CPSC 470a – Artificial Intelligence – Fall 2017

Part 1: Classical AI (Search, CSP, and Logical Agents)

Based on slides by Prof. Dragomir Radev and
"Artificial Intelligence: A Modern Approach, 3rd Edition,"
by Russell and Norvig

James Diao

Table of Contents

Part I: Basics and Background

Chapter 1: Introduction

1. What is AI?

   Reasoning vs. Behavior, Human versus Ideal performance

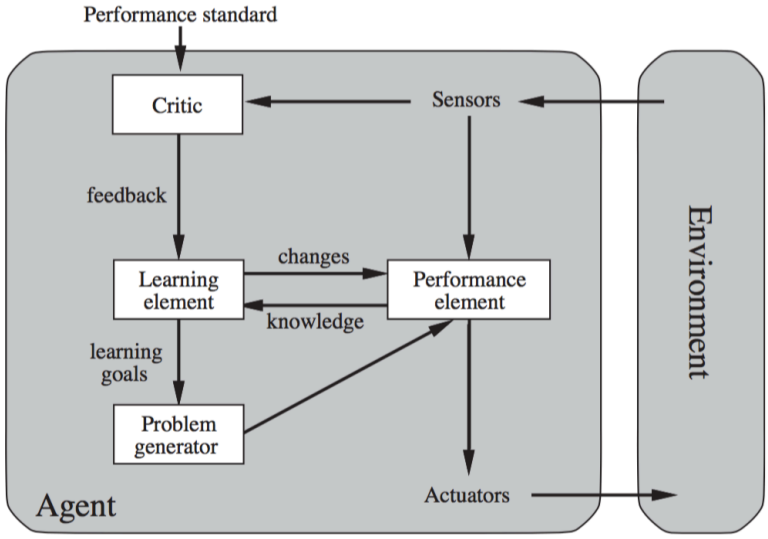
Thinking Humanly	Thinking Rationally
Acting Humanly	Acting Rationally

   a. Thinking humanly: the cognitive modeling approach
      i. Requires a precise theory of the mind from introspection, experimentation, or imagiang.
      ii. Goal: program input-output matches human behavior.
   b. Acting humanly: the Turing Test approach
      i. Requires (1) NLP, (2) knowledge representation, (3) automated reasoning, (4) machine learning.
   c. Thinking rationally: the "laws of thought" approach
      i. Syllogisms: argument structures that are always correct
      ii. Logic: governs the operations of the mind
      iii. Formal logical notation is completely solvable.
      iv. Two obstacles: informal and uncertain knowledge is hard to state in formal notation, and limited resources in practice.
   d. Acting rationally: the rational agent approach (used by book)
      i. Rational agent acts to achieve the best expected outcome.
      ii. Advantages: (1) more general: correct inference is a subset of rationality, (2) mathematically well-defined.
   e. Definitions:
      i. Intelligence: ability to perceive, understand, predict, and manipulate
      ii. Artificial intelligence: study of the design of intelligent behavior in agents
      iii. Agent: perceives environment through sensors and acts through actuators.
      iv. Rationality: maximizing some objective function outcome
2. History of AI (details skipped)
   a. Hebbian learning: early method for updating weights on neural nets.
   b. Lisp: dominant AI programming language for 30 years.
   c. Genetic algorithms / Machine evolution: small mutations + selection yields better algorithms.
   d. Perceptrons could not represent the XOR problem.
   e. Weak methods: elementary reasoning that does not scale up.
   f. Knowledge-intensive (expert) systems: have many special-purpose rules that reduce combinatorial explosion.
   g. Neats: AI theories should be based in mathematical rigor.
   h. Scruffies: AI should evolve from trying ideas and assessing what actually works.
   i. Bayesian Networks: allows rigorous reasoning with uncertain knowledge, learning from experience, etc.
   j. Normative expert systems: act according to decision theory, not human experts.
   k. Artificial general intelligence: universal algorithm for learning + acting in any environment.

Chapter 2: Intelligent Agents

1. Definitions
    a. Percepts: agent's perceptual inputs in each instant.
    b. Percepts sequence: complete history of inputs.
    c. Agent function: maps precept sequence to action. Defines the agent's behavior.
    d. Agent program: concrete implementation of agent function on a physical system. Refers to current percept, not entire sequence.
    e. Rational agent: one that selects an action that maximizes expected value, given its percept sequence and built-in knowledge.
2. Rationality, omniscience, learning, and autonomy
    a. Rational behavior is defined by a performance measure / objective function of environment states.
    b. Rationality maximizes expected performance, not actual performance (omniscience/perfection).
    c. Information gathering: actions that modify future percepts.
    d. Autonomy: compensates for partial or incorrect prior knowledge using percepts.
3. Environments
    a. Task environment:
        i. PEAS: Performance, environment, actuators, sensors
        ii. Real and artificial environments are described by complexity of relationship between PEAS.
    b. Properties of task environments
        i. Fully/partially observable
        ii. Single/multi-agent
            1. Multi-agents may be competitive or cooperative. Important elements are communication and randomized behavior.
        iii. Deterministic/stochastic (refers to environment, not other agents)
            1. Very complex or partially observable environments may appear stochastic.
            2. Stochastic implies quantifiable probabilities. Nondeterministic means possible outcomes, usually requires success for all of them.
        iv. Episodic: individual precept/action pairs
        Sequential: current decision is affected by past decisions and affects future decisions (requires planning).
        v. Static/dynamic: whether the environment changes during deliberation.
            1. Semidynamic: static environment, dynamic performance score.
        vi. Discrete/continuous (time, percepts, and actions)
        vii. Known/unknown (refers to the environment's rules, or outcomes of certain actions).
    c. Environment class: related environments that generalize certain expectations.

4. Agents
    a. Agent = architecture (physical sensors/actuators) + program (agent function)
    b. Table-driven approach is too inefficient: entries scale by sum of P^(1, 2, …, T)
    c. Simple reflex agents
        i. Markov property: actions selected based only on current precept.
        ii. Condition-action rule: if X then Y.
    d. Model-based reflex agents
        i. Internal state that reflects unobserved aspects. Does not have to reflex "what the world is like now"; could be a destination.
        ii. Models require: (1) rules about how the world evolves independently of an agent. (2) info about how the agent's actions affect the world.
    e. Goal-based agents
        i. Binary distinction: happy + unhappy states
        ii. Indirectly maps percepts to actions through a goal. Explicitly representing knowledge lends flexibility.
    f. Utility-based agents
        i. Continuous goals represented by utility function. Allows tradeoffs.
        ii. Difficulties: modeling/tracking the environment, tractable algorithms.
    g. Learning agents



        i. Learning element: makes improvements
        ii. Performance element: selects actions
        iii. Critic: gives feedback on utility function
        iv. Problem generator: suggests actions that lead to new information (for long-term optimality).
    h. Representations of states
        i. Atomic: only property is identity (difference from other states).
        ii. Factored: contains attributes (logical, real-valued, etc.)—can represent uncertainty.
        iii. Structured: contains objects, with their own attributes and relations.
        iv. (increasing complexity/expressiveness)
        v. Expressiveness: can capture, at least as concisely, everything a less expressive one can capture, plus more.

