AGENTS & ENVIRONMENTS

Agent Function: maps from percept histories to actions
Agent Program l runs on machine m to implement f
Not every agent fn can be implemented by some agent program
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

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

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

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
  - Filtering:
    - Forward Checking: When assigning a variable, remove from the domain of the remaining variables values that now violate the constraints

Min-Conflicts Algorithm:
- Randomly select a conflicted var and minimize its conflicts

Arc Consistency: X -> Y consistent iff for every x in tail there is some y in head which could be assigned w/o violating a constraint
Discrete Variables: n variables with domain size d → $O(d^n)$ complete assignments
Unary constraint: involves single variable
Tree-Structured CSPs solvable in $O(n*d^2)$

UNINFORMED SEARCH:

Search problem consists of: State space, Allowable actions, Transition model, Step Cost Function, Start State, Goal Test

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
- Uses Modus Ponens, start with implication and infer conclusion

Satisfiability: Satisfiable if sentence is true in at least one world
DPLL SAT Solver:
- Early termination: all clauses satisfied or any clause is falsified
- Pure literals: all occurrences of symbol have same sign, give symbol that value
- Unit clauses: if clause have 1 literal, set symbol to satisfy clause

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
```

Alpha-Beta Pruning:
- Perfect ordering drops time complexity to O(b^(m/2))

PROBABILITY:
Maximize Expected Utility: a* = max(SUM(P(s|a)*U(s)))
Joint Distribution: specifies distribution over a set of random variables
Marginal Distributions: sub-tables which eliminate variables by summing them out
Conditional Distributions: Prob. distr. over some variables given fixed values of others
Probabilistic Inference: compute probability from other known probabilities
Product Rule: P(y) P(x | y) = P(x, y)
Chain Rule: $P(x_1, x_2, \dots x_n) = \Pi_i (P(x_i \mid x_1, \dots x_{i-1})) \rightarrow$ Ex. $P(x_1, x_2, x_3) = P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)$
Bayes Rule: P(x|y) = P(y|x)/P(y)*P(x)

MARKOV DECISION PROCESS:
Defined by:
set of states s in S, set of actions a in A, transition model T(s,a,s'), reward function R(s,a,s'), start state, terminal state

Q - LEARNING:
$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \gamma \max_a Q(s',a)]$

BAYES NETS:
Bayes Nets: express conditional independence relationships
Independence: P(x,y) = P(x)*P(y) and P(x|y) = P(x)
Conditional Independence: P(x|y,z) = P(x|z) and P(x,y|z) = P(x|z)*P(y|z)
Full joint distribution has $O(d^n)$ [d=domain size, n=num.variables]
Bayes net has size $O(n*d^k)$ [k =max num parents]
$P(x_1,x_2,...,x_n)$ = PROD ($P(x_i$ | Parents(x_i)))
Every variable conditionally indep. of non-descendants given its parents
Markov Blanket: parents, children, and children's parents
Every variable conditionally indep. of all other variables given its Markov blanket

PERCEPTRONS:
Learning Rule: $w \leftarrow w + \alpha(y - h_w(x))x$
Convergence:
- Separable → convergence
- Non-separable → converges to min-error sol'n provided α is decayed appropriately

LAPLACE SMOOTHING:
Different from **Maximum Likelihood** which gives probabilities based only on samples
Purpose is to have probabilities for all values in domain, when only having drawn some portion of that sample size
Draws all probabilities closer to uniform distribution
Adds "fake" samples
$P(A=a_1)$ = (count of a_1+ k) / (total samples drawn + domain of A * k)

EXACT INFERENCE:
Polytree: directed graph with no undirected cycles
Enumeration is exponential. Variable elimination is worst-case exponential, but usually faster in practice.
Variable elimination in polytree is linear in network size if you eliminate from leaf toward root

APPROXIMATE INFERENCE:
Prior sampling: sampling in topological order (parents first)
Rejection sampling: count all outcomes but reject samples not consistent with evidence
Likelihood weighting: fix evidence variables, sample the rest. weight each sample by probability of evidence variables given parents
Gibbs sampling:

MARKOV MODELS:
$(x_0) \rightarrow (x_1) \rightarrow (x_2) \rightarrow \dots \rightarrow (x_t)$
Transition model: $P(x_t \mid x_{t-1})$
Stationary assumption: transition probabilities the same at all times
Markov assumption: x_t independent of $x_o, \dots, x_{t-2}$ given x_{t-1}
Join distribution: $P(x_0, \dots, x_t) = P(x_0)$ PROD($P(x_t|x_{t-1})$)
$P_{inf} = P_{inf+1} = T^T P_{inf}$; P_{inf} = [p, p-1]

HIDDEN MARKOV MODEL:
Like Markov, but we observe evidence which is pointed to by each node x
Initial Distribution: $P(x_0)$
Transition Model: $P(x_t \mid x_{t-1})$
Sensor Model: $P(E_t \mid x_t)$
Observe evidence E_t, must guess x_t

DECISION NETWORKS:
[] Action Node fixed value, <> Utility Node depends on action and chance, () Chance Node