \ returns \ a \ substitution \ to \ make \ x \ and \ y \ identical
inputs: \ x, \ a \ variable, \ constant, \ list, \ or \ compound
\hspace{1em} y, \ a \ variable, \ constant, \ list, \ or \ compound
\hspace{1em} \theta, \ the \ substitution \ built \ up \ so \ far \ (optional, \ defaults \ to \ empty)

if \ \theta = \text{failure} \ then \ \text{return} \ \text{failure}
else \ if \ x = y \ then \ \text{return} \ \theta
else \ if \ \text{VARIABLE?(x)} \ then \ \text{return} \ \textsc{Unify-Var}(x, y, \theta)
else \ if \ \text{VARIABLE?(y)} \ then \ \text{return} \ \textsc{Unify-Var}(y, x, \theta)
else \ if \ \text{COMPOUND?(x)} \ and \ \text{COMPOUND?(y)} \ then
\hspace{1em} \text{return} \ \textsc{Unify}($\text{ARGS}[x]$, $\text{ARGS}[y]$, $\text{Unify}($$\text{OP}[x], \text{OP}[y], \theta)$)
else \ if \ \text{LIST?(x)} \ and \ \text{LIST?(y)} \ then
\hspace{1em} \text{return} \ \textsc{Unify}($\text{REST}[x]$, $\text{REST}[y]$, $\text{Unify}($$\text{FIRST}[x], \text{FIRST}[y], \theta)$)
else \ \text{return} \ \text{failure}

\begin{verbatim}
function \textsc{Unify-Var}(\text{var}, x, \theta) \ returns \ a \ substitution
inputs: \ \text{var}, \ a \ variable
\hspace{1em} x, \ any \ expression
\hspace{1em} \theta, \ the \ substitution \ built \ up \ so \ far

if \ \{\text{var}/\text{val}\} \ \subseteq \ \theta \ then \ \text{return} \ \text{Unify}($\text{val}, x, \theta$)
else \ if \ \{x/\text{val}\} \ \subseteq \ \theta \ then \ \text{return} \ \text{Unify}($\text{var}, \text{val}, \theta$)
else \ if \ \text{Occur-Check?(\text{var}, x)} \ then \ \text{return} \ \text{failure}
else \ \text{return} \ \text{add} \ \{\text{var}/x\} \ \text{to} \ \theta
\end{verbatim}

Figure 9.2
function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses
         α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration
repeat until new is empty
   new ← { }
   for each sentence r in KB do
      (p₁ ∧ . . . ∧ pₙ ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that SUBST(θ, p₁ ∧ . . . ∧ pₙ) = SUBST(θ, p₁' ∧ . . . ∧ pₙ')
         for some p₁', . . . , pₙ' in KB
            q' ← SUBST(θ, q)
            if q' is not a renaming of some sentence already in KB or new then do
               add q' to new
               φ ← UNIFY(q', α)
               if φ is not fail then return φ
            add new to KB
   return false

Figure 9.5

function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
inputs: KB, a knowledge base
         goals, a list of conjuncts forming a query (θ already applied)
         θ, the current substitution, initially the empty substitution { }
local variables: answers, a set of substitutions, initially empty
if goals is empty then return {θ}
q' ← SUBST(θ, FIRST(goals))
for each sentence r in KB where STANDARDIZE-APART(r) = (p₁ ∧ . . . ∧ pₙ ⇒ q)
   and θ' ← UNIFY(q, q') succeeds
   new_goals ← [p₁, . . . , pₙ][REST(goals)]
   answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(θ', θ)) ∪ answers
return answers

Figure 9.9

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

Figure 9.12
procedure OTTER(sos, usable)
    inputs: sos, a set of support—clauses defining the problem (a global variable)
    usable, background knowledge potentially relevant to the problem
    repeat
        clause ← the lightest member of sos
        move clause from sos to usable
        PROCESS(INFER(clause, usable), sos)
    until sos = [] or a refutation has been found

function INFER(clause, usable) returns clauses
    resolve clause with each member of usable
    return the resulting clauses after applying FILTER

procedure PROCESS(clauses, sos)
    for each clause in clauses do
        clause ← SIMPLIFY(clause)
        merge identical literals
        discard clause if it is a tautology
        sos ← [clause — sos]
        if clause has no literals then a refutation has been found
        if clause has one literal then look for unit refutation

Figure 9.19
11 PLANNING

\[ \text{Init}(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK) \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2) \land Airport(JFK) \land Airport(SFO))} \]

\[ \text{Goal}(At(C_1, JFK) \land At(C_2, SFO)) \]

\[ \text{Action(Load}(c, p, a),} \]
\[ \text{Precond: } At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \]
\[ \text{Effect: } \neg At(c, a) \land In(c, p)) \]

\[ \text{Action(Unload}(c, p, a),} \]
\[ \text{Precond: } In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a) \]
\[ \text{Effect: } At(c, a) \land \neg In(c, p)) \]

\[ \text{Action(Fly}(p, \text{from, to}),} \]
\[ \text{Precond: } At(p, \text{from}) \land Plane(p) \land Airport(\text{from}) \land Airport(\text{to}) \]
\[ \text{Effect: } \neg At(p, \text{from}) \land At(p, \text{to})) \]

