1 INTRODUCTION

2 INTELLIGENT AGENTS

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  static: 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 2.8

```
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 2.10

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

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

Figure 2.13

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** an action
  static: *state*, a description of the current world state
      rules, a set of condition–action rules
      action, the most recent action, initially none

  state ← UPDATE-STATE(*state*, *action*, *percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← RULE-ACTION[*rule*]
  return *action*

Figure 2.16

3 SOLVING PROBLEMS BY SEARCHING

```
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 ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.19

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Figure 3.25

4 INFORMED SEARCH AND EXPLORATION

```
function RECURSIVE-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, min(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

```
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 4.17

```
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))
```

Figure 4.21

```
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       LRTA*-COST(s, b, result[b, s], H)
             b∈ ACTIONS(s)
   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

5 CONSTRAINT SATISFACTION PROBLEMS

```
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
   return failure
```

Figure 5.4

```
function AC-3(csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] do
            add (Xk, Xi) to queue
--------------------------------------------------------------------------------
function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff we remove a value
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint between Xi and Xj
         then delete x from DOMAIN[Xi]; removed ← 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 ← an initial complete assignment for csp
   for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen, conflicted variable from VARIABLES[csp]
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = 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

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

Figure 7.15

```
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] do
        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)
  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($KB, \neg S_{x,y}$)
if *breeze* **then** TELL($KB, B_{x,y}$) **else** TELL($KB, \neg B_{x,y}$)
if *glitter* **then** *action* ← *grab*
else if *plan* is nonempty then *action* ← POP(*plan*)
else if for some fringe square [*i*,*j*], ASK($KB, (\neg P_{i,j} \wedge \neg W_{i,j})$) is *true* **or**
for some fringe square [*i*,*j*], ASK($KB, (P_{i,j} \vee W_{i,j})$) 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

9 INFERENCE IN FIRST-ORDER LOGIC

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

   if θ = failure then return failure
   else if x = y then return θ
   else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
   else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
   else if COMPOUND?(x) and COMPOUND?(y) then
       return UNIFY(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
   inputs: var, a variable
           x, any expression
           θ, the substitution built up so far

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

Figure 9.2

```
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, p'1 ∧ ... ∧ p'n)
                    for some p'1, ..., p'n 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

10 KNOWLEDGE REPRESENTATION

11 PLANNING

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

Figure 11.3

$Init(At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
PRECOND: $At(Spare, Trunk)$
EFFECT: $\neg At(Spare, Trunk) \land At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
PRECOND: $At(Flat, Axle)$
EFFECT: $\neg At(Flat, Axle) \land At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
PRECOND: $At(Spare, Ground) \land \neg At(Flat, Axle)$
EFFECT: $\neg At(Spare, Ground) \land At(Spare, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
$\land \neg At(Flat, Ground) \land \neg At(Flat, Axle))$

Figure 11.5

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

Figure 11.7

$Init(At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \land At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \land At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \land \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \land At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
    $\land \neg At(Flat, Ground) \land \neg At(Flat, Axle))$

Figure 11.11

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

Figure 11.16

function GRAPHPLAN(*problem*) **returns** solution or failure

  graph ← INITIAL-PLANNING-GRAPH(*problem*)
  goals ← GOALS[*problem*]
  loop do
    if *goals* all non-mutex 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-GRAPH(*graph*, *problem*)

