1 Planning

Table of Contents

1.1 Introduction

1.1.1 What is planning under uncertainty?

Planning under uncertainty refers to making decisions where the outcomes of actions cannot be known with certainty beforehand. This type of planning is crucial in domains like robotics, artificial intelligence (AI), operations research, and complex system management, where external factors, lack of perfect information, or randomness can impact decision-making processes.

Traditionally, deterministic models assume perfect knowledge of the world and predictability of action outcomes, which simplifies computation and planning. However, real-world systems face stochastic elements where actions may lead to multiple possible outcomes, each with some probability. As a result, planning under uncertainty involves creating robust strategies that consider various scenarios and uncertainties.

From a computational theory perspective, this problem is closely related to the development of algorithms that can operate effectively in non-deterministic environments. These algorithms often rely on probabilistic models (such as Markov Decision Processes or Partially Observable Markov Decision Processes) to navigate uncertainty while attempting to maximize an expected utility or achieve specific goals.

”Uncertainty is an inherent aspect of many real-world decision-making problems. Developing algorithms that can manage this uncertainty is essential for building reliable and adaptable systems” (Russell & Norvig, 2009).

1.1.2 Importance of Theory of Computation

In the Theory of Computation, planning under uncertainty has important theoretical and practical implications. Traditional ToC focuses on deterministic models, such as Turing machines, which assume that given an input, the computation process follows a defined, predictable path to a solution. However, real-world systems—especially in fields like artificial intelligence and robotics—must often handle incomplete or uncertain information.

To handle such challenges, ToC introduces probabilistic and non-deterministic models that allow systems to process incomplete data or uncertain outcomes. These models are necessary for designing algorithms that can make decisions in uncertain environments or when the complete state of the system is not known. Non-deterministic finite automata (NFA), probabilistic algorithms, and randomized algorithms are all tools developed within ToC to manage such problems.

For instance, in non-deterministic polynomial time (NP), a class of problems is defined where a solution, once guessed, can be verified efficiently. This concept lays the foundation for understanding problems where multiple potential paths or outcomes must be evaluated simultaneously, as in planning under uncertainty.

"Incorporating uncertainty into computation theory enhances our ability to model real-world systems where perfect information is seldom available" (Papadimitriou, 1994).

1.2 Basic Concepts

1.2.1 Deterministic vs. Non-Deterministic Planning

Deterministic Planning: Deterministic planning assumes that every action in a given state leads to a predictable and singular outcome. It models systems in which the future state is fully determined by the present state and the actions taken. Deterministic planning algorithms, such as breadth-first search, depth-first search, or A*, are used in scenarios where the environment is entirely known and controlled. Classical computing models like deterministic finite automata (DFA) work under this assumption, where each input leads to a single transition to a next state.

Non-Deterministic Planning: In non-deterministic planning, actions may result in multiple possible outcomes, each with some probability or based on unknown factors. This type of planning requires algorithms that account for these multiple possibilities and work towards optimal strategies despite the uncertainties.

For example, a robot arm in a factory is designed to pick and place objects in a fully controlled environment will use deterministic planning, as there are no unexpected variables while a service robot in a hospital must navigate through hallways that may have people walking around unpredictably or some random items being moved around. It uses non-deterministic planning to prepare for various obstacles and plan alternative routes as needed.

Non-determinism is an essential concept in ToC, where non-deterministic Turing machines (NDTMs) and non-deterministic finite automata (NFA) are used to model systems with uncertainty. These models do not predict a single transition for each input but rather a set of possible transitions. Non-deterministic planning thus allows the planner to explore various possible worlds or outcomes and plan for contingencies.

"Non-deterministic models are powerful because they represent systems in which multiple futures must be considered simultaneously, reflecting real-world complexities" (Papadimitriou, 1994).

1.2.2 Sources of uncertainity

Uncertainty in planning arises from various factors that limit the predictability of outcomes. The three primary sources of uncertainty are environmental uncertainty, perceptual uncertainty, and outcome uncertainty.

1.2.3 Environmental Uncertainty

Environmental uncertainty occurs when the planner has incomplete or imperfect knowledge about the environment. This is particularly relevant in dynamic or partially observable environments, such as outdoor robotics, where factors like weather, terrain, or human interaction may introduce unpredictability.

In computational terms, environmental uncertainty can be modeled using probabilistic models like Markov Decision Processes (MDPs), which capture state transitions and action outcomes probabilistically. These models help systems reason about the environment's evolution and plan under dynamic conditions. For instance, in self-driving cars, the road and weather conditions may change unpredictably, requiring adaptive planning strategies to avoid collisions or hazards.

For example a drone delivering packages outdoors may face environmental uncertainty. It might be due to strong wind or unexpected obstacles like a construction site, requiring real-time adjustments.

"Probabilistic models like MDPs are essential for modeling real-world environments where conditions are dynamic and uncertain" (Sutton & Barto, 2018).

1.2.4 Perceptual Uncertainty

Perceptual uncertainty arises from the limitations of sensors or perception systems that gather data about the environment. Robots, for example, rely on sensors such as cameras, LIDAR, or GPS to gather information. However, these sensors may be noisy, imprecise, or provide incomplete data. This type of uncertainty can be modeled using techniques like Bayesian networks or hidden Markov models (HMMs), which allow systems to infer the true state of the world based on uncertain or partial observations.

Perceptual uncertainty closely relates to computational theory in handling incomplete input data. In ToC, this can be mapped to problems where the input string is partially corrupted or missing, requiring error-correcting algorithms or probabilistic reasoning to infer the correct outcome.

For example an autonomous vacuum cleaner might have trouble detecting small objects like wires on the floor due to limitations in its sensors. It uses probabilistic reasoning to determine whether it should proceed cautiously or choose another path.

"Handling perceptual uncertainty is key to building systems that can make sense of noisy and incomplete data, which is a fundamental challenge in both AI and computation" (Russell & Norvig, 2009).

1.2.5 Outcome Uncertainty

Outcome uncertainty refers to the variability in the results of actions, even when the same action is taken under seemingly identical conditions. For example, a robot may perform the same maneuver twice but achieve different results due to slight mechanical differences, friction, or external factors. In computation, outcome uncertainty can be modeled using probabilistic automata or stochastic processes, which account for the variability in state transitions or results.

Outcome uncertainty introduces challenges in decision-making algorithms, as planners must consider all possible results of an action and develop strategies that optimize performance across these uncertainties. Algorithms like Monte Carlo simulations or randomized algorithms are often employed to model and mitigate outcome uncertainty.

For example a robot arm trying to pick up a fragile object might sometimes drop it if the grip strength or angle slightly varies or it might crush it due to excessive force. To handle this, algorithms like Monte Carlo simulations or stochastic processes are used to optimize performance across all possible outcomes.

"In scenarios where actions have unpredictable outcomes, probabilistic reasoning becomes crucial for designing robust and adaptive systems" (Puterman, 2005).

1.3 Models for Planning Under Uncertainty

1.3.1 Markov Decision Processes (MDPs)

Markov Decision Processes: Markov Decision Processes (MDPs) provide a mathematical framework for planning under uncertainty, where the outcomes of actions are not completely predictable, but the agent will have complete information about the current state of the environment. MDPs model the decision making process as a series of state transitions, where each state transition depends probabilistically on the current state and also depends on the action taken by the agent. MDPs are mainly used when the agent has full knowledge of the state but the outcomes of actions are not certain.

Markov Decision Processes (MDPs) are widely used for decision making in uncertain environments where we don't know what will be the outcome. In robotics, MDPs help autonomous robots plan paths, navigate, and perform tasks in unpredictable situations, optimizing their actions even when outcomes like slipping the object or collisions are uncertain. In autonomous vehicles, MDPs guide in making real-time decisions, such as lane changes and crash avoidance, by considering dynamic factors. In artificial intelligence (AI), especially in reinforcement learning, MDPs provide the foundation for training agents to learn optimal behaviors through trial and error, focusing on long-term rewards.

- **a) Components of MDPs**
  - States (S): The set of all possible configurations of the environment in which the agent operates. Each state represents a "snapshot" of the system at a specific point of time. States provide the conditions and context for decision-making, defining the current situation and surroundings the agent is in. The state space can be continuous or discrete.

  - Actions (A): The set of all actions the agent can take, which influence how the agent transitions from one state to another. The agent must choose an action at each step that determines how it interacts with the environment. Actions can either be discrete or continuous.

