3en

# POMDPs for Continuous States and Observations for Robotics

Master-Thesis von Sanket Shinde aus Indien
Tag der Einreichung:

1. Gutachten: Prof. Dr. Gerhard Neumann.
2. Gutachten: Joni Pajarinen.

TECHNISCHE
UNIVERSITÄT
DARMSTADT

POMDPs for Continuous States and Observations for Robotics

Vorgelegte Master-Thesis von Sanket Shinde aus Indien

1. Gutachten: Prof. Dr. Gerhard Neumann.
2. Gutachten: Joni Pajarinen.

Tag der Einreichung:



# Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. April 2017

(Sanket Shinde)

# Abstract

Motion planning in uncertain and dynamic environments is an essential capability for autonomous robots. Partially observable Markov decision processes provide a principled mathematical framework to support decision-making problems [1] in such environments. Existing approaches, such as [2] [3] [4], provide approximate offline solutions to POMDP problems formulated with an assumption of a discrete state, observation and action space. The real world is, however, continuous and dynamic. Thus, the assumption of discrete spaces limits the ability of a learning algorithm to learn precise and robust control policies desirable in applications related to robotics. Owing to the dynamic nature of the environment in which robotic agents operate, an online planning algorithm that alternates between planning and execution is more desirable. To this end, we explore the existing online planning algorithm - POMCP[3], and extend it to account for continuous state and hybrid observations but discrete action spaces. We propose two approaches to enable the POMCP algorithm to handle continuous observations by partitioning the continuous observation space. In our first approach, we create equal partitions manually of predefined range of continuous values. In the second approach, we generate partitions during planning, on-the-fly. Partitioning the observation space in such a way encapsulates the existing POMCP algorithm from continuous observations, so that it can work seamlessly in its original framework. We specifically focus on a grasping problem in robotics by simulating a two dimensional partially observable environment housing a manipulator and a target body. We experiment with the performance of the extended POMCP algorithm to handle continuous states and hybrid observation spaces and compare it against that of a memory-less policy learned by imitation learning to solve the grasping problem. We present the results of our experiments with the approaches proposed.

# Acknowledgments

I thank Prof. Dr. Gerhard Neumann for giving me the opportunity to work on the topic. I am thankful to Joni for pointing me to relevant reading resources and discussions that we had on the topic. I also thank Patrick Emami for his guidance on programming the POMCP algorithm. I am also thankful to Martin Ritz for the support extended. Lastly, I thank Joni and Adrian for the valuable review comments.

# Contents

# Figures and Tables

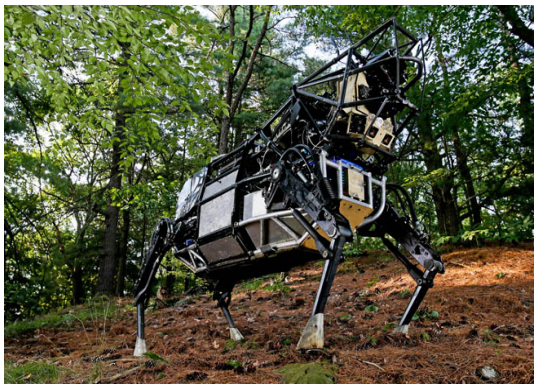
## List of Figures



## List of Tables

# 1 Introduction

We are surrounded by intelligent agents of all kinds all around us, from household items in the kitchen, as common as toasters, to home automation systems. Such agents may be programmed to do specific tasks in a controlled environment and achieve a predetermined acceptable result. The goal is predefined and the agent executes a predefined sequence of actions to reach it. Sequences of actions that guide the agent to reach its goal state are known as policies. A policy is a mapping between the agent's current state to an action. The notion of an action being optimal depends on the agent's current state. Therefore the knowledge of the state is a crucial piece of information when formulating a policy.



(a) LS3 from BostonDynamics [7]

(b) Dabo, the find-and-follow robot [8]

Figure 1.1.: (a) shows the quadruped robot developed by Boston Dynamics. It is capable of operating in dynamic, uncertain and rugged terrain as an assistant to humans. (b) shows the find-and-follow robot developed at CSIC-UPC Barcelona Spain [8] as a part of experiments for planning in continuous state spaces.

The optimality of an action is measured in terms of rewards/returns the agent receives on executing it from a given state. The goal of the agent is to take a series of actions to maximize the rewards it collects as it approaches the goal state. The agent must, therefore, make an optimal choice of action at every step of the decision-making problem. A decision-making problem where the agent's states are fully observable can be formulated as a Markov decision process (MDP). The goal of the planning algorithm is to trace a path for the agent to reach its target/goal. For this, it maps out a policy that maps the observed states to actions and in the process, maximizes some form of reward or minimizes a cost function. Formulation of a problem as an MDP is based on the assumption that the state is, in fact, fully observable. Finding solutions to MDPs is computationally less expensive when the size of the state space is small.

A level of complexity is added when the state is no longer fully observable. The planning algorithm now has to make a guess on the state, that the agent may be in based on the observations received. A decision-making problem where the state is not fully observable is formulated as a partially observable Markov decision process (POMDPs) and can be interpreted in the similar fashion as MDPs with some fundamental differences. Partial observability of the state gives rise to two other challenges/problems. The first being the curse of dimensionality. POMDPs use a belief state to model its belief over the agent's state. A belief state is a probability distribution over all possible states that an agent can be in. Consider that we have $N$ states, following this the belief state will be an $N$-dimensional vector of continuous values indicating the probability of an agent to lie in a particular state. The inherent limitation of such a belief state representation is, that it quickly gets out of hand as the size of the agent's state space increases. In addition, maintaining the belief state is difficult as it needs to be updated at every step of the planning process. The update process becomes cumbersome as the size of state space increases.

Representation of value functions is another challenge in POMDP planning. Existing solution algorithms represent the value function as a vector of values over the belief space, known as $\alpha$ vectors. These vectors are updated iteratively as the algorithm converges to a solution by a process known as backup operations [5]. These operations are particularly expensive when the $\alpha$ vectors have a high dimensionality.

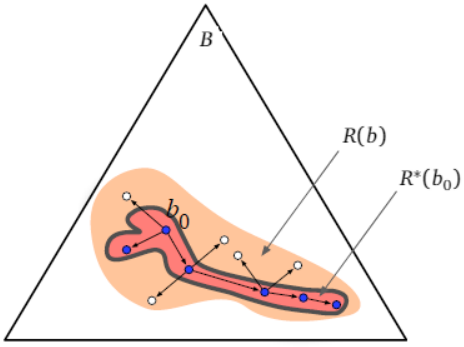



Figure 1.2.: Belief space planning in offline-approximate solver SARSOP[4]. $B$ is the belief space. $R(b)$ is the reachable space. $R^*(b_0)$ is the optimally reachable space. SARSOP focuses the search towards the reachable part of the search tree, thereby addressing the problem of Curse of History.

Another problem is that of the curse of history. A common approach followed by POMDP solvers is a tree search procedure to explore combinations of actions and observations. This approach is particularly limiting when the state and observation spaces are large. Exploring all action-observation combinations causes the search tree to grow wider as a consequence of a large branching factor. The planning algorithm, thus, has to evaluate all nodes in the search tree to derive an optimal solution. A large branching factor in trees demands voluminous computations which are particularly not suitable for planning online.

A great deal of research made in planning for agents, in particular relating to robotics [3] [2] [4], has been greatly limited to discrete state, action and observation spaces. The most influential factor in this is the ease of constructing a planning algorithm that works in limited spaces. Planning is easier in the discretized interpretation of the environment as it is computationally less expensive when compared to a continuous state space. A planning algorithm operating in a discrete space fails to scale to a continuous space that contains an infinite number of possible states that an agent can be in. This worsens the problem of the curse of dimensionality as the belief state will now be an infinite dimensional vector. A continuous state space induces an infinite number of locally optimal policies, making it infeasible to update them by value iteration. The limitations of existing algorithms to work with continuous state spaces extend further as the observation and action spaces also become continuous.

However, the real world is continuous. We live in an environment which flows seamlessly. It is uncertain and dynamic where most of the times, state information for an agent may not be fully available. Incorporating partial observability over states ushers in robust agent behavior, enabling it to plan optimally in an uncertain and dynamic environment. Robustness of a robotic agent brings in reliability. The discussion above is motivating to develop planning algorithms for robotic agents that plan in continuous state, actions and observation spaces.

## 1.1 Motivation

Solutions to decision-making problems formulated as POMDPs can be solved optimally over a finite horizon by value iteration. Value iteration is based on dynamic programming to compute an optimal value function over the belief space. Full-width planning algorithms attempt to find a solution to POMDP problems for the entire belief space. They typically work with a small to medium sized state space. Value function representations in full-width planning algorithms quickly fails to scale up when the size of the state space increases.

As an alternative, substantial research has been made on approximate point based algorithms that attempt to approximate the optimal value function over a subset of sampled belief points from the belief space. The key advantage here is the substantially smaller number of computations that are needed in backup operations, now that the value function is approximated by a small number of belief states. A compact representation of belief space is achieved by sampling belief points based on their reachability from the given belief state. Planning algorithms such as [2] [9] [4] have made a substantial progress in terms of finding approximate solutions to POMDPs. Figure 1.2 illustrates the idea of a reachable belief space for the SARSOP algorithm.

Although point based approximate solutions to POMDPs address the curse of history by approximating the value function only at selected belief points rather than the whole belief space, one key disadvantage with them is that they

still need to explicitly maintain probability transition models for states and observation dynamics. Computing such probability distribution is not straightforward and needs domain knowledge. Additionally, approximate solutions are computed offline and hence cannot be adapted optimally to a dynamically changing environment.

Based on the discussion above, it is clear that a planning algorithm may focus on the following key areas in order to converge to robust optimal policies.

- Curse of history
- Curse of dimensionality
- Representation of value function
- Representation of belief state
- Real-time adaptation to dynamic environments
- Maintaining state and observation transition dynamics
- Accommodation of continuous state, observation and action spaces

An online planning approach seems more plausible for solving policy quests in a dynamic and an uncertain environment. Although online planning algorithms provide a locally optimal solution, the key advantage is that they are able to adapt to a dynamic environment promptly as compared to offline approximate solutions. Following this argument, we explore the existing partially observable Monte Carlo planning (POMCP) algorithm. POMCP is an online planning algorithm for large POMDPs. It uses Monte Carlo tree search from current belief state and performs Monte Carlo updates on the agent's belief state. Figure 1.3 shows a Monte Carlo tree generated during the planning phase of POMCP algorithm. The key advantage of the POMCP algorithm is that it addresses both, the curse of dimensionality and history, unlike the previous approaches. It addresses the curse of dimensionality by using the power of Monte Carlo sampling. The belief state is represented using a particle filter containing $K$ particles, where $K << N$, $N$ being the number of possible states (infinite in the case of continuous state space). The belief state updates are made using rejection sampling and owing to the much compact, particle filter representation, are computationally tractable. The curse of history is tackled by focusing the search in the reachable part of the belief space, thereby reducing the branching factor. Limitations associated with finding an ideal representation of value functions is addressed by simply not maintaining any $\alpha$ vectors at all. Another notable feature of POMCP algorithm's approach is that it uses a black box simulator of the POMDP at hand to account for state transitions and observations generated, thereby eliminating the need for explicit probability distributions.

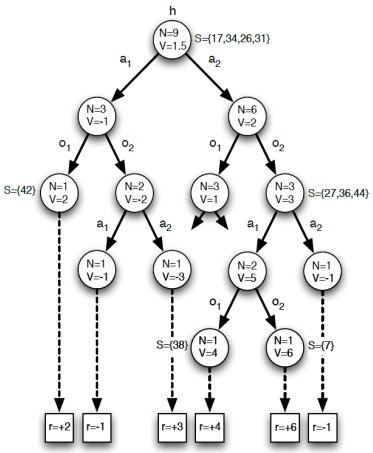



Figure 1.3.: Monte Carlo Tree Search in POMCP. Monte Carlo simulations are carried out from the history node $h$ for two actions and two observations. Circular nodes represent histories with corresponding values $V$ and visitation count $N$. The belief state is denoted by $S$. The accumulated long term reward for each simulation path is denoted by $r$ [3].

Owing to the features of the POMCP algorithm, it makes a great choice for solving policy quests in robotics. We research the existing POMCP algorithm and extend it to operate in the continuous state and observation spaces. We assume a discrete action space.

## 1.2 Outlook

The thesis report is arranged as follows, Chapter 2 provides a brief overview of related work. Chapter 3 covers the preliminaries on planning in partially observable spaces. Chapter 4 describes the methodology for adapting the POMCP algorithm to continuous state and observation spaces. Chapter 5 covers implementation details for adapting the POMCP algorithm to handle continuous state and observation spaces. In Chapter 6, we present the results of our experiments with the deterministic memory-less policy learned via imitation. Following this, we present the results of our experiments with the two proposed ways of handling the continuous observation space in the POMCP algorithm and compare it with the deterministic memory-less policy. Finally, in Chapter 7, we discuss our findings and possible future work.

# 2 Related work

POMDPs provide a principled mathematical framework for solving decision-making problems. The prowess of POMDP algorithms lies in the fact that decision-making problems are modeled with a partially observable state, which accounts for uncertainty in the decision-making process [1]. Incorporation of uncertainty adds to the robustness of the learning algorithm which is desirable in real world applications such as robotics. Although several successful attempts have been made in finding solutions to POMDPs, they have been limited in the context of the size of the state, observation and action spaces. It is computationally easier to solve problems formulated as POMDPs with discrete spaces. Such solutions can be found in tractable time and are accurate for small to medium sized state spaces.

POMDP algorithms attempt to evaluate the value of performing an action in a given state by using a search tree. The branching factor in the search tree is in direct proportion to the size of the state and observation space. Therefore, POMDP algorithms inherently suffer from the curse of dimensionality and history as the size of state and observation space increases.

Representation of value function and belief state is another important aspect of POMDP solution algorithms. Existing solution algorithms such as [4], [9] use a vectorial representation for the value function. A vectorial representation of the value function inherits the curse of dimensionality. Since the value function maintains values for states under the policy that it represents, the dimensionality of the value function vector increases with the increase in the dimensionality of the state space. Further, such a vectorial representation fails to scale when the state space in continuous. The same analogy is extended to belief state representations.

Much effort has been made to develop solution methodologies in this regard. We briefly summarize some recent methodologies to address the above-mentioned problems. In the following paragraphs, we refer to the action space as $A$, observation space as $O$ and state space as $S$.

## 2.1 Offline-Approximate Solvers

Offline-approximate POMDP solvers attempt to find solutions to POMDP problems offline, in the sense that the policy is constructed beforehand to fit for a wide range of agent's states. Value function approximations are made over a subset of the belief space, known as the reachable space. Offline-approximate solvers that follow such reduction in belief space dimensionality are referred to as point based algorithms. Dimensionality reduction in belief space is achieved by sampling the most probable next belief state, instead of having to search the entire belief space [2]. In summary, offline-approximate POMDP solvers address the curse of history and achieve belief space dimensionality reduction.

### 2.1.1 Heuristic Search Value Iteration (HSVI)

HSVI is an algorithm that finds approximate POMDP solutions by minimizing the uncertainty at a belief under consideration [9]. It constructs a tree of belief points that are explored based on forward heuristics. It performs value iteration updates at the most promising belief nodes to approximate an optimal policy. Bounds on the value function are initialized differently, in that the lower bound is represented using a set of alpha vectors, while the upper bound is represented by points $(b, V(b))$, where $b$ is the belief state and $V(b)$ is the value of the belief state. A forward search is carried out from the root node $b_0$ until the gap between the value function bounds is less than or equal to a predetermined threshold. The gap between the value function bounds represents the uncertainty at the belief node $b_0$. The child nodes to be explored are chosen based on a forward heuristic such as the IE-MAX heuristic for selection of optimal action $a^*$ at a node $b$ and an observation $o^*$ that contributes the most to the gap in the value function bounds at the root of the tree, at $b_0$. At every step, the value function bounds at the root node are updated.

