ARTIFICIAL INTELLIGENCE PyQ ANSWERS

Q. Define the relationship between agent and environment

- **Agent**:
  An **agent** is any entity that perceives its environment through sensors and acts upon that environment through actuators. It could be a robot, a software bot, a self-driving car, or even an AI model in a game.
- **Environment**:
  The **environment** is the external world in which the agent operates. It provides the context and conditions for the agent's actions, and it responds to those actions, sometimes changing as a result.

Agent-Environment Relationship
1. Perception-Action Cycle
The interaction between the agent and environment occurs through a continuous cycle:
1. **Perceive**:
   The agent uses its sensors to **observe the environment** (e.g., a camera, temperature sensor, or data stream).
2. **Decide**:
   Based on its observations and internal logic or model, the agent **makes decisions**.
3. **Act**:
   The agent then uses its actuators to **take actions** that affect the environment.
4. **Feedback**:
   The environment changes based on the agent’s actions, and the agent perceives the updated environment in the next cycle.

This loop is often called the Sense-Think-Act cycle.

Types of Environments
The nature of the environment affects how the agent behaves:

Environment Type	Description
Fully Observable	Agent has complete information (e.g., chess).
Partially Observable	Agent sees only part of the environment (e.g., driving in fog).
Deterministic	Outcomes are predictable (e.g., mathematical calculations).
Stochastic	Outcomes are probabilistic (e.g., poker).
Static	Environment doesn’t change while the agent is thinking (e.g., crossword puzzle).
Dynamic	Environment changes over time (e.g., real-time video game).
Discrete	Finite number of states/actions (e.g., board games).
Continuous	Infinite possibilities (e.g., driving).

Goal of the Agent
The goal of an agent is to:
- Maximize **performance** based on a performance measure.
- Make decisions that lead to the **best possible outcome** in its environment over time.

Q. Name the different types of environments and briefly explain effects of each environment on agent.

1. Fully Observable Environment

- **Definition**: The agent has access to the complete state of the environment at every point in time.
- **Example**: Chess, where all pieces and their positions are visible.
- **Effect on Agent**:
  - Easier to design optimal agents.
  - The agent can make decisions based on full knowledge.
  - Reduces uncertainty in decision-making.

2. Partially Observable Environment

- **Definition**: The agent has limited or incomplete access to the environment's state.
- **Example**: Poker, where opponents' cards are hidden.
- **Effect on Agent**:
  - The agent must maintain internal states or beliefs (memory/history).
  - Decision-making is probabilistic and based on assumptions or estimations.
  - Increases complexity and uncertainty.

3. Deterministic Environment

- **Definition**: The next state of the environment is completely determined by the current state and the agent's action.
- **Example**: A puzzle where each move leads to a predictable result.
- **Effect on Agent**:
  - Easier to plan and predict outcomes.
  - Allows for precise strategies without randomness.
  - No need for probabilistic reasoning.

4. Stochastic Environment

- **Definition**: The next state of the environment is influenced by randomness or uncertainty.
- **Example**: Driving a car on a road with unpredictable pedestrian movements.
- **Effect on Agent**:
  - The agent must account for probabilities.
  - Strategies must be flexible and adaptive.
  - Requires robust handling of uncertainty.

5. Episodic Environment

- **Definition**: The agent's experience is divided into distinct episodes, and each decision does not depend on previous actions.
- **Example**: Image recognition tasks where each image is independent.
- **Effect on Agent**:
  - Simplifies the agent's decision-making.
  - No need for long-term planning or memory.
  - Actions are evaluated in isolation.

6. Sequential Environment

- **Definition**: Current decisions affect future outcomes; episodes are interconnected.
- **Example**: Chess or navigation tasks.
- **Effect on Agent**:
  - The agent must plan ahead and consider future consequences.

- Requires memory and foresight.
- More complex decision-making process.

7. Static Environment

- **Definition**: The environment remains unchanged while the agent is making decisions.
- **Example**: Crossword puzzles.
- **Effect on Agent**:
  - No need to worry about time constraints.
  - Easier to deliberate and optimize decisions.
  - Agent can use search and planning techniques effectively.

8. Dynamic Environment

- **Definition**: The environment changes over time, possibly independent of the agent's actions.
- **Example**: Autonomous driving, where other vehicles move independently.
- **Effect on Agent**:
  - The agent must act quickly and adapt continuously.
  - Time-sensitive decision-making is critical.
  - May require real-time data processing.