Figure 11.19

```
function SATPLAN(problem, T_max) returns solution or failure
  inputs: problem, a planning problem
          T_max, an upper limit for plan length

  for T = 0 to T_max do
    cnf, mapping ← TRANSLATE-TO-SAT(problem, T)
    assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
      return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

Figure 11.22

12 PLANNING AND ACTING IN THE REAL WORLD

$$
\begin{aligned}
&Init(Chassis(C_1) \land Chassis(C_2)\\
&\quad \land Engine(E_1, C_1, 30) \land Engine(E_2, C_2, 60)\\
&\quad \land Wheels(W_1, C_1, 30) \land Wheels(W_2, C_2, 15))\\
&Goal(Done(C_1) \land Done(C_2))
\end{aligned}
$$

$$
\begin{aligned}
&Action(AddEngine(e, c, m),\\
&\quad \text{PRECOND:} Engine(e, c, d) \land Chassis(c) \land \neg EngineIn(c),\\
&\quad \text{EFFECT:} EngineIn(c) \land Duration(d))\\
&Action(AddWheels(w, c), \text{PRECOND:} Wheels(w, c, d) \land Chassis(c),\\
&\quad \text{EFFECT:} WheelsOn(c) \land Duration(d))\\
&Action(Inspect(c), \text{PRECOND:} EngineIn(c) \land WheelsOn(c) \land Chassis(c),\\
&\quad \text{EFFECT:} Done(c) \land Duration(10))
\end{aligned}
$$

Figure 12.2

$Init(Chassis(C_1) \wedge Chassis(C_2)$
$\quad \wedge\ Engine(E_1, C_1, 30) \wedge Engine(E_2, C_2, 60)$
$\quad \wedge\ Wheels(W_1, C_1, 30) \wedge Wheels(W_2, C_2, 15)$
$\quad \wedge\ EngineHoists(1) \wedge WheelStations(1) \wedge Inspectors(2))$
$Goal(Done(C_1) \wedge Done(C_2))$

$Action(AddEngine(e, c, m),$
PRECOND:$Engine(e, c, d) \wedge Chassis(c) \wedge \neg EngineIn(c),$
EFFECT:$EngineIn(c) \wedge Duration(d),$
RESOURCE:$EngineHoists(1))$
$Action(AddWheels(w, c),$
PRECOND: $Wheels(w, c, d) \wedge Chassis(c),$
EFFECT: $WheelsOn(c) \wedge Duration(d),$
RESOURCE: $WheelStations(1))$
$Action(Inspect(c),$
PRECOND:$EngineIn(c) \wedge WheelsOn(c),$
EFFECT:$Done(c) \wedge Duration(10),$
RESOURCE:$Inspectors(1))$

Figure 12.5

$Action(BuyLand,$ PRECOND:$Money,$ EFFECT:$Land \wedge \neg Money)$
$Action(GetLoan,$ PRECOND:$GoodCredit,$ EFFECT:$Money \wedge Mortgage)$
$Action(BuildHouse,$ PRECOND:$Land,$ EFFECT:$House)$

$Action(GetPermit,$ PRECOND:$Land,$ EFFECT:$Permit)$
$Action(HireBuilder,$ EFFECT:$Contract)$
$Action(Construction,$ PRECOND:$Permit \wedge Contract,$
EFFECT:$HouseBuilt \wedge \neg Permit)$
$Action(PayBuilder,$ PRECOND:$Money \wedge HouseBuilt,$
EFFECT: $\neg Money \wedge House \wedge \neg Contract)$

$Decompose(BuildHouse,$
$Plan($STEPS: $\{S_1 : GetPermit,\ S_2 : HireBuilder,$
$S_3 : Construction,\ S_4 : PayBuilder\}$
ORDERINGS: $\{Start \prec S_1 \prec S_3 \prec S_4 \prec Finish,\quad Start \prec S_2 \prec S_3\},$
LINKS: $\{Start \xrightarrow{Land} S_1,\ Start \xrightarrow{Money} S_4,$
$S_1 \xrightarrow{Permit} S_3,\ S_2 \xrightarrow{Contract} S_3,\ S_3 \xrightarrow{HouseBuilt} S_4,$
$S_4 \xrightarrow{House} Finish,\ S_4 \xrightarrow{\neg Money} Finish\}))$

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 = 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 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
```

Figure 12.28

$Agents(A,\ B)$
$Init(At(A,\ [Left,\ Baseline])\ \wedge\ At(B,\ [Right,\ Net])\ \wedge$
$\quad Approaching(Ball,\ [Right,\ Baseline]))\ \wedge\ Partner(A, B)\ \wedge\ Partner(B, A)$
$Goal(Returned(Ball)\ \wedge\ At(agent,\ [x,\ Net]))$
$Action(Hit(agent,\ Ball),$
$\quad$ PRECOND:$Approaching(Ball,\ [x,\ y])\ \wedge\ At(agent,\ [x,\ y])\ \wedge$
$\qquad Partner(agent, partner)\ \wedge\ \neg\ At(partner, [x, y])$
$\quad$ EFFECT:$Returned(Ball))$
$Action(Go(agent,\ [x,\ y]),$
$\quad$ PRECOND:$At(agent,\ [a,\ b]),$
$\quad$ EFFECT:$At(agent,\ [x, y])\ \wedge\ \neg\ At(agent,\ [a,\ b]))$

