1 INTRODUCTION

2 INTELLIGENT AGENTS

function TABLE-DRIVEN-AGENT(*percept*) **returns** an action
persistent: *percepts*, a sequence, initially empty
table, a table of actions, indexed by percept sequences, initially fully specified

append *percept* to the end of *percepts*
action ← LOOKUP(*percepts*, *table*)
return *action*

Figure 2.1 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Forward*
else if *location* = *B* **then return** *Backward*

Figure 2.2 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure **??**.

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

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← *rule*.ACTION
return *action*

Figure 2.3 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

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

  $state$ ← UPDATE-STATE($state$, $action$, $percept$, $transition\_model$, $sensor\_model$)
  $rule$ ← RULE-MATCH($state$, $rules$)
  $action$ ← $rule$.ACTION
  return $action$

Figure 2.4 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

3 SOLVING PROBLEMS BY SEARCHING

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  frontier ← a priority queue ordered by f, with a node for the initial state
  reached ← a table of {state: node}, initially empty
  while frontier is not empty do
    node ← POP(frontier)
    if node is a goal then return node
    for child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Figure 3.1 The best-first search algorithm. On each step we choose to expand the node that is "best"—that is, that has the minimum value of $f(n)$ among all the nodes in the frontier. Children of that node are added to the frontier if they have not been reached before, or are re-added if they are reached with a path that has a lower path-cost than the previous path.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node, or failure
  if the initial state is a goal then
    return a node for the initial state
  frontier ← a FIFO queue, with a node for the initial state
  reached ← a set of states, initially empty
  while frontier is not empty do
    node ← POP(frontier)
    if node is a goal then return node
    for child in EXPAND(problem, node) do
      s ← child.STATE
      if s is a goal then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Figure 3.2 Breadth-first search algorithm.

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

---

function DEPTH-LIMITED-SEARCH($problem$, $limit$) **returns** a node or failure or cutoff
  $frontier \leftarrow$ a LIFO queue (stack) with a node for the initial state
  $result \leftarrow$ failure
  while $frontier$ is not empty **do**
    $node \leftarrow$ POP($frontier$)
    if DEPTH($node$) $> limit$ **then**
      $result \leftarrow$ cutoff
    else
      for $child$ **in** EXPAND($problem, node$) **do**
        if $child$ is a goal **then return** $child$
        if not ISSHORTCYCLE($node$) **then**
          add $child$ to the $frontier$
  return $result$

Figure 3.3 Iterative deepening and depth-limited tree search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns $failure$, meaning that no solution exists. The depth-limited search algorithm returns three different types of values: either a solution, or $failure$ when it has exhausted all nodes and proved there is no solution at any depth, or $cutoff$ to mean there might be a solution at a deeper depth than ℓ. Note that this is a tree search algorithm that does not keep track of $reached$ states, and thus uses much less memory that best-first search, but it runs the risk of visiting the same state multiple times on different paths, and failing to be systematic. To partially counter that, the ISSHORTCYCLE(CHILD) test looks at the parent and several generations of grandparents to see if a cycle is detected, and if so refuses to put the offending child on the frontier.

```
function BIDIRECTIONAL-SEARCH(problem) returns a solution path, or failure
  if problem's initial state is the goal then return empty path to initial state
  frontier ← a priority queue ordered by F, with a node for the initial state
  frontier′ ← a priority queue ordered by F, with a node for the goal state
  reached ← a table of {state: node}, initially empty
  reached′ ← a table of {state: node}, initially empty
  solution ← failure
  while not TERMINATED(F(solution), frontier, frontier′))
    if frontier has a node with lower F then frontier′ then
      solution ← PROCEED(FORWARD, frontier, reached, reached′, solution)
    else do
      solution ← PROCEED(BACKWARD, frontier′, reached′, reached, solution)
  return solution
```

```
function PROCEED(direction, frontier, reached, frontier′, solution) returns a solution
  /* Expand one node on one of the frontiers; check against the other frontier. */
  parent ← POP(frontier)
  for child in EXPAND(parent, direction) do
    s ← child.STATE
    if s is not in reached or child is a cheaper path than reached[s] then
      reached[s] ← child
      add child to frontier
      if s is in frontier′ and child is a cheaper path than solution then
        solution ← child + REVERSE(reached′[s])
  return solution
```

```
function F(node) returns a number
  return max(g(node) + h(node), 2 × g(node))
```

```
function TERMINATED(C, frontier, frontier′) returns a boolean
  /* Terminate if all future solutions will be more expensive than C. */
  if frontier is empty or frontier′ is empty then return true
  A ← TOP(frontier)
  B ← TOP(frontier′)
return C < g(A) + g(B) or C < min(f(A), f(B)) or C < min(g(A) + h(A), g(B) +
h(B)
```