9. Discrete Environment

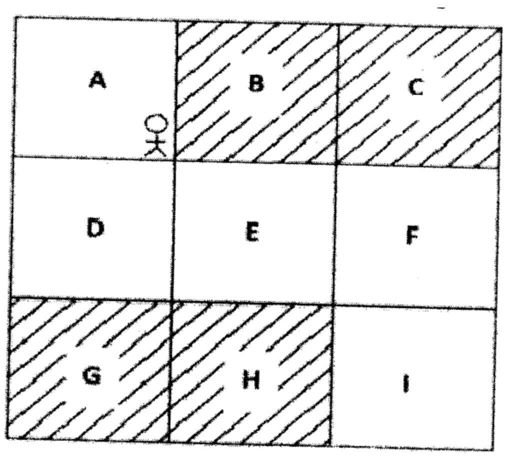
- **Definition**: The environment consists of a finite number of distinct states, actions, and events.
- **Example**: Board games.
- **Effect on Agent**:
  - Easier to model using rules or tables.
  - Ideal for logic-based and rule-based systems.
  - Suitable for classical AI algorithms.

10. Continuous Environment

- **Definition**: The environment has a range of possible values for states and actions.
- **Example**: Robot arm movements or flying a drone.
- **Effect on Agent**:
  - Requires techniques from continuous mathematics (e.g., calculus, control theory).
  - Often involves approximation and learning-based strategies.
  - Harder to model exhaustively.

Each environment type defines **constraints and capabilities** for the agent. Understanding the nature of the environment helps in designing better agents and choosing suitable AI techniques.

Q. In a grid world environment, the goal of the agent is to reach state I starting from state A without visiting the shaded states. In each of the states, the agent can perform any of the four actions: up, down, left, and right to achieve the goal.



Explain what is the outcome of stochastic policy with help of the grid world environment, assuming given a state A, and suppose the stochastic policy returns the probability distribution over the action space as [0.10, 0.70, 0.10, 0.10].

In a **stochastic policy**, instead of choosing a fixed action in a state, the agent chooses actions **based on a probability distribution** over the possible actions. Let's apply this concept to the grid world environment shown in your image.

Given:

- The agent starts at **state A**.
- The **goal** is to reach **state I**.
- The **shaded cells (B, C, G, H)** are **not allowed** (obstacles).
- The **action space** = [Up, Down, Left, Right]
- The stochastic policy for state A is:
  P(actions)=[Up=0.10,Down=0.70,Left=0.10,Right=0.10]

Interpreting the Policy:
The policy is:

- 10% chance to go **Up**
- 70% chance to go **Down**
- 10% chance to go **Left**
- 10% chance to go **Right**

In **state A**, the only valid actions are:

- **Down** → moves to state **D**
- **Right** → moves to **B** (❌ invalid, shaded)
- **Up** and **Left** → go outside the grid (❌ invalid)

Outcome of the Policy in Practice:
Since this is a **stochastic policy**, each time the agent is in state A, it samples an action based on the given probabilities.

- **70% of the time**, the agent goes **Down** to **state D** → ✅ valid move
- **10% of the time**, it tries to go **Right** to **state B** → ❌ invalid (blocked)
- **10%** for **Up** and **10%** for **Left** → ❌ invalid (outside the grid)

So, **only 70% of the time**, the agent will make a successful move from A to D.
In the other **30%** of the times, it attempts an invalid move and likely **remains in state A**.

What This Means for the Agent:

- **Progress is slower** than a deterministic policy that always chooses "Down."
- **Uncertainty** in action selection can cause the agent to **take longer paths** or **get stuck** if not properly handled.
- With enough time (iterations), it can still reach the goal due to the probabilistic nature allowing for correct moves.

Conclusion:

A **stochastic policy** introduces randomness. Even if the optimal action is known (e.g., going "Down" in A), there's a chance the agent will try less optimal or even invalid moves. In your grid world, this means slower or inefficient paths, but it's essential in many RL problems to encourage **exploration**.

Effect of Stochasticity:

Because of randomness:

- At **A**, there's a 30% chance the agent chooses an invalid move and stays in **A**.
- Similar risks apply in **D**, **E**, and **F** if the agent doesn't pick the optimal action.
- Thus, the agent **may take longer** to reach I, or **may oscillate** until it learns or randomly chooses the correct sequence.

🎯 **Summary:**

Step	State	Action Taken	Success?
1	A	Down (70%)	✅ Move to D
2	D	Right	✅ Move to E
3	E	Right	✅ Move to F
4	F	Down	✅ Move to I (Goal)

Q. How the Q-function differs from the value function of Reinforcement Learning?

Value Function (V-function)

- **Denoted as**: V(s)
- **Definition**: The **value function** estimates the expected return (future cumulative reward) starting from state s, and following a certain policy π.
- **Formula**:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s \right]$$

- **Interpretation**: How good it is to be in state s, assuming the agent follows policy π.

