CS3243 Cheatsheet

1. Introduction

1.1 Intelligent Agents

Agents interact with their environment

- Sensors take in percepts
- Actuators perform actions
- Agent function maps *percept histories* to *actions*: $f : P^* \to A$

1.2 Rationality

Rational if selected actions are:

- Based on evidence (prior knowledge/percept sequence)
- Maximise performance measure

Performance measure: how to define/measure?

- Task specificity: easier to define 'performance' for a narrower than more general task

Can be rational to explore (perform actions that gather information)

Agent is *autonomous* if behaviour is determined by its own experience

1.3 Task Environment: PEAS

PEAS: Performance measure, Environment, Actuators, Sensors

E.g. Automated Taxi

- Performance measure: safe, fast, revenue
- Environment: roads, traffic, pedestrians
- Actuators: steering wheel, accelerator, brake
- Sensors: sonar, speedometer, gps, engine sensors

1.4 Properties of Task Environments

- Observability: full or partial?
- Deterministic vs. stochastic: random elements
  - Still deterministic if random elements do not affect the transition function
  - Not deterministic if some elements are unobservable to player
- Episodic vs. sequential
  - Episodic: choice of current action does not depend on actions in past episodes
  - Sequential: need to consider previous actions (e.g. chess); current action affects future ones
  - *Order* is important in sequential, not episodic
- Static vs. dynamic: is environment changing as agent deliberates?
- Discrete vs. continuous: finite/infinite number of distinct states/percepts/actions
- Single vs. multi agent

1.5 Building an Agent

Lookup table agent

- For each possible percept, give optimal action
- Problem: table is huge with too many percepts
- Problem: no autonomy, hard to change on-the-fly if action is wrong. Unmaintainable and rigid

Types of Agents (increasing complexity)

1. Simple reflex agent: passive, only acts when it observes a percept
2. Model-based reflex agent: passive, has state/internal model of the world
3. Goal-based agent: not just passive and based on percept; has goals and acts to achieve them
4. Utility-based agent: has utility function, acts to maximise it

State is updated based on percept, current state, most recent action, model of the world

(*) Utility function is *internal*, performance measure is *external* and used to assess agent

Learning agent: has critic + learner, adapts based on performance standard

Explore vs. Exploit: trade-off the agent must make

- Explore: get knowledge to improve future gains
- Exploit: use knowledge to max current gains

2. Uninformed Search

Problem-solving agent: a goal-based agent

Environment: fully observable, deterministic, discrete

Uninformed search: no additional knowledge incorporated

2.1 Search Problem Formulation

- State: including initial state
  - Abstract ONLY relevant information, and nothing else; everything in the state should be a variable that can change, no constants
  - Everything in the state should be a variable that can change, no constants
- Actions: $\textsc{Actions}(s)$ gives set of all valid actions that can be executed in state s
  - Define it for every possible state s
- Transition model: $\textsc{Result}(s, a)$ gives new state s' upon doing action a in state s
  - Define it for every possible state s and its valid action a
- Goal test: test if a state s is the goal state
  - E.g. $IsCheckmate(s)$ or $IsSolved(s)$
- Path cost: path cost is additive sum of step costs
  - Step cost $c(s, a, s')$ — e.g. 1 per action taken

2.2 Searching for Solutions

Solution: sequence of actions leading from initial to goal state

Example: 8-puzzle

- State: an arrangement of numbers in 3x3 grid, represented as matrix/array
- Actions: moving one filled square to a blank adjacent square
- Transition model: [depends on representation] — function that takes in state + action => new state
- Goal test: whether each cell matches the goal state, one-for-one
- Cost function: uniform cost of 1 for each action

State vs Node

- State: represents physical configuration
- Node: data structure constituting part of search tree: includes state, parent node, action, path cost $g(n)$
- Two different nodes can contain same world state

2.3 Search Strategies

Which order should we expand the nodes in?

