Robot Planning for Achieving Multiple Independent Tasks with Discrepancies through Model Decomposition and Augmentation

Anahita Mohseni Kabir

CMU-RI-TR-21-66

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Robotics

The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

August 26, 2021

Thesis Committee:
Manuela Veloso, *co-chair*
Maxim Likhachev, *co-chair*
Henny Admoni
Sonia Chernova, *Georgia Institute of Technology*

To my family, especially my brother, Arman.

Abstract

This thesis focuses on robotics applications where a robot is required to accomplish a set of tasks that are partially observable and evolve independently of each other according to their dynamics. One such domain is decision-making for a robot waiter waiting tables at a restaurant. The robot waiter should take care of an ongoing stream of requests, namely serving a number of tables, including delivering food to the tables and checking on customers. An action that the robot should take next at any point of time depends on the duration of possible actions, the state of each table, and how these tables evolve over time, e.g., the food becomes cold after a few time steps. A conventional approach to deal with this problem is to combine all the tasks' states and robot actions into one large model and compute an optimal policy for this combined model. For the problems that we are interested in, the number of tasks, e.g., the number of tables in the restaurant domain, can be large making this planning approach computationally impractical and challenging.

This thesis introduces the class of problems that include multiple tasks that evolve independently of each other and presents algorithms to enable a robot to achieve the multiple tasks while expediting robot planning and execution. We develop a class of algorithms that take advantage of the structure in this class of problems, namely the independence between the tasks, to speed up planning and execution. Our key idea is to decompose the combined model of all tasks into a series of much smaller planning problems. We provide a theoretical and experimental analysis of the algorithms. We discuss how we formalize the restaurant domain and define the assumptions under which the restaurant domain can be an instance of this class of problems. We demonstrate the effectiveness of our solutions in a simulated restaurant setting and exemplify it on a real robot in a mock-up restaurant setting.

This thesis then focuses on the real-world applications of our algorithms on domains such as the restaurant setting where having an exact and comprehensive model that works for all the tables is infeasible. Given the nuances of a real-world restaurant setting, a robot might not have access to a complete and accurate model for each table, or the planning model might be an approximation of the complete exhaustive model for computational tractability reasons or due to early deployment of the robot. For example, going to the tables frequently to double-check if the customers have everything that they need might be rewarded by the model and might work for most of the customers; however, the customers on a certain table might be having a lunch business meeting in which people don't want to be interrupted frequently. This scenario might happen very rarely and might not be addressed in a particular restaurant model. Most planning or replanning algorithms are effective and efficient where the planning model is defined accurately, but they might fail to achieve the tasks in situations where the robot's planning model is an approximation of the true model. The remainder of this thesis focuses on formulating these unexpected situations where there is a discrepancy between the robot's observation and what is

expected to be observed given the robot's model as a robot planning problem. We tackle the discrepancies by augmenting the planning problem's state and action space with a set of hypotheses and questions regarding the discrepancies that aim at finding where the potential inaccuracies in the original planning model lie. Our formulation guarantees that the final goals of the tasks will be achieved eventually despite the existing discrepancy, *e.g.*, the robot will get the customers successfully out of the restaurant.

Solving the augmented planning problem with a larger action and state space introduces computational challenges. We provide planning algorithms that efficiently solve this much larger planning problem in both single-task and multi-task settings. We provide algorithms to solve the augmented planning problem more efficiently for single-task problems by leveraging the structure in the augmented model, namely the independent hypotheses. We then discuss how our approaches can be integrated with the efficient planning and execution algorithms that we developed for the multi-task settings. We provide comparisons on the performance of our approaches compared to the state-of-art approaches in both a single-task grid-world domain and a multi-task restaurant setting and show their effectiveness in various scenarios.

Acknowledgments

This thesis would not have been possible without the help of many. First and foremost, I would like to thank my advisors, Manuela Veloso and Maxim Likhachev, for their guidance over the years that I worked towards my PhD. I would like to thank Manuela for always believing in me, supporting me, and pushing me to think about the application of my work on the robots. I would like to thank Max for also supporting me and making sure that I think about the theoretical contributions of my work. It has been an honor to work with both of you.

I would also like to thank the members of my thesis committee: Henny Admoni and Sonia Chernova. I would like to thank Henny for meeting with me regularly and giving me advice regarding my work and how to better present it. Sonia and Charles Rich were my co-advisors for my master's degree, and I learned so much from both of them. I am beyond happy that I could have Sonia's guidance in my PhD as well. It was a pleasure for me to work with Reid Simmons shortly during my PhD. I learned a lot about POMDPs from Reid.

This work would not exist without the CoBot robots. I want to give special thanks to the past members of the CORAL group who made the CoBot robots a reality, including Mike Licitra, who physically built the robots; Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal, who laid the foundation for the complete navigation, task execution, and symbiotic autonomy of the robots; and, finally, everyone else who worked on and contributed to the robots. I would like to thank Sony Group Corporation for partially funding this thesis.

During the years I spent at CMU, the members of the CORAL and SBPL groups have been a constant source of inspiration. I learned a lot from many of my labmates, and some of them became my close friends. I would like to recognize the help and friendship of my CORAL colleagues: Vittorio Perera, Rui Silva, Devin Schwab, Kim Baraka, Philip Cooksey, Ashwin Khadke, Travers Rhodes, Arpit Agarwal and Kevin Zhang. I would like to also recognize the help and friendship of my SBPL colleagues: Ramkumar Natarajan and Muhammad Suhail Saleem.

Last but not least, I would like to thank my friends and family for their support. I would like to thank my lifelong friends Behnoush Golchifar and Sareh Yousefzadeh for all the good times we have spent together. I would like to thank my mom and dad for their support, love and energy. This dissertation would not have been possible without them. I would like to thank my brother and my sister-in-law for always being there for me when I needed them. My brother has been my biggest supporter and believer throughout all stages of my life. This thesis would not be possible without his constant inspiration and love.

Contents

When this dissertation is viewed as a PDF, the page header is a link to this Table of Contents.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Formulation

Many robotics applications, for instance a mobile robot performing multiple tasks such as delivering objects/messages to offices and escorting people to offices, an assistive robot executing multiple tasks such as cooking and vacuuming in a home setting, a robot in search and rescue helping out multiple human rescuers in their tasks, a robot helping out multiple factory workers by providing tools, and a robot waiting tables in a restaurant, involve a robot acting in a stochastic environment under partial observability while completing multiple tasks. In this thesis, we focus on formalizing these multi-task domains as domains with one robot and multiple independent partially observable tasks that evolve over time. We then develop efficient planning and execution algorithms that switch between the multiple tasks to achieve them all while effectively addressing the unexpected situations that arise due to having an imperfect model for some tasks.

One such multi-task domain that we focus on is a restaurant setting where a robot is waiting multiple tables. The robot waiter should take care of an ongoing stream of requests, including delivering food, taking food orders, and checking on customers. The tasks are partially observable as the robot cannot directly observe what people want and their degree of satisfaction. To service all tables, an action that the robot takes next at any point of time depends on the duration of possible actions, the state of each table, and how these tables evolve over time, *e.g.*, the food becomes cold after a few time steps or the people become dissatisfied after a few time steps. In this thesis, the restaurant setting has one robot and multiple tables which go through the dining process independently of each other. The customers at each table have multiple requests that require service from the robot at different points in time until they successfully leave the restaurant. The whole process that each table goes through is referred to as a task that should successfully be accomplished. We formalize the waiting tables for a restaurant as a planning problem with one

robot, and N independent models for the N tables in the restaurant (or N tasks). To enable the robot to perform the waiting tables task, we model each table, the state of the robot and the actions that can be applied to this particular table as a Partially Observable Markov Decision Process (POMDP). The restaurant has N POMDP models for the N tables in the restaurant.

We identify and formalize the class of problems with multiple independent tasks that are partially observable and evolve over time and provide the assumptions under which the restaurant domain can be an instance of such class of problems. A conventional approach to perform planning for these multi-task problems is to combine all the tasks' states and actions into one large model and compute the optimal policy for the combined global model. This combined model includes all possible combinations of the states of the tasks and a union over all their actions. For the problems that we are interested in, the number of tasks, *e.g.*, the number of tables in the restaurant, can be large making this planning approach computationally impractical and challenging. More specifically, solving the combined global model is infeasible in domains where the robot should be able to switch between multiple tasks in a real-time and scalable fashion. The combined model approach does not take advantage of the independence structure in such class of problems.

The first question that we address in this work is how can we leverage the independence structure to expedite task planning and execution for the class of problems with multiple independent tasks? We provide three algorithms to expedite task planning and execution for the class of problems with multiple tasks. One algorithm speeds up the task execution by solving the combined model less often, rather than after each action execution. We evaluate this approach on a mobile robot domain where planning infrequently is sufficient. We then focus on the restaurant setting where the robot is required to plan after each action execution. We provide two algorithms that leverage the independence between the tasks to expedite task planning. We evaluate these two algorithms on the restaurant setting.

The second question targets the aforementioned multi-task problems, but with a focus on the goal achievement aspect of real-world multi-task applications. In a real-world application such as the restaurant setting, there are differences in terms of the types of the restaurant, *e.g.*, fine dining, casual dining, cafes and diners, and the customers they target, *e.g.*, families with children, college students, and seniors. In these applications, coming up with a perfect and accurate model that works for everyone is very challenging. Even if a certain model works for most of the tables in a specific restaurant, it might not work for a particular table that might not belong to the target group that the restaurant model is designed for. Other reasons such as 1) learning the model from insufficient data, 2) making approximations to the true model for computational tractability, and 3) deploying the robot early on before having a comprehensive model can also be the cause

of the inaccuracies in the model. Without an exact planning model, the robot will encounter unexpected situations and should have a way to address these situations to guarantee achieving the goals of all tasks. For example, going to the tables frequently to double-check if the customers have everything that they need might be rewarded by the model and might work for most of the customers; however, the customers on a certain table might be having a lunch business meeting in which people don't want to be interrupted frequently. In another example, delivering bread to the customers before serving the food might be a part of the robot's process to keep the customers satisfied; however, a certain table might have allergies to wheat. These are a few examples of the unexpected situations that the robot might encounter and should be able to tackle. The second question that we address in this work is how can we guarantee that the robot will successfully achieve the goals of all tasks, *e.g.*, getting the customers successfully out of the restaurant, even if the model for a particular task differs from its true model? Specifically, given the multi-task context of the problem, we are interested in methods that address the unexpected situations for that particular inaccurate task while effectively responding to the other tasks.

The remainder of this thesis focuses on formulating these unexpected situations where there is a discrepancy between the robot's observation and what is expected to be observed given the robot's model as a robot planning problem. We build a new planning problem which includes both the original planning problem and the robot's hypotheses regarding where the potential inaccuracies in the original planning model lie. We provide planning algorithms that efficiently solve this much larger planning problem in both single-task and multi-task settings. We provide comparisons on the performance of our approaches compared to the state-of-art approaches in both a single-task grid-world domain and a multi-task restaurant setting and show their effectiveness in various scenarios.

1.2 Planning and Execution for Multiple Independent Tasks

We address the challenges associated with the computational complexity of solving the combined model. We provide algorithms that speed up task planning and execution for domains with multiple tasks.

1.2.1 Expediting Task Execution

The first algorithm for efficient planning and execution looks into speeding up the *task execution* by deciding how often the robot should perform planning. To plan in presence of multiple tasks, a robot has to build a large model with all the tasks' states and actions and at each time step decide

what action should be executed on which task. In a lot of robotics applications, both processing all the sensory input variables for all the tasks and solving the large model are impractical. We provide an algorithm that speeds ups the task execution by deciding how often the robot should perform planning. The robot only focuses on one task at a time and switches to another task when it is triggered. We call the signals that trigger the task-switching, "stimuli". When a stimulus is triggered, the robot builds the large model with all the tasks to decide if a switch is more rewarding than continuing with the current task. We provide an approach to identify the "stimuli" that trigger the task-switching. This approach is applied on a mobile robot domain where the robot is executing a user-requested task, *e.g.*, object delivery, and is also alert for multiple external tasks that might come up. For example, if the robot is scheduled to deliver an object to an office, and on the way to the office it sees a person asking for help, we might want the robot to first assist the person and then finish its delivery. We exploit the assumption that external tasks such as a human-interaction task or a trash cleaning task are optional and happen infrequently. We show that our solution using the switching stimuli compares favorably to the naive approach of building the large combined model. Moreover, leveraging the stimuli significantly decreases the amount of sensory input processing during the execution of the tasks. We explain this algorithm and its results in Chapter 4. For these type of problems, many works have proposed approaches to address multiple task models by using hierarchical planning [10, 15, 103, 105], goal management approaches [89, 128, 187], and behavior-based control systems [117, 118, 146, 151]. We introduce an approach that learns a switching policy, identifies the stimuli, and then uses both the policy and stimuli to switch between multiple task models. Different from the above approaches, our approach learns when to interrupt the task execution to decrease the amount of sensory computations.

This approach drastically improves the task execution compared to solving the combined model at each time step and is effective in domains where responding reactively to the environment is sufficient, but it does not extend to domains with more complex structure such as the restaurant domain. For a domain such as the restaurant domain, the robot has to switch between the tasks frequently and cannot stick to one task and only switch to another task when triggered. For these more complex problems, we focus on providing algorithms that solve the large combined model faster rather than solving it less often.

1.2.2 Expediting Task Planning

The next two algorithms focus on expediting *task planning* for the class of problems with multiple independent tasks. The algorithms target domains where the robot is required to accomplish multiple tasks and address the computational infeasibility of solving large models. The tasks are represented as Partially Observable Markov Decision Processes (POMDPs). We propose

algorithms that expedite planning for multiple partially observable tasks by leveraging the structure of the problem, namely the independence between the tasks. More specifically, we target the class of problems with multiple independent tasks that evolve over time and present algorithms to solve this class of problems by decomposing the problem into a series of much smaller planning problems, the solution to which gives us an optimal solution. In particular, a robot attending a single task can be represented as a standalone smaller planning problem. Using these insights, we develop an algorithm that searches over possible subsets of N tasks, solving each optimally until a provably globally optimal solution is found. Our methods exploit 1) the structure that the tasks are independent and 2) an observation that in many domains the number of tasks that the robot can accomplish within a horizon–determines how far into the future the robot will look to select the optimal action–is very limited. We introduce an approach that optimally and efficiently plans for a short fixed planning horizon.

We then recognize that selecting the right planning horizon can be challenging since an overly short horizon may result in a low-quality solution while supporting a longer horizon quickly becomes computationally impractical. Specifically, in POMDP planning, the complexity of planning grows exponentially with the horizon so a long horizon may easily preclude online planning. We extend the previous algorithm to provide efficient planning over long fixed-length horizons without discounting and infinite-length horizons with discounting. We enable the robot to efficiently plan for long horizons by terminating the search early before reaching the full planning horizon while guaranteeing optimality. Similar to the previous approach, in this algorithm, we leverage the independent tasks structure to decompose the problem into smaller pieces, but we use it in a different fashion than the previous algorithm to provide efficient planning for long and discounted infinite-horizon problems. Our key idea is to compute lower and upper-bounds on the value of an optimal solution for variable horizons. We combine the solutions to multiple individual tasks to compute the bounds rather than solving larger models built from subsets of tasks as done in the previous algorithm.

We test the two approaches on a simplified restaurant environment in simulation and show their effectiveness compared to the state-of-the-art approaches. We provide the theoretical analysis and the assumptions under which the approaches are optimal. We explain these algorithms in Chapters 5 and 6. Many works have proposed approaches to speed up POMDP solvers by using point-based methods [169], hierarchical planning [181], clustering and compression of belief space [160, 175], factored representation [168], and online POMDP approaches [158]. We are interested in domains in which a robot has to attend to multiple independent tasks whereas the above approaches do not make the independence assumption and address the combined model directly.

1.3 Efficient Robot Planning and Execution in Presence of Discrepancies

In dynamic and changing environments with semantically rich tasks and human interactions such as the restaurant domain, unexpected situations that are not predicted by the robot's model might arise. These unexpected situations might prevent the robot from proceeding with a particular task (or table). We enable the robot to address the unexpected situations to effectively attend to the particular table and the other tables. We detect these unexpected situations by checking if the robot's current observation belongs to the set of possible observations expected by the robot's model. We refer to these unexpected situations as discrepancies between the robot's observations and its model. The objective of this work is to enable the robot to still achieve the tasks at hand, *e.g.*, getting the restaurant customers successfully out of the restaurant, in presence of the discrepancies.

We address the discrepancies by building a new augmented model from the discrepancy and the original planning model. In the augmented planning problem, the robot has a set of hypotheses regarding the discrepancy and can ask questions from an oracle to find the hypothesis that best explains the discrepancy. This formulation aims at guaranteeing that the robot will be able to eventually achieve the final goals of all the tasks, *e.g.*, the robot will get the customers successfully out of the restaurant. Some works have been proposed to address the inaccurate models by learning and refining the model's parameters [90, 119, 156], state estimation and replanning approaches [111, 192], planning using model uncertainty [47, 111, 132, 199], and using human input to learn or solve the POMDPs faster [8, 41, 61], or as state information providers [153]. Different from the state estimation and replanning approaches, our formulation addresses the discrepancies even when the discrepancy is due to a fundamental change in the environment and thus repeats itself. Different from the learning approaches, we focus on the multi-task settings where the robot should attend to all the tasks rather than exploring the environment to learn the true planning model for a particular task.

The augmented planning model which includes the hypotheses and questions can become very large and hence challenging to solve. To address its computational complexity challenges, we provide an approach to more efficiently solve the augmented planning problem by leveraging the independent hypotheses. We then integrate this approach with the efficient planning algorithms that we proposed previously for the multi-task settings to expedite task planning and execution for the combined model built from the augmented model and the other original models. We discuss these algorithms in Chapter 7.

1.4 Contributions

The contributions of our work are to:

1. identify and formalize the class of problems with multiple independent tasks that evolve over time.
2. mathematically formalize robot waiting in a restaurant and provide the assumptions under which our algorithms can be applied to it.
3. develop efficient algorithms to expedite task execution in multi-task problems.
4. develop scalable and efficient planning algorithms for solving the class of problems with multiple independent tasks.
5. develop a novel algorithmic framework for robot planning and execution to address the discrepancy between the robot's observations and its model.
6. develop efficient algorithms to expedite task planning in presence of discrepancies in single-task and multi-task problems.
7. implement and test our algorithms on a simulated restaurant setting with large number of tables and exemplify it on a real robot in a mock-up restaurant setting with a few tables.

1.5 Thesis Outline

The following outline summarizes each chapter of the thesis.

Chapter 2 We review the background on two formalisms for sequential decision making, namely Markov Decision Processes (MDPs) and Partially Observable Markov Decision Processes (POMDPs).

Chapter 3 We formalize the waiting tables for a restaurant as a planning problem with one robot, and N independent models for the N tables in the restaurant. We discuss how each table is modeled as a POMDP and under what assumptions the POMDP models associated with the tables are independent of one another.

Chapter 4 We present an algorithm that expedites task execution in problems where the robot should accomplish a user-requested task and is also alert for multiple external tasks that might come up. Even though some of the tasks are optional, the robot should plan over a large combined

model associated with all the tasks. The algorithm speeds up the task execution by only focusing on one task at a time and only performing planning over the large combined model when necessary.

Chapter 5 We identify and formalize the class of problems with multiple independent tasks that are partially observable and evolve over time such as the restaurant domain. We introduce an approach that optimally and efficiently plans for a short fixed planning horizon. We expedite task planning for the aforementioned class of problems by leveraging the independence structure between the tasks and the observation that the number of tasks that the robot can address within a short horizon is limited. Our key idea is to decompose the problem into a series of much smaller planning problems.

Chapter 6 We extend the previous approach to long fixed-length horizons without discounting and infinite-length horizons with discounting problems. In addition to decomposing the problem into smaller problems, to enable planning for longer horizons, we compute lower and upper-bounds on the value of an optimal solution for variable horizons.

Chapter 7 We discuss how we formulate the discrepancy between the robot's observation and its model as a planning problem over an augmented model. We then leverage the structure in the augmented model, namely the independent hypotheses, to solve it more efficiently. Finally, we discuss how the previous approaches of efficient planning for multi-task settings can be integrated with the approach of planning for the augmented model.

Chapter 8 We review the relevant literature.

Chapter 9 We conclude the thesis with a summary of its contributions and present potential directions for future work.

Chapter 2

Background

In this section, we provide the MDP and POMDP formalisms. We will use the MDP formalism in Chapter 4 and the POMDP formalism in Chapters 3, 5, 6 and 7.

2.1 Markov Decision Processes (MDPs)

The most common representations for sequential decision models in decision-theoretic planning under uncertainty are Markov decision processes (MDPs). MDPs are represented by a tuple $M = (S, A, R, T, \gamma)$, where S is a set of states, A is a set of actions, $R(s, a)$ is the reward the agent receives when executing action a in state s, $T(s, a, s')$ is the probability of the agent finding itself in state s' having executed action a in state s, and $\gamma \in (0, 1]$ is a discount factor which specifies how much immediate reward is favored over more distant reward. The agent objective is to choose actions at each time step to maximize its expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward gained at time t. Actions are taken at fixed decision epochs in the continuous time case or time steps in the discrete case which will be the focus on this thesis. The solution to an MDP is a policy $\pi : S \rightarrow A$ which is a mapping from states to actions for every state in the model at which an agent may act.

This model takes advantage of the Markov property as a way of limiting the complexity of the planning problem. The Markov property says that the environment's response, given by the state transition and reward functions at time $t + 1$ depends only on the state and actions at time t and the rest of the history can be ignored. The Markov property is defined as:

$$\Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, ..., r_1, s_0, a_0) = \Pr(s_{t+1} = s', r_{t+1} = r | s_t, a_t) \tag{2.1}$$

Given a policy π and a state s, the return that the agent expects to obtain by following π from s can be defined by the following a recursive function, known as the *value function (or V function)*. The value function (denoted by V) represents what the expected overall value of a state s under the policy π is, *i.e.*, how good it is to be in any given state. The Q function (action-value function) states what the value of a state s and an action a is under policy π, *i.e.*, how good it is to take a certain action given a certain state. The V function and the Q function (updated by Q-learning method) are defined below.

$$V^{\pi}(s) = E_{\pi}\{R_t|s_t = s\} = E_{\pi}\{\sum_{\tau=0}^{\infty} \gamma^{t+\tau+1} r_{\tau}|s_t = s\} \tag{2.2}$$

$$\begin{aligned} Q^{\pi}(s,a) &= E_{\pi}\{R_t|s_t = s, a_t = a\} \\ &= E_{\pi}\{\sum_{\tau=0}^{\infty} \gamma^{t+\tau+1} r_{\tau}|s_t = s, a_t = a\} \end{aligned} \tag{2.3}$$

The value for each state for the planning horizon t can be computed from the state values for the $t-1$ horizon, along with the reward and state transition function for the problem by following the Bellman's Equation [17, 148]:

$$Q_t^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) V_{t-1}^{\pi}(s') \tag{2.4}$$

If the transition function is known, the agent can use one of the model-based Reinforcement Learning (RL) approaches such as value-iteration [178] to compute the exact solution for the MDP. However, in most RL applications the dynamics of the environment is not known, *i.e.*, the agent does not know how the world will change in response of its actions, nor what reward it will receive for executing an action. The class of RL algorithms that learn the policy without using a model of the environment and only by trial-and-error are called model-free algorithms, *e.g.*, Q-learning method. When the state space is large, e.g. when the state space is continuous, it is not feasible to keep a separate value for each state in the memory. In such cases, function approximation methods such as neural networks are used to solve the MDP.

Our work in Chapter 4 leverages the MDP representation. The MDP's state space is continuous, therefore we use a neural network to approximate the Q function. Reinforcement learning is known to be unstable when a nonlinear function approximator such as neural networks is used to represent the Q function [125, 184]. In [125], the authors provide a variant of Q-learning called Deep Q-Network (DQN) that stabilizes reinforcement learning. The deep Q-network method stabilizes the learning by utilizing an experience replay mechanism and a second fixed target

network. In addition to utilizing these two improvements of the DQN approach, we leverage the dueling architecture [195] which compared to DQN provides a more robust estimate of the Q function and have shown better performance in domains with many similar-valued actions. Thus, we leverage the dueling architecture in Chapter 4.

2.2 Partially Observable Markov Decision Processes (POMDPs)

2.2.1 Discounted Reward POMDPs

The partially observable MDP model (POMDP) generalizes the MDP model to allow for even more forms of uncertainty to be accounted for in the process. In POMDPs the true state of the system is not directly observable by the decision-maker. Instead, the decision-maker receives observations from the environment. Formally, a POMDP is a tuple $(S, A, Z, T, O, R, \gamma)$, where S denotes the agent's state space, A denotes the agent's action space, and Z denotes the agent's observation space. At each time step, the agent takes an action $a \in A$ and transitions from a state $s \in S$ to $s' \in S$ with probability $T(s, a, s') = p(s'|s, a)$, makes an observation $z \in Z$, and receives a reward equal to $R(s, a)$. The conditional probability function $O(s', a, z) = p(z|s', a)$ models noisy sensor observations. The discount factor γ specifies how much immediate reward is favored over more distant reward. The agent objective is to choose actions at each time step to maximize its expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward gained at time t.

Planning in a POMDP can be thought of as planning over the continuous belief space of the underlying MDP. A belief state is the probability distribution over which state the agent is actually in when it receives an observation. The Markov property holds for POMDPs in the sense that the belief state is a sufficient statistic of the history of the system. The belief over the states can be computed from an existing belief distribution over states and a new action a and observation z as follows.

$$\begin{aligned} b(s') &= \frac{O(z|s', a) \sum_{s \in S} T(s'|s, a) b(s)}{\Pr(z|b, a)} \\ \Pr(z|b, a) &= \sum_{s' \in S} O(z|s', a) \sum_{s \in S} T(s'|s, a) b(s) \end{aligned} \tag{2.5}$$

The optimal return at stage t, $V_t^*(b)$, can be iteratively computed by Eq. 2.6.

$$
\begin{aligned}
Q_t^*(b,a) &= \sum_{s\in S} b(s)R(s,a) + \gamma \sum_{z\in Z} \Pr(z|b,a)V_{t-1}^*(b_z^a) \\
V_t^*(b) &= \max_{a\in A}\left[Q_t^*(b,a)\right]
\end{aligned}
\tag{2.6}
$$

Offline algorithms to solve POMDP problems have been proposed in [80, 144]. These algorithms specify, prior to the execution, the best action to execute for all possible situations. While these approximate algorithms can achieve very good performance, they often take significant time to solve large problems, where there are too many possible situations to enumerate. On the other hand, online approaches [158] plan only for the current information state and a small horizon of contingency plans. One drawback of online planning is that it generally needs to meet real-time constraints, thus greatly reducing the available planning time, compared to offline approaches. There are works on unifying approximate offline and online solving approaches to address large POMDPs by using offline learning as a heuristic function to guide the online search algorithm [157]. These approaches preserve the theoretical properties of the offline planning and exploits the scalability of the online planning. We use the online POMDP planning where the robot interleaves planning and execution in our work (Chapters 5, 6, and 7).

2.2.2 Goal POMDPs

Formally, a Goal POMDP (or shortest path POMDP) is a tuple given by (S, G, A, Z, T, O, C), where S denotes the agent's state space, $G \subseteq S$ denotes a non-empty set of goal states, A denotes the agent's action space, and Z denotes the agent's observation space. At each time step, the agent takes an action $a \in A$ and transitions from a state $s \in S$ to $s' \in S$ with probability $T(s,a,s') = p(s'|s,a)$, makes an observation $z \in Z$, and receives a positive cost equal to $C(s,a)$. The conditional probability function $O(s',a,z) = p(z|s',a)$ models noisy sensor observations. The goal states $g \in G$ are cost-free $C(g,a) = 0$, absorbing $T(g,a,g) = 1$, and fully observable $g \in Z$, so that $O(s',a,g) = 1$ if $s' = t$ and $O(s',a,g) = 0$ otherwise. The agent objective is to choose actions at each time step to minimize the expected cost to a goal as below.

$$
\begin{aligned}
Q_t^*(b,a) &= \sum_{s\in S} b(s)C(s,a) + \gamma \sum_{z\in Z} \Pr(z|b,a)V_{t-1}^*(b_z^a) \\
V_t^*(b) &= \min_{a\in A}\left[Q_t^*(b,a)\right]
\end{aligned}
\tag{2.7}
$$

Goal POMDPs have a similar formulation as discounted reward POMDPs. Differently, goal POMDPs have a set of goal states (that are absorbing and cost-free), and instead of the reward

function, the robot has access to a cost function (all positive costs). The agent minimizes the cost to reach a goal. It has been proven that any discounted reward POMDPs can be converted to a goal POMDP [26].

Similar to the shortest path MDP assumption [18, 55], when we talk about a policy or a solution to the POMDP p, we refer to a *proper* policy. A policy is said to be proper if, for any possible initial state, it ensures that a goal state is reached with probability 1.0. We assume that our problems have at least one proper policy, and that every improper policy has infinite expected cost for at least one state.

Different from the traditional offline search methods that first plan a path from start to goal state and then move the agent along the path, real-time search methods have been proposed to interleave planning and execution and select actions in a limited amount of time. Since the agent does not plan all the way to the goal, these approaches do not find the optimal solution to a planning problem, but they are guaranteed to achieve the goal of a task. More specifically, real-time heuristic search agents select actions using a limited lookahead search and evaluating the frontier states with a heuristic function. The key idea here is to utilize available domain-knowledge to guide the search and find a solution faster. Over repeated experiences, they refine the heuristic values of the states to avoid infinite loops and to converge to better solutions. This framework can be implemented in different ways, *e.g.*, with a best-first search, as in AO*, LAO* [79] and their extensions, with a depth-first iterative deepening search, as in LDFS [25], or with a random walk search, as in RTDP [16], LRTDP [26] and their extensions. Here, we discuss two popular heuristic search approaches, namely LRTDP-bel and LAO*, that solve goal POMDPs [26, 79]. LRTDP-Bel is an adaptation of the Real-Time Dynamic Programming (RTDP) to goal POMDPs. LAO* is the generalization of A* on And-Or graphs that can be used to solve goal POMDPs and find solutions with loops [79]. We discuss these two approaches in details below.

RTDP-Bel is a generalization of the LRTA* algorithm [81, 104] for non-deterministic settings [26]. RTDP-Bel runs a forward simulation from the start to the goal called a trial. At each node the algorithm uses a greedy approach to select an action and then randomly selects an observation. A hash table with the discretized belief state as its key keeps track of the value function. The algorithm keeps updating the value function for the nodes visited on the path over the iterative trials. A variant of RTDP-Bel called Labeled RTDP-Bel presents an approach to label the states with converged values as solved, so that the value function can converge faster [26]. The algorithm has a good anytime behavior by producing a solution fast and improving it with time. A drawback of this algorithm is that unlikely paths are ignored and RTDP-Bel converges slowly in those cases.