Part II: Problem-Solving

Chapter 3: Classical Search

1. Definitions
    a. Classical search: observable, deterministic, known environments; solution is sequence of actions.
    b. Goal-based agents: can be problem-solving or planning
    c. Problem-solving agents use atomic; planning agents use factored/structured.
    d. Uninformed search: given no additional info
    e. Informed search: told where to look for solutions.
2. Problem-solving agents
    a. Goal formulation: organize behavior by setting objectives and considered actions.
    b. Problem formulation: which actions and states to consider, given a goal.
    c. Search: looking for a sequence of actions that reaches the goal. Returns and executes a solution (action sequence).
    d. Open-loop system: agent ignores percepts during execution because it already knows them in advance.
3. Well-defined problems and solutions
    a. Initial state
    b. Possible actions: f (state)
    c. Successor: any state reachable from given state.
    d. Transition model: f (action, state)
    e. State space: all states reachable from the initial state. Forms a directed graph (nodes are states, links are actions).
    f. Nonnegative step costs add up to a path cost.
    g. Goal test: whether the state is a goal state.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

4. Types of Problems
    a. Abstraction: removing detail from a representation. Valid if can be expanded into the more detailed world. Useful is easier than original problem.
    b. Incremental formulation: operators augment the current state (adding a queen)
    c. Complete-state formulation: operators modify a complete state (moving queens around)
    d. Route-finding problem: optimal path through locations
    e. Touring problem: visiting every location at least once. State includes current AND visited.
    f. Traveling salesperson problem: shortest route visiting each location exactly once.
    g. VLSI layout problem: optimal positioning of chip components/connections
    h. Robot navigation: continuous route-finding with infinite state space.
    i. Automatic assembly sequencing: find an order to assemble some object. Includes protein design.
5. Searching for Solutions
    a. Search tree: initial state is root, nodes are state spaces.
    b. Expansion: applying each legal action to the current state to generate new states.
    c. Leaf node: node with no children
    d. Frontier: all leaf nodes (AKA open list)
    e. Search strategy: choosing which state to expand next.
    f. Loopy path: generates a repeated state.
    g. Redundant path: a path with a lower cost alternative.
6. Tree search and graph search
    a. Rectangular grid: depth d => 4^d leaves, but only $2d^2$ distinct states.
    b. Tree search: chooses from frontier and checks goal.
    c. Graph search: also remembers expanded nodes with the "explored set." These nodes are discarded upon generation.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

    d. Components of a node: n.STATE, n.PARENT, n.ACTION (applied to parent to generate this node), n.PATH-COST (Cumulative)
    e. Queue:
        i. FIFO (queue), LIFO (stack), priority queue (ordering function).
    f. Canonical form: captures all logically equivalent states (e.g., sorted list).
7. Measuring performance
    a. Completeness (guaranteed solution), optimality, time/space complexity.
    b. Time/space complexity are relative to the problem,
        i. Complexity for explicit graph determined by size of the state space graph (#nodes + #edges).
        ii. Complexity for implicit graph determined by **B**ranching factor (maximum successors of any node), **D**epth (of shallowest goal node), and **M**ax (length of any path in the state space).
    c. Total cost = search cost (time/memory) + path cost
8. Uninformed (blind) search strategies
    a. No additional info. All search strategies are distinguished by order of expansion.
    b. Breadth-first search
        i. Expand shallow nodes first. Goal test applied to each node upon generation. Guarantees shallowest goal node (only optimal if cost function is non-decreasing function of depth).
        ii. Uses FIFO queue. New nodes go to the back of the queue.
        iii. Time complexity is $O(b^d)$ with b leaves per node and goal depth d.
    c. Uniform-cost search
        i. Expand node with lowest path cost g(n). Goal test applied to each node before expansion. Continues to test all nodes on the final frontier. Guarantees optimality.
        ii. Uses priority queue ordered by g.
        iii. Problems
            1. Infinite loop if sequence of zero-cost actions.
            2. May explore large trees of small steps before large optimal steps
    d. Depth-first search
        i. Expands the deepest node in the current frontier. A node with no successors is dropped from the frontier, and the search "backs up" to the next deepest node. Goal test upon expansion.
        ii. Uses LIFO stack. New nodes go to the front of the queue.
        iii. Fails with infinite non-goal paths (prevented by depth limits)
        iv. Complexity may be $O(b^m)$, where m (max depth) >> d (goal depth)
        v. Advantage: space complexity of depth-first tree search: only O(bm)
            1. Stores ONLY one path from root to leaf + unexpanded siblings. Deletes node once descendants have been explored.
            2. Checks new states against those on the isolated path (prevents infinite loops in finite state spaces).
        vi. Backtracking search
            1. Only 1 successor generated at a time (each partially expanded node remembers which successor to generate next).
            2. Complexity: O(m)

e. Depth-limited search (limit = L)
    i. Prevents infinite paths, but incomplete if L < d.
    ii. Good choice: diameter of state space (max steps between cities).
    iii. Time complexity $O(B^L)$ and space complexity O(bl).
f. Iterative deepening depth-first search
    i. Gradually increases the limit until a goal is found. Succeeds when L = d.
    ii. Combines memory savings with completeness. Preferred solution when search space is large and depth is unknown.
    iii. Generates states multiple times: not too costly because upper levels are sparse compared to lower levels.
    iv. With d levels, nodes generated = $d(b) + (d-1)b^2 + \ldots + (1)b^d \Rightarrow O(b^d)$
    v. Iterative lengthening search: uses path-cost limits instead of depth limits. Incurs substantial overhead.
g. Bidirectional search
    i. Forward from initial state and backwards from goal, hoping to connect.
    ii. Advantage: smaller search space (two small circles << one large circle)
    iii. Goal test: whether frontiers intersect.
    iv. Non-optimal: there may be a short-cut across the gap.
    v. Time complexity $O(b^{d/2})$ and space complexity is also $O(b^{d/2})$. Reduced by iterative deepening on 1 end.
    vi. Weaknesses
        1. Space requirement: 1 frontier must be stored to check intersection.
        2. Requires method for computing predecessors. Not a problem for reversible actions.
        3. Requires specific goal (fails with 8 queens problem).
h. Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

9. Informed (heuristic) search strategies
    a. Definitions
        i. Informed search: uses problem-specific knowledge
        ii. Best-first search: node is selected based on evaluation function f(n). Same as g(n), but uses cost estimate, not actual cost so far.
        iii. Heuristic function h(n) = estimated cheapest path to goal.
    b. Greedy best-first search
        i. Expands the node closest to the goal: f(n) = h(n)
        ii. Worst time and space complexity is $O(b^m)$
    c. A* search: minimizing total estimated solution cost
        i. Combines two costs:
            1. Start to node: g(n), like in UCS
            2. Node to goal: h(n), not in UCS
        ii. Conditions for optimality
            1. Admissibility: never overestimates cost to goal. Optimistic.
            2. Consistency/monotonicity (like triangle inequality):
                a. For every node n and all its successors n': $h(n) \leq h(n') + \text{Cost (n to n')}$
                b. Stricter requirement than admissibility.
            3. Tree-search A* is optimal if h(n) is admissible.
               Graph-search A* is optimal if h(n) is consistent.
        iii. Claim: if h(n) is consistent, f(n) is nondecreasing along all paths.
        iv. Claim: A* finds the optimal path to every node it expands.
        v. Contours
            1. Uniform-cost search adds circular bands
            2. A* narrows the bands toward the optimal path.