Figure 3.4 Bidirectional search keeps two frontiers and two tables of reached states. When a path in one frontier intersects a path in the other, the two are joined to form the solution.

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

function RBFS($problem$, $node$, f_limit) **returns** a solution, or failure and a new f-cost limit
  if $problem$.GOAL-TEST($node$.STATE) **then return** $node$
  $successors \leftarrow$ EXPAND($node$)
  if successors is empty **then return** $failure$, ∞
  for s **in** $successors$ **do** /* *update f with value from previous search* */
      $s.f \leftarrow \max(s.g + s.h,\ node.f)$)
  loop do
      $best \leftarrow$ the lowest f-value node in $successors$
      if $best.f > f\_limit$ **then return** $failure$, $best.f$
      $alternative \leftarrow$ the second-lowest f-value among $successors$
      $result, best.f \leftarrow$ RBFS($problem$, $best$, $\min(f\_limit, alternative)$)
      if $result \neq failure$ **then return** $result$, $best.f$

Figure 3.5 The algorithm for recursive best-first search.

4 SEARCH IN COMPLEX ENVIRONMENTS

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

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

Figure 4.1 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

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

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

Figure 4.2 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The $schedule$ input determines the value of the "temperature" T as a function of time.

```
function GENETIC-ALGORITHM(population, FITNESS-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
     weights ← [FITNESS-FN(p) for p in population]
     for i = 1 to SIZE(population) do
        x, y ← WEIGHTED-RANDOM-SELECTION(population, weights, 2)
        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 4.3 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure **??**, with one variation: in this version, each recombination of two parents produces only one offspring, not two.

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

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

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

Figure 4.4 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.) [[TODO: make more like other search algorithms?]]

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

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

Figure 4.5 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

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

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ←        min        LRTA*-COST(s, b, result[s, b], H)
              b ∈ ACTIONS(s)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

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

Figure 4.6 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

5 ADVERSARIAL SEARCH AND GAMES

function MINIMAX-DECISION($state$) **returns** $an\ action$
  return $\arg\max_{a\ \in\ \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state$, a))

---

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

---

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

Figure 5.1 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a\,\in\,S}\,f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

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

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

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

Figure 5.2 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure **??**, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACKPROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 5.3 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACKPROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

6 CONSTRAINT SATISFACTION PROBLEMS

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

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

---

function REVISE(csp, $X_i,\ X_j$) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

Figure 6.1 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (?) because it was the third version developed in the paper.

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

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

Figure 6.2 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter **??**. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or *k*-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
     if current is a solution for csp then return current
     var ← a randomly chosen conflicted variable from csp.VARIABLES
     value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
     set var = value in current
  return failure
```

Figure 6.3 The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

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

   n ← number of variables in X
   assignment ← an empty assignment
   root ← any variable in X
   X ← TOPOLOGICALSORT(X, root)
   for j = n down to 2 do
      MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
      if it cannot be made consistent then return failure
   for i = 1 to n do
      assignment[X_i] ← any consistent value from D_i
      if there is no consistent value then return failure
   return assignment
```

Figure 6.4 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

7 LOGICAL AGENTS

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

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

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

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

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

---

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

Figure 7.2 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

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

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ← { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

Figure 7.3 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

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

Figure 7.4 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })
--------------------------------------------------------------------------
function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false}))
```