### 2.1.2 Successive Approximation of Reachable Space under Optimal Policy (SARSOP)

The key idea of the SARSOP [4] algorithm is to approximate optimally reachable spaces through sampling. Like other point based algorithms, SARSOP also samples a set of belief points from the belief space. The sampled points

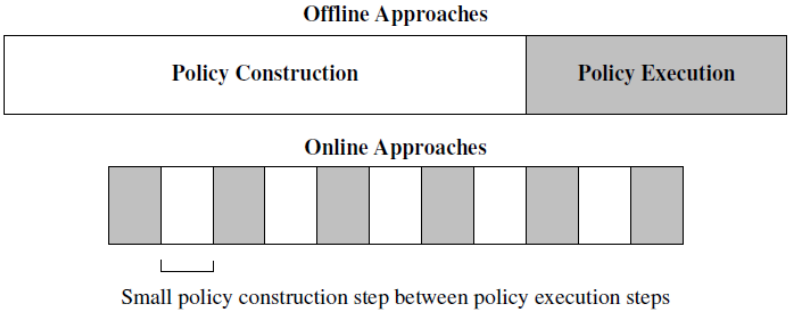



Figure 2.1.: Comparison between online and offline approaches to solving POMDPs [5].

from nodes of a tree. SARSOP approximates the value functions at the sampled points in the form of a piecewise linear set of $\alpha$ vectors. Starting from an initial belief $b$, SARSOP samples a new belief by choosing a suitable action $a \in A$ and observation $o \in$ O, based on suitable probability distributions. The key idea of the algorithm is to focus on promising nodes in terms of rewards. SARSOP uses upper and lower bounds on the value function to guide the learning process. To sample new belief points, SARSOP sets a target gap size $\epsilon$ between upper and lower bounds at the root node and traverses a single path down the tree. At each node, an action is chosen with the highest upper bound while the chosen observation is the one that makes the largest contribution to the gap at the root node of the tree. The process is similar to [10]. The algorithm terminates when the gap between the value function bounds is lower than a predetermined threshold.

### 2.1.3 Monte Carlo Value Iteration for Continuous State POMDPs (MCVI)

MCVI [11] for POMDPs is similar to SARSOP, in that it is also a point based algorithm. However, the key advantage here is that it uses Monte Carlo (MC) simulations to implicitly represent $\alpha$ functions. The result of the algorithm is a policy graph containing nodes that correspond to actions and edges to corresponding observations. The algorithm starts by sampling from the belief space and obtaining a subset of belief points $B$. Next, a backup operator performs backup operations at every point in B. The algorithm is made more efficient at this point by only considering those points from $B$, that are reachable under an optimal policy. This is also where the algorithm demonstrates its similarity to the SARSOP algorithm, the key difference being that it uses MC-backup and particle filtering to handle continuous state spaces. The algorithm performs MC-backups at belief points in $R \subseteq B$. These are the reachable belief points and by making updates at these points the algorithm gains computational efficiency. The points are sampled by constructing a tree of belief points in the same way as in the SARSOP algorithm. The advantage here is the ability of the algorithm to handle continuous state spaces.

## 2.2 Online Planning Algorithms

Offline-approximate methods of finding solutions to POMDPs have a limitation, that they take a considerable amount of computation time planning offline. Although the solution is computed over the entire belief space, it can be only found for POMDP problems with small to medium sized state spaces. The limitation to a small state space is due to the fact that offline-approximate methods use a vectorial representation of the value function whose dimensionality is governed by the size of the state space. Point based algorithms help alleviate this problem by maintaining value function vectors for a limited set of belief nodes, however, the solutions provided can be of low-quality [5]. The greatest advantage of online planning algorithms is that they gain computational advantage and hence speed, as they find a local solution rather than a solution generalized to the whole belief space. Online planning algorithms follow an approach where planning and execution are alternated at every time step. Alternating between these two steps enables online algorithms to adapt faster to a dynamic environment. Figure 2.1 illustrates the difference in planning structure between online and offline approaches.

Online planning algorithms follow a depth-first approach. They construct a tree of belief nodes. Trees are generated from a root belief node by sampling actions and observations. The value of an action is computed by the mean return of simulations, where a particular action was chosen at the root node. Such a tree can be seen in the Figure2.2. Following a depth-first approach, online planning algorithms work with ways to make smarter action and observation choices by using methods such as branch-and-bound pruning [12] and Monte Carlo sampling [3] [13]. Branch-and-bound pruning method maintains lower and upper bounds on values of each fringe node of the tree. Nodes that have

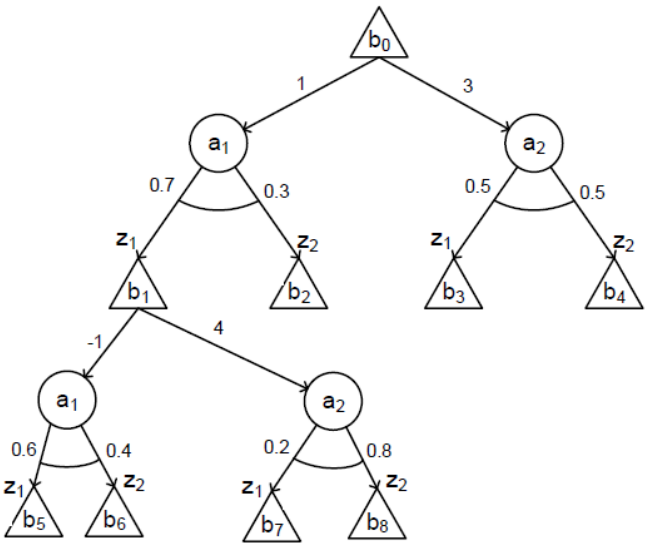



Figure 2.2.: Search tree for a POMDP with two actions and two observations. Triangular nodes denote belief states, and circular nodes denote action nodes. Edges from belief states denote the immediate reward associated with the corresponding action node. Edges from action nodes denote observation probabilities [5].

their value function upper bounds lower than the value function lower bounds of other fringe nodes are pruned out, thus limiting the tree width in the context of actions. A similar strategy is applied for reducing the branching factor at observation nodes by using Monte Carlo sampling as a generative model for observations, thus only highly probable observations are sampled, thereby reducing the branching at observation nodes. Employing pruning and sampling methods addresses the curse of history.

### 2.2.1 Heuristic Search

Online planning algorithms that employ heuristic search, reduce the branching complexity of search trees by limiting the search to a set of reachable belief nodes from the root belief node. The action and observation selection heuristics are similar to the offline planning algorithms such as [9] and [11]. Bounds are maintained on the values of belief states in the search tree. Actions chosen are the ones that maximize the value of the belief node and observations are chosen such that the chosen observation makes the greatest contribution to the gap in the bounds at the root node. Two algorithms that use heuristic search in online planning for POMDPs are [14] [15].

### 2.2.2 Monte Carlo Sampling

Reduction in the complexity of observation branching in forward, depth-first search trees in online planning algorithms for POMDPs is a challenging problem. Monte Carlo sampling uses a generative model to sample observations, rather than expanding all possible combinations of actions and observations. An expansion is carried out from a root node until a limited horizon depth $D$. Sampling alleviates the complexity of approximating the observation dynamics model associated with POMDPs [16]. Other online Monte Carlo approaches, such as the Rollout algorithm [13], greatly reduce the complexity in action selection. The Rollout algorithm randomly samples an action from a set of available actions at a root belief node and runs simulations starting from that node with that particular action being executed. The value of the corresponding action is estimated by the average of the returns of $N$ simulations.

## 2.3 Handling Continuous Observation Spaces

We discuss online planning algorithms that employ depth first, search trees to compute an optimal action. Complexity in search trees relating to actions and observations is controlled using approaches such as Heuristic Search and Monte Carlo Sampling. This greatly controls the breadth of the tree by pruning actions and observations that lead to suboptimal branches of the search tree. Existing pruning approaches work well when the branching factor at action and observation nodes is finite. However, the situation escalates when the observation spaces to be dealt with are

continuous, as now the branching factor at observation nodes is infinite.

Existing approaches, such as [17] [18] [19], are based on the idea that although the continuous observations are infinite, not every unique, continuous observation received from the real world corresponds to a unique policy. Multiple continuous observations may correspond to the same action plan. This induces a partitioning of the continuous observation space such that observations corresponding to a particular policy are grouped together. An attempt is made to find a partitioning of the observation space. Our approach for extending the existing POMCP algorithm to continuous observation spaces is based on this idea. We attempt to learn a partitioning of the continuous observation space and map continuous observations to discrete observation class labels.

# 3 Preliminaries

Consider a robotic manipulator in a room as shown in Figure 3.1. The manipulator can move in 3D space by turning its joints. To keep things simple, let us consider only the end effector location of the manipulator in the 3D space. Suppose that the goal of the robotic manipulator is to navigate the end effector to a stationary target body located somewhere on the floor of this room. It is assumed that the manipulator can navigate itself to any point in the room. The challenge here is that there are no visual cues available to the robot, it cannot see where the target body is located in the entire room. In addition, the manipulator also has no precise information about its own location in the room, nor the dimensions and the shape or appearance of the room that it is stationed in. What the manipulator does have, is a set of haptic sensors around its end effector to register world contacts. In addition, it also has a noisy distance sensor that provides a continuous valued distance reading between the end effector and a special reflector located on the target body. However, the compass of this direction sensor is broken and, therefore, the direction in which the distance is read is not known.

How does one go about solving such a decision-making problem? The manipulator has to make a series of decisions about the actions that it chooses to perform in order to reach the target. The easiest case would be the one in which the manipulator has all information of the world around it with minimal or no uncertainty. Such a scenario could be modeled as a Markov decision process and can be solved via dynamic programming. However, in the case of our manipulator, we do not have complete information about the state of the real world. The manipulator relies on noisy readings from a set of haptic sensors and a noisy distance sensor.

Thus, although the manipulator cannot see the target body for real, it can sense the environment around it and use this indirect information about the target's location to navigate its end effector to the desired target location. Planning in such a scenario is much like playing the game of 'treasure-hunt', where the player is given a series of hints about the location of the treasure and using this indirect information about the treasure, the player can navigate himself to exploit the bounty. This also is a decision-making problem and can be solved by mathematical formulation provided by the partially observable Markov decision processes.

In this chapter, we will cover the basics of decision-making problems using MDPs and POMDPs.

## 3.1 Introduction to Markov decision processes

Markov decision processes provide a mathematical framework to solve decision-making problems via dynamic programming. An MDP models the problem using the following parameters.

- State space S: Finite state space of the agent
- Action space A: Finite action space of the agent
- $T(s', s, a) = \Pr(s'|s, a)$: State transition probabilities that determine the agent's next state distribution $s'$, given an action $a$ was executed in the state $s$
- $R : f(s, a)$: Reward function that returns a real valued reward for executing an action $a$ in state $s$

In the manipulator problem scenario mentioned before, the most intuitive way to find the target would be to move in random directions and keep applying this strategy until the end effector touches the target. However, this may take an uncertain amount of steps and the uncertainty about reaching the target is high. If we were to associate a penalty to every wrong move we make in the process of reaching the target, then the above strategy would most probably receive a heavy penalty. In real world situations, such a penalty could come in the form of energy expenditure for autonomous robots where for every suboptimal action battery life is sacrificed. Therefore, a robotic agent may want to reduce penalties incurred or optimize returns on actions while it reaches its target. This concept brings in the idea of optimal behavior where the robotic agent takes a series of actions such that it has a minimum or no penalty and, therefore, maximizes its long-term reward. Such a behavior is analogous to an optimal policy.

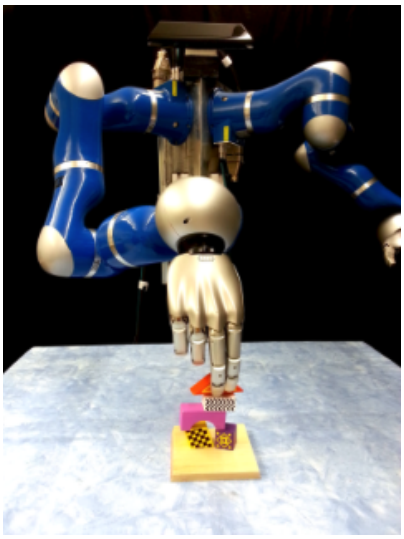


Figure 3.1.: Darius-bimanual manipulation platform from Intelligent Autonomous Systems lab, TU Darmstadt [6].

Markov decision processes model optimal behavior in the form of the expected long-term reward. Two common frameworks to model optimality are, first, finite-horizon optimality criterion,

$$\mathbb{E}\left[\sum_{t=0}^{k-1} r_t\right],$$

where $r_t$ is the reward received at the time step $t$. The goal of the agent is to maximize this expected long-term reward over the horizon length $k$.

Second, the infinite-horizon optimality criterion,

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right],$$

where the horizon $k = \infty$. Multiplication of $r_t$ with $\gamma^t$ reduces the significance of rewards received at a later point, thereby reducing their influence on the choice of actions at an earlier stage.

A notable difference between the two frameworks is that the finite-horizon optimality induces a non-stationary policy, which is time dependent whereas the infinite-horizon optimality induces a stationary policy, which is time independent.

Let us denote a policy as $\pi$. The t-step value of executing a policy $\pi$ from a state $s$ is given by,

$$V_{\pi,t}(s) = R(s, \pi_t(s)) + \gamma \sum_{s' \in S} T(s, \pi_t(s), s') V_{\pi,t-1}(s'),$$

where,

- $R(s, \pi_t(s))$: Immediate reward for executing the policy $\pi$ in state $s$.
- $\gamma$: Discount factor, also known as the planning horizon.
- $T(s, \pi_t(s), s')$: State transition dynamics model.
- $V_{\pi,t}(s)$: Value function at time step $t$.
- $V_{\pi,t-1}(s')$: Value function at time step, $t-1$.

In the infinite-horizon discounted case, the value $V_\pi(s)$ for executing a policy $\pi$ from state $s$ is given by,

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s'),$$

When found for each state $s$, $V_\pi(s)$ forms a set of linear equations, the solution to which is the value function $V_\pi$ for policy $\pi$.

Every policy induces a value function. The value function represents a measure of how good a particular choice of action $a$ is, under a certain policy $\pi$, for a given state $s$ of the agent. Since we desire to maximize our long-term rewards, an optimal policy would be the one that maximizes the value function,

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

Under such a policy we, choose the action that gives the highest returns.

Algorithm 1 summarizes value iteration.

```
t=1;
foreach s ∈ S do
 |  V_1(s) = 0
end
while |V_t(s) − V_{t−1}(s)| ≤ ϵ do
 |  t = t + 1;
 |  foreach s ∈ S do
 |  |  foreach a ∈ A do
 |  |  |  Q_t(s,a) = R(s,a) + γ Σ_{s'∈S} T(s,a,s') V_{t−1}(s')
 |  |  end
 |  |  V_t(s) = max_a Q_t(s,a)
 |  end
end
```

Algorithm 1: Value Iteration Algorithm

$Q_t(s,a)$ is the state-action value function. It is a measure of how good it is to perform an action $a$ in a state $s$.

## 3.2 Introduction to POMDPs

In the previous section, we saw how a decision-making problem could be modeled as an MDP to derive an optimal policy. It must be noted that an optimal policy can be derived using an MDP framework only when the state of the agent is fully observable. Consider our manipulator problem again. In the absence of any visual cues and availability of only haptic sensor data, combined with noisy distance sensor readings, the state of the target body can no longer be derived directly. The manipulator cannot clearly see the real state of the world. Instead, it attempts to predict the state of the world based on the noisy readings that it gets from the sensors at its disposal.

Owing to the partial observability of the state, one can no longer model such a decision-making problem using MDPs. POMDPs are generalised MDPs that handle partial observability of states. A POMDP can be described by,

- State space S: Finite state space of the agent
- Action space A: Finite action space of the agent
- $T(s', s, a) = \Pr(s'|s, a)$: State transition probabilities that determine the agent's next state distribution $s'$, given an action $a$ was executed in the state $s$
- $R : f(s, a)$: Reward function that returns a real valued reward for executing an action $a$, in the state $s$
- $O$: Observation space
- $Z(o, s', a)$: Observation function that gives a probability distribution $\Pr(o|s', a)$ over possible observations $o \in O$, given action $a$ was executed and the agent landed in the state $s'$

POMDPs, like MDPs, have the same variables such as $S, A, T, R$. The additional parameter here is the observation function $Z$ and the observation space$O$.