  - Transition Function ($T(s, a, s') = P(s'|s, a)$): The transition function describes the probability of moving from one state s to another state s' after taking an action a. It introduces uncertainty into the system by representing the stochastic nature of actions.
    Mathematical Form: The transition function is defined as:

    $$T(s, a, s') = P(s' \mid s, a)$$

    This represents the probability that the agent transitions to state s' given that it was in state s and took action a.
    This function models the dynamics of the environment, incorporating the uncertainty and randomness in the results of the agent's actions. The system does not always behave deterministically; sometimes, an action leads to different outcomes with certain probabilities.
    The sum of transition probabilities from a state s for any action a must equal 1:

    $$\sum_{s'} P(s' \mid s, a) = 1$$

    This ensures that the system always transitions to one of the possible next states.

  - Reward Function ($R(s,a)$): The reward function specifies the immediate reward the agent receives for taking action a in state s. Rewards are numerical values that represent the desirability of being in a particular state or taking a specific action.
    Mathematical Form: The transition function is defined as:

    $$R(s,a)$$

    This function returns a scalar value indicating the reward for performing action a in state s.

  - Discount Factor (γ): The discount factor (γ) is a value between 0 and 1. It reflects the agent's preference for immediate rewards over future rewards. It determines how much future rewards contribute to the total reward.
    The discount factor influences the balance between short-term and long-term planning. A high discount factor ($\gamma \approx 1$) means the agent values future rewards almost as much as immediate rewards, while a low discount factor ($\gamma \approx 0$) indicates a preference for immediate rewards.

- **b) Policy and Value Functions**
  - Policy ($\pi(s)$): A policy is a critical component of MDPs that defines the behavior of an agent in a given environment. It specifies the action an agent should take based on its current state. Formally, a policy can be represented as:

    $$\pi : S \to A$$

    where:

    * S is the set of all possible states in the environment.
    * A is the set of all possible actions available to the agent.

    The role of a policy is to guide the decision-making process, enabling the agent to navigate through its environment effectively.

    Types of Policies:

    * Deterministic Policy: In a deterministic policy, the action taken by the agent is fixed for each state. This means that given a state s, the agent will always choose the same action a:

      $$\pi(s) = a$$

    * Stochastic Policy: A stochastic policy allows for randomness in action selection. The action taken in a state is determined by a probability distribution over possible actions:

      $$\pi(a \mid s) = P(A_t = a \mid S_t = s)$$

This means that when the agent is in state s, it selects action a with a certain probability. For instance, if the probabilities for actions a_1 and a_2 in state s are 0.7 and 0.3, respectively, the agent will choose a_1 70% of the time and a_2 30% of the time.

- State-Value Function ($V(s)$): The state-value function $V(s)$ provides the expected cumulative reward that an agent can achieve starting from state s and following a specific policy π:

  $$V(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s \right]$$

  Where:

  * s_t is the state at time t.
  * $R(s_t, a_t)$ is the reward received after taking action a_t in state s_t.
  * γ is the discount factor, where $0 \leq \gamma \leq 1$, which determines how much future rewards are valued compared to immediate rewards.
  * State Evaluation: $V(s)$ indicates how valuable it is to be in state s under the policy π. Higher values suggest that the state is favorable for achieving future rewards.
  * Policy Improvement: By evaluating states, the agent can identify which states lead to better long-term outcomes and adjust its policy accordingly.
- Action-Value Function ($Q(s,a)$): The action-value function $Q(s,a)$ evaluates the expected cumulative reward of taking action a in state s and then following policy π:

  $$Q(s,a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s, a_0 = a \right]$$

  * Action Evaluation: $Q(s,a)$ provides a direct measure of the expected return from taking action a in state s. This allows the agent to assess the potential effectiveness of specific actions.
  * Optimal Action Selection: The agent can select actions that maximize the expected rewards based on the Q-values. This is particularly useful in dynamic environments where the agent needs to adapt its strategy frequently.

    The relationship between $V(s)$ and $Q(s,a)$ allows for the conversion between state and action evaluations:

    $$V(s) = \sum_a \pi(a \mid s) Q(s,a)$$

This means that the value of a state can be determined by averaging the Q-values of all possible actions, weighted by the policy's probabilities.

1.3.2 Partially Observable Markov Decision Processes (POMDPs)

A Partially Observable Markov Decision Process (POMDP) is an extension of a Markov Decision Process (MDP) that models decision-making in environments where the agent cannot fully observe the current state. This framework is particularly useful when an agent must act under uncertainty, both in terms of the state transitions and the incomplete information about its surroundings.

- **a) Difference Between MDPs and POMDPs**
  - Observability:
    - MDPs: Assume full observability, meaning the agent knows exactly which state it is in at every time step. This allows the agent to make decisions with complete information about the environment.
    - POMDPs: Deal with partial observability. The agent cannot directly observe the current state; instead, it relies on observations that provide incomplete or noisy information about the environment. This necessitates a more complex decision-making process.
  - Belief States:
    - MDPs: since the agent knows the state, decisions are made based on this known state, and the state-value or action-value functions are computed accordingly.
    - POMDPs: since the agent cannot directly observe the state, it maintains a belief state—a probability distribution over all possible states. The agent's decisions are based on this belief, which reflects the uncertainty in the current state. As new observations are gathered, the agent updates its belief state using Bayes' rule.
- **b) Belief States in POMDPs**
  - Belief state: A belief state is a key concept in POMDPs, representing the agent's knowledge (or uncertainty) about the environment. Instead of knowing which state the system is in, the agent assigns probabilities to different states based on its observations and the actions it has taken. Formally, a belief state $b(s)$ is a probability distribution over all possible states s in the environment.
  - Belief Update: Belief states are updated every time the agent takes an action and receives an observation. The belief update process relies on Bayes' rule, which combines prior knowledge (previous belief) with the likelihood of receiving a particular observation given the action. The updated belief is

denoted as $b'(s')$ after the agent takes action a and receives observation o. The belief update equation is as follows:

$$b'(s') = \frac{P(o \mid s', a) \sum_s P(s' \mid s, a) b(s)}{P(o \mid a, b)}$$

Where:

* $b'(s')$ is the updated belief for state s',
* $P(o \mid s', a)$ is the probability of receiving observation o given that the system is in state s' after taking action a,
* $P(s' \mid s, a)$ is the transition probability from state s to s' given action a,
* $b(s)$ is the prior belief for state s,
* $P(o \mid a, b)$ is a normalizing factor representing the probability of receiving observation o after taking action a from belief b.

This process allows the agent to refine its belief about the underlying true state as it gathers more information from its actions and observations.

- Planning in POMDPs: Planning in POMDPs is more complex than in MDPs because the agent must plan over belief states rather than actual states. The policy in a POMDP is no longer a simple mapping from states to actions, but from belief states to actions. This introduces additional computational challenges because belief states exist in a continuous space (since they are probability distributions over the state space).
- Policies in POMDPs: A policy in a POMDP is a function that maps belief states to actions:

$$\pi(b) = a$$

Where b is the belief state and a is the action that the policy recommends. The goal of the agent is to find a policy that maximizes the expected total reward over time, given the uncertainty in both the environment's transitions and observations.

- Challenges in POMDP Planning:
  * Belief Space: The belief space is continuous, making the planning process computationally expensive. The number of possible belief states is far larger than the number of actual states in the environment.
  * Value Function: In POMDPs, the value function is defined over belief states, not the actual states. The value of a belief state is the expected cumulative reward that the agent can achieve from that belief state.

$$V(b) = \max_a \left[\sum_o P(o \mid a, b) \left(R(b, a) + \gamma V(b') \right) \right]$$

Where:

  - $R(b, a)$ is the expected reward for taking action a in belief state b,
  - γ is the discount factor,
  - b' is the updated belief after taking action a and observing o.

- Exploration vs. Exploitation: Like MDPs, the agent must balance exploration (gathering more information to improve its belief) with exploitation (acting to maximize immediate rewards based on the current belief). However, in POMDPs, exploration is even more critical because the agent's understanding of the environment is incomplete.
- Applications of POMDPs: POMDPs are used in various real-world applications where full observability is not feasible:
  * Robotics: Autonomous robots that navigate complex environments with limited sensor data often rely on POMDP frameworks for decision-making.
  * Medical Decision-Making: In healthcare, POMDPs can model the uncertainty in patient health states, where medical tests provide only partial information.
  * Game AI: In video games or real-world strategic games, POMDPs help model situations where players have incomplete information about the opponents' states.
  * Autonomous Driving: Self-driving cars can use POMDPs to make decisions based on uncertain observations of other vehicles or pedestrians.