Figure 11.3
\begin{align*}
\text{Init}( & \text{At(Flat, Axle)} \land \text{At(Spare, Trunk)}) \\
\text{Goal}( & \text{At(Spare, Axle)}) \\
\text{Action( } & \text{Remove(Spare, Trunk),} \\
\text{Precond: } & \text{At(Spare, Trunk)} \\
\text{Effect: } & \neg \text{At(Spare, Trunk)} \land \text{At(Spare, Ground)}) \\
\text{Action( } & \text{Remove(Flat, Axle),} \\
\text{Precond: } & \text{At(Flat, Axle)} \\
\text{Effect: } & \neg \text{At(Flat, Axle)} \land \text{At(Flat, Ground)}) \\
\text{Action( } & \text{PutOn(Spare, Axle),} \\
\text{Precond: } & \text{At(Spare, Ground)} \land \neg \text{At(Flat, Axle)} \\
\text{Effect: } & \neg \text{At(Spare, Ground)} \land \text{At(Spare, Axle)}) \\
\text{Action( } & \text{LeaveOvernight,} \\
\text{Precond: } & \neg \text{At(Spare, Ground)} \land \neg \text{At(Flat, Axle)} \land \neg \text{At(Spare, Trunk)} \\
& \land \neg \text{At(Flat, Ground)} \land \neg \text{At(Flat, Axle)})
\end{align*}

Figure 11.5

\begin{align*}
\text{Init( } & \text{On(A, Table)} \land \text{On(B, Table)} \land \text{On(C, Table)} \\
& \land \text{Block(A)} \land \text{Block(B)} \land \text{Block(C)} \\
& \land \text{Clear(A)} \land \text{Clear(B)} \land \text{Clear(C)}) \\
\text{Goal( } & \text{On(A, B)} \land \text{On(B, C)}) \\
\text{Action( } & \text{Move(b, x, y),} \\
\text{Precond: } & \text{On(b, x)} \land \text{Clear(b)} \land \text{Clear(y)} \land \text{Block(b)} \land \\
& (b \neq x) \land (b \neq y) \land (x \neq y), \\
\text{Effect: } & \text{On(b, y)} \land \text{Clear(x)} \land \neg \text{On(b, x)} \land \neg \text{Clear(y)}) \\
\text{Action( } & \text{MoveToTable(b, x),} \\
\text{Precond: } & \text{On(b, x)} \land \text{Clear(b)} \land \text{Block(b)} \land (b \neq x), \\
\text{Effect: } & \text{On(b, Table)} \land \text{Clear(x)} \land \neg \text{On(b, x)})
\end{align*}

Figure 11.7
\[\text{Init}(\text{At(Flat, Axle)} \land \text{At(Spare, Trunk)})\]
\[\text{Goal(At(Spare, Axle))}\]
\[\text{Action(Remove(Spare, Trunk)},\]
\[\quad \text{Precond: } \text{At(Spare, Trunk)}\]
\[\quad \text{Effect: } \neg \text{At(Spare, Trunk)} \land \text{At(Spare, Ground)}\]
\[\text{Action(Remove(Flat, Axle)}),\]
\[\quad \text{Precond: } \text{At(Flat, Axle)}\]
\[\quad \text{Effect: } \neg \text{At(Flat, Axle)} \land \text{At(Flat, Ground)}\]
\[\text{Action(PutOn(Spare, Axle)},\]
\[\quad \text{Precond: } \text{At(Spare, Ground)} \land \neg \text{At(Flat, Axle)}\]
\[\quad \text{Effect: } \neg \text{At(Spare, Ground)} \land \text{At(Spare, Axle)}\]
\[\text{Action(LeaveOvernight)},\]
\[\quad \text{Precond:}\]
\[\quad \text{Effect: } \neg \text{At(Spare, Ground)} \land \neg \text{At(Spare, Axle)}\]
\[\land \neg \text{At(Flat, Ground)} \land \neg \text{At(Flat, Axle)}\]

Figure 11.11

\[\text{Init(Have(Cake))}\]
\[\text{Goal(Have(Cake)} \land \text{Eaten(Cake)}\]
\[\text{Action(Eat(Cake)},\]
\[\quad \text{Precond: } \text{Have(Cake)}\]
\[\quad \text{Effect: } \neg \text{Have(Cake)} \land \text{Eaten(Cake)}\]
\[\text{Action(Bake(Cake)},\]
\[\quad \text{Precond: } \neg \text{Have(Cake)}\]
\[\quad \text{Effect: } \text{Have(Cake)}\]

Figure 11.16

\begin{verbatim}
function GRAPHPLAN(problem) returns solution or failure
    graph ← INITIAL-PLANING-GRAH( problem)
    goals ← GOALS[ problem]
    loop do
        if goals all non-mute in last level of graph then do
            solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
            if solution ≠ failure then return solution
        else if NO-SOLUTION-POSSIBLE(graph) then return failure
        graph ← EXPAND-GRAH(graph, problem)
end loop
\end{verbatim}

