INVERSE LEARNING OF ROBOT BEHAVIOR FOR AD-HOC TEAMWORK

by

MAULESH TRIVEDI

(Under the Direction of PRASHANT DOSHI)

ABSTRACT

Machine Learning and Robotics present a very intriguing combination of research in Artificial Intelligence. Inverse Reinforcement Learning (IRL) algorithms have generated a great deal of interest in the AI community in recent years. However, very little research has been done on modelling agent interactions in multi-robot ad-hoc settings after learning is complete. Moreover, incorporating IRL for practical robot environments that deal with online learning and high levels of uncertainty is a challenge. While decision theoretic frameworks used for planning in these environments provide good approximations for computing an optimal policy for an agent, these model parameters are usually specified by a human designer. We describe a unique Bayesian approach to approximate unknown state transition functions. We then propose a novel multi-agent Best Response Model that plugs in the expert's reward structure learnt through Maximum Entropy Inverse Reinforcement Learning, and use the learnt transition functions from our Bayes Adaptive approach to compute an optimal best response policy for our multi-robot ad-hoc setting. We test our algorithms on a robot debris-sorting task.

INDEX WORDS: Inverse Reinforcement Learning, Markov Decision Process, Bayes Adaptive Markov Decision Process, Best Response Model, Dec MDP, Optimal Policy, Reward Function

INVERSE LEARNING OF ROBOT BEHAVIOR FOR AD-HOC TEAMWORK

by

MAULESH TRIVEDI

B.E., Birla Institute of Technology and Science, Pilani, 2012

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2016

INVERSE LEARNING OF ROBOT BEHAVIOR FOR AD-HOC TEAMWORK

by

MAULESH TRIVEDI

Major Professor:	Prashant Doshi
Committee:	Walter D Potter
	Khaled Rasheed

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2016

DEDICATION

Mummy, Pappa. This one’s for you..

M.P.D.B.B.C.M.M.R.G.S.S.A.K.P.G

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my major professor Dr. Prashant Doshi for guiding me throughout the course of my research. Thank you so much for motivating me and pushing me to achieve the best results. I would also like to express my sincere thanks to Dr. Walter Don Potter and Dr. Khaled Rasheed for not only being on my committee, but also for being wonderful teachers.

I would like to extend a very special thank you to Kenneth Bogert who is a PhD student at Thinc Lab under Dr. Prashant Doshi. Kenny, your deep insights and dedicated work ethic were a true inspiration and I will always remember how you always told me that “I learnt it the hard way, bud”. It has been my inspiration.

The Institute of Artificial Intelligence and the Thinc Lab at UGA, with all the staff and students also deserve a special acknowledgement. It was truly a remarkable experience working and enjoying with each and every one of you.

And last but not the least, a special shout out to all the wonderful friends I made over here, and all of you guys from back home. Special s/o to all my RG buddies, Athens buddies, School buddies, Society buddies, Soccer buddies and anyone I might’ve missed. Thank you for always being there for me. It means a lot.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND AND MOTIVATION

The turn of the century has been accompanied by an abundance of technology in our daily lives. Twenty years ago, mobile phones were devices used only due to professional need or prosperity. Today, we can proudly say that cell phones have found a way into the remotest of corners of the world. This technological advancement has seen us connect to the world around us with the push of a button, send man to the depths of the ocean and even set up manned space stations around our planet. Fortunately, it has also led to the resurgence of one of the most complex fields of science – Artificial Intelligence and Robotics.

For thousands of years, man has tried to get a deeper understanding of himself. Philosophers and scientists alike have spent their entire lives trying to figure out how "we" perceive, comprehend, predict and manipulate the world around us. The study of Artificial Intelligence goes a step further by attempting to recreate intelligent entities. Consequently, robots have come to be associated as the archetypal intelligent agent developed by humans. The main reason for this is the fact that "smart" robots not only have the ability to understand the world around them, but they have the physical capability to manipulate their surrounding environment as well.

This thesis predominantly deals with an emerging field of robotics known as probabilistic robotics that concerns with the use of statistical techniques for representing information and

decision making and planning under uncertainty [8]. In the last few years, probabilistic robotic techniques have become an influential paradigm for designing algorithms in robotics.

In the following section, we discuss some fundamental concepts on decision making and uncertainty that form the basis for our research.

1.1.1 Markov Decision Processes

In many real world scenarios, an agent needs to have the ability to make complex decisions. These decisions can be episodic (one at a time) or sequential (having a relationship between a current decision and a future one). Sequential decision making involves a rational agent trying to maximize its utility by taking into account the effect each action has on the future. Markov Decision Processes are a mathematical formulation that allow us to model sequential decision making tasks. Essentially a finite MDP is a tuple <S, A, T, R, γ,>, where

- S is a finite state of 'n' states
- A is a set of $\{a_1, a_1, \ldots, a_k\}$ k actions
- T(s, a, s') is the transition probability of reaching state s' by performing action 'a' in state 's'
- R(s, a) is the reward associated with taking action 'a' in state 's'
- $\gamma \in [0, 1]$ is the discount factor

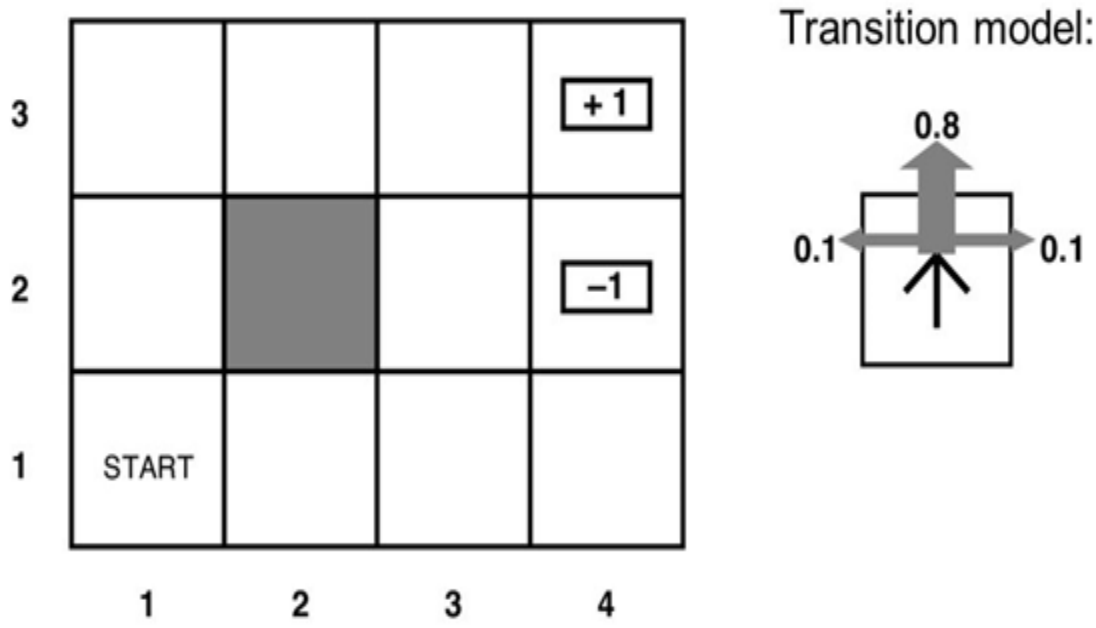


Figure 1.1 Grid World example for an MDP [8]

A Markov Decision Process assumes a 'Markovian' property. This means that the action taken in a particular state depends only on that state and not on the prior history of states. The above figure is a classic example of a grid world problem [8]. MDPs allow us to model the uncertainties involved in such environments. If this environment was strictly deterministic, then the optimal solution would be very straight forward: <Up, Up, Right, Right, Right>. However, in most real world cases, actions are highly stochastic, resulting in uncertain state transitions. In the above example, for instance, going ‘Up’ would result in the correct transition 0.8 percent of the times. This action would result in a going ‘Right’ or ‘Left’ around 0.1 percent of the times. This stochasticity is described by a transition model that assigns a probability to each state-action-state triplets.

Each state is also associated with an immediate reward function. We assume that in each state, the agent receives a reward for performing an action (R(s, a) → Real Number) which might be positive or negative. This reward function is defined for all the states except the terminal states.

Each time we execute a policy from an initial state, it would lead to a different trajectory of states depending on the inherent stochasticity of the environment. The optimal policy (The optimal mapping between [states → actions] for each state) should therefore incorporate this uncertainty while computing the utilities of each state. Thus, the optimal policy results in the highest expected utility. The utility for a state can simply be the sum of rewards the agent receives at each step.

The discount factor, γ, is a number between 0 and 1 that tells us whether an agent prefers immediate rewards over future rewards.

The optimal policy is thus defined as the maximization of the expectation of the discounted rewards for each state.

1.1.2 Value Iteration

Value iteration is one of the most commonly used algorithms for solving MDPs. The utility of each state in value iteration is defined as the expected utilities for all possible next states. Each state in the Markovian environment is assigned a utility function where the agent selects an action based on the principle of Maximum Expected Utility – the agent chooses an action which maximizes the expected utility of a future state. Hence, the utility of a particular state turns out to be the immediate reward plus the expected discounted utility of the next state.

Equation 1.1 is known as the Bellman equation. The value iteration starts with a random set of values for the initial utility for each state. With an arbitrary $U_t(s)$, we calculate $U_{t+1}(s)$ using the Bellman update equation as follows:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s') \quad (1.1)$$

For each time step, we calculate the Bellman update associated with each state and update the corresponding utilities. With enough iterations, the utilities reach a point of convergence and the policy extracted turns out to be the optimum policy for that state.

Figure 1.2 describes the value iteration algorithm as described in [8].

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\ U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Fig. 1.2 Value Iteration for calculating the utilities of each state in an MDP [8]

Markov Decision Processes are highly coveted tools for planning and decision making in stochastic environments. All learning algorithms discussed in this thesis have a base assumption that the underlying structure of the agent's environment is modeled by an MDP.

1.1.3 Motivation for Inverse Reinforcement Learning

Now that we have described a way of modelling uncertainty, let us look into some of the principal learning algorithms used in these domains. One of the most dominant algorithms used by robots in complex unknown environments is Reinforcement Learning (RL). Reinforcement Learning is a sub-field of machine learning that deals with how a rational agent ought to take actions in an environment so that it maximizes a cumulative reward associated with that environment. For instance, in football, a complete pass might generate a positive reward while an incomplete pass might generate a negative reward. It is important to note that reinforcement learning frameworks consider the rewards to be hard-coded inside the system such that receiving a particular set of input signals might trigger a positive or negative reinforcement. The goal of reinforcement learning techniques is to use these rewards to learn an optimal policy for the environment by deploying agents that sense and act upon this said environment. One of the main motivations for using this technique arises from the use of reinforcement learning as a computational model for human and animal learning [9]. This research is supported by both the behavioral studies and the neurological evidence associated in bee foraging and animal training However, it should be noted that while observing human and animal behavior, we must consider this "reward function" to be an unknown variable; one that can only be determined through empirical observations. This pre-requisite is particularly true when we consider multi-attribute reward functions. For example, a simple model based on bee foraging might imagine that the rewards are simply the nectar content of the flowers, while in reality, a bee might follow a reward structure that weighs in time, distance, wind speed and a possible risk of predators in addition to nectar quantity [9]. Furthermore, it becomes even harder to determine the relative magnitudes of these features *a priori*.

Thus, our research addresses the problem of Inverse Reinforcement Learning (IRL), that is, the problem of learning the reward function, given the optimal trajectory followed by an "expert". An agent designer might only have a very generic idea about the optimum reward structure that the robot should follow in order to achieve a desirable behavior. As such, conventional reinforcement learning techniques become mute in these situations. Inverse Reinforcement Learning techniques employ the use of other "expert agents" – agents that have a fixed reward structure or follow a deterministic optimal policy – similar to imitation learning. However, in the case of imitation or apprenticeship learning, the purpose of collecting observations from the expert is to learn a policy for a specific environment, i.e., imitation learning would generate a direct mapping from states to actions (π: S $\rightarrow$ A) irrespective of the underlying reward function. One of the main advantages of using the underlying reward structure of the expert is the fact that reward functions are transferrable definitions of observed behavior, which is to say that, once learnt, they can just as easily be used in a different environment depending on the learner's preference. In fact, one of the key presuppositions of RL techniques is that the reward function, and not the policy, is the most robust and succinct definition of a given task [1].

1.1.4 Related Work

Inverse Reinforcement Learning is a relatively new branch of machine learning. While research has been done into improving the underlying algorithm [3] [4], very few papers deal with IRL and interactions. Ziebert et al. [17], model interactions between learning agents in a unique way. They derive a novel graphical framework that takes into account sequential interactions, latent information and feedback to generate influence diagrams for statistically modelling inverse optimal control tasks. This is however difficult to emulate in online robot settings. Bogert & Doshi on the other hand propose an innovative mix of inverse reinforcement learning and game theory

that allows sparse interactions to be modelled as multiple agents playing a "game" [4]. This is quite a fresh approach to the problem but it faces some criticism for how it deals with key assumptions regarding the underlying game played by each agent.

IRL algorithms generally assume known environmental dynamics. This means that any learning agent in a classical IRL setting, is completely aware of the underlying state transition function. Removing this assumption poses a hard challenge. Bogert & Doshi and Herman et al. offer some original perspective into this problem [24] [25]. The former describes a new approach that maps stochastic transitions to a distribution over features [24]. They explicitly focus on estimating the unknown dynamics prior to learning and test their algorithm on a damaged robot patroller problem. On the other hand, Herman et al. quite recently proposed a different approach that simultaneously learns the reward structure and the unknown dynamics of the system [25]. The authors use a gradient based approach that takes into account the inherent bias of a system to simultaneously estimate all the unknown parameters of the system.

Alternatively, researchers in decision theoretic frameworks propose a different way to estimate unknown transition functions of a Markovian system. Bayesian Reinforcement Learning has been a prominent area of research in the recent past [21] [22]. Algorithms that employ statistical tools to estimate unknown system dynamics have been widely researched [6] [7] [16]. However, these methods have not been used in conjugation with robot applications. Additionally, these algorithms have not been developed to work with multi-agent inverse reinforcement learning domains.

Conversely, to the best of our knowledge, no research has been done on ad-hoc scenarios that would result from a "learner" trying to team up with an "expert" in the near future, resulting in a team-based ad-hoc system. Ad Hoc frameworks exist that allow agents to learn their teammates' task [18] [20] but do not use the inverse reinforcement learning approach. Research in

these domains, as mentioned earlier, has been quite sparse, and existing algorithms have either not been tested on complex robot applications, or have had a hard time adapting to teammates which have a dynamic policy. We believe that using inverse learning techniques coupled together with a principled way of dealing with unknown model parameters will allow us to deal with such multi robot ad hoc scenarios in a much better way. Keeping this in mind, our contributions are two-fold.

