

Midterm Notes

Uninformed and Informed Search

Uninformed Search (Blind Search)

- Searches through the state space without any **domain-specific** knowledge
- Only relies on the problem definition (e.g., goal state, successor function)
- Used when no heuristics are available or the structure of the search space is unknown
- **Examples** ⇒
 - **Breadth-First Search (BFS)** ⇒ Expands the shallowest node first, complete, optimal if step costs are uniform, but has high space complexity
 - **[+]** ⇒ **Guaranteed completeness** (always finds a solution if one exists), **Optimal** for uniform-cost problems
 - **[-]** ⇒ **High Space Complexity, Infeasible for large search spaces**
 - **Depth-First Search (DFS)** ⇒ Expands the deepest node first, uses less memory than BFS, but may get stuck in infinite loops/cycles
 - **[+]** ⇒ **Low memory usage**, Useful when **solutions are deep in the search tree**
 - **[-]** ⇒ **Incomplete, Not optimal**
 - **Uniform Cost Search (UCS)** ⇒ Expands the node with the lowest cost, optimal but slow in large search spaces
 - **[+]** ⇒ **Guaranteed to find the least-cost path** (optimal), **Complete** (as long as costs are positive)
 - **[-]** ⇒ **Slow for large graphs**

Pseudocode

```
# Breadth First Search
def BFS(start, goal):
    frontier = Queue()
    frontier.put(start)
    explored = set()

    while not frontier.empty():
        node = frontier.get()

        if node == goal:
            return reconstruct_path(node) # Return solution

        explored.add(node)

        for child in get_successors(node):
            if child not in explored:
                frontier.put(child)

    return "Failure" # No solution found

# Depth First Search
def DFS(start, goal):
```

```

frontier = Stack()
frontier.push(start)
explored = set()

while not frontier.empty():
    node = frontier.pop()

    if node == goal:
        return reconstruct_path(node)

    explored.add(node)

    for child in get_successors(node):
        if child not in explored:
            frontier.push(child)

return "Failure"

# Uniform Cost Search
# → BFS with a Priority Queue
import heapq # Priority queue

def UCS(start, goal):
    frontier = []
    heapq.heappush(frontier, (0, start)) # (cost, node)
    explored = {}

    while frontier:
        cost, node = heapq.heappop(frontier)

        if node == goal:
            return reconstruct_path(node)

        explored[node] = cost

        for child, step_cost in get_successors(node): # Each child has a step cost
            new_cost = cost + step_cost

            if child not in explored or new_cost < explored[child]:
                heapq.heappush(frontier, (new_cost, child))
                explored[child] = new_cost

    return "Failure"

```



- 🚀 **No heuristics!** → Uninformed search explores blindly.
- 📍 **BFS is complete & optimal** but has **high space complexity**.
- 🔍 **DFS is memory-efficient** but can get **stuck in loops**.
- 💰 **UCS is optimal** but **slow for large graphs**.
- 🚫 **Exponential growth risk!** → Avoid unnecessary node expansion.
- ✨ **Exam Tip:** Know when each search method is **best used**:
 - **BFS:** When the shortest path is required.
 - **DFS:** When memory is limited, and paths are deep.
 - **UCS:** When path cost matters.

Informed Search (Heuristic Search)

- Uses additional problem-specific knowledge to make searching more efficient
- **Examples** ⇒
 - **Greedy Best-First Search** ⇒ Expands the node that appears closest to the goal node based on a heuristic function (e.g., **Manhattan distance in pathfinding/Pacman**). Not necessarily optimal
 - **[+]** ⇒ **Fast in many cases** → focuses on the goal, **Low memory usage** → Only stores necessary nodes
 - **[-]** ⇒ **Not optimal, Incomplete** (in some cases it overlooks better paths)
 - **A* Search** ⇒ Combines UCS and Greedy using the function $f(n) = g(n) + h(n)$, where:
 - $g(n)$ = cost to reach the node
 - $h(n)$ = heuristic estimate of the cost to reach the goal
 - Optimal if $h(n)$ is admissible (never overestimates)
 - **[+]** ⇒ **Guaranteed optimality** if heuristic is admissible, **Efficient pruning** of necessary paths
 - **[-]** ⇒ **Memory-intensive, Can be slow** (if $h(n)$ is not well-designed)
 - **Iterative Deepening A*** ⇒ Reduces memory usage compared to A* while maintaining optimality
 - **[+]** ⇒ **Lower memory usage than A*, Guaranteed optimality**
 - **[-]** ⇒ **Slower than A*, Harder to implement efficiently**





- 🧠 **Heuristics matter!** Good heuristics speed up search.
- 📍 **GBFS is fast** but **not optimal**.
- ⭐ *A is optimal* but **memory-intensive**.
- ✂ *IDA reduces memory* but **is slower than A**.
- ✨ **Exam Tip:**
 - Use **GBFS** for quick approximations.
 - Use **A*** when memory is not a problem.
 - Use **IDA*** when memory is constrained.