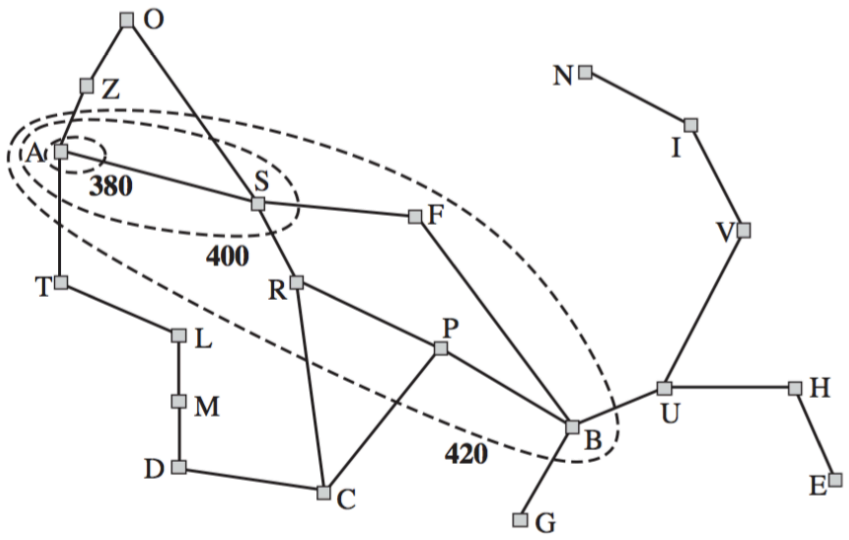


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

d. Optimality of A*
    i. A* expands all nodes with f(n) < C* (true optimal cost), otherwise, you might miss the answer. This is basically all nodes within the contour defined by C*.
    ii. Optimally efficient: expands fewest nodes.
    iii. Pruning: some subtrees are ignored while guaranteeing optimality.
e. Memory-bounded heuristic search
    i. Iterative-deepening (IDA) A*, recursive best-first search (RBFS), memory-bounded (M) A*, simplified memory-bound (SM) A*.

10. Heuristic Functions
    a. Effect on performance
        i. Effective branching factor: what you'd need to generate same number of nodes at depth d if the tree were uniform.
        ii. Better by: higher values (closer to true cost), lower computation time.
    b. Relaxed problems
        i. Relaxed problems have fewer restrictions; same graph but more nodes.
        ii. Heuristics are often true path costs in simplified problems.
        iii. Set maximum of many possible heuristics; dominates all components.
    c. Subproblems and pattern databases
        i. Store exact solution costs for all possible configurations of subproblem, and solve. Look up in the database.
        ii. Ex: heuristic of matching 1-4 only.
        iii. If you work backwards 1, 2, … N, you can compute exact solution costs by dynamic programming.
        iv. Disjoint pattern databases: solve 1-4, but only count moves that involve those tiles. This allows you to add different databases (1-4, 5-8, etc.)

Chapter 4: Non-Classical Search

1. Local Search Algorithms and Optimization
    a. Local search
        i. Evaluate and modify current states
        ii. Good when solutions matter, but not path or path cost.
        iii. Good for optimization.
        iv. Only uses single "current node" – very low memory usage and works in very large state spaces.
    b. Hill-climbing search
        i. Complete-state formulation: each state is "complete," and you optimize a heuristic cost function until you reach a goal state (as opposed to incremental formulation)
        ii. Getting stuck
            1. Local maxima: sub-optimal, but all local moves are worse.
            2. Ridge: equivalent local maxima that are not directly collected.
            3. Plateux: flat local maximum.
            4. Shoulder: flat area allowing progress.
        iii. Sideways move: allows move to node of equal value. Usually capped to prevent infinite loop

iv. Stochastic hill-climbing: randomly choose an option; weight by steepness.
v. First-choice hill-climbing: generate successors randomly until one is better. Good when a state has thousands of successors.
vi. Random restarts
    1. Trivially complete (will eventually choose goal state)
    2. Expected number of restarts to find goal is 1/P(finding in 1 try)

c. Simulated annealing
    i. Method: picks a random move. If improvement, always accept. Otherwise, accept with probability $p < 1$ (lower for worse moves, lower over time).
    ii. Escape local maxima by allowing some "bad" moves, but gradually decrease their frequency.
    iii. Complete if T (effective temperature) is decreased slowly enough.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

d. Local beam search
    i. Method: randomly initialize k states (beam width). At each iteration, generate all k^n successors and choose the k best successors.
    ii. Differs from k random restarts in parallel, because retained states can "jump" between trees.
    iii. Pitfall: k states may quickly concentrate in a small region.
    iv. Stochastic beam search: choose k states at random, with weighted probabilities. Analogous to natural selection with asexual reproduction.

e. Genetic algorithms
    i. Method: successor states are generated by combining parent states. Analogous to natural selection with sexual reproduction.
    ii. Randomly initialize k states (population). Each state (individual) is represented as a string over a finite alphabet and rated by an objective/fitness function.
    iii. Select pairs randomly (with replacement; weighted by fitness; may use culling to set a fitness rank cutoff).
    iv. Generate offspring by crossover + mutation.
        1. Crossover combines uphill tendency with random exploration and information exchange.
        2. Takes large steps in the state space at first (when the population is most diverse).
        3. Advantage: combines blocks that evolved independently.

v. Example with the 8 queen problem

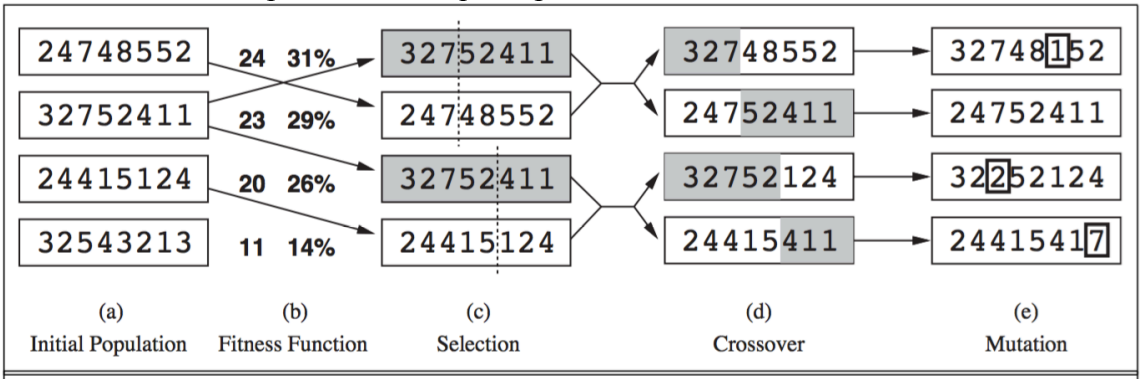


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

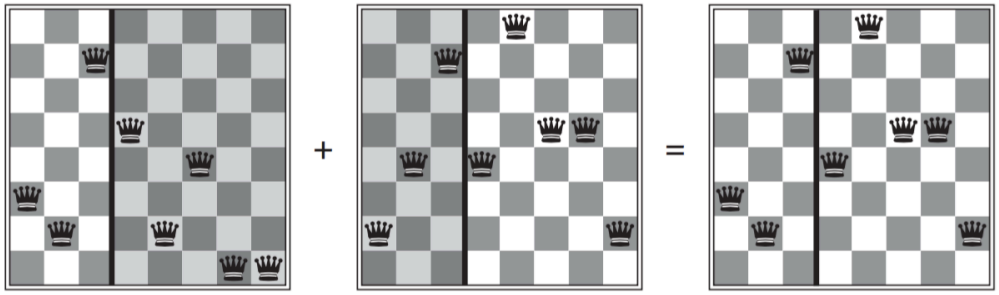


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

f. Pseudocode

```
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
    for i = 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))
```