AO* is the generalization of A* on the And-Or graphs (without loops). A POMDP also

includes the And edges to represent the observations and the Or edges to represents the actions. AO* algorithm repeats forward expansion of the best node in the current belief tree and backward induction from the leaf node back to its predecessors. Forward expansion is guided by an admissible heuristic that helps to explore a region where the optimal solution is likely to reside. LAO* represents AO* algorithm for MDPs or belief MDPs with Loops [79]. LAO* generalizes the backward induction process to value iteration or policy iteration. Thus, it requires more computation compared to AO* but can handle AND-OR graphs with loops. Once it reaches the goal node for the first time, then it returns the solution. If the robot plans all the way to the goal rather than interleaving planning and execution, it is proven that the LAO* algorithm converges to the optimal solution. Different from the RTDB-Bel approach which performs the search in a depth-first fashion. LAO* uses a best-first search approach.

Chapter 3

Formalization of the Restaurant Domain

3.1 Motivation

A robot waitress working in a restaurant should take care of an ongoing stream of tasks, including delivering hot food, taking drink orders, taking food orders, checking on customers who have just received their meal, and finally, cashing out a table about to leave. Attending the customers' needs in a timely and efficient manner is very challenging. One approach to address this problem would be to attend the customers on a first come, first served basis, *i.e.*, based on the amount of time each customer was waiting. This approach might be practical in some domains, but is not effective in a restaurant domain where the customers' satisfaction does not only depend on the wait time, but it also depends on the task's features, and how important the task is. In this chapter, we explain how we formalize the dining process as a robot planning problem.

The restaurant domain has the following two main challenges. First, the robot waitress should be able to effectively multi-task by keeping track of everything that needs to be done and prioritizing them so the most important tasks get done quickly. For example, a robot who has just received an order from a table should send that order to the kitchen as soon as possible so things don't get backed up down the line. Second, the robot waitress should plan routes around the restaurant, taking into account the tasks that needs to be done at each station. The fewer trips the robot makes between the bar and the kitchen, the sooner the customers get their food. For example, if the robot needs to pass a table of customers on the way to the bar, it should stop by the table briefly to check on the customers before proceeding to the bar.

Restaurant waiters are capable of addressing these two challenges (and many more challenges) by 1) leveraging an internal model of how the customers needs and satisfaction evolve throughout the dining process, and 2) multitasking and navigating in the restaurant to serve as many people as possible. By using these models the waiter is able to make decisions on what actions she should

perform to gain the maximum final tip. We model the waiting table task as a planning problem to enable a robot to address the same challenges as a restaurant waiter.

There has been work on robot localization and navigation in a restaurant [149, 206]. Different from these works, we assume the robot is capable of localizing within the restaurant, and we focus on how the customers' satisfaction of the service, which is not directly observable, is affected by the robot's actions. Task planning is another area of research that has been studied in the service robot domain [94, 129]. These works either do not model the customers' satisfaction or model it as an observable variable and use it to prioritize the next task. Differently, we are interested in how the robot's sequence of actions maximizes the customers' long-term satisfaction. Furthermore, these works consider each task as an indivisible task. We allow the robot to diverge from its current task to service more customers and make fewer trips. There has been work on predicting customer's state in a restaurant or a bar [39]. They focus on inferring the customers internal state and using that to select a robot behavior. In contrast, we focus on how the robot's sequence of actions impacts the customers' long-term satisfaction.

We enable the robot to reason about the unknown customers' satisfaction so that it can effectively prioritize the tasks to keep all the customers satisfied. The robot should also reason about its long-term effects of immediate actions to foresee what will happen in the future so it can plan ahead. Partially observable markov decision processes (POMDPs) [36] provide a mathematical tool to achieve these objectives. A POMDP is capable of modeling a robot's sequential decision making process, while also being able to represent uncertainty in the robot's execution and perception. POMDPs have been applied to many real-world problems in the context of human-robot interaction [12, 135]. However, we are not aware of any approaches that leverage multiple POMDP models to formalize the restaurant domain.

3.2 Formulation

As illustrated in Fig. 3.1, we consider a restaurant setting where one robot services N tables. We do not model each customer in the restaurant individually and only assume one model for each table in the restaurant. We assume that the states of the humans on a particular table are aggregated to have one estimate for the table. We formalize the waiting tables for a restaurant as a planning problem with one robot and N independent models for the N tables in the restaurant. We consider each table from when the customers sit at the table to when they leave as one task. The robot keeps a distribution over the state of the tasks and updates them as it executes actions. At each time step, the robot decides what action should be executed with respect to which task. This enables the robot to consider switching between the tasks after each action execution. The

Figure 3.1: A restaurant setting with 3 tables and one robot.

robot can only execute one action at each time that depends on the duration of possible actions, the state of each table, and how these tables evolve, *e.g.*, the table becomes unsatisfied if it is not served soon or the food becomes cold after a few time steps. While the robot services one table, the other tables evolve according to their underlying Hidden Markov Model (HMM).

To enable the robot to perform the waiting tables task, we model each table, the state of the robot and the actions that can be applied to this particular table as a POMDP. Thus, there are N POMDPs in the restaurant that share the robot between themselves as shown in Fig. 3.2. This POMDP representation enables the robot to reason about the uncertainty in the table's internal state and its own actions and how the dining process for that specific table evolves after a sequence of action executions.

We provide the general representation of the POMDP model for a table below. We also discuss an example implementation of each element of the POMDP model for the restaurant setting that we used in the experiments of Chapters 5, 6 and 7.

- **State space** S: For one table, the full state of its POMDP is $S = SR \times SC$ where SR is the robot's state space and SC is the human's state space. If we represent an element of the robot's state $sr \in SR$ and an element of the human's state $sc \in SC$ in vector form, an element of the table's state $s \in S$ is a concatenation of the robot's and human's state $s = (sr, sc)$. The robot's state variables include robot's state information that will be shared between the tables, and it does not include any human specific information. The rest of the state variables sc are specific to each table.

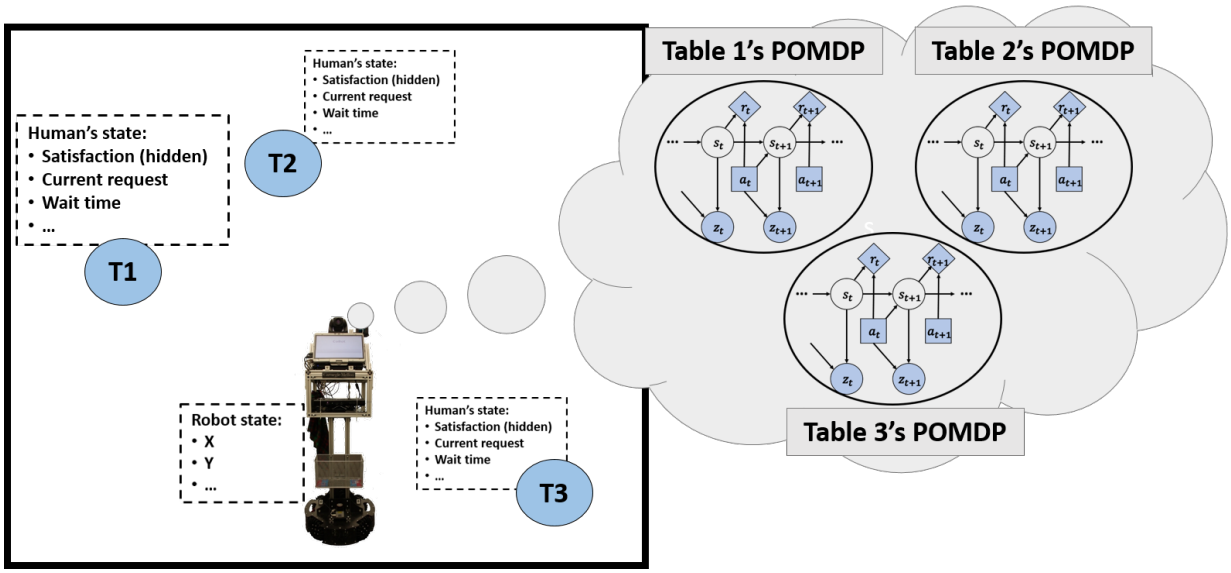


Figure 3.2: The robot operates in a restaurant with 3 tables, *T1*, *T2*, and *T3*.

- Robot's state variables $sr \in SR$ (*e.g.*, *position*)
- Human's state variables $sc \in SC$ (*e.g.*, *satisfaction level*, *wait time*, *cooking status* and *current request*)

Example S: We enumerate what state variables we use with their range in brackets in front of each variable. The state variables can take any integer values from the first number in the bracket to the last number inclusive. We also mention what each integer value represents. The time related variables have values from 0 to $time_{max} = \#$ tables $\times\ sat_{max}$ where sat_{max} is the highest value for the *satisfaction* variable. This accounts for having more time to service the customers when there are more tables.

- Robot's state sr contains x and y (11×11 grid).
- Human's state sc contains
  - Satisfaction $[0, 5]$: very unsatisfied (0), unsatisfied (1), slightly unsatisfied (2), neutral (3), satisfied (4), very satisfied ($sat_{max} = 5$)
  - Food $[0, 3]$: not served (0), plate-full (1), plate-half (2), plate-empty (3)
  - Water $[0, 3]$: not served (0), glass-full (1), glass-half (2), glass-empty (3)
  - Cooking status $[0, 2]$: cooking-started (0), food-half-ready (1), food-ready (2)
  - Current request $[1, 8]$: want-menu (1), ready-to-order (2), want-food (3), want-drinks (4), want-bill (5), cash-ready (6), cash-collected (7), table-needs-to-be-cleaned (8)

  - Hand raise $[0, 1]$: no-hand-raise (0), hand-raise (1). This variable represents if the customers have a request or not. This variable is always 1. After the customers leave, it becomes 0.
  - Time since food or water has been served $[0, time_{max}]$: this variable represents the number of time steps since the customers started eating or drinking. It affects the value of *food* and *water*. We provide more details below.
  - Time since food is ready $[0, time_{max}]$: this variable represents for how many time steps the food (or drinks) has been ready to be delivered to the customers. We provide more details below.
  - Time since request $[0, time_{max}]$: this variable represents for how many time steps the customers at the table have been waiting to be serviced.

- **Action space** A: The full action space is a set with all the following actions $A = \{AN, AC, AS, AI, \textit{no op}\}$.
  - Navigation actions AN (*e.g.*, *go to the table* and *go to the kitchen*)
  - Communication actions AC (*e.g.*, *food is not ready* and *I'll be back to serivce your table*)
  - Service actions AS (*e.g.*, *serve food* and *give the bill*)
  - Information gathering actions AI (*e.g.*, *ask if the customers are ready for the bill*)
  - A special *no operation* action

  Example A: $A = \{AN, AC, AS, \text{no op}\}$.
  - Navigation actions AN: *go to the table*.
  - Communication actions AC: *food is not ready* and *I'll be back to your table later*.
  - Service actions AS: One *serve* action. Depending on the table's *current request*, the appropriate service action gets executed. For example, if the table's *current request* is *want-menu*, executing the *serve* action when the robot is at the table represents handing over the menu.
  - The special *no operation* action.

  In this instance of the restaurant model, we do not include information gathering actions.

- **Transition function** T: We assume the robot's next state sr' is independent of the human's current and next states, sc and sc', and only depends on its own current state sr and action a. Similarly, the human's next state sc' is independent of the robot's current and next states,

sr and sr', and only depends on human's state sc and action a. Thus, we can decouple the transition function as follows $T(s, a, s') = \Pr(sc'|sc, a) \Pr(sr'|sr, a)$.

Example T: We assume that the robot's actions with respect to its own state are deterministic (*i.e.*, robot's position changes deterministically). Each table goes through a consecutive sequence of 8 requests that we mentioned above (the 8 values for the *current request* variable). The table gets served if the robot performs the *serve* action at the table. After each *serve* action, the value of the *current request* is updated to the next value (*current request*$+1$). The *time since request* variable resets to 0 when a table is served, otherwise it goes up till it reaches its maximum value. The actions increase the relevant time-related variables of the state by 1 except the navigation actions which can take 1, 2, or 3 time steps depending on where the robot is with respect to the tables.

If a table is waiting for the food, the *cooking status* variable increases by 1 after $\frac{time_{max}}{3}$ time steps until the food is fully cooked and ready to be served, *i.e.*, *cooking status* $= 2$. This means the kitchen takes $\frac{2*time_{max}}{3}$ time steps to prepare the food. When the food is ready to be served, the *time since food is ready* variable keeps increasing until the robot serves the table, *i.e.*, the robot goes to the table and executes the *serve* action, or it reaches its maximum value. If the value of the *current request* is *want-food* or *want-drinks* and the robot serves food or drinks, the table starts its eating or drinking process. We assume each table takes $\frac{2*time_{max}}{3}$ time steps to finish eating (or drinking). The *time since request* variable does not go up when the people are eating or drinking. The *time since served* variable resets to 0 when the food or drinks are served and goes up till the customers finish their food ($food = 3$) or water ($water = 3$). The *time since served* variable is used to keep track of time while the customers are eating or drinking and affects the value of *food* and *water*. The *food* and *water* variables represent the table's eating and drinking process and increase after $\frac{time_{max}}{3}$ time steps until they reach their maximum value.

All the variables except *satisfaction* transition deterministically. The *satisfaction* variable goes down by 1 after $\frac{time_{max}}{sat_{max}}$ time steps; if a customer is very satisfied at the beginning and does not get served within $time_{max}$, she becomes very unsatisfied. When the table is waiting for food, the *satisfaction* variable goes down by 1 after $\frac{time_{max}}{sat_{max}+1}$ time steps. Here the assumption is that if the people are waiting for food, their satisfaction level decreases faster than when they are waiting for other reasons. When a table is served its *satisfaction level* changes stochastically. If they are very unsatisfied, their satisfaction level increases by 1, 30% of the times, and stays the same, 70% of the times. If they are very satisfied, their satisfaction level does not change. Otherwise, their satisfaction level increases by 1, 60% of

the times, and stays the same, 40% of the times. This means that it is much harder to make the very unsatisfied tables a bit more satisfied than making the satisfied tables very satisfied.

- **Observation space** Z: We assume that the robot's state is fully observable. The human's state sc may be partially observable. For example, the robot may execute an action and observe the human's *neediness* which might give information regarding the human's *satisfaction level*. For the part of the human's state that is observable, the model has the same number of observation variables as the number of state variables. The model can have any number of observation variables for the partially observable part of the state sc.

  Example Z: In an instance of the restaurant model that we consider, we assume that we have one observation variable per each state variable except for the *satisfaction level* that is hidden. As the robot's state is fully observable and the actions of the robot with respect to its own state are deterministic, we do not consider any observations associated with the robot's state. This means we have 8 observation variables for the following state variables: food, water, cooking status, current request, hand raise, time since food or water has been served, time since food is ready, and time since request.

- **Observation function** O: The model does not have observations for the robot's state. The human's state can be partially observable, thus the observation function only depends on the human's state sc', action a and the observation of the human's state z, $O(s', a, z) = \Pr(z|sc', a)$.

  Example O: We only consider *satisfaction level*, sat, as a hidden variable, and the other variables, let's call them zh, are observable. This means that $\Pr(z_{t+1} = zh|sat, zh, a) = 1$ and $\Pr(z_{t+1} = zh'|sat, zh, a) = 0$ where $zh' \neq zh$. Since *satisfaction level* is the only hidden variable, the belief state of the POMDP keeps a distribution over this variable.

- **Reward function** R: The reward function specifies how the robot should service the table.

  Example R: In an instance of the restaurant model that we consider, we consider servicing a table has a positive reward inversely proportional to the table's satisfaction. If the table is unsatisfied and waiting to be served, a negative reward is given. Navigation actions incur a negative reward. This reward function encourages the robot to service the table sooner if it is unsatisfied. The reward function is as follows:

$$time = min(\text{time since request}, 10)$$

$$R(s, \text{serve}) = 5 * (sat_{max} - sat' + 1)$$

$$R(s, \text{go to}) = -1$$

$$R(s, \text{other actions}) = \begin{cases} -2^{time} & sat' = 0 \\ -1.7^{time} & sat' = 1 \\ -1.4^{time} & sat' = 2 \\ 1 & sat' = 3, 4, 5; sat' > sat \\ 0 & \text{otherwise} \end{cases}$$

This concludes the representation of the POMDP model for a single table. In the next section, we provide a discussion on what assumptions we made regarding the restaurant model and how that would relate to a real restaurant setting. We add a subscript i to the elements of table i's POMDP model when we refer to them.

3.3 Assumptions

We assume each table goes through the dining process independently of each other. More specifically, our formulation makes the following assumptions regarding the POMDPs:

- We make the following assumptions regarding the state, action and observation space of the tables. The models' state space only share the robot's state sr between themselves. The rest of the state variables sc are specific to each table. Other than the *no operation* action, the models' action space do not share any other actions. The models' observation space do not share any observation variables. We believe that these assumptions can be valid in real-world restaurants unless the customers on two different tables booked a reservation together and are going through the dining process together, in which case they should be considered as one table.
- Table i's transition probabilities are a function of the human's ($sc, sc' \in SC_i$) and robot's current and next state ($sr, sr' \in SR$), and the robot's action ($a \in A_i$). The transition probabilities do not depend on the state of the other tables. When an action gets executed by the robot, if the action belongs to table i, $a \in A_i$, all other tables other than table i transition as if *no operation* has been executed on them for the duration of a.

  In reality this assumption might be invalid, and T_i might depend on the state of table j or the specific action that is being executed on table j ($a \in A_j$). For example, if table i perceives

that the customers at table j that have been waiting less than them got served before them, their level of satisfaction might decrease.

- Table i's observation probabilities are a function of the human's state ($sc \in SC_i$), the robot's action ($a \in A_i$), and the robot's observations of the table i ($z \in Z_i$). The observation function does not depend on the state or robot's observations of the other tables.

  In reality this assumption might be invalid, *e.g.*, if a table's observed neediness changes based on a nearby table's status, *e.g.*, if the nearby table is eating.

- The tip or the reward that the robot receives from a table only depends on the human's ($sc, sc' \in SC_i$) and robot's current and next state ($sr, sr' \in SR$), and the robot's action ($a \in A_i$). The total tip or the reward the robot gets is a sum of all the tips or rewards from all the tables in the restaurant.

  In reality this assumption might be invalid. For example, if the reward that is given to the robot is based on being served after a nearby table.

Although some of the assumptions that we made regarding the restaurant domain might not always hold in a real-world restaurant setting, we believe they provide a good approximation of a real-world restaurant setting. Relevant works on the restaurant domain also implicitly make these assumptions. Our work extends the previous works by looking at the outcome of robot decisions in the long-run and modeling the internal state of the humans. This section focused on formalizing the restaurant domain and discussing what assumptions we make regarding its model.

One way to approach the waiting tables task could be to solve each POMDP separately and compute the average reward that the robot gets to service each table. The robot can then select the table with the highest reward to attend to (Strategy 1). This strategy does not take into account sequencing the tasks to achieve the maximum total tip or reward from all the tables. Below, we provide an example that illustrates why the robot should combine all the POMDP models to build a model with all the tasks' states, all possible actions, and robot's state, and consider all possible sequences of servicing the tables (Strategy 2).

Let's assume that T3 in Fig. 3.2, is currently extremely unsatisfied and needs to be serviced, and on the other side of the restaurant T1 and T2 are moderately unsatisfied. If the reward function is inversely proportional to the table's satisfaction, Strategy 1 leads the robot to attend to T3 first. In this case by the time the robot services T1 and T2, the tables might be extremely unsatisfied. However, in Strategy 2, the robot services T1, T2 and then T3, rather than servicing T3 first and then coming back all the way to service T1 and T2. Thus, even though we make the above assumptions and assume that the tables are independent, the optimal policy must consider all incomplete tables at each step since they share the robot among themselves. Instead of solving the

huge model of the restaurant domain, our algorithms in Chapters 5 and 6 use the assumptions that we mentioned above to speed up planning.

3.4 Conclusion and Discussion

We focus on planning for a robot waiter that operates in a restaurant and is presented with an ongoing stream of tasks. A typical restaurant waiter has an internal model of how the customers' satisfaction evolve during the dining experience, and how her actions impact them. We present our first steps to enable a robot to take on the waiting tables task. We formalize the task of waiting tables as a robot planning problem to enable a robot waiter to leverage an internal model of the customers' satisfaction to guide its decision making. We discuss why our formalization of the N tables as N independent POMDP models is suitable for the task of waiting tables and expand on the computational challenges of planning for the restaurant setting.

The assumption that the N tables are independent helps us in addressing some of the computational challenges for such domains; however, in a real restaurant setting, the independence assumption might not hold. If the independence assumption does not hold for certain tables, one can combine the POMDPs associated with those dependent tables. Note that although we consider the same POMDP implementation for the different tables, one could have different POMDP implementations depending on different types of customers. The focus of this work is mostly on the algorithmic challenges of solving these multi-task models rather than designing a restaurant model that perfectly matches a real restaurant. We will discuss some of the improvements that can be made on the restaurant model in Chapter 9.

Chapter 4

Efficient Task Execution by Using Interruptions to Switch Among Multiple MDP Models

Our work in this section focuses on a multi-task domain where a service robot is executing a user-requested task, *e.g.*, object delivery, and is also alert for multiple external tasks that might come up. Different from the restaurant domain where the robot should attend to all the tables (or tasks), in this domain the robot is required to accomplish the user-requested task, but the other external tasks are optional. Since the external tasks are optional (and independent), the robot does not need to solve the large combined model associated with all the tasks at all times and can speed up the *task execution* by solving the large combined model only occasionally. This chapter targets the service robot domain and provides an approach to expedite the task execution.

4.1 Motivation

Our service robot can successfully perform user-requested tasks by executing a single planned task at a time [188]. Our goal is to make the robot more responsiveness to its environment by equipping it with additional optional tasks and enabling it to interrupt the execution of the user-requested task to achieve the additional tasks. For example, if the robot is scheduled to deliver a document to an office, and on the way to the office it sees a person asking for help, we might want the robot to first assist the person and then finish its delivery. In the service robot domain, the tasks have internal states, *e.g.*, the document is really needed when the user finishes her lunch, so task-scheduling approaches that only consider temporal constraints are not applicable. The robot should leverage the requested task's model to decide if achieving a few additional tasks before addressing the

requested task will risk the accomplishment of the task or not. This domain is different from the restaurant domain since the robot has one task that it needs to accomplish and the rest of the tasks are optional. The tasks are represented as Markov Decision Processes (MDPs) in which there is uncertainty over the outcomes of actions, but the states of the tasks are fully observable.

We provide an approach that enables a robot to increase its responsiveness to its environment by interrupting its task execution and switching to an optional task when appropriate. We take on a reinforcement learning (RL) approach to learn the task-switching behavior. One way to learn this behavior is to encode all the details of the tasks' structure and environment in one combined model and use an RL algorithm (or a planning approach) to find an optimal policy [178]. Notice that although except one task the other tasks are optional, the robot has to consider all the existing tasks, namely the user-requested task and all the current optional tasks, in one large model to decide what task to execute next. This is because the robot should consider all possible sequences of achieving the tasks to make a decision. This approach has a large number of state variables with the potential of high computational complexity. Another drawback of the combined model approach is that the robot should process all the sensory state variables for all the task models during the execution of the combined model. In most robotics applications, there is a cost associated with processing the sensory state variables, *e.g.*, cost for human pose estimation and speech recognition, and it is often not feasible to process all sensory inputs.

To address the computational impracticality of using the large model, we provide an algorithm that speeds up task execution by deciding how often the robot should solve the large model, rather than solving the large model at each time step. The robot only focuses on one task at a time and uses the large model when it is triggered. We call the signals that trigger the task-switching, "stimuli". When a stimulus is triggered, the robot builds the large model with all the tasks to decide if a switch is more rewarding than continuing with the current task. There are ways to hard-code the switching response for a specific problem. However, we target service robot domains in which the tasks have a lot of state variables (*e.g.*, human's distance and speed, human gaze, distance to obstacles) and hard-coding the switching response is not feasible. We provide an algorithm that speeds up the execution of the tasks by identifying the stimuli that trigger the task-switching. We identify these stimuli by identifying the sensory inputs that have a higher impact on task switching and show that our solution using the switching stimuli compares favorably to the naive approach of considering all the tasks. Moreover, leveraging the stimuli significantly decreases the amount of sensory input processing during the execution of tasks.

We propose a two-step solution. In the first step, *learning*, the robot learns a task selection policy that specifies which task should be executed at each world state. We formulate the task-switching problem as a Markov Decision Process (MDP) and leverage a Dueling Deep Q-Network

architecture to solve it [195]. In the second step, *identification*, we speed up the execution of the task models by identifying the stimuli that trigger the task-switching. Leveraging the stimuli enables the robot to focus on only one task at a time.

In summary, we make the following contributions: 1) a novel approach that learns a mapping from world states to task models (*learning* step), 2) a novel algorithm that identifies the stimuli that trigger the switch between task models (*identification* step), 3) our approach enables a robot to be more responsive to its surrounding environment, and 4) our two-step algorithm significantly decreases the amount of sensory input processing during the execution of the tasks.

4.2 Approach

In this section, we explain how we formalize the task-switching problem as an MDP. We then discuss how we identify the stimuli that trigger the task-switching behavior.

We consider a scenario in which the robot is executing a user-requested task, *e.g.*, object delivery, and is also alert for other observations like humans and objects (*e.g.*, trash) around it. These observations may lead the robot to interrupt the current task execution if the switch to a human interaction or trash cleaning task results in a higher future utility.

4.2.1 Learning Task Selection Policy

Our task selection algorithm is provided with a set of n task models; each task model in this set achieves one goal. This set includes the user-requested task and all the optional tasks. The output of the algorithm is a policy, denoted by $\pi_{select-task}$, that specifies which task model should be executed given an observation of the environment. Each task model, denoted by T_i for the i^{th} task model, is represented as an MDP [99]. Solving the i^{th} task model's MDP by a value-function based approach provides a policy π_i and a value function V_i. The policy π_i specifies the best action that should be taken in each of the task model's states. The value function V_i specifies how good each state is from the perspective of the i^{th} task model. The ultimate goal of the task selection algorithm is to use π_i's and V_i's, and learn when to switch between the task models.

Problem formulation:

We formalize the task-switching problem as an MDP, and we refer to it as "switching MDP". The MDP's representation[1] is as follows:

[1]We assume a discount factor of 0.99 in all our experiments.

- ***States***: In a domain with n task models, the switching MDP's state is $[V_1(S_1), V_2(S_2), \ldots, V_n(S_n)]$. State S_i represents the state of the i^{th} task model, and $V_i(S_i)$ represents the expected reward when starting in S_i for the i^{th} task model.
- ***Actions***: In a domain with n task models, the actions are *execute* π_1, *execute* π_2, ..., and *execute* π_n. Each call to *execute* π_i executes an action based on policy π_i of task model T_i.
- ***Transition function***: This is a model-free approach, so the robot directly learns the policy by trial-and- error interactions with the environment.
- ***Reward function***: The reward function specifies the relative utility of the tasks to each other. Thus, it determines the multi-task behavior of the robot.

Learning $\pi_{select-task}$:

Our MDP's state space is continuous, therefore we use a neural network to approximate the Q function. Some work provides a variant of Q-learning called Deep Q-Network (DQN) that stabilizes RL by utilizing an experience replay mechanism and a second fixed target network [125]. In addition to utilizing these two improvements of DQN approach, we leverage the dueling architecture which compared to DQN provides a more robust estimate of the Q function and has shown better performance in domains with many similar-valued actions [195].

We use our MDP formulation of the task-switching problem and apply the deep Q-network method to approximate the task-switching policy. To learn the task-switching policy for n task models, our deep model gets $[V_1(S_1), V_2(S_2), .., V_n(S_n)]$ as an input. The output of our network specifies which π_i should be executed given the input to the network. Fig. 4.1 shows what happens in one step of learning if our domain consists of n task models. If $n = 3$ the robot observes the environment, updates S_1, S_2, and S_3, and computes the corresponding $V_1(S_1), V_2(S_2)$, and $V_3(S_3)$. The robot picks one of the task models (*e.g.*, T_1) and executes one step of its policy (*e.g.*, if $a_3 = \pi_1(S_1)$, the robot executes action a_3). The robot observes the new state and the reward of the action execution, and updates the parameters of the network. The robot keeps updating the parameters by applying the DQN approach until the loss converges to 0. Each learning episode terminates when the robot accomplishes the user-requested task.

Executing $\pi_{select-task}$:

We explain how the robot uses the trained network at execution time to perform the following task: the robot is scheduled to deliver an object to a location (main task), while interacting with the people around it. The robot starts going to the goal location to deliver the object. At each step during execution, the robot computes an array of V_i's and passes it to the network. The

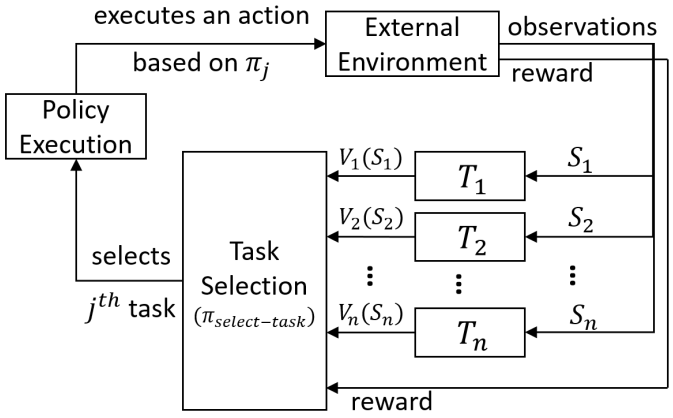


Figure 4.1: Overview of the task selection module.

network then selects a task, and the robot executes an action from the task's policy. In our example scenario, the robot might see multiple people in the scene. Interacting with each person is a different task. The robot might choose to interact with a person, and then decide to interact with another person or return to the object delivery task.

The Q-values and V-values have been used in other work to address the action (model) selection problem [83, 96, 179]. Their algorithms use heuristic values calculated from Q-values and V-values to select actions from different modules. Similar to their approach, our tasks are learned with different reward functions, but our approach trains a model based on the global reward function that specifies the relative utility of the tasks to each other. The robot learns a behavior that specifies how it should switch between its multiple tasks to gain the highest utility. Notice that just getting the maximum of V-values does not work for the following reasons: 1) each task is learned separately with a different reward function, and 2) the robot might greedily select a task that is closer and miss a dense group of rewarding tasks further away. Our task-switching formulation has the following characteristics:

- The structure of our task selection algorithm is independent of the number of state variables and actions in each task model's MDP, and the size of the input to the deep network increases linearly as the number of task models increases. This is because each task model is learned separately using its own reward function, and the high-level task selection module only learns how to switch between the tasks to get the highest utility.
- The task selection module might not learn the combined model's optimal solution in some cases since it only uses the individual task models' policy and value function. However, training the task selection module is faster than solving the combined model. The task selection algorithm favors computational feasibility over optimality.