Figure 12.30

13 UNCERTAINTY

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 13.2

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 $\{X\}$ ∪ **E** ∪ **Y** /* **Y** = *hidden variables* */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X **do**
    Q(x_i) ← ENUMERATE-JOINT(x_i, **e**, **Y**, [ ], **P**)
  return NORMALIZE(**Q**(X))

---

function ENUMERATE-JOINT(x, **e**, $vars$, $values$, **P**) **returns** a real number
  if EMPTY?($vars$) **then return P**$(x$, **e**, $values)$
  Y ← FIRST($vars$)
  return $\sum_y$ ENUMERATE-JOINT(x, **e**, REST($vars$), $[y|values]$, **P**)

Figure 13.6

14 PROBABILISTIC REASONING

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

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
     extend e with value x_i for X
     Q(x_i) ← 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 ← FIRST(vars)
  if Y has value y in e
     then return P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e)
     else return Σ_y P(y | parents(Y)) × 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 Bayesian network specifying joint distribution P(X_1, ..., X_n)

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

Figure 14.12

```
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, ..., X_n)

  x ← an event with n elements
  for i = 1 to n do
      x_i ← a random sample from P(X_i | 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 ← PRIOR-SAMPLE(bn)
      if x is consistent with e then
         N[x] ← N[x]+1 where x is the value of X in x
  return NORMALIZE(N[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 ← WEIGHTED-SAMPLE(bn)
      W[x] ← W[x] + w where x is the value of X in x
  return NORMALIZE(W[X])
-----------------------------------------------------------------
function WEIGHTED-SAMPLE(bn, e) returns an event and a weight

  x ← an event with n elements; w ← 1
  for i = 1 to n do
      if X_i has a value x_i in e
          then w ← w × P(X_i = x_i | parents(X_i))
          else x_i ← a random sample from P(X_i | parents(X_i))
  return x, w
```

Figure 14.19

```
function MCMC-ASK(X, e, bn, 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 bn
                   x, the current state of the network, initially copied from e

  initialize x with random values for the variables in Z
  for j = 1 to N do
     N[x] ← 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 NORMALIZE(N[X])
```

Figure 14.21

15 PROBABILISTIC REASONING OVER TIME

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

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

Figure 15.5

```
function 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 × 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 PRIOR[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 ← diagonal matrix containing P(e_t|X_t)
  if t > d then
     f ← FORWARD(f, e_t)
     remove e_{t-d-1} from the beginning of e_{t-d:t}
     O_{t-d} ← diagonal matrix containing P(e_{t-d}|X_{t-d})
     B ← O_{t-d}^{-1} T^{-1} B T O_t
  else B ← B T O_t
  t ← t + 1
  if t > d then return NORMALIZE(f × B1) else return null
```

Figure 15.8

```
function PARTICLE-FILTERING(e, N, dbn) returns a set of samples for the next time step
  inputs: e, the new incoming evidence
          N, the number of samples to be maintained
          dbn, a DBN with prior P(X_0), transition model P(X_1|X_0), and sensor model P(E_1|X_1)
  static: 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 do
     S[i] ← sample from P(X_1|X_0 = S[i])
     W[i] ← P(e|X_1 = S[i])
  S ← 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 ← 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

17 MAKING COMPLEX DECISIONS

```
function VALUE-ITERATION(mdp, ε) returns a utility function
  inputs: mdp, an MDP with states S, transition model T, reward function R, discount γ
          ε, the maximum error allowed in the utility of any state
  local variables: U, U′, vectors of utilities for states in S, initially zero
                   δ, the maximum change in the utility of any state in an iteration

  repeat
     U ← U′; δ ← 0
     for each state s in S do
         U′[s] ← R[s] + γ max_a Σ_{s′} T(s, a, s′) U[s′]
         if |U′[s] − U[s]| > δ then δ ← |U′[s] − U[s]|
  until δ < ε(1 − γ)/γ
  return U
```

Figure 17.5

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, transition model T
  local variables: U, U', vectors of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_a \sum_{s'} T(s,a,s')\ U[s'] > \sum_{s'} T(s,\pi[s],s')\ U[s']$ **then**
        $\pi[s] \leftarrow \mathrm{argmax}_a \sum_{s'} T(s,a,s')\ U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return P

Figure 17.9

18 LEARNING FROM OBSERVATIONS