1.4 Algorithms for Solving Planning Problems Under Uncertainty

1. Value Iteration Algorithm in Planning Under Uncertainty

The Goal of Value Iteration

The objective is to determine the **optimal policy** π^* that tells the agent what action to take in each state to maximize the expected sum of rewards over time. The solution is represented by the **value function** $V(s)$, which gives the expected cumulative reward starting from state s.

The Bellman Equation

Value Iteration works by repeatedly applying the **Bellman equation** to compute the value of each state. The Bellman equation for a given state s is:

$$V(s) = \max_{a \in A} \left[R(s, a) + \gamma \sum_{s'} T(s, a, s') V(s') \right]$$

where:

- $R(s, a)$: The immediate reward obtained after taking action a from state s.
- $T(s, a, s')$: The probability of transitioning from state s to state s' by taking action a.
- $V(s')$: The value of the next state s', which is the expected cumulative reward from state s' onwards.

This equation states that the value of a state is the maximum expected reward the agent can obtain by choosing the best action a, considering both immediate rewards and the discounted value of future states.

Algorithm 1 Value Iteration Algorithm for MDPs

```
1:  Initialize $V(s)$ for all states $s$ to some arbitrary values (e.g., $V(s) = 0$ for all $s$)
2:  repeat
3:      $\Delta \leftarrow 0$        ▷ Tracks maximum change in value function during iteration
4:      for each state $s$ do
5:          $v \leftarrow V(s)$        ▷ Store current value of $V(s)$
6:          Update $V(s)$ using:
                $$V(s) \leftarrow \max_a \sum_{s'} P(s'|s, a) \times \left[R(s, a, s') + \gamma \times V(s')\right]$$
7:          $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
8:      end for
9:  until $\Delta < \theta$        ▷ Convergence condition: threshold $\theta$
10: Policy Extraction:
11: for each state $s$ do
12:     $\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s, a) \times [R(s, a, s') + \gamma \times V(s')]$
13: end for
14: Return $V$ and $\pi$
```

The Value Iteration Algorithm

The value iteration process consists of the following steps:

1. **Initialization**: Start with an initial value function $V(s)$ for all states (often set to 0 for all states).
2. **Update Step**: For each state s, update the value $V(s)$ using the Bellman equation. This involves computing the expected reward for each possible action a and then taking the maximum.
3. **Convergence Check**: Repeat the update process until the value function $V(s)$ converges, meaning the change in values between iterations is smaller than a predefined threshold, indicating that the values are stable.

4. **Optimal Policy**: Once convergence is achieved, the optimal policy $\pi^*(s)$ can be derived by selecting the action a that maximizes the expected reward in each state:

$$\pi^*(s) = \arg\max_{a \in A} \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s') \right]$$

Handling Uncertainty

Value iteration accounts for uncertainty through the transition model $T(s, a, s')$, which specifies the probability distribution of transitioning to a new state s' given a current state s and an action a. Since the environment is uncertain, the agent doesn't always end up in the same state after taking the same action. The Bellman equation integrates this uncertainty by considering the expected value over all possible next states, weighted by the probabilities of transitioning to those states.

Practical Considerations

- **Computational Complexity**: Value Iteration can be computationally expensive for large state spaces, as it requires iterating over all states and actions multiple times. However, it is guaranteed to converge to the optimal policy given enough iterations.
- **Convergence**: Value Iteration is guaranteed to converge to the optimal value function and policy as long as the discount factor γ is less than 1, and the environment satisfies the assumptions of the MDP (such as finite state and action spaces).
- **Approximation**: In practice, due to the size of real-world problems, approximate methods or variations like **Q-learning** may be used to handle large or continuous state spaces.

Advantages and Limitations

Advantages:

- Provides an exact solution for MDPs with finite state and action spaces.
- Can handle uncertainty in the transition dynamics and rewards.

Limitations:

- Requires knowledge of the complete transition model and rewards, which may not always be available in real-world scenarios.
- Can be computationally infeasible for large state spaces or environments with continuous variables.

2. Policy Iteration Algorithm in Planning Under Uncertainty

Policy Iteration is an algorithm used to solve **Markov Decision Processes (MDPs)** for planning under uncertainty. It alternates between evaluating a given policy and improving it until it converges to an optimal policy. This makes Policy Iteration efficient for MDPs with small to medium-sized state spaces.

The Goal of Policy Iteration

The goal is to find an **optimal policy** π^*, which specifies the best action in each state to maximize the expected sum of rewards over time.

Algorithm 2 Policy Iteration Algorithm for MDPs

```
1: Initialize policy $\pi(s)$ arbitrarily for all states $s$
2: repeat
3:     Policy Evaluation:
4:     Initialize $V(s)$ for all states $s$
5:     repeat
6:         $\Delta \leftarrow 0$                ▷ Track maximum change in value function
7:         for each state $s$ do
8:             $v \leftarrow V(s)$
9:             Update $V(s)$ as:
                   $$V(s) \leftarrow \sum_{s'} P(s'|s,\pi(s)) \times \left[R(s,\pi(s),s') + \gamma \times V(s')\right]$$
10:            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
11:        end for
12:    until $\Delta < \theta$                  ▷ Convergence condition for policy evaluation
13:    Policy Improvement:
14:    Policy_stable $\leftarrow$ True
15:    for each state $s$ do
16:        old_action $\leftarrow \pi(s)$
17:        Update $\pi(s)$ as:
                   $$\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a) \times \left[R(s,a,s') + \gamma \times V(s')\right]$$
18:        if old_action $\neq \pi(s)$ then
19:            Policy_stable $\leftarrow$ False
20:        end if
21:    end for
22: until Policy_stable is True               ▷ Terminate if policy is stable
23: Return $V$ and $\pi$
```

The Two Steps of Policy Iteration

a. Policy Evaluation

In this step, the algorithm calculates the **value function** $V^\pi(s)$ for a given policy π:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V^\pi(s')$$

where:

- $R(s, \pi(s))$ is the reward for taking the action specified by policy $\pi(s)$ in state s.
- $T(s, \pi(s), s')$ is the probability of transitioning to the next state s' from state s by following $\pi(s)$.

Policy Evaluation iteratively calculates $V^\pi(s)$ until it stabilizes.

b. Policy Improvement

After evaluating the value function $V^\pi(s)$, the algorithm improves the policy by choosing actions that maximize the expected reward:

$$\pi'(s) = \arg\max_{a \in A} \left[R(s, a) + \gamma \sum_{s'} T(s, a, s') V^\pi(s') \right]$$

If $\pi'(s) = \pi(s)$ for all s, the policy is optimal and the algorithm terminates.

Policy Iteration Algorithm

The steps of Policy Iteration are:

1. **Initialize** a policy π arbitrarily.
2. **Policy Evaluation**: Compute the value function $V^\pi(s)$ for the current policy.
3. **Policy Improvement**: Update the policy by choosing the best action for each state based on $V^\pi(s)$.
4. **Convergence Check**: Repeat until the policy stabilizes.

Handling Uncertainty

Policy Iteration addresses uncertainty through the transition model $T(s, a, s')$, allowing the algorithm to consider expected values over all possible outcomes.

Advantages and Limitations

Advantages:

- Efficient convergence, often faster than Value Iteration.
- Guaranteed convergence to an optimal policy.

Limitations:

- Computationally intensive for large state spaces.
- Requires knowledge of the complete transition and reward models.

3. Belief State Space Search Algorithm in Planning Under Uncertainty

The **Belief State Space Search** algorithm is used in planning under uncertainty, particularly when an agent cannot determine its exact state. Instead, the agent maintains a **belief state**, a probability distribution over possible states that represents its knowledge about its potential location in the environment. This algorithm is commonly applied in **Partially Observable Markov Decision Processes (POMDPs)**.