Figure 7.5 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
     if model satisfies clauses then return model
     clause ← a randomly selected clause from clauses that is false in model
     with probability p flip the value in model of a randomly selected symbol from clause
     else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

Figure 7.6 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

function HYBRID-WUMPUS-AGENT($percept$) **returns** an $action$
  inputs: $percept$, a list, $[stench, breeze, glitter, bump, scream]$
  persistent: KB, a knowledge base, initially the atemporal "wumpus physics"
    t, a counter, initially 0, indicating time
    $plan$, an action sequence, initially empty

  TELL(KB, MAKE-PERCEPT-SENTENCE($percept, t$))
  TELL the KB the temporal "physics" sentences for time t
  $safe \leftarrow \{[x,y] \; : \; \text{ASK}(KB, OK^t_{x,y}) = true\}$
  if ASK(KB, $Glitter^t$) $= true$ **then**
    $plan \leftarrow [Grab]$ + PLAN-ROUTE($current$, {[1,1]}, $safe$) + $[Climb]$
  if $plan$ is empty **then**
    $unvisited \leftarrow \{[x,y] \; : \; \text{ASK}(KB, L^{t'}_{x,y}) = false$ for all $t' \leq t\}$
    $plan \leftarrow$ PLAN-ROUTE($current$, $unvisited \cap safe$, $safe$)
  if $plan$ is empty and ASK(KB, $HaveArrow^t$) $= true$ **then**
    $possible\_wumpus \leftarrow \{[x,y] \; : \; \text{ASK}(KB, \neg W_{x,y}) = false\}$
    $plan \leftarrow$ PLAN-SHOT($current$, $possible\_wumpus$, $safe$)
  if $plan$ is empty **then** // no choice but to take a risk
    $not\_unsafe \leftarrow \{[x,y] \; : \; \text{ASK}(KB, \neg OK^t_{x,y}) = false\}$
    $plan \leftarrow$ PLAN-ROUTE($current$, $unvisited \cap not\_unsafe$, $safe$)
  if $plan$ is empty **then**
    $plan \leftarrow$ PLAN-ROUTE($current$, $\{[1,1]\}$, $safe$) + $[Climb]$
  $action \leftarrow$ POP($plan$)
  TELL(KB, MAKE-ACTION-SENTENCE($action, t$))
  $t \leftarrow t + 1$
  return $action$

---

function PLAN-ROUTE($current$, $goals$, $allowed$) **returns** an action sequence
  inputs: $current$, the agent's current position
    $goals$, a set of squares; try to plan a route to one of them
    $allowed$, a set of squares that can form part of the route

  $problem \leftarrow$ ROUTE-PROBLEM($current$, $goals$, $allowed$)
  return A*-GRAPH-SEARCH($problem$)

Figure 7.7 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

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

Figure 7.8 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step t and axioms are included for each time step up to t. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned $true$ in the model. If no model exists, then the process is repeated with the goal moved one step later.

8 FIRST-ORDER LOGIC

9 INFERENCE IN FIRST-ORDER LOGIC

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

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

---

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

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

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

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

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

Figure 9.2 A conceptually straightforward, but inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

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

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

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

Figure 9.3 A simple backward-chaining algorithm for first-order knowledge bases.

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

    $trail \leftarrow$ GLOBAL-TRAIL-POINTER()
    if ax = [ ] and UNIFY(y, az) **then** CALL($continuation$)
    RESET-TRAIL($trail$)
    $a, x, z \leftarrow$ NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY($ax, [a \mid x]$) and UNIFY($az, [a \mid z]$) **then** APPEND($x, y, z, continuation$)

Figure 9.4 Pseudocode representing the result of compiling the `Append` predicate. The function NEW-VARIABLE returns a new variable, distinct from all other variables used so far. The procedure CALL($continuation$) continues execution with the specified continuation.