Adversarial Game Tree Search

- Games involve opponents, so the goal is to find the best strategy considering an adversary
- Used in turn-based games like chess, tic-tac-toe, and Go

Minimax Algorithm

- **Maximizing Player (MAX)**: Tries to maximize utility
- **Minimizing Player (MIN)**: Tries to minimize utility
- **Minimax Value**: The best achievable outcome for the player assuming the opponent plays optimally
- **Tree Evaluation**:
 - Terminal nodes contain final utility values
 - Internal nodes take the **minimum or maximum** of their children
- **Example** ⇒
 - **MAX** places  → possible board states are generated
 - **MIN** places  → counters **MAX's** best move
 - The process continues recursively until the game ends
- **[+]** ⇒ **Guaranteed to find the optimal move** if both players play optimally.
- **[+]** ⇒ **Applies to a wide range of games** with deterministic and perfect information.
- **[-]** ⇒ **Computationally expensive** → **Exponential complexity** $O(b^d)$ (where b is branching factor, d is depth)
- **[-]** ⇒ **Does not handle randomness well** (e.g., rolling dice in games like Monopoly).



- ◆ **Explores the entire game tree** and assumes **both players play optimally**
- ◆ **Evaluates terminal states** using a utility function
- ◆ **Backtracks values up the tree**, assigning MAX nodes the **maximum value** of their children and MIN nodes the **minimum value** of their children
- ⚠ **Key Weakness**: Very slow for deep trees.

Negamax Algorithm

- A **variation of Minimax** that **simplifies implementation** by using a **single evaluation function**.
- Instead of having separate logic for **MAX and MIN**, Negamax **flips the sign** of the evaluation function at each level.
- **[+]** ⇒ **Simplifies implementation** (only one evaluation function).
- **[+]** ⇒ **Works the same as Minimax but requires fewer lines of code**.
- **[-]** ⇒ **Still suffers from the same inefficiencies as Minimax** (exponential complexity).
- **[-]** ⇒ **Requires additional logic for games with varying evaluation scales**.



- ◆ Instead of tracking **separate MIN/MAX logic**, **Negamax flips signs** when switching players.
- ◆ **Simplifies code** without changing results.
- ◆ Still has **exponential complexity** like Minimax.
- ⚠ **Key Weakness**: **Does not improve efficiency**, only simplifies implementation.

Alpha-Beta Pruning

- Optimizes Minimax by pruning branches that don't affect the result
 - **Alpha** (α) \Rightarrow Best value found so far for **MAX**
 - **Beta** (β) \Rightarrow Best value found so far for **MIN**
 - If a node's evaluation proves worse than what the opponent can force, we stop evaluating that branch
- **Example** \Rightarrow If we know that one branch leads to a loss of 10, we can prune other branches that will result in the same or worse outcomes
- **[+]** \Rightarrow **Significantly speeds up Minimax** without affecting optimality.
- **[+]** \Rightarrow **Reduces the number of nodes evaluated**, best case $O(b^{d/2})$ (where b is branching factor, d is depth)
- **[-]** \Rightarrow **Only effective if nodes are ordered well** \rightarrow Poor move ordering reduces efficiency.
- **[-]** \Rightarrow **Still exponential in the worst case** \rightarrow Does not solve the fundamental complexity issue.