Figure 11.19
function SATPLAN(problem, \( T_{\text{max}} \)) returns solution or failure

inputs: problem, a planning problem
        \( T_{\text{max}} \), an upper limit for plan length

for \( T = 0 \) to \( T_{\text{max}} \) do
    cnf, mapping \( \leftarrow \) TRANSLATE-TO-SAT(problem, \( T \))
    assignment \( \leftarrow \) SAT-SOLVER(cnf)
    if assignment is not null then
        return EXTRACT-SOLUTION(assignment, mapping)
return failure

Figure 11.22
PLANNING AND ACTING IN THE REAL WORLD

\[
\begin{align*}
\text{Init} & : \ Chassis(C_1) \land \ Chassis(C_2) \\
& \land \ Engine(E_1, C_1, 30) \land \ Engine(E_2, C_2, 60) \\
& \land \ Wheels(W_1, C_1, 30) \land \ Wheels(W_2, C_2, 15) \\
\text{Goal} & : \ Done(C_1) \land \ Done(C_2) \\
\text{Action} & : \ AddEngine(e, c, m), \\
& \text{PRECOND: } \ Engine(e, c, d) \land \ Chassis(c) \land \neg \ EngineIn(c), \\
& \text{EFFECT: } \ EngineIn(c) \land \ Duration(d) \\
\text{Action} & : \ AddWheels(w, c), \text{PRECOND: } \ Wheels(w, c, d) \land \ Chassis(c), \\
& \text{EFFECT: } \ WheelsOn(c) \land \ Duration(d) \\
\text{Action} & : \ Inspect(c), \text{PRECOND: } \ EngineIn(c) \land \ WheelsOn(c) \land \ Chassis(c), \\
& \text{EFFECT: } \ Done(c) \land \ Duration(10))
\end{align*}
\]

Figure 12.2
\[
\text{Init(Chassis}(C_1) \land \text{Chassis}(C_2) \\
\land \text{Engine}(E_1, C_1, 30) \land \text{Engine}(E_2, C_2, 60) \\
\land \text{Wheels}(W_1, C_1, 30) \land \text{Wheels}(W_2, C_2, 15) \\
\land \text{EngineHoists}(1) \land \text{WheelStations}(1) \land \text{Inspectors}(2))
\]
\[
\text{Goal(Done}(C_1) \land \text{Done}(C_2))
\]
\[
\text{Action(AddEngine}(e, c, m), \\
\text{PRECOND: Engine}(e, c, d) \land \text{Chassis}(c) \land \neg \text{EngineIn}(c), \\
\text{EFFECT: EngineIn}(c) \land \text{Duration}(d), \\
\text{RESOURCE: EngineHoists}(1))
\]
\[
\text{Action(AddWheels}(w, c), \\
\text{PRECOND: Wheels}(w, c, d) \land \text{Chassis}(c), \\
\text{EFFECT: WheelsOn}(c) \land \text{Duration}(d), \\
\text{RESOURCE: WheelStations}(1))
\]
\[
\text{Action(Inspect}(c), \\
\text{PRECOND: EngineIn}(c) \land \text{WheelsOn}(c), \\
\text{EFFECT: Done}(c) \land \text{Duration}(10), \\
\text{RESOURCE: Inspectors}(1))
\]

**Figure 12.5**

\[
\text{Action(BuyLand, PRECOND: Money, EFFECT: Land \land \neg Money)}
\]
\[
\text{Action(GetLoan, PRECOND: GoodCredit, EFFECT: Money \land Mortgage)}
\]
\[
\text{Action(BuildHouse, PRECOND: Land, EFFECT: House)}
\]
\[
\text{Action(GetPermit, PRECOND: Land, EFFECT: Permit)}
\]
\[
\text{Action(HireBuilder, EFFECT: Contract)}
\]
\[
\text{Action(Construction, PRECOND: Permit \land Contract, \\
EFFECT: HouseBuilt \land \neg Permit)}
\]
\[
\text{Action(PayBuilder, PRECOND: Money \land HouseBuilt, \\
EFFECT: \neg Money \land House \land \neg Contract)}
\]

\[
\text{Decompose(BuildHouse,} \\
\text{Plan(STEPS: } S_1 \text{: GetPermit, } S_2 \text{: HireBuilder,} \\
S_3 \text{: Construction, } S_4 \text{: PayBuilder})
\]
\[
\text{ORDERINGS: } \{\text{Start} \prec S_1 \prec S_3 \prec S_4 \prec \text{Finish}, \text{ Start} \prec S_2 \prec S_3\},
\]
\[
\text{LINKS: } \{\text{Start} \stackrel{\text{Land}}{\rightarrow} S_1, \text{Start} \stackrel{\text{Money}}{\rightarrow} S_4, \\
S_1 \rightarrow S_3, S_2 \rightarrow S_3, S_3 \rightarrow S_4, \text{HouseBuilt} \rightarrow S_4, \\
S_4 \rightarrow \text{Finish, } S_4 \rightarrow \text{Money} \rightarrow \text{Finish})\}
\]