10 KNOWLEDGE REPRESENTATION

11 AUTOMATED PLANNING

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

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

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

Figure 11.2 The simple spare tire problem.

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

Figure 11.3 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$.

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

$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $a = x \land b = y$
STEPS: $[\,]\ )$
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a-1, b])$
STEPS: $[Left, Navigate([a-1, b], [x, y])]\ )$
$Refinement(Navigate([a, b], [x, y]),$
PRECOND: $Connected([a, b], [a+1, b])$
STEPS: $[Right, Navigate([a+1, b], [x, y])]\ )$
...

Figure 11.4 Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution, or failure
  frontier ← a FIFO queue with [Act] as the only element
  loop do
      if EMPTY?(frontier) then return failure
      plan ← POP(frontier)   /* chooses the shallowest plan in frontier */
      hla ← the first HLA in plan, or null if none
      prefix,suffix ← the action subsequences before and after hla in plan
      outcome ← RESULT(problem.INITIAL-STATE, prefix)
      if hla is null then   /* so plan is primitive and outcome is its result */
          if outcome satisfies problem.GOAL then return plan
      else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
          frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Figure 11.5 A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

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

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

Figure 11.6 A hierarchical planning algorithm that uses angelic semantics to identify and commit to high-level plans that work while avoiding high-level plans that don't. The predicate MAKING-PROGRESS checks to make sure that we aren't stuck in an infinite regression of refinements. At top level, call ANGELIC-SEARCH with $[Act]$ as the $initialPlan$.

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

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

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

Figure 11.7 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.

12 QUANTIFYING UNCERTAINTY

function DT-AGENT($percept$) **returns** an $action$
  persistent: $belief\_state$, probabilistic beliefs about the current state of the world
      $action$, the agent's action

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

Figure 12.1 A decision-theoretic agent that selects rational actions.

13 PROBABILISTIC REASONING

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

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

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

Figure 13.1 The enumeration algorithm for answering queries on Bayes nets.

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

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

Figure 13.2 The variable elimination algorithm for inference in Bayes nets.

```
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 each variable X_i in X_1, ..., X_n do
     x[i] ← a random sample from P(X_i | parents(X_i))
  return x
```

Figure 13.3 A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X | e)
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network
          N, the total number of samples to be generated
  local variables: N, a vector of counts for each value of X, initially zero

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

Figure 13.4 The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

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

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

```
function WEIGHTED-SAMPLE(bn, e) returns an event and a weight
  w ← 1; x ← an event with n elements initialized from e
  for each variable X_i in X_1, ..., X_n do
     if X_i is an evidence variable with value x_i in e
        then w ← w × P(X_i = x_i | parents(X_i))
        else x[i] ← a random sample from P(X_i | parents(X_i))
  return x, w
```

Figure 13.5 The likelihood-weighting algorithm for inference in Bayesian networks. In WEIGHTED-SAMPLE, each nonevidence variable is sampled according to the conditional distribution given the values already sampled for the variable's parents, while a weight is accumulated based on the likelihood for each evidence variable.

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

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

Figure 13.6 The Gibbs sampling algorithm for approximate inference in Bayes nets; this version cycles through the variables, but choosing variables at random also works.