In the case of partial observability, the agent will have to know the entire history of actions and observations in the previous time steps in order to choose the best action in the current time step. This approach is memory intensive and, therefore, not scalable. In the case of MDPs, the optimal action is directly the function of the state that the agent is in. However, since the state is not fully observable in POMDPs, the agent must make a prediction about its state. This is modeled using a belief state $b(s)$. The belief state is a probability distribution over the all possible agent states. It encodes the belief of the agent about its own state. The belief state $b(s)$ is a sufficient statistic to account for the history of action and observations. Every time the agent executes an action, it has an effect on its state and hence its belief state needs to be updated. The observations received by the agent as a result of its action, indicate the changed state of the agent. It is clear, therefore, that the belief state is affected by previous observations. The belief state update follows Bayes' rule as shown below,

$$\begin{aligned} b'(s') &= \Pr(s' \mid o, a, b) \\ &= \frac{Pr(o \mid s', a, b) Pr(s' \mid a, b)}{Pr(o \mid a, b)} \\ &= \frac{Pr(o \mid s', a) \sum_{s \in S} Pr(s' \mid a, b, s) Pr(s \mid a, b)}{Pr(o \mid a, b)} \\ &= \frac{Z(s', a, o) \sum_{s \in S} T(s', a, s) b(s)}{Pr(o \mid a, b)}. \end{aligned}$$

The term $Pr(o \mid a, b)$ in the denominator acts as a normalizing term to ensure that the new belief state $b'$ lies between 0 and 1. A set of all possible belief states forms a belief space $B$.

### 3.2.1 Value Iteration in POMDPs

In POMDPs, the agent plans in a belief space $B$. The belief space is a space of all possible belief states that the agent can be in a partially observable domain. Analogous to the value function in the MDP case, the value function for a POMDP can be constructed as follows.

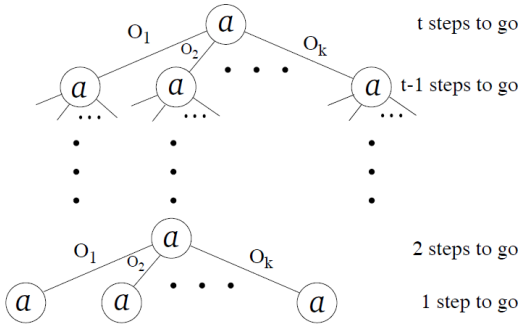



Figure 3.2.: A t-step policy tree capturing a sequence of t-steps conditioned on previous actions. Nodes are labelled with actions that should be taken when that node is reached. Figure borrowed from [20].

Consider a policy tree as shown in Figure 3.2. Consider a situation where an agent performs an action $a$ in a state $s$ and receives an observation $o$. The t-step value of executing action $a$ can now be written as,

$$V_\pi(s) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') \sum_{o \in O} Z(o, s', a) V_{o,\pi}(s'),$$

where the subscript $o$ denotes the sub-tree associated with observation $o$.

Since the state is not fully observable, we estimate the value of executing a policy $\pi$ from a given belief state $b(s)$ as,

$$\tilde{V}_\pi(b) = \sum_{s \in S} b(s) V_\pi(s).$$

The above equation can be compactly written as below,

$$\tilde{V}_{\pi}(b) = b \cdot \alpha,$$

where $\alpha_{\pi} = (V(s1), .....V(s_n))$ is a vector containing values for all states under a particular policy $\pi$ [20]. For a set $P$ of such policies, the t-step optimal value of starting in $b$ is the value of executing the best policy.

$$\tilde{V}_t(b) = \max_{\pi \in P} b \cdot \alpha_{\pi}.$$

The definition of $\tilde{V}_t(b)$ provides an intuition to the geometry of the value function. A policy $\pi$ induces an alpha vector $\alpha_{\pi}$. Since there could exist many policies to address the problem at hand, each of these policies will induce their own alpha vectors over the belief space. Figure 3.3 shows the alpha vectors over the one-dimensional belief space. Each of the alpha vectors is maximal in a subset of the belief space. Thus, corresponding to the highest value for the corresponding belief state. The optimal policy corresponds to the alpha vector that results in the largest dot product between the belief state and the alpha vector over that part of the belief space. Thus the optimal value function is the upper surface of the set of locally optimal alpha vectors. This is indicated by a bold line in Figure 3.3.

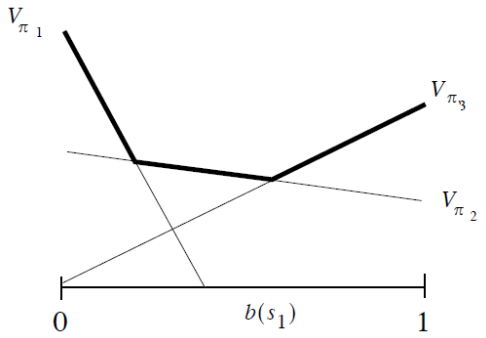



Figure 3.3.: Optimal value function represented as the upper surface of locally optimal value functions over the belief space. $V_{\pi 1}$, $V_{\pi 2}$, $V_{\pi 3}$ represent the locally optimal value functions. Figure borrowed from [20].

There are several ways to finding solutions to POMDPs. We briefly discuss the Witness algorithm [20] here. For the sake of simplicity of notation, we denote the t-step optimal value function as $V_t$. The construction of the optimal value function begins by constructing a set $Q_t^a$ of t-step policy trees with $a$ as the root node action. $V_t$ is constructed by pruning the set $Q_t^a$ of policy trees based on the idea demonstrated in the Figure 3.3. Mathematically, $Q_t^a$ can be expressed as

$$Q_t^a(b) = \sum_{s \in S} b(s) R(s,a) + \gamma \sum_{o \in O} \Pr(o \mid a, b) V_{t-1}(b'_o),$$

where $b'_o$ is the belief state resulting on executing action $a$ and observing $o$. $Q_t^a$ is the value of taking action a in belief state b and then continuing optimally for t-1 steps [20].

A set $Q_t^a$, of policy trees, is constructed for each action. The goal is to find a minimal set of policy trees for representing $Q_t^a$ for each action $a$. Consider a set $U_a$ of policy trees that ideally represents the optimal value function $V_t$. $U_a$ is initialized with a single policy tree with action $a$ at the root node. It is updated incrementally to include only a set of optimal policy trees from $Q_t^a$. The Witness algorithm proceeds by computing $Q_t^a(b)$ from $V_{t-1}$ and $\hat{Q}_t^a(b)$ from $U_a$. If there exists a belief state $b$, such that $Q_t^a(b) > \hat{Q}_t^a(b)$, a policy tree is constructed with action $a$ at the root that yields the best value at the belief $b$. This new policy tree is added to $U_a$. The process is repeated until for all belief states, the error $|V_t(b) - V_{t-1}(b)|$, is less than some $\epsilon$.

## 3.3 Online Planning in POMDPs

As discussed in the related work section, full-width offline solvers attempt to find solutions to POMDPs over the entire belief space. Although the solutions are accurate, they can only be found for a small to medium sized state space. Offline approximate solvers address this problem by using point based methods and find solutions over a finite number of points that are reachable from the starting state. However, they are constructed offline and cannot be dynamically adapted to the uncertain and dynamic real world. A robotic agent interacts with the real world in real time and, therefore, a real-time planning algorithm best suits the purpose.

Online planning alternates between planning and execution of the action in the real world. Thus, observations received after acting in the real world can be used to adapt the policy to optimize the long-term reward. We look at one such online planning algorithm - POMCP.

### 3.3.1 The POMCP Algorithm

POMCP is an online planning algorithm for large POMDPs [3]. Existing solution algorithms for POMDPs are limited mainly by the curse of dimensionality and the curse of history. Since a belief state has to be maintained for an estimate of the agent's state and has to be updated every time the agent performs an action, the update process becomes computationally expensive as the size of the state space increases. Consider that there are $N$ unique states that the agent can be in, then the belief state will be an $N$ dimensional vector of continuous values. It is intuitive to see that this becomes cumbersome when $N$ is large. This is known as the curse of dimensionality.

POMCP addresses the curse of dimensionality by using a particle filter representation of the belief state where all the states are uniformly weighted. Generally, there are $K$ particles where $K \ll N$. Thus, the process of updating and maintaining such a compactly represented belief state is computationally more efficient.

Offline full-width planners attempt to find a solution for all possible combination of states, actions, and observations. Thus, they suffer from what is known as the curse of history. The POMCP algorithm addresses this problem by exploring only those action-observation paths in the search tree, that return the highest reward. Thus, focusing the search in the reachable part of the search tree that leads to computational efficiency.

Apart from the mentioned limitations, offline full width and offline-approximate solvers have one thwarting limitation that they need to maintain distributions such as $T(s',s,a)$ and $Z(o,s',a)$. Modeling such distributions is a complex task. The POMCP algorithm addresses this problem by performing Monte Carlo simulations using a black box simulator.

#### Monte Carlo Tree Search

Online POMDP algorithms use a search tree method for deriving an optimal action at a given point in time. Such a search tree was shown in Figure 1.3. The search tree consists of nodes that represent the belief states. An agent has a belief state at any given point in time. This belief state forms the root of the tree and is represented by the highest node in the tree. At any given belief state, the agent has a set of valid actions that it can take. The choice of an action leads to an intermediate node from where the tree will branch out depending on the real world observation that the agent receives. Having received an observation, the agent now updates its belief state and a new node is reached from where the search process is carried out again.

General search tree methodologies followed in full-width planning try to find a solution for all possible combinations of actions and observations, thus making the tree wide as a result of a high branching factor. Such solution methodologies do not scale with the increase in the size of the state and the observation spaces and are hence not ideal for large POMDPs. Monte Carlo tree search (MCTS) adapts the search methodology from offline-approximate point based solvers. MCTS focuses the search on the reachable part of the belief space $B$. Belief nodes are sampled based on their reachability from the root node [2] [4]. In the context of the POMCP algorithm, MCTS is used in the planning phase to simulate trajectories of states and actions. Simulations start from the belief state at the root of the tree. Actions are sampled either using the Rollout [13] or the UCT [3] algorithm applied to trees. Unlike offline-approximate solvers, the POMCP algorithm samples observations from a black box simulator of the POMDP problem. The advantage of doing that is twofold. First, we do not need to maintain complex distributions for observation dynamics. Second, that this approach circumvents the problem of curse of history, since now only the most probable observations will

be sampled. This leads to a computational advantage as we no longer need to plan for all possible combinations of actions and observations. A simulation trajectory ends when either a terminal state is reached or a predefined planning horizon depth is reached.

At the end of the planning phase, an average long-term reward for every possible action tried out from the root node, is calculated by averaging the returns from a trajectory for which the corresponding action was selected. The action chosen is the one that corresponds to the highest average returns. This approach greedily looks only for actions that return immediate high rewards. However, it may be possible that actions that appear to be suboptimal initially can prove to return larger rewards in the long run. Thus, here lies the classic dilemma of exploration vs. exploitation. The POMCP algorithm addresses this using the partially observable upper confidence bound applied to trees (PO-UCT) [3] that factors in the number of times an action has been chosen from a belief state under exploration. It adds a penalty to the final reward for choosing the same actions repeatedly for a given action. The algorithm is explained in more detail in the following subsection.

After having selected the optimal action, the agent now transits into the execution phase and executes the action in the real world. The real world now returns an observation to the agent. The agent now updates the belief state using rejection sampling. This is depicted in Figure 3.4.

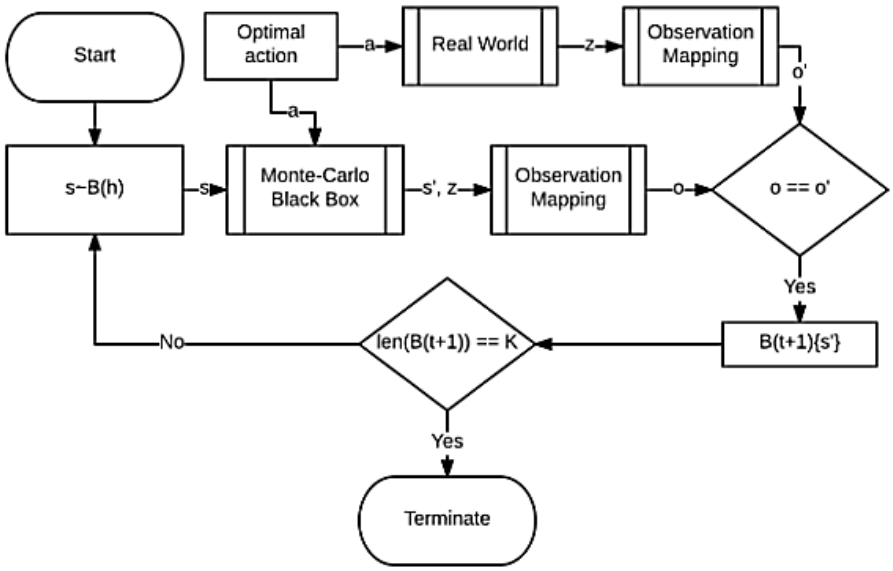



Figure 3.4.: Belief state update in the POMCP algorithm.

The process begins by sampling a state $s$ from the current time step belief state $b_t$. This state is now passed to the black box simulator, which returns a new state $s'$, observation $o$ and a reward. This observation $o$ is compared with the observation $o'$ received from the real world. If the observations match, then the next state $s'$ is added to the updated belief state $b_{t+1}$. The process is repeated until $K$ particles have been added to $b_{t+1}$. Search is now carried out from the new belief state $b_{t+1}$ and the process repeats until the agent reaches its goal.

### 3.3.2 Rollouts

One of the two ways mentioned previously, used to sample actions in the planning phase for MCTS in POMCP algorithm, is Rollout. The POMCP algorithm generates, incrementally, a tree of beliefs. For all the belief nodes in the tree, there exists a statistic for the long term reward for executing a given action. Thus, for such beliefs, the POMCP algorithm uses the partially observable upper confidence bound (PO-UCT) algorithm to choose the optimal action. A Rollout policy is adopted in MCTS for beliefs that are not already encountered in the tree. Since there is no statistical data available for such beliefs to choose an optimal action, the Rollout policy is implemented to sample valid actions uniformly at random from a belief state under exploration to generate a history sequence. This approach defines the behavior of the agent when it transitions to an unexplored belief. It also consolidates statistics for the belief under exploration. The next time the agent transitions into the same belief state, the PO-UCT algorithm is

used instead to select an optimal action. In the implementations of POMCP algorithm done as a part of the thesis, we enhance the Rollout policy by randomly selecting from a pool of valid actions.

### 3.3.3 Partially Observable Upper Confidence Bounds (PO-UCT)

The PO-UCT algorithm addresses the exploration vs. exploitation dilemma in MCTS [21] [22] [3]. In POMCP, we construct a tree of histories instead of states as is the case with MDPs. For each history node $h$, the algorithm maintains a tuple mentioning the value $V(h)$ for the history with the corresponding action $a$ and the number $N(h,a)$ of times the simulations were carried out from the history node using action $a$. The visitation count $N(h)$ for the history $h$ is given as $\sum_{a \in A} N(h,a)$. The choice of action is made corresponding to the largest augmented value calculated for each action as

$$V^{+}(h) = V(h) + c\sqrt{\frac{\log(N(h))}{N(h,a)}},$$

where $c$ is a tuning constant. Algorithm 2 provides an overview of the framework of the POMCP algorithm.

Algorithm `Search`(b)
- Input: Belief b
- while enough time is available do
  - s $\sim$ b;
  - `Simulate`(Ø, s, 0);
- return $\arg\max_a V(a)$;

Procedure `Rollout`(s, d)
- Input: State s, Depth d
- r = 0;
- discount = 1;
- while $d <$ horizon do
  - (s', r') $\sim$ G(s, random action);
  - r = r + $\gamma\times$ r';
  - s = s';
- return r;

Procedure `Simulate`(h, s, d)
- Input: History h, State s, Depth d
- select action $a$ with UCT;
- $(s', o, r) \sim$ G$(s, a)$;
- if $d <$ horizon then
  - if T$(h, a, o)$ is not Ø then
    - futureRew = `Simulate`$((h, a, o), s', d+1)$;
  - else
    - initialize T$(h, a, o)$;
    - futureRew = `Rollout`(s', d+1);
  - r = r + $\gamma\times$ futureRew;
- B(h) = B(h) $\bigcup \{s'\}$;
- N$(h, a)$ = N$(h, a)$ + 1;
- V$(h)$ += $\frac{r - V(h)}{N(h,a)}$ update value for action;
- return r;

Algorithm 2: Partially Observable Monte Carlo Planning

G is the black box simulator of POMDP problem at hand. T(h, a, o) is the search tree.