- ◆ **Keeps track of best possible MAX and MIN values** (α and β).
- ◆ **If a move is clearly worse, it stops searching that branch** (pruning).
- ◆ **Does not affect final result but significantly reduces computation**
- ⚠ **Key Weakness: Pruning is only effective if nodes are ordered well**

Expectimax

- **Used when outcomes include randomness**, such as rolling dice
- Instead of minimizing/maximizing, it **takes the expected value** of child nodes
- Works in games like **Pac-Man**, where ghosts move probabilistically
- **Difference from Minimax** \Rightarrow
 - Minimax assumes the worst-case scenario from the opponent
 - Expectimax **averages over all possible outcomes**
- **Issue** \Rightarrow Can overestimate rewards because it assumes probability-weighted outcomes instead of accounting for worst-case scenarios
- **Example** \Rightarrow
 - If rolling a die determines the next move, Expectimax will take the **average of all possible outcomes** to make a decision
- **[+]** \Rightarrow **Better suited for games with randomness** (e.g., Pac-Man, Backgammon).
- **[+]** \Rightarrow **Considers all possible outcomes** instead of assuming optimal play.
- **[-]** \Rightarrow **Computationally expensive** \rightarrow Evaluates all possible outcomes instead of just one.
- **[-]** \Rightarrow **Can overestimate rewards** because it **assumes probability-weighted outcomes instead of accounting for worst-case scenarios**.



- ◆ Replaces MIN nodes with **chance nodes**, which take the **weighted average** of possible outcomes.
- ◆ Works well in **stochastic games** like **Pac-Man, Backgammon**
- ◆ **No worst-case assumption**, considers **all possible outcomes**
- ⚠ **Key Weakness: Can overestimate rewards**, leading to **non-optimal decisions**



- 🟡 **Minimax is optimal** but **slow**.
- 🔵 **Negamax simplifies Minimax** but doesn't improve performance.
- ⚡ **Alpha-Beta Pruning speeds up Minimax** without losing optimality.
- 🎲 **Expectimax is needed for randomness** (but can overestimate rewards).
- ✨ **Exam Tip:**
 - Use **Minimax** for **perfect information** games (e.g., Chess, Tic-Tac-Toe).
 - Use **Alpha-Beta Pruning** to **reduce search space**.
 - Use **Expectimax** when **randomness is involved** (e.g., dice games, Pac-Man).

Constraint Satisfaction Problems (CSP)

- A **problem defined by variables, domains and constraints**
- **Goal:** Assign values to variables that satisfy constraints
- **Examples** ⇒
 - **Sudoku** (each number must be unique in rows, columns, and squares)
 - **Map Coloring** (adjacent regions must have different colors)

CSP Solving Techniques

- **Backtracking Search** ⇒ Recursively assigns values and backtracks when a constraint is violated
- **Forward Checking** ⇒ Eliminates values from domains of unassigned variables to prevent conflicts
- **Constraint Propagation** ⇒ Uses logical inference (e.g., **Arc Consistency Algorithm** to enforce constraints)
- **Heuristics for CSP** ⇒
 - **Minimum Remaining Values (MRV):** Pick the variable with the fewest legal values first
 - **Degree Heuristic:** Pick the variable involved in the most constraints
 - **Least Constraining Variable:** Choose the value that rules out the fewest future options

In-Class Example

Solve the following puzzle:

```
  ONE
+ TWO
+ SIX
-----
  NINE
```

Step 1: Define Variables

Each letter represents a **unique digit (0-9)**:

- O, N, E, T, W, S, I, X

Step 2: Define Constraints

- Each letter **must be a unique digit** (no repeats).
- The equation **must be mathematically correct**.
- **Leading digits cannot be 0** (e.g., $O \neq 0$, $T \neq 0$, etc.).

Step 3: Apply CSP Techniques

1. **Use MRV** → Solve for letters with the fewest possible values first.
2. **Use Forward Checking** → Remove conflicting values as we assign digits.
3. **Use Constraint Propagation** → Ensure sum constraints hold after each assignment.

Solution

Assigning digits that satisfy constraints:

```
  6 5 1
+ 7 3 8
+ 5 0 9
-----
1 9 9 8
```

Final Assignments:

- $O = 6$, $N = 5$, $E = 1$, $T = 7$, $W = 3$, $S = 5$, $I = 0$, $X = 9$



◆ **CSP problems involve a set of variables, domains, and constraints** that must be satisfied.

◆ **Backtracking Search** → Assign a value, backtrack if constraints fail.

◆ **Forward Checking** → Prevents conflicts **before assignment**.

◆ **MRV & Degree Heuristic** → Helps **prioritize difficult variables** first.

◆ **Constraint Propagation** → **Prunes invalid values**, speeding up search.

◆ **Used in:** Sudoku, Map Coloring, Scheduling, Cryptarithmic Puzzles.

⚠ **Key Weakness:** CSP algorithms **can still be slow** for large problems, but heuristics **help optimize** the search!