Q-Function (Action-Value Function)

- **Denoted as**: Q(s, a)
- **Definition**: The **Q-function** estimates the expected return starting from state s, taking action a, and thereafter following policy π.
- **Formula**:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a \right]$$

- **Interpretation**: How good it is to take action a in state s, assuming the agent follows policy π afterward.

Aspect	Value Function V(s)	Q-Function Q(s, a)
Definition	Expected cumulative reward starting from state s, following a policy π.	Expected cumulative reward starting from state s, taking action a, then following policy π.
Formula	$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s\right]$	$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a\right]$
Meaning	Expected reward from a state	Expected reward from a state-action pair
Used for	Evaluating states	Evaluating actions
What it Evaluates	How good it is to be in a given state under policy π	How good it is to take a particular action in a given state under policy π
Use Case	Evaluates **states**	Evaluates **state-action pairs**
Action Selection	Needs policy to select action	Can directly choose best action via argmax_a Q(s, a)
Policy Improvement	Indirect: needs to combine with a policy (e.g., in Actor-Critic methods)	Direct: optimal policy can be obtained by maximizing Q-values
Computation	Aggregates over all actions under a policy	Evaluates specific actions
Commonly Used In	- Policy Evaluation - Value Iteration - Actor-Critic Algorithms	- Q-Learning - Deep Q-Networks (DQN) - SARSA
Complexity (Dimensionality)	Lower dimensionality (since it depends only on state)	Higher dimensionality (depends on both state and action)
Storage Requirements	Requires storing value for each state	Requires storing value for each state-action pair
Learning Focus	Learn **value of being** in a state	Learn **value of doing** an action in a state
Suitability	Suitable for environments where policy is fixed and known	Suitable for learning **optimal policies**, especially in model-free settings

Q. Write the bellman equation of calculating updated Q-function considering state S and action A

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R_{t+1} + \gamma \cdot \max_{a'} Q(S_{t+1}, a') - Q(s,a) \right]$$

Where:

- s: current state
- a: current action
- S_{t+1}: next state
- $A_{t+1} \sim \pi(\cdot|S_{t+1})$: next action sampled from policy
- R_{t+1}: reward received after taking action `a` in state `s`
- γ: discount factor ($0 \leq \gamma \leq 1$)
- α: learning rate ($0 < \alpha \leq 1$)
- $\max_{a'} Q(S_{t+1}, a')$: maximum expected future reward from next state

Q. Sussman Anomaly

The **Sussman Anomaly** is a classic problem in the field of **Artificial Intelligence (AI) planning**, particularly in **non-linear planning** or **partial-order planning.** It highlights a limitation of early planning systems that used simple goal decomposition.
Let me break it down in a simple and detailed way.

What Is Planning in AI?
In AI, **planning** refers to generating a sequence of actions that an agent can perform to achieve a goal from an initial state.
A planning system tries to:

- Break the goal into subgoals.
- Solve each subgoal.
- Combine those solutions into a plan.

What Is the Sussman Anomaly?
The **Sussman Anomaly** is a **counterexample** that shows how naive **goal decomposition** can fail.
It was introduced by **Gerald Jay Sussman** in the early 1970s to show that solving subgoals independently and then merging them can sometimes **fail to find a solution**.

Problem Setup (Block World Example)
We are working in the **blocks world**, a standard toy domain in AI.
Initial State:
Three blocks: **A**, **B**, and **C**
ON(A, TABLE)

ON(B, TABLE)
ON(C, A)
This means:

- Block A and Block B are on the table.
- Block C is on top of Block A.

Goal:
ON(A, B)
ON(B, C)
This means:

- A should be on B.
- B should be on C.

Where the Problem Occurs

If we try to solve each goal independently:
Subgoal 1: Make ON(A, B)

- We might stack A on B. ✅ Done.

Subgoal 2: Make ON(B, C)

- Now we try to put B on C.
- But **A is on B**, so B can't be moved unless A is removed.
- We have to **undo** the previous subgoal to achieve the next one.

So solving goals independently creates interference.

Why It's an "Anomaly"

In early planners (like STRIPS), goals were solved **one at a time**, in order, without considering interactions. They assumed:

- Once a subgoal is done, it stays done.

But in this case:

- Solving one subgoal prevents the next.
- Solving the second subgoal undoes the first.

This is the **Sussman Anomaly**: naive decomposition fails due to **interactions between subgoals**.