# 4 Accommodating Continuous States and Observations

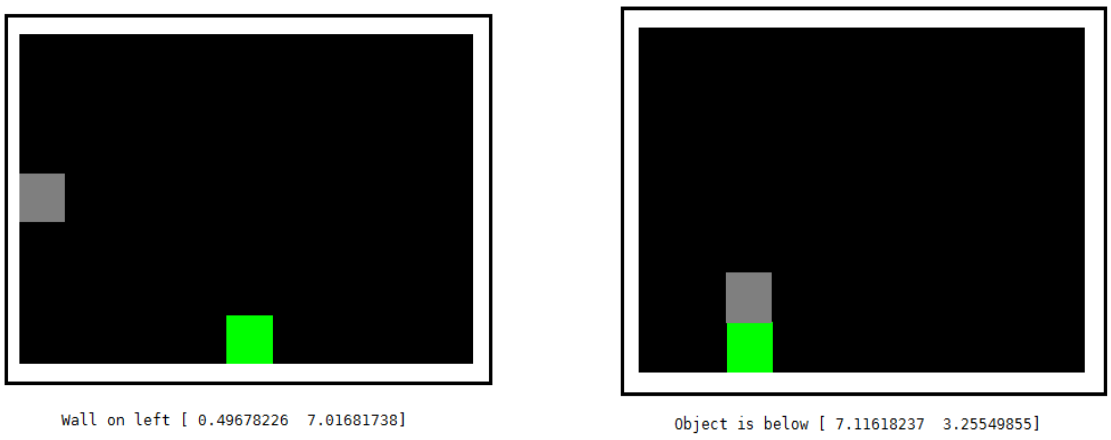
We investigate a grasping problem in robotics. Following that, we setup a 2 dimensional partially observable environment housing a robotic manipulator and a target body. The manipulator has a set of haptic sensors on its end effector along with a noisy distance sensor. The distance sensor provides information about the distance between the end effector and target body, but no information about the direction in which the distance was measured. It is assumed that the target location is initialised randomly along the floor. The manipulator uses information from its sensors to navigate its end effector to the target body. Figure 4.1 shows the environment described. The manipulator end effector and target are illustrated as gray and green boxes respectively. The chapter is organized as follows, in Section 4.1, we start off by describing an intuitive imitation learning approach to achieve the task mentioned above. In Section 4.2, we motivate the purpose of our work with the POMCP algorithm. Sections 4.3 and 4.4 describe our proposed approaches to accommodate continuous observations in the POMCP algorithm. Section 4.5 illustrates the incorporation of our proposed methodologies in the POMCP algorithm framework. Finally, in Section 4.6 we describe a method to focus search in search trees by sampling the most suitable actions using the idea of Thompson sampling.

## 4.1 Learning a Policy by Imitation

The most intuitive way to learn something is to learn by imitation. Children learn several skills by imitating adults. Following this, one of the early approaches that we explored, was applying imitation learning to learn a policy. We set up the simulator environment, where a user is able to navigate the end effector of the manipulator in the simulation using the left, right, up, down keys from a keyboard. The scenario is similar to the one in which a visually impaired individual would try to navigate themselves by painting a picture of the surrounding by touching objects in the environment.

The user has no visual information as to where the end effector is in relation to the target or the room. However, every time the user moves the end effector, he receives a discrete observation that conveys indirect information about the end effector's state in the environment. This is illustrated in Figure 4.1.

Thus, although there is no direct visual information of the target location, the user uses the indirect information about end effector's state in the world from the observations received, to navigate it to the target location. We recorded trajectories from several gameplays from a group of 50 users. The recorded trajectories were used as reference data for the learning process. We assume a discrete observation and action space for the imitation learning approach.




(a) (b)

Figure 4.1.: (a) illustrates a configuration where the end effector is next to a wall. The corresponding observation received is displayed below along with the noisy 2D position of the end effector. Likewise (b) shows a configuration where the end effector is on top of the target and corresponding observation received.

### 4.1.1 Learning Observation-Action Mapping

During the gameplay, the users perform actions based on the observation received at the previous time step. Following this, we attempted to learn deterministic observation-action mappings to formulate a policy. We attempt to learn observation-action mappings on two sets of training data. First, a training data set that contains a noisy reading of the position of the manipulator end effector, $x_H, y_H$, and also the noisy position of the target body, $x_t, y_t$. In addition, we also have the haptic sensor data $h$ and the discrete distance sensor reading $d$ as features. We refer to this dataset as full-observability data since, ideally for our problem, the target position information is not available. Second, a training data set that contains only the noisy readings of the position of the manipulator end effector $x_H, y_H$ in addition to the haptic sensor data $h$ and the discrete distance sensor reading $d$ as features. We refer to this dataset as partial-observability data. We explore whether the additional target position information helps build a better deterministic observation-action mapping. We train three prediction models, namely, multi-class support vector machine-based classifier, stochastic gradient based multi-class classifier and random forest based classifier models, on the two training data sets generated during game play. The prediction label for all the models is the corresponding action that resulted in the recorded observations. An excerpt of the training data we recorded is shown in Table 4.1 and Table 4.2.

| $x_H$ | $y_H$ | $x_t$ | $y_t$ | h | d | a |
|---|---|---|---|---|---|---|
| 16.98 | 7.01 | 19.0 | 3.01 | 7.0 | 1.0 | 3.0 |
| 16.98 | 5.01 | 19.0 | 3.01 | 7.0 | 0.0 | 0.0 |
| 16.98 | 3.01 | 19.0 | 3.01 | 7.0 | 1.0 | 0.0 |
| 3.01 | 9.00 | 5.0 | 3.01 | 7.0 | 1.0 | 2.0 |
| 3.01 | 7.00 | 5.0 | 3.01 | 7.0 | 0.0 | 0.0 |

Table 4.1.: Sample full-observability training data set.

| $x_H$ | $y_H$ | h | d | a |
|---|---|---|---|---|
| 5.00 | 3.01 | 7.0 | 1.0 | 0.0 |
| 5.00 | 3.01 | 7.0 | 1.0 | 0.0 |
| 3.01 | 3.01 | 4.0 | 0.0 | 3.0 |
| 5.01 | 3.01 | 7.0 | 0.0 | 1.0 |
| 7.01 | 3.01 | 7.0 | 1.0 | 1.0 |

Table 4.2.: Sample partial-observability data set.

The trained classifier models mapped recorded positions and observations to action labels. The manipulator used the learned classifier to predict an action based on the observation received in an attempt to navigate towards the target.

We experiment with the performance of this approach with the three specified models. It was observed that the learned deterministic observation-action mapping failed to navigate the end effector to the target in the greater majority of test runs. A deterministic mapping does not take into consideration the stochastic nature of our problem and has a tendency of getting trapped in deterministic loops. Details of the outcomes of our experiments with the approach have been discussed in greater detail in Chapter 6.

## 4.2 POMCP Algorithm Approach

Since the deterministic observation-action mapping of the imitation learning approach fails to account for the stochastic nature of our manipulator problem, we explore the formulation of the problem as a POMDP. In the light of the advantages of the POMCP algorithm discussed in the previous chapter, we use it to find a policy to navigate the manipulator to the target. The POMCP algorithm, in its original framework, is capable of planning in discrete state and observation spaces. We make attempts to adapt the algorithm to accommodate continuous states and observations.

### 4.2.1 Adapting to Continuous States

The POMCP algorithm is inherently capable of handling large state spaces. This ability comes from the algorithm's approach of using a compact particle filter based representation of the belief state. The belief state is represented by $K$ particles, where $K \ll N$, $N$ being the total number of possible agent states. This representation of the belief state is efficient compared to the vectorial representation since by choosing $K$ particles, we compress the belief state dimensionality. We, therefore, use the particle filter based belief state representation of the POMCP algorithm and extend it to accommodate continuous state spaces by using continuous 2D co-ordinates in the particle filter instead of discrete state particles as used in the original version.

### 4.2.2 Adapting to Continuous Observations

POMCP algorithm is capable of handling a large observation space as it uses Monte-Carlo simulations to sample trajectories during simulations [3]. By using a black box simulator of the POMDP problem at hand, it explores only the most probable observation branches.

Despite these advantages, the algorithm inherently fails to accommodate continuous observations. For continuous observations, the branching factor will now be infinite and planning, online, over such a large observation space is infeasible. Another reason for which the POMCP algorithm fails at accommodating continuous observations is because of the mechanism it uses to update the agent's belief state. The belief state update process in POMCP algorithm was introduced in Chapter 2. Let us have a look at it here once again to explain the limitation mentioned. Algorithm 3 illustrates the belief state update process.

```
Algorithm BeliefUpdate(b, a, o')
    Input: Belief b, Action a, Real world observation o'
    while number of particles in b_{t+1} is less than K do
        s ~ b;
        s', o, r ~ Simulator( s, a);
        if o = o' then
            b_{t+1} <- s'
```

Algorithm 3: Belief state update in POMCP algorithm

Belief state updates in POMCP algorithm are made by rejection sampling. The observation $o$ received from the black box simulator is compared to the observation $o'$ received from the real world. If the observations match, it is concluded that the agent may have transitioned in the sampled state $s'$ with a high probability. Therefore the particle $s'$ is added to the updated belief state.

It is clear that this approach fails as the observation space becomes continuous, since the probability that a randomly sampled continuous observation matches the real world continuous observation is low. The immediate consequence of this, is that the belief state update process now takes an uncertain amount of time to converge, therefore rendering the algorithm unusable for online planning.

The choice of actions that an agent performs is influenced by observations it receives from the real world. However, not all observations received correspond to a unique action plan. It happens quite often, that multiple observations result in the same action plan [17] [19]. This causes an inherent partitioning of the observation space, where observations belonging to the same action plan are grouped together.

We attempt to find a grouping of continuous observations to meaningful clusters. Each of the clusters is identified by a unique observation class label. Thus by partitioning the continuous observation space to predefined observation class labels, we encapsulate the POMCP algorithm from the underlying continuous observation space. This enables the POMCP algorithm to work in its original framework.

We propose two approaches to cluster continuous observations. In the following sections, we introduce our observation classification methodologies and demonstrate how it is incorporated in the existing POMCP algorithm.

## 4.3 Partitioning the Observation Space by Hand

Our first attempt to partition the continuous observation space was to generate partitions by hand. The continuous observation that we deal with is the noisy continuous distance sensor reading that the manipulator receives. This one-dimensional continuous observation space is partitioned linearly into equal ranges of continuous values controlled by a pre-specified range parameter $m$. Each of these range of continuous values represents a class of observation label $o_t$. A continuous observation $o_c$ received during planning is allotted to one of these partitions if it lies in the range of continuous values spanned by the observation class label. This is illustrated in Figure 4.2. We experiment with different values of $m$ to partition the continuous observation space. Algorithm 4 illustrates the process of partitioning the observation space.

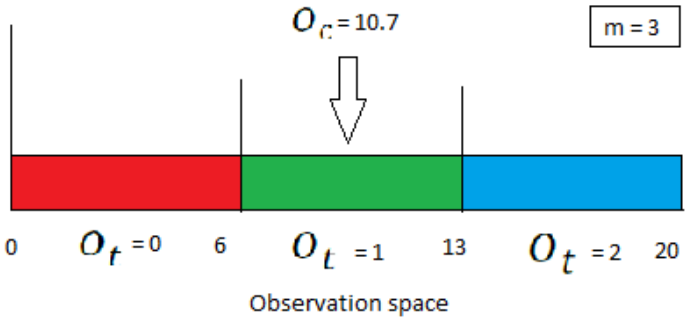



Figure 4.2.: Partitions of continuous, one -dimensional observation space for range parameter $m = 3$. The observation space is partitioned into three equal partitions labelled 0, 1, 2 respectively. An incoming continuous observation $o_c = 10.7$ lies in the range labelled $o_t = 1$, hence the observation $o_c$ is allotted a discrete observation class label $o_t = 1$.

Algorithm `PredictLabel`($o_c$)
  Input: Continuous real world observation $o_c$
  if $o_c \in$ one of the $m$ ranges then
    return $o_t$;

Algorithm 4: Equal partitioning of the observation space.

## 4.4 Partitioning Observation Space On-the-Fly

In this approach we generate partitions of the continuous observation space as in the previous approach, however, this time the observation class labels are generated on-the-fly. The motivation behind the approach is to investigate if partitioning the continuous observation space in this manner would result in better observation class clusters than the hand made partitions. These labels span a range of continuous values controlled by a bandwidth parameter $q$. A continuous distance sensor observation is allotted a discrete observation label if it lies in the range of continuous values spanned by the observation class label. The key difference with this approach from the one discussed earlier is that the generated partitions do not span the whole observation space. Partitions of pre-specified bandwidth are generated only when a suitable partition does not exist. This is illustrated in Algorithm 5 and Figure 4.3.

Algorithm `PredictLabel`($o_c$)
  Input: Continuous real world observation $o_c$
  if $o_c \in$ L-existing labels then
    return $o_t$;
  else
    $o_t = [o_c - q,\ o_c + q]$;
    L <- $o_t$;
    return $o_t$;

Algorithm 5: on-the-fly observation class label prediction.

The agent receives a continuous observation $o_c$ from the real world after performing an action. The algorithm checks if there is an observation class label, in a set $L$ of class labels, existing already, that spans the range of continuous values in which the incoming continuous observation lies. If there is one, then the corresponding observation class label $o_t$ is returned. However, if there is no matching observation class label, then the algorithm generates a new observation class label spanning a range of values such that the lower extreme of this range is $o_c - q$ and the upper extreme is $o_c + q$. The newly generated observation class label is stored in the set $L$ and returned to the planning algorithm.

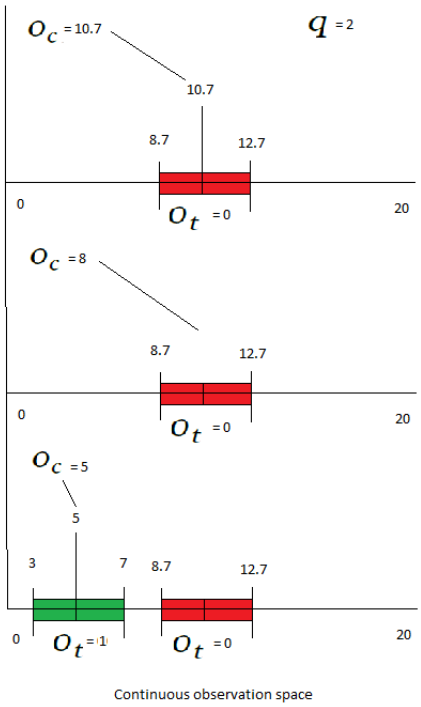



Figure 4.3.: Partitions of continuous, one -dimensional observation space for bandwidth parameter $q = 2$. At the beginning, since there are no partitions, a new partition $o_t = 0$ is generated and $o_c = 10.7$ is allotted to that partition. $o_c = 8$ is allotted to partition $o_t = 0$. As $o_c = 5$ does not lie in the range of $o_t = 0$, a new partition $o_t = 1$ is generated and $o_c = 5$ is allotted to that partition.

## 4.5 Integration Of Observation Prediction Methodologies In POMCP

We demonstrate the incorporation of our proposed methodologies in the POMCP algorithm. Either of the PredictLabel() methods described above can be used to accommodate continuous observations in the existing POMCP algorithm. Algorithm 6 illustrates this incorporation.

Algorithm `Search`(b)
- Input: Belief b
- while enough time is available do
  - s $\sim$ b;
  - `Simulate`(Ø, s, 0);
- return $\arg\max_a V(a)$;

Procedure `Rollout`(s, d)
- Input: State s, Depth d
- r = 0;
- discount = 1;
- while $d <$ horizon do
  - (s', r') $\sim$ G(s, random action);
  - r = r + $\gamma \times r'$ $s = s'$
- return r;

Procedure `Simulate`(h, s, d)
- Input: History h, State s, Depth d
- select action $a$ with UCT;
- $(s', o_c, r) \sim$ G$(s, a)$;
- $o_t$ = `PredictLabel`$(o_c)$;
- if $d <$ horizon then
  - if T$(h, a, o_t)$ is not Ø then
    - futureRew = `Simulate`$((h, a, o_t), s', d+1)$;
  - else
    - initialize T$(h, a, o_t)$;
    - futureRew = `Rollout`(s', d+1);
  - r = r + $\gamma \times futureRew$
- B(h) = B(h) $\bigcup \{s'\}$;
- N$(h, a)$ = N$(h, a)$ + 1;
- V$(h)$ += $\frac{r - V(h)}{N(h,a)}$ update value for action;
- return r;

Algorithm 6: Integrating observation class label prediction in POMCP algorithm.