Belief State Update

Since the agent does not know its exact state, it uses observations to update its belief state after each action. This update involves two steps:

a. Predict

The agent predicts the belief based on the action taken and the transition probabilities. For an action a and current belief state $b(s)$, the predicted belief $b'(s')$ is:

$$b'(s') = \sum_{s} T(s, a, s') b(s)$$

b. Correct

The agent adjusts its belief based on the received observation o using Bayes' rule:

$$b'(s') = \frac{Z(s', a, o) \cdot b'(s')}{\sum_{s'} Z(s', a, o) \cdot b'(s')}$$

This corrected belief provides an updated estimate of the state distribution based on the new information received.

- **State (S)**: Represents possible configurations of the system; the true state is unknown to the agent.

- **Belief State (b)**: A probability distribution over all possible states, indicating the agent's estimate of being in each state.
- **Actions (A)**: Choices available to the agent at each time step.
- **Observations (O)**: Information received by the agent after taking an action, which provides clues about the true state.
- **Transition Model (T)**: The probability $T(s, a, s')$ of transitioning from state s to s' by taking action a.
- **Observation Model (Z)**: The probability $Z(s', a, o)$ of receiving observation o after reaching state s' by taking action a.
- **Reward Function (R)**: The reward received for taking an action in a state, aiming to guide the agent toward desirable outcomes.

Belief State Space Search

The Belief State Space Search algorithm involves searching over the space of belief states rather than over actual states. The key steps are:

1. **Define Initial Belief State**: Start with an initial belief state b_0, representing the agent's initial knowledge about its state.
2. **Generate Successors**: For each possible action, compute the new belief state by applying the belief update equations (predict and correct).
3. **Select Actions**: Choose actions that maximize the expected reward based on the updated belief states.
4. **Policy Evaluation**: Use dynamic programming techniques, such as **Value Iteration** or **Policy Iteration** in the belief space, to determine the best policy.

Handling Uncertainty in Belief Space

The Belief State Space Search algorithm addresses uncertainty by continuously updating the belief state based on observations, allowing the agent to adapt its policy as new information becomes available.

Challenges in Belief State Space Search

Belief State Space Search can be challenging due to the high dimensionality and complexity of the belief space. This complexity arises because:

- Each belief state is a distribution over all possible states, making the belief space much larger than the original state space.

Algorithm 3 Belief State Space Search Algorithm for POMDPs

```
 1: Initialize the initial belief state $b_0$ based on prior knowledge of the environment
 2: repeat
 3:     for each belief state $b$ do
 4:         for each action $a$ available in belief state $b$ do
 5:             Initialize $Q(b, a) \leftarrow 0$
 6:             for each possible observation $o$ do
 7:                 Update the belief state based on $a$ and $o$:
                        $$b' \leftarrow \tau(b, a, o)$$
 8:                 Compute the immediate reward $R(b, a)$ and add the discounted value
                    of future beliefs:
                        $$Q(b, a) \leftarrow Q(b, a) + P(o|b, a) \times \big[R(b, a) + \gamma \times V(b')\big]$$
 9:             end for
10:             Update $V(b)$ with the maximum value of $Q(b, a)$ over all actions:
                        $$V(b) \leftarrow \max_a Q(b, a)$$
11:             if the best action for belief state $b$ has changed then
12:                 Update the policy $\pi(b)$ to the new action
13:             end if
14:         end for
15:     end for
16: until convergence or for a fixed number of iterations
17: Return the optimal policy $\pi$
```

- Computations involving the transition and observation models can be intensive, as each action-observation pair can lead to a unique belief state update.

To manage this complexity, approximations such as **sampling** or **Monte Carlo methods** are often used.

Applications of Belief State Space Search

Belief State Space Search is widely used in domains where exact state information is unavailable. Examples include:

- **Robotics**: When a robot navigates using uncertain sensors, it uses a belief state to represent possible locations and to make movement decisions.
- **Healthcare**: In diagnostic processes, belief states can represent the likelihood of different medical conditions based on symptoms.
- **Autonomous Driving**: Self-driving vehicles use belief states to handle uncertainty about the locations and intentions of nearby objects.

4. Monte Carlo Methods in Planning Under Uncertainty

Monte Carlo methods are valuable in **MDPs** and **POMDPs**, where the goal is to maximize the expected cumulative reward over time. Monte Carlo methods estimate the **value functions** by sampling sequences of states, actions, and rewards (episodes) from the environment. By averaging rewards in these episodes, Monte Carlo methods provide a reliable approximation of the value function, guiding the agent toward an optimal policy.

Components of Monte Carlo Methods

- **Episodes and Sampling**: Monte Carlo methods require complete episodes, sequences of interactions from start to end. Each episode provides data for estimating expected returns based on observed rewards.
- **Returns and Value Estimation**:

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

  The value of a state s or state-action pair (s, a) is the average return of episodes starting in that state or pair.
- **Exploring Starts**: Random initial states and actions ensure thorough exploration of the environment, allowing all states and actions to be sampled.
- **On-Policy and Off-Policy Methods**:
  - **On-Policy** evaluates and improves the policy the agent is using.
  - **Off-Policy** allows the agent to learn an optimal policy independently of the policy used to generate episodes, often using **importance sampling**.

Monte Carlo Control Algorithm

Monte Carlo control finds an optimal policy by alternating between **policy evaluation** and **policy improvement** steps:

1. **Policy Evaluation**: For each state s, compute the expected return by averaging observed returns.
2. **Policy Improvement**: Update the policy to choose actions that maximize expected return:
$$\pi(s) = \arg\max_a Q(s, a)$$
3. **Iterate until Convergence**: Repeat evaluation and improvement until policy converges.

Algorithm 4 Monte Carlo Methods Algorithm for MDPs

```
1: Initialize policy $\pi(s)$ arbitrarily for each state $s$
2: Initialize $Q(s,a) \leftarrow 0$ and returns$(s,a) \leftarrow []$ for all state-action pairs $(s,a)$
3: repeat                                                    ▷ For each episode
4:     Generate an episode using policy $\pi$: $(s_0, a_0, r_1, s_1, a_1, \ldots, s_T)$
5:     for each state-action pair $(s,a)$ in the episode do
6:         Compute the return $G$ from that point onward in the episode
7:         Append $G$ to returns$(s,a)$
8:         Update $Q(s,a)$ as the average of returns$(s,a)$
9:     end for
10:    for each state $s$ in the episode do
11:        Update policy $\pi(s)$ to the action with the highest $Q(s,a)$
12:    end for
13: until convergence or for a fixed number of episodes
14: Return the policy $\pi$ and action-value function $Q$
```

Advantages and Limitations

Advantages:

- **Model-Free**: Does not require knowledge of transition or reward models.
- **Exploration**: Ensures comprehensive exploration of the state space.

Limitations:

- **Episode Requirement**: Needs episodes to terminate to compute returns.
- **Slow Convergence**: Converges slowly in complex or large environments.

Applications

Monte Carlo methods are widely applied in fields such as:

- **Finance**: Portfolio optimization and option pricing.
- **Robotics**: Navigation and decision-making.
- **Gaming**: Strategic planning in uncertain environments.

1.5 Examples

1.5.1 Robot Navigation

Robot Navigation Example: Navigating Through a Room
Imagine a robot named "Robo" that needs to navigate through a room with obstacles like chairs and tables to reach a charging station. Here's how Robo can do this:

1. Understanding the Environment (Probabilistic Robotics)
Robo has sensors (like a camera, lasers, or sonar) to understand the room around it. However, these sensors aren't perfect—sometimes they make mistakes, like misjudging the distance to an object or not seeing a small chair leg.
Robo uses probabilities to make sense of the world. This is called Probabilistic Robotics. It knows that even though its sensor says there is a chair in front, there's a small chance the sensor might be wrong. It combines all its sensor readings using math (Bayes filters) to build a map of the room. This way, Robo can make decisions despite uncertainty in its sensor data.

2. Making Decisions: Where Should Robo Move? (Artificial Intelligence & Markov Decision Processes)
Robo's job is to move from one part of the room to another (say, from the entrance to the charging station). But how does it decide where to go?
Using a concept Markov Decision Process (MDP), Robo can break down the room into a series of states:

- A state might be something like "Robo is next to a chair" or "Robo is 2 feet from the wall."
- It then decides on actions: Should it go forward? Turn left? Turn right?

Every action Robo takes has a reward:

- If Robo moves closer to the charging station, it gets a positive reward.
- If Robo crashes into a chair, it gets a negative reward.