1.2 CONTRIBUTIONS

First, we propose a framework for modelling future interactions using inverse reinforcement learning. By 'future', we mean to say that the learning agent has solved the IRL problem and has the complete underlying reward structure that matches an expert's demonstrated trajectory. The "learner" must now join the "expert" agent in order to form an ad-hoc team, keeping in mind that it must deal with interactions that were not part of the expert's observed trajectory and take into account that the expert does not have a way to model multi-agent scenarios.

Many real world scenarios that include localization and path planning, involve uncertainty arising from the prediction of the underlying system behavior as well as the observations of the system. This is problematic, especially, when most IRL algorithms assume known model parameters when dealing with an expert. We propose a novel approach that allows a learner to create a distribution over the state transition functions resulting in a system where the agent can efficiently model future interactions, i.e., the transition probabilities associated with a multi agent setting without any prior knowledge over the same. These parameters are rarely known and are usually estimated by a designer. As such, in highly complex environments, the resulting MDP/POMDP solution might result in a sub-optimal policy. We therefore seek to incorporate this uncertainty over model parameters during planning, and have our agent learn these unknown parameters through experience. The techniques used in this thesis have, to the best of our

knowledge, never been used in a robotics domain. We hope to provide a practical exhibition of incorporating these algorithms in a complex real world online environment.

Our second contribution is developing a multi-agent framework where the learner and expert form an ad-hoc team. This requires an important underlying assumption. While the learner conscientiously knows the existence of an expert in the system, the expert has no idea of another agent in the environment. We refer to situations where, for example, the expert might be an assembling robot in a manufacturing company. It has all the capabilities of performing its pre-determined task optimally, but any interference is simply treated as "noise". We have a similar domain for our experiments, where the expert robot is tasked with separating out debris into different containers. It follows a deterministic policy but it associates any interference as noise in the environment. The learner must therefore take into account the fact that the expert will have no knowledge of its existence and come up with an optimal policy that models the two agent system as a "best response model" (BRM). We call our implementation a Bayes Adaptive Decentralized Markov Decision Process that gives us an optimal best response policy for our learner that maximizes a cumulative team reward while accounting for model uncertainties associated with a multi-agent setting.

Apart from the algorithmic contributions outlined above, we have another contribution relating to the ROS (Robot Operating System) library developed for testing out our algorithm. We have developed a robust code base for implementing motion planning, navigation and an efficient pick-n-place functionality for a TurtleBot with phantomX robotic arm. The library also includes packages to incorporate multiple robots in a classic Markovian environment defined as a tuple of $<S, A, T, R, \gamma,>$.

1.3 OUTLINE

This thesis consists of five chapters. Chapter 1 outlines a basic introduction and motivation for our work. We start off by defining Markov Decision Processes (MDP) and their uses in artificial intelligence and robotics and go on to provide reasons for using IRL in our application domain. Chapter 2 goes further into Inverse Reinforcement Learning and the research associated with this area of machine learning. We define Inverse Reinforcement Learning as a problem of machine learning in Markovian processes. There has been a lot of research into algorithms for solving inverse reinforcement learning problems. We formulate the problem as a non-linear optimization, subject to constraints of feature matching. We present derivations for solving our non-linear objective function by Lagrangian techniques. We go on to approximate this solution and provide a mathematical argument for doing the same.

Chapter 3 deals with Inverse Reinforcement Learning for Ad-Hoc teaming of robots. We formulate our problem of IRL + interactions as a Decentralized Markov Decision Process with the learner trying to maximize the cumulative team reward. We propose a mathematical formulation of the Bayes Adaptive framework for our Dec MDP that uses Dirichlet distributions to model uncertainties over the state-action transition functions. We go on to define our unique approach for multi-agent ad-hoc domains by defining the internal framework of the learner as a best response model that deals with state uncertainties. We provide a sound mathematical structure to our framework and describe an algorithm to solve it exactly.

In Chapter 4, we discuss the experimental setup, design and implementation of the algorithms described in the previous chapters. We deploy our inverse learning algorithms in simulation and on a physical robot using the Robot Operating System (ROS). Our algorithms have been tested out on the popular debris sorting task where we use TurtleBots with attached robotic

arms for pick-n-place functionalities. We have developed a library that allows the user to manipulate the TurtleBot arm in simulation and physical environments. We analyze the results from different experimental settings and postulate on the practical applications and limitations of using IRL + AdHocInt in real world scenarios. Finally in chapter 5, we summarize our entire work and talk about the future scope of this research.

CHAPTER 2

INVERSE REINFORCEMENT LEARNING

2.1 INTRODUCTION AND BACKGROUND

We consider the problem of learning the behavior of a robot following a fixed optimal policy in a known environment. Abbeel and Ng [1] introduced a new paradigm for solving problems of imitation learning via inverse reinforcement learning. We address the problem of machine learning in a Markovian environment defined by a tuple <S, A, T, R, γ,> where the Reward function R(s, a) is unknown. The process of recovering this reward function is known as Inverse Reinforcement Learning. IRL proposes to learn the true reward function by collecting observations from an "expert" performing a task we wish to emulate. This inverse learning problem seeks to retrieve the rewards that make a subject agent exhibit a near optimal policy that closely mimics the behavior demonstrated by an expert. For the rest of this chapter and this thesis, we will call the agent demonstrating some desired behavior as the 'Expert', while any subject agent trying to imitate the expert will be called the 'Learner'.

Informally, the Inverse Reinforcement Learning (IRL) problem can be defined as follows [2]:

Given: 1) The measurements of an agent's behavior over time, and in a variety of circumstances

2) If needed, the measurements of the sensory inputs to that agent

3) If available, a model of the environment.

Determine: The reward function being optimized

It is important to note the difference between Inverse Reinforcement Learning and Imitation Learning. Imitation Learning seeks to copy the demonstrated trajectory in a given environment as closely as possible. Rather than strictly copying the actions performed by an expert, the aim of apprenticeship learning and inverse reinforcement learning is to recover the rewards under which the agent's demonstrated policy is optimal.

2.2 ALGORITHMS FOR INVERSE REINFORCEMENT LEARNING

The main objective of Inverse Reinforcement Learning is to search for an optimal reward structure that explains the observed behavior of an expert. The reward function is by far the most robust, succinct and transferrable definition of a task [2]. It is imperative, therefore, that we have a large and diverse set of expert trajectories, so as to extract the reward function, taking into account as much information as possible. However, an important issue to deal with while trying to solve IRL is *degeneracy*, the existence of a large set of reward functions that match the observed policy of an expert. For instance, a reward function of zero would inherently result in a solution but clearly it would be incorrect. To remove degeneracy, Ng and Russell suggest heuristics that maximally differentiate an observer policy from some other sub optimal policy. They solve the subsequent IRL problem using an efficient linear programming formulation [2].

The basic concept of Inverse Reinforcement Learning was first introduced by Ng and Russell in [2]. They describe three basic algorithms for solving the IRL problem. The first two assume that the entire policy is known. The authors first derive an algorithm to handle a tabulated form of reward function on a finite state space. The resulting solution characterizes the set of all reward function for which a given policy is optimal. This set contains many degenerate solutions and Ng and Russell try to tackle this problem by proposing a simple natural heuristic which results in a linear programming optimization solution to the inverse reinforcement learning problem. The

authors go on to illustrate our learning problem for large, potentially infinite state spaces and use linear function approximation to solve the new linear programming problem. The final section of the paper deals with a more realistic case where the policy is unknown, but the learning agent has access to a set of expert trajectories from the actual environment. They collect a set of trajectories and the aim of the algorithm is to find the unknown parameters that make the reward function satisfy some optimality criteria. The resulting optimization function is again solved as a linear program.

Abbeel and Ng [1] on the other hand, came up with a novel approach for solving IRL using feature matching. They proposed a new definition for the reward function to be optimized, saying the R(s, a) can be considered as a linear combination of feature functions. Experiments show that, while they may not recover the "true" reward function, the policy resulting from the extracted reward performs as well as the expert's policy, which follows an unknown reward structure.

They demonstrate that matching feature expectations between an expert's observed policy and the behavior of the learner is enough to achieve the same performance as the expert, if, the agent is following a Markov decision process with a reward function with those same exact features linearly summed together [1]. Their algorithm terminates in a small number of iterations and the authors guarantee that the learnt policy will perform just as well or even better than the expert's optimal policy. While this approach is a very good estimate of the observed behavior of the expert, needless to say, it suffers from *degeneracy* that is, having a large set of reward functions for which a given policy is optimal.

Unfortunately, this means that both, the Inverse Reinforcement Learning problem, as well as matching the feature counts, are vague in their crude sense. Not only can each policy be optimal for a large number of reward functions (ex. all zeros), but if sub optimal behavior is demonstrated by an expert, then feature counts can only be matched by a mixture of policies. This is a highly ambiguous problem and critical if the concept of inverse learning were to be applied to real world scenarios.

Zeibart et al. [5] proposed a unique solution to this dilemma. They developed a probabilistic model based on the Principle of Maximum Entropy to elevate the problem of degenerate solutions in IRL while guaranteeing the performance standards set by previous algorithms. We detail their Max-Ent algorithm in Section 2.3 and go on to formulate our problem as a structured apprenticeship learning setup.

2.3 MAXIMUM ENTROPY INVERSE REINFORCEMENT LEARNING

Ziebert et al. [5] propose a novel approach to deal with the ambiguity arising from degenerate solutions in a very principled and mathematical way. The authors define a highly probabilistic technique to deal with the uncertainty arising in problems of imitation learning. They formulate the problem as that of finding a distribution over all deterministic policies. In order to resolve the ambiguity in selecting the correct distribution, the authors employ the principle of maximum entropy.

The entropy of a state is the amount of information contained within a state. The more random or unbiased a state, the more entropy it will contain. The principle of maximum entropy gives us a distribution over input probabilities, given specified constraints. It is as unbiased as possible because it does not confer to any particular input other than the one that matches its constraints. In the case of Inverse Reinforcement Learning, Ziebert et al. [5] proposed employing

the use of the principle of maximum entropy while being constrained to match the feature expectations proposed by Ng and Abbeel in [1].

Maximizing the entropy for a distribution over policies, while matching the observed behavior of an expert, means that we essentially remove all inconsistencies resulting from other approaches for IRL and provide a principled and mathematically sound approach to generate a distribution over the set of all policies for our MDP.

The principle of maximum entropy in IRL simply means that, given the constraints of matching feature expectations between the learner and the expert, the simplest policy that best matches the expert's observations is the one that has the maximum entropy [3, 5]. Ziebert et al. [5] applied this model to the domain of driver training and showed that MaxEnt IRL provides a convex formulation for the problem of learning a reward function while maintaining efficiency and performance. Let us now formulate our problem as apprenticeship learning via inverse reinforcement learning as defined in [3].

Structured Apprenticeship Learning [3] can be formulated as the problem of finding a probability distribution π over a set of deterministic policies that has the highest entropy subject to matching the feature expectations of the expert and the learner.

A key assumption in apprenticeship learning and IRL is that the reward function is defined as a linear combination of k feature vectors $\emptyset_k$ with weights θ_k,

$$\forall(s, a) \in S \times A: R(s, a) = \sum_{k=1}^{K} \theta_k \emptyset_k(s, a) \quad (3.1)$$

We define the expected return of a policy $J(\Pi)$ as the expected sum of rewards received when following a policy Π [3]. It can be written as:

$$J(\Pi) = \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t \emptyset_k(s_t, a_t) \mid \mu_0, \Pi, T\right] \quad (3.2)$$

Conversely, an optimal policy Π* maximizes this expected sum of rewards and we write it as,

$$\Pi^* \in \arg\max_{\Pi} J(\Pi) \tag{3.3}$$

As discussed above, we now wish to find the simplest policy that has the maximum entropy while matching the expected return of demonstrated rewards. Essentially, we wish to say that the feature counts of the learned IRL policy must be equal to the feature counts demonstrated by an expert. Mathematically, we can write this as follows:

$$\forall k \in \{1, 2 \ldots. K\}: \mathbf{E}\left[\sum_{t=0}^{\infty} \gamma^t \emptyset_k(s_t, a_t) | \mu_0, \Pi, T\right] = \hat{\emptyset}_k \tag{3.4}$$

Here, 'K' is the number of features in the environment. The left hand side of the equation denotes the feature counts of the learned policy whereas $\hat{\emptyset}_k$ specifies the empirical expectation of some feature 'k' that is calculated from the expert's demonstrated trajectories.

The Max-Ent Inverse Reinforcement Learning algorithm defined by Ziebert et al. basically involves finding the optimal weights θ of a policy π that maximizes the entropy over the distribution of policies, where the policy space is defined as $|A|^S$.

Thus, we can define the problem statement as follows:

$$\max_{P, \mu^{\pi}} \left(- \sum_{\pi \in A^{|S|}} P(\pi) \log P(\pi) \right)$$

Subject to:

$$\sum_{\pi \epsilon A^{|S|}} P(\pi) = 1, \tag{3.5}$$

$$\forall \emptyset_k: \sum_{\pi \epsilon A^{|S|}} P(\pi) \sum_{s \epsilon S} \mu^{\pi}(s) \emptyset_k\big(s, \pi(s)\big) = \hat{\emptyset}_k$$

The state-visitation frequency, $\mu^{\pi}(s)$, is defined as the likelihood of visiting a particular state, given a uniform initial distribution over the entire state space and a policy. It may be computed using the following approach:

$$\forall \pi, s: \mu^{\pi}(s') = \mu_0(s') + \gamma \sum_s T(s, \pi(s), s') \mu^{\pi}(s) \tag{3.6}$$

As is evident, computing $\mu^{\pi}(s)$ is a problem that takes the initial state distribution and calculates the probability of transitioning into a particular state s' from all possible states in our environment.