**Figure 12.9**
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
    OR-SEARCH(INITIAL-STATE([problem], problem, []))

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

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

Figure 12.14

function REPLANNING-AGENT(percept) returns an action
    static: KB, a knowledge base (includes action descriptions)
    plan, a plan, initially []
    whole_plan, a plan, initially []
    goal, a goal

    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    current ← STATE-DESCRIPTION(KB, t)
    if plan = [] then
        whole_plan ← plan ← PLANNER(current, goal, KB)
    if PRECONDITIONS(FIRST(plan)) not currently true in KB then
        candidates ← SORT(whole_plan, ordered by distance to current)
        find state s in candidates such that
            failure ≠ repair ← PLANNER(current, s, KB)
        continuation ← the tail of whole_plan starting at s
        whole_plan ← plan ← APPEND(repair, continuation)
    return POP(plan)

Figure 12.18
function CONTINUOUS-POP-AGENT(percept) returns an action
static: plan, a plan, initially with just Start, Finish

action ← NoOp (the default)
EFFECTS[Start] = UPDATE(EFFECTS[Start], percept)
REMOVE-FLAW(plan) // possibly updating action
return action
13 Uncertainty

**Figure 13.2**

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

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

**Figure 13.6**

```plaintext
function ENUMERATE-JOINT-ASK(X, e, P) returns a distribution over X
inputs: X, the query variable
e, observed values for variables E
P, a joint distribution on variables {X} \cup \, E \cup \, Y \quad / * Y = hidden variables */

Q(X) \leftarrow a \text{ distribution over } X, \text{ initially empty}
for each value \( x_i \) of \( X \) do
Q(\( x_i \)) \leftarrow ENUMERATE-JOINT(\( x_i \), e, Y, [\], P)
return NORMALIZE(Q(X))
```

```plaintext
function ENUMERATE-JOINT(x, e, \textit{vars}, values, P) returns a real number
if EMPTY?(\textit{vars}) then return \( P(x, e, \textit{values}) \)
Y \leftarrow \text{FIRST}(\textit{vars})
return \sum_{\textit{vars}} ENUMERATE-JOINT(x, e, \text{REST}(\textit{vars}), [y|\textit{values}], P)
```

31
14 PROBABILISTIC REASONING

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

$Q(X) \leftarrow$ a distribution over $X$, initially empty
for each value $x_i$ of $X$ do
  extend $e$ with value $x_i$ for $X$
  $Q(x_i) \leftarrow$ ENUMERATE-ALL(VARS[$bn$], $e$)
return NORMALIZE($Q(X)$)

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

Figure 14.10

function ELIMINATION-ASK($X$, $e$, $bn$) returns a distribution over $X$
inputs: $X$, the query variable
         $e$, evidence specified as an event
         $bn$, a Bayes network specifying joint distribution $P(X_1, \ldots, X_n)$

$factors \leftarrow []; vars \leftarrow$ REVERSE(VARS[$bn$])
for each $var$ in $vars$ do
  $factors \leftarrow [MAKE-FACTO($var$, $e$)]factors$
  if $var$ is a hidden variable then $factors \leftarrow$ SUM-OUT($var$, $factors$)
return NORMALIZE(POINTWISE-PRODUCT($factors$))

Figure 14.12
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
- for $i = 1$ to *n* do
  - $x_i \leftarrow$ a random sample from $P(X_i | \text{parents}(X_i))$
- return $x$

**Figure 14.15**

function **REJECTION-SAMPLING**(*X, e, bn, N*) **returns** an estimate of $P(X|e)$

inputs: *X*, the query variable
- *e*, evidence specified as an event
- *bn*, a Bayesian network
- *N*, the total number of samples to be generated

local variables: *N*, a vector of counts over *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[*X*])

**Figure 14.17**

function **LIKELIHOOD-WEIGHTING**(*X, e, bn, N*) **returns** an estimate of $P(X|e)$

inputs: *X*, the query variable
- *e*, evidence specified as an event
- *bn*, a Bayesian network
- *N*, the total number of samples to be generated

local variables: *W*, a vector of weighted counts over *X*, initially zero

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

**Figure 14.19**

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

- $x \leftarrow$ an event with *n* elements; $w \leftarrow 1$
- for $i = 1$ to *n* do
  - if $X_i$ has a 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 $x, w$
function MCMC-ASK(\(X, e, b_n, N\)) returns an estimate of \(P(X|e)\)

**local variables:**
- \(N[X]\), a vector of counts over \(X\), initially zero
- \(Z\), the nonevidence variables in \(b_n\)
- \(x\), 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
  \(N[x] \leftarrow N[x] + 1\) where \(x\) is the value of \(X\) in \(x\)
  for each \(Z_i\) in \(Z\) do
    sample the value of \(Z_i\) in \(x\) from \(P(Z_i|mb(Z_i))\) given the values of \(MB(Z_i)\) in \(x\)
  return \(\text{NORMALIZE}(N[X])\)
function \textsc{Forward-Backward}(\textit{ev}, \textit{prior}) \textbf{returns} a vector of probability distributions

