ARTIFICIAL INTELLIGENCE PyQ ANSWERS

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

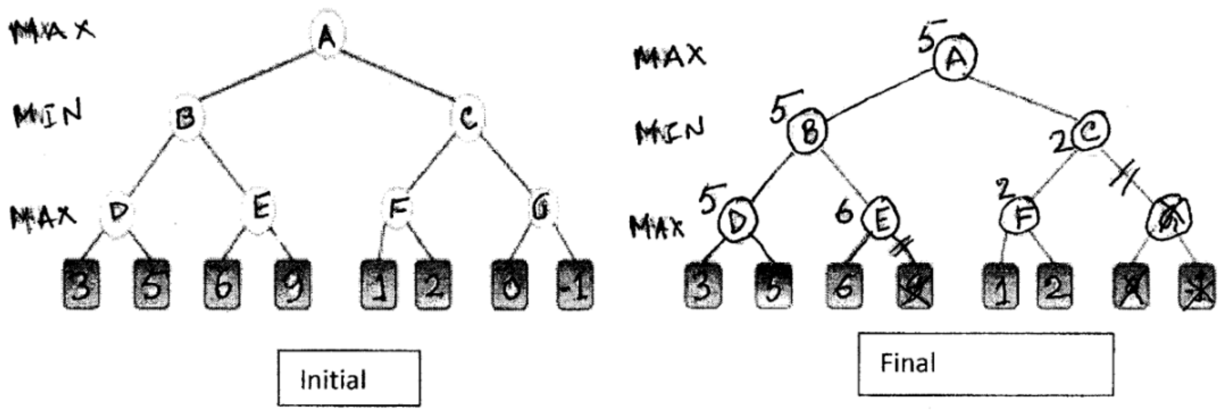
- **Small γ (e.g., 0.2 - 0.4)**: Good for **short-term tasks** like **robotic arms**, where immediate feedback is crucial.
- **Medium γ (e.g., 0.5 - 0.7)**: Balanced strategy for **episodic tasks** like **board games**, where future outcomes matter but immediate actions are still important.

- **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.

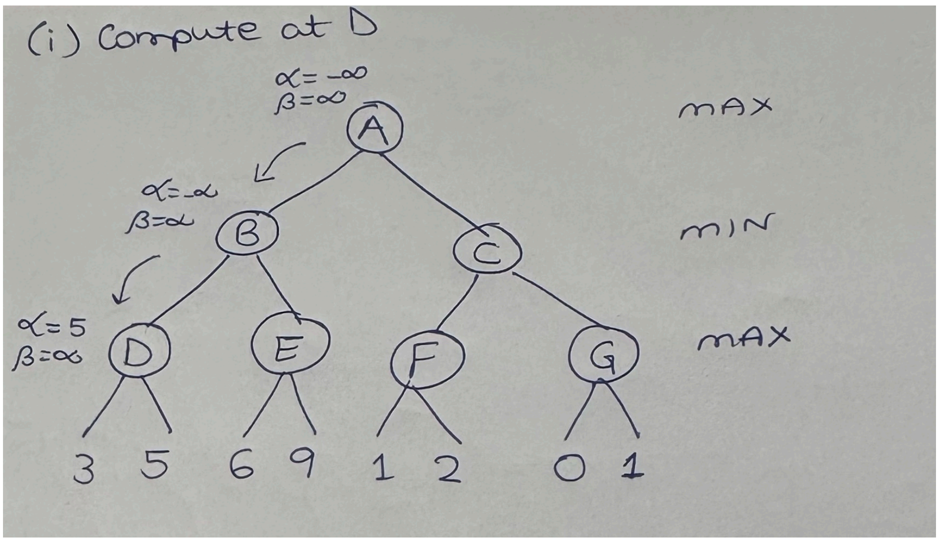
4. Choosing γ in Practice
- If γ is **too low**, the agent acts **short-sighted** and may **miss optimal strategies**.
- If γ is **too high**, the agent may **struggle with long-term credit assignment** and may not learn efficiently.

Q.

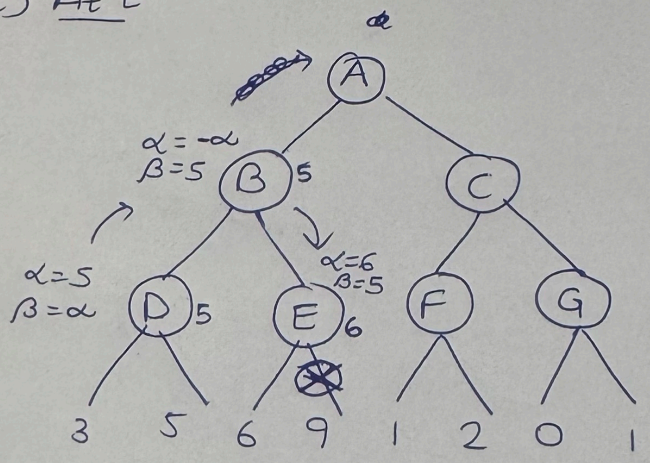
(i) Initial and Final game trees are given below. Explain how the final game tree is achieved using appropriate algorithm.