The Lagrangian for our problem statement is given by:

$$L(P, \eta, \theta, \lambda) = -\sum_{\pi \epsilon \pi^{|S|}} P(\pi) log P(\pi) + \eta \left(\sum_{\pi \epsilon A^{|S|}} P(\pi) - 1 \right) + \sum_k \theta_k \left(\sum_{\pi \epsilon A^{|S|}} P(\pi) \sum_{s \epsilon S} \mu^{\pi}(s) \emptyset_k \left(s, \pi(s) \right) - \hat{\emptyset}_k \right) \tag{3.7}$$

Therefore, we can say that,

$$\partial_{P(\pi)} L(P, \eta, \theta, \lambda) = \sum_s \mu^{\pi}(s) \sum_k \theta_k \emptyset_k \left(s, \pi(s) \right) - log P(\pi) + \eta + 1 \tag{3.8}$$

The optimal solution satisfies the equation $\partial_{P(\pi)} \mathcal{L}(P, \eta, \theta, \lambda) = 0$ then,

$$log P(\pi) = \sum_s \mu^{\pi}(s) \sum_k \theta_k \emptyset_k \left(s, \pi(s) \right) - log Z(\theta, \lambda) \tag{3.9}$$

Where, Z (θ, λ) is a partition function. Hence, we can say that,

$$P(\pi) \propto \exp \left(\sum_s \mu^{\pi}(s) \sum_k \theta_k \emptyset_k (s, \pi(s)) \right) \tag{3.10}$$

We end up with the following functions defined over specific Lagrangian multipliers.

$$\frac{\partial L}{\partial \theta} = \sum_{\pi_I \epsilon \Pi_I} Pr(\pi_I) \sum_k \sum_s \mu_{\pi_I}(s) \emptyset_k (s, \pi_I(s)) - \hat{\emptyset}_k \tag{3.12}$$

$$\frac{\partial L}{\partial \Pr(\pi_I)} = \sum_s \mu_{\pi_I}(s) \sum_k \theta_k \emptyset_k (s, \pi_I(s) - log Pr(\pi_I) + \eta - 1 \tag{3.13}$$

We solve this equation using Lagrangian relaxation techniques by bringing together the objective function and the constraints into one non-linear program. Abbeel and Ng [1] propose a novel way to compute the partition function associated with the Lagrangian for the above problem. Essentially, we wish to learn a Markov Decision Process whose optimal policy matches the observed actions of the expert. We can approximate the Lagrangian derivative of the policy under consideration by summing over the entire set of state-action trajectory of the expert and comparing the Q value of the expert's action with the optimal value of the policy under consideration [2, 4]. Essentially, the Q(s, a) denotes the action value of taking the observed action 'a' of the expert, and then following the optimal policy π_i after that. $V\pi_i$ (s) is the optimal utility of the learner's policy under consideration. Hence, we wish to compare the utility associated with comparing the expert's optimal action in a particular state with the utility of performing our optimal policy in the same state. $V\pi$ (s) maximizes the action value of a state. If the weights under consideration are approximately correct, then the optimal action of the expert should match the optimal action 'a' of the learner, which is obtained after the value iteration converges. Hence, when $a = \pi(s)$, then $Q_\pi(s, a) - V_\pi(s) = 0$. Since we wish to match the expert's optimal actions for all observed trajectories, we wish to minimize the function [Q(s, a) – V(s)] for all (s, a) in the trajectory [4, 2]. Abbeel and Ng [1] propose and defend a hypothesis that states that the derivative of the Lagrangian (for the objective function of MaxEnt) will be approximately equal to the summation over Q minus V, with respect to the policy under consideration. If that policy has a sub-optimal action, then Q(s, a), - following the expert's observed action 'a' - will be less than V(s) for the policy, thereby resulting in a negative Q minus V. It is important to know that this value can at best be equal to zero, when the action 'a' matches the policy.

Even for a small environment, the space of all policies increases exponentially by a magnitude of $A^{|S|}$. Hence, it is crucial to come up with a good approximation to solve the objective function of the MaxEnt IRL problem. The approach detailed above is a good method - however, it may also result in a sub-optimal action whose Q value is the same as the optimal value. This is why this method is an approximation. If the policy under consideration fails to match the observed action of the expert, the difference of Q minus V would be negative and this effect would be added up further along the trajectory.

In the end, solving MaxEnt IRL boils down to minimizing the objective function. In our case, the objective function is a sum of the Lagrangian derivatives with respect to the weights and the policy under consideration. The objective function may have an optimal solution as a saddle point in the Lagrangian space. We therefore use numerical optimization techniques like Nelder-Mead, genetic algorithms and hill climbing in order to find the minima or maxima. However, we may modify the Lagrangian slightly by squaring the Lagrangian derivatives so that the revised objective function is optimized to find the global minima [3].

Let us now briefly discuss the implementation structure of the Maximum Entropy Inverse Reinforcement Learning algorithm in detail. We start off by defining the expert's reward function as a linear combination of feature functions. For simplification purposes, these features can be binary representations. The next step involves collecting sample trajectories from the expert by observations. Ideally, we need to collect a diverse sample set encompassing a variety of states. We initialize our optimization function by randomly assigning arbitrary values within specific bounds (for instance [0, 1]) to each of our features. These are essentially the weights for the features defining the expert's reward structure. The reward function is defined by a linear combination of these binary feature functions.

We perform value iteration using these weights until we converge to an optimal set of utilities and a fixed policy. We need to now approximate the derivative of the Lagrangian w.r.t. the policy under consideration by summing over the length of the expert trajectory and finding the difference between the optimal action of the policy under consideration and the optimal action of the expert [2, 4]. We now calculate the state visitation frequency, $\mu^{\pi}(s)$, by forward programming. This corresponds to the number of times a state 's' is visited by following the policy π. Note, that we initialize $\mu^{\pi}(s)$ with a zero for all states except for a start state. The transition function contains all the information necessary to calculate the likelihood of state visitations. Additionally, we need to compute this state-visitation frequency the exact number of iterations as the expert's trajectory. This is a crucial step as it would give us a very good approximation over the state-visitation counts of the expert and the learner.

Equation (3.13) describes the partial derivative of the Lagrangian with respect to the weights. Essentially, we wish to find the likelihood of the weights under consideration by comparing the feature expectations of the expert with that of the learner. The feature expectations of the expert are defined as the cumulative sum of observed features over the entire length of the trajectory. On the other hand, we obtain the feature expectations of the learner by summing out over the expectation of the feature function for the policy which is essentially given by the equation: $\sum_s \mu_{\pi_I}(s)\, \emptyset_k\,(s, \pi_I\,(s))$

Equation (3.12) defines the partial derivative of the Lagrangian with respect to the policy under consideration. $log\ Pr\ (\pi_I)$ is estimated using the following approximation [2, 4]:

$$log\ Pr(\pi) = \frac{\delta L}{\delta\ \Pr(\pi_I)} = \sum_{(s,a)\epsilon traj.} Q_{\pi_I}\,(s,a) - V_{\pi_I}(s) \qquad (3.14)$$

We may now square the Lagrangian derivatives so that the revised objective function is optimized to find the global minima and subsequently plug in these two partial derivatives into a function minimizer (Nelder-meads, genetic algorithm, etc.) and let the optimization proceed normally. After convergence, we end up with a set of weights that relatively match the feature weights of the expert for some demonstrated trajectory.

2.4 SUMMARY

Chapter 2 introduced the concept of Inverse Reinforcement Learning. We discussed the key motivation behind this algorithm and described the importance of learning the reward structure over a policy. Section 2.2 went on to detail some of the first few algorithms to solve the IRL problem and we discussed their shortcomings as well.

We then defined the Maximum Entropy IRL approach detailed in Ziebert et al. [5] and formulated our domain as a structured apprenticeship learning problem using maximum entropy inverse reinforcement learning. We expanded our problem statement and derived the solution to our objective function using Lagrangian relaxation techniques. In the end, we gave a detailed description for implementing the Max-Ent IRL algorithm using approximation techniques.

CHAPTER 3

INVERSE REINFORCEMENT LEARNING FOR AD HOC TEAMING OF ROBOTS

3.1 INTRODUCTION AND BACKGROUND

We consider the task of learning the behavior of an expert robot working in a known environment and subsequently joining it to form an ad hoc team in order to make the underlying task completion faster and more efficient. Existing research on multi-agent teamwork settings usually assumes a standard communication protocol between different agents. We present an ad hoc scenario where the two robots do not have any communication guidelines, yet must work together to accomplish their common goals.

We believe that the field of robotics serves as a strong test base for Inverse Reinforcement Learning (IRL) algorithms and their practical applications. Previous work on IRL has focused on developing techniques to reduce the fundamental assumptions of the learning algorithm and to apply it to other stochastic settings. Research on multi-agent inverse reinforcement learning has thus been very sparse. We propose a novel framework for learner robots that need to team up in unknown environments without the awareness of existing expert agents. Our proposed framework involves a “Learner” robot attempting to team up with an “Expert” robot that has no idea about the existence of another agent in the environment. We stem our motivation from disaster management situations where learner agents with the essential physical and cognitive capabilities can assist other robots in their local environment after they finish their primary tasks; thereby making the entire process much more resourceful. A key assumption in our scenario is the fact that

the expert has no knowledge of another agent in the environment and any interference from human or a robot is simply treated as noise. As such, the learner needs to compute an optimal best response policy that takes into account transitions that have yet to occur, all the while trying to maximize a cumulative team reward for the expert's goal(s).

Chapter 3 goes into detail about the main contributions of our research. Section 3.2 describes a Bayesian approach to approximating unknown transition functions. We then propose a multi-agent best response model that plugs in the expert's reward structure learnt through Maximum Entropy Inverse Reinforcement Learning and use the learnt transition functions from our Bayes Adaptive approach to compute an optimal best response policy for our multi-robot ad-hoc setting.

3.2 BAYES ADAPTIVE MARKOV DECISION PROCESS

Robotic environments generally deal with a high level of noise and uncertainty both, in terms of predicting the system's behavior, as well as the observations of the state of a system. Markov Decision Processes and their family of frameworks are powerful tools that allow us to deal with such high levels of uncertainty by providing a principled way to calculate the optimal policy an agent should follow in order to maximize its long term utilities in this environment.

However, a major drawback for these frameworks is that the MDPs and POMDPs generally assume a known model for the system, thereby using a predefined set of parameters for state transition functions. Knowing the exact model parameters *a priori* is rarely going to be the case, especially in robotic scenarios where sensor and actuator uncertainty can never be precisely known. As such, these parameters, even if estimated by a human designer are rarely accurate. We therefore seek a decision theoretic planning algorithm that accurately models the distribution over

unknown transition functions so that the resulting policy that we get by solving MDPs/POMDPs is optimal for our environment.

Bayesian Reinforcement Learning has explored this problem for the fully observable [7], as well for the partially observable Markov Decision Processes [6]. We draw inspiration from the research on Bayes Adaptive POMDPs proposed in [6] in order to estimate the unknown transitions that our 'learner' must approximate when dealing with possible interactions with the 'expert' in an ad-hoc setting. The following sections detail the mathematical formulation for our approach.

3.2.1 Bayesian Parameter Learning

Let us consider a simple example of batting averages in baseball [11]. A batting average is defined as the number of times a particular player gets a hit divided by the number of times he bats. This is essentially defined on the interval [0, 1]. Now let us consider a scenario where we wish to predict the batting average of a player for the entire season. Using procedures like maximum-likelihood learning is a problem because if the player goes up to bat for the first time and gets a hit, his average is briefly 1.0, i.e. maximum likelihood will return 1.0. While on the other hand, if the batsman strikes out, his average becomes 0.0. A player may even go out and get a strike out three to four times in a row at the start of the season. Yet, this is not a good indicator of how well the player might perform over the course of an entire season. Note that in baseball, on average, a player has a batting average between 0.20 on the lower end and 0.35 on the upper end. Rarely do players perform better or worse than these averages in a given season. So the question now arises, how do we predict these averages given that a single hit or a miss, or even a small streak of hits and misses is not a good indicator of how well a batsman will perform throughout a season?

Beta Distributions are a family of continuous probability distributions that allow us to model a "distribution of probabilities". More intuitively, Beta distributions allow us to define prior distributions over all the possible values of probabilities. It is a versatile way for modelling a probability over probabilities. A beta distribution is parametrized by two hyper parameters α and β. These are shape parameters that control the shape of the distribution. The Beta, or more precisely, the Probability Density Function (pdf) of a Beta distribution is represented by [8]:

$$\text{Beta}\,[\alpha, \beta]\,(\theta) = N\,\theta^{\alpha-1}\,(1-\theta)^{\beta-1} \tag{3.1}$$

It is defined for a random variable θ, between [0, 1]. 'N' is the normalization constant that integrates this equation to 1 (i.e. it ensures that the probability sums up to 1) and it depends on the value of the two hyper parameters. The mean of the beta distribution is given by (α/α+β). Larger values of the numerator α indicate a stronger belief over the value of the random variable θ, while a larger value of the denominator (α+β) reflect a more peaked distribution, suggesting greater certainty about the value of θ under consideration. However, the most interesting property of beta distributions that make them most suitable for our distribution (like our batting average scenario of success/failure) is that in Bayesian statistics, Beta distributions are the conjugate prior distribution to Binomial distributions like our batting averages. In simple terms, this means that if a random variable θ follows a Beta prior distribution with two hyper parameters (α, β), then, after a new data point is observed, the posterior distribution for the random variable θ is also a beta distribution [8].

Let us now have another look at our batting averages problem domain [11]. Please note however that these numbers are merely conjectures and might not reflect the true distribution over batting averages. We established a while ago that generally in baseball, a player has a batting average between 0.20 on the lower end and 0.35 on the upper end. With a few trials, we were able

to establish the shape hyper parameters (parameters that control the shape of a distribution) that model such a distribution (using the Kelsan online beta distribution chart calculator):

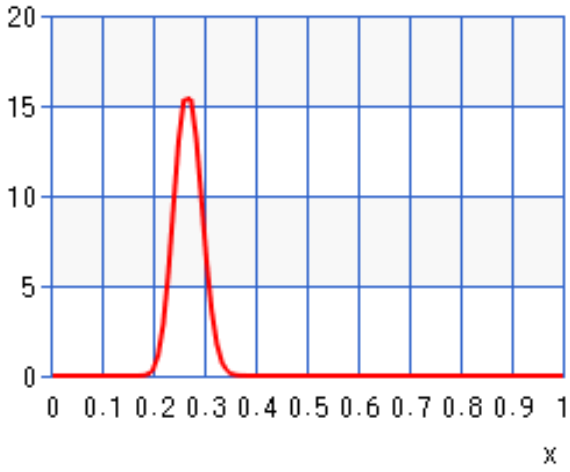


Figure 3.1 A Beta distribution with (α, β) = (80, 220)

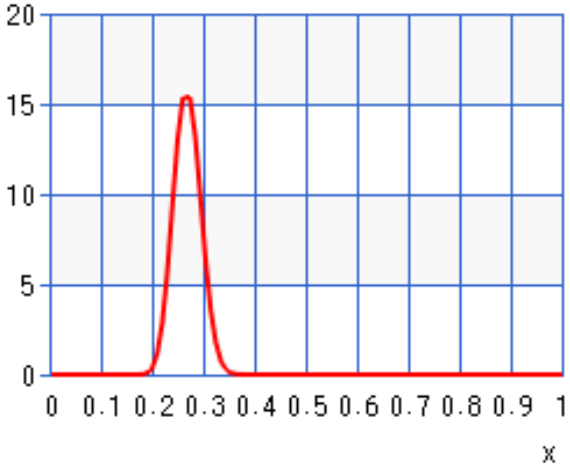


Figure 3.2 A Beta distribution with (α, β) = (81, 220)

The mean of a Beta as discussed before is given by (α/α+β). For the above example, the resulting mean is approximately 0.266. This means that, we expect a player's batting average throughout a season to be, to be most likely equal to, 0.266, but, with a small range from 0.200 to 0.350. The x-axis in Figure 3.1 represents a player's batting average (which is, essentially, just a probability of hits and misses!). The y-axis on the other hand represents the probability density

function. Thus, our Beta distribution is representing a distribution of probabilities over probabilities.