As described in the sections in the previous chapters, the Rollout policy is used to generate history sequences from belief states that are not already in the search tree. While generating the history sequences, the Rollout policy samples actions uniformly at random from a pool of all possible actions available for the agent. We attempt to focus this sampling by sampling only the valid actions. For this, we take the idea of Thompson sampling algorithm. Traditionally, Tompson sampling has been used to address the exploration-exploitation dilemma in multi-armed bandit problem. The Thompson sampling algorithm iteratively learns probability distributions for random variables. To explain the underlying idea, let us consider a Bernoulli bandit problem. Imagine that we have three bandits $a_i$, $i \in (1,2,3)$. Let $\mu_i$ be the probability that an arm $i$ results in success (reward=1). The idea of Thompson sampling is to learn a probability distribution over $\mu_i$ for each bandit. To solve the Bernoulli bandit problem at hand, we start with Beta distributions to model the prior on $\mu_i$. Assume that we start off with $\beta(1,1)$, a uniform distribution, as the prior on $\mu_i$ for each arm $a_i$. We iteratively play all the arms and update the distributions for each arm depending on whether we observed success(reward=1) or failure(reward=0). At each iteration, having observed $S_i(t)$ successes(reward=1) and $F_i(t)$ failures, in $K_i(t) = S_i(t) + F_i(t)$ trials for arm $i$, the algorithm updates the distribution on $\mu_i$ as $\beta(S_i(t)+1, F_i(t)+1)$. The algorithm then samples $\mu_i$ from these distributions and plays the arm with largest $\mu_i$. The Thompson sampling algorithm is summarized in Algorithm 7. We assumed three bandits, however, the idea can be extended to multiple such bandits.

Algorithm Thompson Sampling
- foreach $i \in [1,2,3]$ do
  - $a_i : S_i = 0, F_i = 0$;
- foreach $t \in [1,2,3,....]$ do
  - For each arm $i \in [1,2,3]$, Sample $\mu_i(t)$ from $\beta(S_{i(t)} + 1, F_{i(t)} + 1)$;
  - Play arm $i(t) := \arg\max_i \mu_i(t)$ and observe reward $r_t$;
  - if $r_t = 1$ then
    - $S_{i(t)} = S_{i(t)} + 1$;
  - else
    - $F_{i(t)} = F_{i(t)+1}$;

Algorithm 7: Thompson sampling algorithm

We take inspiration from the Thompson sampling algorithm to enhance the action sampling process in the Rollout policy. Instead of using complex distributions to model the action sampling, we use a uniformly weighted particle filter that contains valid actions as particles. In the case of our manipulator, we have four actions, down, right, up and left. At the beginning, we have an equal number of instances of each of these actions in our particle filter. An action is now sampled from the particle filter uniformly at random and is executed. Depending on whether the action resulted in our manipulator moving closer to the target or further away from it, as indicated by the discrete distance sensor reading $d$, an instance of that action is either added or removed from the particle filter respectively. Thus by iteratively doing the above procedure, the particle filter contains a greater number of instances of the most promising actions. In the case where there are no particles left in the particle filter, we replenish it by adding equal instances of all valid actions and repeat the procedure above. This, however, in our experience occurs rarely as the manipulator reaches the target before all particles are deleted from the particle filter in the majority of the cases. Algorithm 8 illustrates the idea.

Procedure `InitialiseActions()`
- A = [down, right, up, left];

Algorithm `ValidActions(`$o_d$, $a$`)`
- Input: Binary distance sensor $o_d$, Action performed $a$
- if $o_d = 0$ then
  - delete $a$ from A;
  - return A;
- else
  - add $a$ to A;
  - return A;

Algorithm 8: Enhancing the Rollout policy using Thompson sampling.

We begin by initializing a set $A$ of valid actions that includes all possible actions to navigate in the 2D environment. $o_d$ is a binary distance sensor reading from the manipulator end effector. It is $0$ when the end effector moves away from the target as a result of the action performed, or $1$ otherwise. Every time the executed action $a$ causes the end effector to move close to the target, it is added to $A$. On the other hand, if the executed action $a$ causes the end effector to move away from the target, it is deleted from $A$. Thus, by doing so, the MCTS process in POMCP algorithm is focused further by sampling only the most probable actions.

A comprehensive review of the implementation of the proposed methodologies to handle continuous observations discussed in this chapter has been provided in the Chapter 5.

# 5 POMCP Algorithm Implementations

## 5.1 Introduction

The master thesis incorporates four implementations of POMCP algorithm, that are distinguished based on the manner in which they handle the state and observation spaces. The four implementations of the POMCP algorithms are mentioned with their abbreviations below.

- `POMCP_DS_DO:` Partially Observable Monte Carlo Planning with Discrete State and Observation spaces
- `POMCP_CS_DO:` Partially Observable Monte Carlo Planning with Continuous State and Discrete Observation spaces
- `POMCP_CS_CO_EqualPartitions:` Partially Observable Monte Carlo Planning with Continuous State and hybrid Observation spaces with equal partitioning of continuous observation space
- `POMCP_CS_CO_OnTheFlyPartitions:` Partially Observable Monte Carlo Planning with Continuous State and Hybrid Observation spaces with dynamic partitioning of continuous observation space.

In the following sections, we summarize briefly the functionality of the component blocks of the POMCP algorithm.

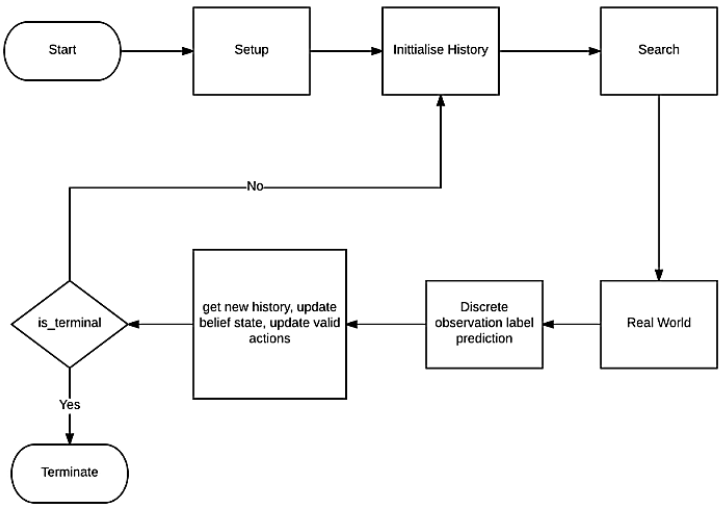



Figure 5.1.: POMCP algorithm overview.

## 5.2 POMCP Algorithm Structure

This section elaborates the functionality of underlying structural components of the various versions of POMCP algorithm implemented in the thesis. Figure 5.1 shows POMCP algorithm structure.

### 5.2.1 Setup

The Setup block sets up an environment for the algorithm to work in. It starts off by initializing a data queue that is used to communicate between the planning algorithm and a simulation of the real world environment. Following this, it starts a parallel thread that runs a real world environment. The data queue mentioned before is used by the real world thread to post observations to the planning algorithm running in the main thread. Finally, events that indicate an observation being posted, the action being posted to the real world, and an observation being posted from the real world to the planning algorithm are initialized.

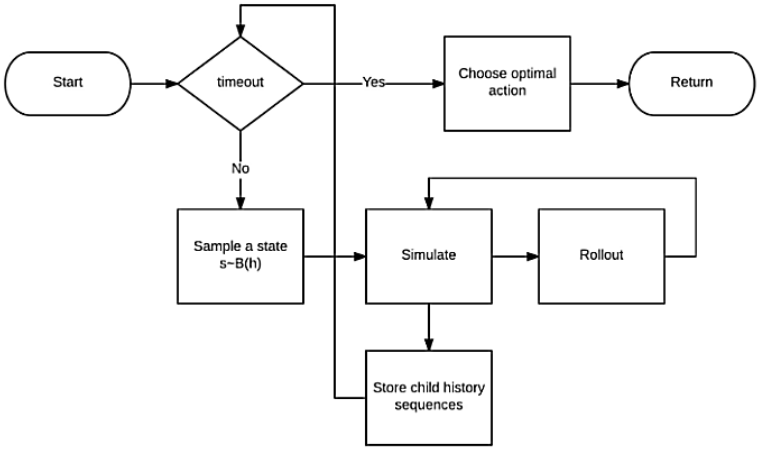



Figure 5.2.: Search for an optimal action.

### 5.2.2 Initialise history

This block initializes an object of SearchHistory class which serves as a starting node for the algorithm. At this point, the algorithm starts off by initialising the SearchHistory object to an empty history as there is no data at the start of planning. A list containing a set of actions that are considered valid for that history node are also initialized at this step. At the beginning, this set contains actions in cardinal directions. These are further modified as the algorithm proceeds to reach the target. This history node is further passed on to the next block to start the search process for an optimal action.

### 5.2.3 Search

The block contains functionality that carries out a forward search from the input history node. It runs a set of simulations from the input history node for a set amount of planning time that can be changed manually. The functionality of this block is implemented in the search() method of the SearchHistory class. At the timeout, the method returns an action that is optimal for input history. This action corresponds to the one that returns the largest long-term reward. It also returns the received reward. It then posts this optimal action over the data queue to the real world environment
Figure 5.2 summarizes the functionality of the Search block in Figure 5.1.

A state is sampled from the belief state corresponding to the history under current exploration. Simulations are run from this sampled state. Simulate method chooses an optimal action from the Monte Carlo Tree if the history has been previously visited, otherwise a Rollout is performed. The simulations are stopped when either there is a timeout or a terminal state has been reached. The propagated long term rewards are then consolidated for each action for the history sequences generated and an action corresponding to the largest returned reward is returned from the method along with the corresponding reward. This action is further executed in the real world thread.

### 5.2.4 Real World

This block represents the real world hosted in the parallel thread initialized in the setup block. It receives an optimal action from the Search block and carries out that action in the real world. It then posts an observation as a result of the action execution to the data queue. The observation is in the form of a list as shown below.

- observation: [haptic, nf, cd] where,
  - haptic: is the discrete haptic sensor feedback from the manipulator
  - nf: is the discrete distance sensor feedback from the manipulator
  - cd: is the continuous valued distance between the manipulator and the target body

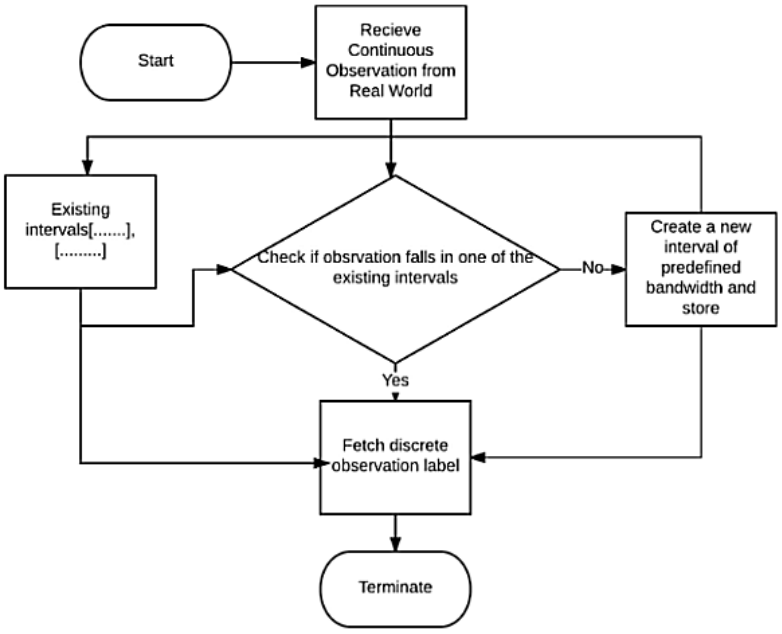



Figure 5.3.: On-the-fly partitioning the continuous observation space.

### 5.2.5 Discrete Observation Label Prediction

This block implements the functionality of mapping continuous observations to discrete observation labels to circumvent the problem of infinite branching at observation nodes in Monte Carlo tree search. This block is relevant to the following listed versions of POMCP algorithm implementations.

- `POMCP_CS_CO_EqualPartitions`
- `POMCP_CS_CO_OnTheFlyPartitions`

#### `POMCP_CS_CO_EqualPartitions`

As explained in the previous chapters, the idea here is to equally partition the one-dimensional continuous observation space into partitions of equal size. The range of continuous values spanned by the partition is controlled by a range parameter. Each partition forms a discrete observation class label.

#### `POMCP_CS_CO_OnTheFlyPartitions`

Uses dynamic partitioning of continuous observation into partitions of programmable predefined bandwidth. The number of discrete observation labels generated in this case is variable. Figure 5.3 shows a block diagram explaining an alternative approach to handling continuous observations. The idea is to partition the continuous observation space in real time as we plan and progress towards reaching the target. These partitions are not known at the beginning of the algorithm and are generated on the fly as we move towards the target. Each of these partitions is governed by a bandwidth parameter that spans a range of continuous values. Partitions represent discrete observation class labels for the incoming continuous observations. After receiving a continuous observation from the real world, the planning algorithm checks if it lies in one of the existing partitions that it has in its online memory. When there is a match, it immediately returns the corresponding discrete observation class label and continues either planning or simulating, wherever the method was called from. In the case when there is no match with the existing partitions, the planning algorithm creates a new partition of a predefined bandwidth, assigns a unique observation class label and returns this label to the calling method. Thus the implemented methodology encapsulates continuous observation values received from the real world, from the POMCP algorithm.

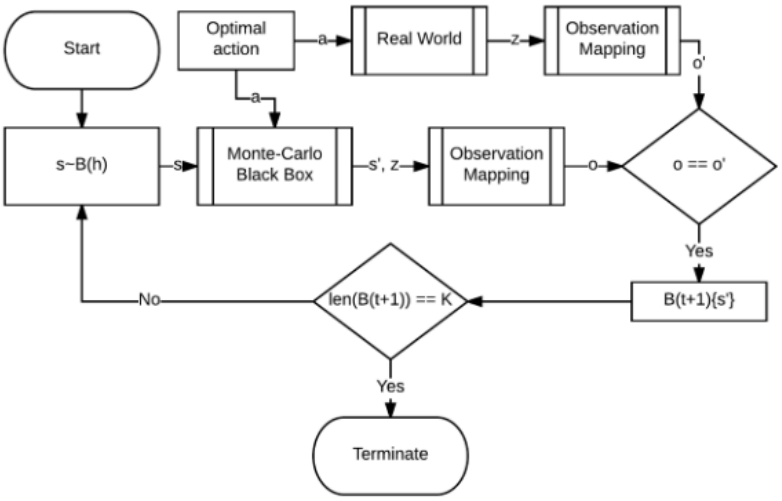



Figure 5.4.: Monte Carlo belief state update.

### 5.2.6 Updates

This block implements the following functionality.

- Get new history: This implements functionality to get a new history node to explore based on the observation received from the real world. A new history sequence is chosen from the child history sequences generated during simulations where the observation matches the incoming observation from the real world. This functionality is implemented in the new_history method of the SearchHistory class. Refer to appendix for more details. The returned new history is used in the next iteration of planning to explore optimal actions.

- Update belief state: This part implements functionality to update the belief state of the history, currently under exploration. Belief state updates are performed using rejection sampling. This is illustrated in Figure 5.4. The planning algorithm computes the optimal action for a given time step and executes it in the real world. The manipulator in the real world now receives a continuous valued observation. The state of the manipulator in the real world has changed on account of having performed the optimal action. This state is however not observable to the planning algorithm and therefore the algorithm instead updates its belief state. This is done by rejection sampling. At the beginning of the algorithm, as illustrated in the Figure 5.4, a state is sampled from the current time step belief state of the history under exploration. The state along with the computed optimal action for the corresponding time step are then passed to the Monte Carlo black box simulator. The simulator returns a news state $s'$ and a continuous observation $z$. Using one of the continuous observation mapping methods described in the previous section, the received continuous observation is now mapped to a discrete observation class label, $o$. The same is done with the continuous observation received from the Real World, to receive a discrete label, $o'$. If the two mappings $o$ and $o'$ match, then the corresponding new state $s'$ is added to the updated belief state for the current history, when not, the algorithm starts off the whole process from sampling a state. The process is repeated until $K$ particles have been added to the updated belief state. We experiment with various values of $K$.

- Update valid actions: Valid actions are chosen based on information on the current belief state and received observation. The discrete distance and haptic sensor observations influence an action being considered as valid for the current belief state. This significantly reduces the time needed to guide the manipulator to the target, since actions deemed invalid are no longer explored during the planning phase.The approach is inspired by Thompson sampling as discussed in previous chapters.