2. Local Search in Continuous Space (skipped)
3. Searching with Nondeterministic Actions (skipped)
4. Searching with Partial Observations (skipped)
5. Online Search Agents and Unknown Environments (skipped)
    a. Involves exploration (agent is blind to the states and actions of its environments)
    b. Agent must build a map by making actions

Chapter 5: Adversarial Search

1. Adversarial Search Problems (Games)
    a. Definitions
        i. Zero-sum game: total utility across all players is always the same.
        ii. Pruning: strategically ignoring parts of the search tree.
        iii. Evaluation function: heuristic for the true utility of a state (when a complete search is impossible).
        iv. Pseudocode definitions
            1. S_0: initial state
            2. Player(s): which player has the move
            3. Actions(s) returns all legal moves
            4. Result(s,a) defines the result given a state and action.
            5. Terminal-test(s): 1 or 0 depending on whether the game has ended.
            6. Utility(s,p): numeric value for a terminal state.

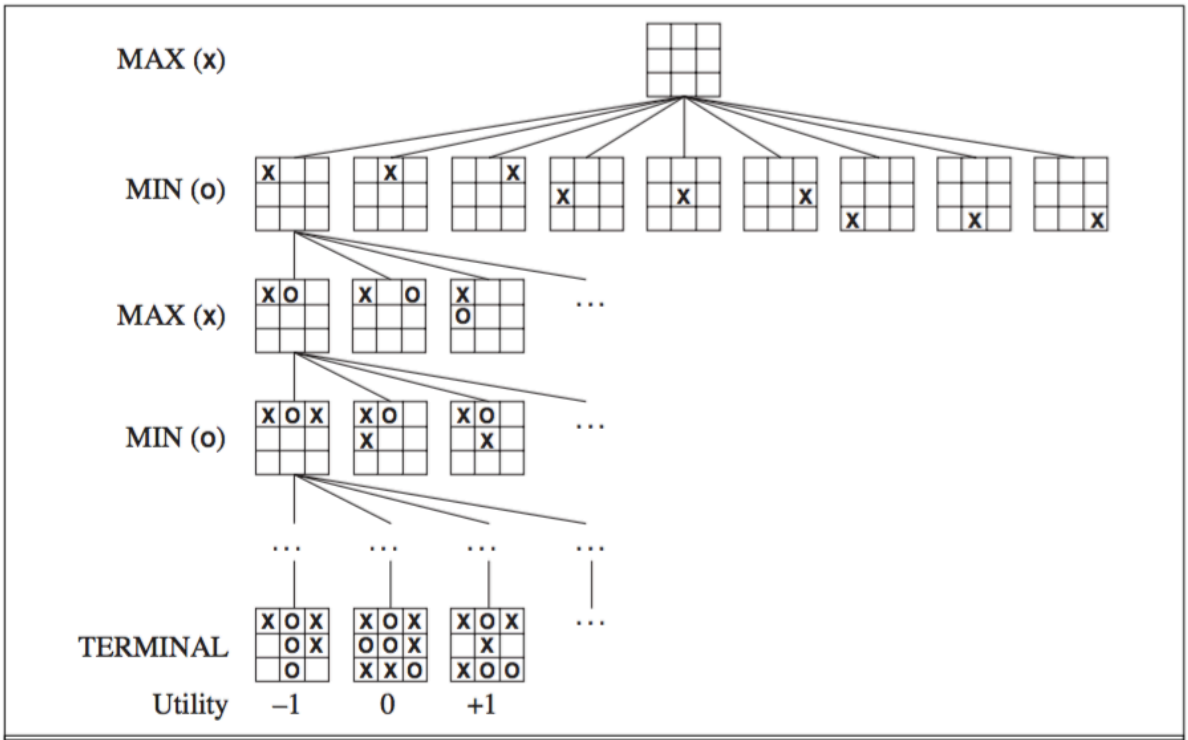


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

2. Minimax Search
    a. **Mini**mizing the **max**imum possible loss.
    b. Minimax value: best achievable payoff against optimal play.
    c. Pertains to deterministic turn-based games with 2 players and perfect knowledge.
    d. Search problem: nodes are states, edges are decisions, levels are called "plys."
    e. Pseudocode:

```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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
```

    f. Intuition: utility values "bubble up" the tree.
    g. Properties: optimal, complete (for finite tree)
    h. Complexity
        i. Time: $P(b^m)$
        ii. Space: O(bm) for DFS
        iii. For chess, b = 35 moves, m = 100 moves per position.
3. MinimaxCutoff
    a. Accounting for resource limits
        i. Horizon effect: V(n) >> 0, but V(n+1) << 0.
        ii. Cutoff test (stop searching at depth X)
        iii. Quiescence search: "peek beyond the horizon" for interesting positions (e.g., where payoff can change quickly).
            1. This is judged by heuristic. Ex: captures and checks in chess.
            2. Must be careful to prevent indefinite extension.
    b. Evaluation functions
        i. Usually +Inf for win and –Inf for loss only at terminal states.
        ii. However, with MinimaxCutoff, assign heuristic utility to ALL states.
            1. Heuristic utility usually a linear weighted sum of features, learned by ML/Monte Carlo (value of knight, bishop, etc.)
        iii. Exact evaluation functions matter—behavior is only preserved by POSITIVE and LINEAR transformations.

4. Alpha-Beta Pruning
    a. Alpha: best score via any path. Initialized to –Inf.
    b. Beta: opponent's best score via any path (still relative to you). Initialized to Inf.
    c. Method:
        i. Prune branches with value < alpha (you would never make a move where your opponent could force a worse score).
        ii. Prune branches with value > beta (your opponent would never make a move where you could force a better score).
    d. Move ordering: improves alpha-beta minimax (but not vanilla).
        i. Expand nodes with highest utility first.
        ii. Perfect ordering time complexity is $O(b^{m/2})$—DOUBLES search depth.
    e. Pseudocode

```
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
```