Now, let us understand why Beta distributions are so good for handling such scenarios. We assumed that prior to stepping out for the first time on the pitch, the batting average for any given player is represented by the above distribution. Now let us imagine that he scores a home run on the first time at the bat. We need to incorporate this new data and generate a new beta distribution that models the batting average while taking into account this new piece of information.

The update function for a beta distribution is defined as follows [8]:

$$\text{Beta}\,[\alpha', \beta'] = \text{Beta}\,[\alpha_0 + \text{successes}, \beta_0 + \text{misses}] \tag{3.2}$$

Hence, for our scenario, we need to incorporate this new 'hit' or success into our beta. Our new beta function now becomes Beta (80 + 1, 220).

Note the minimal change of the resulting plot (Figure 3.2), drawn out after incorporating the new piece of information. The change is so small that it is practically invisible to the human eye (Compare Figure 3.1 and Figure 3.2). The graph above shows a beta that is mathematically different from Figure 3.1 but visually, it is the same. The reason for this is simple – one single hit does not mean anything at all. This goes well with our hypothesis about predicting the average of a batsman throughout the season. We prefer making predictions after some consistent performances, while we reserve ourselves from judgements made over a small dataset.

So now, let us consider a couple of future scenarios which will affect the curve reasonably. Consider, for example, a player who has scored 50 hits (so far) and missed a 100.

This can be drawn out using a beta distribution as follows:

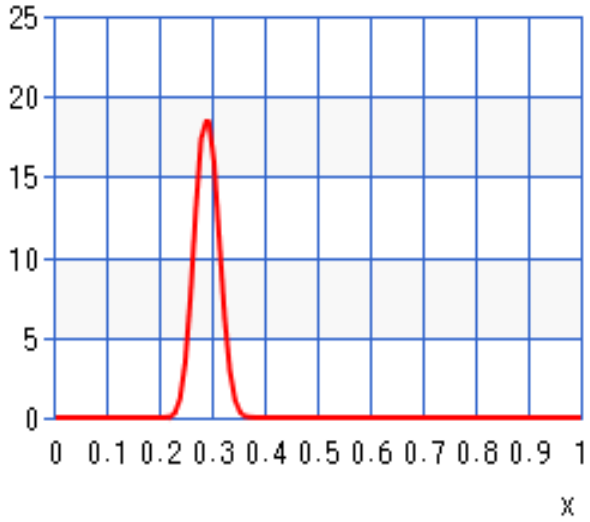


Figure 3.3 A Beta distribution with (α, β) = (80 + 50, 220 + 100)

Notice the slightly thinner curve which has also shifted a bit more towards the right. If we calculate the mean of this new beta distribution using the formula (α/α+β) we get a value of approximately 0.3. Thus, we can say that we have a more realistic approximation over the player's performance over the course of the season.

Thus, Beta distributions are a very good method for representing an accurate probability distribution of probabilities. Figure 3.4 illustrates how the Beta distribution varies for different value of (α, β). As we get more and more information, the beta distribution converges to the true value of the model.

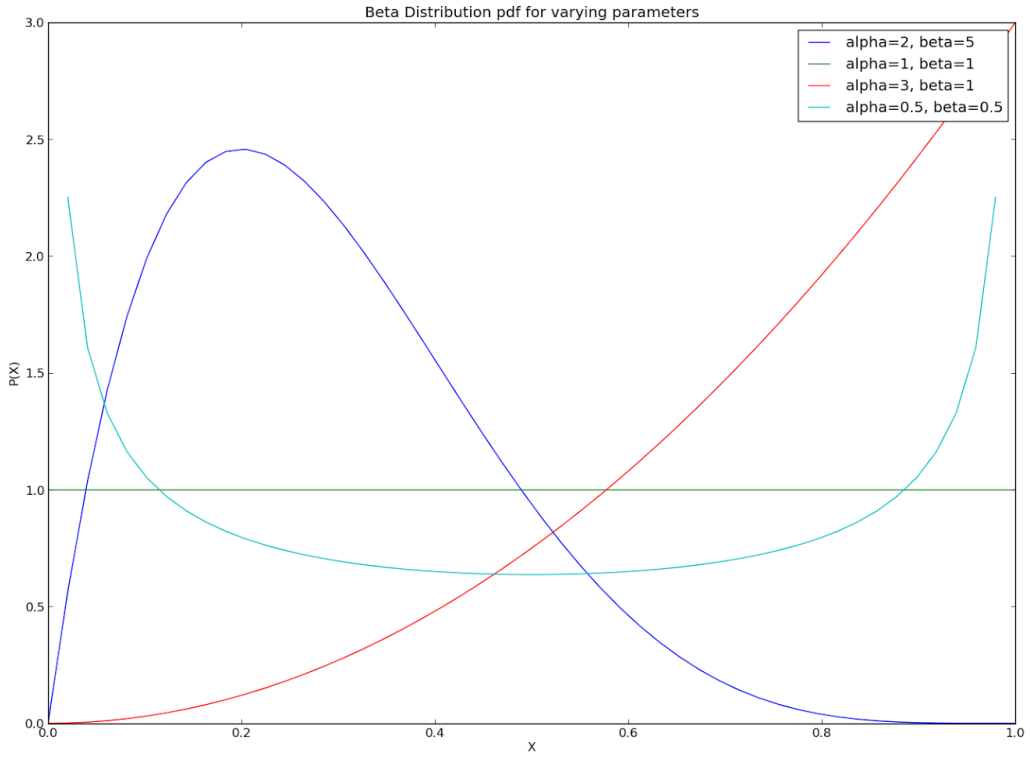


Figure 3.4 The above figure represents a beta distribution for different values of α and β. The initial prior of (1, 1) represents a uniform prior distribution while successive pairs of (α, β) map success and failures to their respective beta probability density function. As we increase the number of samples, this posterior probability density distribution starts to converge around the true state of the environment.

While the Beta distribution is a conjugate prior for the family of binomial distributions, the Dirichlet distribution is the conjugate prior for multinomial distributions. Imagine a scenario where we have been given a large data set. However, we are told that this data was not generated from a single distribution, but from a mixture of an unknown number of Gaussian distributions. We would therefore wish to cluster this data set in a proper, mathematically sound, and accurate manner. Figure 3.5 illustrates our situation pretty well.

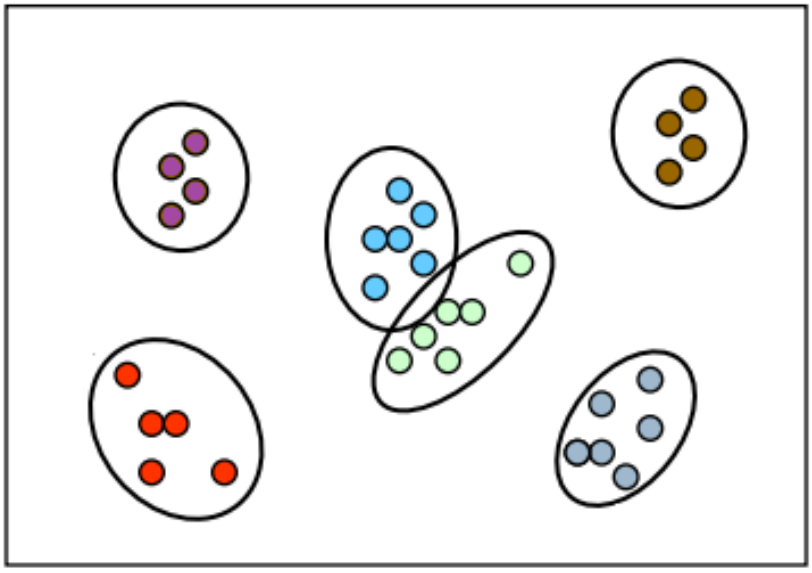


Figure 3.5 Example data set generated from a mixture of Gaussians

This distribution is parametrized by a K dimensional vector α (α_1 …. α_k) representing concentration parameters. These set of real numbers indicate how a particular distribution is spread out. If the value of a concentration parameter is large, it indicates a wider spread – which is to say that the distribution looks more like a uniform distribution. On the other hand, smaller values of α imply an extreme distribution with a lot of values having a probability of near zero.

Beta and Dirichlet distributions both allow us to soundly model how proportions vary each time we get new data. This learning-from-experience property to estimate priors, forms the basis of our Bayesian approach to modelling unknown transition probabilities.

3.2.2 The Bayes Adaptive MDP model

In this section, we introduce the BA-MDP model for our framework and detail an optimal decision making algorithm for planning in Markovian environments under parameter uncertainty. This section assumes that the state and actions are finite and known. The reward function, estimated using the inverse reinforcement learning algorithm, is also known.

In order to model the uncertainty over the transition $T^{sas'}$, we use *Dirichlet distributions* which we defined above. The BA-MDP model assumes that probabilities p_i of an event e_i follow a Dirichlet distribution [6]. Given $Ø_i$, the count for the number of occurrences of an event e_i over n trials, the probabilities p_i of an event e_i are drawn from a Dirichlet distribution, i.e.

$$(p_1, p_2 \ldots\ldots p_k) \sim \text{Dir}\ (Ø_1, Ø_2 \ldots\ldots Ø_k)$$

Consider the example of a bag of unfair coins. Each coin has a different probability distribution to model the probabilities of drawing heads and tails. The above distribution represents the probability that some discrete random variable behaves according to a distribution given by $(p_1, p_2 \ldots\ldots p_k)$, when the count vectors $(Ø_1, Ø_2 \ldots\ldots Ø_k)$ have been observed over the course of 'n' trials (where $n = \sum_{i=1}^{k} Ø_i$).

Thus, the expected value for the probability of an event 'i' becomes,

$$\mathbf{E}(p_i) = \frac{Ø_i}{\sum_{i=1}^{k} Ø_j} \tag{3.3}$$

The probability density function for the above Dirichlet is defined by:

$$f(p, Ø) = \frac{1}{B(Ø)} \prod_{i=1}^{k} p_i^{Ø-1} \tag{3.4}$$

The multinomial Beta function is represented by the variable 'B' in the above equation.

We now construct the BA-MDP model from the standard MDP framework. A finite state MDP is a tuple <S, A, T, R, γ>. We represent the uncertainty over the transition probability distribution $T^{sas'}$ using experience counts Ø. $Ø^{a}_{ss'} \forall s'$ represents the number of times an agent transitioned to a state s' from state 's' after performing an action 'a' [T(s' | s, a)]. Subsequently, the count vector Ø is defined as the vector of all possible transition counts. It can be written as follows:

$$Ø = < Ø^{a}_{ss'}, Ø^{a}_{ss'} \dots \dots Ø^{a}_{ss'} > \tag{3.5}$$

It is defined over all states 's', all actions 'a' and all subsequent states s'.

The expected value of a transition $T^{sas'}$, given the count vector Ø thus becomes:

$$T^{sas'}_{Ø} = \frac{Ø^{a}_{ss'}}{\sum_{s'' \epsilon S} Ø^{a}_{ss''}} \tag{3.6}$$

The key objective for the BA-MDP is to learn an optimal policy that maximizes the long term reward while taking into account parameter uncertainties over the unknown transitions. We model this using the BAPOMDP model described in [6]. The augmented state space S' of the Bayes Adaptive MDP thus becomes S' = S x T where,

$$T = \{Ø \epsilon \mathbb{N}^{|S|^2|A|} \mid \forall (s, a), \sum_{s' \epsilon S} Ø^{a}_{ss'} > 0\} \tag{3.7}$$

The augmented state T represents the space in which the count vector Ø lies. The action space of the BA-MDP is the same as that of a normal MDP.

The key to Bayes Adaptive MDP lies in the update function for the count vectors. The Transition Function T(s, a, s') must capture the evolution of the count vector Ø at each time step [6]. Consider a robot in some state 's' with some count vector 'Ø', that performs some action 'a' causing it to transition to a state s'. The count vector Ø' is updated as follows:

$$Ø' = Ø + \delta^{a}_{ss'} \tag{3.8}$$

$where, \delta^{a}_{ss'}$ is a vector full of zeros, except for a 1 for the transition count $Ø^{a}_{ss'}$.

These transition probabilities must be defined over all models and their corresponding probabilities as indicated by the current Dirichlet distribution [6], which is nothing but the expectations of these probabilities.

We can therefore define T', the Transition Probability over the augmented space as:

$$T'((s, ø), a, (s', ø')) = \begin{cases} T^{sas'}_{ø}, if\ Ø' = Ø + \delta^{a}_{ss'} \\ 0, \qquad\qquad Otherwise \end{cases} \tag{3.9}$$

After an action is taken, the uncertainty over the parameters is represented by the mixture of Dirichlet distributions i.e. the mixture of count vectors.

On the other hand, the reward function of the Bayes Adaptive MDP is simply defined as R(s, ø, a) = R(s, a). The count vectors augmented onto the state space has no tangible effect on the rewards. The discount factor γ, also remains the same. Hence, we can now define the BA-MDP model as a tuple of <S', A, T', R, γ,>, where the state and the transition functions are augmented with the count vectors to model parameter uncertainties.

3.2.3 Online Sample Based Planning

The key to accurately estimating the transition probabilities for a process lies in the way count vectors are generated and captured at each time step. We use an online d-step look ahead tree to update the counts. The count vector Ø is initialized with a prior distribution $Ø_0$. Each simulation consists of a fixed number of episodes. Episodes terminate when a particular condition is met or we reach the end of our look-ahead and we begin sampling from a random start state once again. The definition of an episode is domain specific and we will talk more about this in Chapter 4.

Our online sample based planning proceeds under the assumption that we have access to a simulator which models the real environment as closely as possible. Each simulation starts off with a randomly selected start state. At each stage we sample an action from our action space and subsequently sample the next state. It is important to note that at the end of each episode, we sample a new start state. However, the belief over the count vectors is still maintained. That way, an agent learns across all episodes.

The following diagram explains the planning algorithm described above.