# 6 Experiments

As mentioned in chapter 4, we discuss two approaches to address the manipulator problem at hand, namely, imitation learning and with POMCP algorithm. In section 6.1, we explain the general experimental setup. In section 6.2 we discuss the experiments with the imitation learning approach followed by the results and conclusion. In section 6.3, we discuss the experimental setup for the POMCP algorithm approach. We also discuss the results of experiments with the extension of the POMCP algorithm to accommodate for continuous state and observation spaces with the two proposed approaches. Finally, in section 6.4 we summarize our conclusions.

## 6.1 Experimental Setup

We model the manipulator problem described in chapter 3 by programming a simulated environment in python. We used the pybox2D physics library to simulate real world physics. The state space of the simulated environment is discretized to span a discrete state space of $10 \times 7$, resulting in 70 discrete states. It is bounded by walls on its left, right and top positions. The bottom of the environment is bounded by a body we refer to as ground. The manipulator end effector is represented by a gray box and the target body is represented by a green box. The end effector is initialised anywhere in the simulated environment, except along the ground. The target body is always initialised anywhere along the ground. We assume a discrete action space containing actions that move the end effector in the cardinal directions (up, down, right and left). The manipulator end effector steps into one of the 70 discrete states every time an action is applied in a cardinal direction. The simulated real world is shown in Figure 6.1.

The goal is to navigate the end effector to the target body. We simulate partial observability by encapsulating the actual position of the target body from the manipulator end effector. The end effector(hand body) is laced with haptic sensors that provide discrete observations about the real world. For eg. when the hand body touches a wall on the right, the returned observation is 'wall on right'. Similar descriptive observations are received for other situations. In addition to the haptic sensors, the hand body is also equipped with a discrete distance sensor that indicates whether the hand body moved close to or away from the target body after having performed an action in the environment. The manipulator end effector is also equipped with a noisy position sensor that gives noisy estimates of the end effector's position in the environment. Table 6.1 summarizes the discrete distance sensor codes and Table 6.2 summarizes the available discrete haptic observations and their corresponding codes.

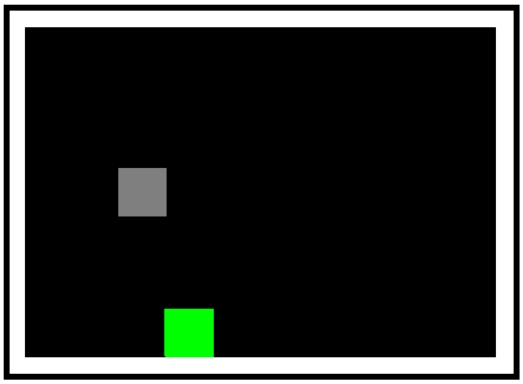


Figure 6.1.: Model of simulated real world.

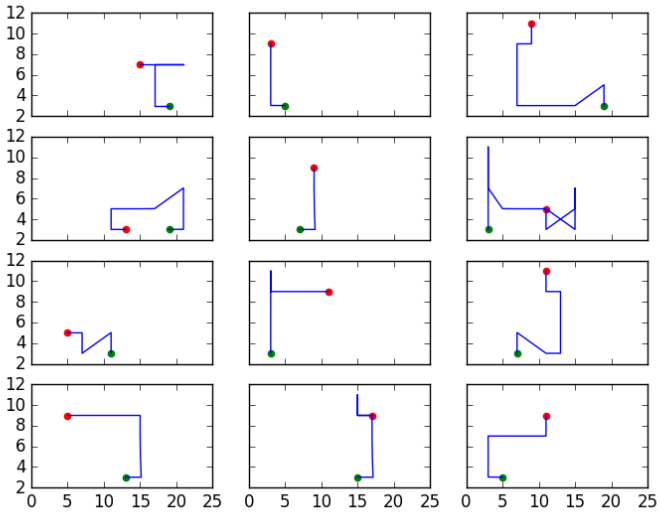


Figure 6.2.: A collection of randomly sampled player trajectories. The hand body starts at a point indicated by a red dot and the location of the target body is denoted by a green dot. The colored dots indicate the centers of both bodies. The blue line indicates the followed trajectory.

| Code | Description |
|---|---|
| 0 | Moved away |
| 1 | Moved near |

Table 6.1.: Discrete distance sensor codes.

| Code | Description | Code | Description |
|---|---|---|---|
| 0 | wall right | 4 | ground below |
| 1 | wall left | 5 | object below |
| 2 | object right | 6 | wall above |
| 3 | object left | 7 | no information |

Table 6.2.: Haptic sensor codes.

## 6.2 Imitation Learning Approach

Our first approach towards solving the manipulator problem was to learn by imitation. The idea is to imitate the behavior of a player to learn a policy. We interfaced the simulated environment with a keyboard. Using the arrow keys on the keyboard, a player could manipulate the position of the hand body. Every time the player moves the hand body, he/she receives observations from the haptic and discrete distance sensors. The end of game play is reached when the player lands the hand body on top of the target body. To simulate partial observability, the window displaying the simulated environment is hidden from the player until the end of the gameplay. Thus the player attempts to navigate the hand body towards the target body by performing actions in the current time step that are purely influenced by a set of observations that he/she receives from the environment in the previous time step. We attempted to imitate this observation-action mapping to learn a policy.

### 6.2.1 Methodology

We recorded game plays from a group of 50 players, a total of 189 trajectories where they attempt to navigate the hand body to the target body. Figure 6.2 displays the recorded trajectories. For every player trajectory, we recorded the following features.

- $x_H, y_H$: 2D coordinate of the hand body in the simulated environment
- $h$: haptic sensor observation at the previous time step
- $d$: discrete distance sensor observation at previous time step

- $a$: action performed at the current time step

Table 6.3 illustrates an excerpt of the trajectory data recorded for players. To learn observation-action mappings, we train 3 models from scikit learn[23] python library on the recorded trajectories. The models used are,

- LinearSVC
- SGDClassifier
- RandomForestClassifier.

The linearSVC model is a support vector machine(SVM)-based multi-class classifier that learns to predict the mapping between a received observation $[x_H, y_H, h, d]$ and of the 4 action labels $a$. The SGDClassifier is also a SVM-based multi-class classifier that uses stochastic gradient descent learning to learn a linear classifier. The RandomForestClassifier is a random forest-based classifier initialised to 20 estimators in the forest. It uses an ensemble of decision tree classifiers and fits them to sub-samples of the training dataset. It averages over the decisions of the decision trees to improve predictor accuracy and control over-fitting. In our experiments with the imitation learning approach, we attempt to find if a deterministic observation-action mapping learned using the trained classifiers would result in a usable policy.

| $x_H$ | $y_H$ | h | d | a |
|---|---|---|---|---|
| 5.00 | 13.00 | 7.0 | 1.0 | 2.0 |
| 5.00 | 11.00 | 7.0 | 0.0 | 0.0 |
| 5.00 | 9.00 | 7.0 | 1.0 | 0.0 |
| 5.00 | 7.00 | 7.0 | 1.0 | 0.0 |
| 5.00 | 5.00 | 7.0 | 1.0 | 0.0 |

Table 6.3.: Sample partial-observability data set.

| action | label |
|---|---|
| down | 0 |
| right | 1 |
| up | 2 |
| left | 3 |

Table 6.4.: Discrete action codes.

We explored the trajectories followed by the players. The density of 2D coordinates traced by players is shown in Figure 6.3.

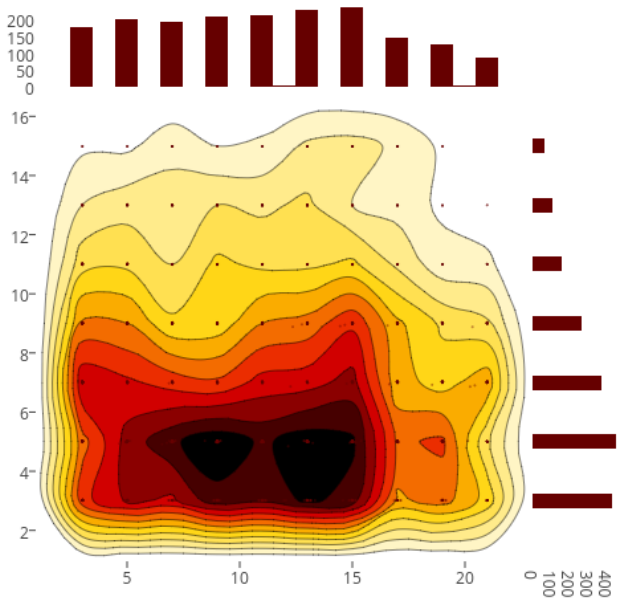



Figure 6.3.: A 2D histogram showing the distribution of the most common 2D locations followed by players during game play.

We used the $x_H$, $y_H$, $h$ and $d$ as features to learn a predictor that predicts the target action label $a$. The mean squared error and model score were used as a metrics of the predictor accuracy. The model score is the mean accuracy of the classifier model on the unseen values for the data features. Table 6.5 illustrates the performance of the 3 models.

| Model | MSE | Score |
|---|---|---|
| LinearSVC | 2.631 | 0.424 |
| SGDClassifier | 2.757 | 0.420 |
| RandomForestClassifier | 0.792 | 0.800 |

Table 6.5.: Model performance metrics with partial-observability data.

| Feature | Importance |
|---|---|
| $x_H$ | 0.447 |
| $y_H$ | 0.460 |
| h | 0.047 |
| d | 0.043 |

Table 6.6.: Random forest based classifier feature importances.

Having learned observation-action mappings, we made the learned models available to the manipulator end effector(agent) and ran independent games. The agent used the model for predicting the best action based on the observations it received in the previous time step. We observed that the agent got stuck in a loop where it would predict an action repeatedly after hitting the environment boundaries. Thus, the agent was not able to navigate itself to the target body.

Of the three classification models that we trained, the random forest based classifier performed the best on both our metrics. It was also observed that although by chance, the random forest based classifier model was successful to navigate the agent more often than when compared to the other models we learned. We investigated the feature importances returned by the random forest classifier and found that the noisy 2D location information played a key role in node splitting in decision trees. The feature importances returned by the random forest classifier are shown in the Table 6.6.

Therefore, in an attempt to investigate if the performance of the classifier models could be further improved at all, we step outside our initial setting and make the information of the location of the target body available in the training data for the classifier models. Table 6.7 shows an excerpt of the new training data we used.

| $x_H$ | $y_H$ | $x_t$ | $y_t$ | h | d | a |
|---|---|---|---|---|---|---|
| 15.00 | 7.01 | 19.0 | 3.01 | 7.0 | 0.0 | 2.0 |
| 17.00 | 7.01 | 19.0 | 3.01 | 7.0 | 0.0 | 1.0 |
| 19.00 | 7.01 | 19.0 | 3.01 | 7.0 | 1.0 | 1.0 |
| 20.98 | 7.01 | 19.0 | 3.01 | 7.0 | 1.0 | 1.0 |
| 18.98 | 7.01 | 19.0 | 3.01 | 7.0 | 0.0 | 3.0 |

Table 6.7.: Sample full-observability training data set.

| Model | MSE | Score |
|---|---|---|
| LinearSVC | 0.304 | 0.695 |
| SGDClassifier | 0.414 | 0.585 |
| RandomForestClassifier | 0.109 | 0.890 |

Table 6.8.: Model performance metrics with full-observability data.

It was observed that the performance of our learned models improved substantially on both metrics. However, despite the improvements on the metrics none of the learned models were able to navigate the agent to the target body. Since we learn a deterministic observation-action mapping, the agent would get stuck into deterministic observation-action loops. Our models attempt to learn a deterministic memory-less mapping between the observations and actions which do not take into consideration the past history of the agent's behavior before deciding an action. Also, our learned memory-less mapping assumes a Markovian environment when, in fact, the simulated environment is non-Markovian. Nevertheless, finding an optimal deterministic memory-less policy in such a setting is NP-complete [24]. Ideally, to improve the performance of our learned models, a search would have to be made across the entire policy space. Attempting to search an optimal policy by brute force would require a planning algorithm based on such a deterministic mapping based approach to search a space of policies of size $|A|^{|O|}$, where $A$ and $O$ denote the action and observation spaces respectively. The size of the policy search space is exponential in the size of the observation space. It is clear that the problem will only worsen when the observation space becomes continuous. Although there exist approaches such as branch-and-bound heuristics to improve the policy search [24], we do not pursue our efforts in that direction in the scope of this thesis.

## 6.3 The POMCP Algorithm Approach

We implemented the POMCP algorithm in its original framework as described in [3]. The planning algorithm was made available to the agent. It was observed in our experiments that the agent was successfully able to navigate itself to the target body. We experimented with the following implementations of the POMCP algorithm as a part of the thesis.

- POMCP_DS_DO - Discrete states and observations
- POMCP_CS_DO - Continuous states and discrete observations
- POMCP_CS_CO - Continuous states and continuous observations
  - POMCP_CS_CO_EqualPartitions - hand made partitions
  - POMCP_CS_CO_OnTheFlyPartitions - generating partitions on-the-fly

### 6.3.1 Methodology

For our experiments with the implementations of the POMCP algorithm, we recorded data for the following parameters.

- $t$ - planning time for which simulations are run during planning
- $K$ - belief state particle filter size
- $\gamma$ - planning horizon
- $m$ - number of partitions for POMCP_CS_CO_EqualPartitions approach
- $q$ - bandwidth parameter for POMCP_CS_CO_OnTheFlyPartitions approach

We experimented for $t$ = [1.5, 2.5, 3.5, 4.5, 5.5], $K$ = [10, 15, 20, 25, 30], $\gamma$ = [0.1, 0.2, 0.3, .... ,0.9], $m$ = [2, 3, 4, 5] and $q$ = [1, 2, 3, 4, 5]. We attempt to answer the following research questions.

1. What is the impact of using the enhanced Rollout policy on the number of steps it takes for the agent to reach the target in the original version of the POMCP algorithm - for POMCP_DS_DO implementation?
2. Does the idea of using continuous 2D coordinates of the state as particles of the particle filter representing the belief state enable the POMCP algorithm to work in the continuous state space for POMCP_CS_DO implementation?
3. What size particle filter and duration of planning time result in the best performance in terms of the average number of steps the agent takes to the target for POMCP_CS_DO implementation?
4. How do the two approaches to handle the continuous observation space compare with each other in terms of the average steps they take to reach the target for different values of $t$, $\gamma$ and $K$?
5. What value, from the set of experimental values, of the number of partitions $m$ results in the best performance of the RP approach?
6. What value, from the set of experimental values, of the bandwidth parameter $q$ results in the best performance of the OTF approach?
7. How do the implementations for POMCP_CS_CO with approaches RP and OTF compare in performance to the POMCP_CS_DO approach in terms of the average steps to the target?

In the following subsections, we describe the experiments with the above-mentioned implementations of the POMCP algorithm and summarize our results.

### 6.3.2 Discrete States and Discrete Observations - POMCP_DS_DO

We first experimented with the implementation of the POMCP algorithm working in discrete state and observation space in our simulated environment to answer our research question 1. The belief state consisted of discrete state particles. The observation space consisted of discrete haptic sensor readings along with the discrete distance sensor reading. For the experiment, we ran the algorithm in the simulated environment for 100 iterations and collected the average number of steps it took for the agent to reach the target. We did this for two cases, first, using all action branches during planning (NVA) and second, with the enhanced Rollout policy (VA). The results of the experimental runs are summarized in Table 6.9. The probability density function of the number of steps for each of the approach is shown in Figure 6.4.

| Action strategy | Average steps to target |
|---|---|
| NVA | 12 |
| VA | 10 |

Table 6.9.: Average steps to target for the two action selection strategies during planning. NVA indicates the average number of steps the agent took to reach the target when the entire action space was used during planning to generate simulations. VA indicates the average number of steps the agent took to reach the target when the enhanced Rollout policy was used to explore the action space.