Markov Decision Processes (MDPs)

- **Use in decision-making under uncertainty**
- **Consists of:**
 - **States (S)** ⇒ Possible configurations of the world
 - **Actions (A)** ⇒ Choices available to an agent
 - **Transition Function (T)** ⇒ Probability of moving to state S' given action A
 - **Rewards (R)** ⇒ Immediate payoff for an action (Reward)
 - **Policy (π)** ⇒ A mapping from states to actions that defines the agent's behavior

- **Discount (γ)** \Rightarrow Discount factor, should one be punished for taking a certain route

Optimal Policy

- A policy π^* maximizes expected long-term rewards
- Optimal Value Function $V^*(S)$: Expected return from state s following π^*

Bellman Equation

- **Defines the recursive relationship between values of states**
- **For a given policy π :**

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s')$$

where γ is the discount factor, and $[\pi, *, q]$ are interchangeable

Example \Rightarrow Robot in a Grid

- **Goal** \Rightarrow Navigate to a target while avoiding obstacles
- **Transitions** \Rightarrow Moving in a direction has a 10% chance of slipping to an adjacent space
- **MDP can handle uncertainty**, while Expectimax assumes known outcomes



◆ **MDPs model decision-making in uncertain environments** by considering states, actions, rewards, and probabilities.

◆ **Goal:** Find an **optimal policy** π^* that **maximizes expected long-term rewards**.

◆ **Bellman Equation** provides a recursive way to compute the **value of each state**.

◆ **MDPs are different from Expectimax** because they handle **probabilistic transitions** rather than assuming known outcomes.

◆ **Key components:**

- **States SS** \rightarrow All possible configurations of the environment.
- **Actions AA** \rightarrow Choices available to the agent.
- **Transition Function $P(s' | s, a)$** \rightarrow Probability of reaching s' from s after action a .
- **Rewards $R(s, a)$** \rightarrow Immediate payoff for taking action a in state s .
- **Discount Factor γ** \rightarrow Determines the importance of future rewards ($0 \leq \gamma \leq 1$).

💡 **Example: Robot in a Grid**

🚀 A robot navigating a grid **faces uncertainty**—each move has a **10% chance of slipping** into an unintended direction.

✅ **MDPs provide a robust framework** to model this randomness and find the best strategy despite uncertainty.

⚠️ **Key Weakness: Solving MDPs exactly (via Value Iteration or Policy Iteration) can be computationally expensive** for large state spaces. However, **approximate methods** (like Reinforcement Learning) help manage complexity!

Partially Observable MDPs (POMDPs)

- MDPs assume the agent knows the state perfectly
- POMDPs add uncertainty in state observation

- Includes \Rightarrow
 - **Observation function:** $P(o|s)$: Probability of observing state o given state s
 - **Belief state:** Probability distribution over possible states

Why POMDPs?

- **More realistic modeling** (e.g., **robot navigation with noisy sensors**)
- **Policies must be belief-dependent**, not state-dependent

Example \Rightarrow Autonomous Drone

- The drone doesn't know its exact location due to **sensor noise**
- User **belief states** to estimate position and decide the best action



◆ **POMDPs model decision-making under partial observability**, where the agent doesn't have perfect knowledge of the environment's state.

◆ **Goal:** Make decisions based on **belief states**, which represent the agent's uncertainty about the true state.

◆ **Key components:**

- **Observation Function** $P(o|s)$ \rightarrow Probability of observing a particular outcome o given the actual state s .
- **Belief State** \rightarrow Probability distribution over all possible states that the agent may be in.
- **Actions** A \rightarrow Choices available to the agent.
- **Transition Function** $P(s'|s,a)$ \rightarrow Probability of transitioning to state s' from s after action a .

💡 **Example: Autonomous Drone**

🚁 A drone navigating a map may have noisy sensor data, making it uncertain about its precise location.

✅ **POMDPs allow the drone to make decisions based on belief states**—it uses sensor information to maintain a probability distribution over possible locations and chooses actions to maximize its chances of success.

⚠️ **Challenge: Solving POMDPs** is computationally more complex than MDPs due to the need to maintain and update belief states, requiring advanced methods like Monte Carlo simulations or approximate dynamic programming.

