CS188 Midterm Cheat Sheet

simonxie2004.github.io

Lec1: Introduction

Lec2: Uninformed Search

1. Reflex Agents V.S. Planning Agents:
   1. Reflex Agents: Consider how the world IS
   2. Planning Agents: Consider how the world WOULD BE
2. Properties of Agents
   1. Completeness: Guaranteed to find a solution if one exists.
   2. Optimality: Guaranteed to find the least cost path.
3. Definition of Search Problem:
   `State Space`, `Successor Function`, `Start State` & `Goal Test`
4. Definition of State Space: World State & Search State
5. State Space Graph: Nodes = states, Arcs = successors (action results)
6. Tree Search
   1. Main Idea: Expand out potential nodes; Maintain a fringe of partial plans under consideration; Expand less nodes.
   2. Key notions: Expansion, Expansion Strategy, Fringe
   3. Common tree search patterns
      (Suppose b = branching factor, m = tree depth.)
      Nodes in search tree? $\sum_{i=0}^{m} b^i = O(b^m)$
      (For BFS, suppose s = depth of shallowest solution)
      (For Uniform Cost Search, suppose solution costs C*, min(arc_cost) = eps
   4. Special Idea: Iterative Deepening
      Run DFS(depth_limit=1), DFS(depth_limit=2), ...
   5. Example Problem: Pancake flipping; Cost: Number of pancakes flipped
7. Graph Search
   1. Idea: never expand a state twice
   2. Method: record set of expanded states where elements = (state, cost).
      If a node popped from queue is NOT visited, visit it.
      If a node popped from queue is visited, check its cost.
      If the cost if lower, expand it. Else skip it.

	Strategy	Fringe	Time	Memory	Completeness	Optimality
DFS	Expand deepest node first	LIFO Stack	$O(b^m)$	$O(bm)$	True (if no cycles)	False
BFS	Expand shallowest node first	FIFO Queue	$O(b^s)$	$O(b^s)$	True	True (if cost=1)
UCS	Expand cheapest node first	Priority Queue (p=cumulative cost)	$O(C^*/\epsilon)$	$O(b^{C^*/\epsilon})$	True	True

Lec3: Informed Search

1. Definition of heuristic:
   Function that estimates how close a state is to a goal; Problem specific!
2. Example heuristics: (Relaxed-problem heuristic)
   3. Pancake flipping: heuristic = the number of largest pancake that is still out of place
   4. Dot-Eating Pacman: heuristic = the sum of all weights in a MST (of dots & current coordinate)
   5. Classic 8 Puzzle: heuristic = number of tiles misplaced
   6. Easy 8 Puzzle (allow tile to be piled intermediately): heuristic = total Manhattan distance
3. Remark: Can't use actual cost as heuristic, since have to solve that first!
4. Comparison of algorithms:
   1. Greedy Search: expand closest node (to goal);
      Orders by forward cost h(n); suboptimal
   2. UCS: expand closest node (to start state);
      Orders by backward cost g(n); suboptimal
   3. A* Search: orders by sum f(n) = g(n) + h(n)
5. A* Search
   1. When to stop: Only if we dequeue a goal
   2. Admissible (optimistic) heuristic: $\forall n, 0 \leq h(n) \leq h^*(n)$.
      A* Tree Search is optimal if heuristic is admissible.
      Proof: Suppose A is optimal, B is suboptimal. B is on fringe.
      Claim: n will be expanded first. Because f(n) = g(n) + h(n) < f(A) < f(B)
   3. Consistent heuristic: $\forall A, B, h(A) - h(B) \leq cost(A, B)$
      A* Graph Search is optimal if heuristic is consistent.

6.Semi-Lattice of Heuristics
   1.Dominance: define $h_a \geq h_c$ if $\forall n, h_a(n) \geq h_c(n)$
   2.Heuristics form semi-lattice because: $\forall h(n) = max(h_a(n), h_b(n)) \in H$
   3.Bottom of lattice is zero-heuristic. Top of lattice is exact-heuristic

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

```
function A*-GRAPH-SEARCH(problem, frontier) return a solution or failure
    reached ← an empty dict mapping nodes to the cost to each one
    frontier← INSERT((MAKE-NODE(INITIAL-STATE[problem]),0), frontier)
    while not IS-EMPTY(frontier) do
        node, node.CostToNode ← POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        end if
        if node.STATE is not in reached or reached[node.STATE] > node.CostToNode then
            reached[node.STATE] = node.CostToNode
            for each child-node in EXPAND(problem, node) do
                frontier ← INSERT((child-node, child-node.COST + CostToNode), frontier)
            end for
        end if
    end while
    return failure
```