Solution: Nonlinear or Partial-Order Planning

To solve the anomaly, we use more sophisticated planning methods that:

- Allow actions to be partially ordered.
- Interleave actions for different subgoals.
- Consider interactions (threats, dependencies).

One such planner is **NOAH (Nets of Action Hierarchies)** or later **TWEAK, SNLP**, etc.
A valid plan might be:

1. Move C to table.
2. Move B onto C.
3. Move A onto B.

Now the goal is reached **without interference**.

Summary

Concept	Description
Sussman Anomaly	A problem where independent goal solving fails due to interference.
Domain	Blocks world (stacking blocks).
Issue	Solving subgoal A-B prevents solving B-C.

Lesson	Planning must consider **goal interactions**.
Fix	Use **non-linear planning** that can handle interleaved goals.

Q. What is Markov decision process?

Definition:
A **Markov Decision Process (MDP)** is a **mathematical framework** used to model decision-making in situations where outcomes are partly **random** and partly under the control of a **decision maker** (agent).
An MDP provides a formalization for **reinforcement learning problems** and defines the environment in which an agent operates.

Components of MDP:
An MDP is defined by a **5-tuple (S, A, P, R, γ)**:

Component	Description
S	A finite set of **states** representing all possible situations the agent can be in.
A	A finite set of **actions** the agent can take. The set of actions may depend on the current state.
P	A **transition probability function**: $P(s' \mid s,a)$, which defines the probability of moving to state s' when action a is taken in state s.
R	A **reward function**: $R(s,a,s')$, which gives the expected reward received after transitioning from state s to state s'
γ	A **discount factor** $0 \le \gamma \le 1$, which determines the importance of future rewards. A value close to 0 makes the ager it far-sighted.

Markov Property:
The core assumption in an MDP is the **Markov Property**, which states that:
The future state depends only on the current state and action, and not on the sequence of events that preceded it.
Formally:

$$P(S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \ldots, S_0, A_0) = P(S_{t+1} \mid S_t, A_t)$$

This "memoryless" property is essential for simplifying the problem and applying dynamic programming techniques.

Objective of an Agent in an MDP:
The agent's goal is to learn a **policy**
$\pi(a|s)$ that **maximizes the expected cumulative reward** over time:

$$\text{Maximize } \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \right]$$

Q. Write Q-learning algorithm

Q-learning is a value-based **off-policy** algorithm used to find the **optimal action-selection policy** for any given finite Markov Decision Process (MDP).
It learns the **optimal Q-values**: the expected future rewards for a state–action pair under the best policy.

Update the Q-value using the Bellman Optimality Equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \right]$$

Where:

- s: current state
- a: action taken
- r: reward received
- s': next state
- a': possible next actions
- α: learning rate ($0 < \alpha \leq 1$)
- γ: discount factor ($0 \leq \gamma \leq 1$)

Q-Learning Algorithm Steps

Step	**Description**
1	Initialize Q-values: Q(s,a)←0 for all state–action pairs (s, a)
2	Repeat for each episode (until convergence or max episodes):
3	→ Initialize the starting state s
4	→ Repeat (for each step in the episode):
5	→→ Choose an action a from state s using a **policy** (e.g., ε-greedy)
6	→→ Take action a, observe reward r and next state s′
7	→→ Update Q-value using:
	$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$
8	→→ Set s←s′
9	→ Until state s is terminal
10	End Episode
11	After all episodes, the optimal policy is: $\pi(s) = \arg\max_a Q(s,a)$

ARTIFICIAL INTELLIGENCE PyQ ANSWERS ~ FROM MIDSEM

Q. Explain the role of discount factor in RL, considering γ = 0, 1 and varies b/w 0.2 to 0.8

The **discount factor** (γ) in Reinforcement Learning (RL) plays a crucial role in determining how much future rewards contribute to the agent's decision-making. It is a value between **0 and 1** that balances **immediate vs. future rewards** in the **return (cumulative reward)** calculation.

1. Mathematical Role of Discount Factor

The **return** at time step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

where:

- R_{t+1} is the reward at time $t+1$,
- γ determines how much the agent values future rewards.

A **higher γ** makes the agent **more future-focused**, while a **lower γ** makes it **short-sighted**.

2. Effects of Different Values of γ

(i) When γ = 0

- The agent **only considers immediate rewards** and **completely ignores future rewards**.
- The return simplifies to G
  Gt =Rt+1 , meaning it behaves in a **greedy manner**, maximizing only the next reward.
- Useful in **one-step decision-making** problems where only the next action matters (e.g., reflex-based tasks).