Evaluation criteria

- Completeness: always find a solution if it exists
- Optimality: finds a least-cost solution
- Time complexity: number nodes generated
- Space complexity: max # nodes in memory

Problem parameters

- b: maximum # of successors for each node — branching factor
- d: depth of *shallowest* goal node
- m: maximum depth of search tree

2.4 Breadth-First Search (BFS)

Frontier: Queue

- Complete: yes, as long as b is finite
- Optimal: no, unless uniform step cost, or uniform across each level
- Time: $O(b^d) = O(b) + O(b^2) + \ldots + O(b^d)$
- Space: $O(b^d)$ (max size of frontier)

Applies goal test when pushing to frontier: reduces time and space complexity from $O(b^{d+1})$ to $O(b^d)$

2.5 Uniform-Cost Search (UCS)

Frontier: Priority queue, by least path cost

- Equivalent to BFS if all step costs are equal
- Complete: yes, if all step costs are $\geq \epsilon$
  - If not, ever-decreasing step costs may get you stuck infinitely on a suboptimal path
  - Still yes even if b or d is infinite, or search space is infinite
- Optimal: yes, when it is complete
- Time: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$ where C^* is the optimal cost
- Space: $O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$

2.6 Depth-First Search (DFS)

Frontier: Stack

- Complete: yes, as long as depth is finite
- Optimal: no
- Time: $O(b^m)$
- Space: $O(bm)$ (can be $O(m)$ — at each level, just keep track of self and parent)

2.7 Depth-Limited Search (DLS)

Idea: run DFS with depth limit ℓ

- Only works if we know the goal is within ℓ steps
- Time: $O(b^\ell)$
- Space: $O(b\ell)$ (can be $O(\ell)$)

2.8 Iterative Deepening Search (IDS)

Idea: keep performing DLSs with increasing depth limit, until goal node is found

- Good if state space is large, depth of solution unknown
- Can be wasteful with repeated effort, but overhead not that large (e.g. $b = 10, d = 5$: 11%)
- Complete: yes, if b is finite
- Optimal: no, unless step cost is uniform
- Time: $O(b^d)$
- Space: $O(bd)$ (can be $O(d)$)

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes¹	Yes²	No	No	Yes¹
Optimal	No³	Yes	No	No	No³
Time	$O(b^d)$	$O\left(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}\right)$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O\left(b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}\right)$	$O(bm)$	$O(b\ell)$	$O(bd)$

1. Complete if b is finite
2. Complete b is finite and step cost $\geq \epsilon$
3. Optimal if step costs are identical

2.9 Choosing a Search Strategy

Depends on the problem

- Depth: finite/infinite?
- Solution depth: known/unkwown?
- Repeated states
- Step costs: identical/different?
- Completeness and optimality – are they needed?
- Resource constraints (time/space)?

2.10 Search Tracing Problems

Tree-Search

Frontier
S(0)
A(1) B(5) C(15)
S(2) B(5) G(11) C(15)
...

Graph-Search

Frontier	Explored
S(0)	
A(1) B(5) C(15)	S
B(5) G(11) C(15)	S, A
G(10) C(15)	S, A, B

3. Informed Search

Informed search: exploits problem-specific knowledge, uses *heuristics* to guide search

3.1 Best-First Search

Idea: use *evaluation function* $f(n)$ for each node n

- Measures *cost estimate*
- Expand node with lowest estimated cost first

Implementation: priority queue, ordered by non-decreasing cost f

3.2 Greedy Best-First Search (special case of Best-FS)

Evaluation function: $f(n) = h(n)$

- Idea: expand the node that appears the closest to goal
- $h(n)$: cost estimate from n to goal (heuristic)
- Complete: yes, if b is finite
- Optimal: no
- Time: $O(b^m)$, but if heuristic is good can reduce complexity substantially
- Space: $O(b^m)$ (max size of frontier)

3.3 A* Search (special case of Best-FS)

Evaluation function: $f(n) = g(n) + h(n)$

- Idea: expand the path that appears the cheapest
- $g(n)$: cost of reaching n from start node, under the current path (not necessarily the smallest among all paths!)
- $h(n)$: cost estimate from n to goal (heuristic)
- $f(n)$: estimated cost of cheapest path *through* n to goal
- Complete: yes, if there is finite number of nodes and $f(n) \leq f(G)$
- Optimal: yes, if you have an admissible/consistent heuristic
- Time: $O(b^{h^*(s_0)-h(s_0)})$ where $h^*(s_0)$ is actual cost from root to goal
- Space: $O(b^m)$ (max size of frontier)

3.4 Heuristic Design

Admissibility

- $h(n)$ is *admissible* if it never overestimates the cost to reach goal
- Definition: $\forall n, h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from n to goal state

Theorem: if $h(n)$ is admissible, then A* using Tree-Search is optimal

- (Proof: see lecture 3 slide 22)

Consistency

- $h(n)$ is *consistent* if it satisfies triangle inequality
- Definition: $h(n) \leq d(n, n') + h(n')$, where n' is a successor of n
- Lemma: $f(n)$ is non-decreasing along any path, i.e. if h is consistent, then $f(n') \geq f(n)$

Theorem: if $h(n)$ is consistent, then A* using Graph-Search is optimal

- Claim: when A* selects a node n for expansion, the shortest path to n has been found
- (Proof: see lecture 3 slide 26)

Admissibility & Consistency

All consistent heuristics are admissible, but not the other way round.