\textbf{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\textit{)}

\textbf{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}

\begin{align*}
\text{fv}[0] &\leftarrow \textit{prior} \\
\text{for } i = 1 \text{ to } \text{t} \text{ do} \\
& \quad \text{fv}[i] \leftarrow \textsc{Forward(fv}[i - 1\textit{], ev}[i\textit{])} \\
\text{for } i = \text{t} \text{ downto } 1 \text{ do} \\
& \quad \text{sv}[i] \leftarrow \textsc{Normalize(fv}[i\textit{] \times \textit{b})} \\
& \quad \text{b} \leftarrow \textsc{Backward(b, ev}[i\textit{])} \\
\text{return } \textit{sv}
\end{align*}

\begin{figure}[h]
\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{15} & \textbf{PROBABILISTIC} \\
& \textbf{REASONING OVER TIME} \\
\hline
\end{tabular}
\end{center}
\caption{Figure 15.5}
\end{figure}
function **FIXED-LAG-SMOOTHING**($e_t, \text{hmm}, d$) returns a distribution over $X_{t-d}$

**inputs:**

- $e_t$, the current evidence for time step $t$
- $\text{hmm}$, a hidden Markov model with $S \times S$ transition matrix $T$
- $d$, the length of the lag for smoothing

**static:**

- $t$, the current time, initially 1
- $f$, a probability distribution, the forward message $P(X_t|e_{1:t})$, initially $\text{PRIOR}[\text{hmm}]$
- $B$, the $d$-step backward transformation matrix, initially the identity matrix $e_{t-d:t}$, double-ended list of evidence from $t-d$ to $t$, initially empty

**local variables:**

- $O_{t-d}$, $O_t$, 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

$\phantom{\text{else }} f \leftarrow \text{FORWARD}(f, e_t)$
remove $e_{t-d:t-1}$ from the beginning of $e_{t-d:t}$

$O_{t-d} \leftarrow$ diagonal matrix containing $P(e_{t-d}|X_{t-d})$

$B \leftarrow O_{t-d}^{-1}T^{-1}BTO_t$

else $B \leftarrow BTO_t$

$t \leftarrow t + 1$

if $t > d$ then return $\text{NORMALIZE}(f \times B1)$ else return null

---

**Figure 15.8**

---

**function** **PARTICLE-FILTERING**($e, N, \text{dbn}$) returns a set of samples for the next time step

**inputs:**

- $e$, the new incoming evidence
- $N$, the number of samples to be maintained
- $\text{dbn}$, a DBN with prior $P(X_0)$, transition model $P(X_t|X_0)$, and sensor model $P(E_t|X_t)$

**static:**

- $S$, a vector of samples of size $N$, initially generated from $P(X_0)$
- $W$, a vector of weights of size $N$

for $i = 1$ to $N$

$S[i] \leftarrow$ sample from $P(X_1|X_0 = S[i])$

$W[i] \leftarrow P(e|X_t = S[i])$

$S \leftarrow \text{WEIGHTED-SAMPLE-WITH-REPLACEMENT}(N, S, W)$

return $S$

---

**Figure 15.18**
16 MAKING SIMPLE DECISIONS

function INFORMATION-GATHERING-AGENT( percept ) returns an action
    static: $D$, a decision network
    integrate percept into $D$
    $j \leftarrow$ the value that maximizes $VPI(E_j) - Cost(E_j)$
    if $VPI(E_j) > Cost(E_j)$
        then return REQUEST($E_j$)
    else return the best action from $D$

Figure 16.9
MAKING COMPLEX DECISIONS

function VALUE-ITERATION($mdp, \epsilon$) returns a utility function
inputs: $mdp$, an MDP with states $S$, transition model $T$, reward function $R$, discount $\gamma$
\hspace{1em} $\epsilon$, the maximum error allowed in the utility of any state

local variables: $U$, $U'$, vectors of utilities for states in $S$, initially zero
\hspace{1em} $\delta$, the maximum change in the utility of any state in an iteration

repeat
\hspace{1em} $U \leftarrow U'$; $\delta \leftarrow 0$
\hspace{2em} for each state $s$ in $S$ do
\hspace{3em} $U'[s] \leftarrow R[s] + \gamma \max_{a} \sum_{s'} T(s, a, s') U[s']$
\hspace{3em} if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
\hspace{2em} until $\delta < \epsilon(1 - \gamma)/\gamma$
return $U$

Figure 17.5
function \textsc{Policy-Iteration}(mdp) returns a policy

inputs: \textit{mdp}, an MDP with states \( \mathcal{S} \), transition model \( T \)

local variables: \( U, U' \), vectors of utilities for states in \( \mathcal{S} \), initially zero
\( \pi \), a policy vector indexed by state, initially random

repeat
  \( U \leftarrow \textsc{Policy-Evaluation}(\pi, U, mdp) \)
  unchanged? \( \leftarrow \) true
  for each state \( s \) in \( \mathcal{S} \) do
    if \( \max_a \sum_j T(s, a, s') U[s'] > \sum_j T(s, \pi[s], s') U[s'] \) then
      \( \pi[s] \leftarrow \arg\max_a \sum_{s'} T(s, a, s') U[s'] \)
    unchanged? \( \leftarrow \) false
  until unchanged?
return \( P \)