14 PROBABILISTIC REASONING OVER TIME

function FORWARD-BACKWARD(**ev**, $prior$) **returns** a vector of probability distributions
inputs: **ev**, a vector of evidence values for steps $1, \ldots, t$
$prior$, the prior distribution on the initial state, $\mathbf{P}(\mathbf{X}_0)$
local variables: **fv**, a vector of forward messages for steps $0, \ldots, t$
b, a representation of the backward message, initially all 1s
sv, a vector of smoothed estimates for steps $1, \ldots, t$

fv$[0] \leftarrow prior$
for $i = 1$ **to** t **do**
fv$[i] \leftarrow$ FORWARD(**fv**$[i-1]$, **ev**$[i]$)
for $i = t$ **downto** 1 **do**
sv$[i] \leftarrow$ NORMALIZE(**fv**$[i] \times$ **b**)
b $\leftarrow$ BACKWARD(**b**, **ev**$[i]$)
return sv

Figure 14.1 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (**??**) and (**??**), respectively.

```
function FIXED-LAG-SMOOTHING(e_t, hmm, d) returns a distribution over 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
   persistent: t, the current time, initially 1
               f, the forward message P(X_t | e_{1:t}), initially hmm.PRIOR
               B, the d-step backward transformation matrix, initially the identity matrix
               e_{t-d:t}, double-ended list of evidence from t - d to t, initially empty
   local variables: 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-d})
      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 + 1 then return NORMALIZE(f × B1) else return null
```

Figure 14.2 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step. Notice that the final output NORMALIZE($\mathbf{f} \times \mathbf{B1}$) is just $\alpha\,\mathbf{f} \times \mathbf{b}$, by Equation (**??**).

```
function PARTICLE-FILTERING(e, N, dbn) returns a set of samples for the next time step
   inputs: e, the new incoming evidence
           N, the number of samples to be maintained
           dbn, a DBN defined by P(X_0), P(X_1 | X_0), and P(E_1 | X_1)
   persistent: S, a vector of samples of size N, initially generated from P(X_0)
   local variables: W, a vector of weights of size N

   for i = 1 to N do
      S[i] ← sample from P(X_1 | X_0 = S[i])    /* step 1 */
      W[i] ← P(e | X_1 = S[i])                  /* step 2 */
   S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)        /* step 3 */
   return S
```

Figure 14.3 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.

15 MAKING SIMPLE DECISIONS

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

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

Figure 15.1 Design of a simple, myopic information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.