(ii) When γ = 1

- The agent **considers the entire future rewards without discounting**.
- It aims for the **longest-term reward maximization**, making it **highly strategic**.
- However, in **infinite-horizon problems**, the return may **not converge**, making it computationally unstable.

(iii) When γ varies (0.2 to 0.8)

- **γ = 0.2** → The agent values **immediate rewards much more** and slightly considers future rewards.
- **γ = 0.5** → The agent balances **short-term and long-term rewards**.
- **γ = 0.8** → The agent **strongly considers future rewards**, optimizing for a **longer horizon** while still discounting somewhat.

3. Practical Considerations

- **Small γ (e.g., 0.2 - 0.4)**: Good for **short-term tasks** like **robotic arms**, where immediate feedback is crucial.
- **Medium γ (e.g., 0.5 - 0.7)**: Balanced strategy for **episodic tasks** like **board games**, where future outcomes matter but immediate actions are still important.

- **High γ (e.g., 0.8 - 0.99)**: Preferred for **long-horizon tasks** like **autonomous driving** or **stock trading**, where long-term success is critical.

4. Choosing γ in Practice
- If γ is **too low**, the agent acts **short-sighted** and may **miss optimal strategies**.
- If γ is **too high**, the agent may **struggle with long-term credit assignment** and may not learn efficiently.

Q. Write the properties of MINIMAX game search algorithm

The **Minimax algorithm** is used in **two-player, zero-sum games** like Chess, Tic-Tac-Toe, and Checkers. It systematically explores possible moves, assuming both players play optimally.

1. Completeness
Minimax is complete if the search tree is finite.
It guarantees finding a solution if a terminal state exists.
Example: In **Tic-Tac-Toe**, Minimax explores all possible moves, ensuring it finds a winning or drawing strategy.

2. Optimality
Minimax is optimal if both players play perfectly.
It assumes the opponent plays optimally and chooses the best possible move to minimize the worst-case loss.
Example: In **Chess**, Minimax ensures the best possible outcome based on available information.

3. Time Complexity
Minimax has exponential time complexity of **O(b^d)**, where:
- b = branching factor (average number of moves per turn).
- d = depth of the game tree.

Example: In Chess (b ≈ 35, d ≈ 100), Minimax becomes infeasible without optimizations like Alpha-Beta Pruning.

4. Space Complexity
Depends on the implementation:
DFS-based Minimax → O(d) (depth-first, stores only one path at a time).
BFS-based Minimax → O(b^d) (breadth-first, stores the entire tree).
Example:
- **Tic-Tac-Toe** (small tree) → Can store the full tree.
- **Chess** (huge tree) → Uses depth-limited search with Alpha-Beta pruning.

5. Deterministic & Zero-Sum
Minimax works in deterministic, zero-sum games.
- **Deterministic:** No randomness; every move leads to a known state.
- **Zero-Sum:** One player's gain is another's loss.

Example:
- **Applicable** → Chess, Tic-Tac-Toe (Fixed moves, no randomness).
- **Not applicable** → Poker (Random cards, bluffing).

6. Limited by Depth and Pruning

Minimax is inefficient for large trees but can be improved using:
- **Depth-limited Minimax –** Stops at a fixed depth (d).
- **Alpha-Beta Pruning –** Reduces explored nodes, improving efficiency.
Example: In Chess, Alpha-Beta Pruning reduces **b^d** complexity to **b^(d/2)**, making deeper searches feasible.

Q. When do you apply Alpha-Beta Pruning in the Minimax Tree?

Alpha-Beta Pruning is applied when we can **avoid evaluating parts of the Minimax tree** that won't affect the final decision. It helps **reduce the number of nodes explored**, making Minimax faster **without changing the result**.
1. Pruning Condition
Prune a branch if we find that it cannot influence the final decision.
- **α (alpha) → Best value for MAX (maximizing player) so far**
- **β (beta) → Best value for MIN (minimizing player) so far**
- If **α ≥ β**, further exploration is **useless**, and we prune that branch.

2. When to Apply Alpha-Beta Pruning?
Pruning occurs in two cases:
1. **Beta Cutoff (β ≤ α) in the Maximizing Level**
   - If a **MAX** node finds a move with a value **≥ β**, further children **are ignored**.
2. **Alpha Cutoff (α ≥ β) in the Minimizing Level**
   - If a **MIN** node finds a move with a value **≤ α**, further children **are ignored**.

3. Benefits of Alpha-Beta Pruning
- Reduces nodes explored from O(b^d) to O(b^(d/2)) → Much faster!
- Works best when the tree is sorted (Best moves first).
- No change in the final Minimax decision.