P1 and P2

Approach in P1

`getSuccessors(self, state)`

- **Purpose:** This function determines all possible next states from a given state in the `CornersProblem`. Each successor state is represented as a tuple containing:
 - The new position `(x, y)`.
 - A tuple tracking which corners have been visited.
 - The action taken to get there.
 - A step cost of `1`.
- **Approach:**

1. State Representation:

- The function receives `state`, which consists of the current position and a tuple tracking visited corners.

2. Generating Successors:

- It iterates over four possible movement directions: `NORTH`, `SOUTH`, `EAST`, and `WEST`.
- The `Actions.directionToVector(action)` function is used to compute movement offsets `(dx, dy)`, which are added to the current position `(x, y)` to get `next_x, next_y`.

3. Wall Collision Check:

- If the computed next position is a wall, the move is skipped.

4. Corner Tracking:

- If Pacman reaches a corner, the corresponding index in `visitedCorners` is updated to `True`.
- The list is then converted back into a tuple for immutability.

5. Adding Successors:

- A tuple representing the new state is appended to `successors`.

6. Expanded Node Counter:

- `self._expanded += 1` ensures that the number of expanded nodes is correctly tracked.

MSTcost_2Dplane(points)

- **Purpose:** Computes the cost of the Minimum Spanning Tree (MST) for a set of points in a 2D plane using Manhattan distance as the edge weight.

- **Approach:**

1. Edge List Generation:

- Uses `itertools.combinations` to generate all possible edges between the points.
- Each edge is represented as `(cost, p1, p2)`, where `cost` is the Manhattan distance between `p1` and `p2`.

2. Sorting Edges:

- Sorts edges in ascending order of cost to facilitate Kruskal's MST algorithm.

3. Union-Find Data Structure:

- Implements path compression in `find(x)` to speed up lookups.
- Implements `union(x, y)` to merge disjoint sets, ensuring no cycles.

4. MST Construction:

- Iterates over sorted edges, adding an edge if it connects two previously disconnected nodes.
- The total MST cost is accumulated and returned.

cornersHeuristic(state, problem)

- **Purpose:** Computes a heuristic estimate of the shortest path from the current position to the nearest unvisited corners.

- **Approach:**

1. Unvisited Corners Extraction:

- Uses list comprehension to filter out visited corners.

2. Base Case:

- If all corners have been visited, the heuristic is `0`.

3. MST-Based Heuristic:

- Selects the closest corner using Manhattan distance.

- Computes the MST of the remaining unvisited corners.
- Adds the closest corner's distance to the MST cost for the final heuristic value.

foodHeuristic(state, problem)

- **Purpose:** Computes a heuristic for the `FoodSearchProblem`, ensuring admissibility and consistency.
- **Approach:**
 1. **Food Location Extraction:**
 - Uses `foodGrid.asList()` to get a list of food positions.
 2. **Base Case:**
 - If no food remains, the heuristic returns `0`.
 3. **Heuristic Calculation:**
 - Computes the MST cost of food positions.
 - Finds the minimum distance from Pacman to any food dot.
 - Returns the sum of the MST cost and the minimum distance.

Approach in P2

ReflexAgent

- **Concept**
 - The ReflexAgent acts based on the immediate game state rather than planning ahead. It selects an action that maximizes its evaluation function at each step.
- **Implementation Details**
 - The agent evaluates each legal move using a heuristic that considers:
 - Distance to food (prefers actions that bring Pac-Man closer to food).
 - Distance to ghosts (avoids actions that move Pac-Man closer to ghosts).
 - Power pellets (favors eating them if ghosts are nearby).
 - Uses the **Manhattan Distance** to measure proximity to important objects.
 - Chooses the action with the highest evaluation score.

MinimaxAgent

- **Concept**
 - The MinimaxAgent models an **adversarial** game environment. It assumes that Pac-Man plays optimally while ghosts play to minimize Pac-Man's score.
- **Implementation Details**
 - Uses a **Minimax tree** where:
 - **Pac-Man (Max node)** tries to maximize the score.
 - **Ghosts (Min nodes)** try to minimize Pac-Man's score.
 - Recursively expands the game tree up to a given depth (**self.depth**).
 - Evaluates the game state at the leaf nodes using an **evaluation function**.
 - Selects the best action based on the minimax value computed.
 - **Base case:** If the game is won, lost, or the max depth is reached, returns the evaluation function value.

AlphaBetaAgent

- **Concept**

- The AlphaBetaAgent optimizes the MinimaxAgent by pruning branches that do not need to be explored, reducing computational complexity.

