AGENTS & ENVIRONMENTS

Agent Function: maps from percept histories to actions
Agent Program I runs on machine m to implement f
Not every agent fn can be implemented by some agent program
Facts: There exist task environments where no pure reflex is rational, there exists an environment where every agent is rational, there is a deterministic task environment where random acting agent is rational, and one agent can be rational in 2+ task environments, an agent can be perfectly rational with only partial info, an agent can be irrational in an unobservable environment.
Task Environment:
- Performance Measure: scoring
- Environment: rules & laws
- Actuators: moves
- Sensors: what's visible

Fully Observable vs. Partially Observable:
- Fully => agent can see entire state
- No sensors => environment is unobservable, need memory.

Single Agent vs. Multiagent:
- Agents: aim is to maximize performance measure whose value depends on agent's behavior
- Competitive vs. Cooperative, may need to behave randomly

Deterministic vs. Stochastic:
- Deterministic: next env determined by curr state & agent action
- Uncertain => environment is stochastic or partially observable prepare for contingencies

Episodic vs. Sequential:
- Episodic: next episode doesn't depend on previous actions

Static vs. Dynamic:
- Environment doesn't change while agent is thinking

Discrete vs. Continuous: Relates to time
Known vs. Unknown:
- Refers to agent's state of knowledge about the laws of the environment

Agent Types:
- Simple Reflex Agent: (fastest to implement, least flexible)
  - Select actions based on current percepts
- Model-Based Agent: Agent has model for how environm works
- Goal-Based Agent: Acts to attain a certain goal
- Utility-Based Agent: Maximizes utility

CONSTRAINT SATISFACTION PROBLEMS:

Backtracking Search: Move forward until something fails, step back and choose something else
- DFS with 2 ideas: 1 var at a time; check constraints as you go
- Improved with:
  - Ordering:
    - Min. Remain Vals: choose var with less legal vals, fail fast
    - Least Constraining Value: choose value that rules out fewest values in remaining variables

UNINFORMED SEARCH:

Search problem consists of: State space, Allowable actions, Transition model, Step Cost Function, Start State, Goal Test
State space size: Need to store all possible states, examples include M * N board = MN states (xy locations), M * N board with pacman pellets possibly there = MN2^(MN) states.

```
def tree-search(problem):
  frontier = [start-state]
  while True:
    if frontier is empty: return Failure
    node = frontier.pop()
    if node == goal state: return solution
    for child in node.neighbors:
      frontier.append(child)
```

DFS uses LIFO stack: (m tiers, b branching factor)
- Runtime: O(b^m); Memory: O(bm)
- Complete only if we prevent cycles
- Not optimal (finds leftmost solution regardless of depth or cost)

BFS uses queue: (s shallowest depth of solution, b branching)
- Runtime: O(b^s); Memory: O(b^s)
- Complete, optimal if costs are all 1

UCS (Dijkstra's) uses priority queue:
- Sol'n costs C*, arcs cost >= E, then effective depth is C*/E
- Runtime: O(b^(C*/E)); Memory: O(b^(C*/E))
- Compl. if sol'n has finite cost and min arc cost is +, and optim.

Complete -> guaranteed to find a solution if one exists
Optimal -> guaranteed to find least cost path

```
def graph-search(problem):
  frontier = [start-state]
  explored = []
  while True:
    if frontier is empty: return Failure
    node = frontier.pop()
    if node == goal state: return solution
    explore.append(node)
    if node not in frontier or explored set:
        for child in node.neighbors:
          frontier.append(child)
```

PROPOSITIONAL LOGIC:

Conjunction = and; ***Disjunction*** = or
P => Q === not P or Q
not P and not Q <=> not (P or Q)
not (P and Q) <=> not P or not Q
Distribution works
P and (P => Q) , infer Q by Modus Ponens
not (P => Q) === P and not B
Entailment: a |= b iff in every world where a is true, b is also true
Model-Checking: if a is true, make sure b is true too
Theorem-Proving: Search for sequence of proof steps (applications of inference rules) leading from a to b
Forward Chaining: Theorem proving algorithm

INFORMED SEARCH:

Greedy Search: Expand node seems closest to goal
A* = UCS + Greedy
A* Search: f(n) = g(n) + h(n)
Admissibility: Optimism
- Often solutions to relaxed problems
- Admissible heuristics tend to be consistent, relaxed probs

Consistent: Triangle Inequality, consistency
→ admissibility

Heuristics:
- Max of admissible heuristics is admissible and dominates both

Optimality:
- Tree A* optimal if heuristic admissible
- Graph A* optimal if heuristic is consistent

LOCAL SEARCH AND AGENTS:

```
def hill-climbing(problem):
  current = start-state
  while True:
  neighbor = highest valued successor of current
    if neighbor.value <= current.value: return current.state
    current = neighbor
```

```
def simulated-annealing(problem, schedule):
  current = start-state
  for t in range(inf):
    T = schedule(t)
    if T=0: return current
    next = random successor of current
    delta_E = next.value - current.value
    if delta_E > 0: current = next
    else: current = next (only with prob. e^(delta_E/T)
```

Local beam search:
- K copies of local search algorithm, initialized randomly
- Searches communicate (like evolution)

Nondeterminism: actions are unpredictable (need contingency plan)
Partial observability: have belief state
And-Or Search:
- Call Or-Search on root node (you decide next move)
- Call And-Search on children (nature's decision)

```
def minimax(s):
  return a in Action(s) with highest min-value(Result(s,a))
def max-value(s):
  if Terminal-Test(s): return Utility(s)
  initialize v = -inf
  for a in Action(s):
    v = max(v, min-value(Result(s,a)))
  return v
def min-value(s):
  if Terminal-Test(s): return Utility(s)
  initialize v = inf
  for a in Action(s):
    v = min(v, max-value(Result(s,a)))
  return v
```