Lec4-5: CSP Problems

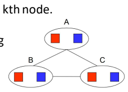
1. Definition of CSP Problems: (A special subset of search problems)
   1. State: Varibles {Xi}, with values from domain D
   2. Goal Test: set of constraints specifying allowable combinations of values
2. Example of CSP Problems:
   1. N-Queens $\forall i, j, k, (X_{ij}, X_{jk}) \neq (1,1), \cdots$ and $\sum_{i,j} X_{ij} = N$
      Formulation 1: Variables: Xij, Domains: {0, 1}, Constraints:
      Formulation 2: Variables Qk, Domains: {1, ..., N}, Constraints: $\forall (i, j), \text{non-threatening}(Q_i, Q_j)$
   2. Cryptarithmetic
3. Constraint Graph:
   1. Circle nodes = Variables; Rectangular nodes = Constraints.
   2. If there is a relation between some variables,
      They are connected to a constraint node.
4. Simple Backtracking Search
   1. One variable at a time
   2. Check constraints as you go. (Only consider constraints not conflicting to previous assignments)
5. Simple Backtracking Algorithm = DFS + variable-ordering + fail-on-violation

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

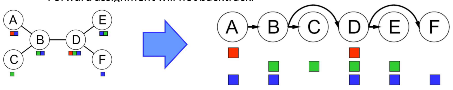
6. Filtering & Arc Consistency
   1. Definition: Arc X->Y is consistent iff $\forall x \in X, \exists y \in Y$ that could be assigned. (Basically X is enforcing constraints on Y)
   2. Filtering: Forward Checking: Enforcing consistency of arcs pointing to each new assignment
   3. Filtering: Constraint Propagation: If X loses a Value, neighbors of X need to be rechecked.
   4. Usage: run arc consistency as a preprocessor or after each assignment
   5. Algorithm with Runtime O(n^2d^3)

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
    local variables: queue, a queue of arcs, initially all the arcs in csp

    while queue is not empty do
        (Xi, Xj) ← REMOVE-FIRST(queue)
        if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
            for each Xk in NEIGHBORS[Xi] do
                add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff succeeds
    removed ← false
    for each x in DOMAIN[Xi] do
        if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint Xi ↔ Xj
            then delete x from DOMAIN[Xi]; removed ← true
    return removed
```

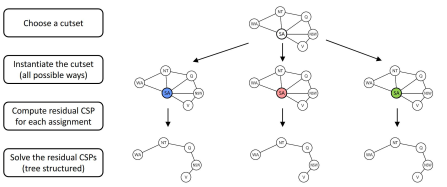
7. Advanced Definition: K-Consistency
   1. K-Consistency: For each k nodes,
      Any consistent assignment to k-1 nodes can be extended to kth node.
   2. Strong K-Consistency: also k-1, k-2, ..., 1-Consistent;
      Can be solved immediately without searching / backtracking
   3. Problems of Arc-consistency: only considers 2-consistency
   4. Example of being NOT 3-consistent:



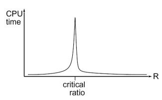
8. Advanced Arc-Consistency: Ordering
   1. Variable Ordering: MRV (Minimum Remaining Value):
      Choose the variable with fewest legal left values in domain
   2. Value Ordering: LCV (Least Constraining Value):
      Choose the value that rules out fewest values in remaining variables.
      (May require re-running filtering.)
9. Advanced Arc-Consistency: Observing Problem Structure
   1. Suppose graph of n variables can be broken into subproblems with c variables: Can solve in O(n/c * d^c)
   2. Suppose graph is a tree: Can solve in O(nd^2). Method as follows
      1. Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(Xi),Xi)
      2. Assign forward: For i = 1 : n, assign Xi consistently with Parent(Xi)
      3. *Remark: After backward pass, all root-to-leaf are consistent.
         Forward assignment will not backtrack.



10. Advanced Arc-Consistency: Improving Problem Structure
    1. Idea: Initiate a variable and prune its neighbors' domains.
    2. Method: instantiate a set of vars
       such that remaining constraint graph is a tree (cutset conditioning)
    3. Runtime: O(d^c * (n-c)d^2) to solve CSP.



11. Iterative Methods:
    1. Local Search
       1. Algorithm: While not solved, randomly select any conflicted variable.
          Assign value by min-conflicts heuristic.
       2. Performance: can solve n-queens in almost constant time for arbitrary n with high probability, except a few of them.



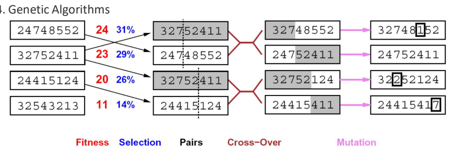
    2. Hill Climbing