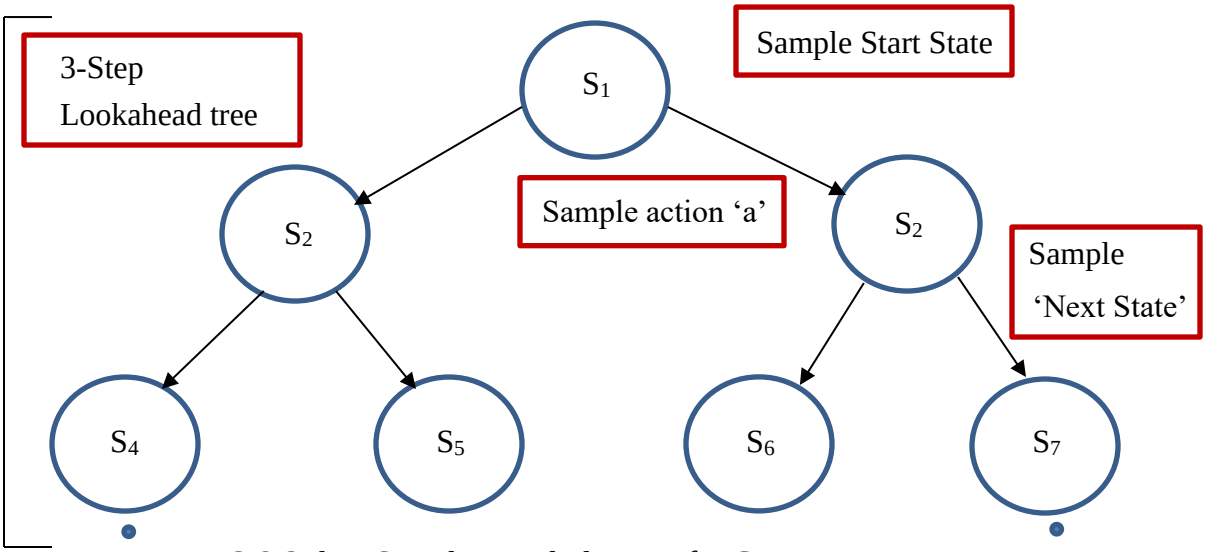


Figure 3.6 Online Sample Based Planning for Count Vector estimation

Figure 3.6 illustrates the sample based planning approach we use to get an estimate of the count vectors associated with the transitions of our MDP. We sample the start state each time an episode ends while maintaining the count vector. The simulation proceeds till some convergence criteria is met. At each stage, we use the equation Ø' = Ø + δ_{ss}^{a}, to update the count vectors based on the transition sampled.

Chapter 4 details our application domain and goes into detail about how episodes are defined for our problem. We compare model accuracy between the learnt probabilities and exact parameter uncertainty using the L_∞ -norm to estimate the maximum divergence from the true model.

3.3 BEST RESPONSE MODEL AS A BAYES ADAPTIVE DEC-MDP

Section 3.2 gave us a principled way to approximate unknown transition probabilities. This is a crucial step for our ad-hoc setting. It allows the 'learner' to associate accurate probabilities to transitions that it has never seen. Essentially, we now have a sound statistical technique to deal with 'future interactions' between an 'expert' and a 'learner'. We will now formulate a novel algorithm that we developed for multi-robot ad-hoc settings that allows an expert and a learner robot to efficiently team up in order to maximize some common goal(s).

3.3.1 Dec-MDP for Multi-Agent Coordination frameworks

In order to extend MDPs to our multi-agent settings, we examine a new class of Markovian processes known as Decentralized Markov Decision Process (Dec-MDP) [12]. This model allows agents to act in a completely decentralized manner while maximizing the performance of an entire team. Often, agents in a decentralized problem may not be able to observe the entire state of the environment. Dec-MDPs therefore define a vector of observation sets Ω for each agent Ag_i, and a

corresponding observation function *O* that gives the probability for each agent Ag_i to observe o_i given that the system is in a state 's'.

Formally, a Dec-MDP for 'n' agents can be defined as follows [12]:

- S is a finite state of 'n' states of the system
- $A = \langle A_1 \ldots\ldots A_n \rangle$ is a set joint actions where A_i is a set of k actions $\{a_1, a_1 \ldots. a_k\}$ that agent Ag_i can perform
- T(s, a, s') is the transition probability of reaching state s' by performing joint action 'a' in state 's' $\{T = S \times A \times S \rightarrow [0, 1]\}$
- $\Omega = \Omega_1 \times \Omega_2 \ldots\ldots \Omega_n$ is a finite set of observations corresponding to each agent Ag_i
- $O = S \times A \times S \times \Omega \rightarrow [0, 1]$ is the Observation Function that gives the probability that an agent Ag_i observes o_i given that the system is in a state 's'
- R(S, A) is the reward associated with taking joint action 'A_i' in state 'S'
- $\gamma \in [0, 1]$ is the discount factor

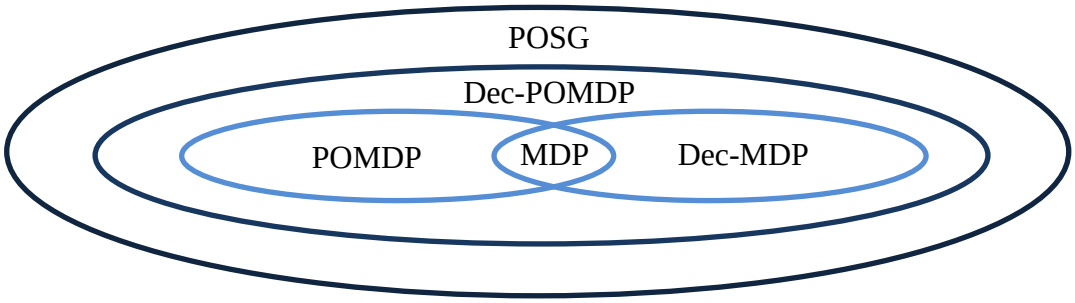


Figure 3.7 Types of Markovian Processes

Figure 3.7 illustrates the relationships between different types of Markovian processes. All of these come under the category of partially observable stochastic games. An MDP can be thought of as a single agent Dec-MDP problem. Additionally, a Dec-MDP is a special case of Dec-POMDP where the system is jointly fully observable. This property can be formally written as follows [12]:

$$\text{If } O(s, a, s', o = < o_1, o_2 \ldots\ldots o_n >) > 0, \text{ then } Pr(s' \mid < o_1, o_2 \ldots\ldots o_n >) = 1 \quad (3.10)$$

It is important to note that this property does not imply that all agents can independently observe the entire state space. However, for our problem domain, we look at a special case of Dec-MDPs where there is one agent that can fully observe the entire state space. Our inverse learning problem involves two agents (it can have more than two as well!); an ‘expert’ and a ‘learner’. Decentralized Markovian processes generally have joint full observability, while each agent has but a partial view of the environment. Agents must therefore base their decisions on an incomplete view of the environment. In our expert-learner system however, only the expert has a partial view of the environment. Its cognitive abilities do not allow multi-agent coordination and this means that the presence of any other agent/robot in the system is simply treated as ‘noise’. On the other hand, the ‘learner’ is not only aware of the ‘expert’, but we also assume complete local and global observability. Essentially, a ‘learning agent’ in our problem domain models the entire joint space of two robots and must make decisions based on the awareness and actions of the ‘expert’ robot agent.

We will now formally define our Ad-Hoc team setting where multiple agents must coordinate with each other to achieve a common goal without any prior knowledge on how to work together.

3.3.2 Best Response Model for Ad Hoc teaming of Robots

The Dec-MDP model is a well-defined framework for multi agent team decision problems. Solving a Dec-MDP consists of finding a joint policy for $\pi = < \pi_1, \pi_2 \ldots\ldots \pi_n >$ where π_i is the optimal policy for an agent Ag_i. $\Pi_{i:} S \times A_i$ maps the joint state of a system to an action a_i for an agent Ag_i.

Solving a Dec-MDP generally involves massive computations that often make large state systems intractable. Exact algorithms for optimally solving finite horizon Dec MDPs frequently consist of an incremental exhaustive generation of all possible policies and a subsequent elimination of dominated strategies. Dimensionality reduction therefore plays a very important role in deciding an algorithm to solve a Dec-MDP.

In many real world problems, noise free instantaneous communication is highly impractical. Moreover, robots may have different communication protocols which make it impossible to transfer information between two agents. In this section, we describe a novel approach to deal with a multi-agent scenario by defining a Best Response Model (BRM) from a single agent's perspective. We propose a principled approach that allows us to model a BRM as a Bayes Adaptive Decentralized Markov Decision Process (BA-DecMDP). We describe the structure of our BRM and go on to formulate an algorithm to optimally solve it.

Let us look back at our multi-agent teamwork problem once more. Our Ad Hoc team of agents consists of an 'expert' and a 'learner' robot. The expert robot has a fixed policy that does not take into account other agents in the environment. We can exploit this fact by significantly pruning the set of possible policies for our Dec MDP. The learner robot on the other hand has

complete awareness of itself and any other agent in the environment. Keeping this in mind, we can now define our best response model by expanding on this constraint.

Formally speaking, such a Dec-MDP can be folded into a Best Response Model (BRM) as follows:

State Space: $S' = S_E \, x \, S_L$ where,

S_E represents the state space of the Expert agent and,

S_L represents the state space of the Learner agent

We factorize our state space into Interaction and Non-Interaction spaces ($S = S_I + S_{NI}$). This factorization allows us to separate out interactions between the expert and the learner in a very principled way. Inverse Reinforcement Learning gives us the reward structure and the optimal policy followed by an expert agent. Additionally, solving the sample based planning algorithm detailed in Section 3.2 allows us to estimate the transition probabilities associated with these Interaction states. We therefore define the Transition Function for our Bayes Adaptive Dec-MDP Best Response Model as follows:

$$\Pr(S' \mid S, A) = \begin{cases} Pr_I\,(S'|S,A) & if,\ S \in S_I \\ Pr_{NI}\,(S'|S,A) & if,\ S \in S_{NI} \end{cases} \tag{3.10}$$

Pr-$_I$ is the Transition Function for the Interaction States

Pr-$_{NI}$ is the Transition Function for the Non-Interaction States

We separate out the states into Interaction and Non-Interaction spaces for an important reason. A coherent factorization also allows us to separate out the transition functions of interacting and non-interacting parts of our state space. Doing this allows us to make a very important claim regarding the transition functions of the two agents. We propose that for any Non-Interaction state, the transition probabilities associated with the Expert and the Learner are independent of each other. We believe that this is valid assumption, especially for Robotics domains where state space may be spread out over a large environment.

This means that we can define $Pr_{\text{-NI}}$ as follows.

$$Pr_{\text{-NI}}(S' \mid S, A) = Pr_{\text{-NI}}(S_E', S_L' \mid S_E, S_L, \Pi_E^*, a_L) \text{ and,} \tag{3.11}$$

$$Pr_{\text{-NI}}(S_E', S_L' \mid S_E, S_L, \Pi_E^*, a_L) = Pr(S_E' \mid S_E, \Pi_E^*) \; x \; \Pr(S_L' \mid S_L, a_L) \tag{3.12}$$

In the above definition, S_E and S_L refer to the States of the expert and learner respectively whereas a_L refers to the action performed by the learner. We have replaced $[a_E]$, the action performed by the expert, with Π_E^*, which is nothing but the optimal policy followed by the expert in state 's' that we learnt from inverse reinforcement learning.

On the other hand, the transition function for the Interaction states can be calculated using the Dirichlet count vectors that we get after performing online sample based planning discussed in Section 3.2. The expected value of a transition $T^{sas'}$, given the count vector Ø is calculated using the Equation (3.6):

$$T_{\emptyset}^{sas\prime} = \frac{\emptyset_{ss\prime}^{a}}{\sum_{s\prime\prime \epsilon S} \emptyset_{ss\prime\prime}^{a}}$$

Conversely, we define the Transition function for our Interaction states as follows:

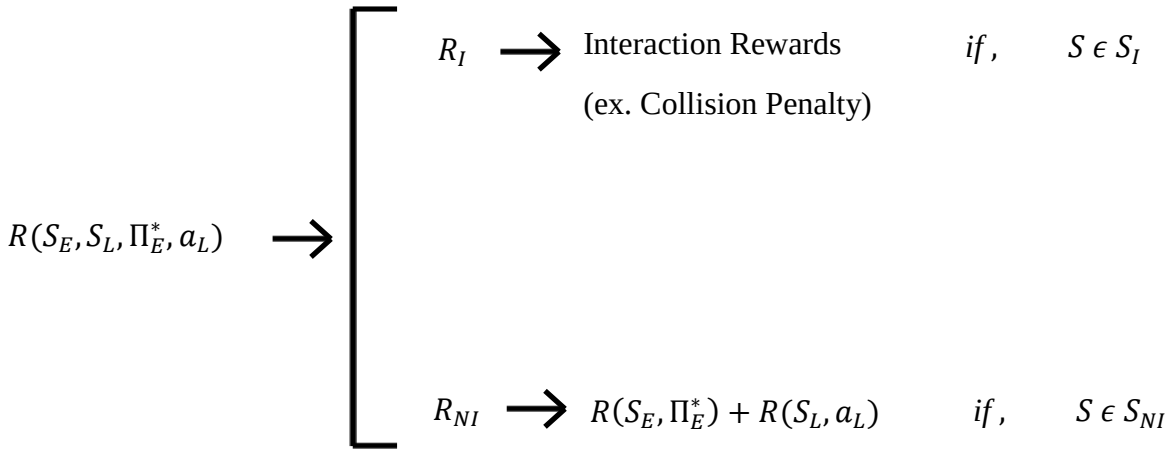
$$Pr_{\text{-I}}(S' \mid S, A) = Pr_{\text{-I}}(S_E', S_L' \mid S_E, S_L, \Pi_E^*, a_L) = \frac{\emptyset_{s_E s_L s_E' s_L'}^{\Pi_E^* a_L}}{\sum_{s_E'' \epsilon s_E} \sum_{s_L'' \epsilon s_L} \emptyset_{s_E s_L s_E'' s_L''}^{\Pi_E^* a_L}} \tag{3.13}$$

By factorizing the state space in a principled way, we have also been able to factorize the Transition functions of Interaction and Non-Interaction spaces. This is the most important step for developing a sound mathematical algorithm for solving an Ad Hoc Best Response Model as a Bayes Adaptive Dec-MDP.

On the other hand, it is crucial to remember that because we have factorized the state space into Interaction and Non-Interaction Spaces, we need to resolve the reward function as well, in order to reflect this separation. More importantly, while we can directly import the reward function

learnt from the expert into the reward function for Non-Interaction states (R_{NI}), it is imperative that we define a new reward structure for states that the expert has never encountered; namely the Interaction states (R_I). Since the BRM that we have is a multi-agent framework, the reward function used is defined over joint states and joint actions.

The following diagram illustrates the new reward structure that we will incorporate in our model.



One of the key attributes of IRL is that we learn a reward function rather than a policy. Rewards, by definition, are transferrable, which means that we can even use them when the environment has changed. It is important to note, however, while designing such a function, that reward consistency is critically important. While the above structure holds for a generic reward function in a BRM, we can always factorize the rewards further when dealing with complex Interaction states. We exploit this property for our application domain and detail the reward structure in Chapter 4.

Generally, the Action Space for the BRM remains the same as that of a normal Dec-MDP described above. However, it is important to note that for Robotics domains, we can always decompose the Action Space A into (A_I + A_{NI}). Consider the example of a robot dealing with interactions only while moving around in the environment. In such a situation, actions associated

with movement (ex. Go-Forward, Turn Right, Go-Target etc.) will be a part of the Interaction space while any other actions (ex. Pick up, Arm extend etc.) will be a part of the Non-Interaction space. It then becomes crucial to incorporate this new information into our Transition and Reward Function. The transition function in this case will be defined as:

$$Pr(\text{S' | S, A}) = \begin{cases} Pr_I(S'|S,A) & if,\ S \epsilon S_I\ and\ \ A \epsilon A_I \\ Pr_{NI}(S'|S,A) & if,\ S \epsilon S_{NI}\ and\ \ A \epsilon A_{NI} \end{cases} \quad (3.14)$$

Pr-$_I$ is the Transition Function for the Interaction States

Pr-$_{NI}$ is the Transition Function for the Non-Interaction States

The reward function reflects this change in exactly the same manner.