- Although the task selection algorithm enables the robot to switch between multiple task models, the robot is still processing the same amount of sensory input as the combined model during the execution phase. More specifically, the robot should first update all the state variables and then use them to calculate V_i's. That is to say, in a real scenario, while the robot is executing a navigation task, it should also process all the state variables of a Human-Robot Interaction task to decide if a switch to a different task is appropriate. We introduce the notion of task-switching stimuli in the next section to decrease the amount of sensory computations at the execution time.

4.2.2 Identifying Task-Switching Stimuli

On one end of the spectrum, we have approaches like our task selection algorithm that process all the sensory information for decision making. These approaches work if the total time of processing the sensory information is less than the desired response time of the robot. However, as we increase the number of sensors on robots and the tasks become more complex, these approaches become infeasible. On the other end of the spectrum, we have approaches that only focus on one task at a time. These approaches are not responsive to their external environment which makes them less effective in real-world applications. We are interested in a method that trades-off between the amount of sensory input computations and responsiveness of robots. We introduce another algorithm that leverages stimuli to address this trade-off.

A "stimulus" (plural stimuli) is a detectable change in the internal or external environment of a robot that causes a reaction in the robot. These stimuli, when triggered, interrupt the robot to assess if switching to another task model is beneficial. The information that a robot requires to execute actions and achieve a task model's goal is already provided in the task model's state variables (features). We introduce an algorithm that takes as input the state variables of each task model and selects the most informative state variable as the stimulus that triggers the reevaluations of the task selection policy $\pi_{select-task}$. For each task model, our algorithm computes a sorted list of the task model's features with their associated importance percentage.

Our key idea is to gather examples of observations where the robot switched to another task or did not switch to another task and then use this labeled data to find out what features have the biggest impact on the task-switching. Alg. 1 presents the stimuli identification algorithm. The stimuli identification algorithm takes as an input a list of target task models U, the task selection policy $\pi_{select-task}$, and the main robot task c_{main}, and computes U's features' importances. We run N simulations (line 2) with different goal locations and random initial values for the features (line 3) and evaluate the task selection policy at each step of the simulation (line 5). To find the features that have the most impact on the task-switching policy, the algorithm gathers examples

of observations where the robot decides to switch to another task or continue with the current task. For each target task model, the algorithm considers an empty positive and negative example sets. At each simulation step, if we switch from the task model A to the task model B, we add the current state of the task models $U - B$ to their negative example set (line 8) and the current state of the task model B to its positive example set (line 7); otherwise if we do not switch from the task model A to the task model B, we add the current state of the task model A to its positive example set, and the current state of the other task models $U - A$ to their negative example set. Notice that if the robot does not switch from A to another task model, we still consider the state as a positive example for A, since A is preferred to the other task models in that state. We keep updating the positive and negative example sets for the tasks until all the simulations terminate.

Algorithm 1: Task-Switching Stimuli Identification. The algorithm takes the task selection policy $\pi_{select-task}$, the number of simulations N, the main robot task c_{main}, and a list of target task models U as input.

```
1  StimuliIdentification (π_select−task, c_main, N, U)
2      for 1 to N do
3          Randomly reset all task models
4          while c_main not done do
5              c_state ← ComputeTaskmodelValues() // computes state of the switching MDP (V_i's)
6              c_task ← π_select−task(c_state) // selects a task model
7              Add the state of c_task to its positive set
8              Add the state of U − c_task to their negative sets
9              Execute(c_task) // executes one step of c_task
10     for task ∈ U do
11         data, label ← GetSensoryData(task)
12         clf ← FitClassifier(data,label)
13         task.importances ← FeatureImportances(clf)
```

Algorithm 2: Task-switching behavior. The algorithm takes the task selection policy $\pi_{select-task}$ and the main robot task c_{main} as input.

```
1  MultiTaskMDPPlanner (π_select−task, c_main)
2      c_task ← c_main
3      while c_main not done do
4          while stimuli not triggered & c_task not done do
5              Execute(c_task)
6          c_state ← ComputeTaskmodelValues()
7          c_task ← π_select−task(c_state)
```

We formalize the identification problem as a classification problem. The stimuli identification

algorithm first filters out all nonsensory features since they cannot be used as a stimulus to interrupt the task execution (line 11). The sensory feature vectors with their labels are then provided to the classification algorithm (line 12), and the feature importances are computed (line 13). The feature with the highest importance is selected as the stimulus for the target task.

Alg. 2 describes how the *learning* and *identification* steps of our approach are integrated and executed by the robot. The robot starts with the main robot task, *e.g.*, object delivery task, and keeps executing it (line 5) until a stimulus is triggered, *e.g.*, the robot sees a person, or the current task is done (line 4). The robot then computes all the sensory variables and selects a task model to execute (lines 6-7). Notice that while executing the current task c_{task}, the robot only processes the sensory variables of c_{task} and the stimuli.

4.3 Experiments

In this section, we discuss the results of our task selection and stimuli identification algorithms in a scenario with 1 to 6 tasks (*e.g.*, the object delivery and human interaction tasks) that our service robot encounters everyday in our building.

4.3.1 Neural Network Structure

The network gets as input an array with size equal to the number of task models. This is followed by 3 hidden layers, each with 60 neurons and ReLU activation functions. The output layer has size equal to the number of task models. We sample uniformly a batch of size 32 from the replay memory of size $50,000$ to perform each update. We use a linear decay epsilon greedy policy with maximum value 1 and minimum value 0.1 and the Adam stochastic gradient descent method as the optimizer with learning rate 0.001 [98]. We use the same network structure and parameters in all our experiments. Instead of applying a hard update on the network, we use a soft update method with smoothing parameter $\alpha = e^{-2}$ to update the model. The parameters of the DQN approach, *e.g.*, minibatch and replay memory size, are equivalent to the ones used by other works in deep RL [125].

4.3.2 Feature Importance Computation

In order to compute the feature importances, we apply the Extra-trees algorithm on the positive and negative example sets[2] [73]. We used the extra-trees algorithm with 1000 estimators, *i.e.*, 1000 trees in the ensemble, gini criterion and maximum depth of 4. Features with higher ranks,

[2]We used the scikit-learn implementation of the algorithm [141].

i.e., at the top of the tree, contribute more to the final classification decision of a larger fraction of the examples. The expected fraction of the examples that each feature contributes to is used as an estimate of the relative importance of the feature. Averaging the relative importances over several randomized trees produces the feature importances for each target task model.

4.3.3 Simulation Setup

We tested our task-switching behavior in an 11×11-grid environment (Fig. 4.2) with three types of tasks: an object delivery task with 3 features and 3 actions, a trash cleaning task with 4 features and 5 actions, and a Human-Robot Interaction task (HRI) with 7 features and 7 actions. When an action gets executed, the tasks that do not include the action in their action space transition as *no operation* has been executed on them. Except for the x and y position of the robot, all other features are binary. The robot's position and the robot's navigation actions are only shared between the tasks. In all experiments, the robot is performing an object delivery task while interacting with 0 to n people or executing 0 to n trash cleaning tasks. Thus, the number of tasks ranges from 1 to $n + 1$.

We build a huge MDP with the $n + 1$ tasks, and we call it "exact MDP" since it computes the exact solution to our problem. The exact MDP would have $5n + 3$ state variables, $121 \times 2^{5n+1}$ states, and 8 actions if we use the HRI task and $2n + 3$ state variables, $121 \times 2^{2n+1}$ states, and 6 actions if we use the trash cleaning task. We compare our switching MDP's solution to the solution of the value-iteration algorithm on the exact MDP. In both MDPs, the robot gets -0.1 reward for each action execution. The reward of interacting with a human goal, cleaning trash, and delivering an object is 1, 1, and 0 respectively. The episode terminates when the robot achieves the delivery task regardless of whether it interacts with the humans or performs the trash cleaning tasks. We consider the same setup for our switching MDP. With $n + 1$ tasks, our switching MDP has $n + 1$ state variables and $n + 1$ actions.

4.3.4 Results of Task-Switching Behavior

We evaluate our task-switching behavior in different setups and show its benefits over the exact MDP. Here, we assume that the switching stimulus is detected by the stimuli identification algorithm, and we provide the results of the identification step in the next section. We report the final results of our task-switching approach in three evaluations:

1. In the first evaluation, we used one object delivery task and 1 to 4 different trash cleaning tasks and compared the switching MDP and the exact MDP's performance during training. While

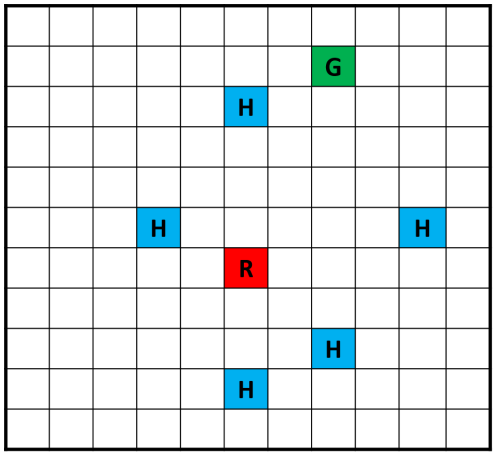


Figure 4.2: 11×11-grid environment with the robot (R), the navigation goal (G), and 5 humans goals (H's).

executing the object delivery task, the robot observes 1 to 4 different trash goals, and it should decide if it should switch to another task. Fig. 4.3 shows the performance of our switching MDP compared to the exact MDP (dashed lines). To compute the performance, we average the final reward of running 120 simulations with random initial values for the state variables. The final reward of a simulation is the total reward of its episode until it terminates or the robot reaches the maximum number of steps, which is set to 150. Fig. 4.3 shows our dueling Q-network performance is very close to the exact solution at the end of the training process.

As the number of tasks increases, the neural network requires more training steps to converge to the optimal solution. For two, three, four, and five tasks, the network required $25,000$, $30,000$, $60,000$, and $70,000$ training steps respectively. Due to space constraints, here we briefly highlight the advantages of our approach, in terms of time complexity of the learning phase, compared to the naive approach. To better illustrate the computational complexity of the combined model when the state space is just slightly bigger, we introduce a new task HRI-13 that has the same 7 variables as the HRI task, and we add 6 more binary variables to it.

The value-iteration algorithm computes the optimal solutions of the object delivery and HRI-13 tasks in 0.08 and 66.04 seconds (s) respectively. For 2 tasks, object delivery and HRI-13, the naive approach takes 137.01 s to solve the MDP, and our approach takes 183.03 s to learn the switching policy. The time complexity of our approach, in total 249.15 s ($183.03 + 0.08 + 66.04$), is almost twice as much as the complexity of the naive approach. If we add one more task with 13 variables (total 25 variables), value-iteration was not able to compute the solution even after hours of waiting, but our approach in total took 336.98 s to compute the solution.

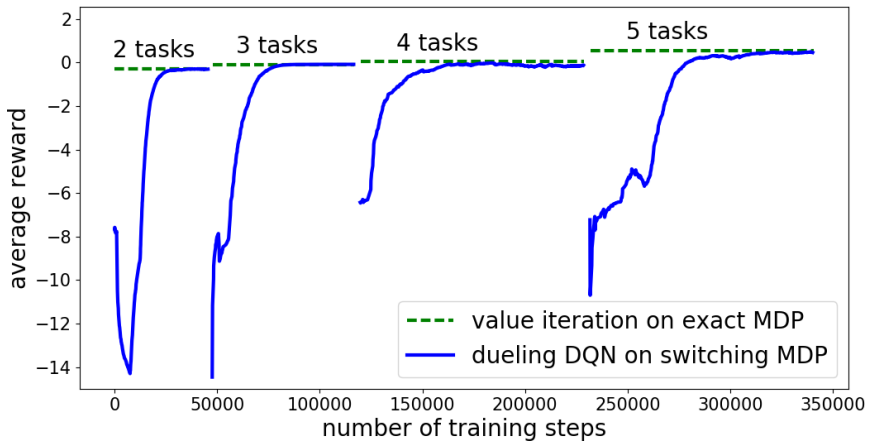


Figure 4.3: Performance of the switching MDP during the training phase for 1 delivery task and 1 to 4 trash cleaning tasks.

2. In the second evaluation, we compared the performance of our switching MDP with the exact MDP when the robot executes the final learned network. In addition to the cost of each action execution, the robot receives a negative reward for updating each sensory variable, we call this cost "observation cost" (oc). For example, the cost of detecting a person in the scene is 0.01 ($oc = 0.01$). In the exact MDP, all the task models' state variables are being computed. However, in the switching MDP, only the state variables of the current task model and one stimulus for each one of the other task models are being computed. We used the same setup as before and included the observation cost. Fig. 4.4 shows how the difference between the performance, in terms of average reward, of the exact MDP, denoted by e_r, and the switching MDP, denoted by s_r, increases as the number of task models increases. Notice the difference in performance of the MDPs is 0 when the robot is only scheduled to execute the delivery task. Although adding each trash cleaning task to the task models only increases the number of state variables by 2, with 6 task models the exact MDP on average processes 89 more state variables than the switching MDP.

3. In the third evaluation, we incorporated the observation cost into the training phase. In addition to the positive reward for achieving the goals and negative reward for each action execution, the robot gets a negative reward equal to $\#\ sensory\ variables \times oc$ in each state. This ensures that the robot considers the cost of observations during training, and if the observation and execution cost of achieving another goal exceeds its reward, the robot will not switch to the other task. Fig. 4.5 shows the results of the task-switching behavior with different observations costs. The performance is computed as before for 3 tasks, 1 object delivery task and 2 trash cleaning tasks. As the observation cost increases by 0.01, the exact MDP's average reward decreases by almost 0.8, but the switching MDP's average reward only decreases by 0.3.

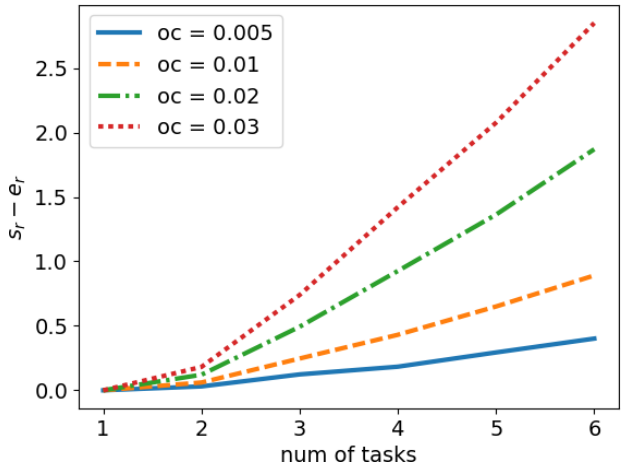


Figure 4.4: Difference between the performance of the exact MDP (e_r) and the switching MDP (s_r) when there is an observation cost (oc) for processing each sensory variable.

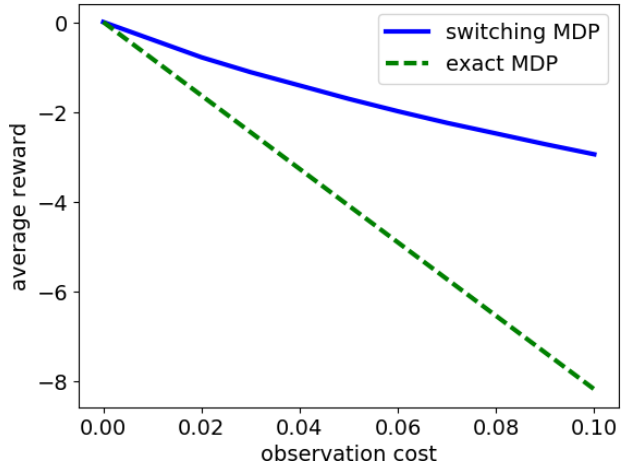


Figure 4.5: Average reward that the robot gains for 3 tasks as we increase the observation cost by 0.01.

Discussion We tested different versions of neural networks with a different number of layers and neurons, but all other network structures degraded our results or didn't improve them. Whether a deep network is needed or just a linear network suffices depends on the target domain. We attempted to use the linear combination of basis functions [19, 178], but this approach did not achieve satisfactory results.

4.3.5 Results of Identifying Task-Switching Stimuli

To evaluate our stimuli identification algorithm, we ran multiple simulations of the task selection policy for 2 tasks (a delivery task and a trash cleaning, or a delivery task and an HRI task) with random initial values for the state variables. The sensory state variables (features) of the trash

feature	*present*	x	y
mean $\pm$ std %	69 ± 31	27 ± 29	4 ± 8

Table 4.1: Feature importances for the trash cleaning task.

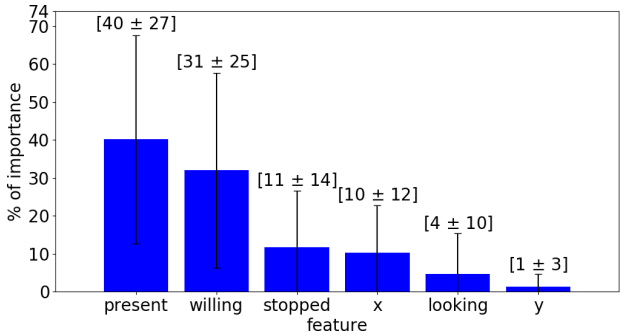


Figure 4.6: Feature importances for the HRI task.

cleaning and HRI tasks are shown in Table 4.1 and Fig. 4.6. The algorithm randomly selects the value of the *willing*, *looking*, and *stopped* variables. The value of the *present* variable is initially set to 0, and it becomes 1 when the robot is 5 steps away from a human goal or a trash goal, *i.e.*, we assume that the sensor range is 5. We applied our proposed algorithm on our data and calculated the feature importances. Table 4.1 and Fig. 4.6 show the feature importances for the trash cleaning and HRI tasks respectively for 40 simulation runs. Feature importances for each task model sum to 100. We evaluated the performance of the extra-trees classifier by 5-fold cross-validation technique. The classifier's accuracy is 89% on the HRI and 76% on the trash cleaning task.

Table 4.1 shows that the *present* feature is more important than the x and y features, so the algorithm selects it as the stimulus for the trash cleaning task. The importance of the *present* feature is not close to 100% since the robot will only switch to another task if the switch is beneficial. We observed similar results for the HRI task. The *present* and *willing* features are the most important features, and their sum of importances (71%) is almost the same as the importance of the *present* feature in the trash cleaning task (69%). Although the *looking* and *stopped* features are involved in the termination conditions of the HRI task, the robot can execute actions to change their value, so they do not affect the task-switching behavior. The *present* feature is more important than the other features, so it is selected as the stimulus for the HRI task.

We decreased the sensor range from 5 to 3 and observed that the importance of the *present* and *willing* features is 46% and 24% respectively. We increased the sensor range from 5 to 8, and we observed that the importance of the *present* and *willing* features is 26% and 43% respectively. As the sensor range increases, the *present* feature becomes less important since the robot can see the person from most places, and the *present* feature does not significantly affect the task-switching

behavior. However, if we decrease the sensor range, the *present* feature becomes more important for the task-switching behavior.

Discussion We tested different ensemble approaches (averaging and boosting) on our problem. In summary, averaging methods, which use a set of strong classifiers, such as extra-trees and random forest performed quite well on our dataset. However, boosting methods, which use a set of weak classifiers, such as gradient boosting and adaptive boosting (AdaBoost) performed poorly.

We select one feature from each task to serve as the stimulus to interrupt the task execution; however, an information theory-based approach that can select a subset of features that are the most informative while also considering their computation cost would be required in order to apply our approach to tasks with a lot more features. Our current approach is not efficient for a continuous stimulus if its value changes constantly, *e.g.*, constant changes in battery level. In tasks with multiple features as the stimuli or in presence of continuous features, we can train a classifier that detects when the stimuli get triggered.

4.4 Application on the Restaurant Domain

Consider the restaurant formalization from Chapter 3. If we were to apply the approach in this chapter on the restaurant setting:

- The state of the tables (or tasks) would need to be fully observable. This would mean that rather than keeping track of the internal state of the customers, *e.g.*, their satisfaction level, the robot would only consider their external observable state. The interaction between the customers on a table and the robot is more long-term compared to the interaction between a service robot and a person asking for directions in an office building, thus the customers' satisfaction level in a restaurant is not just limited to their current external state and depends on the history of actions performed for them and the observations received as a response.
- The robot would focus on the current table and only switch to another table when a signal is triggered. The restaurant domain is very dynamic and the requests are added and removed frequently, *e.g.*, the customers ask for more water or the customers ask questions about the menu. This would mean that the robot should solve the combined model frequently to decide what table it should attend to. Solving the combined model frequently hinders the benefits of our approach to expedite task execution, *i.e.*, solving the combined model less often.
- The robot could select one stimulus such as the *hand raise* or the *satisfaction level* as the signal to trigger the task-switching. In a restaurant setting, there are many other variables

such as *food ready to be served* and *customers have finished their food* that can be important for the task-switching behavior. Considering all these stimuli would results in processing most of the sensory inputs and hinders the benefits of our approach, *i.e.*, decreasing the amount of sensory input computations. If we only consider variables such as the *hand raise* or the *satisfaction level* for the task-switching, the robot would wait until the customers are unsatisfied or needy to address them. In a restaurant setting were the reward or tip depends on the customers' satisfaction, this strategy would be very ineffective.

Therefore, our approach of expediting task execution can be applied to the restaurant setting, but it will require simplifications that prevent it from being effective.

4.5 Conclusion and Discussion

We contribute a novel approach for switching among multiple task models. We explain how we leverage the stimuli to interrupt the robot's task execution and reevaluate the task selection policy. We evaluate our algorithms in a scenario with 1 to 6 service tasks and show that our approach requires less sensory computations compared to the combined model approach. Our method drastically improves task execution compared to the combined model method and is effective in domains where responding reactively to the environment is sufficient; however, it does not extend to domains with more complex structure such as the restaurant setting where the state of the tasks are partially observable, and all the tasks are required to be accomplished by the robot.

This approach does not extend to domains with more complex structure for the following reasons. First, this approach is not suitable for the domains where the robot has to accomplish all the tasks. The approach only processes the stimuli (*i.e.*, a subset of state variables) to decide when to solve the large combined planning model. If achieving the external tasks was necessary, the robot would need to process all the variables associated with all the tasks to ensure that it will not miss a feasible task. Second, for a domain such as the restaurant domain, the robot has to switch between the tasks frequently. In the restaurant domain, the robot cannot stick to one task and only switch to another task when triggered; the robot should decide what table should be attended to at each time step. Hence, the robot should build and solve the large model with all the tasks at each time step. This will be impractical for a restaurant with a large number of tasks. Third, we used a representation in this work that does not generalize to tasks with partially observability. Although less often, our approach still solves the large planning model with all the tasks when the stimuli are triggered; solving the large combined planning model is infeasible for tasks with partially observable states. For these more complex problems, in Chapters 5 and 6 we focus on providing algorithms that solve the large combined model faster rather than less frequently.

Chapter 5

Optimal Short-Horizon Planning for Achieving Multiple Independent POMDPs

5.1 Motivation

Many robotics applications, such as waiting tables in a restaurant and robots in search and rescue, involve a robot acting in a stochastic environment under partial observability while completing multiple independent tasks. Although in such domains the dynamics of each task is independent of one another, they all share a single robot, and an optimal policy for the robot should consider all tasks at each step. Many of these multi-task domains can be very dynamic with requests being added and removed at each time step, *e.g.*, a customer leaves or a table wants to order an extra dish. Thus, the robot only needs to plan for a short horizon of actions as any long-term plan quickly becomes sub-optimal or even infeasible after a few time steps.

A conventional planning approach for such domains is to combine all the tasks' states and actions into one large model and compute the h-step optimal policy for the combined model at each time step. However, this approach is impractical if the number of tasks are large. We leverage the structure of the problem, namely the independence between the tasks, and the observation that given a short horizon only a subset of tasks can be accomplished within this horizon to expedite planning and prove the optimality of our approach.

We utilize Partially Observable Markov Decision Process (POMDP) representation. POMDP is a powerful mathematical tool to model the robot's sequential decision making under uncertainty [36]. However, planning algorithms for POMDPs can only handle small state and action spaces and consequently do not scale well as the number of tasks increase. The combined model of the robot and all tasks grow as the number of tasks grow.

Many works have proposed approaches to speed up POMDP solvers by using point-based

methods [169], hierarchical planning [181], clustering and compression of belief space [160, 175], factored representation [168], and online POMDP approaches [158]. We leverage online POMDP approaches which only compute the optimal policy for the current information state and a small horizon of contingency plans. We are interested in domains in which a robot has to attend to multiple independent tasks whereas the above approaches do not make the independency assumption and address the combined model directly.

Our algorithm decomposes the problem into a series of smaller planning problems. In particular, a robot attending to a single task can be represented as a standalone smaller POMDP. We show how to compute lower and upper-bounds on the cost of an optimal solution involving N tasks. Using these insights, we develop an algorithm that searches over possible subsets of N tasks, solving each optimally until a provably globally optimal solution is found. We test our approach on a simplified restaurant environment in simulation. We present how we model the waiting table task as a robot planning problem and show the effectiveness of our approach compared to the combined model, a hierarchical POMDP approach, and a related paper [166].

In the next sections, we describe our notation, provide a pseudo code for the algorithm, and prove the optimality of the approach. We then discuss the performance of the algorithm compared to the existing approaches.

5.2 Problem Formulation

We focus on domains where one robot is addressing a set of N independent tasks. At each time step the robot decides what action should be executed with respect to which task. We model each task, the robot's state and the actions that can be applied to the task as a POMDP and call it *client POMDP*. In this section, we first explain how we represent the client POMDP. We then discuss how the N *client POMDPs* are combined into one large POMDP model called an *agent POMDP*. Finally, we discuss how we use the independence property among the N client POMDPs to compute the agent POMDP's solution.

5.2.1 Client POMDP

The client POMDP for task i is represented as a tuple $(S_i, A_i, Z_i, T_i, O_i, R_i, \gamma, H)$, where $S_i = SR \times SC_i$ denotes the state space, $sr \in SR$ denotes the robot's state (it is shared between the client POMDPs), and $sc_i \in SC_i$ represents the other state variables that are specific to task i. For example, in our restaurant domain, sr contains the robot's position, and sc_i contains state variables such as level of satisfaction. A_i denotes the robot's action space which contains a special

action called *no operation* (*no op*). Z_i denotes the robot's observation space which assumes there is no partial observability on the robot's state and only contains task i's observation space ZC_i, $Z_i = ZC_i$. For example, zc_i contains table i's neediness. The robot takes an action $a \in A_i$ and transitions from a state $s \in S_i$ to $s' \in S_i$ with probability $T_i(s, a, s') = \Pr(sc_i'|sc_i, a) \Pr(sr'|sr, a)$ where $s = (sr, sc_i)$ and $s' = (sr', sc_i')$. The robot makes an observation $z \in Z_i$, and receives a reward equal to $R_i(sr, sc_i, a)$. The probability function $O_i(s', a, z) = \Pr(z|sc_i', a)$ models noisy sensor observations. The discount factor γ specifies how much immediate reward is favored over more distant reward, and H denotes the robot's horizon. The planning horizon is the number of time steps a robot will look into the future when coming up with a plan.

The robot's objective is to choose actions at each time step to maximize its expected future discounted reward: $\mathbb{E}\left[\sum_{t=0}^{H} \gamma^t r_{i,t}\right]$, where $r_{i,t}$ is the reward gained at time t from POMDP i. In POMDP planning, the robot keeps a probability distribution over the states S_i, which is called a belief state B_i. POMDP planning searches for a policy $\pi : B_i \rightarrow A$ that maximize the expected future discounted reward at each belief $b \in B_i$. After executing an action, the robot's belief is updated by Eq. 5.1, where $\Pr(z|b_i, a)$, the probability of observing z after doing action a in belief b_i, is a normalizing constant.

$$b_i(s') = \frac{O_i(z|s', a) \sum\limits_{s \in S_i} T_i(s'|s, a) b_i(s)}{\Pr(z|b_i, a)} \tag{5.1}$$

The optimal return at stage t, $V_{i,t}^*(b)$, can be iteratively computed by Eq. 5.2.

$$\begin{aligned} Q_{i,t}^*(b_i, a) &= \sum_{s \in S_i} b_i(s) R_i(s, a) + \gamma \sum_{z \in Z_i} \Pr(z|b_i, a) V_{i,t-1}^*(b_{i,z}^a) \\ V_{i,t}^*(b_i) &= \max_{a \in A_i} \left[Q_{i,t}^*(b_i, a) \right] \end{aligned} \tag{5.2}$$

The value of following a deterministic trajectory τ at belief state b_i and continuing according to the rest of τ for the remaining $t-1$ steps is computed by Eq. 5.3.

$$V_{i,t}^{\tau}(b_i) = \sum_{s \in S_i} b_i(s) R_i(s, \tau_t) + \gamma \sum_{z \in Z_i} \Pr(z|b_i, \tau_t) V_{i,t-1}^{\tau}(b_{i,z}^{\tau_t}) \tag{5.3}$$

For the *no op* action, the reward function only depends on the state variables that are specific to the client POMDP $R_i(sr, sc_i, \text{no op}) = R_i(sc_i, \text{no op})$.

5.2.2 Agent POMDP

We call a POMDP created from multiple client POMDPs *agent POMDP* (or *robot POMDP*). Formally, the agent POMDP for a domain with N tasks is represented by $(N, S, A, Z, T, O, R, \gamma, H)$ where $S = SR \times SC_1 \times SC_2 \times \ldots \times SC_N$, $s \in S$ represents the agent POMDP's state. Let $P = \{i \in \mathbb{N} : i \leq N\}$. The robot's action set A (Eq. 5.4) contains vectors of length N in which except one element, all other elements are *no op*. The observation space is $Z = Z_1 \times Z_2 \times \ldots \times Z_N$. The robot's probability distribution over the states S is $b \in B$ where $B = B_1 \times B_2 \times \ldots \times B_N$. We assume that the agent POMDP's reward function is additive in terms of its underlying tasks $\mathbb{E}\left[\sum_{i=1}^{N}\sum_{t=0}^{H}\gamma^t r_{i,t}\right]$, where $r_{i,t}$ is the reward gained at time t from task i.

$$A = \bigcup_{i \in P}\bigcup_{a \in A_i} \overbrace{[\text{no op...no op}, \underbrace{a}_{i\text{th element}}, \text{no op...no op}]}^{\text{length N}} \tag{5.4}$$

The properties in Def. 1 should hold for a set of N client POMDPs to be independent. The N client POMDPs can only share robot's state space and the *no op* action. The transition and observation functions of different client POMDP models are independent of one another.