Figure 17.9
function DECISION-TREE-LEARNING(examples, attrs, default) returns a decision tree
inputs: examples, set of examples
        attrs, set of attributes
        default, default value for the goal predicate

if examples is empty then return default
else if all examples have the same classification then return the classification
else if attrs is empty then return MAJORITY-VALUE(examples)
else
    best ← CHOOSE-ATTRIBUTE(attrs, examples)
    tree ← a new decision tree with root test best
    m ← MAJORITY-VALUE(examples)
    for each value $v_i$ of best do
        $examples_i$ ← {elements of examples with $best = v_i$}
        $subtree$ ← DECISION-TREE-LEARNING($examples_i$, attrs $-$ best, m)
        add a branch to $tree$ with label $v_i$ and subtree $subtree$
    return $tree$

Figure 18.6
function \textsc{AdaBoost}(\textit{examples}, L, M) returns a weighted-majority hypothesis

\textbf{inputs:} \textit{examples}, set of \(N\) labelled examples \((\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\)

\(L\), a learning algorithm

\(M\), the number of hypotheses in the ensemble

\textbf{local variables:} \(\vec{w}\), a vector of \(N\) example weights, initially \(1/N\)

\(\vec{h}\), a vector of \(M\) hypotheses

\(\vec{z}\), a vector of \(M\) hypothesis weights

\begin{verbatim}
for \(m = 1\) to \(M\) do
    \(h[m] \leftarrow L(\textit{examples}, \vec{w})\)
    error \leftarrow 0
    for \(j = 1\) to \(N\) do
        if \(h[m](\vec{x}_j) \neq y_j\) then error \leftarrow error + \vec{w}[j]
    for \(j = 1\) to \(N\) do
        if \(h[m](\vec{x}_j) = y_j\) then \(\vec{w}[j] \leftarrow \vec{w}[j] : error/(1 - error)\)
    \(\vec{w} \leftarrow \text{NORMALIZE}(\vec{w})\)
    \(z[m] \leftarrow \log(1 - error)/error\)
    return \textsc{Weighted-Majority}(\(\vec{h}, \vec{z}\))
\end{verbatim}

\textbf{Figure 18.12}

function \textsc{Decision-List-Learning}(\textit{examples}) returns a decision list, or failure

\begin{verbatim}
if \textit{examples} is empty then return the trivial decision list \(\textit{No}\)
\(t \leftarrow \) a test that matches a nonempty subset \textit{examples} \(_s\) of \textit{examples}
    such that the members of \textit{examples} \(_s\) are all positive or all negative
if there is no such \(t\) then return \textbf{failure}
if the examples in \textit{examples} \(_s\) are positive then \(o \leftarrow \textbf{Yes}\) else \(o \leftarrow \textbf{No}\)
return a decision list with initial test \(t\) and outcome \(o\) and remaining tests given by
\textsc{Decision-List-Learning}(\textit{examples} \(-\textit{examples} \(_s\))
\end{verbatim}

\textbf{Figure 18.17}
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

\[ H \leftarrow \text{any hypothesis consistent with the first example in examples} \]

for each remaining example in examples do
  \[ \text{if } e \text{ is false positive for } H \text{ then} \]
  \[ H \leftarrow \text{choose a specialization of } H \text{ consistent with examples} \]
  \[ \text{else if } e \text{ is false negative for } H \text{ then} \]
  \[ H \leftarrow \text{choose a generalization of } H \text{ consistent with examples} \]
  \[ \text{if no consistent specialization/generalization can be found then fail} \]
return \( H \)

Figure 19.3

function VERSION-SPACE-LEARNING(examples) returns a version space

local variables: \( V \), the version space: the set of all hypotheses

\[ V \leftarrow \text{the set of all hypotheses} \]

for each example \( e \) in examples do
  \[ \text{if } V \text{ is not empty then } V \leftarrow \text{VERSION-SPACE-UPDATE}(V, e) \]
return \( V \)

function VERSION-SPACE-UPDATE(\( V, e \)) returns an updated version space

\[ V \leftarrow \{ h \in V : h \text{ is consistent with } e \} \]

Figure 19.5
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 $i \leftarrow 0, \ldots, n$ do
    for each subset $A_i$ of $A$ of size $i$ do
      if `Consistent-DET?(A_i, E)` then return $A_i$

function `Consistent-DET?(A, E)` returns a truth-value

  inputs: $A$, a set of attributes
  $E$, a set of examples

  local variables: $H$, a hash table

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

  return `true`

Figure 19.11
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 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
if example satisfies literal
    then return the set of examples created by extending example with
each possible constant value for each new variable in literal
else return the empty set