We will now define a dynamic programming algorithm to solve the BRM and generate an optimal policy for the learner in response to the optimal policy of the expert (generated by the learnt rewards from the Inverse Reinforcement Learning algorithm). The algorithm described below exponentially reduces the number of computations when compared to a standard Dec-MDP exact algorithm. This is because we plug in the learnt policy of the 'expert' into the algorithm and fix it. Thus, the algorithm does not need to iteratively generate an exhaustive list of possible candidate policies for each agent, which substantially increases the speed and time.

The Bellman equation is the crux of value iteration for optimally solving a standard MDP. The first step is to redefine the utility of a state and use the modified utility function in the Bellman equation for our best response model. Equation (3.15) describes the Bellman update for our algorithm.

$$V'(S_E, S_L) = \max_{a_L \epsilon A_L} \left[R\left(S_E, S_L, \Pi^*_{S_E}, a_l\right) + \gamma [\sum_{S'_E \epsilon S} \sum_{S'_L \epsilon S} \Pr(S_E, S_L, \Pi^*_{S_E}, a_L)\ x\ V(S'_E, S'_L)] \right] \quad (3.15)$$

The above equation defines the Bellman update required to solve the dynamic programming algorithm for our Bayes Adaptive Dec-MDP. We maximize over the actions of the learner while fixing the policy of the expert.

We can now use the standard value iteration algorithm with the above Bellman update to generate an optimal policy for the learner. The iteration step for the bellman update loops over the entire joint state of the expert and the learner. The optimal policy generated by value iteration will be the best response to the expert's optimal action in the joint state space.

3.4 SUMMARY

Chapter 3 highlighted the key contributions of our Thesis. We discussed the importance of developing a framework for expanding on the Inverse Reinforcement Learning algorithm in order to form an Ad Hoc team between an 'Expert' and a 'Learner'. Section 3.2 described a Bayesian approach to deal with uncertainties over the model parameters in a Markov Decision Process. We used this technique in Section 3.3 to develop a novel Best Response Model that allows us to optimally tackle multi-agent decision making from a single agent's perspective in a very principled and mathematically sound way.

CHAPTER 4

DEBRIS SORTING TASK USING IRL + AdHocInt

4.1 PROBLEM FORMULATION

One of the biggest impacts that robots will have on our lives will be in disaster management circumstances. Consider the recent tsunami in Japan. Deploying robots instead of human beings in such affected areas would not only be more cost effective in the future, it would undoubtedly be more advisable to do so in order to mitigate damage to life and property.

It is not hard then, to imagine a situation where multiple robots need to team up together in order to complete a particular task. Particularly in disaster prone areas, it is not far-fetched to imagine a case where robots might need to team up in highly stochastic ad-hoc environments without any communication guidelines to enable coordination between multiple agents. These classes of multi-agent team work problems have been the fundamental source of inspiration for this thesis research.

4.1.1 Problem Definition

We therefore propose a machine learning scenario where there is Robot I, which is already working on sorting out debris; for instance recyclable materials in a green box, non-recyclable in a blue and dangerous products in a red box. Robot J, which has more or less the same physical characteristics as Robot I arrives on the scene. However, Robot J has a much better cognitive aptitude and it has the ability to learn from observations just as human beings. We consider the problem of Robot J learning from observing the “expert” Robot I sorting out different debris into

boxes depending on their colors. After collecting sufficient observations and learning the reward function/optimal policy of the expert, Robot J must join the expert. But in doing so, Robot J must be able to comprehend the ad-hoc situation it has landed itself in. By this, we mean that Robot J must be able to plan out an optimal best response to the expert's actions, taking into account their joint states and joint actions which it needs to learn before joining the expert. We call our framework IRL + AdHocInt, which stands for Inverse Reinforcement Learning with Ad-Hoc Interactions. The goal of the "learner" robot is to maximize the cumulative team reward by correctly sorting out debris faster than the expert would do it by itself.

4.1.2 Formulating the Problem as an MDP

Markov Decision Processes are powerful tools that allow us to model decision-theoretic agents acting in highly stochastic environments. Moreover, many learning algorithms that deal with online planning and decision making, including Inverse Reinforcement Learning, require the problem to be formulated as an MDP. Let us now define the above debris sorting task for the expert and the learner as a tuple of $<S, A, T, R, \gamma>$.

MDP Formulation for the Debris sorting task – Single Agent

- STATE SPACE

  Before we define the state space of our robots, it is imperative that we discuss the robot model that we use in our experiments. We use the TurtleBot robots with attached phantomx_robot_arm in our experiments. These robots have the capabilities to localize, plan their motion trajectories, capture their surroundings using a Kinect camera, as well as the ability to pick and place objects.

  Keeping this in mind, we now define the state space of a single agent in our environment as a tuple of:

S: <X, Y, inHand, armPos, isEmpty>

X: It is the discretized X-Cord. Location of the robot on the map

Y: It is the discretized Y-Cord. Location of the robot on the map

InHand: It is the status of a Debris being held by the robot

ArmPos: It is the status of the robot's Arm Position

IsEmpty: It is the status of the Debris' to-be sorted

The domain for each of these state variables in the vector is as follows:

X: The domain of X is [1, 3]

Y: The domain of Y is [1, 6]

InHand: It can take the values NULL, Red, Green

ArmPos: It can take the values Extended or Closed

IsEmpty: It can take the values Empty or notEmpty

The dimensionality for a single agent thus becomes 3x6x3x2x2 = 216

- ACTION SPACE

The TurtleBots used in our application domain have physical capabilities to safely navigate in a stochastic environment. Additionally, our TurtleBots are equipped with a phantomx_arm that allows us to perform pick-n-place tasks using inverse kinematics and motion planning.

As such, we now define our action space as follows:

Goto_pickUp_Loc: Move to the pick-up table
Goto_target_1: Move to target_1 [drop-off zone for the Red debris]
Goto_target_2: Move to target_2 [drop-off zone for the Green debris]
Extend: Extend the Arm [move arm to pick-up position]
Close: Close the Arm [move arm to rest position]
PickUp: Pick up the Debris
Drop: Drop the Debris
NoOp: Do nothing!

- TRANSITION FUNCTION

The Transition Function for the agent was user modelled through empirical observations.

- REWARD FUNCTION

Reward functions are defined as a linear combination of binary feature functions.

For our application domain, we define $R(s, a)$ as a combination of the following:

RedInGreen: returns true if an agent drops a Red debris in a Red box
GreenInRed: returns true if agent drops a Green debris in a Green box
RedInGreen: returns true if an agent drops a Red debris in a Green box
GreenInRed: returns true if an agent drops a Green debris in a Red box
ActionPenalty: returns true whenever an action is performed

- DISCOUNT FACTOR

We have a standard discount factor of 0.99 for all our Markovian models

Now that we have formulated our application domain as an MDP, let us have a brief look at the Robotics framework that we use for our domain.

4.2 ROS AND THE phantomx_arm REPOSITORY

This project has been implemented on the Robotic Operating System (ROS) [14]. Originally developed in 2007, ROS is the most powerful open source framework for research and development in robotics. On a very basic level, ROS has two operational parts. The operating system side of ROS provides standard OS services like hardware abstraction, low level device control, package management etc. On the other hand, we have a collection of user contributed packages that implement various robot functionalities such as planning, navigation, inverse kinematics etc.

We used ROS Indigo on TurtleBot 2.0 to test our algorithm. We also attached a phantomx turtlebot arm to our robot for pick-n-place. The following sections briefly describe the phantomx_arm_turtlebot library that we developed for effective multi-robot navigation and pick-and-place functionality in gazebo and real world environments.

4.2.1 TurtleBot phantomx_arm in Gazebo

Gazebo is a 3D dynamic simulator that has a very powerful physics engine which allows us to simulate multiple robots in highly complex user defined environments. It is similar to game engines and offers a wide variety of robot models and sensors that work on top of a highly realistic physics simulator.

Gazebo ROS packages allow us to integrate ROS functionality onto a very powerful robot simulation engine. The Universal Robot Description Format (URDF) is an XML format used by ROS to describe all the elements of a robot model. We developed the URDF files for our phantomx_arm_turtlebot that models each and every link, sensor and joint on our robot. Figure 4.1 shows a sample section of our URDF file.

```
<link name="arm_base_link">
   <visual>
      <origin xyz="${M_SCALE*10} ${M_SCALE*10} ${M_SCALE*0}" rpy="${M_PI*0} ${M_PI*0} ${M_PI}" />
      <geometry>
         <mesh filename="package://phantomx_arm_description/meshes/visual/base_link.stl" scale="${M_SCALE} ${M_SCALE} ${M_SCALE}"/>
      </geometry>
      <xacro:material_black />
   </visual>
   <collision>
      <origin xyz="${M_SCALE*10} ${M_SCALE*10} ${M_SCALE*0}" rpy="${M_PI*0} ${M_PI*0} ${M_PI}" />
      <geometry>
         <mesh filename="package://phantomx_arm_description/meshes/collision/base_link.stl" scale="${M_SCALE} ${M_SCALE} ${M_SCALE}"
      </geometry>
   </collision>
   <inertial>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <mass value="0.005" />
      <inertia ixx="0.005" ixy="0.0" ixz="0.0" iyy="0.005" iyz="0.0" izz="0.005" />
   </inertial>
</link>
```

Figure 4.1 Sample section from the phantomx_arm_turtlebot URDF File

We also developed a highly user friendly code architecture so that we can have multiple namespace references for the same robot description format with minimal user effort.

4.2.2 Motion Planning and Inverse Kinematics

One of the fundamental requirements of a robot arm is to be able to easily move to any available point in its Free Space. We implemented the ROS MoveIt inverse kinematics algorithm to efficiently control our robotic arm's movements. Additionally, we integrated the IK algorithm onto the gazebo model that we developed for our robot.

The phantomx_arm_moveit repository has a well-structured pick and place library that allows a user to seamlessly integrate pick-n-place onto multiple robots with distinct namespace references (user specified). It allows the user to effectively compute inverse kinematic solutions and implement motion planning on the physical robot as well as on gazebo. The user simply needs to specify the target pose for the robot's end effector.

4.2.3 AMCL and move_base with Fine Tuning

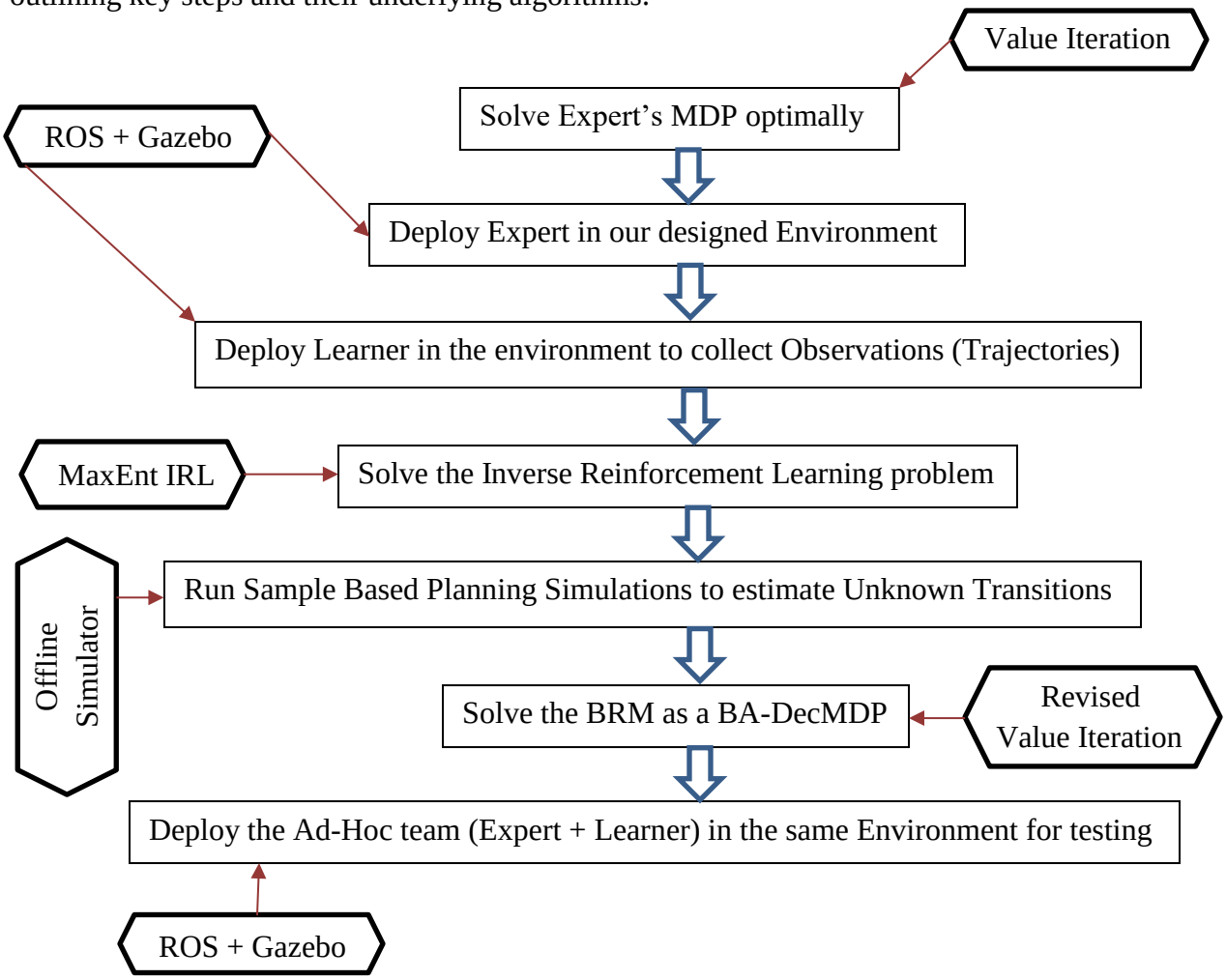
Localization and path planning algorithms are extremely sensitive to noise. Even in the gazebo simulator, onboard robot sensors are prone to the slightest disturbances in the environment. As such robot localization and path planning is very tricky. The most prominent ROS library for localization implements the Augmented Monte Carlo Localization or AMCL and its most widely used path planning library, move_base, works in collaboration with AMCL.

