ARTIFICIAL INTELLIGENCE OTHER TOPICS

CLASS TEST QUESTIONS

1. (a) What is the Q-value?

The **Q-value** (Quality value) is a function used in reinforcement learning (RL) that estimates the quality of a state-action pair. It represents the expected cumulative reward that can be obtained by taking a specific action in a given state and following a certain policy thereafter. Specifically, the Q-value is used to determine the best possible action to take at each state in order to maximize the cumulative future reward.

Mathematically, the Q-value Q(s,a) for a state s and an action a can be interpreted as:

$$Q(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a]$$

where:

- $\mathbb{E}$ denotes the expected value over future rewards,
- R_t is the total future reward (the sum of rewards from time t onward),
- s_t and a_t represent the state and action at time step t, respectively.

1. (b) Write and explain the equation of calculating updated Q-value for a state-action pair Q(st ,at) using the Bellman Equation in Q-learning. How long is the Q-function updated?

In **Q-learning**, the Q-value is updated using the **Bellman Equation**, which is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

where:

- $Q(s_t, a_t)$ is the current estimate of the Q-value for state s_t and action a_t,
- r_{t+1} is the reward received after taking action a_t in state s_t and transitioning to the next state s_{t+1},
- γ is the **discount factor**, which determines how much future rewards are considered compared to immediate rewards,
- $\max_{a'} Q(s_{t+1}, a')$ is the maximum Q-value over all possible actions a' in the next state s_{t+1},
- α is the **learning rate**, which controls how much the Q-value is adjusted based on the new information.

The Q-function is updated iteratively during the learning process after each action taken. The updates continue until the Q-values converge, meaning the agent has learned the optimal policy. This usually takes a large number of interactions with the environment.

1. (c) What is the role of alpha and gamma in Q-learning?

- **Alpha (α)**: The learning rate, α, determines how much new information overrides the old information. It controls the step size of the update. If α is high (close to 1), the Q-values are adjusted significantly by the new information, meaning the agent learns quickly but may overshoot the optimal policy. If α is low (close to 0), the agent makes smaller updates, leading to slower learning but more stable convergence.
- **Gamma (γ)**: The discount factor, γ, determines the importance of future rewards relative to immediate rewards. If γ is close to 1, the agent values long-term rewards highly, considering the future benefits almost as much as immediate rewards. If γ is close to 0, the agent focuses more on immediate rewards and largely ignores long-term consequences. The choice of γ depends on the problem and the environment.

2. (a) Why is the Target Network used in Deep Q Networks (DQN)?

In **Deep Q Networks (DQN)**, a **Target Network** is used to stabilize training. This is because, in Q-learning, the Q-values are updated iteratively, and using the current Q-values for both the target and the predicted values can lead to instability due to the recursive nature of the updates. The target network is a copy of the Q-network, and its weights are updated less frequently than the primary Q-network.

The role of the target network is to provide a stable target for the Q-value updates. This allows the agent to learn from a consistent target over multiple iterations instead of constantly changing the target Q-values, which helps mitigate issues such as oscillations and divergence during training.

The update rule with the target network is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} Q_{\text{target}}(s_{t+1}, a') - Q(s_t, a_t) \right)$$

Here, Q_{target} refers to the target network.

2. (b) Compare Q-learning and Deep Q-learning

Aspect	Q-learning	Deep Q-learning (DQN)
Function Approximation	Uses a table to store Q-values for each state-action pair.	Uses a neural network to approximate Q-values for state-action pairs.
Scalability	Struggles with large state-action spaces as the Q-table grows exponentially.	Scales well with large state spaces because neural networks can generalize.
Policy Representation	Relies on a simple table to store policy values.	Uses a neural network to represent a policy that can handle complex environments.
Exploration Strategy	Standard epsilon-greedy approach.	Uses epsilon-greedy with additional tricks like experience replay and target networks.

Stability	Can be unstable in complex environments due to the direct update of Q-values.	Stability is improved with the target network and experience replay.
Computational Cost	Low computational overhead as it uses simple table lookups and updates.	Higher computational overhead due to the use of neural networks and backpropagation.