Example: 8-puzzle

- Heuristic 1: number of misplaced tiles
- Heuristic 2: total Manhattan distance

Dominance

h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ for all n, where both heuristics are admissible

- Dominating heuristics are better: incur lower search costs under A*

3.5 Local Search

Path to the goal is irrelevant; we only want to reach the goal state

Local search algorithms: maintain single "current best" state, and try to improve it

Advantages

- Very little/constant memory
- Find reasonable solutions in large state space

Hill-Climbing Algorithm

- current $\leftarrow$ initial state
- while True:
  - neighbour $\leftarrow$ best successor of current
  - if neighbour's value $\leq$ current's value: return current
  - current $\leftarrow$ neighbour

Problem: depending on initial state, can get stuck in local maxima (or minima)

Solution: try random restarts or sideway moves

4. Adversarial Search

4.1 Adversarial Search Problems

Game: agent vs. agent(s)

- There are other utility-maximising agents
- Solution: a strategy that specifies a move for every possible opponent response

Zero-sum game: agent utilities sum to zero; completely adversarial

Two-player zero-sum game

- *MAX* player: wants to maximise value
- *MIN* player: wants to minimise value

Problem Formulation

- States s, initial state s_0
- Player Player(s): defines which player has the move in state s
- Actions Actions(s): returns set of legal moves in state s
- Transition model Result(s, a): returns state that results from move a in state s
- Terminal test Terminal(s): check whether game has ended
- Utility function Utility(s, p): final numeric value for game with terminal state s for player p

Assume 2-player, deterministic, turn-taking

4.2 Strategies

Strategy s for player i: for every node of the tree that the player can possibly make a move in, specify what player will do

- Winning: s_1^* for player 1 is *winning* — if for any strategy s_2 by player 2, game ends with player 1 as the winner
- Non-losing: t_1^* for player 1 is *non-losing* — if for any strategy s_2 by player 2, game ends with EITHER player 1 as the winner or tie

4.3 Optimal Decisions (Minimax)

Minimax(s)

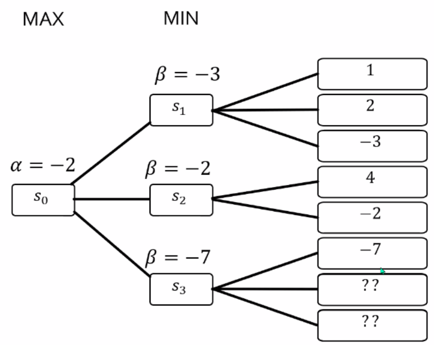
- Utility(s) if TerminalTest(s)
- $\max_{a \in \text{Actions}(s)}$ Minimax(Result(s, a)) if Player$(s) = MAX$
- $\min_{a \in \text{Actions}(s)}$ Minimax(Result(s, a)) if Player$(s) = MIN$

Properties

- Complete: yes, if game tree is finite
- Optimal: yes
- Time: $O(b^m)$ (similar to DFS)
- Space: $O(bm)$ (similar to DFS)

4.4 α-β Pruning

- α: largest value so far for MAX
- β: smallest value so far for MIN



Example above: in the bottom branch, β=-7, but α=-2 $> \beta$. So no need to explore the remaining

α-β pruning

- MAX node n: $\alpha(n)$ = highest observed value found on path from n. Initially $\alpha(n) = -\infty$
- MIN node n: $\beta(n)$ = lowest observed value found on path from n. Initially $\alpha(n) = -\infty$
- ($\star$) Given MIN node n, stop searching below n if there is some MAX ancestor i of n with $\alpha(i) \geq \beta(n)$
- ($\star$) Given MAX node n, stop searching below n if there is some MIN ancestor i of n with $\beta(i) \leq \alpha(n)$

Analysis of α-β pruning

- "Perfect" ordering: time complexity $= O(b^{\frac{m}{2}})$ — can search twice as deep!
- Random ordering: time complexity $= O(b^{\frac{3}{4}m})$ for $b < 1000$