Definition 1 We call a set of N client POMDPs independent iff $\forall i, j \in P$, $a \in A_i$ and $i \neq j$, the following holds:

1. $SC_i \cap SC_j = \emptyset$
2. $Z_i \cap Z_j = \emptyset$
3. $(A_i \setminus \{\textit{no op}\}) \cap (A_j \setminus \{\textit{no op}\}) = \emptyset$
4. $\Pr(sc'_i | sc_1, sc_2, \ldots, sc_N, a) = \Pr(sc'_i | sc_i, a)$
5. $\Pr(z_i | sc'_1, sc'_2, \ldots, sc'_N, a) = \Pr(z_i | sc'_i, a)$

Given that the tasks are independent and the reward is additive, $R(s, a) = \sum_{i \in P} R_i(s_i, a[i])$, the optimal return at stage t, $V_t^*(b)$, can be iteratively computed as follows:

$$\Pr(z|b, a) = \prod_{k \in P} \Pr(z_k | b_k, a[k]) \tag{5.5}$$

$$V_t^*(b) = \max_{a \in A} \Big[\sum_{i \in P} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) + \gamma \sum_{z \in Z} \Pr(z|b,a) V_{t-1}^*(b_z^a) \Big] \tag{5.6}$$

To compute the value of the current belief state of the robot for a fixed horizon H, the planner starts with the robot's current belief as the root of a tree and builds the tree of all reachable beliefs by considering all possible actions and observations. For each action and observation, a new belief node is added to the tree as a child node of its immediate previous belief. To solve the agent POMDP, a combined belief tree of all tasks is built till horizon H. The value of the robot's current belief is computed by propagating value estimates up from the fringe nodes, to their ancestors, all the way to the root, according to Eq. 5.6. We call this approach *agent POMDP with a fixed horizon* or *agent-POMDP-FH*.

Note that the agent POMDP approach is impractical if the number of tasks are large. If each client POMDP has $|S|$ states and $|A|$ actions (excluding the *no op* action), and there are N tables in the restaurant, the robot should plan over an agent POMDP with $|S|^N$ states and $N \times |A| + 1$ actions which is infeasible.

Definition 2 We introduce a parameter k^* which represents the maximum number of tasks that the robot can potentially attend to within H steps. For example, k^* can be set to $\lceil \frac{H}{l} \rceil$ where $\forall i, j \in P$ and $i \neq j$, l is the minimum number of time steps that the robot takes to transition from task i to task j and affect the task j's state variables sc_j.

Considering the restaurant setting in Fig. 5.6, within $H = 4$, the robot can only attend to 2 tables as it should navigate to one table and serve it (*e.g.*, give the menu to $T2$), and then navigate to another table and attend to it (*e.g.*, take the cash from $T0$).

In the next section, we provide a pseudocode for the algorithm by assuming an arbitrary k. We then show that our approach is optimal for $k \geq k^*$ and discuss its performance with different values for k compared to the other methods.

5.3 Approach

We exploit the observation that in some domains the number of tasks that the robot can accomplish within h-steps is limited. We present an algorithm that exploits this observation and decomposes the problem into a series of much smaller planning problems, the solution to which gives us an optimal solution. We denote a subset of k tasks out of P as tpl or k-tuple, $tpl \subseteq P$. We

refer to a set including all combinations of k out of N tasks, $\binom{N}{k}$ tasks, as $tpls$ or k-tuples $tpls = \{tpl \in \mathcal{P}(P) : |tpl| = k\}$. The issue is the planner does not know apriori which k tasks it should consider.

Algorithm 3: Online Planner with Fixed Horizon

```
1 MultiTaskFixedHorizonPlanner (env, P, H, k)
2     while not AllTasksDone() do
3         a ← SelectAction(P, H, k)
4         observations ← Step(env, a)
5         UpdateBeliefs(P,observations)
```

5.3.1 Proposed Method

Alg. 3 provides a pseudo code of the main loop of our approach. We follow the online POMDP planning framework where the planning and execution steps are interleaved until all the tasks are terminated (line 2). P represents a set of POMDPs. During the planning phase, the algorithm computes the best action to execute given the POMDPs' belief state (line 3). The execution step executes the selected action (line 4) and updates the belief state of the POMDPs based on the obtained observation (line 5). The robot replans after each action execution. All the baseline algorithms that we compare against modify *SelectAction* in some way.

Overview of the algorithm (Alg. 4)

We first solve each client POMDP separately (lines 4-5). We use the solutions of the client POMDPs to compute a lower-bound on the optimal value of the agent POMDP with N tasks (Function $LowerBound$, line 6). We consider a set with all possible combinations of k tasks (k-tuples). We use the solutions of the client POMDPs to compute an upper-bound on the value of an agent POMDP created from a k-tuple and use the lower-bound to remove the ineffective candidate k-tuples (Function $BestKTuples$). For each remaining candidate k-tuple (line 8), we build and solve the agent POMDP model created from the k-tuple optimally to get an action and the Q-value associated with it (line 9). The action from the k-tuple with the maximum Q-value is selected (lines 10-11) by the robot.

Compute lower-bound

To compute a lower-bound on the optimal value of the agent POMDP with N tasks (Function $LowerBound$), the robot only considers one client POMDP in its horizon H and will perform *no*

op on the other POMDPs. For a client POMDP p out of N POMDPs, the algorithm sums up the optimal V-value of the pth POMDP (V_p^*) and the value of performing *no op* on the other POMDPs ($\sum V_q^\tau$, line 21). The sum with the maximum value is returned. This calculation provides a lower-bound on the value of the agent POMDP since it does not take into account that 1) the optimal policy might involve switching from one task to another, or 2) an action other than a table's optimal action might be optimal in the agent POMDP.

Algorithm 4: Short-horizon Multi-task POMDP Planner

1 **SelectAction** (P,H,k)
2 $\quad\tau \leftarrow$ array $[1..H]$ filled with no op; $Q_{best} \leftarrow -\infty$
3 $\quad V^*$,V^τ, $Q^* \leftarrow$ empty array $[1..|P|]$
4 $\quad$ **for** $p \in P$ **do**
5 $\quad\quad V^*[p]$,$V^\tau[p]$,$Q^*[p] \leftarrow$ SolvePomdp(p,H)
6 $\quad LB \leftarrow$ LowerBound(V^*,V^τ)
7 $\quad tpls \leftarrow$ BestKTuples(P,V^τ,Q^*,LB,k)
8 $\quad$ **for** $tpl \in tpls$ **do**
9 $\quad\quad a_{max}$, $Q_{max} \leftarrow$ SolveAgentPOMDP(tpl,H)
10 $\quad\quad$ **if** $Q_{max} > Q_{best}$ **then**
11 $\quad\quad\quad a_{best} \leftarrow a_{max}$; $Q_{best} \leftarrow Q_{max}$
12 $\quad$ **return** a_{best}
13 **BestKTuples** (P, V^τ, Q^*, LB, k)
14 $\quad tpls \leftarrow$ a set with all combinations of k out of P
15 $\quad$ **for** $tpl \in tpls$ **do**
16 $\quad\quad UB_{tpl} \leftarrow \max_{a \in A_{tpl}} \left(\sum_{p \in tpl} Q^*[p, a[p]]\right) + \sum_{q \in P \setminus tpl} V^\tau[q]$
17 $\quad\quad$ **if** $UB_{tpl} < LB$ **then**
18 $\quad\quad\quad$ remove tpl from $tpls$
19 $\quad$ **return** $tpls$
20 **LowerBound** (V^*, V^τ)
21 $\quad$ **return** $\max_{p \in P}(V^*[p] + \sum_{q \in P \setminus \{p\}} V^\tau[q])$

Find best k-tuples

To remove the ineffective POMDP tuples, we compute an upper-bound on the value of each k-tuple. We start with all $\binom{N}{k}$ k-tuples (line 14). For each k-tuple (line 15) if its computed upper-bound UB_{tpl} is lower than the lower-bound (line 17), we remove it from the candidate k-tuples set (line 18).

To compute a k-tuple's upper-bound, we assume that the robot only considers the selected k

tasks and performs *no op* on the other POMDPs ($\sum V_q^\tau$, line 16). For the selected k-tuple, the robot executes one of the actions from the k-tuples' set of valid actions A_{tpl} which only considers the actions associated with the POMDPs in tpl (Eq. 5.7). We assume that after executing the first action, each of the k POMDPs follow their optimal policies $Q_p^*(b, a[p])$. This breaks the assumption that the robot cannot address all the tasks in parallel and gives an upper-bound on the value of the k-tuple.

5.3.2 Optimality Proofs

Given Def. 1 and an assumption that we define later in this section, we prove that Alg. 4 computes an optimal solution for the agent POMDP with N tasks.

Some notation:

- $V_{p,t}^*$: the optimal value of the client POMDP p at time t.
- $V_{P,t}^*$: the optimal value of the agent POMDP created from the N tasks at time t (Eq. 5.6).
- A_{tpl}: only considers the actions associated with the POMDPs in a given k-tuple and performs *no op* on the other POMDPs, Eq. 5.7. In this equation, the union is only over $tpl \subseteq P$, so $A_{tpl} \subseteq A$.

$$A_{tpl} = \bigcup_{i \in tpl} \bigcup_{a \in A_i} \overbrace{[\text{no op...no op}, \underbrace{a}_{i\text{th element}}, \text{no op...no op}]}^{\text{length N}} \tag{5.7}$$

- $V_{tpl,t}^*$: the optimal value of the agent POMDP created from only the client POMDPs in tpl at time t.

$$\begin{aligned} V_{tpl,t}^*(b_{tpl}) &= \max_{a \in A_{tpl}} Q_{tpl,t}^*(b_{tpl}, a) \\ &= \max_{a \in A_{tpl}} \Big[\overbrace{\sum_{i \in tpl} \sum_{s \in S_i} b_i(s) R_i(s, a[i])}^{\text{immediate reward}} + \\ &\qquad \gamma \sum_{z \in Z_{tpl}} \Pr(z | b_{tpl}, a) V_{tpl,t-1}^*(b_{tpl,z}^a) \Big] \end{aligned} \tag{5.8}$$

- $U_{tpl,t}^*$: the optimal value of the agent POMDP created from the client POMDPs in P at time t with action set A_{tpl}. Intuitively, $V_{tpl,t}^*$ considers the value of the client POMDPs in tpl, but

$U^*_{tpl,t}$ also considers the utility of performing *no op* on the POMDPs that are not in tpl.

$$U^*_{tpl,t}(b) = \max_{a \in A_{tpl}} \Big[\sum_{i \in P} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) + \gamma \sum_{z \in Z} \Pr(z|b,a) U^*_{tpl,t-1}(b^a_z) \Big] \tag{5.9}$$

Assumption 1 *The robot has a short horizon H and can only consider k^* tasks in its horizon (Def. 2). Under this assumption, the optimal value for the agent POMDP is called $\hat{V}^*_P$.*

As mentioned earlier, $U^*_{tpl,t}$ considers the action sets of all the POMDPs that are in tpl and performs *no op* on the POMDPs that are not in tpl (Eq. 5.9). Given Def. 1 and Ass. 1, to compute $\hat{V}^*_{P,t}$, the robot can take a maximum over $U^*_{tpl,t}$ where $|tpl| = k, k \geq k^*$ for all possible $tpl \in tpls$.

$$\hat{V}^*_{P,t}(b) = \max_{tpl \in tpls} U^*_{tpl,t}(b) \tag{5.10}$$

Lemma 1 *Eq. 5.11 provides a lower-bound on the value of the agent POMDP created from set P.*

$$\max_{p \in P} (V^*_{p,t}(b_p) + \sum_{q \in P \setminus \{p\}} V^\tau_{q,t}(b_q)) \leq \hat{V}^*_{P,t}(b) \tag{5.11}$$

Proof: We compute $U^*_{\{p\},t}$ (or $U^*_{p,t}$) by using Eq. 5.9 where the robot only considers POMDP p for horizon H, $tpl = \{p\}$, and performs *no op* on the other POMDPs over that horizon. In Eq. 5.9, the maximum is taken over A_p, whereas in Eq. 5.6, the maximum is taken over A. We know $A_p \subseteq A$ as it follows from Eq. 5.7, thus $U^*_{p,t}$ is a lower-bound on $\hat{V}^*_{P,t}$, and Eq. 5.12 holds. We will show that $U^*_{p,t}$ is in fact $V^*_{p,t}(b_p) + \sum_{q \in P \setminus \{p\}} V^\tau_{q,t}(b_q)$.

$$\forall p \in P : U^*_{p,t}(b) \leq \hat{V}^*_{P,t}(b) \Rightarrow (\max_{p \in P} U^*_{p,t}(b)) \leq \hat{V}^*_{P,t}(b) \tag{5.12}$$

Using Eq. 5.5, we expand Eq. 5.9 as follows:

$$\begin{aligned} U^*_{p,t}(b) = \max_{a \in A_p} \Big[& \sum_{i \in P} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) \\ & + \gamma \underbrace{\sum_{z_1 \in Z_1} \Pr(z_1 | b_1, \text{no op}) \ldots \sum_{z_N \in Z_N} \Pr(z_N | b_N, \text{no op})}_{\text{does not include } \sum_{z_p \in Z_p}} \\ & \sum_{z_p \in Z_P} \Pr(z_p | b_p, a[p]) U^*_{p,t-1}(b^a_z) \Big] \end{aligned} \tag{5.13}$$

The proof goes by mathematical induction. If $H = 1$ and assuming that $U^*_{p,0}(b) = 0$, the following equation holds.

$$U^*_{p,t}(b) = \max_{a \in A_p} \Big[\sum_{i \in P} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) \Big] =$$
$$V^\tau_{1,1}(b_1) + \ldots + V^*_{p,1}(b_p) + \ldots + V^\tau_{N,1}(b_N)$$

If $H = t - 1$, we assume Eq. 5.14 where τ is a trajectory consisting of only *no op*s, $\tau =$ no op$[1..H]$, and show that Eq. 5.14 also holds for $H = t$. Intuitively, Eq. 5.14 holds since the reward is additive, the POMDPs are independent, and the *no op* actions are executed in parallel while the robot addresses POMDP p.

$$U^*_{p,t-1}(b) = V^\tau_{1,t-1}(b_1) + \ldots + V^*_{p,t-1}(b_p) + \ldots + V^\tau_{N,t-1}(b_N) \tag{5.14}$$

We substitute Eq. 5.14 in Eq. 5.13. Given Def. 1, for a specific Z_i, we can marginalize out the sum over Z_js ($j \neq i$) to obtain:

$$\begin{aligned} U^*_{p,t}(b) &= V^*_{p,t}(b_p) + \underbrace{V^\tau_{1,t}(b_1) + \ldots + V^\tau_{N,t}(b_N)}_{\text{does not include } V_p} \\ &= V^*_{p,t}(b_p) + \sum_{q \in P \setminus \{p\}} V^\tau_{q,t}(b_q) \end{aligned} \tag{5.15}$$

Thus, Eq. 5.15 holds for every $H = t$.

Lemma 2 *Under Ass. 1, the optimal value of the agent POMDP created from the set P can be computed by Eq. 5.16.*

$$\begin{aligned}\hat{V}^*_{P,t}(b) &= \max_{tpl \in tpls} U^*_{tpl,t}(b) \\ &= \max_{tpl \in tpls} (V^*_{tpl,t}(b_{tpl}) + \sum_{q \in P \setminus tpl} V^{\tau}_{q,t}(b_q))\end{aligned} \tag{5.16}$$

Proof: If we show Eq. 5.17 holds, under Ass. 1, Eq. 5.16 follows from Eq. 5.10 and Eq. 5.17.

$$U^*_{tpl,t}(b) = V^*_{tpl,t}(b_{tpl}) + \sum_{q \in P \setminus tpl} V^{\tau}_{q,t}(b_q) \tag{5.17}$$

We consider an agent POMDP with the set of tasks tpl and call it P_{tpl}. We then build a new set of client POMDPs as follows: $\mathcal{A} = \{P_{tpl}\} \cup \{q | q \in P \setminus tpl\}$. Since all the members of P follow Def. 1, the POMDPs in the set $\mathcal{A}$ also follow Def. 1 and are independent; thus, Eq. 5.17 follows from Eq. 5.15.

Lemma 3 *For a given tpl, Eq. 5.18 provides an upper-bound on the value of $U^*_{tpl,t}$.*

$$U^*_{tpl,t}(b) \leq \max_{a \in A_{tpl}} (\sum_{p \in tpl} Q^*_{p,t}(b_p, a[p])) + \sum_{q \in P \setminus tpl} V^{\tau}_{q,t}(b_q) \tag{5.18}$$

Proof: Substituting U^*_{tpl} from Eq. 5.17 in Eq. 5.18:

$$V^*_{tpl,t}(b_{tpl}) \leq \max_{a \in A_{tpl}} (\sum_{p \in tpl} Q^*_{p,t}(b_p, a[p]]) \tag{5.19}$$

Thus, we only need to show that Eq. 5.19 holds. We use Eq. 5.2 to compute the Q^* of POMDP p and take a sum of the Q^*s over all members of tpl:

$$\begin{aligned}\sum_{p \in tpl} Q^*_{p,t}(b_p, a[p]) = &\overbrace{\sum_{p \in tpl} \sum_{s \in S_p} b_p(s) R_p(s, a[p])}^{\text{immediate reward } (IR)} \\ &+ \gamma \sum_{p \in tpl} \sum_{z \in Z_p} \Pr(z | b_p, a[p]) V^*_{p,t-1}(b^{a[p]}_{p,z})\end{aligned} \tag{5.20}$$

The immediate reward IR operand in Eq. 5.20 and Eq. 5.8 are equal, so we only need to show the inequality for the second operand. For time step $t-1$, if the robot could address all the k

tasks in parallel, we could compute an upper-bound on the value of $V^*_{tpl,t-1}(b^a_{tpl,z})$ by summing over the optimal value of each POMDP in tpl, $(m, n, o, \ldots) \in tpl$, Eq. 5.21. This can be proved using a similar induction procedure that we used earlier.

$$\begin{aligned} V^*_{tpl,t-1}(b^a_{tpl,z}) \leq V^*_{m,t-1}(b^{a[m]}_{m,z_m}) + \ldots + V^*_{n,t-1}(b^{a[n]}_{n,z_n}) + \\ \ldots + V^*_{o,t-1}(b^{a[o]}_{o,z_o}) = \sum_{q \in tpl} V^*_{q,t-1}(b^{a[q]}_{q,z_q}) \end{aligned} \tag{5.21}$$

Substituting Eq. 5.21 in Eq. 5.8:

$$\begin{aligned} Q^*_{tpl,t}(b_{tpl}, a) &\leq IR + \sum_{p \in tpl} \sum_{z \in Z_p} \Pr(z|b_p, a[p]) V^*_{p,t-1}(b^{a[p]}_{p,z}) \\ &= \sum_{p \in tpl} Q^*_{p,t}(b_p, a[p]) \end{aligned} \tag{5.22}$$

Thus, $V^*_{tpl,t}(b_{tpl}) \leq \max_{a \in A_{tpl}} \sum_{p \in tpl} Q^*_{p,t}(b_p, a[p])$.

Therefore, Alg. 4 computes an optimal solution for the agent POMDP created from set P. The proof should follow from Lemma 1, 2 and 3.

5.4 Experiments

We call our approach *method-k*, *method-2 (A)* or *method-3 (F)*, as we evaluate it for different values for k (2 or 3). We compare our method against the following baselines.

- **Agent POMDP (B)**: We use the agent POMDP model that we described in the approach section.
- **N-samples-k (C,G)**: We use the approach by [166] where they select N k-tuples from all possible combinations of k-tuples and solve them optimally. They iterate over the set P; for each task they randomly select $k-1$ additional tasks from set P. We refer to the *N-samples-2* method as *C* and the *N-samples-3* method as *G*.
- **Hierarchical POMDP or HPOMDP (D)**: We represent each task as a macro action; in total there are N macro actions for all the N tasks. We use the agent POMDP model and replace its action set with the set of macro actions. During planning when a macro action i, m_i, gets selected, the agent POMDP evolves according to the POMDP i's action set, while the other POMDPs evolve as *no op* has been executed on them. Each macro action takes $min(\text{horizon}, \#\text{ time steps left till } m_i \text{ terminates})$ time steps to get executed and is atomic.

- **Greedy (E)**: The robot solves each client POMDP separately and selects an action according to Eq. 5.23. This approach assumes that after the first action execution, for the remaining horizon, the tasks can be executed at the same time in parallel.

$$\arg\max_{a \in A}(\sum_{p \in P} Q_p^*(b_p, a[p])) \tag{5.23}$$

5.4.1 Restaurant Model

We run experiments in the restaurant scenarios with 2 to 12 tables. We used the POMDP models from Chapter 3 to run the experiments. Given this model description, if $horizon \leq 4$, the robot can only address two tasks in its 4-step limited horizon, so our algorithm computes an optimal solution if it considers all the 2-tuples. If $horizon = 5$, the robot can address 3 tasks, so the algorithm should consider all the 3-tuples.

5.4.2 Results

For each algorithm, we run 30 episodes each for 20 actions. Each episode starts with a random initialization of the state variables with its belief probability set to 1. The random initialization for each episode is the same across all algorithms.

Quantitative Results We compare our method in terms of planning time and average reward against the baselines. For each episode, we compute the average time that the planner takes to plan over 20 actions. The reward for each action execution is the expected reward over the belief state distribution $r(b, a) = \sum_{s \in S} b(s)R(s, a)$. We report the average reward over 20 actions. To remove the variations that results from the initial randomization, for each episode we take the difference between the average reward of *method-2* and other approaches.

Fig. 5.1 shows a comparison of the mean and standard deviation of the planning time for *method-2* against the baselines. We could only run the *agent POMDP* for 30 episodes up to a certain number of tables ≤ 6 (shown by the orange label on the x-axis). Beyond that we run the *agent POMDP* approach for one episode and report the results in a table on each figure. As the number of tables increase, the *agent POMDP* approach has a higher positive slope than other approaches. The planning time for different approaches mostly follow $B > A > C > D > E$.

Fig. 5.2 shows the mean and standard deviation of the difference between the average reward of *method-2* and that of other approaches. We compare all the baselines against a zero line which represents our approach. We proved earlier that our approach is optimal, but an optimal action for horizon H might not result in higher average return for an episode because 1) an action

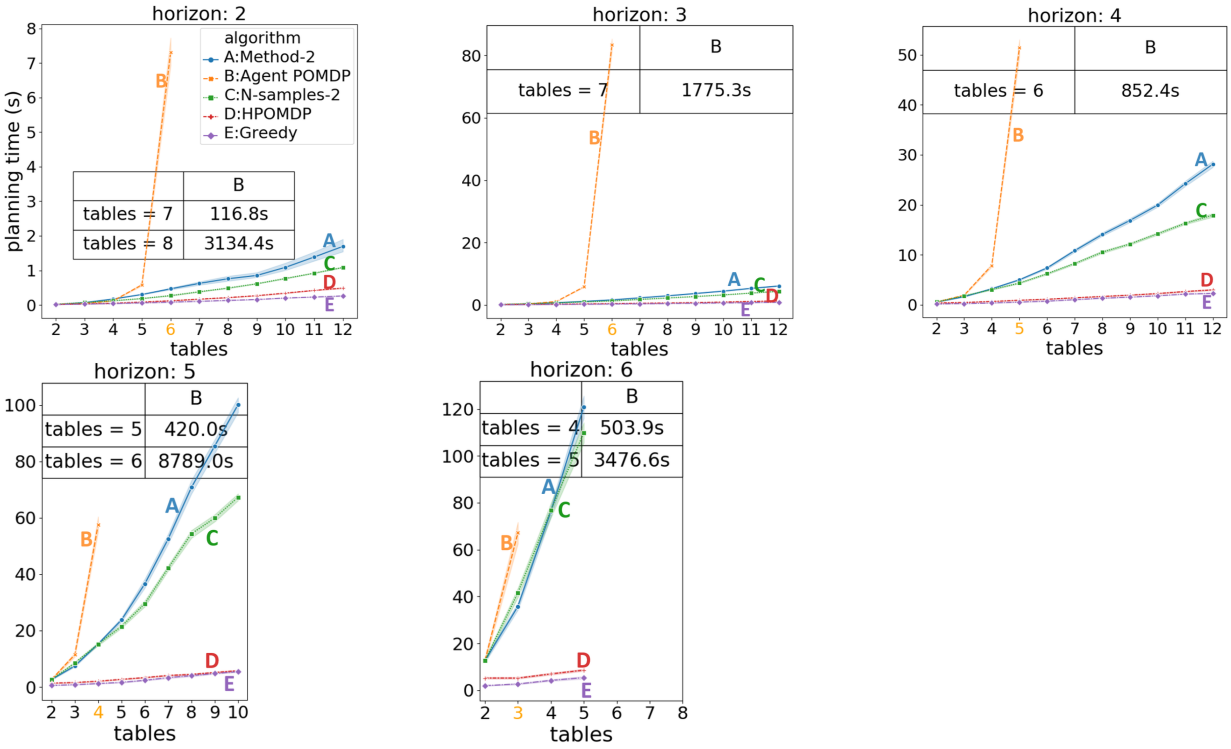


Figure 5.1: Planning times for different horizons and number of tables.

that is optimal for a short horizon might not be optimal for a longer horizon, and 2) different approaches have different tie-breaking strategies; *i.e.*, actions with equal average returns for a short horizon would have different average returns for a longer horizon. This is why for horizon 2 our approach performs exactly the same as the *agent POMDP* approach, but other methods can perform better. For the horizons other than 2, the average reward for different approaches mostly follow $B \approx A > C$.

We also compare the reward for each episode with the same initialization across multiple algorithms. We report the results in terms of the percentage of the episodes (out of 30) that the rewards are equal or one is better. For different horizons, our approach's reward is exactly the same as the *agent POMDP*'s reward for 2,4,5, and 6 tables. For 3 tables and horizons 3 and 6, the *agent POMDP* is better in 3% of the episodes (one episode) because of having a different tie-breaking strategy. For 5 tables and horizon 3, our approach is better in 3% of the episodes.

Comparing the reward of *method-2* against *N-samples-2* for each episode, we observe that for 2 and 3 tables and for different horizons, the rewards are exactly the same. For horizon 2, if $tables = 7$, the rewards are exactly the same in 87% of the episodes and *N-samples-2* is better in 10% of the episodes. For 11 tables, the rewards are exactly the same in 77% of the episodes and our approach performs better in 13% of the episodes. For a longer horizon 3, if

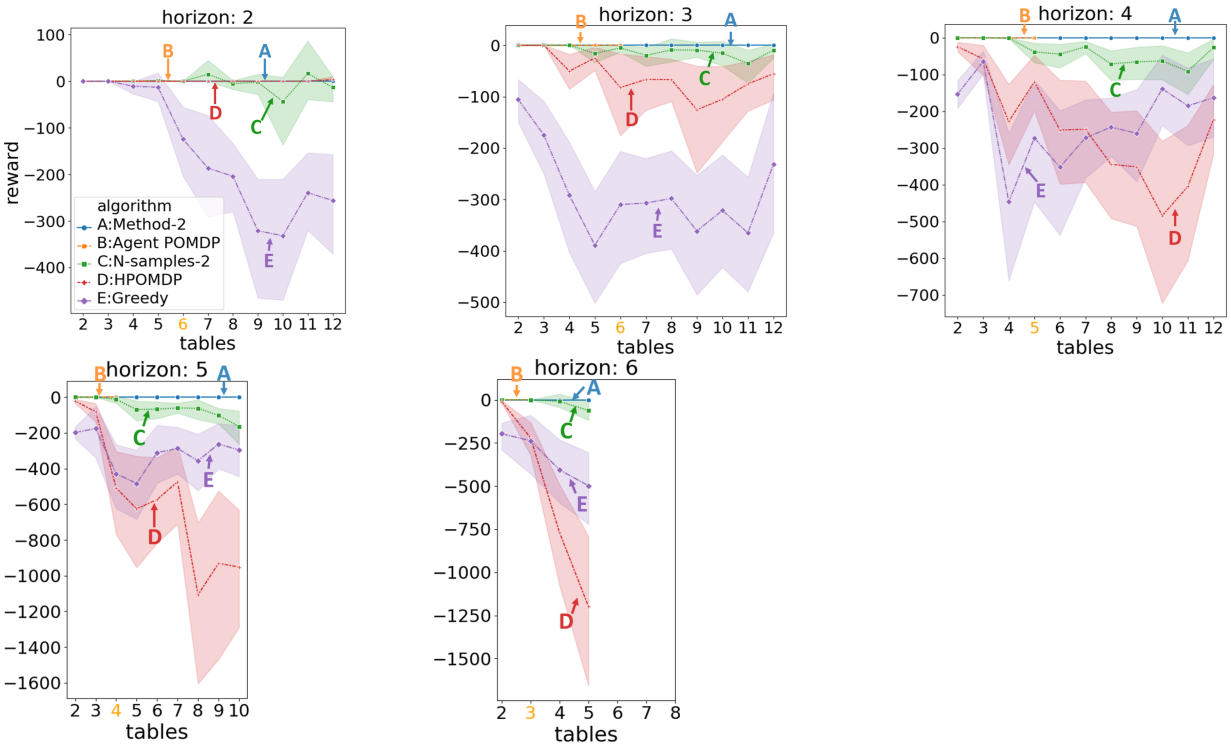


Figure 5.2: Difference between the average reward of our method and the other methods.

$tables = 8$, the rewards of *method-2* and *N-samples-2* are exactly the same 50% of the times and *method-2* is better 27% of the times. Thus, for horizon 2, our approach has a similar performance as *N-samples-2*. Our algorithm mostly performs better than *N-samples-2* for horizons ≥ 3. For horizon 2, our method does not benefit from considering 2-tuples, so the *HPOMDP* approach provides similar average reward as our algorithm. The *HPOMDP* approach performs better than our approach when horizon is 2 and $tables > 8$ in 3% of the episodes because of the tie-breaking strategy.

We run the algorithms on a simpler version of the restaurant that includes *satisfaction level*, *current request*, *hand raise*, and *time since request* as the human's state SC, and has all the actions except *food is not ready yet* and compare its performance with different k values against other methods. As can be seen in Fig. 5.3, given horizon 5, the planning time for different approaches mostly follow $B > F > G > A > C$. Comparing *method-2* and *method-3*, for horizon 5, if $tables = 4$, the rewards are exactly the same. If $tables = 6, 7$, the rewards are exactly the same in 67% of the cases and *method-3* is better in 23% of the episodes. If $tables = 8$, the rewards are exactly the same in 77% of the cases and *method-2* is better in 13% of the cases. Both *method-2* and *method-3* perform better that the other baselines.