Figure 19.16
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
inputs: examples, a set of examples, each with input $x = x_1, \ldots, x_n$ and output $y$ 
network, a perceptron with weights $W_j$, $j = 0 \ldots n$, and activation function $g$

repeat
    for each $e$ in examples do
        \[ in = \sum_{j=0}^{n} W_j x_j[e] \]
        \[ err = y[e] - g(in) \]
        \[ W_j \leftarrow W_j + \alpha \times err \times g'(in) \times x_j[e] \]
    until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(network)

Figure 20.22
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 \( W_{j,i} \), activation function \( g \)

repeat

for each \( e \) in examples do

for each node \( j \) in the input layer do \( a_j \leftarrow x_j[e] \)

for \( \ell = 2 \) to \( M \) do

\[ \text{in}_i \leftarrow \sum_j W_{j,i} a_j \]

\[ a_i \leftarrow g(\text{in}_i) \]

for each node \( i \) in the output layer do

\[ \Delta_i \leftarrow g'(\text{in}_i) \times (y_i[e] - a_i) \]

for \( \ell = M - 1 \) to \( 1 \) do

for each node \( j \) in layer \( \ell \) do

\[ \Delta_j \leftarrow g'(\text{in}_j) \sum_i W_{j,i} \Delta_i \]

for each node \( i \) in layer \( \ell + 1 \) do

\[ W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \]

until some stopping criterion is satisfied

return NEURAL-NET-HYPOTHESIS(network)
21 REINFORCEMENT LEARNING

function PASSIVE-ADP-AGENT(\textit{percept}) returns an action

\textbf{inputs:} \textit{percept}, a percept indicating the current state \( s' \) and reward signal \( r' \)

\textbf{static:} \( \pi \), a fixed policy
- \( mdp \), an MDP with model \( T \), rewards \( R \), discount \( \gamma \)
- \( U \), a table of utilities, initially empty
- \( N_{sa} \), a table of frequencies for state-action pairs, initially zero
- \( N_{sat'} \), a table of frequencies for state-action-state triples, initially zero
- \( s, a \), the previous state and action, initially null

\textbf{if} \( s' \) is new \textbf{then do} \( U[s'] \leftarrow r' \); \( R[s'] \leftarrow r' \)
\textbf{if} \( s \) is not null \textbf{then do}
- increment \( N_{sa}[s, a] \) and \( N_{sat'}[s, a, s'] \)
- \textbf{for each} \( t \) such that \( N_{sat'}[s, a, t] \) is nonzero \textbf{do}
  \( T[s, a, t] \leftarrow N_{sat'}[s, a, t] / N_{sa}[s, a] \)
- \( U \leftarrow \text{VALUE-DETERMINATION}(\pi, U, mdp) \)
\textbf{if} \( \text{TERMINAL?}(s') \) \textbf{then do} \( s, a \leftarrow \text{null} \) \textbf{else do} \( s, a \leftarrow s', \pi[s'] \)
\textbf{return} \( a \)

Figure 21.3
function \textsc{Passive-TD-Agent}(\textit{percept}) returns an action
\begin{itemize}
\item \textbf{inputs:} \textit{percept}, a percept indicating the current state \textit{s'} and reward signal \textit{r'}
\item \textbf{static:} \textit{\pi}, a fixed policy
\begin{itemize}
\item \textit{U}, a table of utilities, initially empty
\item \textit{N_s}, a table of frequencies for states, initially zero
\item \textit{s, a, r}, the previous state, action, and reward, initially null
\end{itemize}
\end{itemize}
\begin{itemize}
\item if \textit{s'} is new then \textit{U}[\textit{s'}] \leftarrow \textit{r'}
\item if \textit{s} is not null then do
\begin{itemize}
\item increment \textit{N_s}[\textit{s}]
\item \textit{U}[\textit{s}] \leftarrow \textit{U}[\textit{s}] + \alpha (\textit{N_s}[\textit{s}])(\textit{r} + \gamma \textit{U}[\textit{s'}] - \textit{U}[\textit{s}])
\end{itemize}
\item if \textit{Terminal?}[\textit{s'}] then \textit{s, a, r} \leftarrow \text{null} else \textit{s, a, r} \leftarrow \textit{\pi}[\textit{s'}], \textit{r'}
\item return \textit{a}
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure21-6.png}
\caption{Figure 21.6}
\end{figure}