4. When to Avoid Alpha-Beta Pruning?
- If the tree is unstructured/random, pruning may not help much.
- Not useful in non-deterministic games (like Poker, where chance affects outcomes).
- Sorting moves before searching increases efficiency but adds extra cost.

Q. What is the purpose of a Belief Network ?
A **Belief Network**, also known as a **Bayesian Network (BN)**, is a **probabilistic graphical model** that represents **dependencies** among random variables using **Directed Acyclic Graphs (DAGs)**. It is used for **reasoning under uncertainty** in AI.

Purpose of a Belief Network
1. Probabilistic Reasoning
- Helps **infer hidden (unknown) variables** based on known evidence.
- Computes the **probability of events** occurring.
Example:
- **Medical Diagnosis:** If a patient has a fever, what is the probability they have the flu?
- **Spam Detection:** Given features like sender and keywords, what is the probability an email is spam?

2. Handling Uncertainty in AI
- Real-world AI applications involve uncertainty (e.g., noisy data, incomplete info).
- Bayesian Networks model relationships probabilistically, unlike deterministic logic.
Example:

- A **robotic system** must decide if an object in front is a wall or a door based on noisy sensor data.

3. Causal Relationship Representation
- Unlike simple probability models, BNs **represent cause-and-effect relationships**.
- Helps AI **predict outcomes** when conditions change.
Example:

- **Traffic Prediction:** If it rains, what is the probability of a traffic jam?
    - Rain → Slippery Roads → More Accidents → Traffic Jam

4. Decision Making
- Used in **decision-making systems** to evaluate different actions and their probabilities.
- Supports **decision trees, reinforcement learning, and AI agents**.
Example:

- **Self-driving cars:** Given current road conditions and pedestrian movement, what is the best driving action to take?

5. Learning from Data
- Bayesian Networks can be **built from data** using **Bayesian inference**.
- Allows AI to **learn probabilistic dependencies** and improve over time.
Example:

- AI learns **which symptoms are highly correlated** with specific diseases by analyzing **medical datasets**.

Q. Why is Probabilistic Reasoning Needed in AI?
What is Probabilistic Reasoning?
Probabilistic reasoning in AI deals with **uncertainty** by assigning probabilities to different outcomes. Instead of making rigid, deterministic decisions, AI can **infer and predict outcomes based on likelihoods**. It is used in **Bayesian networks, Hidden Markov Models, Decision Trees, and Reinforcement Learning**.

Why Do We Need Probabilistic Reasoning in AI?
1. Handling Uncertainty
- **Real-world data is incomplete, noisy, or ambiguous—**probabilistic reasoning allows AI to make the **best possible decision** even when full information isn't available.
- AI must **infer missing details** instead of assuming absolute truths.
Example:

- A self-driving car detects a blurry object ahead. Is it **a pedestrian or just a shadow?**
- Using probabilities, the AI can determine the **most likely scenario** and react accordingly.

2. Making Rational Decisions
- AI applications like **medical diagnosis, stock prediction, and robotics** require **decision-making under uncertainty**.

- Probabilistic models help AI **weigh different possibilities** and choose the **most rational action**.
Example:

- In **medical diagnosis**, if a patient has symptoms A, B, and C, what is the probability of **Disease X** vs. **Disease Y**?
- AI computes probabilities and recommends **the most likely diagnosis**.

3. Learning from Data
- AI can learn **patterns and trends from data** using **probability distributions**.
- Unlike rule-based systems, probabilistic models can **adapt and update** based on new information.
Example:

- A **spam filter** assigns a probability score based on **words, sender, and email history** to decide if a message is spam or not.
- If a user marks an email as spam, the AI **updates its probability model** to improve future predictions.

4. Modeling Cause-and-Effect
- Probabilistic reasoning allows AI to **understand causal relationships** rather than just correlations.
- Helps in **predictive modeling** where past events influence future outcomes.
Example:

- **Traffic Prediction System**:
    - If it **rains**, the probability of **traffic congestion** increases.
    - If it **rains and there's an accident**, congestion probability is **even higher**.

5. Optimizing AI Performance
- AI models like **Hidden Markov Models (HMMs), Bayesian Networks, and Reinforcement Learning** use probability to **balance exploration vs. exploitation**.
- This improves AI's ability to **adapt dynamically**.
Example:

- **Reinforcement Learning in Games**
    - AI **chooses moves based on the probability** of winning.
    - Over time, it learns which actions are more **rewarding** and adjusts its strategy.