The goal is to choose actions that maximize the rewards, i.e., get to the charging station as quickly as possible without hitting obstacles.

3. Learning from Experience (Reinforcement Learning)
What if Robo doesn't know the best path to the charging station? It can learn by trial and error using Reinforcement Learning (RL).
In the beginning, Robo might move randomly, bumping into things. Over time, it learns:

- "If I take this path, I usually reach the charging station quickly."
- "If I go that way, I always crash into the chair."

Robo uses this learning to update its behavior. It tries to choose actions that give it the best outcomes based on its past experiences. The more Robo practices, the better it gets at navigating the room without crashing.

4. Avoiding Obstacles Using Probabilities (Particle Filters)
Robo doesn't always know exactly where it is because its sensors aren't perfect. It uses a method called a Particle Filter to track its position.

Think of Robo's knowledge of its position as many tiny guesses (particles). Each guess represents where Robo might be. As it moves and gets more sensor data (like seeing a wall or a chair), Robo updates these guesses, throwing out the bad ones and keeping the good ones. Over time, it gets a better estimate of its actual position.

1.5.2 Inventory Management

Inventory Management Example: Managing a Warehouse
Imagine a warehouse that stocks various products, like toys, electronics, and clothes. The goal of the warehouse manager is to keep the right amount of stock for each item while avoiding overstock (which wastes money) and understock (which leads to unhappy customers).

1. Understanding the Inventory State (Probabilistic Robotics)
Manager keeps track of every product in the warehouse using sensors, like barcodes and RFID tags, which scan items as they are stocked or sold. However, just like the robot's sensors in the navigation example, these inventory sensors might sometimes make mistakes (like missing a scan or double-counting).

To handle these uncertainties, Manager uses probabilistic models to estimate the actual stock levels. For example, if the system scanned 95 out of 100 items, it uses probabilities to estimate how many items might really be in stock. This idea is similar to Bayesian filtering in robotics, where the system constantly updates its belief based on new information.

2. Deciding How Much to Order (Artificial Intelligence & Markov Decision Process)

Manager needs to decide when and how much to order for each product. The decision depends on several factors:

- Current stock levels
- Expected customer demand
- Delivery time from suppliers

This decision-making process can be modeled using Markov Decision Process (MDP), just like in robot navigation:

- **States:** A state might represent the current stock levels of each product, like "50 toys in stock, 20 electronics, and 30 clothes."
- **Actions:** The actions are how many units to order (e.g., "Order 10 toys, 5 electronics").
- **Rewards:** The reward is based on the outcome. If Manager orders the right amount, the reward is high because customers are happy and money is saved. If it orders too much or too little, the reward is lower.

3. Learning from Experience (Reinforcement Learning)
Manager doesn't always know the perfect amount to order right away. Instead, it learns over time using Reinforcement Learning (RL).

At first, Manager might make random decisions ordering too much or too little. But over time, it learns:

- "If I order 20 toys when demand is high, I usually avoid stockouts (running out of stock)."
- "If I order too many electronics, they sit in the warehouse for too long, costing storage fees."

Manager uses these lessons to improve its decisions. Over time, it gets better at balancing stock levels to meet customer demand without overstocking.

4. Predicting Demand with Probabilities (Demand Forecasting)
One of the most important things in inventory management is predicting demand. Manager uses past sales data to estimate how many units will be needed in the future. However, like sensor data, sales data isn't always perfect (e.g., holidays can cause a sudden spike in demand).
Manager uses probabilistic models to predict demand. It calculates the probability of different demand scenarios, like:

- There's a 70% chance that customers will buy 50 toys this week.
- There's a 20% chance they'll buy 70 toys (if there's a promotion or holiday).

By predicting demand this way, Manager can make smarter ordering decisions.

1.5.3 Autonomous Vehicle Decision Making

Autonomous Vehicle Example: AutoDrive in the City
AutoDrive, the self-driving car, needs to safely navigate through a busy city. It has to handle other cars, pedestrians, traffic signals, and even unexpected events like a dog running into the street. Here's how it makes decisions:

1. Sensing the Environment (Probabilistic Robotics)
AutoDrive uses sensors like cameras, radars, and LIDAR (a laser-based sensor) to detect everything around it—cars, pedestrians, buildings, and road signs. However, sensors aren't perfect. Sometimes a tree's shadow might look like a pedestrian, or rain might make a car appear farther than it is.

To handle this uncertainty, AutoDrive uses Probabilistic Robotics. It doesn't just rely on one sensor reading—it combines multiple sensor data points and calculates the probability of what it's seeing. For example:

- "There's an 85% chance that object ahead is a pedestrian."
- "There's a 95% chance that's a stop sign."

This helps AutoDrive make better decisions, even when its sensors aren't 100% accurate.

2. Understanding the Situation: What's Happening Around AutoDrive? (Artificial Intelligence)
AutoDrive needs to understand the state of the world around it:

- How fast are other cars moving?
- Where are pedestrians?
- Are there traffic lights, and what color are they?

This is where Artificial Intelligence (AI) comes in. AutoDrive uses models to predict what's happening in its environment. It also looks at what might happen next. For example:

- "That car in the next lane is slowing down. It might be about to change lanes."
- "The pedestrian at the corner might cross the road when the light turns green."

AutoDrive uses these predictions to decide what to do next.

3. Deciding What to Do: Should AutoDrive Stop or Keep Going? (Markov Decision Process)
AutoDrive constantly makes decisions about what to do:

- Should it slow down or keep its speed?
- Should it change lanes to avoid a slow-moving car?

- Should it stop at the crosswalk for pedestrians?

These decisions are made using Markov Decision Process (MDP). AutoDrive breaks down each situation into states and actions:

- **States:** The current situation, like "AutoDrive is approaching a red light, with a pedestrian crossing the road."
- **Actions:** AutoDrive's possible moves, like "slow down," "stop," or "continue driving."

Each action has a reward:

- Stopping for a pedestrian earns a positive reward (safety is important!).
- Running a red light would give a big negative reward (dangerous and illegal).

AutoDrive chooses the action that maximizes its overall rewards, ensuring safe driving while getting to its destination efficiently.

4. Learning from Experience: Getting Better Over Time (Reinforcement Learning)
AutoDrive doesn't always know the perfect move in every situation at first. It learns over time through Reinforcement Learning (RL). For example, if AutoDrive tries to change lanes without enough space and ends up too close to another car, it gets a negative reward (unsafe). If it successfully avoids traffic by choosing the right lane at the right time, it gets a positive reward.

By repeating these actions and seeing the results, AutoDrive learns:

- "When traffic is heavy, it's better to stay in the current lane than to keep switching."
- "If I see a pedestrian about to cross, slowing down early is safer."

This learning allows AutoDrive to improve its driving decisions over time, just like a human driver becomes more experienced.

5. Handling Uncertainty: What If Things Are Unpredictable? (Probabilistic Decision Making)
The world isn't always predictable. AutoDrive might have to deal with sudden changes, like:

- A pedestrian stepping into the street unexpectedly.
- A car ahead slamming on the brakes.

To manage these unpredictable events, AutoDrive uses probability models. It knows that even though it can't predict everything, it can still estimate the likelihood of certain events. For example:

- "There's a 20% chance that a pedestrian will cross the road suddenly."
- "There's a 50% chance the car ahead will stop at the yellow light."

By considering these probabilities, AutoDrive can plan for unexpected events, like slowing down when there's a higher chance of something unexpected happening (e.g., near schools or busy intersections).

1.6 Case Study

1.6.1 Autonomous Robotics

'Slamcore Aware's Person-Detection Feature Drives Safety and Efficiency

One of the features of Slamcore Aware that is really grabbing the attention of intralogistics and manufacturing facility operators is its ability to detect people as they move among vehicles in busy environments.

Slamcore Aware, retrofitted to existing intralogistics vehicles – anything from forklifts to tuggers – gives that vehicle the ability to 'see' its surroundings. Not only is that valuable in providing real-time location information, but it also has a significant role in improving safety whilst enhancing efficiency.

Using sophisticated algorithms and AI, it can detect a person and distinguish it from any other obstacles. By using camera data, it can provide far richer and higher quality information of what is around a vehicle. It does this 'at the edge' with no need for an internet or cellular connection. Whilst LIDAR, UWB, and other existing location sensors can provide an accurate estimation of where objects are in relation to a vehicle, they do not autonomously identify what those objects are.