function \textsc{Q-Learning-Agent}(\textit{percept}) returns an action
\begin{itemize}
\item \textbf{inputs:} \textit{percept}, a percept indicating the current state \textit{s'} and reward signal \textit{r'}
\item \textbf{static:} \textit{Q}, a table of action values index by state and action
\item \textit{N_{sa}}, a table of frequencies for state-action pairs
\item \textit{s, a, r}, the previous state, action, and reward, initially null
\end{itemize}
\begin{itemize}
\item if \textit{s} is not null then do
\begin{itemize}
\item increment \textit{N_{sa}}[\textit{s}, \textit{a}]
\item \textit{Q}[\textit{a, s}] \leftarrow \textit{Q}[\textit{a, s}] + \alpha (\textit{N_{sa}}[\textit{s}, \textit{a}])(\textit{r} + \gamma \max_{\textit{a'}} \textit{Q}[\textit{a', s'}] - \textit{Q}[\textit{a, s}])
\end{itemize}
\item if \textit{Terminal?}[\textit{s'}] then \textit{s, a, r} \leftarrow \text{null}
\item else \textit{s, a, r} \leftarrow \textit{s'}, \arg\max_{\textit{a'}} f(\textit{Q}[\textit{a', s'}], \textit{N_{sa}}[\textit{a', s'}]), \textit{r'}
\item return \textit{a}
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure21-11.png}
\caption{Figure 21.11}
\end{figure}
**22 COMMUNICATION**

```plaintext
function Naive-Communicating-Agent(percept) returns action
  static: KB, a knowledge base
    state, the current state of the environment
    action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept)
  words ← SPEECH-PART(percept)
  semantics ← DISAMBIGUATE(Pragmatics(Semantics(Parse(words))))
  if words = None and action is not a SAY then /* Describe the state */
    return SAY(GENERATE-DESCRIPTION(state))
  else if TYPE[semantics] = Command then /* Obey the command */
    return CONTENTS[semantics]
  else if TYPE[semantics] = Question then /* Answer the question */
    answer ← ASK(KB, semantics)
    return SAY(GENERATE-DESCRIPTION(answer))
  else if TYPE[semantics] = Statement then /* Believe the statement */
    TELL(KB, CONTENTS[semantics])
    /* If we fall through to here, do a "regular" action */
  return FIRST(PLANNER(KB, state))
```

Figure 22.3
function CHART-PARSE(words, grammar) returns chart

chart ← array[0...LENGTH(words)] of empty lists
ADD-EDGE([0, 0, S' → • S])
for i ← from 0 to LENGTH(words) do
  SCANNER(i, words[i])
return chart

procedure ADD-EDGE(edge)
  /* Add edge to chart, and see if it extends or predicts another edge. */
  if edge not in chart[END(edge)] then
    append edge to chart[END(edge)]
  if edge has nothing after the dot then EXTENDER(edge)
  else PREDICTOR(edge)

procedure SCANNER(j, word)
  /* For each edge expecting a word of this category here, extend the edge. */
  for each [i, j, A → α • B β] in chart[j] do
    if word is of category B then
      ADD-EDGE([i, j+1, A → α B • β])

procedure PREDICTOR(i, j, A → α • B β)
  /* Add to chart any rules for B that could help extend this edge */
  for each (B → γ) in Rewrites-For(B, grammar) do
    ADD-EDGE([j, j, B → • γ])

procedure EXTENDER(j, k, B → γ •)
  /* See what edges can be extended by this edge */
  eB ← the edge that is the input to this procedure
  for each [i, j, A → α • B' β] in chart[j] do
    if B = B' then
      ADD-EDGE([i, k, A → α eB • β])

Figure 22.9
function VITERBI-SEGMENTATION(text, P) returns best words and their probabilities
inputs: text, a string of characters with spaces removed
        P, a unigram probability distribution over words

n ← LENGTH(text)
words ← empty vector of length n + 1
best ← vector of length n + 1, initially all 0.0
best[0] ← 1.0
/* Fill in the vectors best, words via dynamic programming */
for i = 0 to n do
    for j = 0 to i - 1 do
        word ← text[j:i]
        w ← LENGTH(word)
        if P[word] × best[i - w] ≥ best[i] then
            best[i] ← P[word] × best[i - w]
            words[i] ← word
/* Now recover the sequence of best words */
sequence ← the empty list
i ← n
while i > 0 do
    push words[i] onto front of sequence
    i ← i − LENGTH(words[i])
/* Return sequence of best words and overall probability of sequence */
return sequence, best[i]
24 PERCEPTION

function ALIGN(image, model) returns a solution or failure
inputs: image, a list of image feature points
        model, a list of model feature points

for each p1, p2, p3 in TRIPLETS(image) do
    for each m1, m2, m3 in TRIPLETS(model) do
        Q ← FIND-TRANSFORM(p1, p2, p3, m1, m2, m3)
        if projection according to Q explains image then
            return Q
    return failure

Figure 24.22
function MONTE-CARLO-LOCALIZATION(a, z, N, model, map) returns a set of samples
inputs: a, the previous robot motion command
z, a range scan with M readings z₁, . . . , zₘ
N, the number of samples to be maintained
model, a probabilistic environment model with pose prior P(X₀),
motion model P(Xₜ | X₀, A₀), and range sensor noise model P(Z | Ẑ)
map, a 2D map of the environment
static: 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], A₀ = a)
    W[i] ← 1
for j = 1 to M do
    ẑ ← EXACT-RANGE(j, S[i], map)
    W[i] ← W[i] \cdot P(Z = ẑ | Z = ẑ)
S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)
return S

Figure 25.8
PHILOSOPHICAL FOUNDATIONS
AI: PRESENT AND FUTURE