2. (c) Is there any relation between the discount factor and epsilon-greedy algorithm?
The **discount factor**
γ and the **epsilon-greedy algorithm** are two distinct concepts in reinforcement learning, but there is an indirect relationship in how they influence the agent's exploration-exploitation trade-off:

- **Discount Factor (γ)**: It controls how much the agent values future rewards compared to immediate rewards. A higher discount factor means the agent will plan for the long-term, considering future rewards more heavily.
- **Epsilon-Greedy**: This strategy controls the agent's exploration versus exploitation balance. In the epsilon-greedy approach, with probability ϵ, the agent explores by selecting a random action, and with probability $1-\epsilon$, it exploits the action that has the highest Q-value.

While these two factors do not directly interact, the discount factor γ indirectly influences the agent's behavior, which may in turn affect the exploration-exploitation strategy:

- If the agent values future rewards highly (i.e., γ is close to 1), it may favor exploiting actions that maximize long-term rewards. In this case, epsilon-greedy might see less exploration as the agent starts to prioritize exploitation.
- If the agent is more focused on immediate rewards (i.e., γ is small), it might explore more frequently, especially in environments where immediate feedback is important.

In summary, both parameters influence the agent's learning strategy, but they affect different aspects (long-term reward planning vs. exploration behavior). They are typically tuned independently, but their combined effect determines how the agent learns and explores the environment.

OTHER TOPICS

Q. Differences between Q-learning and Deep-Q-Networks

Aspect	Q-Learning	Deep Q-Network (DQN)
Definition	A value-based reinforcement learning algorithm that uses a Q-table to store action-values.	An extension of Q-learning that uses deep neural networks to approximate the Q-function.
Function Approximation	Uses a **Q-table** (tabular method).	Uses a **deep neural network** (function approximator).
State Space	Works well for **small, discrete** state spaces.	Suitable for **large or continuous** state spaces.
Action Space	Discrete actions only.	Typically used for discrete actions (DQN), but variants like DDPG allow continuous actions.
Q-Value Representation	Q(s, a) stored in a table (array).	Q(s, a) estimated by a deep neural network.
Memory Requirement	Low memory usage (depends on state-action pairs).	High memory usage due to neural networks and experience replay buffer.
Computation Time	Fast for small environments.	Slower due to forward/backward passes in neural network training.
Exploration Strategy	Often uses ε-greedy exploration.	Uses ε-greedy, but may also incorporate techniques like **decaying ε**, **noisy nets**.
Generalization	Poor generalization (only knows seen states).	Better generalization due to learned feature representations.
Scalability	Not scalable to complex environments.	Scales well to complex and high-dimensional environments.
Experience Replay	Not applicable.	Uses **experience replay buffer** to break correlations between sequential observations.
Target Network	Not applicable.	Maintains a **target network** to stabilize training.
Training Stability	More stable (less variance).	Less stable (can diverge), but improved using techniques like **Double DQN**, **Dueling DQN**.
Applications	Simple problems like grid-world, maze solving.	Complex tasks like Atari games, robot control, autonomous driving.
Library Support	Easy to implement using NumPy or basic Python.	Requires deep learning libraries (e.g., TensorFlow, PyTorch).
Update Equation	$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$	Similar concept, but Q-values are estimated via neural network prediction and updated via gradient descent.

Q. What is Partial-Order Planning?
Partial-Order Planning (POP) is a planning technique where:
The planner doesn't fix the exact order of all actions up front — only the **necessary ordering** constraints are enforced.
It contrasts with **total-order planning**, where actions are fixed in a single, linear sequence from the start.

Why Partial-Order Planning?
Because in real-world planning:

- Not all actions need a strict order.
- Over-ordering can reduce flexibility.
- Subgoals might interact (like in the **Sussman Anomaly**).

So partial-order planning:

- Delays decisions.
- Allows actions to be interleaved only when needed.
- Avoids unnecessary constraints.