f. Explained alpha-beta pruning example

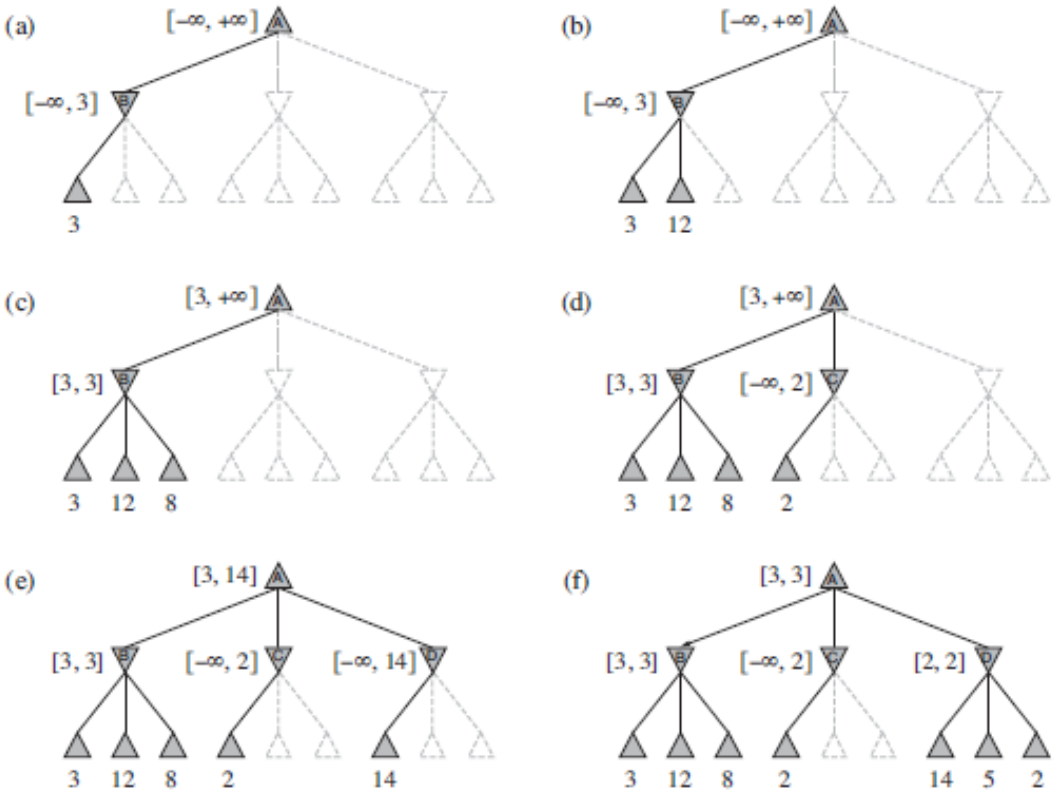


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

5. ExpectiMinimax (nondeterministic games).
   a. Exactly like minimax, but with chance nodes. These dilute the value of looking ahead because the value of each later state drops quickly.
   b. Exact same idea for imperfect information games.

Chapter 6: Constraint Satisfaction

1. Definitions
    a. Requirements: X (variables), D (domains), C (constraints); successor function, heuristic function, and goal test (satisfying the specified constraints)
    b. Constraint graph: nodes are variables, edges are constraints.
    c. Variables can be discrete or continuous; domains can be finite or infinite.
    d. Constraints can be unary, binary, or higher-order (number of variables involved).
    e. Examples: assignment, scheduling, etc.
2. State-based search: search tree
    a. Terminates at depth n (all solutions at depth n) => use depth-first.
    b. Path is irrelevant: incremental or complete-state formulations OK
3. Backtracking search (basic uninformed algorithm)
    a. DFS for CSP with single-variable assignments.
    b. Nodes are variables, NOT variable-value pairs, bc assignments are commutative.

```
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)
  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, value)
      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 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. 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.

c. Search heuristics
    i. Variable ordering
        1. Minimum remaining values (MRV) / fail-first heuristic: prunes failures faster.
        2. Degree heuristic: selects the most constraining variable (highest degree) to reduce branching factor.
    ii. Value ordering
        1. Least constraining value heuristic (fail-last): assign value that maximizes options for neighboring variables.
    iii. Pruning inevitable failures.
        1. Forward checking: tracks remaining legal values for each variable; terminates if any == 0.
        2. Constraint propagation
            a. Goal: prune local inconsistencies from subgraphs. Can be enforced before search, or after each assignment.
            b. Node consistency: every value in a variable's domain satisfies its unary constraints
            c. Arc consistency: every value in a variable's domain satisfies its binary constraints
                i. X→Y is consistent iff for every value of X there is some allowed Y
                ii. If X loses a value, its neighbors Yi must be rechecked and domain-reduced.
                iii. Time complexity: $O(n^2d^3)$

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*

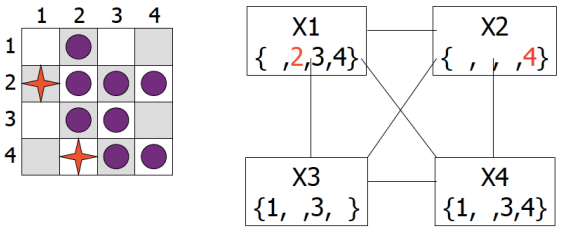
---

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

Figure 6.3 The arc-consistency algorithm AC-3. After applying AC-3, either every arc is arc-consistent, or some variable has an empty domain, indicating that the CSP cannot be solved. The name "AC-3" was used by the algorithm's inventor (Mackworth, 1977) because it's the third version developed in the paper.

4. Local search
    a. Hill-climbing, simulated annealing, etc. typically use complete-state formulation
    b. Variable selection: any conflicted
    c. Value selection: minimum conflicts heuristic

Example: 4-Queens Problem



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 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. 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.

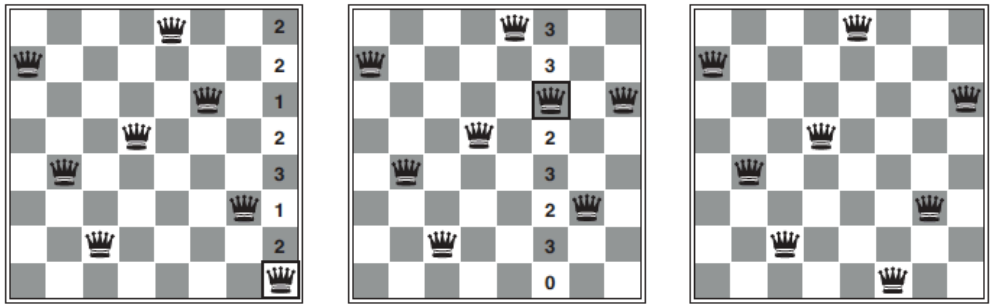
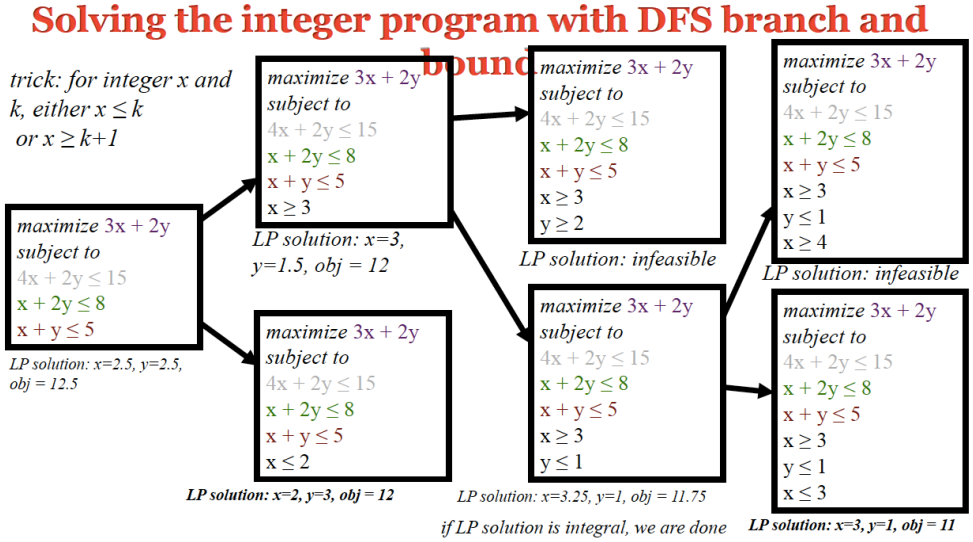


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

    d. Given random initialization, can solve most "dense-solution" CSP in constant time for arbitrary n EXCEPT in a narrow range of R = constraints/variables.

e. Beam search: BFS by heuristic cost; keeps [beam size] nodes in memory.
    i. Incomplete, suboptimal, but good for large search spaces.
f. Linear programming
    i. Maximize [linear expression] subject to [linear constraints]
    ii. Mixed and integer programs are NP-hard.
    iii. LP is solved efficiently. Provides upper bound heuristic for (M)IP.
    iv. Satisfiability problem can be reduced to IP.
    v. Solving IP
        1. Trick: 2 successors per node, defined by ($x \leq k$) or ($x \geq k+1$)
        2. Nodes branch if LP solution is not integral.
        3. Nodes terminate with (1) no solution, or (2) integral LP solution.



5. Dr. Fill
    a. Background
        i. Variables: entries of the puzzle
        ii. Domains: possible words
        iii. Neighbors: entries sharing an intersection
        iv. Constraints: neighbors must share letter in intersection
        v. Initialization: all entries to "---…----".
    b. Goal: maximize likelihood of selected words that fulfill the constraints.
    c. Procedure:

```
Given a crossword puzzle CSP, a partial solution S, best solution so far B, and
    previously pitched assignments P:
solve(CSP, S, B, P):
if S assigns every variable, return B or S, whichever is better
S' ← S with the addition of the most likely assignment not in P to an unassigned
    variable
CSP' ← propagate(CSP, S')
if propagation succeeded, B ← solve(CSP', S', B, P)
if P contains all values, return B
P' ← P with the addition of the assignment we tried in S'
B ← solve(CSP, S, B, P')
```

    d. Postprocessing: replace cases where single letters are missing with less likely but functional words.

Part III: Knowledge, Reasoning, and Planning

Chapter 7: Logical Agents

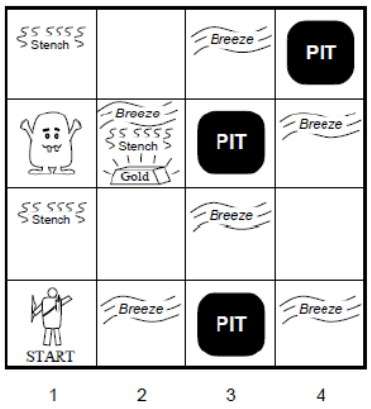
1. Knowledge-based agents
    a. Approach: reasoning on internal representations of knowledge.
    b. Components
        i. Knowledge base: domain-specific content: sentences in formal language.
        ii. Inference engine: domain-independent algorithms.
    c. Declarative approach: tells an agent what it needs to know. Query answers must follow from the KB.
    d. Agent procedure
        i. Make-percept-sentence: TELL the KB what it perceives
        ii. Make-action-query: ASK the KB which action to perform (involves inference, deduction).
        iii. Make-action sentence: TELL the KB which action was chosen
        iv. Execute the action

```
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.

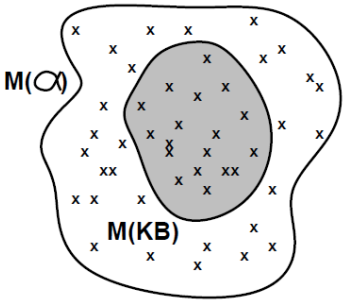
2. Example: Wumpus World
    a. PEAS
        - Performance measure
            - gold +1000, death -1000
            - -1 per step, -10 for using the arrow
        - Environment
            - Squares adjacent to wumpus are smelly
            - Squares adjacent to pit are breezy
            - Glitter iff gold is in the same square
            - Shooting kills wumpus if you are facing it
            - Shooting uses up the only arrow
            - Grabbing picks up gold if in same square
            - Releasing drops the gold in same square



        - Sensors: Stench, Breeze, Glitter, Bump, Scream
        - Actuators: Left turn, Right turn, Forward, Grab, Release, Shoot

b. Characterization: deterministic, static, discrete, single-agent, local perception (not fully observable), sequential (not episodic/Markov).
c. Problems/no safe actions
    i. Compute P(X) and U(X) to find best E[X] step.
    ii. Strategy of coercion: shoot arrow to clear a safe space.

3. Logic:
    a. Definition: formal language for representing info (to draw conclusions)
    b. Syntax: defines "well-formed" sentences in the language.
    c. Semantics: defines truth/meaning of sentences (with respect to "possible world").
    d. Model: abstraction that fixes all relevant truths/falsehoods.
        i. Ex: plugging in for variables x and y.
        ii. If sentence A is true in model M, then we say M satisfies A, or M is a model of A.
        iii. M(A) is the set of all models of A.
    e. Entailment: A entails B iff every model that satisfies A also satisfies B.
        i. A |= B iff M(A) is a subset of M(B).



        ii. Note: A is a stronger assertion because it rules out more possible worlds.
        iii. Model checking: enumerating all possible models for subsetness.
    f. Derivation
        i. KB $|-_i$ A if sentence A can be derived from KB by procedure i.
        ii. Soundness (i): KB $|-_i$ A means KB |= A. Everything derivable is entailed.
        iii. Completeness (i): KB |= A means KB $|-_i$ A. Everything entailed is derivable.

4. Studied Models
    a. State-based
        i. Uses states, actions, and costs
        ii. Ex: route-finding, game-playing
    b. Variable-based
        i. Uses variables, values, and domains
        ii. Ex: CSP, scheduling, assignment.
    c. Logic-based
        i. Logical formulas and inference rules
        ii. Ex: theorem proving, verification, reasoning.
        iii. Bad: Doesn't handle uncertainty well
        iv. Bad: Doesn't fine tune with data
        v. Good: compact expressiveness

5. Language and Logic
    a. Constituency parse tree: breaks a text into successively smaller subphrases; leaves are individual words, labeled with parts of speech (noun, verb, determiner, etc.)
    b. Dependency parse tree: all nodes are words. Children are dependent on parents; their relationship is labeled on the connecting edge.
    c. Modern NLP
        i. Vector semantics, supervised learning, etc.
6. Prepositional Logic
    a. Backus-Naur Form (BNF) Syntax (in order):
        i. ¬ (not): negation. Literal means variable with or without negation.
        ii. ∧ (and): A ∧ B is a conjunction of the conjuncts.
        iii. ∨ (or): A ∨ B is a disjunction of the disjuncts.
        iv. => (implies) premise/antecedent => conclusion/consequent. Also known as if-then statements.
        v. <=> (if and only if): biconditional.
    b. Semantics: defines truth/meaning of sentence.
        i. Understanding meaning: if an agent can hear a sentence and act accordingly.
        ii. Five rules:
            1. ¬P is true iff P is false in M.
            2. P ∧ Q is true iff P and Q are true in M.
            3. P ∨ Q is true iff P or Q is true in M.
            4. P => Q is true unless P is true and Q is false in M.
                a. True even if no causation or relevance
                b. ALWAYS true if P is false.
            5. P <=> Q is true iff P and Q are both true or both false in M.
        iii. Validity; S is true in **all** models, e.g., A=>A
            1. KB |= A iff KB=>A is valid
        iv. Satisfiability: S is true in **some** models.
        v. Unsatisfiable: S is true in **no** models.
            1. KB |= A iff KB & ¬A is unsatisfiable.
    c. Components:
        i. Variables (A, B, etc.), models (truth values), formulas (A=>B), KB
    d. Clauses
        i. Definite clause: disjunction of literals; exactly 1 is positive
            1. Show that if P-R hold, then S also holds.
            2. Implication form: P & Q & R => S
            3. Disjunctive form: ¬P | ¬Q | ¬R | S
        ii. Horn clause: more general; can have no positive literals (goal clause)
            1. Show that if P-R all hold.
            2. Implication form: P & Q & R & S => False
            3. Disjunctive form: ¬P | ¬Q | ¬R | ¬S => True
    e. Standard logical equivalences: (A|=B and B|=A)
        i. Commutativity, associativity, and distributivity of AND/OR
        ii. Implication elimination: (A=>B) == (¬A | B)

f. Truth table for inference
   i. For N variables, there are 2^N rows.
   ii. KB = R_1 & R_2 & … & R_N, where R_i are all axioms or derived claims.
   iii. Check that A is true in all rows where KB is true.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. ***KB*** is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

g. DFS enumeration algorithm
   i. Recursive enumeration of truth assignments (similar to backtracking)
   ii. Sound and complete (always right, always terminates)