Ans:



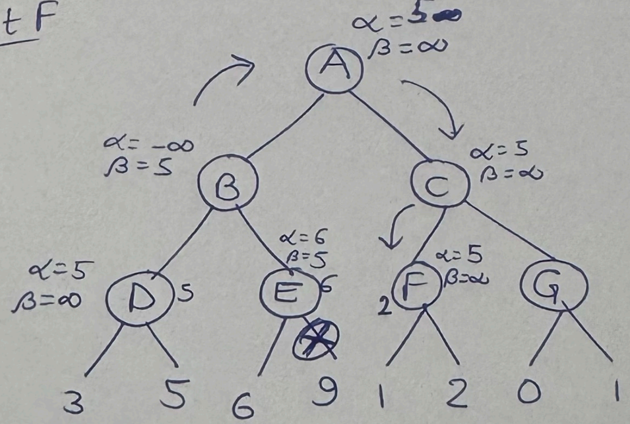
(ii) At E



$\alpha = 6$; $\beta = 5$

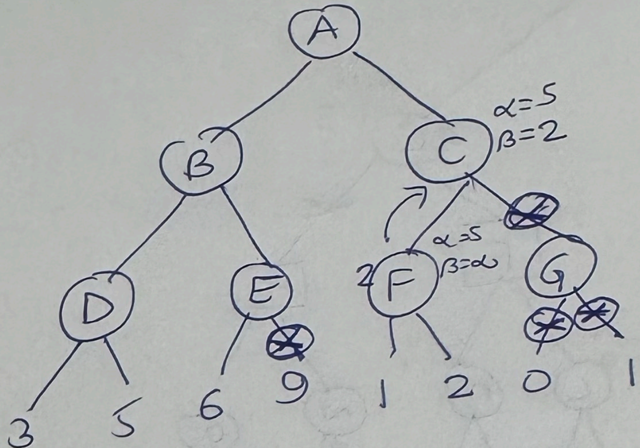
As $\alpha \geq \beta$, so prune right subtree

(iii) At F



Both 1 and 2 are less than 5

(iv) At C,



As $\alpha \geq \beta$, Right subtree of C is pruned.

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.**

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.

Exploration vs. Exploitation Dilemma in AI

1. Definition

The **exploration vs. exploitation dilemma** is a fundamental trade-off in **decision-making AI systems**, especially in **reinforcement learning (RL)** and **multi-armed bandit (MAB) problems**.

- **Exploration**: The AI **tries new actions** to discover **better long-term rewards**.
- **Exploitation**: The AI **chooses the best-known action** to **maximize immediate reward**.

The challenge is to **balance both**—too much exploration wastes time on suboptimal actions, while too much exploitation might prevent discovering better alternatives.

2. Why is this Important?

- If an AI **explores too much**, it wastes time trying bad choices.
- If an AI **exploits too much**, it might get stuck in a **local optimum** and miss better options.

Example:
Imagine an **AI-powered investment system** that picks stocks:

- **Exploration**: Invests in new stocks to **test their profitability**.
- **Exploitation**: Continues investing in **the best-performing stocks so far**.
  Balancing both helps the AI find the **most profitable portfolio** over time.

3. Consequences of More Exploration or More Exploitation

Scenario	Effect
Too much exploration	The AI keeps trying **new, uncertain choices**, leading to **inconsistent rewards** and slow learning.
Too much exploitation	The AI sticks to **known best choices**, missing **better opportunities** and getting trapped in **suboptimal strategies**.

Example: In a **self-driving car**, choosing between two routes:

- **Exploration:** Tries different roads to find a faster route.
- **Exploitation:** Always takes the current best-known route.
- **Risk:** If it never explores, it might never discover a **better traffic-free road**.

4. Strategies to Balance Exploration and Exploitation

Strategy	Description
ε-Greedy Algorithm	AI selects a random action **ε% of the time** (exploration) and the best-known action **(1-ε)% of the time** (exploitation).

Q. How many number of nodes are generated in Depth Limited Search and Iterative Depth Search Algorithms considering depth d=4 and branching factor b=12

$b = 12, \; d = 4$

$$N_{DLS} = b^0 + b^1 + \cdots + b^d$$
$$= 12^0 + 12^1 + \cdots + 12^4$$
$$= 1 + 12 + 144 + 1728 + 20736$$
$$= 22621$$

$$N_{IDS} = (d+1)b^0 + db^1 + (d-1)b^2 + \cdots 2b^{d-1} + b^d$$
$$= 5 \times 1 + 4 \times 12^1 + 3 \times 12^2 + 2 \times 12^3 + 1 \times 12^4$$
$$= 5 + 48 + 432 + 3456 + 20736$$
$$= 24677$$

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	$O(b^{1+floor(C^{*}/\epsilon)})$	$O(b^{d})$
Space Complexity	$O(b^{1+floor(C^{*}/\epsilon)})$	$O(b^{d})$
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 $h(A) \le h(B)+c$
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.