Core Concepts in POP
Let's define the building blocks:

Term	Meaning
Action	Has preconditions and effects (like STRIPS actions).
Plan	A set of actions + ordering constraints + causal links.
Ordering Constraints	E.g., "A before B" when needed.
Causal Link	A link from an action that provides a precondition to another.
Open Condition	A precondition not yet satisfied by any action.
Threat	An action that could break a causal link by undoing a precondition.

Structure of a Partial Plan
A partial-order plan is a triple:
Plan = (A, O, L)

- A: Set of actions (including dummy Start and Finish)
- O: Ordering constraints (e.g., A < B)
- L: Causal links (e.g., A —p→ B means A provides precondition p for B)

POP Algorithm (Simplified)

1. **Start with an empty plan:**
    - Only dummy actions: Start and Finish
    - Start has all initial facts
    - Finish needs all goal facts
2. **Select an open condition (precondition not yet supported).**
3. **Resolve the condition** by:
    - Finding an existing action that satisfies it, or
    - Adding a new action that satisfies it
4. **Add a causal link** from provider to consumer.
5. **Add ordering constraints** to preserve the causal link.
6. **Check for threats**:
    - If any action threatens a causal link, add constraints to avoid the threat.

7. **Repeat** until all open conditions are resolved and no threats remain.

Example: Blocks World
Goal: ON(A, B) and ON(B, C)
Initial State:
ON(A, TABLE), ON(B, TABLE), ON(C, TABLE), CLEAR(A), CLEAR(B), CLEAR(C)
Actions:
Action: STACK(x, y)
Pre: HOLDING(x), CLEAR(y)
Eff: ON(x, y), CLEAR(x)=false, HOLDING(x)=false, ARM_EMPTY=true

Action: PICKUP(x)
Pre: CLEAR(x), ON(x, TABLE), ARM_EMPTY
Eff: HOLDING(x), ON(x, TABLE)=false, CLEAR(x)=false
Plan:
1. Start → PICKUP(A) → STACK(A, B)
2. Start → PICKUP(B) → STACK(B, C)
3. Ordering constraints:
   - PICKUP(A) < STACK(A, B)
   - PICKUP(B) < STACK(B, C)
4. Causal Links:
   - PICKUP(A) —HOLDING(A)→ STACK(A, B)
   - PICKUP(B) —HOLDING(B)→ STACK(B, C)
5. No unnecessary order between actions unless required!

Handling Threats
Suppose an action **THREAT(X)** deletes CLEAR(B), which is a precondition for STACK(A, B).
POP would handle this by:
- **Promotion**: make the threatening action occur **after** the causal link
- **Demotion**: make the threatening action occur **before** the causal link

This way, the threat is avoided without failing the plan.

Advantages of Partial-Order Planning

Benefit	Description
Flexibility	More freedom in execution; only necessary constraints are enforced
Fewer Failures	Can handle interacting subgoals (e.g., Sussman Anomaly)
Efficient	Can reuse subplans and interleave them
Better Plan Reuse	Parts of the plan can be re-applied to new goals easily

Summary Table

Concept	Explanation
Partial-Order Planning	Planning where only necessary order constraints are specified
Goal	Desired state (e.g., ON(A, B) ∧ ON(B, C))
Actions	Defined by preconditions and effects
Causal Link	One action supports another's precondition

Threat	An action that might undo a precondition
Solution	Add constraints to avoid threats

Q. Target Stack Plan

The **Target Stack Plan** is rooted in **hierarchical planning**, **goal regression**, and **partial-order planning**. It's a **domain-specific strategy** that leverages the natural structure of **stacked-object domains** — like the **Blocks World** — to systematically reduce the problem.

1. Classical Planning Model

Target Stack Planning operates within the **classical planning framework**, which is formally defined as:

Planning Problem Definition:

A classical planning problem is a tuple:

$P = (S, A, \gamma, s_0, G)$

Where:

- S: Set of all possible **states** of the world.
- A: Set of **actions/operators**.
- γ: **Transition function** $\gamma: S \times A \rightarrow S$.
- s_0: **Initial state**.
- G: Set of **goal conditions** (conjunctive goal literals).