```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attribs, 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 attribs is empty then return MAJORITY-VALUE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attribs, examples)
     tree ← a new decision tree with root test best
     m ← MAJORITY-VALUE(examples_i)
     for each value v_i of best do
        examples_i ← {elements of examples with best = v_i}
        subtree ← DECISION-TREE-LEARNING(examples_i, attribs − best, m)
        add a branch to tree with label v_i and subtree subtree
     return tree
```

Figure 18.6

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

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

Figure 18.12

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

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

Figure 18.17

19 KNOWLEDGE IN LEARNING

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

  H ← any hypothesis consistent with the first example in examples
  for each remaining example in examples do
     if e is false positive for H then
        H ← choose a specialization of H consistent with examples
     else if e is false negative for H then
        H ← choose a generalization of H consistent with examples
     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 ← the set of all hypotheses
  for each example e in examples do
     if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  return V
---------------------------------------------------------------------
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

Figure 19.5

```
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 ← 0, ..., 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

20 STATISTICAL LEARNING METHODS

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

   repeat
      for each e in examples do
         in ← Σ_{j=0}^{n} W_j x_j[e]
         Err ← y[e] − g(in)
         W_j ← W_j + α × Err × g′(in) × 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 x and output vector 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 ← x_j[e]
        for ℓ = 2 to M do
           in_i ← Σ_j W_{j,i} a_j
           a_i ← g(in_i)
        for each node i in the output layer do
           Δ_i ← g'(in_i) × (y_i[e] − a_i)
        for ℓ = M − 1 to 1 do
           for each node j in layer ℓ do
              Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
              for each node i in layer ℓ + 1 do
                 W_{j,i} ← W_{j,i} + α × a_j × Δ_i
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.27

21 REINFORCEMENT LEARNING

```
function PASSIVE-ADP-AGENT(percept) returns an action
   inputs: percept, a percept indicating the current state s' and reward signal r'
   static: π, a fixed policy
           mdp, an MDP with model T, rewards R, discount γ
           U, a table of utilities, initially empty
           N_sa, a table of frequencies for state-action pairs, initially zero
           N_sas', a table of frequencies for state-action-state triples, initially zero
           s, a, the previous state and action, initially null

   if s' is new then do U[s'] ← r'; R[s'] ← r'
   if s is not null then do
      increment N_sa[s, a] and N_sas'[s, a, s']
      for each t such that N_sas'[s, a, t] is nonzero do
         T[s, a, t] ← N_sas'[s, a, t] / N_sa[s, a]
   U ← VALUE-DETERMINATION(π, U, mdp)
   if TERMINAL?[s'] then s, a ← null else s, a ← s', π[s']
   return a
```

Figure 21.3

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

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

Figure 21.6

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

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

Figure 21.11

22 COMMUNICATION

```
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 */
   e_B ← 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 e_B • β])
```

Figure 22.9

23 PROBABILISTIC LANGUAGE PROCESSING

```
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]
```

Figure 23.2

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

25 ROBOTICS

```
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 z_1, ..., z_M
           N, the number of samples to be maintained
           model, a probabilistic environment model with pose prior P(X_0),
              motion model P(X_1|X_0, A_0), 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(X_0)
   local variables: W, a vector of weights of size N

   for i = 1 to N do
      S[i] ← sample from P(X_1|X_0 = S[i], A_0 = a)
      W[i] ← 1
      for j =1 to M do
         ẑ ← EXACT-RANGE(j, S[i], map)
         W[i] ← W[i] · P(Z = z_j|Ẑ = ẑ)
   S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)
   return S
```

Figure 25.8

26 PHILOSOPHICAL FOUNDATIONS

27 AI: PRESENT AND FUTURE