16 MAKING COMPLEX DECISIONS

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

  repeat
     U ← U′; δ ← 0
     for each state s in S do
        U′[s] ← max_{a ∈ A(s)} Q-VALUE(mdp, s, a, U)
        if |U′[s] − U[s]| > δ then δ ← |U′[s] − U[s]|
  until δ < ϵ(1 − γ)/γ
  return U
```

Figure 16.1 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (**??**).

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

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

Figure 16.2 The policy iteration algorithm for calculating an optimal policy.

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

  $U' \leftarrow$ a set containing just the empty plan [ ], with $\alpha_{[\,]}(s) = R(s)$
  repeat
    $U \leftarrow U'$
    $U' \leftarrow$ the set of all plans consisting of an action and, for each possible next percept,
      a plan in U with utility vectors computed according to Equation (**??**)
    $U' \leftarrow$ REMOVE-DOMINATED-PLANS(U')
  until MAX-DIFFERENCE(U, U') $< \epsilon(1-\gamma)/\gamma$
  return U

Figure 16.3 A high-level sketch of the value iteration algorithm for POMDPs. The REMOVE-DOMINATED-PLANS step and MAX-DIFFERENCE test are typically implemented as linear programs.

17 MAKING DECISIONS IN MULTIAGENT ENVIRONMENTS

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

Figure 17.1 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: $LeftBaseline$, $RightBaseline$, $LeftNet$, and $RightNet$. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

18 LEARNING FROM EXAMPLES

function DECISION-TREE-LEARNING($examples, attributes, parent\_examples$) **returns** a tree

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

Figure 18.1 The decision-tree learning algorithm. The function IMPORTANCE is described in Section **??**. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

function MODEL-SELECTION($Learner, examples, k$) **returns** a hypothesis

local variables: err, an array, indexed by $size$, storing validation-set error rates
for $size$ = 1 **to** ∞ **do**
  $err[size] \leftarrow$ CROSS-VALIDATION($Learner, size, examples, k$)
  if err is starting to increase significantly **then do**
    $best\_size \leftarrow$ the value of $size$ with minimum $err[size]$
    return $Learner(best\_size, examples)$

---

function CROSS-VALIDATION($Learner, size, examples, k$) **returns** error rate
  average training set error rate,

$errs \leftarrow 0$
for $fold$ = 1 $to\ k\ do$
  $training\_set, validation\_set \leftarrow$ PARTITION($examples, fold, k$)
  $h \leftarrow Learner(size, training\_set)$
  $errs \leftarrow errs$ + ERROR-RATE($h, validation\_set$)
return $errs/k$ // average error rate on validation sets, across k-fold cross-validation

Figure 18.2 An algorithm to select the model that has the lowest error rate on validation data by building models of increasing complexity, and choosing the one with best empirical error rate, err, on the validation data set. $Learner(size, examples)$ returns a hypothesis whose complexity is set by the parameter $size$, and which is trained on the $examples$. DATA-PARTITION(*examples*, *fold*, *k*) splits *examples* into two subsets: a validation set of size N/k and a training set with all the other examples. The split is different for each value of *fold*.

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

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

Figure 18.3 An algorithm for learning decision lists.

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

  for k = 1 **to** K **do**
    $\mathbf{h}[k] \leftarrow L(examples, \mathbf{w})$
    $error \leftarrow 0$
    for j = 1 **to** N **do**
      if $\mathbf{h}[k](x_j) \neq y_j$ **then** $error \leftarrow error + \mathbf{w}[j]$
    for j = 1 **to** N **do**
      if $\mathbf{h}[k](x_j) = y_j$ **then** $\mathbf{w}[j] \leftarrow \mathbf{w}[j] \cdot error/(1 - error)$
    $\mathbf{w} \leftarrow$ NORMALIZE(**w**)
    $\mathbf{z}[k] \leftarrow \log\,(1 - error)/error$
  return WEIGHTED-MAJORITY(**h**, **z**)

Figure 18.4 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**.

19 DEEP NEURAL NETWORKS

function ADAM-OPTIMIZER($f,\ L,\ \boldsymbol{\theta},\ \rho,\ \alpha,\ \delta$) **returns** updated $\boldsymbol{\theta}$
  /* *Defaults:* $\rho_1 = 0.9;\ \rho_2 = 0.999;\ \alpha = 0.001;\ \delta = 10^{-8}$ */
$\boldsymbol{s} \leftarrow \mathbf{0}$
$\boldsymbol{r} \leftarrow \mathbf{0}$
$t \leftarrow 0$
  while $\boldsymbol{\theta}$ has not converged
    $\boldsymbol{x},\ \boldsymbol{y} \leftarrow$ a minibatch of m examples from training set
    $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$ /* *compute gradient* */
    $t \leftarrow t + 1$
    $\boldsymbol{s} \leftarrow \rho_1\, \boldsymbol{s} + (1 - \rho_1)\, \boldsymbol{g}$ /* *Update biased first moment estimate* */
    $\boldsymbol{r} \leftarrow \rho_2\, \boldsymbol{r} + (1 - \rho_2)\, \boldsymbol{g} \odot \boldsymbol{g}$ /* *Update biased second moment estimate* */
    $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$ /* *Correct bias in first moment* */
    $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$ /* *Correct bias in second moment* */
    $\Delta\, \boldsymbol{\theta} = -\,\epsilon\, \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ /* *Compute update (operations applied element-wise)* */
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\, \boldsymbol{\theta}$ /* *Apply update* */

Figure 19.1 The Adam (adaptive moments) optimizer. The function $f(\boldsymbol{x}, \boldsymbol{\theta})$ describes the model and L describes the loss function. ρ_1 and ρ_2 are decay rates for estimates of the two moments, and α is the learning rate, while δ is a small constant used for numerical stabilization.

20 LEARNING PROBABILISTIC MODELS

21 REINFORCEMENT LEARNING

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

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

Figure 21.1 A passive reinforcement learning agent based on adaptive dynamic programming. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page **??**.

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

  if s′ is new then U[s′] ← r′
  if s is not null then
     increment N_s[s]
     U[s] ← U[s] + α(N_s[s])(r + γ U[s′] − U[s])
  if s′.TERMINAL? then s, a, r ← null else s, a, r ← s′, π[s′], r′
  return a
```

Figure 21.2 A passive reinforcement learning agent that learns utility estimates using temporal differences. The step-size function $\alpha(n)$ is chosen to ensure convergence, as described in the text.