2. STRIPS Representation

Each action in the planning domain (e.g., PICKUP, STACK) is defined using STRIPS notation:

Example:

Action: STACK(x, y)
Preconditions: HOLDING(x), CLEAR(y)
Add Effects: ON(x, y), ARM_EMPTY, ¬CLEAR(y)
Delete Effects: HOLDING(x), CLEAR(x)

In Target Stack Planning, we work with such STRIPS actions and arrange them in a way that constructs a stack **bottom-up**, ensuring all preconditions are recursively satisfied.

3. Goal Stack Planning and Target Stack Plan

Target Stack Plan is a **structured instance** of the more general **Goal Stack Planning (GSP)** approach, where:

- Goals are pushed onto a stack.
- To satisfy a goal (e.g., ON(A, B)), we push its preconditions.
- When preconditions are met, we apply the action, and pop the goal.

However, the **Target Stack Plan** specifically **leverages stack structure**:

- We decompose the global goal (e.g., $ON(A, B) \wedge ON(B, C) \wedge ON(C, TABLE)$)
- **Sequentially solve subgoals** from the **bottom block** (ON(C, TABLE)) upwards.

This avoids **goal clobbering**, a phenomenon where solving one goal undoes a previously achieved one (e.g., the **Sussman Anomaly**).

4. Planning Strategy and Subgoal Interaction

Recursive Structure:

Let's say your goal is a stack:
[A, B, C, TABLE]
We define:

Goal(i) = ON(Block_i, Block_{i+1})
We recursively plan for:
Goal(3): ON(C, TABLE)
Goal(2): ON(B, C)
Goal(1): ON(A, B)
Each goal Goal(i) introduces:

- A supporting action (like STACK)
- Preconditions which become **subgoals**:
  - E.g., HOLDING(A) and CLEAR(B) for ON(A, B)

These subgoals **must not interfere** with already satisfied conditions like ON(B, C) → hence, the need for a **bottom-up ordering**.

5. Theoretical Motivation: Partial-Order Causality

Causal Links:

A Target Stack Plan can be represented as a **causal chain** of actions:
START → PICKUP(C) → STACK(C, TABLE)
→ PICKUP(B) → STACK(B, C)
→ PICKUP(A) → STACK(A, B) → GOAL
Each link forms:
PICKUP(X) → [HOLDING(X)] → STACK(X, Y)
These causal links must be **protected** from threats (any action that deletes a precondition before it is used).
Target Stack Planning naturally builds plans **without backward dependencies**, reducing threats.

6. Formal Properties

Soundness:

If all preconditions and subgoals are satisfied through valid actions, the resulting plan will satisfy the top-level goal.

Completeness:

If a solution exists (under the assumption of full observability and deterministic actions), Target Stack Planning will find it — provided the goal can be expressed as a stack.

Optimality:

Not guaranteed. Target Stack Plan is **goal-directed**, not cost-optimized. It may produce **longer plans** than optimal planners if subgoal order is not optimal.

7. Complexity Analysis

- State-space size: O(b^d), where:
  - b: branching factor (number of applicable actions)
  - d: depth (number of actions in the plan)
- Planning in the block world is **NP-hard** in general.
- Target Stack Planning **reduces complexity** by pruning impossible or irrelevant sequences using structure.

Summary Table

Concept	Description
Domain	Block-stacking (e.g., Blocks World)
Strategy	Decompose goal stack into base-to-top subgoals
Planning Type	Domain-specific, goal-regression-based

Links to Theory	STRIPS, classical planning, goal stack, causal link planning
Strength	Avoids subgoal interaction (Sussman anomaly), ensures dependency preservation
Limitation	Assumes stack-structured goals; doesn't always find optimal plans

Q. What is a Planning Graph and GRAPHPLAN Algorithm?
A **Planning Graph** is a **data structure** used in **automated planning** to represent how actions and their effects evolve over time. It's used to:

- Efficiently represent the space of possible plans,
- Help determine whether a goal is reachable,
- Prune irrelevant actions using **mutual exclusion (mutex)** information.