In summary, we observe that our approach results in a similar average reward compared to

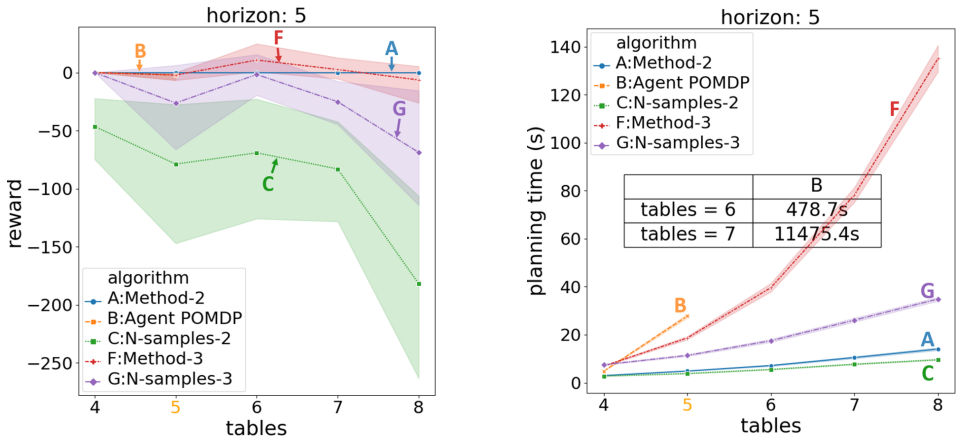


Figure 5.3: Performance comparison between our approach and other baselines when $k = 2, 3$. We run the algorithms on a simpler version of the restaurant model.

the *agent POMDP* approach while being significantly faster. Although our approach has a higher planning time compared to some of the baselines, it has a higher average reward than them.

Qualitative Results Fig. 5.4 shows a sample output policy for 5 tables. The histograms show the belief over satisfaction for different tables. The leftmost bar is 0 (very unsatisfied) and the rightmost is 5 (very satisfied). The robot's action is shown on top of each figure. Each table's request and the amount of time they have been waiting is shown above it. The leftmost figure shows the restaurant configuration at time step 11 after Table 0 is served. Table 3 has been waiting for 8 time steps and compared to others is less satisfied, so the robot services it to increase the table's satisfaction. The robot then goes to Table 4 since it soon becomes very unsatisfied. The robot then services Table 1 as their food is ready before going to Table 2 to update them that their food is not ready. The *greedy* approach selects the *no op* action at time step 11 as it assumes that after 1 time step, it can service all tasks in parallel. Two consecutive *go to* actions appear frequently in the output policy of the *greedy* approach, *e.g.*, *greedy* selects *go to T*4 at time step 12 instead of serving T3.

5.4.3 Further Analysis

Here, we provide a discussion on how much the effectiveness of our approach depends on the parameters of the model. In general if we apply the approach on a different planning problem of similar complexity, the planning time would still be similar to the planning time that we computed for the restaurant model, and the optimality guarantees of our approach would still hold. The *HPOMDP*, *N-samples-k*, and *greedy* approaches do not have any optimality guarantees. On a different application domain which requires switching among the tasks, we expect our method

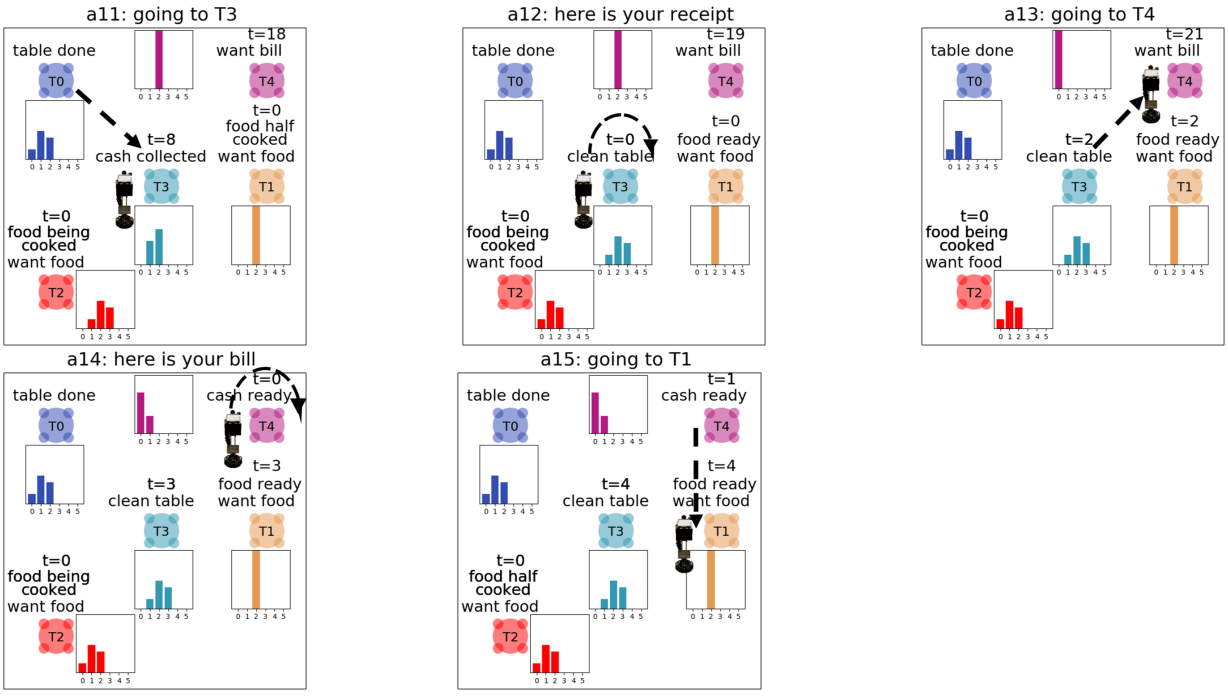


Figure 5.4: Example output policy for $H = 4$ and 5 tables.

to perform better in terms of average reward than the *HPOMDP* approach since the *HPOMDP* approach does not consider switching between the tasks in each planning step. We also expect our approach to perform better than *N-samples-k* since *N-samples-k* samples from the subset of tasks whereas our method finds an optimal solution by computing upper and lower optimal value bounds for subsets of tasks to prune the subsets.

The parameters of the reward function does not affect our approach's planning time and the optimality of our approach. However, the parameters will change the output policy of the robot, and how much better our approach is compared to the other baseline approaches. For example, if the negative reward for going from one table to another table is high, the robot would prefer to stay as much as it can at the current table, even if the other tables are very unsatisfied. This way of defining the reward function would make our approach just a bit better than the *HPOMDP* approach in terms of the average reward. The parameters are engineered such that we can get a sensible and interesting output policy as shown in Fig. 5.4 where the robot switches among multiple tables instead of just servicing one table mostly.

The *greedy* approach sometimes outperforms the *HPOMDP* approach since the domain requires the robot to switch between the tables to keep the customers satisfied. The *HPOMDP* approach assumes that only one table can be serviced in the given horizon. When this assumption is valid, for example for horizon 2, the *HPOMDP* approach performs much better than the *greedy*

approach. For longer horizons (and more tables), this assumption becomes less valid and the *HPOMDP* approach performs poorly. The *greedy* approach does not provide an accurate estimate, but can consider more than one task in the short horizon since it assumes that the tasks can be executed at the same time in parallel after the first action execution.

5.4.4 Robot Experiments

In this section, we illustrate an example run of the planner and the restaurant model that we discussed earlier on the robot. The exact details of the experiments are in Appendix A.

CoBot Robot

We have developed the CoBot robots (Fig. 5.5), which navigate autonomously in an office building performing tasks for users. The CoBot has an omnidirectional base. They have LIDAR and Kinect sensors to localize and detect obstacles, a touchscreen to interact with humans, and speakers for voice interactions. They can successfully deliver messages, transport items, escort people, and interact with humans in our indoor environment. To fulfill these tasks, CoBot localizes in the building using Episodic non-Markov localization [23]. It navigates on a graph of the environment, and autonomously avoids obstacles. Since CoBot does not have arms to manipulate the world with, it relies on human help for some of its capabilities, utilizing the idea of symbiotic autonomy, where robots and humans cooperate to accomplish tasks. CoBot asks humans to place items in its basket and to help it press elevator buttons [190]. CoBot has completed hundreds of tasks and travelled hundreds of kilometers [190]. To drive to a destination office, the robot first retrieves the (x, y, θ) coordinates of the location. Next, the robot needs to plan its route through the building.

The robot computes its path to the destination using a Navigation Graph [22]. The robot then follows the path computed, updating its position on the map using Episodic non-Markov localization [23]. As the robot drives to its destination, it may find obstacles on its path (e.g., people talking in the hallways). When the robot finds an obstacle on its path, the robot stops, and requests passage, saying "Please excuse me.". Upon arrival at its destination, the robot announces that it has completed its task and waits for a user to confirm the execution. Here are some of the core abilities of the CoBot:

- detects obstacles
- localizes and navigates in office buildings
- schedules and executes tasks
- speaks to human users and uses a user interface to get the users' input

Figure 5.5: CoBot mobile service robots.

We use the main functionalities on the robot and adapt them to our restaurant setting. Specifically, we use the obstacle detection, localization and navigation, and speaking to human users modules.

Restaurant Setup

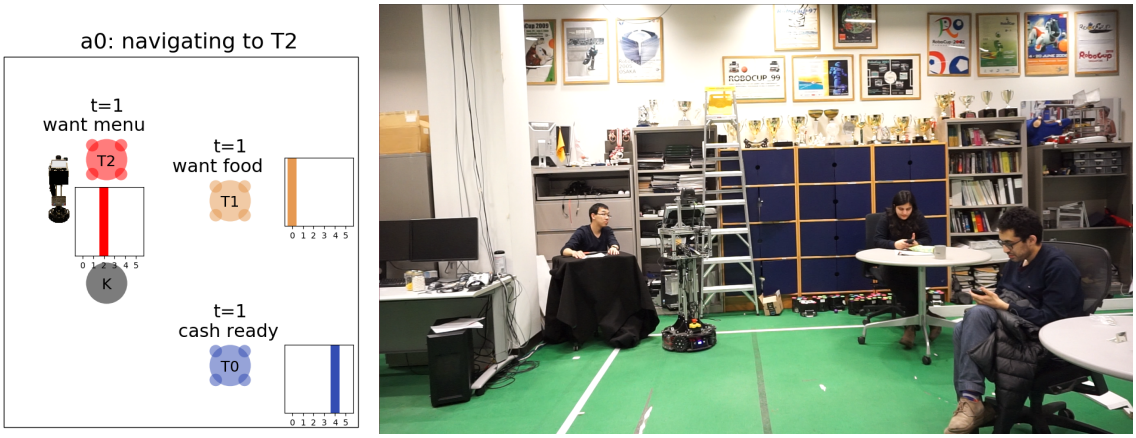
Our restaurant setup has three tables with one person on each table as shown in Fig. 5.6 and one robot. In our restaurant setting, the positions that the robot navigates to, the kitchen and the tables, are hard-coded. The restaurant follows the same model description that we explained earlier. We added two new actions to each POMDP model, *go to the kitchen* and *pick up food for a table*. The kitchen is shown with k in Fig. 5.6a. We also added a new state variable *food_pickedup* to each POMDP model that shows if the food or drink is picked up by the robot. While attending the customers, the robot runs the simulator in the background to get the observations after each action execution.

Since CoBot does not have arms to manipulate the world with, it relies on human help for some of its capabilities, *e.g.*, placing the food in its basket or handing over the menu. The robot uses a speaker to ask for help and inform the customers about its planning and execution status.

Example Scenario on the Robot

Here[1] is a video of one example scenario using our setup. The restaurant has 3 tables and a kitchen area. The robot is running a robot simulator in the background that produces all the observations.

[1] https://youtu.be/cq2TpFoPc60



(a) Simulation (b) Real-world

Figure 5.6: A restaurant setting with 3 tables ($T0$, $T1$ and $T2$) and one robot. The top-view configuration of the restaurant is shown on the left and is explained in the experiments section.

The status of the tables, wait time, current request and the belief over the satisfaction level, is shown in the top-left corner of the video. A detailed explanation of the video is in Appendix A.

5.5 Conclusion and Discussion

We propose an algorithm to speed up POMDP planning for domains where a robot is required to accomplish a set of tasks that are partially observable and evolve independently of each other. We exploit the observation that the number of tasks that the robot can accomplish within a short horizon is limited and present an algorithm that leverages the solutions to much smaller POMDP models to optimally solve the combined model with all the tasks. We prove the optimality of our approach and evaluate it on a restaurant setting.

The proof for the optimality of the approach exploits 1) an assumption that the tasks are independent and 2) an observation that in many domains the number of tasks that the robot can accomplish within a horizon is very limited. In some domains, these two assumptions might serve as the limitations of our approach. Our approach in this chapter is limited to domains where short horizon planning is sufficient. Specifically, short horizon planning works well in domains where the long-term plan quickly becomes sub-optimal or even infeasible after a few time steps. In the next chapter, we extend this approach to provide efficient planning over long fixed-length horizons without discounting and infinite-length horizons with discounting.

Chapter 6

Optimal Long-Horizon Planning for Achieving Multiple Independent POMDPs

6.1 Motivation

In this chapter, we focus on long-sighted planning for a class of problems with multiple independent tasks that are partially observable and evolve over time. In the previous chapter, we exploited the structure found in these problems, namely the independence between the tasks, to optimally and efficiently plan for a short fixed planning horizon. Selecting the right planning horizon can be challenging since an overly short horizon may result in a low-quality solution while supporting a longer horizon quickly becomes computationally impractical. Specifically, in POMDP planning, the complexity of planning grows exponentially with the horizon so a long horizon may easily preclude online planning. In this chapter, we address this challenge. In particular, we extend the previous algorithm to provide efficient planning over long fixed-length horizons without discounting and infinite-length horizons with discounting. The key idea we exploit to achieve efficiency is to compute lower and upper-bounds on the value of an optimal solution for variable horizons which allow us to terminate the search early while guaranteeing optimality. We present the algorithm, analyze its theoretical properties, and demonstrate its efficiency on the waiting tables domain. Before getting into the details of our algorithm, we briefly discuss the previous algorithm and the key ideas we use to extend and apply it on long-horizon problems.

Review of Short-horizon Multi-task Planner In the previous chapter, we exploited the observation that in some domains the number of tasks, k^*, that the robot can attend to within the horizon H is limited. Given this observation, if the robot optimally solves all possible sub-problems of size k^* with different combinations of tasks, it can find the optimal solution to the agent POMDP.

We proved that decomposing the agent POMDP into a series of sub-problems of size k^* and solving all combinations of k^* out of N tasks[1], $tpls = \{tpl \in \mathcal{P}(P) : |tpl| = k^*\}$, and returning the action with the highest value from them is the optimal solution to the agent POMDP. Note that each member tpl (tuple of size k^*) of the set $tpls$ is a sub-problem that can be solved by building a combined POMDP from the POMDPs in tpl. The robot assumes that a trajectory of *no op* actions is being executed on the POMDPs that are not in tpl. We call this approach *multi-task fixed-horizon planner* or *Multi-task-FH*.

This algorithm used the solutions to the individual client POMDPs to compute lower and upper-bounds on the optimal value of the agent POMDP to prune the $tpls$ set. Different from this algorithm that only uses the solutions to the single tasks to prune the low-quality tasks, *e.g.*, in the restaurant domain, tasks that do not need immediate attention from the robot and can be ignored momentarily, we take a more gradual approach and monotonically improve the bounds to prune the low-quality tasks. We start with single tasks ($k = 1$), but gradually increase k and solve sub-problems of size k ($< k^*$) to prune the tasks. Since our algorithm gradually improves the bounds to prune as many tasks as possible, it eventually solves less number of sub-problems of size k^* compared to the algorithm in Chapter 5. We first use the single tasks to prune, then pairs, then triplets, and so on. In addition, we use a truncated horizon h ($h < H$) to compute the solutions to the sub-problems of size k rather than the full horizon H which is needed to solve the sub-problems of size k^*. This gradual and monotonic improvement of the bounds and planning until a truncated horizon h enables the robot to efficiently and optimally plan over *long* fixed-length horizons without discounting and *infinite-length* horizons with discounting rather than planning for a short fixed horizon as done previously.

6.2 Approach

In this section, we first explain the main ideas that we use to extend the agent POMDP planner (explained in section 5.2.2) to be applied on long-horizon problems. We call this new approach *agent POMDP with adaptive horizon* or *agent-POMDP-AH*. We then explain how agent-POMDP-AH is extended to include the key insights and the efficiency of the short-horizon planner explained in Chapter 5. We call our approach *multi-task POMDP with adaptive horizon* or *multi-task-AH* since in addition to leveraging the multiple independent tasks structure, we adapt the horizon (specifically, iteratively increase it) to improve the solution's quality.

Similar to the previous algorithm, we use an online planning framework which interleaves planning and execution. Its main loop is in Alg. 5. During the planning phase, the algorithm

[1]Symbol $\mathcal{P}$ represents the power set.

computes the best action to execute given the robot's current belief (lines 3-7). In the execution phase, the robot executes the selected action (line 8), updates the belief state (line 9), and replans after each action execution.

Algorithm 5: Online Planner with Adaptive Horizon

```
1 MultiTaskAdaptiveHorizonPlanner (env, P, h, H)
2     while not AllTasksDone() do
3         tpls ← InitializeTuples(P,h,H)
4         while V̲ ≠ V̄ or h ≠ H do
5             a,tpls,V̲,V̄ ← SelectAction(P,h,H,tpls)
6             h ← h+1
7             tpls ← RecomputeTuples(h,tpls) // this function is only needed in the multi-task-AH approach
8         observations ← Step(env, a)
9         UpdateBeliefs(P,observations)
```

6.2.1 Agent POMDP with Adaptive Horizon

We adapt the agent-POMDP-FH approach for the class of problems with multiple independent tasks to enable the robot to efficiently plan for long horizons. This approach uses a similar procedure to solve the agent POMDP as agent-POMDP-FH, but modifies it with two main ideas. The key ideas are that instead of expanding the belief tree of all the tasks for the full horizon H, the robot 1) builds the belief tree until a truncated but gradually increasing horizon h and 2) computes the lower and upper-bounds on the value of the fringe nodes at the truncated horizon. To compute the bounds for the fringe nodes, the robot only solves the individual tasks for the remaining horizon $H - h$ (or ∞ in the infinite-horizon case) and combines their solutions. It then computes the lower and upper-bounds for the non-fringe nodes by propagating the bound values up from the fringe nodes by following the Bellman equation (Eq. 5.6). Note that when planning with a truncated horizon h, the planner expands the combined model of all the tasks only till the truncated horizon h, but the individual tasks are solved till the full horizon H to compute the bounds. When the lower and upper-bounds on the value of the robot's belief become equal, the optimal solution is found and the search is terminated. This enables the robot to terminate the search before reaching the full planning horizon H. We call the agent POMDP solver that follows this process `TruncatedAgentPOMDP`. For long horizons, solving the individual tasks (to compute the bounds) is much faster than expanding the belief tree of the combined model; thus, this approach is efficient compared to agent-POMDP-FH.

Instead of planning for a fixed horizon H, this algorithm (Alg. 5) performs planning for increasing values of horizon h until one of the following conditions are satisfied: 1) the horizon

limit H is reached, or 2) the lower-bound $\underline{V}$ on the value of the robot's belief is equal to its upper-bound $\bar{V}$ (line 4). The first condition assures that the algorithm is terminated when it reaches the maximum horizon H and outputs the same solution as planning for a fixed horizon H. The second condition enables the robot to terminate planning before reaching the full planning horizon, thus being more efficient than the agent POMDP approach with a fixed horizon H.

Alg. 6 provides the implementation of some of the functions in Alg. 5 for the agent-POMDP-AH approach. The `TruncatedAgentPOMDP` solver builds a combined model with all the client POMDPs in P. It finds the bounds for the fringe nodes using the `ComputeBounds` function and propagates the bounds up to compute the bounds for the non-fringe nodes. We refer to all the POMDPs in tpl as tpl_u; for the agent POMDP, $tpl_u = P$ (all possible tasks). The intuition behind the lower-bound computation (line 7) is to only consider the best client POMDP from tpl and perform *no ops* on the other POMDPs. This is similar to taking a greedy approach of always selecting the best task to attend to rather than interleaving the tasks. This is indeed a possible solution, hence it is the lower-bound. The intuition behind the upper-bound computation is to assume that the robot can address all the client POMDPs (tasks) in tpl in parallel. We only have one robot so this is an upper-bound.

Since the client POMDPs are solved over and over for different beliefs and horizons during planning to compute the bounds, their solutions are cached and reused in the process.

Algorithm 6: Agent POMDP with Adaptive Horizon

1 **InitializeTuples** (P,h,H) **return** $tpls \leftarrow \{(P,\emptyset)\}$

2 **SelectAction** (P,h,H,$tpls$)

3 $(\underline{V}_P,\bar{V}_P) \leftarrow$ TruncatedAgentPOMDP(h,H,$tpls$)

4 $a_{best} \leftarrow$ action with highest $\bar{V}_P$

5 **return** a_{best},$tpls$,$\underline{V}_P$,$\bar{V}_P$

6 **ComputeBounds** (b,tpl) // for the remaining horizon $H-h$

7 $\underline{V} \leftarrow \max_{p \in tpl_u}(V_p^*(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V_q^n(b_q))$; $\bar{V} \leftarrow \sum_{p \in tpl_u} V_p^*(b_p)$

8 **return** $\underline{V}$,$\bar{V}$

6.2.2 Multi-task POMDP with Adaptive Horizon

We exploit the two key ideas from the previous section and extend multi-task-FH to address long horizon planning. Multi-task-FH is able to leverage the independent tasks structure in the problem to efficiently solve the agent POMDP, and agent-POMDP-AH speeds up planning for long horizons by terminating the search earlier through the truncated horizon and bound

computations. We combine the benefits of the two approaches in multi-task-AH.

Multi-task-FH exploits the observation that within a fixed horizon H, the robot can only consider a limited number of tasks k^*. Similarly here, we also consider all possible subsets of size k^* as it is needed to ensure optimality. However, in addition to this, we leverage the observation that within the truncated horizon h, $h \leq H$, the robot can only consider k tasks ($k \leq k^*$), and it performs *no ops* on the other tasks. Intuitively, we use the key idea from the multi-task-FH planner twice, once to divide the agent POMDP of size P into smaller problems of size k^*, and the second time to divide the smaller problems of size k^* into sub-problems of size k, $k \leq k^*$, that can be solved more efficiently. Leveraging the truncated horizon to further limit the number of tasks that the robot can attend to enables us to significantly speed up planning. The robot only considers combined models of size k till horizon h, rather than combined models of size k^*, but computes the lower and upper-bounds on k^* individual tasks for the remaining horizon $H - h$. Note that the lower and upper-bound computations are done on the individual tasks till the full horizon H; so their computations should consider all k^* tasks to ensure similar optimality guarantees as the previous algorithm (as the agent can attend to k^* tasks within horizon H). Our approach is powerful in the infinite-horizon problems with N tasks. In such problems the number of tasks that the robot can attend to within $H = \infty$ is N, $k^* = N$; thus, if $k \ll N$, the algorithm significantly expedites planning by solving multiple sub-problems of smaller sizes rather than solving the agent POMDP with all the N tasks. As we increase the truncated horizon h, we might need to increase the size of the subsets, *i.e.*, increase k. We explain how we address this important aspect of the problem later.

Alg. 7 shows the multi-task-AH algorithm. The function `InitializeTuples` considers all possible subsets of P with size k^* (line 2) and further divides it into subsets of size k (line 3). Each subset of size k^* ($tpl \in tpls$) is divided into two sets, tpl_c with size k and tpl_l with size $k^* - k$. The truncated agent POMDP is built from the POMDPs in tpl_c while executing *no ops* on the POMDPs in tpl_l, but the bound computations for the fringe nodes are done on all POMDPs in $tpl_u = tpl_c \cup tpl_l$ to assure valid lower and upper-bounds on the value of the tuple tpl (`TruncatedAgentPOMDP` function). The `SelectAction` function solves a truncated agent POMDP for each tpl (line 7) to compute its bounds while executing *no ops* on other POMDPs that are not in tpl (line 8). It then updates the bounds on the value of the full agent POMDP (line 9). The algorithm then removes the tuples for which the upper-bounds are less than the lower-bound of the agent POMDP and returns the action from the tpl with the highest upper-bound (lines 10-11).

The size of the sub-problems and their bounds gets updated as the truncated horizon h increases. Function `RecomputeTuples` updates the $tpls$ set as the number of tasks that the

robot can attend to within the horizon increases from k to $k' = k + 1$. For each $tpl \in tpls$, a member of tpl_l is removed and added to its tpl_c set. We consider removing any element from the tpl_l set to generate all possible new tuples. This is to ensure that the optimality guarantees hold as we increase the truncated horizon h.

Algorithm 7: Multi-task POMDP with Adaptive Horizon

InitializeTuples (P,h,H)
  k,k^* $\leftarrow$ the maximum # tasks the robot can attend to within h and H;
  $T \leftarrow \{tpl : tpl \in \mathcal{P}(P), |tpl| = k^*\}$
  $tpls' \leftarrow \{(tpl_c, tpl_l) : tpl_u \in T, tpl_c \in \mathcal{P}(tpl_u), |tpl_c| = k, tpl_l = tpl_u \setminus tpl_c\}$
  return $tpls'$
SelectAction (P,h,H,$tpls$)
  for $tpl \in tpls$ **do**
    ($\underline{V}_{tpl}$,$\bar{V}_{tpl}$) $\leftarrow$ TruncatedAgentPOMDP(h,H,tpl)
    $(\underline{U}_{tpl}, \bar{U}_{tpl}) \leftarrow (\underline{V}_{tpl}, \bar{V}_{tpl}) + \sum_{q \in P \setminus tpl_u} V_q^n$
    $\bar{V}_P = \max(\bar{V}_P, \bar{U}_{tpl})$; $\underline{V}_P = \max(\underline{V}_P, \underline{U}_{tpl})$
  $tpls \leftarrow \{tpl : tpl \in tpls, \bar{U}_{tpl} \geq \underline{V}_P\}$
  $a_{best} \leftarrow$ action from the tpl with highest $\bar{U}_{tpl}$
  return a_{best},$tpls$,$\underline{V}_P$,$\bar{V}_P$
RecomputeTuples (h,$tpls$)
  k,k' $\leftarrow$ the maximum # tasks the robot can attend to within $h-1$ and h; $tpls' \leftarrow tpls$
  if $k \neq k'$ **then**
    $tpls' \leftarrow \{(tpl_c \cup \{p\}, tpl_l \setminus \{p\}) : tpl \in tpls, p \in tpl_l\}$
  return $tpls'$

6.3 Optimality Proofs

We first provide the intuition on why our algorithm is optimal and discuss the details of the proofs later for the interested reader.

6.3.1 Summary of the Proofs

Lemma *Alg. 7 converges to the optimal solution of the agent POMDP with a fixed horizon H.*

Intuition behind proof: We first prove the optimality of agent-POMDP-AH by discussing why the lower and upper-bounds are valid and monotone. We then prove that multi-task-AH finds the same solution as agent-POMDP-AH.

The lower-bound is computed by selecting the best task to attend to and performing *no ops* on the other tasks. This is indeed a possible solution, hence it is a valid lower-bound. The upper-bound is computed by assuming that the robot can address all the tasks in parallel. Since we only have one robot, this is a valid upper-bound. The monotonicity property assures that the lower and upper-bounds on the value of a belief node do not change or improve after each iteration of the algorithm (improve as the truncated horizon h increases). The intuition behind the proof is that as we increase the horizon (h to $h+1$) more of the belief tree is expanded, and the bounds are computed for the remaining horizon $H-h-1$ which gives a better estimate of the values compared to when the bounds are computed before the expansion for the remaining horizon $H-h$. Since the bounds are valid and monotone, and we compute the bounds for a given tpl using all the k^* tasks in tpl till the full horizon H (or all the P tasks in the agent-POMDP-AH algorithm), the bound computations for tpl is also valid and monotone.

We use the key idea from Chapter 5 twice, once to divide the agent POMDP into multiple small problems of size k^*, and the second time to divide the small problems into smaller sub-problems of size k. In Chapter 5, it is shown that the former maintains the optimality; the optimality proofs of the latter follows the same procedure as the former to prove that the multi-task-AH approach is optimal.

6.3.2 Complete Proofs

In this section, we provide the details of the proofs. We first prove that agent-POMDP-AH computes an optimal solution. We then prove that multi-task-AH finds the same solution as agent-POMDP-AH. We discuss both the proofs and the intuition behind them. The proofs use the independent tasks definition, as stated in [127].

Notation required for understanding the intuition behind the proofs (mostly borrowed from Chapter 5):

- $V^*_{p,t}$: the optimal value of the client POMDP p at time t.
- $V^n_{p,t}$: the value of following a trajectory of *no ops* for the client POMDP p at time t.
- $V^*_{P,t}$: the optimal value of the agent POMDP created from the POMDPs in P at time t (Eq. 5.6).
- $V^*_{tpl,t}$: the optimal value of the agent POMDP created from only the client POMDPs in tpl at time t.

- $\underline{V}^h_{tpl,t}(b_{tpl})$,$\bar{V}^h_{tpl,t}(b_{tpl})$: the lower and upper-bound on the value of a belief node b_{tpl} in the belief tree of a truncated agent POMDP created only from the members of tpl till h. The bounds on the values of the fringe nodes of the truncated belief tree are computed using Eq. 6.1 and Eq. 6.8.

More notation required for understanding the proofs (mostly borrowed from Chapter 5)

- $\mathbb{B}^*$: this refers to the Bellman operator.
- A_{tpl}: only considers the actions associated with the POMDPs in tpl and performs *no op* on the other POMDPs (same as Eq. 5.4, but the union is over tpl, not P).
- $Q^*_{p,t}(b, a)$: the optimal value of the client POMDP p at time t for belief b and action a.
- $U^*_{tpl,t}$: the optimal value of the agent POMDP built from P with the action set A_{tpl}. Intuitively, $U^*_{tpl,t}$ considers both the value of the POMDPs in tpl ($V^*_{tpl,t}$) and the value of executing *no ops* on the ones that are not in tpl.

Lower and upper-bound

We show that the bound computations are valid (Lem. 1 and 2) and monotone (Lem. 3 and 4). The monotonicity property assures that the lower and upper-bounds on the value of a belief node does not change or improves after each iteration of the algorithm (increase in the truncated horizon h). The bounds on the value of the fringe nodes are computed for the remaining horizon $H - h$ (or ∞ in the infinite-horizon case) using Eq. 6.1 and Eq. 6.8. The bounds for the non-fringe nodes are computed by propagating the bound computations of the fringe nodes up to the root belief node. We do not make any assumptions regarding the maximum possible horizon in the bound computations, thus the lemmas also hold for the infinite-horizon problems with discounting. We use mathematical induction to prove Lem. 1 to 4.

