```
function ALPHA-BETA(node, depth, α, β, maximizing):
    if depth = 0 or TERMINAL(node):
        return EVALUATE(node)

    if maximizing:
        value = -∞
        for each child in CHILDREN(node):
            value = max(value, ALPHA-BETA(child, depth-1, α, β, false))
            α = max(α, value)
            if α ≥ β: break  // β cutoff
        return value
    else:
        value = +∞
        for each child in CHILDREN(node):
            value = min(value, ALPHA-BETA(child, depth-1, α, β, true))
            β = min(β, value)
            if β ≤ α: break  // α cutoff
        return value


function FORWARD-CHAIN(KB, query):
    agenda = KB.facts
    inferred = ∅

    while agenda ≠ ∅:
        p = POP(agenda)
        if p = query: return true
        if p ∉ inferred:
            ADD(inferred, p)
            for each rule in KB.rules:
                if rule premises SATISFIED by inferred:
                    new_fact = rule.conclusion
                    if new_fact ∉ inferred:
                        PUSH(agenda, new_fact)
    return false

function TO-CNF(ϕ):
    ϕ1 = ELIMINATE-IMPLICATIONS(ϕ)
    ϕ2 = MOVE-NEGATIONS-INWARD(ϕ1)
    ϕ3 = SKOLEMIZE(ϕ2)
    ϕ4 = DROP-UNIVERSALS(ϕ3)
    ϕ5 = DISTRIBUTE-OR-OVER-AND(ϕ4)
    return ϕ5

function ELIMINATE-IMPLICATIONS(ϕ):
    if ϕ = P→Q: return ¬P ∨ Q
    if ϕ = P↔Q: return (P∧Q) ∨ (¬P∧¬Q)
    else: apply recursively

function MOVE-NEGATIONS-INWARD(ϕ):
    if ϕ = ¬¬P: return P
    if ϕ = ¬(P∧Q): return ¬P ∨ ¬Q
    if ϕ = ¬(P∨Q): return ¬P ∧ ¬Q
    if ϕ = ¬∀xP: return ∃x¬P
    if ϕ = ¬∃xP: return ∀x¬P
    else: apply recursively

function SKOLEMIZE(ϕ, uvars=[]):
    if ϕ = ∀xψ: return ∀x SKOLEMIZE(ψ, uvars∪[x])
```

```
  if ϕ = ∃xψ: return SUBSTITUTE(ψ, x, SKOLEM-FUNC(x,uvars))
  else: apply recursively

function DISTRIBUTE-OR-OVER-AND(ϕ):
  if ϕ = P ∨ (Q∧R): return (P∨Q) ∧ (P∨R)
  if ϕ = (P∧Q) ∨ R: return (P∨R) ∧ (Q∨R)
  else: apply recursively


function UNIFY(x, y, θ):
  if θ = None: return None
  if x = y: return θ
  if VARIABLE(x): return UNIFY-VAR(x, y, θ)
  if VARIABLE(y): return UNIFY-VAR(y, x, θ)
  if COMPOUND(x) and COMPOUND(y) and SAME-FUNCTOR(x, y):
    return UNIFY-LIST(ARGS(x), ARGS(y), θ)
  return None

function UNIFY-VAR(var, term, θ):
  if var in θ: return UNIFY(LOOKUP(θ, var), term, θ)
  if term in θ: return UNIFY(var, LOOKUP(θ, term), θ)
  if OCCURS-CHECK(var, term): return None
  return θ ∪ {var/term}

function UNIFY-LIST(xs, ys, θ):
  if θ = None: return None
  if LEN(xs) ≠ LEN(ys): return None
  for each (x, y) in zip(xs, ys):
    θ = UNIFY(x, y, θ)
  return θ

function OCCURS-CHECK(var, term):
  if var = term: return True
  if COMPOUND(term): return any OCCURS-CHECK(var, arg) for arg in ARGS(term)
  return False

function RESOLUTION(KB, α):
  clauses = KB ∪ {¬α}          // Convert to CNF and add negated query
  new = ∅

  while true:
    for each pair (Ci, Cj) in clauses:
      resolvents = PL-RESOLVE(Ci, Cj)
      if ∅ ∈ resolvents: return true
      new = new ∪ resolvents

    if new ⊆ clauses: return false
    clauses = clauses ∪ new

function PL-RESOLVE(Ci, Cj):
  resolvents = ∅
  for each literal li in Ci:
    for each literal lj in Cj:
      if li = ¬lj or ¬li = lj:
        resolvent = (Ci - {li}) ∪ (Cj - {lj})
        resolvents = resolvents ∪ {resolvent}
  return resolvents
```

```
function FOL-RESOLUTION(KB, query):