It is the **core structure** in the **Graphplan algorithm**, which balances the brute-force and heuristic-based approaches.

Theoretical Foundation
Key Features:

- **Levels**: Alternating layers of **states (propositions)** and **actions**.
- **Forward expansion**: Graph is built **forward** from the initial state.
- **Mutexes**: Mark **incompatible actions or facts** to prune impossible combinations.

Why it works:
Graphplan uses planning graphs to represent a **relaxed planning problem**, where actions are applied in parallel and negative interactions are tracked using mutexes.

Structure of a Planning Graph
The Planning Graph is built in **levels**, like this:
Level 0: P0 (Initial propositions)
Level 1: A0 (Applicable actions)
Level 2: P1 (New propositions from A0)
Level 3: A1 (Next actions)
Level 4: P2
...
Each level contains:

- **Proposition level (P):** Facts or states that are true at this level.
- **Action level (A):** Actions that could be applied (whose preconditions are met).
- **Mutexes:** Between actions or between propositions to capture conflicts.

Planning Graph Construction
Step-by-step:
1. Start from Initial State

- Proposition level P0 = Initial facts

2. Generate Action Level A0

- Include all actions whose preconditions are in P0.

3. Generate Proposition Level P1

- For each action in A0, add its **positive effects** to P1.

4. Add Mutex Links

- Between actions:
  - **Interference**: One deletes a precondition/effect of another.
  - **Competing needs**: Preconditions are mutex.

- Between propositions:
  - If **all ways of achieving two propositions are mutex**, they are mutex.

5. Repeat Until:

- All goals appear in a proposition level **without being mutex**, or
- The graph **levels off** (reaches a fixed point with no new facts/actions).

Goal Extraction (Backward Search)

Once the goals are all present at some level and not mutex:

- Graphplan performs a **backward search** through the planning graph:
  - At level Pi, choose actions in Ai-1 to support each proposition.
  - Ensure selected actions are not mutex.
  - Recurse until reaching level 0.

If search fails → extend the graph one level further.

Example (Simple Blocks World)

Problem:

Initial state: ON(A, TABLE), ON(B, TABLE), CLEAR(A), CLEAR(B) Goal: ON(A, B)

Planning Graph:

P0: ON(A, TABLE), ON(B, TABLE), CLEAR(A), CLEAR(B)
A0: PICKUP(A), PICKUP(B), ...
P1: HOLDING(A), ¬CLEAR(A), ...
...
Continue building until ON(A, B) appears in a proposition level without mutexes.

Mutual Exclusion (Mutex) Details

Mutexes are **crucial** to avoid invalid plans.

Between Actions:

- **Inconsistent Effects**: One action undoes another's effect.
- **Interference**: One deletes another's precondition.
- **Competing Needs**: Preconditions are mutex.

Between Propositions:

- All actions that produce them are pairwise mutex.

Graphplan Algorithm Summary

1. **Construct planning graph** from initial state
2. **Expand until** all goals appear in same level with no mutex
3. **Perform backward search** to find non-mutex actions that achieve the goal
4. **If search fails**, extend graph and repeat

Advantages

Feature	Benefit
Mutex Analysis	Prunes infeasible plans early
Efficient	Often faster than full STRIPS regression
Forward + Backward	Combines forward expansion with backward extraction
Polynomial Graph	Compact representation of exponential plan space

Theoretical Insight

- Planning graphs help estimate **reachability**.
- Graphplan can be used as a **heuristic** in other planners (e.g., FF Planner).

- Planning Graph builds a **relaxed plan space**: ignores delete effects during forward expansion (optimistic assumption).
- Used in **heuristic search**, **partial-order planning**, and **constraint satisfaction**.

Summary Table

Concept	Description
Planning Graph	Layered structure of facts/actions used in Graphplan
Levels	Alternate between propositions and actions
Mutexes	Constraints marking incompatible actions/facts
Uses	Plan feasibility, relaxed planning, heuristic guidance
Benefits	Efficient plan representation and pruning