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
    $P \leftarrow$ FIRST($symbols$)
    $rest \leftarrow$ 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 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

7. Proof Methods
    a. Proof: sequence of inference rule applications => desired sentence.
    b. Natural deduction: application of inference
        i. Sound derivation of new sentences
        ii. Rules
            1. Modus Ponens: (A, A=>B) |= (B)
            2. And-Elimination: (A^B) |= (B)
            3. Any logical equivalence generates 2 opposite rules.
        iii. Problem definition: initial state (KB_0), actions (inference rules), result (adding to KB), goal (state containing sentence we're trying to prove).

We start with the knowledge base containing R_1 through R_5 and show how to prove $\neg P_{1,2}$, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R_2 to obtain

$$R_6 : \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Then we apply And-Elimination to R_6 to obtain

$$R_7 : \quad ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) .$$

Logical equivalence for contrapositives gives

$$R_8 : \quad (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})) .$$

Now we can apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

$$R_9 : \quad \neg(P_{1,2} \vee P_{2,1}) .$$

Finally, we apply De Morgan's rule, giving the conclusion

$$R_{10} : \quad \neg P_{1,2} \wedge \neg P_{2,1} .$$

That is, neither [1,2] nor [2,1] contains a pit.

    c. Model checking
        i. Truth table enumeration (always exponential in n).
        ii. Improved backtracking, e.g., Davis-Putnam-Logemann-Loveland (DPLL)
        iii. Heuristic search in model space (sound but iincomplete)
            1. Min-conflicts-like hill-climbing

8. Quiz
    a. Questions

7.4 Which of the following are correct?

a. $False \models True$.
b. $True \models False$.
c. $(A \wedge B) \models (A \Leftrightarrow B)$.
d. $A \Leftrightarrow B \models A \vee B$.
e. $A \Leftrightarrow B \models \neg A \vee B$.
f. $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.
g. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.
h. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$.
i. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.
j. $(A \vee B) \wedge \neg(A \Rightarrow B)$ is satisfiable.
k. $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ is satisfiable.
l. $(A \Leftrightarrow B) \Leftrightarrow C$ has the same number of models as $(A \Leftrightarrow B)$ for any fixed set of proposition symbols that includes A, B, C.

b. Answers

7.4 In all cases, the question can be resolved easily by referring to the definition of entailment.

a. $False \models True$ is true because $False$ has no models and hence entails every sentence AND because $True$ is true in all models and hence is entailed by every sentence.

b. $True \models False$ is false.

c. $(A \wedge B) \models (A \Leftrightarrow B)$ is true because the left-hand side has exactly one model that is one of the two models of the right-hand side.

d. $A \Leftrightarrow B \models A \vee B$ is false because one of the models of $A \Leftrightarrow B$ has both A and B false, which does not satisfy $A \vee B$.

e. $A \Leftrightarrow B \models \neg A \vee B$ is true because the RHS is $A \Rightarrow B$, one of the conjuncts in the definition of $A \Leftrightarrow B$.

f. $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$ is true because the RHS is false only when both disjuncts are false, i.e., when A and B are true and C is false, in which case the LHS is also false. This may seem counterintuitive, and would not hold if $\Rightarrow$ is interpreted as "causes."

g. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$ is true; proof by truth table enumeration, or by application of distributivity (Fig 7.11).

h. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$ is true; removing a conjunct only allows more models.

i. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$ is false; removing a disjunct allows fewer models.

j. $(A \vee B) \wedge \neg(A \Rightarrow B)$ *is* satisfiable; model has A and $\neg B$.

k. $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ *is* satisfiable; RHS is entailed by LHS so models are those of $A \Leftrightarrow B$.

l. $(A \Leftrightarrow B) \Leftrightarrow C$ does have the same number of models as $(A \Leftrightarrow B)$; half the models of $(A \Leftrightarrow B)$ satisfy $(A \Leftrightarrow B) \Leftrightarrow C$, as do half the non-models, and there are the same numbers of models and non-models.

9. Quiz 2

a. Questions and Answers

7.10 Decide whether each of the following sentences is valid, unsatisfiable, or neither. Verify your decisions using truth tables or the equivalence rules of Figure 7.11 (page 249).

a. $Smoke \Rightarrow Smoke$

b. $Smoke \Rightarrow Fire$

c. $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

d. $Smoke \vee Fire \vee \neg Fire$

e. $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire))$

f. $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$

g. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

7.10

a. Valid.

b. Neither.

c. Neither.

d. Valid.

e. Valid.

f. Valid.

g. Valid.

10. Proof by resolution
    a. Sound and complete inference with any complete search algorithm.
    b. Clause: disjunction of literals.
    c. In A | B | C, the literal ¬A *resolves with* A to give the resolvent: B | C.
    d. Unit resolution inference rule: clause + complementary literal => smaller clause

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

    e. Full resolution rule: clause 1 + clause 2 => clause with all literals except any complementary literals.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

11. Conjunctive Normal Form (CNF)
    a. Every sentence of prepositional logic has a logically equivalent in CNF.
    b. Conjunction of clauses (disjunctions of literals).
    c. Ex: (A | ¬B) & (B | ¬C | ¬D)
    d. Procedure
        i. Eliminate A<=>B with (A=>B) & (B=>A)
        ii. Eliminate A=>B with ¬A | B
        iii. Move ¬ inwards with ¬¬ and De Morgan
        iv. Distribute | over & with distributivity
        v. Flatten with associativity
12. Resolution algorithm
    a. Proof by contradiction: Show that (KB & ¬A) is unsatisfiable.
    b. Procedure:
        i. Convert (KB & ¬A) to CNF
        ii. Resolve each clause pair with complementary literals and repeat.
        iii. Discard any clause with complementary literals (P | ¬P is always true).
        iv. Termination
            1. No clauses can be resolved: KB !|= A
            2. ANY pair generates an empty clause: KB |= A
                a. Empty clause == False: disjunctions are only true iff at least one disjunct is true.
                b. Empty clause only arises from resolving complementary unit clauses, like P and ¬P.
    c. Resolution closure: RC(S)
        i. Set of all clauses derivable by repeated resolution of S and its derivatives.
        ii. Finite combinations of symbols => finite RC(S) => always terminates.
    d. Ground resolution (completeness) theorem
        i. If a set of clauses (S) is unsatisfiable, then RC(S) has the empty clause.
        ii. Proof by contrapositive.