Applications of Probabilistic Reasoning in AI
- **Robotics –** Navigate uncertain environments.
- **Natural Language Processing (NLP) –** Understand speech and text ambiguities.
- **Medical Diagnosis –** Predict diseases based on symptoms.
- **Fraud Detection –** Identify suspicious transactions using probability.
- **Self-Driving Cars –** Make safe driving decisions under uncertainty.
- **Weather Forecasting –** Predict rain, storms, or temperature changes.

Difference Between Games and Search Problems in AI

Aspect	Games in AI	Search Problems in AI
Definition	Games involve **two or more agents** competing to achieve a goal, where each agent's actions affect the others.	Search problems involve finding a **sequence of actions** that leads to a desired goal state.
Number of Agents	**Multi-agent** environment with competing entities.	**Single-agent** environment, solving a problem independently.
Nature of Environment	**Adversarial**, as agents have **conflicting** objectives (e.g., one wins, the other loses).	**Non-adversarial**, as there is no competition, only finding an optimal solution.
Objective	To **maximize an agent's utility** while minimizing the opponent's success.	To find the **best or optimal path** from an initial state to the goal state.
Decision Process	Agents make **strategic decisions** based on the opponent's possible moves.	The search algorithm explores possible paths systematically to find a solution.
Types of Problems	Chess, Tic-Tac-Toe, Go, Poker, AlphaGo.	Route Planning, Puzzle Solving, Pathfinding, AI Planning.
Evaluation	Uses a **utility function** or evaluation function to decide the best move.	Uses **heuristics, cost functions, and goal tests** to evaluate paths.
Complexity	Often **more complex** due to the need to predict an opponent's moves (e.g., exponential growth in possibilities).	Complexity depends on the **state space** and branching factor but is usually **deterministic**.
Algorithms Used	**Minimax, Alpha-Beta Pruning, Monte Carlo Tree Search (MCTS)**.	**A*, BFS, DFS, Dijkstra's Algorithm, Greedy Best-First Search**.
Example of States	In Chess, a state represents **board positions of all pieces** and the turn of a player.	In a pathfinding problem, a state represents the **current location** of an agent in a graph.

Q. Difference Between Uniform Cost Search (UCS) and Breadth-First Search (BFS)

Aspect	Uniform Cost Search (UCS)	Breadth-First Search (BFS)
Definition	A search algorithm that expands the least-cost node first.	A search algorithm that expands nodes level by level.
Type of Algorithm	**Informed Search** (uses path cost).	**Uninformed Search** (no cost consideration).
Expansion Strategy	Expands the node with the **lowest total path cost (g(n))**.	Expands all nodes at the **current depth** before moving to the next level.
Uses a Cost Function?	Yes, it considers the **cumulative cost (g(n))** from the start node.	No, it treats all edge costs as **equal** (assumes unit cost).
Queue Type (Data Structure)	**Priority Queue** (sorted by path cost).	**FIFO Queue** (First In, First Out).
Optimality	**Yes**, UCS finds the optimal path when costs are positive.	**Yes**, BFS finds the optimal path **only if all edges have the same cost**.
Completeness	**Yes**, UCS is complete if costs are non-negative.	**Yes**, BFS is complete in a finite state space.
Time Complexity	$O(b^{1+\text{floor}(C^*/\epsilon)})$	$O(b^d)$
Space Complexity	$O(b^{1+\text{floor}(C^*/\epsilon)})$	$O(b^d)$
When to Use?	When **path costs vary** and we need the **least-cost solution**.	When **all edge costs are equal**, and we need the **shortest path in terms of steps**.
Example Use Cases	Finding the **cheapest** flight between two cities, shortest path in a weighted graph.	Solving **mazes, shortest path problems with uniform cost** (e.g., unweighted graphs).

Q. Define evaluation function or heuristic function to solve an informed search problem

1. Evaluation Function (f(n)):

- In **informed search**, the **evaluation function** determines the desirability of expanding a node.
- It guides the search by assigning a numerical value to each node.
- The most common form is: $f(n)=g(n)+h(n)$ where:
  - $g(n)$ = Cost from the start node to n
  - $h(n)$ = Heuristic estimate of the cost from n to the goal.

2. Heuristic Function (h(n)):

- A **heuristic function** is an approximation of the remaining cost to the goal.
- It is problem-specific and helps the algorithm prioritize nodes.
- A good heuristic function is **efficient to compute and leads the search efficiently towards the goal**.

Example: A Search in a Grid (Manhattan Distance Heuristic)*

Consider a **grid-based pathfinding problem**, where you must move from **start (2,2)** to **goal (6,6)** using up, down, left, or right moves.