AMCL and move_base are some of the best implementations for effective path planning, yet they succumb to noise quite frequently. In an application domain like ours, where we need inch perfect localization and path planning, this becomes tricky. The margin of error for TurtleBot move_base planning in gazebo is, at best, a few inches off target and at worst, a couple of meters. This is a huge problem for our domain because even a few inches off base means that our inverse kinematic fails to find a solution. To counter this, we implemented a small fine tuning algorithm on top of move_base to tackle the problem. First, we wait for move_base to publish a successful ‘goal reached’ message. After this, we use our onboard camera and Kinect sensors to recalibrate the offset to actual target pose and manually override the wheel controllers to move the robot towards the target position. Perfect accuracy is not required - it should just be within an arm's length of the pick-up object. This method has proved very successful in our operations.

Apart from these main algorithms implemented in our repository, we have also created some smaller packages that allow users to model the gazebo environment as an MDP state space. The phantom_arm library allows effective ROS management of TurtleBots with robotic arms and we have an extensive repository that allows users to implement, modify and debug the ROS nodes executing various functionalities. This entire ROS library, including the implementation of all our machine learning algorithms can be found on the GitLab account of Thinc Lab, UGA.

4.3 PROCESS PIPELINE FOR A BRM BA-DecMDP

The following section describes the entire flow of a single experiment in our thesis by outlining key steps and their underlying algorithms.



Each trial consisted of the above steps and the results were collected over 100 different experiments. Section 4.4 details experimental outcomes and a subsequent analysis of our findings.

4.4 EXPERIMENTS AND ANALYSIS

This section details the different experiments we performed to test our IRL + AdHocInt framework. We performed 100 trials in order get a good estimate of our algorithm. The following subsections detail a step-by-step analysis of our experiments with relevant charts.

Our main contribution is the development of an effective Best Response Model for a two robot Ad-Hoc setting where a learner passively observes an expert in order to extract a reward function that best describes the demonstrated behavior and uses it to assist the expert in order to improve task performance. Consequently, the best way to test the efficiency of our algorithm is to compare task accuracy and the time taken to complete the task for a single agent versus a team.

4.3.1 Expert Trajectory Collection

We begin each trial by putting the learner in the same environment as the expert so that it can collect the demonstrated trajectories to perform Inverse Reinforcement Learning.

The expert follows an optimal policy for sorting out the debris in their respective boxes. Figure 4.2 shows a sample trajectory collected by a learner during the observation phase while Figure 4.3 illustrates the experimental setup in Gazebo. Please note the following state mappings used by us while implementing the code:

InHand: Red → 1; Green → 3; NULL → 0
ArmPos: Extended → 0; Closed → 1
IsEmpty: notEmpty → 0; isEmpty → 1

For the State Space *S*: [X, Y, InHand, ArmPos, IsEmpty]

```
[3, 3, 3, 0, 0] = extend
[3, 3, 3, 1, 0] = drop
[3, 3, 0, 1, 0] = close
[3, 3, 0, 0, 0] = goto_pickUp_Loc
[3, 4, 0, 0, 0] = goto_pickUp_Loc
[3, 5, 0, 0, 0] = extend
[3, 5, 0, 1, 0] = pickUp
[3, 5, 1, 1, 0] = close
[3, 5, 1, 0, 0] = goto_Target_1
[3, 4, 1, 0, 0] = goto_Target_1
[3, 3, 1, 0, 0] = goto_Target_1
[3, 2, 1, 0, 0] = goto_Target_1
[3, 1, 1, 0, 0] = extend
[3, 1, 1, 1, 0] = drop
[3, 1, 0, 1, 0] = close
[3, 1, 0, 0, 0] = goto_pickUp_Loc
[3, 2, 0, 0, 0] = goto_pickUp_Loc
[3, 3, 0, 0, 0] = goto_pickUp_Loc
[3, 4, 0, 0, 0] = goto_pickUp_Loc
[3, 5, 0, 0, 0] = extend
[3, 5, 0, 1, 0] = pickUp
[3, 5, 1, 1, 0] = close
[3, 5, 1, 0, 0] = goto_Target_1
```

```
[3, 4, 0, 0, 0] = goto_pickUp_Loc
[3, 5, 0, 0, 0] = extend
[3, 5, 0, 1, 0] = pickUp
[3, 5, 3, 1, 0] = close
[3, 5, 3, 0, 0] = goto_Target_3
[3, 4, 3, 0, 0] = goto_Target_3
[3, 3, 3, 0, 0] = extend
[3, 3, 3, 1, 0] = drop
[3, 3, 0, 1, 0] = close
[3, 3, 0, 0, 0] = goto_pickUp_Loc
[3, 4, 0, 0, 0] = goto_pickUp_Loc
[3, 5, 0, 0, 0] = extend
[3, 5, 0, 1, 0] = pickUp
```

Figure 4.2 Sample trajectories collected by the Learner

Figure 4.3 Learner passively observing the Expert

We plot the mean simulation time over the course of a 100 trials. The error bars show the variation in the time taken to sort out individual debris. In general, a single agent takes around 32 minutes in real time to complete the task.

Figure 4.4 Sim Time vs Debris sorted for the Expert

**Simulation time in Gazebo is calculated using an internal clock in the simulator. It takes into account processor speed and computer lag times among other things. It uses this information to calculate the real time factor, which is how "fast" the robots are moving inside the simulator compared to the time outside. For our single agent simulation, the real time factor was roughly 0.25, while the multi-robot simulation saw the real time factor average a mere 0.16.

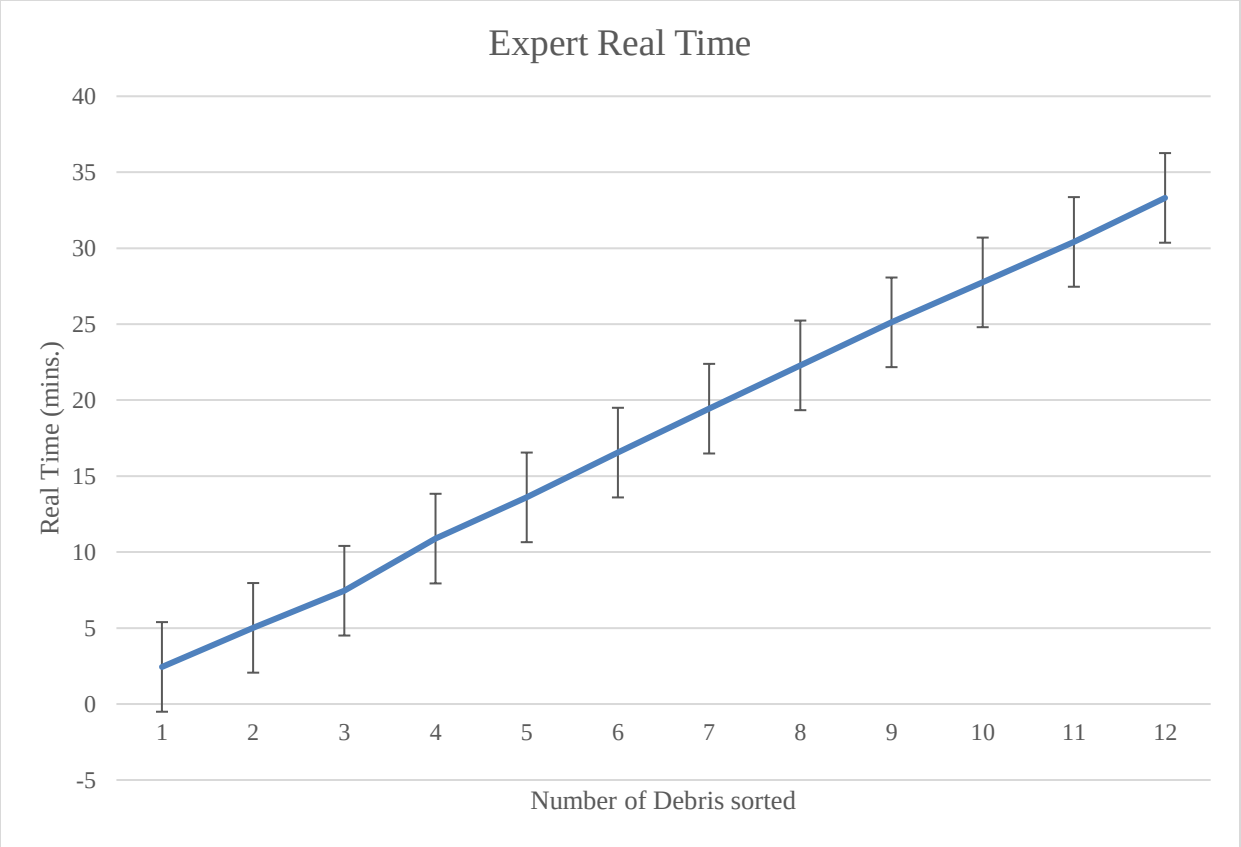


Figure 4.5 Real Time vs Debris sorted for the Expert

The Blue line in each of these charts shows the mean time for 100 trials of a single agent sorting out the Debris. Real Time for 12 debris is around 32 minutes which might seem pretty high but it is mainly because we have a large environment (see Fig. 4.3) and each time the expert picks up a debris from the table, it needs to do an entire new round of motion planning. Additionally, computing the inverse kinematics for accurately picking up and dropping the debris also takes up a lot of computational resources. A fascinating aspect of using the gazebo simulator is the realistic error bars associated with each debris. These relatively high fluctuations are seen because the ROS move_base algorithm computes a new motion plan each time we send it a goal. And in order to generate these plans, it needs to be properly localized in the environment. The fluctuations are observed when the robot fails to generate a new plan because it needs to localize itself again.

The more complex an environment, the more agents it has, the more likely move_base will fail to generate a good plan. This is a very realistic measure of how the robots (and our algorithm) will perform in the real world setting.

4.3.2 Solving the Inverse Reinforcement Learning problem

Once the learner has collected enough trajectories, the next step is to solve the IRL problem in order to recover the reward function that matches the behavior demonstrated by the expert. The learner starts solving the IRL algorithm once the expert has sorted out all the 12 Debris seen on the pick-up table.

In order to evaluate the performance of our Inverse Reinforcement Learning Algorithm, it might be tempting to directly compare reward function of the expert with that learnt by our subject agent (if available of course!) However, it is important to note that such comparisons are useless because the reward functions can be relatively equivalent and still generate the optimal policy used by the expert. As such, we use a different metric to compare the results of our IRL.

Choi and Kim [15] devised a neat method to compare the value functions of the policies obtained from optimally solving the MDP generated by using the learnt reward function, to that of the value function obtained from solving the expert's MDP. If R_I^L is the reward function learnt by a subject agent and π_I^* is the optimal policy followed by an expert (Agent 'I') and π_I^L is the policy of the learner then, we can define the Inverse Learning Error (ILE) as follows:

$$\text{ILE} = \| V^{\pi_I^*} - V^{\pi_I^L} \|$$

$V^{\pi_I^*}$ is the optimal value function of the expert I's MDP and $V^{\pi_I^L}$ is the value function obtained by following the learnt policy π_I^L on I's MDP.

An interesting property of ILE is that it monotonically increases as the policies of the expert and the learner diverge. On the other hand, when $\pi_I^* = \pi_I^L$ and the learner follows the exact same policy as that of the expert, ILE comes out to be 0.

We also used another metric to test the accuracy of our IRL algorithm. We compare the expert's true policy with that of the learnt policy to test for divergence. It is a simple equivalence test to see where the policies of the two agents differ.

Table 4.1 illustrates our results for the two metrics, averaged over 100 trials.

Average Policy Divergence	ILE
1.71296 %	1.549747691

Table 4.1 IRL Metrics

We tested out one more metric to see the extent of our IRL algorithm. The learner was, for all 100 trials, forced to collect trajectories until the expert sorted out all the debris. However, we wished to test the limits of our IRL model for our application domain. We carried out multiple experiments by reducing the size of the trajectories fed into our IRL algorithm. An interesting result that we discovered was that our IRL solver was able to get an ILE of ~2.25 even if the trajectory size is reduced to as low as 5 debris. The number of iterations until convergence is slightly more and the consistency and robustness is slightly less than that of a full trajectory and yet, with such a good result, we can surely spend much less time in collecting the trajectories. We have not yet fully explored this feature so we do not yet know the extent of this accuracy.

We believe that it might be because the number of features is very less and IRL does not need a larger trajectory for such a small feature space. We will however need to conduct further tests to confirm our hypothesis.

4.3.3 Online Sample based Planning for Transition Function estimation

The next part of our BA-DecMDP model is the transition function estimation. Chapter 3 describes the update function for the transition count vectors. One assumption that we make during this experiment is that the 'learner' has access to a simulator where it can execute this sample based planning step in order to get a good estimate for the transition function used by the expert.

The Best Response Model that we have developed factorizes the state space into interaction and non-interaction states ($S = S_I + S_{NI}$). Critically, we wish to estimate the transition functions only for the Interacting part of the state space. Our Bayes Adaptive approach, thus models the transition functions as a mixture of Dirichlet distributions (a mixture of count vectors) only over the Interacting part of the state space. As such, we need to be very clear about the definition of our Interaction states for the current model.

We consider any state where the 'learner' and the 'expert' can collide with each other as part of the interaction space. Our environment in gazebo is modelled such that when both, the expert and the learner at the pick-up table, then we say that there is a chance that they will collide; either by trying to pick-up the same debris, or by trying to extend/close the arm while the other agent is trying to do the opposite. This can lead to a collision resulting in damage to the arms or failure to complete the task.

Nevertheless, it is the transitions involved in these interaction states that the learner has never seen, it has never encountered. We therefore perform sample based online planning in order to get a good estimate of the transition probabilities. Please refer to Chapter 3 for a reminder about the mathematical formulation of the transition function using count vectors.

We define an episode over a 3-step look ahead tree or if a transition results in collision. Every time an episode ends, we randomly sample a new start state. Each time we sample a new transition, we update the count vector. This way, we learn across all the episodes. Each simulation is run for a 1000 episodes.

We test our learnt transition model by comparing it with the true transition probability. We begin our simulation by initializing the count vector by a uniform distribution. Sampling is done only over interaction states and actions that affect transitions in-those-states. Hence, we define our interaction actions as a subset of all the actions in our state space, which is to say that the only actions that can affect our interaction states are taken in consideration -- [extend, close, pickup, drop, noOp]. Please note that while 'drop' is a valid action at the pick-up table, we explicitly model the environment such that there is no interaction (which is to say no Collision) at the drop off locations.

We can now define a metric to measure the divergence of the learnt transitions from the true transition probabilities [Pr (Collision | S_I, A_I = 0.95)] as an $L_{infinity}$ - norm,

$$\forall\ s \in S_I,\ a \in A_I\text{: } L_{infinity}(\emptyset) = max_{-i} \mid T_{\emptyset}^{sas'} - T^{sas'}|_i$$

We use the L-infinity norm to find the maximum divergence of the learnt transitions at each stage of the simulation. Figure 4.6 shows the transition function update as a function of the mean $L_{infinity}$ -Norm averaged out over 5 trials. As the episodes increase, we see an evident update in the transition function that reflects the true nature of the system. Table 4.2 shows the evolution of the count vectors to model the underlying uncertainty of the environment.