Summary

- Initially, $\alpha(n) = -\infty$, $\beta(n) = +\infty$
- $\alpha(n)$ is MAX along search path containing n
- $\beta(n)$ is MIN along search path containing n
- If a MIN node has value $v \leq \alpha(n)$, no need to explore further
- If a MAX node has value $v \geq \beta(n)$, no need to explore further

4.5 Imperfect, Real-Time Solutions

Time limit

- How to deal with super large search trees? $\Rightarrow$ Limit maximum depth of tree
- Evaluation function: estimated expected utility of state (similar to heuristic)
- Cutoff test: e.g. depth limit

Cutting-Off Search: similar to Depth-Limited Search (DLS)

- Previously: Minimax(s) = Utility(s) if Terminal-Test(s)
- Now: H-Minimax(s) = Eval(s) if Cutoff-Test(s)
- i.e. run minimax until depth d, then use evaluation function to choose nodes
- Can also consider iterative deepening approach

Stochastic Games

- How to deal with games with *randomisation*?
- Game tree now has added *chance layers* — even more complex
- Calculating the expected value of a state — much harder than deterministic games

5. CSPs

5.1 CSP Formulation

- Variables $\vec{X} = X_1, \ldots, X_n$, each with its own domain D_i
- Constraints $\vec{C}$ written in some formal constraint language (logic/algebra)

Objective: find a legal assignment $(y_1, \ldots, y_n)$ for all $y_i \in D_i$

- *Complete*: all variables assigned values
- *Consistent*: all constraints satisfied

Constraint graph: nodes are variables X, edges are constraints

- Unary constraint: draw a self-edge, if not don't need to
- Binary constraint: draw an edge between 2 nodes
- Global constraints: draw a new square, draw edge between square and all nodes

5.2 CSP Search Formulation

- State: intially the empty assignment []
- Transition function: assign a valid value to an unassigned variable, fail if no valid assignments
- Goal test: all variables assigned
- Every solution appears at exactly depth n, search path is irrelevant
- Search tree: has maximum size $n! \times d^n$ (why?)

5.3 Backtracking Search Algorithm

$\textsc{Backtrack}(assignment, csp)$ returns a solution, or failure

- if assignment is complete, return it
- var $\leftarrow$ $\textsc{Select-Unassigned-Variable}(csp)$
- for each value in $\textsc{Order-Domain-Values}(var, assignment, csp)$:
  - if value is consistent with assignment:
    - add $\{var = value\}$ to assignment
    - inferences $\leftarrow$ $\textsc{Inference}(csp, var, value)$
    - if inferences == failure: continue
    - add inferences to assignment
    - result $\leftarrow$ $\textsc{Backtrack}(assignment, csp)$
    - if result $\neq$ failure: return result
  - remove $\{var = value\}$ and inferences from assignment
- return failure

5.4 Backtracking Heuristics

Variable-Order Heuristics: $\textsc{Select-Unassigned-Variable}$

1. Most constraining variable a.k.a. degree heuristic: choose variable that imposes the most constraints on the remaining unassigned variables
   - This is best: it reduces the branching factor => likely get to terminal state faster
2. Most constrained variable a.k.a. Minimum-Remaining-Values (MRV): choose variable with the fewest remaining legal values. Good as tiebreaker

Value-Order Heuristic: $\textsc{Order-Domain-Values}$

1. Least constraining value: choose the value that rules out the fewest values for the neighbouring unassigned variables
   - Because we're "actually trying to solve the problem" in this stage, unlike the variable stage

5.5 Inference

Forward Checking

Terminate search when any variable has no legal values left

AC-3

Arc consistency: X is arc-consistent wrt X_j i.e. arc (X_i, X_j) is consistent, iff for every $x \in D_i$ there exists some $y \in D_j$ that satisfies binary constraint on arc (X_i, X_j)

- ($\star$) Arcs are *directed*
- To maintain AC: remove a value if it makes a constraint impossible to satisfy

AC-3 Algorithm

- *queue* $\leftarrow$ all the arcs in *csp*
- while *queue*:
  - $(X_i, X_j) \leftarrow \textsc{Remove-First}(queue)$
  - if $\textsc{Revise}(csp, X_i, X_j)$:
    - if size of $D_i = 0$ then return *false*
    - for each X_k in $\textsc{Neighbours}(X_i) - \{X_j\}$:
      - add (X_k, X_i) to *queue*

$\textsc{Revise}(csp, X_i, X_j)$ deletes values in D_i that cannot satisfy arc (X_i, X_j)

Time complexity: $O(n^2 d^3)$

- CSP has at most n^2 directed arcs
- Each arc (X_i, X_j) can be inserted at most d times into the queue, since X_i has at most d values
- $\textsc{Revise}$: checking consistency of arc takes $O(d^2)$ time
- AC-3: $O(n^2 \times d \times d^2) = O(n^2 d^3)$

When to use AC-3?