Heuristic Calculation (Manhattan Distance)

One common heuristic for grid-based search is the **Manhattan Distance**, given by:

$h(n)=|x_{goal} - x_n|+|y_{goal} - y_n|$

For a node at (2,2) with a goal at (6,6):

$h(2,2)=|6-2|+|6-2|=4+4=8$

If a move costs **1 unit**, this heuristic gives a reasonable estimate of how far we are from the goal.

Choosing a Good Heuristic

A heuristic should be:

1. **Admissible** → Never overestimates the actual cost.
2. **Consistent (Monotonicity Condition)** → If moving from node A to B incurs cost c $h(A) \leq h(B)+c$
3. **Computationally Efficient** → Should be quick to compute.

Q. Design the heuristic functions for the 8 puzzle problem and show that the heuristic functions are admissible

The **8-puzzle problem** consists of a **3×3 grid** with 8 numbered tiles and one empty space. The goal is to reach a specific arrangement from a given initial configuration by sliding tiles into the empty space.

Two Common Heuristic Functions

1. **Misplaced Tiles Heuristic (h1 (n))**
   - Counts the number of tiles **not in their goal position**.
   - Example:

$$h_1(n) = \sum_{i=1}^{8} \mathbb{I}(tile_i \neq goal\_position_i)$$

   - I is an indicator function (1 if true, 0 otherwise).

2. **Manhattan Distance Heuristic (h2 (n))**
   - Computes the sum of **horizontal and vertical** moves needed for each tile to reach its goal position.
   - Given by:

$$h_2(n) = \sum_{i=1}^{8} (|x_i - x_{\text{goal}}| + |y_i - y_{\text{goal}}|)$$

Proof of Admissibility

A heuristic is **admissible** if it **never overestimates** the actual cost to the goal.

1. Misplaced Tiles Heuristic (h1 (n))

- Each misplaced tile requires at least **one move** to reach the correct position.
- Since each move costs **exactly 1**, the heuristic never **overestimates** the number of moves.
- **Thus, h1 (n) is admissible.**

2. Manhattan Distance Heuristic (h2 (n))

- Each tile must move at least **as many steps as Manhattan Distance suggests**.
- **No tile can reach its goal in fewer moves than its Manhattan Distance.**
- Since **tile swaps are not allowed**, h2 (n) is a **lower bound** on the actual cost.
- **Thus, h2 (n) is admissible.**

Which Heuristic is Better?

- **h1 (n) (Misplaced Tiles) → Simpler but less accurate.**
- **h2 (n) (Manhattan Distance) → More informative and generally performs better in A* search.**

Q. Difference Between A* and AO* Search Algorithm

Aspect	A* Search Algorithm	AO* Search Algorithm
Type of Search	Finds the **shortest path** in a state-space graph.	Finds the **optimal solution** in an AND-OR graph.
Search Space	Works in **state-space graphs or trees**.	Works in **AND-OR graphs**, where nodes represent **decisions and subproblems**.
Graph Type	Uses **single-path search**.	Uses **graph search with AND-OR nodes** (useful in problem decomposition).
Node Type	Each node represents a **state**.	Nodes can be **AND nodes (subproblems must be solved together)** or **OR nodes (one subproblem is sufficient to solve the problem)**.
Expansion	Expands the most promising node based on f(n)=g(n)+h(n).	Expands nodes **recursively** based on subproblem dependencies.
Heuristic Function	Uses a heuristic function h(n) to estimate the cost from node n to the goal.	Uses a heuristic function that **guides search in AND-OR graphs**, considering both subproblems and their dependencies.
Cost Function	Uses f(n)=g(n)+h(n), where: - g(n) = cost from start to n. - h(n) = estimated cost from n to goal.	Uses a cost function based on **aggregated cost of all subproblems** in AND-OR graphs.
Purpose	Used for **pathfinding and shortest path problems**.	Used in **hierarchical problem-solving and game trees**.
Application Areas	- Pathfinding (e.g., Google Maps, Robotics). - Game AI (minimax search). - Planning and scheduling.	- Expert systems. - Hierarchical problem-solving (e.g., medical diagnosis). - Decision-making in uncertain environments.
Optimality	**Guaranteed optimal solution** if h(n) is **admissible and consistent**.	Finds an **optimal solution in an AND-OR graph** but depends on **how problems decompose**.
Computational Efficiency	Can be computationally expensive for large state spaces.	Efficient in **hierarchical problem-solving** but can be complex when AND-OR dependencies are large.

Q.