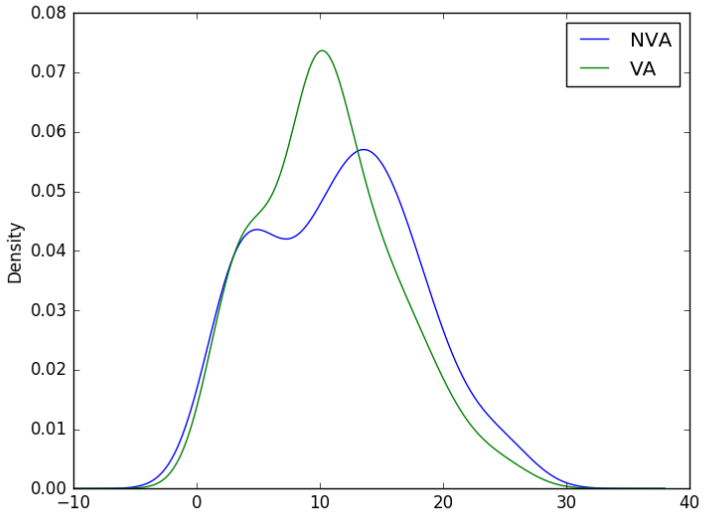



Figure 6.4.: Probability density function of the number of steps for NVA- entire action space and VA - using enhanced Rollout policy. The data is collected from 250 simulations.

It was observed that the agent was able to reach the target in fewer steps when using the enhanced Rollout policy.

### 6.3.3 Continuous States and Discrete Observations - POMCP_CS_DO

Next, we experimented with our implementation of the POMCP algorithm to accommodate continuous state spaces. We attempt to answer our research questions 2 and 3. We evaluate if the approach converges to the target in a continuous state space. The observation space, as in the previous experiment, was discrete. We observed that the agent was able to navigate itself to the target body in the continuous state space. We used the average number of steps that the agent took to reach the target as a metric to compare the performance of the algorithm implementation with the performance of planning in a discrete state space, (VA). Figure 6.5 shows the probability density functions for the number of steps that the agent takes to reach the target in both cases. We refer to this implementation of the POMCP algorithm in shorthand as CS for future references.

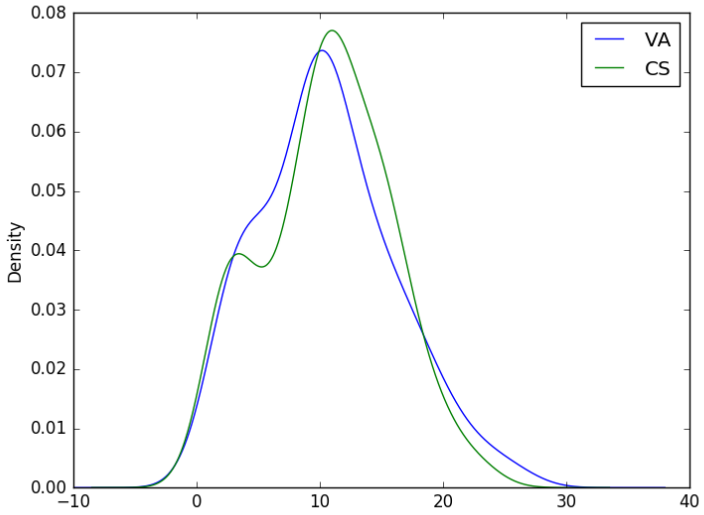
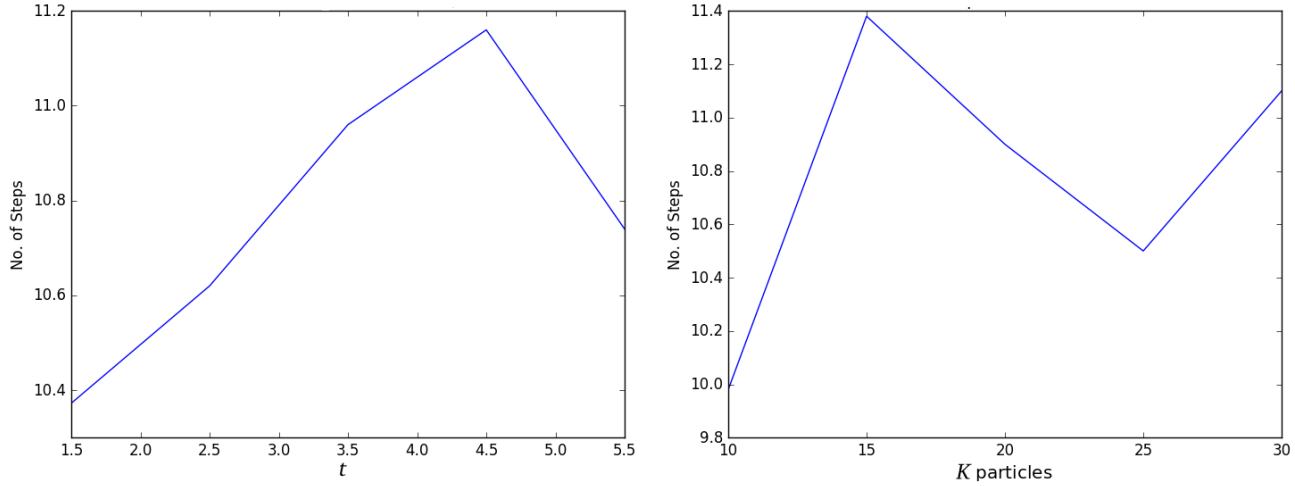



Figure 6.5.: Comparison of probability density function (pdf) of the number of steps for VA - using enhanced Rollout policy and discrete states against pdf of number of steps for the POMCP algorithm adapted to accommodate continuous states indicated in the legend as CS. The data is collected from 250 simulations.

We used the enhanced Rollout policy for the continuous state adaptation as was the case with VA. It was observed that our implementation of the POMCP algorithm to accommodate continuous states enables the agent to reach the target in an average of 10.7 steps which is comparable to the performance of the POMCP algorithm in its original framework. Further, we ran experiments for a range of planning times and values of $K$ and measured the number of steps the agent takes to reach the target in each of the cases. The results of our experiment are shown in Figure 6.6.

We observed that the algorithm performed the best in terms of the average number of steps to target for a planning time of 1.5 seconds and a particle filter containing 10 particles.




(a) (b)

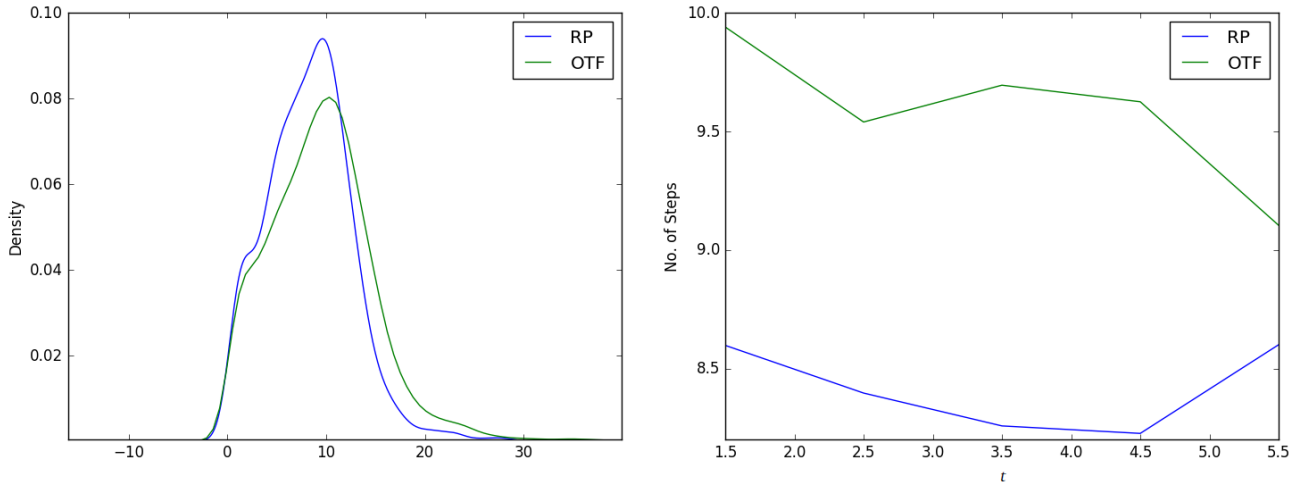
Figure 6.6.: (a) shows a graph of average steps to target for a range of planning times $t$. (b) shows a graph of average steps to target for a range of values for $K$. The data is collected from 250 simulations.

### 6.3.4 Continuous State and Observation Spaces

We experimented with our two approaches to handling the continuous observation space, namely,

- POMCP_CS_CO_EqualPartitions - RP
- POMCP_CS_CO_OnTheFlyPartitions - OTF

We choose the number of steps it takes for the agent to navigate itself to the target as a metric to compare the performance of the two approaches. Figure 6.7 (a) shows the pdf of an average number of steps for each of the approaches. Table 6.10 summarizes the performance for each of the approaches on the metric.We attempt to answer our research questions 4, 5, 6 and 7 in this experiment.



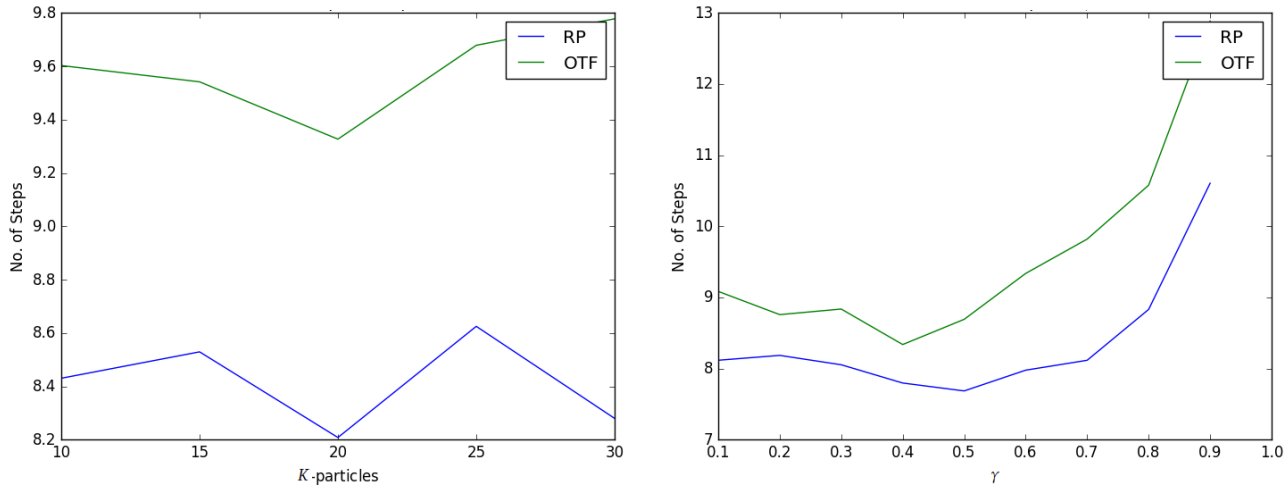

(a) (b)

Figure 6.7.: (a) shows a comparison of pdf of the number of steps for the RP approach against pdf of number of steps for the OTF approach. (b) shows performance comparison of continuous observation handling approaches for a range of values for planning time $t$. Data collected from 6000 simulations.

| Approach | Average steps to target |
|---|---|
| RP | 8.41 |
| OTF | 9.58 |

Table 6.10.: Average steps to target for the two approaches, RP and OTF.

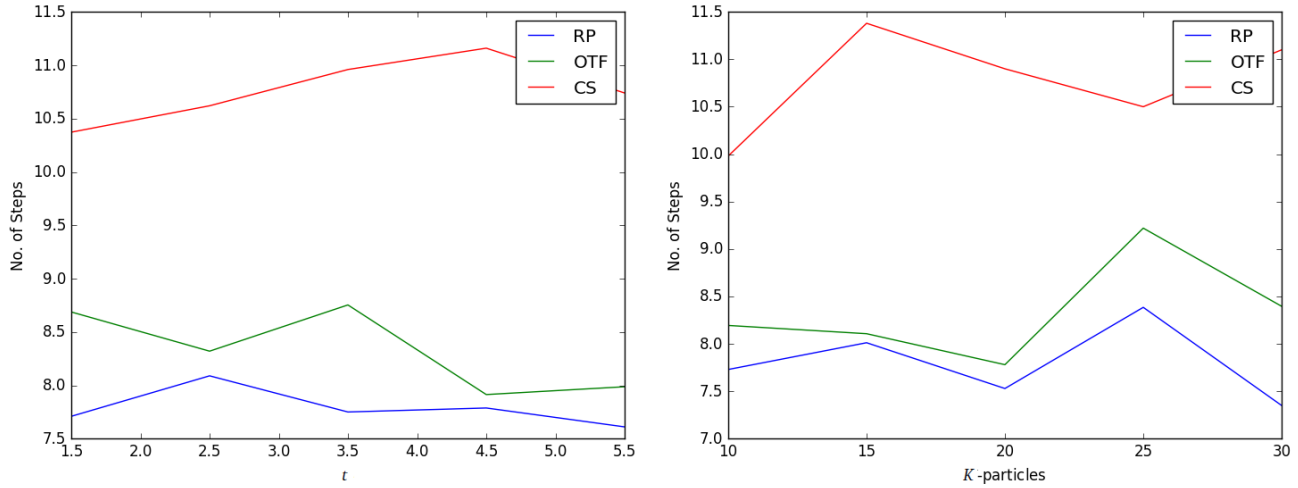
We then experimented with the performance of the two approaches across the planning time $t$, particle filter size $K$ and the planning horizon $\gamma$ as common features for comparison. Figure 6.7 (b) and Figure 6.8 show the results of our experimental runs. We observed that the RP approach reaches the target in the least number of steps for a planning time of 4.5 seconds. The OTF approach, however, shows a downward trend for average steps it takes to reach the target for larger planning times. The RP approach performed the best for $K$=20 when measured for different values of $K$. The OTF approach also performed best for a particle filter containing 20 particles. An interesting finding was the performance of the approaches for different values for the planning horizon $\gamma$. We see that the RP approach performed the best for $\gamma$=0.5, implying it searched approximately 7 steps ahead from the current state during planning to collect its statistics for the choice of the best action. On the other hand, the OTF approach needed just 5 steps to do the same. Nevertheless, the performance of the two approaches differed by approximately just one step on an average that they took to reach the target.




(a) (b)

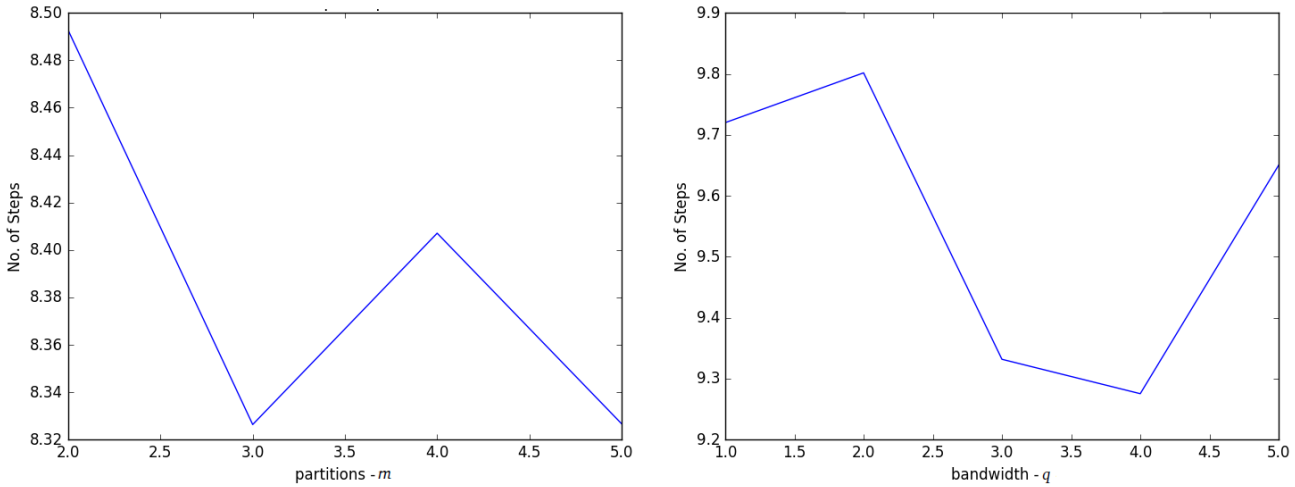
Figure 6.8.: (a) shows a performance comparison of continuous observation handling approaches for a range of particles $K$. (b) shows performance comparison of continuous observation handling approaches for a range of values for planning horizon $\gamma$. Data collected from 6000 simulations.