The data provided by Slamcore Aware can not only augment driver awareness systems by identifying the presence of pedestrians around a vehicle, but through APIs to suitable fleet management systems, will be shareable with other vehicles. In this way, drivers of blindsided vehicles (such as those approaching a tight corner) can be alerted to the position of people who are 'seen' by another vehicle.

Slamcore system can work alongside and complement other location and mapping technologies. Although autonomous mobile robots and guided vehicles (AMRs and AGVs) are increasingly present in some factories and warehouses, manually driven forklifts and similar vehicles remain the mainstay of the industry. Equipping the machines you already rely on with the ability to 'see' and identify people ensures that they can be fully integrated into a safe and optimized management system.

1.7 Challenges and Limitations

In planning under uncertainty, decision-making is complex because the environment or outcomes are unpredictable.

1. **Complexity and Scalability**
   - **High computational complexity**: Planning in uncertain environments requires evaluating a large number of state-action sequences, which can become computationally expensive, especially in Markov Decision Processes (MDPs) and Partially Observable MDPs (POMDPs).
   - **Curse of dimensionality**: As the number of states, actions, or observations increases, the space of possible outcomes grows exponentially, making it challenging to store and manage the necessary information.
   - **Curse of history**: In sequential decision-making, the number of action-observation histories grows exponentially, complicating optimization of decision sequences.

2. **Partial Observability**
   - **Incomplete information**: Agents often lack complete knowledge, requiring them to maintain and update a belief state—a probability distribution over possible states.
   - **Ambiguity in observations**: Noisy observations can lead to uncertainty about the agent's state, resulting in suboptimal decisions as states cannot be fully distinguished.

3. **Model Uncertainty**
   - **Inaccurate models**: Transition dynamics, rewards, and observations may not be fully known or accurate, making models less reliable for planning.
   - **Defining transition and reward functions**: Transition probabilities and rewards are often approximated, leading to potential inaccuracies in planning.

4. **Computational Resources and Real-Time Constraints**
   - **Real-time decision-making**: Applications requiring quick decisions (e.g., autonomous driving) are challenging due to the iterative nature of planning algorithms under uncertainty.
   - **Limited computational power**: In systems with limited resources, complex algorithms may be impractical.

5. **Exploration-Exploitation Dilemma**
   - **Balancing exploration and exploitation**: Agents must decide between exploring uncertain states to gather information and exploiting known states to maximize reward. Focusing too much on exploration may reduce short-term rewards, while too much exploitation may prevent discovery of better strategies.

6. Reward and Risk Trade-offs

- **Balancing short-term and long-term rewards**: Agents need to weigh immediate rewards against long-term outcomes. Unpredictable future rewards make this challenging.
- **Risk tolerance**: Different applications require varying levels of risk tolerance, balancing critical negative outcomes with optimal performance.

7. Scalability in Multi-Agent Systems

- **Coordination among agents**: In multi-agent environments, coordination increases complexity as agents' actions impact the overall state.
- **Communication constraints**: Limited communication prevents agents from sharing information, reducing coordination and effectiveness.

8. Evaluation and Validation

- **Difficulty in testing and validation**: Probabilistic outcomes make testing robustness and reliability challenging.
- **Lack of robust performance measures**: Defining consistent success metrics is difficult in uncertain environments where performance may vary across scenarios.

1.8 Recent Advances and Future Directions

1.8.1 Recent Advances

- **Deep Reinforcement Learning (DRL)**
  - Improved decision-making: DRL combines deep learning with reinforcement learning, allowing agents to learn policies directly from raw sensory data.
  - Success in complex environments: DRL has been applied successfully in complex tasks, such as autonomous driving, robotics, and games, where uncertainty is a key challenge.
- **Model-Based Reinforcement Learning (MBRL)**
  - Enhanced sample efficiency: MBRL improves efficiency by learning a model of the environment, allowing for planning via simulation.
  - Application in real-world tasks: MBRL is promising in robotics, where realistic simulations can reduce training costs.
- **Partially Observable MDPs (POMDPs) and Belief Space Planning**
  - Advances in algorithms: New approximation methods for solving POMDPs, like point-based methods, make them feasible for real-world use.

  - Improved belief representation: Modern belief space planning techniques enable more accurate tracking of partially observable states.

- **Probabilistic Graphical Models**
  - Flexible representation of uncertainty: Graphical models, like Bayesian networks, offer structured representations of dependencies in uncertain environments.
  - Integration with machine learning: Combining graphical models with machine learning enables dynamic updates based on observed data.
- **Risk-Aware and Robust Planning**
  - Risk sensitivity in planning: New frameworks consider risk tolerance, allowing for cautious or aggressive decisions as needed.
  - Robust optimization: Ensures reliable operation in uncertain environments by optimizing worst-case performance.
- **Multi-Agent Planning under Uncertainty**
  - Coordination mechanisms: Dec-POMDPs and mean-field approaches improve multi-agent coordination in uncertain settings.
  - Scalability improvements: Approximation algorithms and hierarchical frameworks allow for planning in larger multi-agent systems.

1.8.2 Future Directions

- **Integrating Human-AI Collaboration**
  - Shared autonomy: Algorithms for seamless human-AI collaboration, as in assistive robotics or shared vehicle control.
  - Interpretability and trust: Making decision-making processes interpretable to humans will enhance trust in systems.
- **Scalable and Real-Time Solutions**
  - Real-time decision-making: Developing algorithms for real-time planning, critical in applications like robotics and autonomous driving.
  - Scalability for large state spaces: Hierarchical reinforcement learning and distributed computing can help scale algorithms to larger spaces.
- **Combining Model-Free and Model-Based Methods**
  - Hybrid approaches: Combining model-free and model-based methods can create more efficient and adaptive planning algorithms.
  - Transfer learning in uncertain domains: Leveraging knowledge from related domains to improve learning in new environments.

- **Risk-Aware AI and Ethical Considerations**
  - Ethical AI in uncertain environments: Incorporating ethical considerations in high-stakes applications, like healthcare and defense.
  - Enhanced risk management: Building on risk-aware planning to manage long-term and catastrophic risks.

1.9 Conclusion

Planning under uncertainty is a complex field focused on making decisions in environments where outcomes are not predictable, often due to the inherent randomness of the environment or limited observability. This chapter covers foundational concepts, recent advances, challenges, and future directions within this area.

Initially, the distinction between deterministic and non-deterministic planning is explained, where deterministic approaches assume clear outcomes for each action, while non-deterministic planning considers a range of possible outcomes, each associated with a probability. Uncertainty in planning often arises from environmental variability, perceptual limitations, and stochastic outcomes, necessitating the use of probabilistic models such as Markov Decision Processes (MDPs) and Partially Observable MDPs (POMDPs). MDPs provide a framework where outcomes are probabilistic, yet the agent has full knowledge of the current state. In contrast, POMDPs extend this model by introducing partial observability, requiring agents to work with a belief state—a probability distribution over possible states.

The chapter further describes key algorithms in planning under uncertainty. Value Iteration and Policy Iteration are foundational methods, each iteratively updating the agent's decision-making policy to maximize expected rewards. For POMDPs, Belief State Space Search focuses on maintaining and updating belief states rather than actual states. Monte Carlo methods apply random sampling to estimate state or action values, useful when exact models of the environment are unavailable or difficult to obtain.

Real-world applications of these algorithms include robot navigation, inventory management, and autonomous vehicle control, all of which operate under significant uncertainty. However, several challenges hinder planning under uncertainty. High computational complexity, due to the need to evaluate numerous state-action scenarios, can make algorithms impractical to scale. Partial observability and model uncertainty further complicate planning, as agents must make decisions with incomplete or noisy information, often relying on approximations of state transitions and rewards. Additionally, balancing exploration and exploitation is difficult; agents must explore to learn about the environment while also exploiting known information to maximize reward. This trade-off is crucial but challenging, especially when balancing immediate rewards with long-term outcomes. Other complications include coordinating multiple agents in multi-agent environments and the difficulty of testing and validating solutions, as performance under uncertainty can vary widely across scenarios.