```
function HILL-CLIMBING(problem) returns a state
    current ← make-node(problem.initial-state)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.value ≤ current.value then
            return current.state
        current ← neighbor
```

    3. Hill Climbing
       Remark: Stationary distribution: p(x) propto e^(E(x)/kT)

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

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

4. Genetic Algorithms



Lec6: Game Trees (MiniMax)

1. Zero-Sum Games V.S. General Games: Opposite utilities v.s. Independent utilities
   1. Examples of Zero-Sum Games: Tic-tac-toe, chess, checkers, ...
2. Value of State: Best achievable outcome (utility) from that state.
   1. For MAX players $max_{s' \in \text{children}(s)} V(s')$
      For MIN players, min...
3. Search Strategy: Minimax

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

4. Minimax properties:
   1. Optimal against perfect player. Sub-optimal otherwise.
   2. Time: O(b^m), Space: O(bm)
5. Alpha-Beta Pruning
   1. Algorithm:
      α: MAX's best option on path to root
      β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v

def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

   2. Properties:
      1. Meaning of Alpha: maximum reward for MAX players, best option so far for MAX player
      2. Meaning of Beta: minimum loss for MIN players, best option so far for MIN player
      3. Have no effect on root value; intermediate values might be wrong.
      4. With perfect ordering, time complexity drops to O(b^(m/2))
6. Depth-Limited Minimax: replace terminal utilities with an evaluation function for non-terminate positions
   1. Evaluation Functions: weighted sum of features observed
7. Iterative Deepening: run minimax with depth_limit = 1, 2, 3, ... until timeout

Lec7: Game Trees (Expectimax, Utilities)

1. Expetimax Algorithm:

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

2. Assumptions vs Reality:
   Rational & Irrational Agents

	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

3. Axioms of Rationality

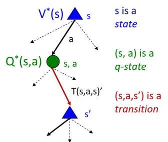
Orderability
$(A \succ B) \vee (B \succ A) \vee (A \sim B)$
Transitivity
$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$
Continuity
$A \succ B \succ C \Rightarrow \exists p\ [p, A;\ 1-p, C] \sim B$
Substitutability
$A \sim B \Rightarrow [p, A;\ 1-p, C] \sim [p, B; 1-p, C]$
Monotonicity
$A \succ B \Rightarrow$
$(p \geq q \Leftrightarrow [p, A;\ 1-p, B] \succeq [q, A;\ 1-q, B])$

4. MEU Principle
   Given any preferences satisfying these constraints,
   there exists a real valued function U s.t.:
   $$U(A) \geq U(B) \ \Leftrightarrow\ A \succeq B$$
   $$U([p_1, S_1;\ \ldots\ ;\ p_n, S_n]) = \sum_i p_i U(S_i)$$
5. Risk-adverse v.s. Risk-prone
   1. Def. L = [p, X, 1-p, Y]
   2. If U(L) < U(EMV(L)), risk-adverse
      Where U(L) = pU(X) + (1-p)U(Y), U(EMV(L)) = U(pX + (1-p)Y)
      i.e. if U is concave, like y=log2x, then risk-adverse
   3. Otherwise, risk-prone.
      i.e. if U is convex, like y=x^2, then risk-prone

Lec8-9: Markov Decision Process

1. MDP World: Noisy movement, maze-like problem, receives rewards.
   1. "Markov": Successor only depends on current state (not the history)
2. MDP World Definition:
   1. States, Actions
   2. Transition Function T(s, a, s') or Pr(s' | s, a), Reward Function R(s, a, s')
   3. Start State, (Probably) Terminal State
3. MDP Target: optimal policy pi*: S -> A

4. Discounting: Earlier is Better! No infinity rewards!
   $U([r_0, \cdots, r_{\inf}]) = \sum_{t=0}^{\inf} \gamma^t r_t = \leq R_{\max}/(1-\gamma)$



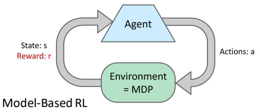
5. MDP Search Trees:
   1. Value of State: expected utility starting in s and acting optimally.
      $V^*(s) = \max_a Q^*(s,a)$
   2. Value of Q-State: expected utility starting out having taken action a from state s and (thereafter) acting optimally.
      $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma V^*(s'))]$
   3. Optimal Policy: $\pi^*(s)$
6. Solving MDP Equations: Value Iteration
   1. Bellman Equation: $V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V(s')]$
   2. Value Calculation: $V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$
   3. Policy Extraction: $\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$
   4. Complexity (of each iteration): O(S^2A)
   5. Must converge to optimal values. Policy may converge much earlier.
7. Solving MDP Equations: Q-Value Iteration