Lemma 1 *Eq. 6.1 provides a lower-bound on the value of a tuple* $tpl = (tpl_c, tpl_l)$ *where* $tpl_u = tpl_c \cup tpl_l$.

$$\underline{V}_{tpl,t}(b_{tpl}) = \max_{p \in tpl_u} \left[V^*_{p,t}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,t}(b_q) \right] \leq V^*_{tpl,t}(b_{tpl}) \tag{6.1}$$

Intuition behind proof: Let us consider that only one task from tpl_u, $p \in tpl_u$, can be executed till the full horizon (V^*_p), and we perform *no ops* on the other tasks ($\sum V^n_q$). The best task will

then be selected as the lower-bound on V^*_{tpl}, $\max_p [V^*_p + \sum V^n_q]$.

Proof: The proof goes by mathematical induction. For $h' = 1$, if $\forall p \in P, V^*_{p,0}(b_p) = 0$, Eq. 6.2 follows from Eq. 5.6:

$$\begin{aligned} V^*_{tpl,1}(b_{tpl}) &= \overbrace{\max_{p \in tpl_u}}^{\max_{a \in A_{tpl_u}}} \Big[\max_{a \in A_p} \Big[\sum_{i \in tpl_u} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) \Big] \Big] \\ &= \max_{p \in tpl_u} \Big[V^*_{p,1}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,1}(b_q) \Big] \end{aligned} \tag{6.2}$$

If $h' = t - 1$, we assume Eq. 6.3 and consequently Eq. 6.4 and show that they both hold for $h' = t$.

$$V^*_{tpl,t-1}(b_{tpl}) \geq \max_{p \in tpl_u} \Big[V^*_{p,t-1}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,t-1}(b_q) \Big] \tag{6.3}$$

$$\forall p \in tpl_u : V^*_{tpl,t-1}(b_{tpl}) \geq V^*_{p,t-1}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,t-1}(b_q) \tag{6.4}$$

We expand Eq. 5.6 as follows (b_{tpl} or b):

$$\begin{aligned} V^*_{tpl,t}(b) = \max_{a \in A_{tpl_u}} \Big[& \sum_{i \in tpl_u} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) \\ & + \gamma \sum_{z_q \in Z_q} \Pr(z_q | b_q, a_q) \ldots \sum_{z_r \in Z_r} \Pr(z_r | b_r, a_r) V^*_{tpl,t-1}(b^a_z) \Big] \end{aligned} \tag{6.5}$$

We substitute Eq. 6.4 in Eq. 6.5. Given the independence assumption, for a specific Z_i, we can marginalize out the sum over Z_js ($j \neq i$). $\forall p \in tpl_u$, we obtain:

$$\begin{aligned} V^*_{tpl,t}(b) &\geq \max_{a \in A_{tpl}} \Big[Q^*_{p,t-1}(b_p, a[p]) + \overbrace{\sum_{q \in tpl_u \setminus \{p\}} Q^n_{q,t-1}(b_q, a[q])}^{Q_{noop}} \Big] \\ &\geq \max_{a \in A_p} \Big[Q^*_{p,t-1}(b_p, a[p]) + Q_{noop} \Big] \geq V^*_{p,t}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,t}(b_q) \end{aligned} \tag{6.6}$$

Thus, Eq. 6.7 holds for every $h' = t$.

$$V^*_{tpl,t}(b) \geq \max_{p \in tpl_u} \Big[V^*_{p,t}(b_p) + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,t}(b_q) \Big] \tag{6.7}$$

Lemma 2 *Eq. 6.8 provides an upper-bound on the value of a tuple* $tpl = (tpl_c, tpl_l)$.

$$\bar{V}_{tpl,t}(b_{tpl}) = \sum_{p \in tpl_u} V^*_{p,t}(b_p) \geq V^*_{tpl,t}(b_{tpl}) \tag{6.8}$$

Intuition behind proof: The idea behind the upper-bound computation is to assume that the robot can attend to all the tasks in tpl, $p \in tpl_u$, in parallel ($\sum V^*_p$). We only have one robot, so this is an upper-bound on V^*_{tpl}.

Proof: Similar to Lem. 1, the proof goes by mathematical induction. For $h' = 1$, the following equation holds.

$$\begin{aligned} V^*_{tpl,1}(b_{tpl}) &= \max_{a \in A_{tpl_u}} \Big[\sum_{i \in tpl_u} \sum_{s \in S_i} b_i(s) R_i(s, a[i]) \Big] \\ &\leq \sum_{i \in tpl_u} \max_{a \in A_{tpl_u}} \Big[\sum_{s \in S_i} b_i(s) R_i(s, a[i]) \Big] = \sum_{i \in tpl_u} V^*_{i,1}(b_i) \end{aligned} \tag{6.9}$$

We assume Eq. 6.10 holds for $h' = t-1$ ($p, q, r, \ldots \in tpl_u$) and show that it also holds for $h' = t$.

$$V^*_{tpl,t-1}(b) \leq V^*_{p,t-1}(b_p) + \ldots + V^*_{q,t-1}(b_q) + \ldots + V^*_{r,t-1}(b_r) \tag{6.10}$$

Similar to Lem. 1, Eq. 6.10 is substituted in Eq. 6.5, and simplified to obtain Eq. 6.11. Thus, Eq. 6.8 holds for every $h' = t$.

$$\begin{aligned} V^*_{tpl,t}(b) &\leq \max_{a \in A_{tpl_u}} \Big[\sum_{i \in tpl_u} Q^*_{i,t}(b_i, a[i]) \Big] \\ &\leq \sum_{i \in tpl_u} \max_{a \in A_{tpl_u}} Q^*_{i,t}(b_i, a[i]) = \sum_{p \in tpl_u} V^*_{p,t}(b_p) \end{aligned} \tag{6.11}$$

Lemma 3 *The lower-bound computation is monotone.*

$$\underline{V}^{h}_{tpl,t}(b_{tpl}) \leq \underline{V}^{h'}_{tpl,t}(b_{tpl}) \quad \text{where } h < h' \text{ and } h, h' \leq H \tag{6.12}$$

In both $\underline{V}^{h}_{tpl,t}$ and $\underline{V}^{h'}_{tpl,t}$'s computations, the belief tree is built till horizon h. To compute $\underline{V}^{h}_{tpl,t}$, the lower-bound on the value of the fringe belief nodes at horizon h are computed using Eq. 6.1 and are propagated up the belief tree. To compute $\underline{V}^{h'}_{tpl,t}$, the algorithm expands the tree for d more steps, $h' = h + d$, and then uses Eq. 6.1 to compute the lower-bound for the fringe nodes at depth $h + d$ and propagates the bounds up the belief tree. In both cases the lower-bound on the value of the fringe nodes are computed using Eq. 6.1 till the full horizon H. This property guarantees that as the truncated horizon increases, from h to h' ($h < h'$), the lower-bound on the value of a certain fringe node at horizon h and consequently the non-fringe nodes are non-decreasing.

Intuition behind proof: The main difference between $\underline{V}^{h}(b')$ and $\underline{V}^{h'}(b')$ for a certain belief node b' at depth h (or horizon h) is that the former uses the trivial lower-bound estimate for the node, but the latter does more computation to expand the belief tree further before using a similar trivial lower-bound estimate for the nodes at depth $h + d$. To compute the lower-bound for a fringe node at depth h, $\underline{V}^{h}(b')$, the algorithm assumes that from there on till H, only one task can be executed and *no ops* are executed on the other tasks (one possible solution). So, expanding the belief tree (exhaustive search) for d more steps till horizon $h + d$ to compute $\underline{V}^{h'}(b')$ will only find the same or a better solution than achieving a single task. *I.e.*, the lower-bound on b' is non-decreasing as we increase the horizon.

Proof: For a certain leaf node b at horizon h, we compare its lower-bound when the truncated agent POMDP is built till h against when it is built till h'. The proof goes by mathematical induction. First, we show that $\underline{V}_{tpl,H-h} \leq \mathbb{B}^* \underline{V}_{tpl,H-h-1}$ holds for $d = 1$. We proved this previously when we substitute Eq. 6.4 in Eq. 6.5 to get Eq. 6.6, thus:

$$\begin{aligned} V^*_{tpl,H-h} &= \mathbb{B}^* V^*_{tpl,H-h-1} \geq \mathbb{B}^* \underline{V}_{tpl,H-h-1} \\ &\geq \max_{p \in tpl_u} \Big[V^*_{p,H-h} + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,H-h} \Big] = \underline{V}_{tpl,H-h} \end{aligned} \tag{6.13}$$

Now, we assume that for $h' = h+d$, the following holds for the belief node b: $\underline{V}_{tpl,H-h} \leq \mathbb{B}^*_d \underline{V}_{tpl,H-h-d}$, and we prove that the same equation also holds if $h' = h + d + 1$. For a certain belief b, both $\underline{V}_{tpl,H-h} \leq \mathbb{B}^* \underline{V}_{tpl,H-h-1}$ and $\underline{V}_{tpl,H-h} \leq \mathbb{B}^*_d \underline{V}_{tpl,H-h-d}$ hold, thus the following equation holds for

$h' = h + d + 1$:

$$\begin{aligned} V^*_{tpl,H-h} &\geq \mathbb{B}^*_d\Big[\mathbb{B}^* \underline{V}_{tpl,H-h-d-1}\Big] \geq \mathbb{B}^*_d \underline{V}_{tpl,H-h-d} \\ &\geq \max_{p \in tpl_u}\Big[V^*_{p,H-h} + \sum_{q \in tpl_u \setminus \{p\}} V^n_{q,H-h}\Big] = \underline{V}_{tpl,H-h} \end{aligned} \tag{6.14}$$

Lemma 4 *The upper-bound computation is monotone.*

$$\begin{aligned} &\bar{V}^h_{tpl,t}(b_{tpl}) \geq \bar{V}^{h'}_{tpl,t}(b_{tpl}) \\ &\text{where } h < h' \text{ and } h, h' \leq H \end{aligned} \tag{6.15}$$

This property guarantees that as the horizon increases, from h to h', the upper-bound on the value of a certain fringe node and consequently the non-fringe nodes are non-increasing.

Intuition behind proof: Similar to the intuition we gave for the lower-bound's monotonicity, for a certain belief node b' at depth h, $\bar{V}^h(b')$ estimates the upper-bound by assuming that all the tasks can be performed in parallel. However, $\bar{V}^{h'}(b')$ expands the belief tree for d more steps before assuming that all the tasks can be performed in parallel. Thus, given that $\bar{V}^{h'}(b')$ uses the Bellman equation during the d steps, it has a better estimate of the upper-bound than the assumption that all the tasks can be attended to in parallel during that d steps as assumed in $\bar{V}^h(b')$. *I.e.*, as the horizon increases and more of the belief tree is expanded, the upper-bound on the value of b' improves (*i.e.*, is non-increasing).

Proof: Similar to Lem. 3's proof, the proof goes by mathematical induction. First, we show that $\mathbb{B}^* \bar{V}_{tpl,H-h-1} \leq \bar{V}_{tpl,H-h}$ holds for $d = 1$. We proved this previously when we substitute Eq. 6.10 in Eq. 6.5 to get Eq. 6.11, thus:

$$\begin{aligned} V^*_{tpl,H-h} &= \mathbb{B}^* V^*_{tpl,H-h-1} \leq \max_{a \in A_{tpl_u}} \Big[\sum_{i \in tpl_u} Q^*_{i,H-h-1}\Big] \\ &\leq \sum_{i \in tpl_u} \max_{a \in A_{tpl_u}} Q^*_{i,H-h-1} = \sum_{i \in tpl_u} V^*_{i,H-h} = \bar{V}_{tpl,H-h} \end{aligned} \tag{6.16}$$

We assume that for the belief node b and $h' = h + d$, $\mathbb{B}^*_d \bar{V}_{tpl,H-h-d} \leq \bar{V}_{tpl,H-h}$ holds, and we prove it also holds if $h' = h + d + 1$. We know both $\mathbb{B}^* \bar{V}_{tpl,H-h-1} \leq \bar{V}_{tpl,H-h}$ and $\mathbb{B}^*_d \bar{V}_{tpl,H-h-d} \leq \bar{V}_{tpl,H-h}$ hold, thus,

$$V^*_{tpl,H-h} \leq \mathbb{B}^*_d \mathbb{B}^* \bar{V}_{tpl,H-h-d-1} \leq \mathbb{B}^*_d \bar{V}_{tpl,H-h-d} \leq \bar{V}_{tpl,H-h} \tag{6.17}$$

In summary, we proved that the bound computations are valid and monotone; thus if $tpl = (P, \emptyset)$, Lem. 1 to 4 prove the optimality of agent-POMDP-AH. Given the iterative nature of the horizon, in the worst case, the agent-POMDP-AH approach reaches the full horizon H and obtains the same solution as the agent-POMDP-FH approach.

Multi-task-AH

We prove Alg. 7 is optimal. We assume k^* and k are the maximum number of tasks that the robot can attend to within H and the truncated horizon h respectively. $\hat{V}_P$ denotes the value of the agent POMDP under such assumptions, referred to as *limited tasks assumption*.

Lemma 5 *The lower and upper-bounds on the value of the agent POMDP created from the set P, $\hat{\underline{V}}_P$ and $\hat{\bar{V}}_P$, can be computed by Eq. 6.18 and Eq. 6.19 respectively where $tpls = \{tpl \in \mathcal{P}(P) : |tpl| = k^*\}$, and the bounds are monotone. (proof of `SelectAction` function in Alg. 7)*

$$\hat{\underline{V}}_{P,t}(b) = \max_{tpl \in tpls} (\underline{V}_{tpl,t}(b_{tpl}) + \sum_{q \in P \setminus tpl_u} V^n_{q,t}(b_q)) \leq \hat{V}^*_{P,t}(b) \tag{6.18}$$

$$\hat{\bar{V}}_{P,t}(b) = \max_{tpl \in tpls} (\bar{V}_{tpl,t}(b_{tpl}) + \sum_{q \in P \setminus tpl_u} V^n_{q,t}(b_q)) \geq \hat{V}^*_{P,t}(b) \tag{6.19}$$

Intuition behind proof: In [127], we proved that finding the optimal values of all $tpl \in tpls$ (V^*_{tpl}) while performing *no ops* on the other POMDPs ($\sum V^n_q$) and selecting the best V-value, $\max_{tpl \in tpls} (V^*_{tpl} + \sum V^n_q)$, provides the optimal solution to the agent POMDP. We proved in Lem. 1 to 4 that the lower and upper-bounds on V^*_{tpl} are valid and monotone. The validity and monotonicity of $\hat{\underline{V}}_P$ and $\hat{\bar{V}}_P$ then simply follow from the validity and monotonicity of $\underline{V}_{tpl}$ and $\bar{V}_{tpl}$.

Proof: We show that the bounds are valid and then argue why they are also monotone. From [127], we know:

$$\hat{V}^*_{P,t}(b) = \max_{tpl \in tpls} U^*_{tpl,t}(b) \tag{6.20}$$

$$U^*_{tpl,t}(b) = V^*_{tpl,t}(b_{tpl}) + \sum_{q \in P \setminus tpl} V^n_{q,t}(b_q) \tag{6.21}$$

We proved in Lem. 1 and 2 that $\underline{V}_{tpl,t}(b_{tpl}) \leq V^*_{tpl,t}(b_{tpl})$ and $\bar{V}_{tpl,t}(b_{tpl}) \geq V^*_{tpl,t}(b_{tpl})$ respectively. Thus, $\underline{U}_{tpl,t}(b)$ and $\bar{U}_{tpl,t}(b)$ computed by substituting $V^*_{tpl,t}(b_{tpl})$ by $\underline{V}_{tpl,t}(b_{tpl})$ and $\bar{V}_{tpl,t}(b_{tpl})$ in Eq. 6.21 are lower and upper-bounds on ${U^*}_{tpl,t}(b)$. We substitute $\underline{U}_{tpl,t}(b)$ and $\bar{U}_{tpl,t}(b)$ in Eq. 6.20 to prove Eq. 6.18 and Eq. 6.19. Given that the bound computations for $V^*_{tpl,t}$ are monotone, $\sum V^n_{q,t}(b_q)$ does not change for a given tuple as we increases the horizon, and the max operator does not change the monotonicity of $\underline{U}_{tpl,t}$ and $\bar{U}_{tpl,t}$, $\hat{\underline{V}}_{P,t}$ and $\hat{\bar{V}}_{P,t}$ are monotone.

Lemma 6 *Alg. 7 converges to the optimal solution of the agent POMDP in both finite horizon problems without discounting and infinite horizon problems with discounting.*

Intuition behind proof: In Lem. 1 to 5, we proved that dividing the agent POMDP into subtasks ($tpl \in tpls$) and computing the lower and upper-bounds for all the tuples in $tpls$ provide valid and monotone bounds on the value of the agent POMDP. In those lemmas, we assumed that $k = k^*$, *i.e.*, a combined model of all the k^* tasks is expanded till the truncated horizon h even though we know that the robot can only attend to k tasks within h. Differently, in Alg. 7, to efficiently solve each tpl for a small truncated horizon h, we only consider subsets of size k, but compute the bounds on all the k^* POMDPs in tpl, so $k < k^*$. Both cases $k = k^*$ and $k < k^*$ use the same lower and upper-bound computations and have the same h as their truncated horizon. However, in the former we perform the tree expansion on a combined model built from all the POMDPs in the tpl set, but in the latter we consider all combinations of the tasks with size k out of the POMDPs in tpl and perform the tree expansion on those only. The proof uses the same idea as [127]. It uses the assumption that within a certain horizon h, only k tasks can be attended to, so if we consider all combinations of k tasks out of the members of the tpl (tpl_u), we will get the same solution as the combined model of all the tasks in tpl. Given that the bound computations are the same in both cases, when $k < k^*$, we get the same solution as when $k = k^*$ (proof for line 3 in Alg. 7 and the `RecomputeTuples` function), and Alg. 7 computes valid and monotone bounds on the value of the agent POMDP.

Proof: In Lem. 1 to 5, we proved that dividing the agent POMDP into subtasks ($tpl \in tpls$) and computing the lower and upper-bounds for all the members of $tpls$ provide valid and monotone bounds on the value of the agent POMDP. In these lemmas, we assumed that $k = k^*$, so line

3 of Alg. 7 would become $tpls' = \{(tpl_c, tpl_l) : tpl \in tpls, tpl_c = tpl, tpl_l = \emptyset\}$, and the `RecomputeTuples` function would not change the $tpls$ set. However, the benefits of our approach are manifested when the truncated horizon h is smaller than the full planning horizon H, and consequently $k < k^*$. In Alg. 7, we divide each tpl into two sets, tpl_c with k tasks and tpl_l with $k^* - k$ tasks (all possible combinations of k tasks out of k^* tasks), perform the tree expansion for the POMDPs in tpl_c while executing *no ops* on the members of tpl_l, and compute the bounds on all members of $tpl_u = tpl_c \cup tpl_l$. When $k < k^*$, if we prove that by using this approach, we get the same solution as when $k = k^*$, we prove that Alg. 7 computes valid and monotone bounds on the value of the agent POMDP.

Notice that the only difference between $k = k^*$ and $k < k^*$ is that in the former we perform the tree expansion on all POMDPs in the tpl set, but in the latter we consider all combinations of tasks with size k for the tpl_c set and perform the tree expansion on the POMDPs in tpl_c only. The lower and upper-bound computations are the same in both cases.

We use the same idea as [127], Lem. 2 and Asm. 1 in [127]. For a set of tasks called tpl and the maximum number of tasks that the robot can attend to within the horizon h (k), the robot can optimally solve the combined model of all tasks by considering all subsets of tasks of size k (tpl_c with size k). Given the independence between the tasks and the limited tasks assumption, Eq. 6.22 was proved in [127] for $tpl = P$ and $h = H$ (k^* tasks). Same deductions also apply here to prove Eq. 6.22.

$$\hat{V}^*_{tpl,t}(b) = \max_{tpl' \in tpls'} V^*_{tpl'_c,t}(b_{tpl'_c}) + \sum_{q \in tpl'_l} V^n_{q,t}(b_q) \tag{6.22}$$

$$tpls' = \{(tpl_c, tpl_l) : tpl_c \in \mathcal{P}(tpl), |tpl_c| = k, tpl_l = tpl_u \setminus tpl_c\}$$

This explains why dividing tpl further into subsets of size k (line 3 in Alg. 7) does not change the validity and monotonicity of the bounds and gives us the same bounds as if we were to build a combined model of all the POMDPs in tpl_u.

As we increase h, k should also increase to ensure that Eq. 6.22 is still valid. More specifically, we have to update each tuple in the $tpls'$ set ($tpl' \in tpls'$) to have $k+1$ POMDPs in tpl_c and $k^* - k - 1$ POMDPs in tpl_l. This is done by the `RecomputeTuples` function. The algorithm simply removes a POMDP from tpl_l and adds it to the POMDPs in tpl_c to create a new tpl'_c set of size $k+1$ and a new tpl'_l set of size $k^* - k - 1$, $tpl' = (tpl'_c, tpl'_l)$. The algorithm considers removing any POMDP from tpl_l, to create all possible new tuples. Since the new $tpls'$ set satisfies the limited tasks assumption as we increase the horizon, Eq. 6.22 holds.

Therefore, all parts of the algorithm preserve the optimality guarantees, and Alg. 7 computes

an optimal solution for the agent POMDP. In the worst case, the multi-task-AH approach reaches the full horizon H and obtains the same solution as the multi-task-FH approach. In infinite-horizon problems with discounting, when $h \to \infty$, Alg. 7 converges to the optimal solution of the agent POMDP with $H = \infty$.

6.4 Experiments

The *multi-task-AH* approach (Alg. 7) caches and reuses the solutions to the client POMDPs during each episode while the *multi-task-AH wo reuse* approach does not reuse. In both, the agent starts with $h = 2$. We compare these two versions of our algorithm against the alternatives listed below.

- **Multi-task-FH**: We compare against the multi-task approach with a fixed horizon as proposed in [127].
- **Agent POMDP**: We use the agent POMDP model that we described in the approach section. We compare against 2 versions of this algorithm: 1) agent-POMDP-FH and 2) agent-POMDP-AH.
- **N-samples**: This approach was proposed by [166]. They select N tuples randomly from $tpls = \{tpl \in \mathcal{P}(P) : |tpl| = k^*\}$. They show that solving subsets of tasks by using a sampling-based method is faster than using the same method to solve the combined model.
- **HPOMDP**: Each task is represented as a macro action, so there are N atomic macro actions. While one task is getting executed, *no ops* are executed on the other tasks. This approach greedily selects a task to attend to.

We first compare against the approaches that provide optimality guarantees, some of which exploit the independent tasks structure and some of which do not. In all the algorithms, we use the same optimal POMDP solver to solve each task or subsets of tasks. It is possible to expedite the planning further by changing the system's specifications[2] and the specifics of the implementation, *e.g.,* using a different programming language, or changing the underlying POMDP planner that solves each task or subsets, but the algorithms should still perform similarly in relation to one another. Note that the contributions of this work is not on what POMDP algorithm we use for solving each task or subsets of tasks, but rather on how to leverage the multi-task structure to expedite long horizon planning.

[2]We use python 3 with the seed 100 for randomly initializing the episodes. The CPU time for all the experiments have been calculated on a cluster with Intel Xeon E5-2609 processors (2.40GHz) with a memory per CPU of 500MB, 128GB RAM and CentOS Linux 7 compared to Intel Core i7-8700K processor (3.70GHz), 64GB RAM and Ubuntu 16.04 in [127]. Different from [127], in our restaurant domain γ is 1, not 0.95.

We also compare against sub-optimal methods such as HPOMDP and N-samples that leverage the independent tasks but do not consider all combinations of interleaving the tasks.

6.4.1 Restaurant Model

We use the same restaurant model as discussed in Chapter 3 with 3 to 12 tables. For this restaurant model, if $H \leq 4$, the robot should solve pairs of tasks ($k^* = 2$) to find the optimal solution. If $H = 5, 6$, or $H = 7, 8$, the robot should consider triplets ($k^* = 3$) or quadruplets ($k^* = 4$), respectively. We run each algorithm for 10 episodes, each for 20 actions. In each episode, the state of the restaurant is randomly initialized with the belief probability of 1. We use the same random initialization for the episodes across all the algorithms to remove the variations that result from the random initialization in our comparisons.

6.4.2 Quantitative Results

We report on the performance of our method using three metrics. For each episode, we average each metric over 20 actions (by adding $\#\text{actions} \leq 20$ to the while loop on line 2 of Alg. 5). For all the metrics, we report the mean and variance for 10 episodes (if available) for different number of tables and maximum horizons. First, we compare the algorithms in terms of average rewards (Fig. 6.2). For each episode with the same initialization, we take the difference between the average reward of multi-task-AH and other approaches. The average reward for an algorithm in one episode is computed as follows $r_{avg} = \sum_{t=0}^{20} \sum_{i=0}^{N} \frac{\sum_{s \in S_i} b_{i,t}(s) R_{i,t}(s, a_t)}{|a_t|}$ where $|a_t|$ is the length of the selected action at time t. Second, we compute the average planning time over 20 actions by running the algorithms until they terminate (Fig. 6.1 and Fig. 6.2). *I.e.,*, we run the algorithms till they reach H for the fixed horizon ones, and until Alg. 5 terminates for the adaptive horizon ones. Third, we plot the maximum h (final horizon) that multi-task-AH reaches before terminating the planning while loop in Alg. 5. We compute the final horizon for one episode by averaging the final horizons of 20 action selections. We plot the mean and standard deviation of the final horizon for 10 episodes (Fig. 6.3). Lastly, we report on the percentage of the runs (out of 200 action selection runs) where the algorithm terminates before reaching the full planning horizon H in Tab. 6.1.

In terms of average reward in Fig. 6.2 (on the right), all the optimal approaches are mostly a constant zero line as we take the difference between the (optimal) multi-task-AH's average reward and other algorithms' average reward, and the sub-optimal approaches mostly perform worse than the optimal ones (are below the line). This is especially the case for higher number of tables and longer horizons. The HPOMDP approach has a very low average reward and hence

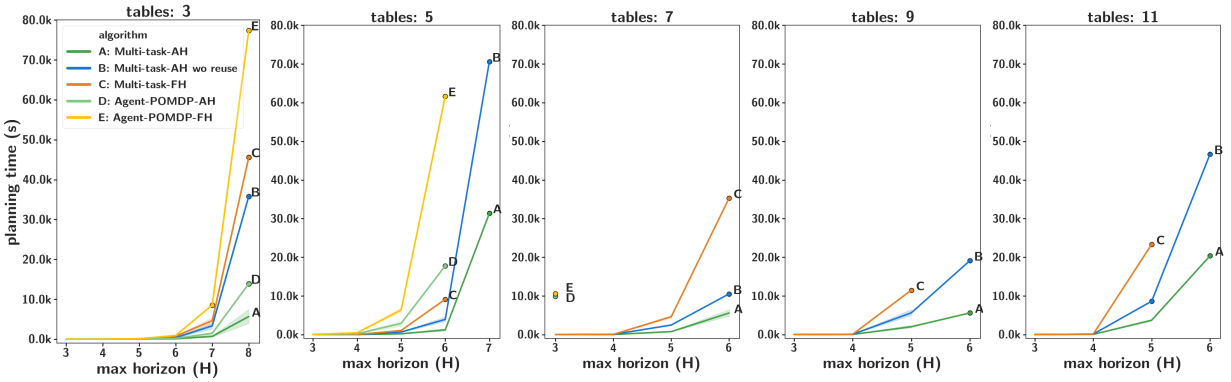


Figure 6.1: Planning times of the optimal algorithms for different horizons H and number of tables.

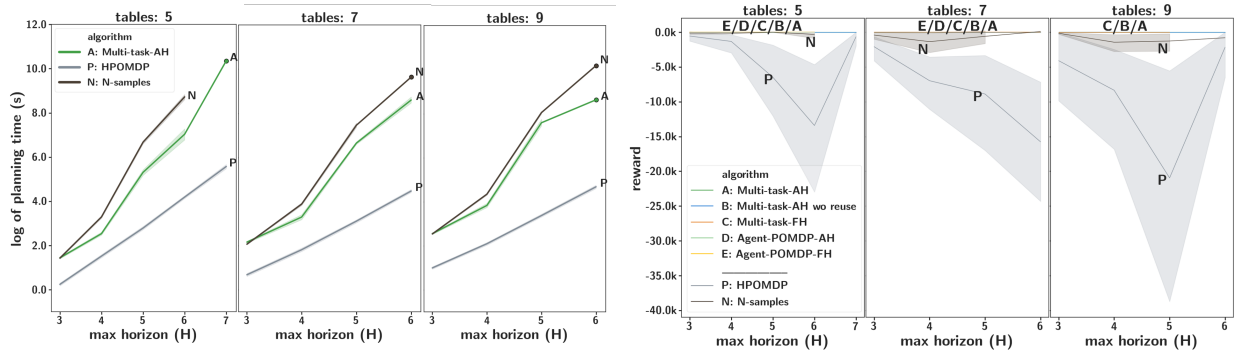


Figure 6.2: Planning time comparisons between the performance of our algorithm and the sub-optimal algorithms in natural logarithmic scale on the left. The average reward comparisons for all the algorithms on the right.

is highly sub-optimal. This is because it only considers the solutions to the individual tasks to select an action to execute, rather than interleaving the tasks. This highlights the significance of interleaving the tasks to compute an optimal solution. The very low average reward indicates the customers' highly dissatisfaction with the service.

We plot the planning time in Fig. 6.1 and Fig. 6.2 (on the left). The lines and the shadows around each line show the mean and variance of each algorithm over 10 episodes. For some horizons and number of tables, some algorithms take a long time to finish the 10 episodes. For those, we only report the results of one episode (shown with a small circle). In most cases, the variance on the planning time computed over multiple episodes is very small. Our approach has a much better planning time than all the optimal approaches, especially as the number of tables increases, since it is able to terminate the search before reaching the full planning horizon. We evaluated the impact of having an adaptive horizon by comparing it against the algorithm in Chapter 5 (multi-task-FH) which uses the independent tasks structure but does not have an

adaptive horizon. We observed that both approaches are scalable as the number of tables increase, but our approach is more than 5 times faster for longer horizons. Even for a few tables, *e.g.*, the 3 tables shown in Fig. 5.6, for the maximum horizons of 4, 5, 6, 7 and 8, our algorithm is 2.7, 6.1, 8.2, 6.5, and 8.1 times better than multi-task-FH, respectively. We evaluated the impact of leveraging the independent tasks structure by comparing it against agent-POMDP-AH. We observed that agent-POMDP-AH is not scalable to a large number of tables (does not even finish in a reasonable amount of time for more than 7 tables), and our approach has a much better planning time even for a small number of tables.