- **Implementation Details**

- Uses **Alpha-Beta Pruning** to ignore branches that won't affect the final decision.
 - **Alpha (α)**: Best score for the maximizer (Pac-Man) found so far.
 - **Beta (β)**: Best score for the minimizer (Ghosts) found so far.
- If a move's value is worse than a previously explored option, it is **pruned**, saving computation time.
- Follows the same decision-making process as **Minimax**, but more efficiently.
- Still considers Pac-Man as a **Max node** and ghosts as **Min nodes**.
- Improves performance significantly in deeper game trees.

ExpectimaxAgent

- **Concept**

- Instead of assuming ghosts play optimally, the ExpectimaxAgent models ghosts as **random agents** that choose moves uniformly at random.

- **Implementation Details**

- Like Minimax, but **chance nodes** replace the minimizer.
- Instead of choosing the worst-case outcome for Pac-Man, it computes the **expected value** of a state.
- Uses:
 - **Max nodes (Pac-Man)**: Chooses the move with the highest expectimax value.
 - **Chance nodes (Ghosts)**: Computes the average expected value of all possible moves.
- Uses a **probability distribution** to simulate random ghost movement.
- More realistic in environments where ghosts don't play optimally.

betterEvaluationFunction

- **Concept**

- A heuristic function designed to improve Pac-Man's decision-making based on multiple factors.

- **Implementation Details**

- **Base Score**: Uses the game's internal score as a baseline.
- **Food Distance & Count**:
 - Encourages Pac-Man to eat food.
 - Uses **Manhattan Distance** to find the closest food pellet.
 - More reward for being closer to food.
- **Ghost Distance & Scared Timer**:
 - Avoids ghosts if they are dangerous.
 - Moves toward ghosts if they are scared (to eat them).
 - Heavy penalties for being close to an active ghost.
- **Capsules (Power Pellets)**:
 - Encourages Pac-Man to eat them by adding a positive score.
 - Penalizes states where they remain uneaten.
- **Mobility**:

- Encourages Pac-Man to have more available moves.
- Avoids dead-end states.
- **Final Score Computation:**
 - Combines all factors into a weighted sum to determine the best state.

Summary

Agent Type	Strategy
ReflexAgent	Chooses the best immediate move using a heuristic function.
MinimaxAgent	Uses adversarial search, assuming ghosts play optimally.
AlphaBetaAgent	Optimized Minimax with pruning for better efficiency.
ExpectimaxAgent	Assumes ghosts move randomly and calculates expected values.
betterEvaluationFunction	A heuristic function considering food, ghosts, power pellets, and mobility.

Key Concepts

Rational Agent

- **Utility** ⇒ Sum of rewards accumulated over time.
- **Acts to minimize expected utility** based on available information, ensuring decisions are aligned with achieving the agent's goal.

Maximum Expected Utility (MEU)

- The agent **chooses the action** that **maximizes the expected sum of rewards** over its available options.
- The MEU principle helps determine the optimal action in both deterministic and stochastic environments.

Heuristic Pruning

- Used in **search algorithms** to reduce the search space by eliminating less promising paths early, improving efficiency.
- Common in algorithms like *A search*, where heuristics guide the search towards the most promising solutions.

Plan vs Policy

- **Plan** ⇒ A **fixed sequence** of actions to achieve a specific goal, typically used in deterministic settings.
- **Policy** ⇒ A strategy that maps each **state** to an **action**, allowing the agent to decide dynamically. Useful in **MDPs** and **POMDPs**, where future states are uncertain.

Deterministic vs Stochastic

- **Deterministic** ⇒ No randomness involved; given a state and action, the next state is always predictable (e.g., **DFS**, **BFS**).
- **Stochastic** ⇒ Includes randomness, making future states uncertain (e.g., **Expectimax**, **MDPs**, **POMDPs**). The outcome of an action may vary due to probabilistic transitions.

State, Action, Observations

- **State** ⇒ A snapshot of the world at a given time, encompassing all relevant information needed for decision-making.
- **Actions** ⇒ The set of possible choices available to the agent to transition between states.

- **Observations** \Rightarrow Data received from the environment, especially in **POMDPs**, where the agent does not have perfect knowledge of the current state. Observations help update the belief state, which is used to guide decision-making.

Belief State (in POMDPs)

- A **belief state** represents the **probability distribution** over possible states, capturing the agent's uncertainty about the world.
- The belief state evolves as the agent observes more of the environment, refining its understanding of the current situation and updating its decisions accordingly.