	BA Transition Trials					
Episodes	MAX Norm_1	MAX Norm_2	MAX Norm_3	MAX Norm_4	MAX Norm_5	**Mean**
100	0.1107142	0.1464285	0.1017857	0.1017857	0.1017857	0.112499
200	0.0632075	0.0726415	0.0726415	0.0679245	0.0773584	0.070754
300	0.0493589	0.0525641	0.0461538	0.0429487	0.0397435	0.046153
400	0.0373786	0.0349514	0.0373786	0.0325242	0.0349514	0.035436
500	0.0320312	0.0300781	0.028125	0.0378906	0.0300781	0.031640
600	0.0251633	0.0316993	0.0202614	0.0284313	0.0235294	0.025816
700	0.0258426	0.0258426	0.0244382	0.0314606	0.0146067	0.024438
800	0.0201970	0.0300492	0.0140394	0.0238916	0.0362068	0.024876
900	0.0146929	0.0212719	0.0212719	0.0234649	0.0146929	0.019078
1000	0.0191699	0.0201581	0.0152173	0.0221343	0.0122529	0.017786

Table 4.2 Evolution of the maximally divergent transition with increasing Episode count

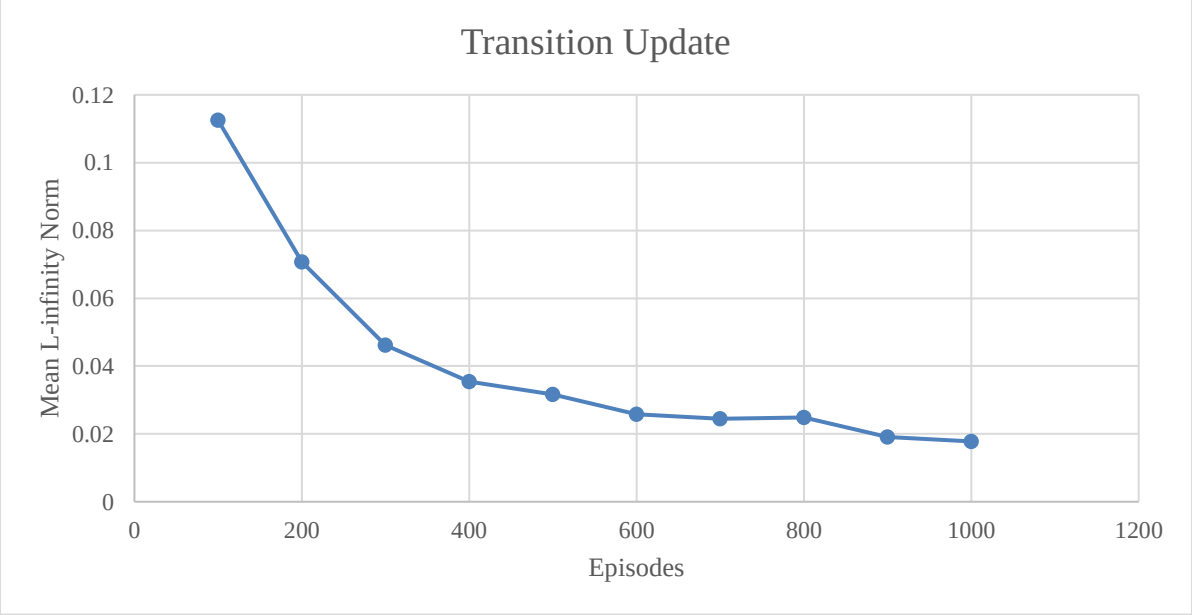


Figure 4.6 Count Vectors converging to the true state of the system

4.3.4 Testing the BA-DecMDP Best Response Model

The final step for testing the efficiency of our algorithm is to solve the BA-DecMDP by using the algorithm detailed in Chapter 3. We modify the reward function for the BRM by separating it out for the interaction and non-interaction states. In our domain, we specifically apply a very high negative reward to the action of performing pick-up actions in any interaction states. This is because we have defined interaction states as potential collision zones in our domain. We therefore wish to avoid them as much as possible. Once our dynamic program converges to an optimal joint policy, (which is nothing but the best response to the expert's learnt policy), then we deploy the learner and the expert into the same environment together.

A successful test run is defined by a fast task completion rate with a high accuracy in sorting out the debris. Figure 4.7 shows the accuracy of the Ad-Hoc Team. We use a histogram to demonstrate the number of debris correctly sorted. The histograms indicating extremely low count of debris sorted were a result of an extremely bad collision that ended with the two robot arms locked into a tight grip. Figure 4.8 and Figure 4.9 draw comparisons between the simulation times and the real times for a single robot versus a multi-robot setting.

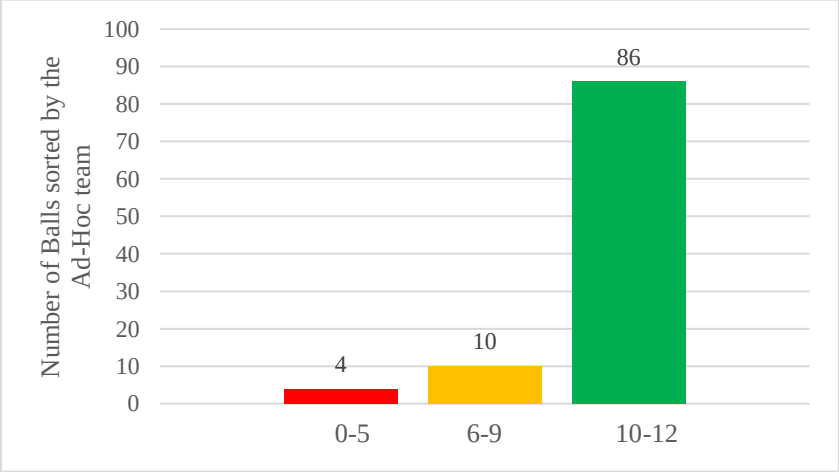


Figure 4.7 Accuracy as a function of number of debris correctly sorted in each Experiment

Figure 4.8 Sim Time vs Debris sorted for Single robot and Multi robot settings

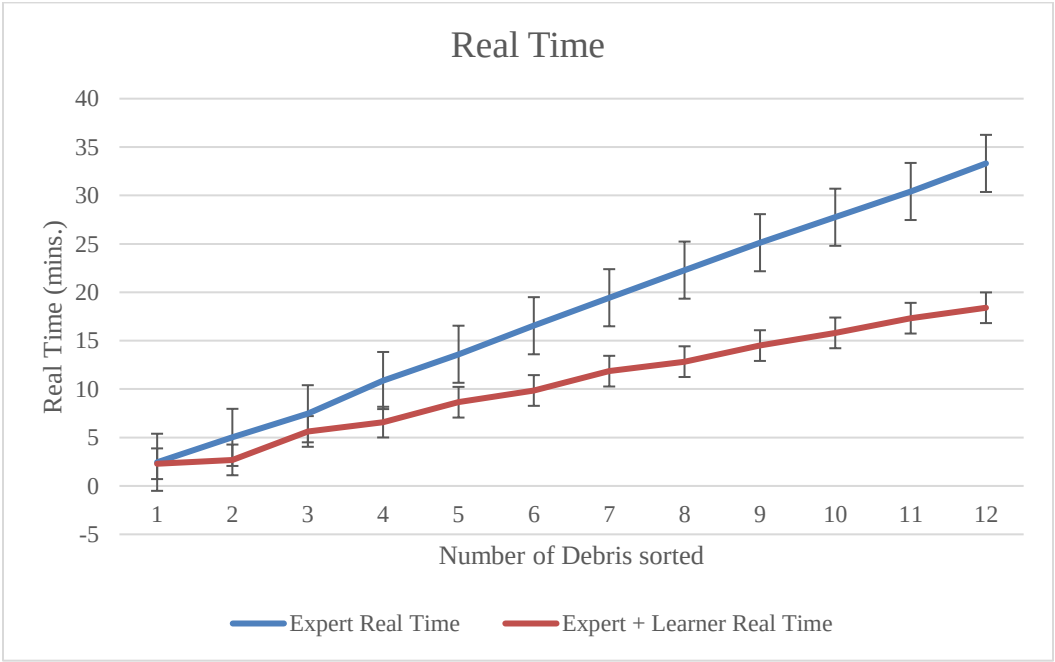


Figure 4.9 Real Time vs Debris sorted for Single robot and Multi robot settings

Figure 4.7 shows the accuracy for our best response model over a course of 100 trials. Out of 12 debris to be sorted, our multi–agent Ad-Hoc team correctly sorted more than 10 debris 86% of the times, while facing a completely disastrous run only 4% of the times.

Likewise, Figure 4.8 and Figure 4.9 show a statistical decrease in the amount of time it takes to sort out all the balls. Interestingly, the two robot team still takes less time even after taking into account the error bars associated with each sort. These results indicate a significant improvement over the performances of a single agent. The underlying algorithm for our framework has a strong statistical foundation. We hope to take this idea forward and extend it to more complex and uncertain environments.

CHAPTER 4

CONCLUSION AND FUTURE WORK

In this thesis, we established a novel framework for dealing with Ad-Hoc scenarios arising from a robot inverse learning problem. We provide a principled way to deal with uncertainties over the model parameters rising from unseen transitions. We then formulate a Best Response Model as a Bayes Adaptive Dec-MDP in order to compute an optimal policy that maximizes the overall performance of the team.

Now that we have got some really motivating results from our experiments, we need to think about the next step. The Best Response Model described in this thesis is expressed as a Bayes Adaptive Dec-MDP. While Dec-MDPs are extremely intractable in larger spaces, we mitigate this problem slightly by "fixing" the policy of the expert generated by our IRL algorithm. Be that as it may, our approach will face the same shortcomings as a classic Dec-MDP should the state space become extremely large. However, if we are to use this model in robotics domains, we might be able to come up with a decomposition technique that enables us to modify our BRM solution algorithm and speed up the computations significantly. One reason we believe that decomposition techniques might provide a way forward in robotics frameworks is because robots, by default have a control architecture that is decomposed into multiple sensor and actuator architectures. What we mean by this is that, activating a pick-up action will not affect the robot's wheel controllers or vice-versa. This high-level decomposition might be the answer to expanding our BRM algorithm to more robots; even if the state space is large.

Possible future works also include developing a model for partially observable MDPs. Extending our algorithm into partially observable settings offers an extremely real world challenge. There is a growing interest in research on inverse reinforcement learning and its real world implications. Research into using IRL for Ad-Hoc Robot Interactions is extremely sparse, if any. The experiments done in this thesis are, to the best of our knowledge, the first time anyone has tried to use inverse learning for Ad-Hoc teaming of robots. We hope that this paper instigates more interest in this budding field.

BIBLIOGRAPHY

[1] Abbeel, P. and Ng, A.Y., 2004, July. *Apprenticeship learning via inverse reinforcement learning*. In Proceedings of the twenty-first international conference on Machine learning (p. 1). ACM.

[2] Ng, A.Y. and Russell, S.J., 2000, June. *Algorithms for inverse reinforcement learning*. In ICML (pp. 663-670).

[3] Boularias, A., Krömer, O. and Peters, J., 2012, September. *Structured apprenticeship learning*. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases (pp. 227-242). Springer Berlin Heidelber

[4] Bogert, K. and Doshi, P., 2014, May. *Multi-robot inverse reinforcement learning under occlusion with interactions*. In Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems (pp. 173-180). International Foundation for Autonomous Agents and Multiagent Systems.

[5] Ziebart, B.D., Maas, A.L., Bagnell, J.A. and Dey, A.K., 2008, July. *Maximum Entropy Inverse Reinforcement Learning*. In AAAI (pp. 1433-1438).

[6] Ross, S., Chaib-draa, B. and Pineau, J., 2007. *Bayes-adaptive POMDPs*. In Advances in neural information processing systems (pp. 1225-1232).

[7] Duff, M.O.G., 2002. *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes* (Doctoral dissertation, University of Massachusetts Amherst).

[8] Russell, S.J., Norvig, P., Canny, J.F., Malik, J.M. and Edwards, D.D., 2003. *Artificial intelligence: a modern approach* (Vol. 2). Upper Saddle River: Prentice hall.

[9] Montague, P.R., Dayan, P., Person, C. and Sejnowski, T.J., 1995. *Bee foraging in uncertain environments using predictive hebbian learning*. Nature, 377(6551), pp.725-728.

[10] Schmajuk, N.A. and Zanutto, B.S., 1997. *Escape, avoidance, and imitation: A neural network approach*. Adaptive Behavior, 6(1), pp.63-129.

[11] George, O. and Frey, J., 2007. Is an 0.833 *Hitter Better than a 0.338 Hitter*. The American Statistician, 61 (2), pp 105-111.

[12] Sigaud, O. and Buffet, O. eds., 2013. *Markov decision processes in artificial intelligence*. John Wiley & Sons.

[13] Garage, W., 2011. *Turtlebot*. Website: http://turtlebot.com/last visited, pp.11-25.

[14] Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng AY. *ROS: an open-source Robot Operating System*. In ICRA workshop on open source software 2009 May 12 (Vol. 3, No. 3.2, p. 5).

[15] J Choi and Kee-eung Kim. *Inverse reinforcement learning in partially observable environments*. Machine Learning Research, 12:691–730, 2011.

[16] Amato, Christopher, and Frans A. Oliehoek. *Bayesian reinforcement learning for multiagent systems with state uncertainty.* Workshop on Multi-Agent Sequential Decision Making in Uncertain Domains. 2013.

[17] Ziebart, Brian D. *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. (2010).

[18] Melo, Francisco S., and Alberto Sardinha. "*Ad hoc teamwork by learning teammates' task.* Autonomous Agents and Multi-Agent Systems, 30.2 (2016): 175-219.

[19] Barrett, Samuel, and Peter Stone. *An analysis framework for ad hoc teamwork tasks*. Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[20] Barrett, Samuel, et al. *Learning teammate models for ad hoc teamwork. AAMAS* Adaptive Learning Agents (ALA) Workshop. 2012.

[21] Ramachandran, Deepak, and Eyal Amir. *Bayesian inverse reinforcement learning.* Urbana 51.61801 (2007): 1-4.

[22] Beal, Matthew J., Zoubin Ghahramani, and Carl E. Rasmussen. *The infinite hidden Markov model*. Advances in neural information processing systems. 2001.

[23] Becker, Raphen, et al. *Solving transition independent decentralized Markov decision processes*. Journal of Artificial Intelligence Research, 22 (2004): 423-455.

[24] Bogert, Kenneth, and Prashant Doshi. *Toward estimating others' transition models under occlusion for multi-robot irl.* 24th International Joint Conference on Artificial Intelligence (IJCAI). 2015.

[25] Herman, Michael, et al. *Inverse Reinforcement Learning with Simultaneous Estimation of Rewards and Dynamics.* arXiv preprint arXiv:1604.03912(2016).