The sub-optimal algorithms leverage the multi-task structure of the problem, so they are more scalable that the optimal approaches. The sub-optimal HPOMDP approach has the lowest planning time, but at the cost of optimality as its average reward is much worse than that of the optimal approaches. Even the N-samples approach (fixed horizon) which gives a sub-optimal solution has a higher planning time than our (provably optimal) approach. Considering all possible combinations of interleaving the tasks (computing an optimal solution) significantly improves the customers' satisfaction.

We analyze why our algorithm is faster than the approaches with a fixed horizon by plotting the final horizon h that the multi-task-AH algorithm reaches before terminating in Fig. 6.3. For most horizons and number of tables, we provide the mean and variance on 10 episodes. For others we only provide the average final horizon for one episode (*e.g.*, $H = 7$ and 5 tables, and $H = 6$ and more than 8 tables). We observe that on average our algorithm is much more efficient than the baselines since it mostly expands the belief tree until a horizon that is smaller than the full planning horizon. Within each episode, the final horizon at which the search terminates ranges from the minimum $h = 2$ to the maximum H, as planning for the full horizon is occasionally needed to guarantee optimality; however, all episodes on average have smaller horizons than the maximum horizon H. For a specific value of H, when the number of tables is small, the number of tasks that need immediate attention is also small, consequently the algorithm can terminate at shorter horizons more often than when the number of tables is large. For the curves associated with different values of H, this is why we observe an increase in the final horizon as the number of tables increases (at the beginning of each graph), and each graph becomes almost flat for large number of tables. On a different note, given a fixed number of tables and for increasing values of H, the final horizon is also increasing. This is because as H increases, the algorithm searches for a better solution and terminates at higher final horizons.

We further evaluate our algorithm (Alg. 5 and Alg. 7) by analyzing how often the runs terminate with the condition $\underline{V} = \bar{V}$ versus $h = H$ (line 4 in Alg. 5). We average the results over different number of tables (if available) as shown in Tab. 6.1. As an example, for $H = 4$,

Table 6.1: The percentage of the runs where the algorithm terminates when the bounds are equal versus when the full horizon H is reached.

	$\underline{V} = \bar{V}$	$h = H$	num of tables
$H = 4$	56.7%	43.3%	3 to 12
$H = 5$	39.2%	60.8%	3 to 12
$H = 6$	60.1%	39.9%	3 to 8
$H = 7$	91.5%	8.5%	3 to 4
$H = 8$	100%	0%	3

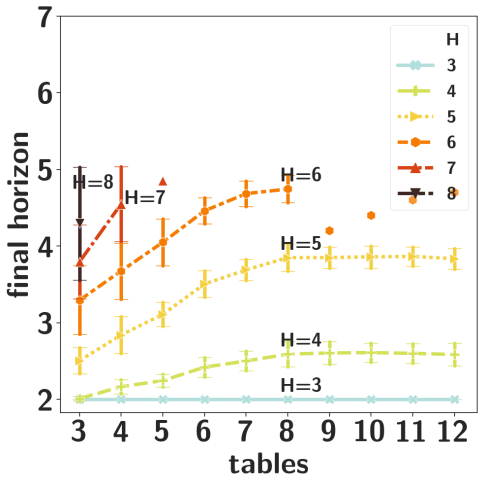


Figure 6.3: Final horizon at which Alg. 7 terminates.

in 56.7% of the runs the algorithm finished before reaching the final horizon H ($\underline{V} = \bar{V}$) and in 43.3% of the runs, it finished when $h = H$ (aggregated over 3 to 12 tables). As discussed above, the percentages are different for different number of tables, and with fewer number of tables, the algorithm mostly terminates when $\underline{V} = \bar{V}$. Overall, our algorithm is more efficient than the fixed horizon approaches as it is able to terminate the search before reaching the full horizon.

6.4.3 Qualitative Results

We discuss why two different final horizons are determined by the algorithm in the two restaurant configurations in Fig. 6.4. Each histogram shows the belief over satisfaction (partially observable) for a particular table. The leftmost bar is 0 (very unsatisfied) and the rightmost is 5 (very satisfied). *E.g.*, in the right figure, the robot believes that $T1$ is 40% slightly unsatisfied and 60% neutral. Variable t represents the tables' wait time. The robot's goal is to increase the tables' satisfaction by attending to their requests (above each table) as soon as possible. Each table might need service at different points in time, *e.g.*, collecting cash. In the left, the robot goes to $T2$ to perform the

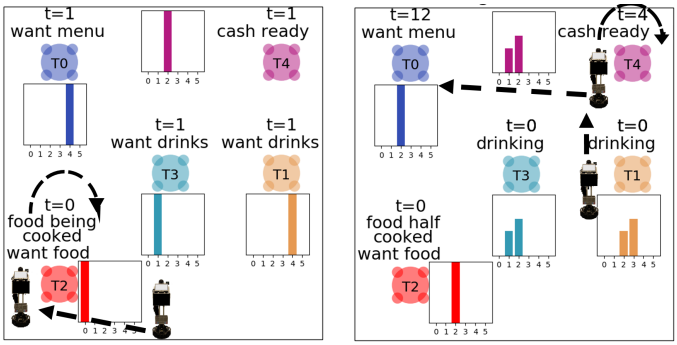
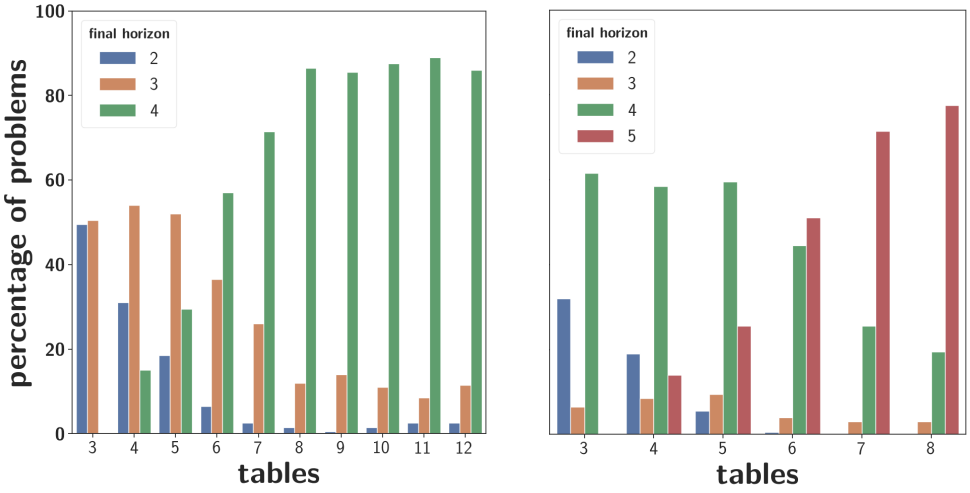


Figure 6.4: Two different configurations of the domain with 5 tables.



(a) Maximum horizon $H = 5$

(b) Maximum horizon $H = 6$

Figure 6.5: A plot illustrating the distribution over the termination length of the horizon (final horizon) for different number of tables with prespecified maximum horizon H.

action “your food is not ready". In the right, the robot goes to $T4$ to execute “take cash" and then goes to service $T0$. The maximum horizon in both cases is $H = 6$. The final horizon in the left is $h = 5$ (solves triplets) while it is $h = 3$ in the right (solves pairs only). In the left, although $T2$ and $T3$ have lower satisfaction levels, all the tables are in need of immediate attention; thus solving only pairs of tasks ($h \leq 4$) is not sufficient to make a decision ($\underline{V} \neq \bar{V}$). For $h = 5$, the robot builds and solves triplets, and obtains $\underline{V} = \bar{V}$. In the right belief state, only $T0$ and $T4$ are in need of immediate attention (and servicing $T2$ does not worth it since $T2$ is far away from $T4$ and $T0$). For $h = 3$, the robot solves pairs of tasks and obtains $\underline{V} = \bar{V}$.

The reduction in computation time depends on how busy the restaurant is and at what point the tables are in the dining process. The following characteristics of the restaurant domain contribute to the reduction in computation time 1) the multiple tasks do not depend on one another, and 2)

the tasks should be interleaved and attended to at different points during the dining process. The robot reasons about what tasks require immediate attention and what tasks can be pruned early on during planning (deferred to the next action-selection). In the extreme case, if all the tasks should be attended to immediately at all action-selection steps, there is no reduction in computation as the tasks cannot be pruned and should be considered to find the optimal solution. As we move towards less number of tasks in need of immediate attention, the reduction in computation is higher.

In Fig. 6.3, we showed the mean and variance of the final horizon for different values of H and number of tables. In this section, we look into the percentage distribution of the final horizon associated with $H = 5$ and $H = 6$. Fig. 6.5 illustrates why our algorithm expedites planning for $H = 5$ and $H = 6$. Note that for long horizons and large number of tables, if even a small percentage of the problems are solved faster, this will have a significant effect on reducing the planning time.

A total of 200 problems (10 episodes, each 20 action selections) are considered for this evaluation. To give an example, for 3 tables and a maximum horizon $H = 5$, around 50% of the problems terminate at $h = 2$ and the other 50% terminate at $h = 3$. For $H = 5$, as the number of tables increases, the percentage of the problems that terminate at final horizon $h = 4$ also increases and the percentage of the problems that terminate at final horizons $h = 2$ and $h = 3$ decreases. Similarly, when $H = 6$, as the number of tables increases the percentage of the problems with final horizon $h = 5$ increases. As we discussed before, when the number of tables is small, the number of tasks that need immediate attention is also small, consequently the algorithm can terminate at shorter horizons more often than when the number of tables is large. In conclusion, depending on the configuration of the restaurant and how needy the tables are, the termination horizon varies.

6.5 Conclusion and Discussion

We propose an approach that uses the structure in the class of problems with multiple independent tasks that are partially observable and evolve over time to perform efficient planning for long-horizon problems and infinite-horizon problems with discounting. Leveraging both the independent tasks structure and having an adaptive horizon enable the robot to optimally solve long horizon problems more efficiently than other optimal approaches. We prove that our approach is optimal and demonstrate its efficiency on the restaurant domain. We evaluate the performance of our algorithm compared to the state-of-the-art algorithms in a restaurant with 2 to 12 tables and a maximum horizon of 8, $H = 8$. We observe that our optimal planner performs significantly better

than the other approaches as the number of tables increases. Nevertheless, our optimal approach is still impractical for large number of tables and a maximum planning horizon greater than 8, $H > 8$. In future work, we will focus on real-time planning approaches where optimal planning is impossible. We plan to provide practical anytime algorithms that use both the independent tasks and the adaptive horizon (which inherently is anytime) and integrate them with the practicality of anytime sampling-based approaches such as [176] which randomly samples a subset of scenarios to speed up planning. Furthermore, future work involves running the multi-task-AH planner on the real robot as was demonstrated in Chapter 5 and analyzing the complexity of the problems that our robotic system can address (including the planning, perception, and execution modules).

Chapter 7

Robot Planning and Execution in Presence of Discrepancy between Robot's Observations and the POMDP Model

7.1 Motivation

Autonomous robots that face a diversity of environments, a variety of tasks and a range of interactions cannot be pre-programmed by foreseeing at the design stage all possible courses of actions they may require. Especially, in dynamic and changing environments with semantically rich tasks and human interactions such as the restaurant domain, unexpected situations that are not predicted by the robot's model might arise. In a real-world application such as the restaurant setting, there are differences in terms of the types of the restaurant, *e.g.*, fine dining, casual dining, cafes and diners, and the customers they target, *e.g.*, families with children, college students, and seniors. In these applications, coming up with a perfect and accurate model that works for everyone is very challenging. Even if a certain model works for most of the tables in a specific restaurant, it might not work for a particular table that might not belong to the target group that the restaurant model is designed for. For example, in a restaurant setting similar to the one we described before, going to the tables frequently to double-check if the customers have everything that they need might be rewarded by the model and might work for most of the customers; however, the customers on a certain table might be having a lunch business meeting in which people don't want to be interrupted frequently. In another example, delivering bread to the customers before serving the food might be a part of the robot's process to keep the customers satisfied; however, a certain table might have allergies to wheat. These are a few examples of the unexpected situations that the robot might encounter and should be able to tackle.

Two very common ways of designing a robot model are 1) asking an expert in both robotics and the domain to design a model, or 2) learning the model from data. In the first approach, an expert would require considerable expertise in both robotics and the domain in order to design a perfect model of a real-world application such as the restaurant domain. For a restaurant domain, including all possible customer preferences and constraints is error-prone and time-consuming. Even if possible, in a lot of domains, approximations such as removing the rare events or considering the majority vote from all experts are often introduced for computational tractability, thereby resulting in unexpected situations. In the second category, learning approaches have been introduced to learn a model through supervised learning or trial-and-error interactions with the environment. The performance of these methods heavily depend on the amount of data that they have access to, *i.e.*, a model learned in a few instances of the environment might not generalize well to other instances. In some of these cases, the data might have been gathered with a different robot or learned from human demonstrations of the behaviors, *e.g.*, the waiting tables task. Identifying the mapping between the human and the robot or the two different robots that allows the transfer of information from one to the other is known as the correspondence problem in the Learning from Demonstration (LfD) literature [6] and can result in learning an approximation of the robot's exact planning model. More specifically, in the restaurant domain, the capabilities of the human waiters differ from that of the robot waiters; thus, learning a model from real restaurant settings with human waiters will introduce approximations and errors.

Even for applications in well-structured environments with a reduced range of tasks, where engineered robotics operations are feasible, deployment and adaptation costs can be reduced if the robot is equipped with monitoring and replanning capabilities. For a real-world application such as the restaurant setting, deploying the robot in the real-world early on before having a complete model is a key step to figuring out what needs to be added to the model or if more data needs to be gathered. However, given that the model has inaccuracies, the robot should be able to handle the unexpected situations gracefully and should not cause interruption in the restaurant's service procedures when the unexpected situations arise. Thus, no matter what approach we use to come up with the model, without an exact planning model, the robot will encounter unexpected situations where its observations do not match its expectations and should have a way to resolve these situations to eventually achieve the goal of the task. Although being theoretically well-founded, our planning algorithms and many other planning algorithms depend heavily on the accuracy of the domain model. In this chapter, we present algorithms that address the unexpected situations that arise as a result of planning over inaccurate models.

There are two categories of approaches that could be used to address the unexpected situation, namely replanning and learning approaches. Replanning, plan repair, and state estimation

approaches are able to address the unexpected situations if they are not due to a fundamental change in the environment [72, 111, 164, 192]. In these approaches the premise is that planning from scratch in the new, unexpected situation would produce a plan that is valid. However, these approaches fail if the unexpected situation is due to a fundamental change in the environment and thus repeats itself. This is for instance the case when the agent applies an incorrect model of its own actions during planning. In such situations, preceding replanning with a model-adjustment step to alter the planning operators might be required. Learning approaches enable a robot to explore the environment in order to improve its planning model [61, 90, 119, 156]. Differently, we focus on a multi-task setting where the robot should be able to tackle the unexpected situations for a particular inaccurate task while effectively responding to the other tasks, and there is no time for learning the exact probabilities through exploration. In addition, learning the true parameters of the model might not be useful beyond interactions with specific customers, thus making learning exact parameters for future customers not useful. The objective of this work is to enable the robot to still achieve the task at hand, *e.g.*, getting the restaurant customers successfully out of the restaurant, when an unexpected situation arises rather than how to update the model with the new information from the unexpected situation. We will discuss the existing approaches that could be used to update a model in Chapter 8.3.

In this work, an unexpected situation is due to a discrepancy between the robot's observation and what is expected to be observed based on the robot's planning model. We address the discrepancies by integrating them into the original planning problem and then provide algorithms to efficiently solve the new augmented planning problem. We will use a simple navigation domain similar to the one proposed in [161] and available in [35] to explain the ideas in this chapter and then discuss how the ideas are applied to the restaurant domain. In the rest of this chapter, we follow an online POMDP planning framework where the planning and execution steps are interleaved until the robot reaches a goal (goal is observable). During the planning phase, the algorithm computes the best action to execute given the POMDP's belief state. The execution step executes the selected action and updates the belief state of the POMDP by using the obtained observation. The robot replans after each action execution. When a discrepancy happens, the robot should plan its next action by using the unexpected observation.

Example - Navigation Domain

Let's consider the 2D navigation domain shown in Fig. 7.1 where the robot should reach the star goal (state 3). The agent's true initial state is 4, but the initial belief state of the robot is that the robot can be in any state other than the goal uniformly (all states 0 to 10 except 3). The 2D domain has 11 states. The only actions available to the robot are "north", "south", "east" and "west". With

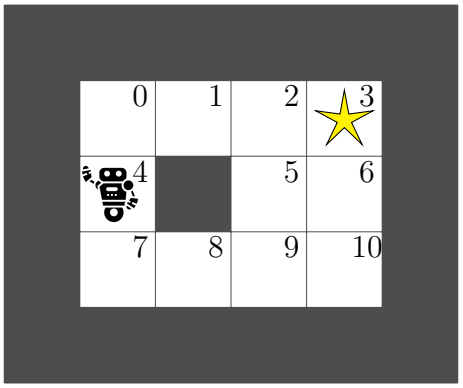


Figure 7.1: A 3×4 grid with the goal located at state 3, and the robot located at state 4.

any of these actions, the agent will take a step in that direction with probability 0.8 and with 0.1 probability in the states that are adjacent to the robot in that direction. For example if the robot goes north from the state 9, with probability 0.8 it will end up in the state 5 and with probability 0.1 in the states 8 and 10. If any of these transitions are not possible because of an obstacle, the robot stays in its place with that probability. For example if the robot goes east from the state 8, with probability 0.8 it ends up in 9 and with probability 0.2 it stays in 8. The only exception is that if the robot is in the state 4 and executes north, with probability 1.0 it ends up in the state 0. The robot has two sensors on its sides. The robot gets the following observations regarding where the obstacles are: "left" (obstacle on the left), "right" (obstacle on the right), "neither" (both adjacent cells empty), "both" (obstacles in both of the adjacent cells), and "goal". The goal is fully observable. The observation function does not depend on the action (only depends on the state) and has a probability of 1.0 for the true observation. For example, in the state 8, the robot observes "neither", and in the state 4, it observes "both" (assumes that the robot is pointing to the north always). The robot receives a negative cost of 0.4 with any action execution except if it executes an action in the goal state.

In our navigation example, the planner goes through the scenario below. The belief state comes in the form of a list of states and their probabilities $(state, prob)$. The robot has a uniform distribution over the states as shown below. In the planning phase the robot selects the action east and observes "both". Given the observation, the robot can only be in the state 4. The planner then selects the action north as it is parts of the optimal route to the goal. The robot then observes "both". If the robot is in the state 4 and executes action north, with 1.0 probability it should end up in the state 0. In the state 0, the robot can only observe "left". The observation "both" does not match what is expected by the model, so a discrepancy happens.

Grid-world domain - the first two planning steps before a discrepancy occurs
Robot's initial belief*: uniform probability over states other than the goal, (0,0.1),(1,0.1),(2,0.1),(3,0.0),(4,0.1),(5,0.1),(6,0.1),(7,0.1),(8,0.1),(9,0.1),(10,0.1)* **Robot's action:** east **Robot's observation:** both ***Robot's updated belief****: (4,1.0)* **Robot's action:** north **Robot's observation:** both ***Robot's updated belief****: discrepancy happens as the expectation was that the robot ends up in the state* 0 *and observes "left".*

We say a discrepancy has happened when the robot's observation of the world does not belong to the set of observations that the robot could receive from the environment by reasoning on its planning model. Under the assumption that the robot's sensors are not defective, the planning model should be the reason for erroneous expectation, and the different elements of the POMDP model should be examined for their inaccuracies. Similar to many other planning and execution monitoring approaches (even learning), we assume that the state and observations spaces of the planning model are complete. Thus, the transition and observation functions of the POMDP are the main cause of the discrepancy.

In the previous example, the truth is that there is a thick carpet between the state 4 and 0, and the robot actually did not move at all and will not be able to move if it keeps executing the north action (Fig. 7.2). However, it could also be possible that the carpet is traversable with some probability (the carpet is thin), and the robot would be able to go to the state 0 after multiple executions of the north action. Both of these explanations of the discrepancy are concerned with the accuracy of the POMDP's transition model (dynamics). Differently, it could also be the case that the robot is in the state 0 (Fig. 7.3), but because of an incorrect heading it observes "both" (although it should observe "left"). This last explanation is concerned with the accuracy of the robot's observation function.

The discrepancy problem sounds very similar to a popular problem called the kidnapped-robot problem where a robot is unable to estimate its own position via the localization process [64]. Some approaches address the discrepancies by resetting the belief state of the robot when a discrepancy is detected, or specifically in the localization domain, when the robot is lost due to an inaccurate model [47, 111]. A prominent approach within this body of literature called Sensor Resetting Localization (SRL) is an extension of Monte Carlo Localization and addresses the kidnapped-robot problem by inserting additional hypotheses generated from sensing in the belief state when the robot is uncertain of its position. These approaches provide promising

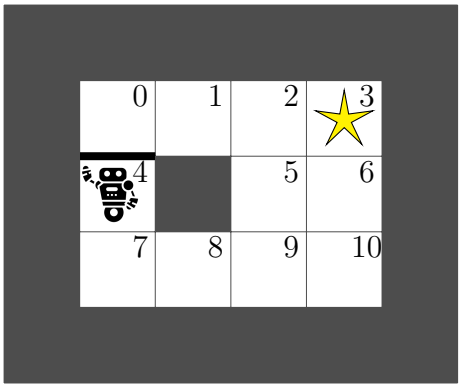


Figure 7.2: There is a thick carpet between the states 4 and 0, so the robot cannot transition from 4 to 0 anymore.

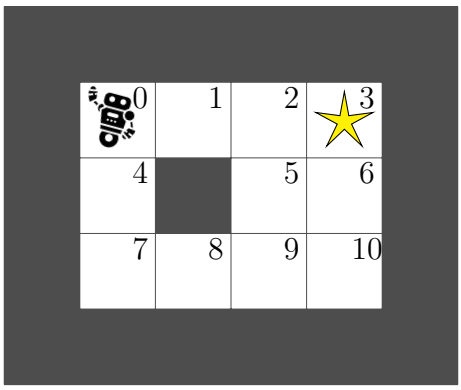


Figure 7.3: A 3×4 grid with the robot located at state 0.

results in domains such as robot soccer where external disturbances are the main reason for the discrepancy [47, 111]. However, in some cases, a discrepancy might not be a result of having a wrong estimate of the robot's state and might be due to systematic changes in the environment that would require changes to the model. In these cases, if the model is used as it is, the robot might never be able to achieve its goal. For example, in the navigation domain above, if we do not reason about the change to the model that the robot cannot transition from the state 4 to 0, and only reset the belief state to a uniform distribution over all states (similar to the initial belief state of this problem), the robot would still try to go north rather than realizing that route is infeasible now and taking an alternative route of going south. The robot will get into a loop of detecting a discrepancy and resetting the belief state after each execution of the north action which prevents it from reaching its goal. Please see below a scenario where the robot only resets the belief state.

Resetting the belief state to address the discrepancy
Robot's initial belief: *uniform probability over states other than the goal, (0,0.1),(1,0.1),(2,0.1),(3,0.0),(4,0.1),(5,0.1),(6,0.1),(7,0.1),(8,0.1),(9,0.1),(10,0.1)* **Robot's action:** east **Robot's observation:** both ***Robot's updated belief***: *(4,1.0)* **Robot's action:** north **Robot's observation:** both ***Robot's updated belief***: *discrepancy happens as the expectation was that the robot ends up in the state* 0 *and observes "left".* ******We reset the belief state to a uniform distribution over all the states except the goal.*** ***Robot's updated belief***: *uniform probability over states other than the goal, (0,0.1),(1,0.1),(2,0.1),(3,0.0),(4,0.1),(5,0.1),(6,0.1),(7,0.1),(8,0.1),(9,0.1),(10,0.1)* **Robot's action:** east **Robot's observation:** both ***Robot's updated belief***: *(4,1.0)* **Robot's action:** north **Robot's observation:** both ***Robot's updated belief***: *the same discrepancy happens again.*

7.2 Formulation of Discrepancy Recovery as a Planning Problem

We address the discrepancies by integrating them into the original planning problem. Our algorithm follows the following three steps to handle the discrepancy:

- **Discrepancy detection**: we say a discrepancy has occurred when $\Pr(z_t|b_t, a_t) = 0$. This says that the probability of perceiving the current observation z_t given the current belief b_t and action a_t is zero. The robot does not know how to update the belief state and proceed from that point on as none of the states have a nonzero probability. Following the example, $\Pr(both|(4, 1.0), north) = 0$.
- **Discrepancy diagnosis**: in this step we find a set of hypotheses that can explain the discrepancy. For each hypothesis, we come up with a query targeted at an oracle regarding the hypothesis. We call these queries *clarification actions*. Each clarification action can invalidate a given hypothesis.
- **Discrepancy reasoning and recovery**: using the set of hypotheses and their associated

clarification actions, this step decides what the robot should do at each step to achieve the goal of the task. The need for discrepancy recovery is situation-dependent. If, for instance, discrepancy diagnosis is able to specify a subset of hypotheses in all of which the current plan should be continued, then there is no need to disambiguate between two candidate hypotheses right away when both candidates belong to this subset.

Hypotheses and Clarifications This section assumes that the discrepancy diagnosis step is done beforehand, and the robot has access to a set of hypotheses on why the discrepancy has happened. The main focus will be on discrepancy reasoning and recovery, and more specifically using the set of hypotheses to enable the robot to still achieve the goal of the task. The robot has access to a set of explanations which we call a hypotheses set and denote it by MH, and a set of clarification actions denoted by A_{cl}. Each hypothesis $h \in MH$ is a potential explanation for the discrepancy that states what changes to the model could explain the discrepancy. We assume that at least one of the hypothesis in MH truly explains the discrepancy. If a hypothesis is valid, we can apply its associated changes to the original model and use it in the rest of the planning episode. For each $h \in MH$, we have a corresponding clarification action with a cost associated with it. Each clarification action $a \in A_{cl}$ can invalidate a certain explanation or hypothesis.

Formally, a hypothesis h is represented as a set consisting of tuples $[type, elements, values]$, $h = \{[t, e, v], \ldots, [t', e', v']\}$. Each tuple enumerates what changes should be made to the original planning model; we call each tuple, a *modification tuple*. The set of all the modification tuples applied to the original planning model explains the current discrepancy without introducing discrepancies in the previous planning steps. The $type$ (t) specifies if the tuple is associated with the transition function T or the observation function O. In the case of the transition function, $elements$ $(e) = (s, a, s')$ specifies the state s, action a and next state s' that the transition function operates on. The s, a and s' variables can also take $*$ as their values meaning that they can be replaced with any state, any action and any state, respectively. We also refer to these $elements$ of type T as the parameters of the transition function. In the case of the observation function, $elements = (s', a, z)$ specifies the next state s', action a and observation z that the observation function operates on. The s', a and z variables can also take $*$ as their values meaning that they can be replaced with any state, any action and any observation, respectively. We also refer to these $elements$ of type O as the parameters of the observation function. The $elements$ variable basically describes how the parameters of the transition or observation functions should change. The $values$ (v) specifies if the probability for that transition or observation should change from nonzero to zero or change from zero to nonzero (binary variable). If the changes associated with all the $elements$ members of the modification tuples of h are applied to the original planning

model, the discrepancy would get resolved. The robot might have multiple hypotheses that explain the discrepancy, but it does not know which hypothesis is valid.

The objective of our planner is to achieve the goal of the task, *e.g.*, reaching the state 3, rather than figuring out which hypothesis is valid. Thus, instead of investigating the validity of the hypotheses, we include the hypotheses and the clarification actions associated with them in the original planning model. We let the planner to decide if a hypothesis actually affects the planning outcome and should be validated or not. For example, in the simple navigation domain above, let's assume that there is no chance that the robot can go from the state 4 to the state 0; let's call this hypothesis $h1$, $h1 = \{[T, (4, *, 0), 0], [T, (4, *, 4), 1]\}$. Now if we only have $h1$ in the hypotheses set, $MH = \{h1\}$, the robot should for sure take the alternative route of going south as $h1$ is the only valid hypothesis. However, if $MH = \{h1, h2\}$ where $h2 = \{[T, (4, *, 0), 1], [T, (4, *, 4), 1]\}$, which states that there is a good chance that the robot can still go north (the carpet is thin), the robot might decide to validate $h1$ versus $h2$ before deciding on a direction. This decision would depend on the cost of asking a clarification action compared to the cost of taking an alternative route of going south. If the former is small, it is worth to ask a clarification action rather than directly taking the alternative route of going south. If $h1$ is valid and $h2$ is invalid, only going south can take the robot to its navigation goal.

For each $h \in MH$, we have a corresponding clarification action with a cost associated with it. If a hypothesis is valid, we can apply its associated changes (*i.e.*, the modification tuples) to the original model and use the new model in the rest of the planning episode. Each clarification action asks questions regarding the tuples in the hypothesis' modification tuples set to invalidate it. Even if one modification tuple from a hypothesis' modification tuples set is invalid, the whole hypothesis is invalid. The clarification action cost is finite if it has not been asked before, and it becomes infinite when it is asked. We keep track of if a clarification action has been asked before by using a binary variable and including it in the augmented model's state, but for simplicity of notation, we do not enumerate it when we talk about the augmented model's state. If the validity or invalidity of a hypothesis is not verified yet, we have to consider its potential invalidity while planning. This is done by including a large cost associated with using the transitions or observations from the hypotheses' modification tuples ($elements$) in the cost function of the original planning model.

We will enable the robot to decide if and when a clarification action should be asked to invalidate a specific hypothesis. The robot will also decide which clarification action should be asked. The objective of our planner is to achieve the goal of the task, and the validity of a hypothesis or the invalidity of certain hypotheses might help with achieving the goal. We will explain how we come up with the MH and A_{cl} sets later, but it will not be the main focus of

this thesis. We augment the original POMDP model with the hypotheses set and the clarification actions and reformulate it such that the robot stays away from the regions of the state space where the discrepancy happened and the model is inaccurate. If staying away from these regions is inevitable or not preferred (*i.e.*, costly), the robot asks clarification questions to figure out where the actual inaccuracy lies and which explanation or hypothesis can address the discrepancy.

7.2.1 Discrepancy POMDP Model

Please see below how the original POMDP model is augmented to include the hypotheses and clarification actions. We use a goal POMDP representation described in Chapter 2.2.2 for both the original and the augmented POMDPs. It has been proven that any discounted reward POMDP can be converted to a goal POMDP [26]. Thus, one can convert the discounted reward POMDP from the previous chapters to a goal POMDP before augmenting it. The original POMDP model is represented with the tuple (S, G, A, Z, T, O, C), and the augmented POMDP model is represented with the tuple $(S', G', A', Z', T', O', C')$.