- Preprocessing: do it as first step only
- Backtracking: perform it if domain of X' is updated: check each arc (X_i, X')

6. Logical Agents

Logical agent: Inference Engine + Knowledge Base

6.1 Logic

Logic: formal language of syntax + semantics

- Syntax: defines valid sentences in a language
- Semantics: defines the truth of each sentence, wrt to some possible world of value assignments

Modelling: m models sentence α if α is true under m

- Model represents a "possible world", i.e. assigns truth value to all variables
- $M(\alpha)$ is the set of all models satisfying α

Entailment: $\alpha \Vdash \beta$ means one sentence follows logically from the other

- $\alpha \vDash \beta$ is equivalent to $M(\alpha) \subseteq M(\beta)$
- To infer α from KB, show that $M(KB) \subseteq M(\alpha)$

Validity and satisfiability

- Valid: α is valid if it is true in *all* models
- Satisfiable: α is satisfiable if it is true in *some* model
- Unsatisfiable: α is unsatisfiable if it is true in *no* models
- $KB \Vdash \alpha$ iff $(KB \wedge \neg\alpha)$ is unsatisfiable

6.2 Inference

- Sound: A is sound if $KB \vdash_A \alpha$ implies $KB \Vdash \alpha$, i.e. whatever is derived is correctly entailed
- Complete: A is complete if $KB \Vdash \alpha$ implies $KB \vdash_A \alpha$, i.e. whatever is entailed is derived

Objective of inference: Given KB and α, we want to know if $KB \Vdash \alpha$

Truth Table Enumeration

1. Build truth table of all possible values
2. Evaluate all the models where KB is true
3. $KB \Vdash \alpha$ if all the rows satisfying KB are true for α

Properties

- Sound: directly implements entailment, and calculates all possible inferences from KB by brute force
- Complete: only finitely many combinations of truth assignments, and goes through all
- Time: $O(2^n)$
- Space: $O(n)$ — as enumeration is depth-first

Resolution Algorithm

1. Add $\neg\alpha$ into KB
2. Convert KB to CNF, i.e. 'and's of 'or's, e.g. $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$
3. Pick 2 rules and reduce; repeat
4. If eventual KB is $\emptyset$ i.e. contradiction, then $KB \Vdash \alpha$

Properties: sound and complete (why?)

Forward Chaining

Horn clauses: form $B_1 \wedge B_2 \wedge \ldots \wedge B_k \Rightarrow A$

- Clause with at most 1 positive literal, $\neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_k \vee A$

Algorithm: take the AND-OR graph, and keep popping literals from *agenda* with in-degree 0; these will be true

Properties: sound and complete

- Complete: because FC derives every atomic literal entailed by horn KB

Backward Chaining

Algorithm: work backwards from query Q

- If Q is not known already, then prove by BC the premise of some rule concluding in Q
- Avoid loops: check if the new subgoal is already on the goal stack
- Backtracking DFS

Properties: sound and complete

7. Bayesian Networks

Bayesian network: represents joint distributions via a graph

- Nodes: random variables
- Edges: direction of influence i.e. conditionality
- Joint distribution: $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | Parents(X_i))$
- Conplexity: if each variable has $\leq k$ parents, then network representation requires $O(n2^k)$ values, compared to $O(2^n)$ for full joint distribution

Types of triples

- Common effect: A and B separately $\rightarrow$ C
  - $P(A, B, C) = P(C|A, B) \cdot P(A) \cdot P(B)$
- Common cause: A $\rightarrow$ B and C separately
  - $P(A, B, C) = P(C|A) \cdot P(B|A) \cdot P(A)$
- Causal chain: A $\rightarrow$ B $\rightarrow$ C
  - $P(A, B, C) = P(C|B) \cdot P(B|A) \cdot P(A)$

Markov blanket: a node is conditionally independent of all else, given the values of its *parents*, *children*, and *children's parents*

7.1 Bayesian Network Inference

Bayesian network lets you find the full joint distribution. Infer any query by summing over all cases of the other variables.

7.2 d-Separation

Are X and Y independent given known variables $\epsilon = \{E_1, \ldots, E_k\}$?

- Active path: path is active $\leftrightarrow$ every triple on the path is active
- Active triple: see the chart