   clauses = CNF-CONVERT(KB ∪ {¬query})
   while true:
      new = ∅
      for each pair (Ci, Cj) in clauses:
         resolvents = RESOLVE(Ci, Cj)
         if ∅ ∈ resolvents: return true
         new = new ∪ resolvents
      if new ⊆ clauses: return false
      clauses = clauses ∪ new
```

```
function ENTAILS(KB, query):
   return not SATISFIABLE(KB ∧ ¬query)

function SATISFIABLE(φ):
   return not RESOLUTION(φ, false)

function RESOLUTION(clauses):
   // clauses: KB ∧ ¬query in CNF
   new = ∅

   while true:
      for each (Ci, Cj) in clauses:
         resolvent = RESOLVE(Ci, Cj)
         if resolvent = ∅: return true  // Unsatisfiable
         new = new ∪ {resolvent}

      if new ⊆ clauses: return false     // Satisfiable
      clauses = clauses ∪ new

function RESOLVE(Ci, Cj):
   // Find complementary literals and resolve
   for literal in Ci:
      if ¬literal in Cj:
         return (Ci - {literal}) ∪ (Cj - {¬literal})
   return null
```

```
function SIMULATED-ANNEALING():
   current = RANDOM-BOARD()
   T = INITIAL-TEMP()

   while T > FINAL-TEMP():
      next = RANDOM-NEIGHBOR(current)
      ΔE = CONFLICTS(current) - CONFLICTS(next)

      if ΔE > 0 or RANDOM(0,1) < e^(ΔE/T):
         current = next

      T = COOL(T)

   return current

function RANDOM-NEIGHBOR(board):
   // Move one queen to random position in its column
   col = RANDOM(1,8)
   new_row = RANDOM(1,8) excluding current row
   return board with queen in col moved to new_row
```

```
function CONFLICTS(board):
   count = 0
   for each pair of queens (i,j):
      if same row or same diagonal: count++
   return count

function HILL-CLIMBING-N-QUEENS(n):
   current = RANDOM-STATE(n)
   while True:
      neighbors = GENERATE-NEIGHBORS(current)  // move one queen in its column
      best_neighbor = neighbor with MIN-CONFLICTS()
      if CONFLICTS(best_neighbor) ≥ CONFLICTS(current):
         return current  // local optimum
      current = best_neighbor

function CONFLICTS(state):
   count = 0
   for each pair of queens:
      if same row or diagonal: count++
   return count


function HILL-CLIMBING-8-PUZZLE(start):
   current = start
   while True:
      neighbors = GET-NEIGHBORS(current)  // all valid moves
      best_neighbor = neighbor with MIN-H(neighbor)  // heuristic value
      if H(best_neighbor) ≥ H(current):
         return current  // local minimum
      current = best_neighbor

function H(state):
   // Manhattan distance heuristic
   total = 0
   for each tile in state:
      if tile ≠ 0:
         goal_pos = GOAL-POSITION(tile)
         current_pos = CURRENT-POSITION(tile)
         total += |goal_x - current_x| + |goal_y - current_y|
   return total


function A*-SEARCH(start, goal):
   openSet = {start}
   gScore[start] = 0
   fScore[start] = HEURISTIC(start, goal)

   while openSet ≠ ∅:
      current = node in openSet with lowest fScore
      if current = goal: return RECONSTRUCT-PATH(current)

      openSet.remove(current)
      for each neighbor of current:
         tentative_g = gScore[current] + COST(current, neighbor)
         if tentative_g < gScore[neighbor]:
            // Better path found
            parent[neighbor] = current
            gScore[neighbor] = tentative_g
            fScore[neighbor] = gScore[neighbor] + HEURISTIC(neighbor, goal)
```

```
            if neighbor ∉ openSet:
                openSet.add(neighbor)

    return failure

function RECONSTRUCT-PATH(node):
    path = [node]
    while node in parent:
        node = parent[node]
        path.prepend(node)
    return path


function VACUUM-AGENT(percept):
    loc, status = percept
    if status = Dirty: return Suck
    if loc = A: return Right
    if loc = B: return Left

state = {location, dirty[A,B]}
UPDATE-STATE(percept)
if dirty[location]: return Suck
else: return MOVE-TO-OTHER()

function VACUUM-AGENT():
    cleaned = set()
    while cleaned < all rooms:
        if current room dirty:
            clean it
            add to cleaned
        else:
            add to cleaned
        if cleaned < all rooms:
            move to next dirty room


function IDDFS(start, goal):
    depth = 0
    while True:
        result = DLS(start, goal, depth, [start], visited)
        if result ≠ None: return result
        depth++

function DLS(state, goal, limit, path, visited):
    if state = goal: return path
    if limit = 0: return None

    visited.add(state)
    for each successor in GET-SUCCESSORS(state):
        if successor not in visited:
            result = DLS(successor, goal, limit-1, path+[successor], visited)
            if result ≠ None: return result
    visited.remove(state)
    return None

function GET-SUCCESSORS(state):
    blank = state.index(0)
    for move in [Up, Down, Left, Right]:
        if MOVE-VALID(blank, move):
            swap blank with adjacent tile
```

```
        return new state


function BFS(start, goal):
    queue = [start_state]
    visited = set()

    while queue not empty:
        current = dequeue(queue)
        if current.board = goal: return PATH(current)

        for each neighbor in GET-MOVES(current):
            if neighbor.board not in visited:
                mark visited
                enqueue(neighbor)
    return failure

function GET-MOVES(state):
    blank = FIND-BLANK(state.board)
    for each direction (Up, Down, Left, Right):
        if move valid:
            swap blank with adjacent tile
            return new state


function MAIN():
    initialize board[1..9] = ' '
    while not GAME-OVER():
        COMPUTER-MOVE()  // uses minimax
        if GAME-OVER(): break
        PLAYER-MOVE()

function COMPUTER-MOVE():
    bestScore = -∞
    bestMove = 0
    for each empty position in board:
        board[position] = 'X'
        score = MINIMAX(board, false)
        board[position] = ' '
        if score > bestScore:
            bestScore = score
            bestMove = position
    make move at bestMove

function MINIMAX(board, isMaximizing):
    if CHECK-WIN('X'): return 1
    if CHECK-WIN('O'): return -1
    if CHECK-DRAW(): return 0

    if isMaximizing:
        bestScore = -∞
        for each empty position:
            board[position] = 'X'
            score = MINIMAX(board, false)
            board[position] = ' '
            bestScore = max(score, bestScore)
        return bestScore
    else:
        bestScore = ∞
        for each empty position:
```

```
            board[position] = 'O'
            score = MINIMAX(board, true)
            board[position] = ' '
            bestScore = min(score, bestScore)
        return bestScore

function CHECK-WIN(player):
    check all 8 winning lines for 3 of player's marks

function CHECK-DRAW():
    return all positions filled
```