Recent advances have addressed some of these challenges. Deep Reinforcement Learning (DRL) combines deep learning with reinforcement learning, enabling agents to handle high-dimensional data directly from sensory inputs. Model-Based Reinforcement Learning (MBRL) improves sample efficiency by allowing agents to simulate future states before taking actions, which has proven especially useful in robotics. Probabilistic Graphical Models, such as Bayesian networks, offer structured representations of dependencies in uncertain environments, while risk-aware planning frameworks allow agents to adjust risk tolerance based on application needs. In multi-agent contexts, decentralized approaches, such as Decentralized POMDPs (Dec-POMDPs), have improved coordination, even when communication is limited.

Looking forward, the chapter ends by highlighting several promising directions. Human-AI collaboration and shared autonomy are expected to play a greater role, particularly in areas like assistive robotics, where agents can work alongside humans. Increasing algorithm scalability and improving real-time performance are also priorities, especially for applications like autonomous driving, where real-time decision-making is essential. Hybrid methods that combine model-free and model-based approaches hold promise for creating adaptive algorithms capable of handling both known and unknown dynamics. Finally, ethical considerations, particularly in high-stakes applications such as healthcare, will play a more significant role, with an emphasis on developing AI systems that incorporate long-term risk management and ethical decision-making.

Exercise

Question 1:

How does the concept of planning under uncertainty differ from deterministic planning, and what are the key computational models and techniques used to handle various sources of uncertainty in real-world systems, such as environmental, perceptual, and outcome uncertainties?

Question 2:

Illustrate with examples the importance of incorporating non-deterministic models and probabilistic approaches in fields like AI and robotics, and explain how the Theory of Computation contributes to the development of algorithms that manage such uncertainties effectively.

Question 3:

Imagine a disaster-relief robot deployed in a hazardous environment to search for survivors across a multi-room building. The robot's sensors are unreliable due to debris and smoke, leading to partial observability. The robot's goal is to maximize the number of rooms it searches successfully while avoiding hazardous areas, operating under uncertainties in movement, observation, and reward outcomes.

Quetion 4:

In a 5-armed bandit problem, you need to find the optimal arm to pull to maximize your reward. The reward distribution for each arm is unknown but follows a normal distribution with different means and variances. You are given a Greedy Action-Selection Strategy (always selecting the arm with the highest estimated reward). After 100 steps of pulling arms, the estimated rewards for each arm are: Arm 1: 1.5 Arm 2: 2.3 Arm 3: 1.8 Arm 4: 2.1 Arm 5: 2.0 The true reward values for each arm are unknown.
Tasks:

1. What is the next action according to the Greedy Action-Selection strategy?
2. How would using an ϵ-greedy strategy with $\epsilon = 0.1$ change the decision?

Question 5:

A robot is trying to estimate its position along a straight line (1D). It starts at position 0 with an initial velocity of 1 m/s. The robot uses a sensor to measure its position every second, but the sensor has Gaussian noise with a mean of 0 and standard deviation of 0.5 m. You are tasked to apply the Kalman Filter to estimate the robot's position at time t = 2 seconds, given the following sensor measurements: At t = 1 second: Measured position = 1.2 m At t = 2 seconds: Measured position = 1.9 m
Tasks:

1. Predict the position at t = 2 seconds before incorporating the measurement.
2. Update the position estimate after incorporating the measurement at t = 2 seconds.

Question 6:

Imagine an autonomous car, AutoDrive, driving on a straight road. The car can perform three actions: accelerate, maintain speed, or brake. The environment provides rewards based on the car's speed: The car should aim to stay at a safe speed (50–60 mph), which provides the highest reward. If the car goes over 70 mph, it gets a large negative reward (unsafe). If it drives too slowly (below 40 mph), it gets a small negative reward (inefficient). The car's speed transitions depend on its current speed and the chosen action: Accelerate: Increases speed by 10 mph. Maintain speed: Keeps the current speed. Brake: Decreases speed by 10 mph. The car starts at 50 mph, and you need to model its decision-making using a Markov Decision Process (MDP).
Tasks:

1. Define the states, actions, and reward function for this problem.
2. Given the following reward function, calculate the optimal policy using the rewards provided:
   - Speed 30 mph: reward = -2
   - Speed 40 mph: reward = -1
   - Speed 50 mph: reward = +5
   - Speed 60 mph: reward = +5
   - Speed 70 mph: reward = -10

Question 7:

Compare value iteration and policy iteration algorithms.

Question 8:

Explain how Monte Carlo methods and belief state space search address the challenges of planning under uncertainty. Discuss the differences in their approaches and highlight scenarios where each method might be preferable.

Solutions

Solution 1:

Planning under uncertainty differs from deterministic planning in that outcomes of actions are not always predictable. Deterministic planning uses algorithms like A* or depth-first search for known, controlled environments. Conversely, non-deterministic planning employs probabilistic models like MDPs and POMDPs to handle incomplete or dynamic information. Key sources of uncertainty include:

1. Environmental Uncertainty: Addressed using MDPs to model unpredictable conditions, such as changing weather for self-driving cars.
2. Perceptual Uncertainty: Managed using Bayesian networks or HMMs to deal with noisy sensor data, like imprecise GPS readings.
3. Outcome Uncertainty: Handled with probabilistic automata or Monte Carlo simulations for variability in results, such as robotic actions affected by mechanical inconsistencies.

The Theory of Computation introduces models like NFAs and NDTMs, helping design algorithms that efficiently handle multiple possible outcomes, making systems in AI and robotics more reliable and adaptable.

Solution 2:

Non-deterministic models and probabilistic approaches are vital in AI and robotics to manage the unpredictable nature of real-world environments. For example, self-driving cars use Markov Decision Processes (MDPs) to make decisions under uncertainty, such as adjusting speed based on the probability of a pedestrian crossing. Similarly, robots operating in unstructured settings—like a warehouse with moving obstacles—rely on Bayesian networks or Hidden Markov Models (HMMs) to process noisy and incomplete sensor data, allowing them to adapt to changes effectively.
The Theory of Computation (ToC) supports this by introducing concepts like non-deterministic finite automata (NFA) and non-deterministic Turing machines (NDTMs), which provide a theoretical basis for handling multiple possible outcomes. Additionally, randomized algorithms developed within ToC help systems explore numerous options efficiently, ensuring optimal decision-making even when faced with uncertainty. These computational models make AI and robotic systems more reliable and adaptive in complex, ever-changing environments.

Solution 3:

This problem can be understood using a concept called a **Partially Observable Markov Decision Process (POMDP)**, which helps when a robot can't see everything clearly in its environment.

In this example: - **States (S):** Each state represents different situations the robot might be in, such as a room with a survivor or a room with hazards. Due to smoke and debris, the robot doesn't know exactly which state it's in but has a general idea.

- **Actions (A):** The robot can take various actions: - Move to a new room - Turn to face a new direction - Use its sensors to check for survivors or dangers. Each action may not work perfectly due to obstacles.

- **Observations (O):** After each action, the robot receives some information about its surroundings. However, due to poor visibility, the information may not always be accurate.

- **Reward (R):** The robot earns points (rewards) for its actions: - A positive reward for finding a survivor - A negative reward if it enters a hazardous area.

- **Belief State:** Since the robot doesn't know exactly where it is, it maintains a "belief state" — a best guess of its position and situation based on observations so far.

In a POMDP, the robot uses its belief (its best guess about its state) to choose actions. The robot aims to pick actions that will help it find survivors while staying safe, even though it doesn't have a clear view of everything.

Challenges: Planning in this situation is difficult because: - The robot must keep updating its belief as it gathers new information. - It has to balance searching for survivors with avoiding hazards. - POMDPs involve complex calculations since the robot doesn't know exactly what will happen.

This approach helps the robot make smart decisions in a dangerous environment, even with limited information.

Solution 4:

1. Greedy Action-Selection: The greedy strategy chooses the arm with the highest estimated reward. In this case, Arm 2 has the highest estimated reward of 2.3, so the next action would be to pull Arm 2.

2. ϵ-greedy Strategy: In an ϵ-greedy strategy, with probability ϵ (0.1 or 10%), the agent selects a random arm (to explore new options), and with probability 1-ϵ (90%), it selects the arm with the highest estimated reward.

   - 90% chance: It will choose Arm 2 (greedy choice).
   - 10% chance: It will randomly pick any of the 5 arms, including those with lower rewards, to explore.

   The ϵ-greedy strategy allows the algorithm to balance exploitation (choosing the best-known arm) with exploration(trying other arms to gather more information).