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

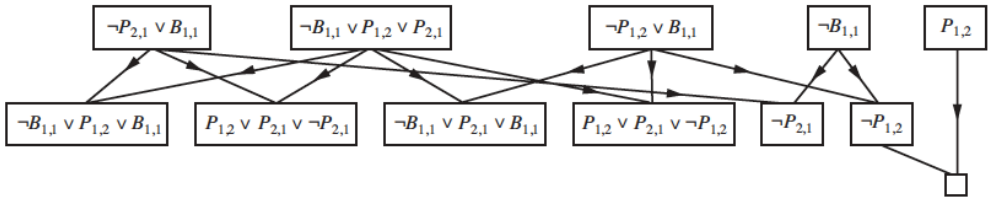


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

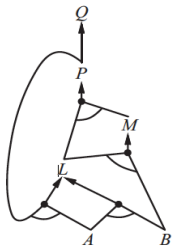
13. Grammar

$$
\begin{aligned}
\mathit{CNFSentence} &\rightarrow \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} &\rightarrow \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} &\rightarrow \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} &\rightarrow \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{aligned}
$$

Figure 7.14 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as $A \wedge B \Rightarrow C$ is still a definite clause when it is written as $\neg A \vee \neg B \vee C$, but only the former is considered the canonical form for definite clauses. One more class is the *k*-CNF sentence, which is a CNF sentence where each clause has at most *k* literals.

14. Chaining
    a. AND-OR graph
        i. Each parent is connected to its premise nodes.
        ii. Arc indicates conjunction; no arc indicates disjunction.

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



    b. Forward chaining
        i. Goal: determine if Horn KB |= query (q), where Horn KB contains definite clauses, and q is a SINGLE proposition symbol.
        ii. Properties
            1. Runs in linear time relative to size(KB)!
            2. Soundness: every inference from Modus Ponens
            3. Complete: every entailed atomic sentence is derived.
                a. The fixed point is a model of KB (no clause is false bc that would mean the premises are true but conclusion false, which only happens when FC is incomplete).
                b. Every atomic q that is entailed by KB must be true in every model of KB, including the fixed point model.
            4. Undirected reasoning: generates many irrelevant consequences.
        iii. Procedure
            1. Begin with all positive literals and propagate as far as possible.

```
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 7.15 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 $Q \Rightarrow P$.

c. Backward chaining
    i. Procedure:
        1. Check if q is true; if not, find premises in KB that => q.
        2. For each premise; for each symbol S_i, check if true. Otherwise, find premises in KB that => S_i.
        3. Early termination heuristic:
            a. Check if new subgoal is already on the goal stack
            b. Check if new subgoal has already been proven true/failed
    ii. Properties
        1. Runs in linear time relative to size(KB)!
        2. Goal-directed reasoning: more efficient; often VERY sub-linear.

15. SAT problem: checking satisfiability
    a. Previous algorithms tested entailment by testing unsatisfiability of KB & ¬A
    b. Backtracking Search
        i. DPLL (Davis-Putnam-Logemann-Loveland)
            1. Input: sentence in CNF.
            2. Procedure: recursive DFS enumeration of models.

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

            3. Improvements over TT-Entails?
                a. Early termination
                    i. Any literal is true => clause true
                    ii. Any clause is false => sentence false.
                b. Pure symbol heuristic
                    i. Pure symbol: same sign in relevant clauses.

ii. There exists a model where all pure symbols are true => easy assignment.

c. Unit clause heuristic
   i. Unit clause: has 1 non-false literal; assigns first.
   ii. Trying to prove a literal already in KB succeeds immediately.
   iii. Unit propagation: cascade of forced assignments from UCH.

c. Tricks for SAT Solvers
   i. Component analysis: clauses may be separable into disjoint components that share no unassigned variables => work on each component separately.
   ii. Variable and value ordering: ex: degree heuristic
   iii. Random restarts:
      1. Makes different choices (in variable/value selection).
      2. Retain learned clauses to prune the search space.
      3. Reduces variance on solution time (not necessarily faster).
   iv. Intelligent indexing?
   v. Intelligent backtracking?

d. Incomplete Local Search
   i. May include hill-climbing, simulated annealing, etc.
   ii. WalkSAT
      1. Procedure
         a. Initializes random T/F assignments to all symbols.
         b. Picks an unsatisfied clause and picks a symbol to flip.
         c. Chooses randomly between (1) minimizing unsatisfied clauses in new state, and (2) random step.

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

      2. Output
         a. If returns model, input is satisfiable.
         b. If returns failure: has not yet found a solution.
         c. If returns failure after infinite flips: input is unsatisfiable.
         d. Good empirical indicator of unsatisfiability, but not proof.

iii. Landscape of random SAT problems
1. Hard problems cluster near #clauses/#symbols = 4.3

e. Time-based KB must encode all propositions at EACH time point => inefficient.

16. Summary
    a. Logical agents apply inference to a KB to derive new info and make decisions

syntax: formal structure of sentences

semantics: truth of sentences wrt models

entailment: necessary truth of one sentence given another

inference: deriving sentences from other sentences

soundness: derivations produce only entailed sentences

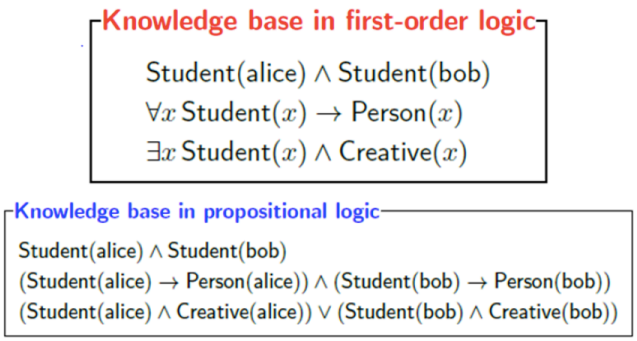
completeness: derivations can produce all entailed sentences

    b. Resolution is complete for propositional logic
    c. F/B chaining are linear time and complete for Horn clauses.
17. Features of propositional logic
    a. Pros
        i. Declarative
        ii. Allows partial/disjunctive/negated information
        iii. Compositional (A | B from A and B)
        iv. Context-independent
    b. Cons
        i. Lacks expressive power. Requires many lines for simple ideas.

Chapter 8: First-Order Logic

1. Logics
    a. Prepositional logic: world contains facts.
    b. First-order logic: world contains objects, relations, and functions.
    c. Temporal logic: FOL + times.
    d. Fuzzy logic: PL + degrees of truth.
2. Elements of FOL
    - Constants KingJohn, 2, CU,...
    - Predicates Brother, >,...
    - Functions Sqrt, LeftLegOf,...
    - Variables x, y, a, b,...
    - Connectives ¬, ⇒, ∧, ∨, ⇔
    - Equality =
    - Quantifiers ∀, ∃

    a. Predicate = relation
3. Types of sentences
    a. Atomic sentences: predicate(term1, term2) or term1 = term2
    b. Term: function(term1, term2) or constant/variable.
    c. Complex sentences: atomic sentences joined by connetives (¬, &, |, =>, <=>)
4. Models in FOL
    a. Constants: exact mapping to object model(Mary) = x1
    b. Predicates: model(CapitalOf) = ((x1, y1), (x2, y2) …)
5. FOL can be expressed as PL



6. Truth in FOL
    a. Sentences are true relative to a MODEL and INTERPRETATION.