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s′ and reward signal r′
  persistent: Q, a table of action values indexed by state and action, initially zero
              N_sa, a table of frequencies for state–action pairs, initially zero
              s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s′) then Q[s′, None] ← r′
  if s is not null then
     increment N_sa[s, a]
     Q[s, a] ← Q[s, a] + α(N_sa[s, a])(r + γ max_a′ Q[s′, a′] − Q[s, a])
  s, a, r ← s′, argmax_a′ f(Q[s′, a′], N_sa[s′, a′]), r′
  return a
```

Figure 21.3 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

22 NATURAL LANGUAGE PROCESSING

```
function CYK-PARSE(words, grammar) returns a table of parse trees
  P ← a table, initially all 0 /* P[X, i, k] is probability of an X spanning words_{i:k} */
  T ← a table /* T[X, i, k] is best X tree spanning words_{i:k} */
  /* Insert lexical categories for each word. */
  for i = 1 to LEN(words) do
    for each grammar lexical rule of form (X → words_i [p]) do
      P[X, i, i] ← p
      T[X, i, i] ← TREE(X, words_i)
  /* Construct X_{i:k} from Y_{i:j} + Z_{j+1:k}, shortest spans first. */
  for (i, j, k) in SUBSPANS(LEN(words)) do
    for each grammar rule of the form (X → Y Z [p]) do
      PYZ ← P[Y, i, j] × P[Z, j + 1, k] × p
      if PYZ > P[X, i, k] do
        P[X, i, k] ← PYZ
        T[X, i, k] ← TREE(X, T[Y, i, j], T[Z, j + 1, k])
  return T
```

```
function SUBSPANS(N) returns (i, j, k) tuples
  for length = 2 to N do
    for i = 1 to N + 1 − varlength do
      k ← i + length − 1
      for j = i to k − 1 do
        yield (i, j, k)
```

Figure 22.1 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable parse tree for the whole sequence, and for each subsequence. It keeps a table of $P[X, i, k]$ giving the probability of the most probable tree of category X spanning $words_{i:k}$. It returns a table, T, in which an entry $T[X, i, k]$ is the most probable tree of category X spanning positions i to k inclusive. The function SUBSPANS returns all tuples (i, j, k) covering a span of $words_{i:k}$, with $i \leq j < k$, listing the tuples by increasing length of the $i : k$ span, so that when we go to combine two shorter spans into a longer one, the shorter spans are already in the table.

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

Figure 22.2 Annotated tree for the sentence "Her eyes were glazed as if she didn't hear or even see him." from the Penn Treebank. Note that in this grammar there is a distinction between an object noun phrase (NP) and a subject noun phrase (NP-SBJ). Note also a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase "hear or even see him" as consisting of two constituent VPs, [VP **hear** [NP *-1]] and [VP [$ADVP$ **even**] **see** [NP *-1]], both of which have a missing object, denoted *-1, which refers to the NP labeled elsewhere in the tree as [NP-1 **him**].

23 DEEP LEARNING FOR NATURAL LANGUAGE PROCESSING

24 PERCEPTION

25 ROBOTICS

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

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

Figure 25.1 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

26 PHILOSOPHY AND ETHICS OF AI

27 THE FUTURE OF AI

28 MATHEMATICAL BACKGROUND

29 NOTES ON LANGUAGES AND ALGORITHMS

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

---

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

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