Solution 5:

1. 1. Predict the Position (using the motion model):

   At $t = 1$ second, the robot's velocity is 1 m/s. The predicted position at $t = 2$ seconds is given by:

$$x_{\text{pred}} = x_1 + v \times (t_2 - t_1)$$

Substituting the values:

$$x_{\text{pred}} = 1.2 + 1 \times (2 - 1) = 2.2\,\text{m}$$

Thus, the predicted position at $t = 2$ seconds is $x_{\text{pred}} = \mathbf{2.2}\,\text{m}$.

2. Update the Position (using the Kalman Gain and the measurement at $t = 2$):

   The Kalman Gain, K, determines how much the new measurement should affect the estimate. For simplicity, assume $K = 0.8$.

   The measurement at $t = 2$ is $z = 1.9\,\text{m}$.

   The updated estimate is calculated using the formula:

$$x_{\text{update}} = x_{\text{pred}} + K \times (z - x_{\text{pred}})$$

   Substituting the values:

$$x_{\text{update}} = 2.2 + 0.8 \times (1.9 - 2.2) = 2.2 + 0.8 \times (-0.3) = 2.2 - 0.24 = 1.96\,\text{m}$$

   Thus, the updated position estimate at $t = 2$ seconds is $x_{\text{update}} = 1.96\,\text{m}$.

Solution 6:

1. Defining the MDP Components:
   - States: The current speed of the car. The possible states are the discrete speeds: 30 mph, 40 mph, 50 mph, 60 mph, 70 mph.
   - Actions: The actions available to AutoDrive are:
     - Accelerate: Increase speed by 10 mph.
     - Maintain Speed: Keep the same speed.
     - Brake: Decrease speed by 10 mph.
   - Reward Function: The reward is based on how safe or efficient the current speed is:
     - Speed 30 mph: reward = -2 (too slow, inefficient)
     - Speed 40 mph: reward = -1 (still slow)
     - Speed 50 mph: reward = +5 (ideal speed)
     - Speed 60 mph: reward = +5 (ideal speed)
     - Speed 70 mph: reward = -10 (too fast, unsafe)

2. Optimal Policy Calculation: To calculate the optimal policy, AutoDrive should aim to maximize its total reward by choosing the best action in each state. Let's analyze the best action for each speed:

   a) Speed 30 mph:

      - If AutoDrive accelerates: It goes to 40 mph (reward = -1).
      - If AutoDrive maintains speed: It stays at 30 mph (reward = -2).
      - If AutoDrive brakes: It would be stuck at 30 mph or go lower, which isn't allowed in this problem (considered as staying at 30 mph).

      Best Action: Accelerate to increase the reward (moving from -2 to -1).

   b) Speed 40 mph:

      - If AutoDrive accelerates: It goes to 50 mph (reward = +5).
      - If AutoDrive maintains speed: It stays at 40 mph (reward = -1).
      - If AutoDrive brakes: It goes to 30 mph (reward = -2).

      Best Action: Accelerate to reach the ideal speed (50 mph).

   c) Speed 50 mph:

      - If AutoDrive accelerates: It goes to 60 mph (reward = +5).
      - If AutoDrive maintains speed: It stays at 50 mph (reward = +5).
      - If AutoDrive brakes: It goes to 40 mph (reward = -1).

      Best Action: Maintain speed or accelerate (both give the same reward, +5).

   d) Speed 60 mph:

      - If AutoDrive accelerates: It goes to 70 mph (reward = -10).
      - If AutoDrive maintains speed: It stays at 60 mph (reward = +5).
      - If AutoDrive brakes: It goes to 50 mph (reward = +5).

      Best Action: Maintain speed or brake (both give the same reward, +5, but maintaining speed keeps the car at the optimal speed).

   e) Speed 70 mph:

      - If AutoDrive accelerates: The speed goes beyond 70 mph (which is not allowed), so it remains at 70 mph.
      - If AutoDrive maintains speed: It stays at 70 mph (reward = -10).
      - If AutoDrive brakes: It goes to 60 mph (reward = +5).

      Best Action: Brake to decrease speed and avoid the large negative reward.

Final Optimal Policy:

- At 30 mph: Accelerate to 40 mph.
- At 40 mph: Accelerate to 50 mph.

- At 50 mph: Maintain speed or accelerate (either keeps the car in the optimal range).
- At 60 mph: Maintain speed (staying in the optimal range).
- At 70 mph: Brake to reduce speed to a safer level.

Solution 7:

Value Iteration and Policy Iteration are both algorithms used in MDPs, but they have distinct approaches and use cases.

1. Approach

- **Value Iteration**: Focuses on finding the **optimal value function** by iteratively updating values for each state.
- **Policy Iteration**: Alternates between **policy evaluation** and **policy improvement**.

2. Convergence Process

- **Value Iteration**: Stops when the values reach a stable point.
- **Policy Iteration**: Stops when the policy stabilizes and no longer changes after improvement.

3. Efficiency

- **Value Iteration**: More efficient for large state spaces, as it updates values without explicitly evaluating a policy.
- **Policy Iteration**: Often faster in small to medium-sized MDPs.

4. Suitability for Large State Spaces

- **Value Iteration**: Better suited for larger or continuous state spaces where approximate solutions are acceptable.
- **Policy Iteration**: Less efficient for large state spaces due to the intensive policy evaluation step.

5. Implementation

- **Value Iteration**: The algorithm is conceptually simple, making it relatively straightforward to implement.
- **Policy Iteration**: The algorithm is complex. It requires implementation of policy evaluation and policy improvement steps function

Solution 8:

Monte Carlo methods and belief state space search are both critical tools in planning under uncertainty, but they tackle this challenge in different ways suited to various scenarios.

Monte Carlo methods use random sampling to estimate expected values by averaging rewards from simulated experiences. These methods are useful in large or continuous state spaces where exact computation of value functions (like in value or policy iteration) would be computationally infeasible. Monte Carlo methods are especially popular in environments where the model of transition probabilities is unknown, as they don't rely on having a full model. Instead, they can approximate optimal policies by exploring the state space through a balance of exploration (trying out new actions) and exploitation (choosing the best-known actions). Monte Carlo Tree Search (MCTS), for instance, is highly effective in applications like games, where the decision tree can be vast and dynamic, allowing MCTS to explore promising branches while pruning less likely ones.

Belief State Space Search is crucial for situations involving partial observability, such as in Partially Observable Markov Decision Processes (POMDPs). Here, instead of working with actual states, the agent maintains a belief state—a probability distribution representing its uncertainty about the true state. Belief state space search attempts to find the optimal policy by planning over this belief space. This approach can be more complex than Monte Carlo methods due to the continuous nature of belief spaces and the computational demands of updating belief states after each action. Algorithms like Point-Based Value Iteration (PBVI) address these complexities by only considering a representative set of belief points, which makes it feasible to approximate the optimal policy in environments where full belief state search would be computationally prohibitive.

Comparing the Two Approaches:

- **Monte Carlo methods** are advantageous in fully observable environments with large state spaces or unknown models, where sample-based learning from simulations can provide good policy approximations without the need for exhaustive state-space exploration.
- **Belief State Space Search** is necessary for partially observable environments where the agent's limited visibility requires maintaining and updating a belief over possible states to make informed decisions.

Use Cases:

- **Monte Carlo methods** are well-suited for games, large-scale simulations, and scenarios where model-free approaches are necessary.
- **Belief State Space Search** is essential in robotics and autonomous systems, where sensors may provide limited or noisy information, making it important to plan over belief distributions to account for uncertainty in perceptions.

Bibliography

[1] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1957.

[2] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, USA, 2012.

[3] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, USA, 1960.

[4] Kai Huang and Rajesh P. N. Rao. Neural models for partially observable decision-making under uncertainty. *Trends in Cognitive Sciences*, 20(11):941–949, 2016.

[5] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, 1998.

[6] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques.* MIT Press, Cambridge, MA, USA, 2009.

[7] Steven M. LaValle. *Planning Algorithms.* Cambridge University Press, Cambridge, UK, 2006.

[8] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, USA, 1994.

[9] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley-Interscience, Hoboken, NJ, USA, 2005.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[11] David Silver, Aja Huang, Chris J. Maddison, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA, 2nd edition, 2018.

[13] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics.* MIT Press, Cambridge, MA, 2005.