   1. Bellman Equation: $Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s' + \gamma \max_{a'} Q^*(s',a'))]$
   2. Policy Extraction: $\pi^*(s) = \arg\max_a Q^*(s,a)$
8. MDP Policy Evalutaion: Evaluating V for fixed policy
   1. Idea 1: remove the max'es from Bellman, iterating
      $V_{k+1}(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k(s')]$
   2. Idea 2: is a linear system. Use a linear system solver.
9. Solving MDP Equations: Policy Iteration
   1. Idea: Update Policy & Value meanwhile, much much faster!
   2. Algorithm:
   - Initialize $\pi_0(s) = some\ default\ action$ for all s
   - for i of policy iteration:

     Policy evaluation:
     - Initialize $V_0^{\pi_i}(s) = 0$ for all s
     - for k of policy evaluation:
       - $V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s,\pi_i(s),s') \left[R(s,\pi_i(s),s') + \gamma V_k^{\pi_i}(s')\right]$

     Policy improvement:
     - $\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_i}(s')\right]$

Lec 10-11: Reinforcement Learning

1. Intuition: Suppose we know nothing about the world. Don't know T or R.

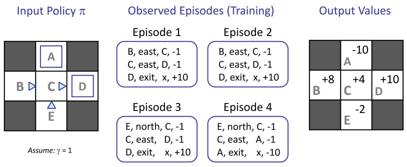


2. Passive RL I: Model-Based RL
   1. Count outcomes s' for each s, a; Record R;
   2. Calculate MDP through any iteration
   3. Run policy. If not satisfied, add data and goto step 1
3. Passive RL II: Model-Free RL (Direct Evaluation, Sample-Based Bellman Updates)
   1. Intuition: Direct evaluation from samples.
      Improve our estimate of V by computing averages of samples.
   2. Input: fixed policy pi(s)
   3. Act according to pi. Each time we visit a state,
      write down what the sum of discounted rewards turned out to be.
   4. Average the samples, we get estimate of V(s)

$$sample_1 = R(s,\pi(s),s_1') + \gamma V_k^{\pi}(s_1')$$
$$sample_2 = R(s,\pi(s),s_2') + \gamma V_k^{\pi}(s_2')$$
$$\dots$$
$$sample_n = R(s,\pi(s),s_n') + \gamma V_k^{\pi}(s_n')$$

$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n}\sum_i sample_i$$

   5. Problem: wastes information about state connections.
      Each state learned separately. Takes long time.



4. Passive RL II: Model-Free RL (Temporal Difference Learning)
   1. Intuition: learn from everywhere / every experience.
      Recent samples are more important. $\hat{x}_n = (1-a)x_{n-1} + ax_n$

$$\bar{x}_n = \frac{x_n + (1-\alpha)\cdot x_{n-1} + (1-\alpha)^2 \cdot x_{n-2} + \dots}{1 + (1-\alpha) + (1-\alpha)^2 + \dots}$$

   2. Update:

      Sample of V(s): $sample = R(s,\pi(s),s') + \gamma V^{\pi}(s')$

      Update to V(s): $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$

      Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$
   3. Decreasing learning rate (alpha) converges
   4. Problem: Can't do policy extraction, can't calculate Q(s, a) without T or R
   5. Idea: learn Q-values directly! Make action selection model-free too!
5. Passive RL III: Model-Free RL (Q-Learning + Time Difference Learning)
   1. Intuition: Learn as you go.
   2. Update:
      1. Receive a sample (s, a, s', r)
      2. Let sample = $R(s,a,s') + \gamma \max_{a'} Q(s',a')$
      3. Incorporate new estimate into a running average:
         $Q(s,a) \leftarrow (1-a)Q(s,a) + a \cdot \text{sample}$
      4. Another representation: $Q(s,a) \leftarrow Q(s,a) + a \cdot \text{Difference}$
         where diff = sample - orig
   3. This is off-policy learning!
6. Active RL: How to act to collect data
   1. Exploration schemes: eps-greedy
      1. With probability eps, act randomly from all options
      2. With probability 1 – eps, act on current policy
   2. Exploration functions: use an optimistic utility instead of real utility
      1. Def. optimistic utility $f(u,n) = u + k/n$
         suppose u = value estimate, n = visit count
      2. Modified Q-Update: $Q(s,a) \leftarrow_a R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$
3. Measuring total mistake cost: sum of difference between expected rewards and suboptimality rewards.

7. Scaling up RL: Approximate Q Learning
   1. State space too large & sparse?
      Use linear functions to approximately learn Q(s,a) or V(s)
   2. Definition: $Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \dots$
   3. Q-learning with linear Q-fuctions:
      Transition := (s, a, r, s')
      Difference := $[r + \gamma \max_{a'} Q(s',a')] - Q(s,a)$
      Approx. Update weight: $w_i \leftarrow w_i + a \cdot \text{Difference} \cdot f_i(s,a)$