To see the larger picture, we plot the results of our experiments with the CS, RP and OTF approaches together in Figure 6.9. We observed that the RP and OTF approach substantially outperformed the CS approach. For the RP approach, we experimented with different values for $m$ to partition the observation space. We observed that the approach recorded the minimum number of steps to target when the observation space was divided into 3 partitions. The graph, further, showed a downward trend. The results of this experiment are shown in Figure 6.10 (a). For the OTF approach, we experimented with different values of the bandwidth parameter $q$. We recorded the best performance for $q$=4. This is shown in Figure 6.10 (b).




(a) (b)

Figure 6.9.: (a) shows a performance comparison of continuous observation handling approaches RP and OTF against that of CS for a range of values for planning time $t$. (b) shows performance comparison of continuous observation handling approaches RP and OTF against that of CS for a range of values for particle filter size $K$. The values calculated in (a) and (b) assume the value of $\gamma$=0.4.




(a) (b)

Figure 6.10.: (a) shows the average number of steps that the agent takes for a given partition size $m$ for the RP approach. (b) shows the same for given size of the bandwidth parameter $q$ for the OTF approach. Data collected from 6000 simulations.

## 6.4 Summary

Our experiments with simple classification models to learn a policy in our manipulator problem highlight the fact that a simple memory-less policy that learns deterministic observation-action mappings fails to achieve the target as it tends to get stuck in deterministic observation-action loops. In our approach to formulate the problem as a POMDP, we successfully implemented the POMCP algorithm as described in [3]. Our experiments with enhanced Rollout policy show that the agent is able to reach the target in fewer steps. In our experiments with the two approaches to handle the continuous observation space, we found that dividing the observation space into equal partitions (RP) performs better than OTF.

# 7 Discussion and Future Work

Solution methodologies for decision-making problems formulated as POMDPs have several challenges as mentioned in chapter 1. The POMCP algorithm addresses these by its unique Monte Carlo sampling approach. The particle filter representation of the belief state makes accommodating continuous state spaces practical. However, in our experiments, we see that the performance can be tuned by experimenting with the particle filter size. In our study, we experimented with a range of sizes for the particle filter to find its optimal size. A more practical approach would be to tune this parameter online. A learning algorithm could be developed to optimize the particle filter size as the agent navigates itself to the target in real time. We also observed that the belief state update process sometimes has a tendency of getting stuck. This happens when there is a large gap in the belief representation of the POMCP planning algorithm and the actual state of the agent in the real world. An alternative would be to investigate approaches that are free of belief state representations such as [18]. For our approaches to handle continuous observations, generating partitions of the continuous observation space that increase the agent's long-term reward would be a part of future work. For our study, we assumed a discrete action space which brings with it a certain degree of discretization of the environment. Accommodating continuous actions by sampling approaches such as [25] could also be a part of future work.

# Bibliography


[1] E. J. Sondik, "The optimal control of partially observable markov processes.," tech. rep., DTIC Document, 1971.

[2] J. Pineau, G. Gordon, S. Thrun, et al., "Point-based value iteration: An anytime algorithm for pomdps," in IJCAI, vol. 3, pp. 1025–1032, 2003.

[3] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," in Advances in neural information processing systems, pp. 2164–2172, 2010.

[4] H. Kurniawati, D. Hsu, and W. S. Lee, "Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces.," in Robotics: Science and systems, vol. 2008, Zurich, Switzerland., 2008.

[5] S. Ross, J. Pineau, S. Paquet, and B. Chaib-Draa, "Online planning algorithms for pomdps," Journal of Artificial Intelligence Research, vol. 32, pp. 663–704, 2008.

[6] "Darius-bimanual manipulation platform from intelligent autonomous systems lab, tu darmstadt," March 2017.

[7] B. Dynamics, "Ls3 - legged squad support systems," March 2017.

[8] A. Goldhoorn, A. Garrell, R. Alquézar, and A. Sanfeliu, "Continuous real time pomcp to find-and-follow people by a humanoid service robot," in Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on, pp. 741–747, IEEE, 2014.

[9] T. Smith and R. Simmons, "Heuristic search value iteration for pomdps," in Proceedings of the 20th conference on Uncertainty in artificial intelligence, pp. 520–527, AUAI Press, 2004.

[10] T. Smith and R. Simmons, "Point-based pomdp algorithms: Improved analysis and implementation," arXiv preprint arXiv:1207.1412, 2012.

[11] H. Bai, D. Hsu, W. S. Lee, and V. A. Ngo, "Monte carlo value iteration for continuous-state pomdps," in Algorithmic foundations of robotics IX, pp. 175–191, Springer, 2010.

[12] S. Paquet, L. Tobin, and B. Chaib-draa, "Real-time decision making for large pomdps," in Conference of the Canadian Society for Computational Studies of Intelligence, pp. 450–455, Springer, 2005.

[13] D. P. Bertsekas and D. A. Castanon, "Rollout algorithms for stochastic scheduling problems," Journal of Heuristics, vol. 5, no. 1, pp. 89–108, 1999.

[14] J. Satia and R. Lave, "Markovian decision processes with probabilistic observation of states," Management Science, vol. 20, no. 1, pp. 1–13, 1973.

[15] R. Washington, "Bi-pomdp: Bounded, incremental partially-observable markov-model planning," in European Conference on Planning, pp. 440–451, Springer, 1997.

[16] D. A. McAllester and S. Singh, "Approximate planning for factored pomdps using belief state simplification," in Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence, pp. 409–416, Morgan Kaufmann Publishers Inc., 1999.

[17] J. Hoey and P. Poupart, "Solving pomdps with continuous or large discrete observation spaces," in IJCAI, pp. 1332–1338, 2005.

[18] H. Bai, D. Hsu, and W. S. Lee, "Integrated perception and planning in the continuous space: A pomdp approach," The International Journal of Robotics Research, vol. 33, no. 9, pp. 1288–1302, 2014.

[19] Z. Zamani, S. Sanner, P. Poupart, and K. Kersting, "Symbolic dynamic programming for continuous state and observation pomdps," in Advances in Neural Information Processing Systems, pp. 1394–1402, 2012.

[20] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," Artificial intelligence, vol. 101, no. 1, pp. 99–134, 1998.

[21] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in European conference on machine learning, pp. 282–293, Springer, 2006.

[22] J. Schäfer, M. Buro, and K. Hartmann, "The uct algorithm applied to games with imperfect information," Diploma, Otto-Von-Guericke Univ. Magdeburg, Magdeburg, Germany, 2008.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[24] M. L. Littman, "Memoryless policies: Theoretical limitations and practical results," in From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior, vol. 3, p. 238, MIT Press, 1994.

[25] J. M. Porta, N. Vlassis, M. T. Spaan, and P. Poupart, "Point-based value iteration for continuous pomdps," Journal of Machine Learning Research, vol. 7, no. Nov, pp. 2329–2367, 2006.

[26] Z. Zhang, D. Hsu, W. S. Lee, Z. W. Lim, and A. Bai, "Please: Palm leaf search for pomdps with large observation spaces," in Eighth Annual Symposium on Combinatorial Search, 2015.

[27] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous upper confidence trees," in International Conference on Learning and Intelligent Optimization, pp. 433–445, Springer, 2011.

[28] A. Bai, F. Wu, Z. Zhang, and X. Chen, "Thompson sampling based monte-carlo planning in pomdps.," in ICAPS, 2014.

[29] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem.,"

# A Code Documentation

## A.1 Introduction

This chapter describes the code structure for the various implementations of the POMCP algorithm made as a part of the thesis. The chapter is divided into subsections dedicated to describe in a stepwise manner, the functionality of individual blocks and their inter-operation to produce an expected result. We start off by explaining the dependencies for the code in section A.2, followed by an list of all the relevant files in the code in section A.3.

## A.2 Dependencies

The code was developed using a Windows 10, 64 bit machine. The implementation of the code has several dependencies that have been briefly summarized in the list below.

1. Box2D: Physics engine to simulate dynamics of real world. Used in the making of black box simulator and the real world.
2. pygame: Windowed application to display the real world.
3. threading: Threading support to run the real world in an independent thread while the algorithm computes an online policy in a parallel thread.
4. Queue: Data queue to post actions computed by the online planning algorithm to the real world and post observations from the real world to the planning algorithm for processes such as belief state update and selection of valid actions.
5. sklearn: Supports the computation of a regression model to learn continuous to discrete observation mapping is versions of POMCP with continuous observations and generating a basic policy by imitation learning in one of the early approaches explained in the thesis to solve the discrete version of the problem.
6. graphviz: Visualization of the process of tree generation during online planning.
7. pandas, numpy: Serves as a medium for doing certain basic, intermediate data analysis.
8. pickle: Storing variables such as the learnt classification models.

## A.3 Relevant Files

It is important to note that the files listed below have variations in implementations when compared across the various implementations of POMCP algorithm made in this thesis. However they serve the same functionality as described below.

### A.3.1 RealWorld.py

As the name suggests, this file implements the real world of our manipulator problem.
method:

- `real_world`: Simulates the real world.
  parameters:
  - q: data queue object
  - l: lock object

returns:

- haptic: is the discrete haptic sensor observation
- nf: is a discrete observation that indicates whether the manipulator moved close to, or further away from the target after the action has been performed. This helps the selection of valid actions
- cd: this indicates the continuous distance between the manipulator and target

### A.3.2 Threads.py

This file implements a class myThread that runs a simulation of the real world. Class myThread extends the built in class, Thread, of the built in threading library in python.
methods:

- `__init__:` This is a constructor of the class and initializes the data queue and lock object of the queue.
  parameters:
  - q: data queue object
  - l: lock object
- run: this method overrides the run method from built in Thread class from threading library. It calls the `real_world` method to simulate the real world in a parallel thread.

### A.3.3 History.py

This file implements a class SearchHistory that handles manipulations to the history under consideration. The following is a list of all the methods that it implements.

- `__init__:` This is a class constructor and initialises the class variables.
  parameters:
  - timeout: indicates the time in seconds for which simulations are run during planning
  - `belief_state:` is the belief state for the corresponding history
  - `history_node:` is the current history node under exploration
- `observation_action_queue:` This method updates a list of discrete observations 'nf' and actions from the previous two states. This is supplementary for the selection of valid actions, per history under exploration.
  parameters:
  - observation: discrete distance sensor observation from the current step
  - action: current optimal action that was executed
- `get_valid_actions:` This method works on the observation and action queue maintained, in order to generate a list of valid actions for the history node under exploration.
- search: This method begins searching from a current history by running simulations for a period equal to a pre-specified timeout and returns an optimal action and the correspondingly value.
  parameters:
  - act: optimal action for the history under exploration
  - value: corresponding long term reward obtained when following the optimal action path in simulation during planning.
- `new_history:` This method returns a new history node based on the input discrete observation received from the real world. The new history node becomes the history under exploration for planning in the next time step.
  parameters:
  - observation: discrete real world observation.

  returns:
  - optimal child node

- `initialise_belief_state:` This method initialises the belief state of the history under exploration in case it was not initialised in the constructor.
- `belief_state_update:` This method updates the belief state of the history under exploration and stores it in the for further reference. The update process is based on rejection sampling. The method samples a start state from the current belief state and passes it to the black box simulator along with the currently executed optimal action. The black box simulator returns a next state and an observation. If the received observation matches the input observation then the next state particle is added to the new belief state that forms the updated belief state. This process is repeated until K particles are added to the updated belief state.
  parameters:
  - observation: input continuous real world observation

### A.3.4 `KDE_Regression.py`

This file defines a class KDE and implements its method that estimation of predefined observation classes for the following approaches.

- Equal partitioning of continuous observation space
- On-the-fly partitioning of continuous observation space

The following is a list of all its member and additional methods housed in this file.

- `clear_graph_data:` This is a static method that clears any data related to graphs used to visualize the simulation process.
- `generate_code:` This is a static method that generates dot code for generation of graph structures used for visualization in graphviz based on `graph_data`.
- `clear_temp_data:` This is a static method that clears temporary training data generated during the process of learning pdfs for discretized observation labels in `POMCP_CS_CO_hybrid`.
- `random_range_label:` This method predicts the discrete observation label given a continuous observation from the real world. It is part of POMCP_CS_CO_EqualPartitions approach implementation
  parameters:
  - value: continuous observation

  returns:
  - prediction: discrete observation label
- `observation_slice:` This method predicts the discrete observation label given a continuous observation from the real world. It is part of POMCP_CS_CO_OnTheFlyPartitions approach implementation
  parameters:
  - value: continuous observation

  returns:
  - prediction: discrete observation label

### A.3.5 Additional Methods:

- rollout: This method generates a history sequence from the given history node by running simulations on a black box simulator. It implements a rollout policy for the history under exploration when the history hasn't yet been encountered.
  parameters:
  - state: a state sampled from current belief state of the history under exploration
  - history: history node under exploration
  - depth: Monte Carlo tree depth at current time step

    - `actions_valid:` a list containing valid actions to be explored from history under exploration
    - `target_position:` current position of the target, this is encapsulated from the planning algorithm and is only used to have coherence between simulations.

  returns:

    - reward: long term reward of following a particular course of actions and observation during simulation

- simulate: This method generates a history sequence from the given history node by running simulations on a black box simulator. It checks if the incoming history node under exploration is in the tree. If so, directly goes ahead to fetch the optimal action from history and returns the corresponding reward. When the history node under exploration is not in the tree, the algorithms proceeds by calling the rollout method. parameters:
    - state: a state sampled from current belief state of the history under exploration
    - history: history node under exploration
    - depth: Monte Carlo tree depth at current time step
    - `actions_valid:` a list containing valid actions to be explored from history under exploration
    - `target_position:` current position of the target, this is encapsulated from the planning algorithm and is only used to have coherence between simulations.

  returns:

    - reward: long term reward of following a particular course of actions and observation during simulation

- `Class Observation_slice:` This class implements methodology for one of the approaches used in the partitioning of continuous observation space used in the thesis. The following is the method implemented in the class.
  - `observation_slice:` This static method partitions the observation space and associates a unique discrete observation label to each of the generated partitions.
    parameters:
    - observation: continuous observation from the real world
    - bandwidth: size of partition generated with the continuous observation from real world at the centre

### A.3.6 `History_sequence.py`

This file implements a class `History_sequence` which serves as a container class that is used to manipulate history sequence related data generated while simulating action-observation paths from history under exploration in the planning phase. The following are the methods of the class.

- update: This static method updates the long term reward for an action chosen from the history node under exploration as a part of internal record keeping. This record is used to choose an optimal action when the same history node is encountered at a later point in time in planning.
  parameters:
  - reward: long term reward for an action chosen for exploration.
- `reset_child_t_ha:` This static method is used to reset internal data containers 'child' and `t_ha` that are used to hold simulation related data.
- `reset_belief_particle:` This static method resets the internal list that holds particles added in the `KDE_Regression.simulate` method.

### A.3.7 Simulator.py

This file implements functionality of a policy playout and the black box simulator used to carry out simulations. The following are the methods implemented in this file.

- `simulator_static:` This method implements the black box functionality used to generate history sequences during simulation.
  parameters:
  - state: a state sampled from current belief state of the history under exploration
  - act: sampled action to be executed in the black box simulator
  - `target_position:` fixed position of the target for a given simulation

  returns:
  - `next_state`: the state resulting from taking the input action 'act' from the input state 'state'
  - observation: list containing [haptic, nf, cd]
  - reward: immediate reward obtained for taking the input action 'act' from the input state 'state'
- `playout_policy:` This method plays offline, a policy received in the from of a list of actions to replay the course of actions taken.
  parameter:
  - `policy_vector:` A list of optimal action recorded during online execution

### A.3.8 Utility.py

This file implements methods that serve as utility for the execution of the algorithm implementation. The following are the methods implemented in the file.

- `encode_position:` Encodes a 2D co-ordinate into a linear index
- `decode_position:` Decodes the linear index to return a 2D co-ordinate
- `get_valid_states:` Returns a list of valid 2D co-ordinates in relation to the simulation environment

Following are the container classes implemented in the file. These are used supplemetary to the main program execution

- Class `One_step_sim_data:` Holds the current time step distance between the manipulator and target in the simulation during planning
- Class Distance: Holds the current step distance between the manipulator and target in the real world

### A.3.9 `Terminal_State_Information.py`

This file implements functionality to recognize the terminal state. The following is the method implemented in this class.

- `is_terminal:` This is a static method that check if the current state in which the manipulator is, in the real world, is a terminal state or not. This serves as an indication to stop further exploration and return from the execution of the main program.
  returns:
  - Boolean values

### A.3.10 Policy.py

Container class to hold a list of optimal actions performed during online execution of algorithm.

### A.3.11 Tree.py

Container class to hold MCTS tree structure.