- State space (S'): we augment the state space of the original POMDP to include the possible hypotheses as follows $S' = S \times MH$. The belief state is a distribution over the potential hypotheses and their associated states. This enables the robot to keep track of which hypotheses have been invalidated and which ones could potentially be valid. Each hypothesis $h \in MH$ specifies what changes in the parameters of the transition or observation function resolves the discrepancy. We call the parameters associated with each hypothesis *unreliable parameters* as they can be the reason for the discrepancy, unless the hypothesis is invalidated.
- Goal states (G'): any member of G irrespective of its associated hypothesis is a goal state, $\forall g \in G \wedge \forall h \in MH, g' = [g, h] \in G'$.
- Action space (A'): we augment the action space of the robot to include the clarification actions, $A' = A \cup A_{cl}$. These actions enable the robot to invalidate the hypotheses.
- Observation space (Z'): we augment the observation space of the robot with "yes" and "no" answers to the clarification questions $Z' = Z \cup \{\text{yes}, \text{no}\}$.
- Transition function ($T'([s, h], a, [s', h'])$): the transition function is as follows. If h says that a certain hypothesis is valid, after applying any action, we still believe that the same hypothesis is valid (*i.e.*, $h = h'$). If $a \in A_{cl}$, a is treated as a *no op* action. $T_h(s, a, s')$ refers to the original transition function and includes the changes suggested by the hypothesis h. In $T_h(s, a, s')$, the robot considers a uniform distribution over the transition function parameters from the hypothesis h that are type T, $type = T$, and have a nonzero value of 1, $values = 1$.

$$T'([s,h],a,[s',h']) = \begin{cases} T'([s,h],\text{no op},[s',h']) & a \in A_{cl} \\ T_h(s,a,s') & a \notin A_{cl} \text{ and } h = h' \\ 0 & \text{otherwise} \end{cases}$$

- Observation function ($O'([s',h'],a,z)$): if we believe in the hypothesis h' and a clarification action asks about its validity, the answer should be "yes" with the probability 1. If the robot asks about the validity of a different hypothesis, and we believe in the hypothesis h', the answer should be "no" with the probability 1. Otherwise, the observation should be according to the original observation function including the changes suggested by the hypothesis h' denoted by $O_{h'}(s',a,z)$. In $O_h(s',a,z)$, the robot considers a uniform distribution over the observation parameters from the hypothesis h' that are type O, $type = O$, and have a nonzero value of 1, $values = 1$.

$$O'([s',h'],a,z) = \begin{cases} \mathbb{1}(z = \text{yes}) & a \in A_{cl} \text{ and } a \text{ asks about the hypothesis } h' \\ \mathbb{1}(z = \text{no}) & a \in A_{cl} \text{ and } a \text{ asks about another hypothesis} \\ O_{h'}(s',a,z) & \text{otherwise} \end{cases}$$

- Cost function $C'([s,h],a,[s',h'])$: we modify the original objective function and include the cost for the unreliable parameters. The actions and states that have unreliable transitions and observations are penalized through the cost function. In T', it is always the case that $h = h'$, so we only consider the case where $h = h'$. We use the following cost function $C'([s,h],a,[s',h]) = C(s,a,s') + wU_h(s,a,s')$ where C is the original cost function. For a list of modification tuples in the hypothesis h of form $[type, elements, value]$, U_h is equal to 0 if neither the transition associated with (s,a,s') nor any observations z associated with (s',a) are unreliable, $[T,(s,a,s'),*] \notin h \wedge [O,(s',a,*),*] \notin h$, and 1 otherwise. The parameter w specifies how much we want to avoid the unreliable parameters. A large w encourages the robot to find alternative paths that achieve the goal of the task without using the unreliable parts of the transition and observation functions. If not possible, the robot minimizes the use of unreliable parameters as much as possible. Clarification actions reduce the number of hypotheses, thus reducing the number of unreliable parameters that the robot might use along its route to the goal.

We can then solve this augmented POMDP model by using the goal POMDP approaches that we described in Chapter 2.2.2.

7.2.2 How to Compute the Hypotheses Set and the Clarification Actions?

When a discrepancy is detected, our algorithm investigates the model to compute a set of hypotheses that explain the discrepancy and a set of clarification actions associated with the hypotheses. We call this step discrepancy diagnosis. The hypotheses and the clarification actions are then given to the discrepancy reasoning step to be included in the augmented POMDP model.

In this step we look at the model and reason how the model can change so that the discrepancy can be accounted for. Similar to many other planning and execution monitoring approaches (even learning approaches), we assume that the state and observations spaces of the planning model are complete. The discrepancy could come up for the following reasons or a combination of the following reasons:

1. The robot has an imprecise model for the dynamics of the domain, *i.e.*, $T(s, a, s')$ is incorrect.
2. The robot has an imprecise model for how the robot's sensory observation correspond to its state, *i.e.*, $O(s', a, z)$ is incorrect.

If the current action a_t is not the reason for the discrepancy. The discrepancy can be the result of a previous action, a_1 through a_{t-1}. Given that the initial belief state b_0 that the robot started with is correct, this can happen because the transition model T and/or the observation model O associated with the actions taken in the previous steps are not accurate which lead to not having the current state in the belief state $b_{t-1}(s)$, *i.e.*, $b_{t-1}(s)$ was incorrect.

Hypotheses Set We reason about the model and find the modified transitions and observations (parameters) that when are nonzero result in a nonzero probability for the robot's history of actions and observations, *i.e.*, $\Pr(z_1, z_2, \ldots, z_t | b_0, a_1, a_2, \ldots, a_t) > 0$. We mark these parameters as unreliable, and call them *discrepancy parameters*.

Imagine a matrix representation for the transition and observation functions where each row in this matrix should sum to one. Modifying the value of the discrepancy parameters from zero to nonzero will also affect some other parameters that share the same row in the transition and observation functions. We mark the nonzero probabilities in the rows associated with the discrepancy parameters as also unreliable since the values for those would change when the zero parameters become nonzero. We call these parameters *unknown parameters*. We make a distinction between the *discrepancy parameters* and the *unknown parameters* as they are treated differently when we define the clarification action set. The unknown parameters can become zero as the result of the discrepancy. We call each potential set of the parameters, including both the discrepancy parameters and the unknown parameters, that can explain the discrepancy,

a hypothesis. At least one of these hypotheses explains the discrepancy. The set with both the discrepancy parameters and the unknown parameters is called *unreliable parameters* and is assigned a high cost in the cost function C that we discussed above.

The brute-force algorithm to generate the hypotheses set should consider making any of the zero-valued parameters or combinations of them nonzero and evaluating if it results in a nonzero probability for the robot's history of actions and observations, *i.e.*, if $\Pr(z_1, z_2, \ldots, z_t | b_0, a_1, a_2, \ldots, a_t) > 0$. Note that in general coming up with the hypotheses set can be very challenging as it depends on the whole history of actions and observations, and the number of the parameters in the planning model; however, certain assumptions and knowledge in the domain can decrease the number of hypotheses. We assume 1) the discrepancy is only associated with the current execution step, and 2) only a minimum number of changes should be applied to the planning model. Thus, our algorithm to generate the hypotheses set starts with making one change to the transition or observation functions associated with the current belief state, action and observation. The modifications that result in $\Pr(z_t | b_{t-1}, a_t) > 0$ are all valid hypotheses. If the algorithm cannot find any valid hypothesis, the algorithm considers changing all possible combinations of pairs of zero-valued parameters to nonzero and evaluating if $\Pr(z_t | b_{t-1}, a_t) > 0$. The goal is fully observable, so if the agent receives a discrepancy observation that it is at the goal, the execution can terminate as we are in a goal state.

Clarification Actions We augment the action space of the robot to include the clarification actions, $A' = A \cup A_{cl}$. To invalidate a particular hypothesis, the robot can ask the clarification questions that are associated with the hypothesis' unreliable parameters. The clarification actions help with invalidating a hypothesis, *e.g.*, if a parameter that was zero-valued in the original model and hypothesized to be nonzero in the augmented model is verified to be zero, or validating a hypothesis, *e.g.*, a parameter that was nonzero in the original model became zero and is verified to be zero.

A specific clarification action could ask if the unreliable parameters of a hypothesis are zero or nonzero. Alternatively, it could also ask about each and every unreliable parameter separately. In the latter case, a sequence of clarification actions can invalidate a given hypothesis. The former approach can only invalidate 1 hypothesis at a time, but the latter approach can invalidate 0 or multiple hypotheses at the same time. In our domains, since only one change to the transition function or the observation function is enough to address the discrepancy, the two approaches work in the same way. Here, we focus on the latter approach. There are two types of clarification actions:

- A zero-valued parameter from the original model should be nonzero.

The robot asks about the discrepancy parameters and if their value is nonzero. In the case of asking about the transition function, the robot asks the following question: "Is it possible to transition from x to y with the action a?". In the case of asking about the observation function, the robot asks the following question: "Is it possible to observe z in y with the action a?"

  - Yes: all the hypotheses for which this transition or observation is impossible become invalid.
  - No: all the hypotheses for which this transition or observation is possible become invalid.

- A nonzero-valued parameter from the original model should be set to zero.

  We ask about the unknown parameters. In the case of asking about the transition function, we ask the following question: "Is it **still** possible to transition from x to y with the action a?". In the case of asking about the observation function, we ask the following question: "Is it **still** possible to observe z in y with the action a?"

  - Yes: all the hypotheses for which the transition or observation is impossible become invalid.
  - No: all the hypotheses for which this transition or observation is possible become invalid.

7.2.3 Example Formulations

In this section, we describe how the discrepancy model is formulated for both the navigation and the restaurant domain. In these domains, to come up with a list of potential hypotheses, we assume that the discrepancy is only associated with the current execution step. Thus, the robot should only consider modifications to the transition and observation functions that are relevant to the current belief state, action and observation. We also assume that only minimum number of changes should be applied to the planning model.

Navigation Domain

Recall the example navigation domain from Section 7.1 where the robot observed "both" after executing the action north in the state 4 even though it expected to observe "left".

Discrepancy Diagnosis If we were making only one change to the transition function, changing the transition function such that $T(s'|s,a) = \Pr(4|4, north) > 0$ would explain the discrepancy.

This says that the robot might actually be able to transition to the state 4 by executing north from the state 4. The original transition function stated that if the robot is in the state 4, and it executes north, with 100% probability, it ends up in the state 0. If $\Pr(4|4, north) > 0$, then $\Pr(0|4, north)$ is either greater than 0 or equal to 0. Note that we make a distinction between the two cases as the latter makes the 4 to 0 transition untraversable and can invalidate a previously known feasible and optimal path.

We create two hypotheses, $h1$ and $h2$, associated with the cases that we mentioned above. The hypotheses $h1$ and $h2$ say that if we are in the state 4, and we execute north with $\beta\%$ probability, we end up in 0 and with $\alpha\%$ probability we end up in the state 4 ($\beta + \alpha = 100$ and $\beta, \alpha \leq 100$). The parameter α is the discrepancy parameter, and β is the unknown parameter. Hypothesis $h1$ considers that $\beta = 0$, and $h2$ considers that $\beta > 0$. If $\alpha = 0$ (discrepancy parameter), both of these hypotheses are invalid as both hypotheses rely on the discrepancy parameter; if $\beta = 0$, then only $h2$ is invalid. Two clarification actions are associated with these hypotheses. The first clarification action asks if α is nonzero, and the second clarification action asks if β is nonzero to invalidate the hypotheses.

Let's assume that in the domain that we illustrated in Fig. 7.1, the truth is that there is a thick carpet from the state 4 to 0 as shown in Fig. 7.2, and the robot did not move at all and will not be able to move if it keeps executing the action north ($h1$ is correct). The robot should take an alternative path to reach the goal by going south. However, it could also be possible that the carpet is thin and hence traversable with some probability, and the robot would be able to go to the state 0 after multiple executions of north ($h2$). Notice that in $h1$, going north will not in any way take the robot to the goal. However, in $h2$, after multiple executions of the north action (probabilistic transition), the robot will be able to reach state 0 and eventually reach the goal (although might not be optimal). Both of the hypotheses have doubts regarding the transition from 4 to 0. One says that it is not possible to transition from 4 to 0, and the robot stays in 4 (the route through the state 0 is infeasible). The second says that the robot can stay in 4 with some probability and transition to 0 with some other probability (the route through the state 0 is feasible).

If we were only modifying the observation function, changing the observation function such that $\Pr(both|0, north) > 0$ would explain the discrepancy. This claims that the robot might be able to observe "both" in the state 0. The observation function for the original model says that if the robot ends up in the state 0, with 100% probability it observes `left`. The hypotheses $h3$ and $h4$ say that if we end up in the state 0, with $\gamma\%$ probability we observe "left", and with $\delta\%$ probability we observe "both" ($\gamma + \delta = 100$). The parameter δ is the discrepancy parameter, and γ is the unknown parameter.

If $\delta = 0$, it means that $h3$ and $h4$ are invalid ($h1$ or $h2$ is valid). If $\gamma = 0$, $h4$ is valid, and if

both $\delta > 0$ and $\gamma > 0$, $h3$ is valid. We have two hypotheses regarding the observation function. One states that the robot ended up in the state 0, and it is possible to observe both "left" and "both" in the state 0. Another possible hypothesis is that the robot can observe only "both" in the state 0. These are the only two hypotheses associated with the observation function that can explain the discrepancy ($h3$ and $h4$). Two clarification actions are associated with these two hypotheses. The first clarification action asks if δ is nonzero, and the second clarification action asks if γ is nonzero to invalidate the hypotheses.

For this simple example, we have 4 hypotheses that might explain the discrepancy with only one change to the planning model (*i.e.*, one discrepancy parameter per hypothesis). There are 4 clarification actions associated with the 4 hypotheses. Each clarification action might be able to invalidate more than one hypothesis. We assume a uniform distribution over the possible hypotheses. Note that our goal is to achieve the goal of the task rather than learning the model parameters. Thus, rather than going for the exact values for the uncertain parameters to find an optimal path to the goal, we look for paths that avoid using those parameters by associating a high cost to using them. The robot can also take clarification actions to find the most plausible hypothesis. Given that the parameters will not match the parameters of the true model, optimality is not guaranteed, but achieving the goal is. We will prove the soundness and completeness guarantees of this formulation later.

Discrepancy Reasoning and Recovery Now that we have the hypotheses set and the clarification actions, let's see what the augmented POMDP model looks like. Originally the state space only included the position of the robot. We add the hypotheses to the state space, so the state vector becomes: $[pos, h1 - correct, h2 - correct, h3 - correct, h4 - correct]$. The $h1 - correct$, $h2 - correct$, $h3 - correct$, and $h4 - correct$ state variables specify which model is valid. Only one of these state variables could be 1. For example, if we believe that the robot is located in the state 4 and $h1$ is valid, then we believe we are in the following state $[4, 1, 0, 0, 0]$ with the probability 1. When a discrepancy happens, the robot does not know which of the 4 hypotheses is valid; thus, the belief state of the robot after the discrepancy becomes $([4, 1, 0, 0, 0], 0.25)$,$([4, 0, 1, 0, 0], 0.25)$,$([0, 0, 0, 1, 0], 0.25)$, $([0, 0, 0, 0, 1], 0.25)$ as we have a uniform distribution over the hypotheses. The first two hypotheses ($h1$ and $h2$) claim that the robot is located in the state 4 after executing the action north, and the second two hypotheses ($h3$ and $h4$) say that the robot is located in the state 0 after executing the action north.

Regarding $h1$ and $h2$, there are two questions that the robot can ask. One question asks if it is possible to go from the state 4 to the state 4 with the action north. If the answer is "yes", either $h1$ or $h2$ is valid. Hypotheses $h3$ and $h4$ still believe in the original transition function. They assume

that the agent ends up in the state 0 with the action north, so the answer "yes" to the question invalidates them. If the answer is "no" then either $h3$ or $h4$ is valid. The second question asks if it is still possible to get to the state 0 from the state 4 with the action north. If the answer is "yes", either $h2$, $h3$ or $h4$ can be valid, otherwise if the answer is "no", only $h1$ is valid. Similarly, we have 2 questions associated with the hypotheses $h3$ and $h4$. In total we have 4 clarification actions that can invalidate the hypotheses if necessary.

In addition to the above changes to the model, the robot changes the cost function to assign a high cost to the unreliable parameters (unreliable action and state pairs in the transition or observation function) if they are used during planning. In this cost function, if there is an alternative route that can take the robot to the goal, the robot might take that instead of taking any of the clarification actions. The robot's policy depends on the trade-off between the cost of taking the clarification actions or trying an alternative route. If there is no alternative route, the robot will try to determine the valid hypothesis by asking the clarification questions. This explanation concludes how we formulate the augmented POMDP model for the navigation domain. The augmented planning model can then be solved using the ILAO* algorithm [79] to compute a policy.

In the navigation example, we assign a cost of 100 to using the unreliable parameters (the unreliable action and state pairs) during planning. If the cost of each clarification action is 1, and in the ground-truth grid-world the robot cannot transition from the state 4 to 0, the robot follows the scenario below. The robot asks "is it still possible to go from the state 4 to 0 with the action north?". The oracle responds "no" and the belief state becomes $([4, 1, 0, 0, 0], 1.0)$. The robot realizes that $h1$ is valid, and it should go south as a result. In a different ground-truth grid-world, if the answer was "yes", the robot would have realized that $h1$ is invalid and any of the other hypotheses are a viable hypothesis, so the belief state would have become $[([4, 0, 1, 0, 0], 0.2), ([0, 0, 0, 1, 0], 0.4), ([0, 0, 0, 0, 1], 0.4)]$. In this case, the robot might have asked another clarification action, namely "is it still possible to observe left in the state 0 with the action north?".

Given the ground-truth environment in Fig. 7.2, the human answered "no" to the first question. If the cost of a clarification action was 4 rather than 1, the robot would have not taken a clarification action and would have taken the alternative route to reach the goal by going south. Since there is an alternative route that takes the robot to the goal, it doesn't even worth it to ask a question when the clarification actions are costly.

Grid-world domain - the first two planning steps after the discrepancy
Robot's initial belief: $([4,1,0,0,0],0.25),([4,0,1,0,0],0.25),([0,0,0,1,0],0.25)$ $,([0,0,0,0,1],0.25)$ **Robot's action:** is it still possible to go from the state 4 to the state 0 with the action north? **Robot's observation:** no ***Robot's updated belief:*** $([4,1,0,0,0],1.0)$ **Robot's action:** south **Robot's observation:** left

Restaurant Domain

Let's consider a slightly modified restaurant domain than the one we described in Chapter 3. Each table goes through a sequence of states from wanting the menu to paying for the meal and leaving. The customers level of `satisfaction` is not observable, and there are 2 satisfaction levels, namely `unsatisfied` and `satisfied`. There are 3 observations associated with the satisfaction level, `happy`, `unhappy` and `neutral`. If the customers are `unsatisfied`, the robot observes `unhappy` and `neutral` with probability 70% and 30%, respectively. Similarly, if the customers are `satisfied`, the robot observes `happy` and `neutral` with probability 70% and 30%, respectively. After each timely service (action execution), the customers' satisfaction level increases, but if the customers wait time is high, the customers' satisfaction decreases. In our examples, we assume that all the services are done within a time frame that does not decrease the customers' satisfaction level.

Now, let's consider the following examples. In both of these examples, the customers enter the restaurant and sit at a table. The robot brings the menu to the customers and get their orders. After both the action executions, the robot observes that the customers are happy.

Example 1 While the customers are waiting for their food, the robot brings them bread if it does not see bread on their table to keep them satisfied (and busy) while waiting. Given that the customers in our scenario have switched to the `waiting for food` state, the robot performs the `bring bread` action. The robot expects the following scenario, but it observes the scenario below it.

Restaurant setting - expected interaction with Table 0
Robot's initial belief: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them the menu **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and get their orders **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them bread **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them bread **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them bread **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and deliver their food **Robot's observation:** happy

Restaurant setting - actual interaction with Table 0
Robot's initial belief: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them the menu **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and get their orders **Robot's observation:** happy ***Robot's updated belief***: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them bread **Robot's observation:** unhappy ***Robot's updated belief***: *discrepancy happens as the expectation was that the robot ends up in the state* $satisfied$ *and observes happy*

The robot observes that the customers are `unhappy`. As the robot only expected that the customers should be satisfied (switch to the `satisfied` state), and hence `happy`, a discrepancy occurs. It is not immediately clear if the customers are unhappy because they do not want bread (*e.g.*, have an allergy to wheat) or they are having an argument regarding something else (so no change in the robot's policy). Maybe the customers are unsatisfied because they have an allergy to wheat, so the robot has to consider for this specific table, bringing bread actually makes the

customers unsatisfied rather than satisfied and plan accordingly.

Example 2 In this example, the robot delivers the customers' food and water (and observes `happy` after both action executions). While the customers are eating, the robot fills up their glasses if it sees that they do not have much water in their glasses (to keep them satisfied). The robot brings water, but it observes that the customers are unhappy. As the robot only expected that the customers should be satisfied (switch to the `satisfied` state), and hence `happy`, a discrepancy occurs. It is not immediately clear if the customers are unhappy because they do not want more water (*e.g.*, they are having a lunch business meeting in which people don't want to be interrupted frequently) or for some other reasons. If it becomes clear that the customers became unsatisfied because they received water, the robot should plan its next actions accordingly.

Notice that in both of these examples, one table has different preferences that do not match with the general model that works for all the other tables in the restaurant; thus, it should plan differently for that particular table to get the customers successfully through their dining experience. We will go through how we formulate the augmented POMDP model for Example 1's discrepancy scenario next. A similar formulation applies to Example 2.

Discrepancy Detection Following Example 1,
$\Pr(\text{unhappy}|\text{customers are satisfied \& waiting for food \& do not have bread}, \text{bring bread}) = 0.$

Discrepancy Diagnosis In Example 1, we assume that when the customers entered the restaurant, they were satisfied. There were 2 actions that the robot executed `bring menu` and `get orders`. After both the actions were executed, the robot observed `happy`. Thus, from the perspective of the robot, both the observations match the robot's model, and the robot updates its belief state that the customers are 100% satisfied. The customers could not be in an unsatisfied state in the previous action executions as the robot did not observe `unhappy` in any of those executions.

If we were making only one change to the transition function, changing the transition function such that $T(s'|s,a) = \Pr(\text{unsatisfied \& waiting \& no bread}|\text{satisfied \& waiting \& no bread}, \text{bring bread}) > 0$ would explain the discrepancy. This says that we might actually transition to the `unsatisfied` state by executing `bring bread` from the `satisfied` state. The transition function for the original model was if the table is in the `satisfied` state, and the robot brings bread for the table, with 100% probability, the table transitions to the `satisfied` state. The hypotheses $h1$ and $h2$ say that if the table is in the `satisfied` state, and the robot brings bread for the table, with $\beta\%$ probability, the table switches to the `satisfied` state and with $\alpha\%$ probability the table becomes `unsatisfied` ($\beta + \alpha = 100$ and $\beta, \alpha \leq 100$). If $\alpha = 0$, these

hypotheses are invalid. The hypotheses set and the clarification actions set for this case are very similar to what the robot came up with in the navigation domain.

If we were only modifying the observation function, changing the observation function such that $\Pr(\text{unhappy}|\text{satisfied \& waiting \& no bread}, \text{bring bread}) > 0$ would explain the discrepancy. This says that it is possible to also observe `unhappy` when the customers are `satisfied` (and they are waiting for the food and do not have bread). The observation function for the original model says that if we end up in a `satisfied` state by bringing bread, with 70% probability we observe `happy`, and with 30% probability we observe `neutral`. The hypotheses for this case say that if we end up in the `satisfied` state by bringing bread, with $\tau\%$ probability we observe `happy`, with $\delta\%$ probability we observe `neutral`, and with $\gamma\%$ probability we observe `unhappy` ($\tau + \delta + \gamma = 100$). If $\gamma = 0$, none of these hypotheses are valid. If $\gamma > 0$, there are 4 hypotheses associated with this case as both τ and δ can go from nonzero to zero.

There are 6 hypotheses and 6 clarification actions. One can assume a uniform distribution over the possible 6 hypotheses (each with 16% probability). Alternatively, since there are 2 hypotheses associated with the α discrepancy parameter, and 4 hypotheses associated with the γ discrepancy parameter, one can also assume a uniform distribution over the discrepancy parameters. This approach assigns 25% probability to $h1$ and $h2$, and 12.5% probability to $h3$ to $h6$. We use the latter method in this work.

Discrepancy Reasoning and Recovery In the restaurant setting, the original state space was $S = SR \times SC$ where SR denoted the robot's state space and SC denoted the table's state space. This particular table's state space now includes the hypotheses set, $SC' = SC \times MH$; thus the augmented state space is $S' = SR \times SC'$. This particular table's action space now includes the clarification actions, $A' = A \cup A_{cl}$. Since the augmented state and action spaces are specific to this particular table, our formulation maintains the independence between the tables.

We add the hypotheses to the state space, so the state vector becomes: $[original - state - variables, h1, h2, h3, h4, h5, h6]$. The $h1$ to $h6$ state variables specify which model is valid. The value of the original state variables are specified according to the different potential hypotheses. Similar to the navigation domain, the robot changes the cost function to assign a high cost to the unreliable parameters (unreliable action and state pairs in the transition or observation function) if they are used during planning. The augmented planning model is then solved using the ILAO* algorithm [79] to compute a policy. The discrepancy scenario for Table 0 is below. In the restaurant setting with multiple tables, the other tables' actions are interleaved with the Table 0's actions to effectively address all the tables. More specifically, in the interaction below since Table 0 does not want bread anymore, the robot can attend to the other tables in the meantime while Table 0's

food become ready.

Restaurant setting - actual interaction with Table 0 after the discrepancy
Robot's updated belief: $([\ldots, satisfied], 1.0)$ **Robot's action:** go to table 0 and give them bread **Robot's observation:** unhappy *A discrepancy happens, and the augmented POMDP model is built.* ***Robot's updated belief***: $([\ldots, unsatisfied, 1, 0, 0, 0, 0, 0], 0.25), ([\ldots, unsatisfied, 0, 1, 0, 0, 0, 0], 0.25),$ $([\ldots, satisfied, 0, 0, 1, 0, 0, 0], 0.125), ([\ldots, satisfied, 0, 0, 0, 1, 0, 0], 0.125),$ $([\ldots, satisfied, 0, 0, 0, 0, 1, 0], 0.125), ([\ldots, satisfied, 0, 0, 0, 0, 0, 1], 0.125)$ **Robot's action:** is it possible for the satisfied customers on Table 0 to become unsatisfied if the robot brings bread for them? **Robot's observation:** yes. ***Robot's updated belief***: $([\ldots, unsatisfied, 1, 0, 0, 0, 0, 0], 0.5), ([\ldots, unsatisfied, 0, 1, 0, 0, 0, 0], 0.5)$ **Robot's action:** is it still possible to make the customers on Table 0 satisfied, if the robot brings bread for them? **Robot's observation:** no ***Robot's updated belief***: $([\ldots, unsatisfied, 1, 0, 0, 0, 0, 0], 1.0)$ **Robot's action:** go to table 0 and deliver their food **Robot's observation:** happy

7.3 Efficient Planning on the Discrepancy Model

The size of the state and action spaces of the augmented POMDP model depends on the number of hypotheses which makes solving these models very challenging. In this section, we describe how the augmented POMDP model is solved efficiently. Before getting into the details of our approach, we describe our framework. In general solving the original planning problem optimally to assure the real-time performance of the robot in the restaurant setting is not feasible because of the complexity of POMDP models. Thus, in this chapter we focus on real-time search algorithms that interleave planning and action execution and guarantee that an agent's next action is selected within a hard prespecified time bound. These algorithms usually incorporate a learning mechanism that prevents infinite loops and guarantees that, under certain conditions, the agent will reach a goal despite limited planning. Planning usually consists of a lookahead phase, during which an expansion-bounded search generates the local search space around the current state, discovering

the optimal path from the initial node to all frontier nodes, and a learning phase, during which the heuristic values of the nodes within the local search space are updated based on the information obtained during lookahead. Action execution is simply the transitioning of an agent from node to node along the cost-minimal path discovered during the lookahead search.

7.3.1 Background on ILAO* Algorithm

Within the body of literature on real-time POMDP algorithms, we leverage the ILAO* algorithm described in [79] to solve the augmented POMDP model. LAO* is a heuristic search algorithm that can find optimal solutions for MDPs or POMDPs by expanding the reachable states using a greedy policy. The algorithm has three main steps: expansion, revision, and convergence test. In the expansion step, the algorithm expands some nonterminal tip state of the best partial solution graph and add any new successor states to the explicit graph G. In the cost revision step, it creates a set that contains the expanded state and all of its ancestors in the explicit graph by following the current best solution, and then performs dynamic programming (value iteration or policy iteration) on the set. For convergence test, it performs value iteration (or policy iteration) on the states in the best solution graph until the best current solution graph does not have an unexpanded tip state and the error bound falls below ϵ. LAO* shares the properties of AO* and other heuristic search algorithms. Given an admissible heuristic function, all state costs in the explicit graph are admissible after each step and LAO* converges to an optimal or ϵ-optimal solution without (necessarily) evaluating all problem states [79].

Improved LAO* (ILAO*) improves the performance of LAO* by using various techniques. One technique is to expand multiple states on the fringe of the best partial solution graph before performing the cost revision step. For certain problems, expanding all states on the solution fringe before performing the cost revision step worked better than expanding any subset of the fringe states [79]. They also found it helpful to limit the number of iterations of value iteration in each cost revision step because value and policy iteration are much more computationally expensive than expanding the best partial solution graph. Thus, ILAO* limits the number of state backups by integrating them into the solution expansion step. The algorithm is as follows. While the best solution graph has some nonterminal tip state, the algorithm performs a depth-first search of the best partial solution graph. For each visited state, if the state is not expanded, it gets expanded. Later, states of the greedy policy graph are backed up only once in the postorder when they are visited. ILAO* has the same convergence test as the LAO* algorithm. The full ILAO* algorithm from [79] is shown in Algorithm 8. The algorithm is originally designed for MDPs, but they can also be applied to belief MDPs (POMDPs).

Different from the original algorithm in [79] that runs the algorithm until convergence, we

Algorithm 8: ILAO* Algorithm from [79] on POMDP p and with a time limit tl.

ILAO* (b,p,tl)

The explicit graph G' asscoiated with p initially consists of the start belief state b.

while *time-limit* tl *is not reached* **do**

1. Expand best partial solution, update state costs, and mark best actions:
   While the best solution graph has some nonterminal tip state, perform a depth-first search of the best partial solution graph.
   For each visited state i, in postorder traversal:
   - If belief state i is not expanded, expand it.
   - Use the Bellman equation to set $V(i)$ and mark the best action for i.
2. Convergence test: Perform value iteration on the states in the best solution graph. Continue until one of the following two conditions is met.
   (1) If the best solution graph changes so that it has an unexpanded tip state, continue.
   (2) If the error bound falls below ϵ, break.

return the current solution graph.

only run the ILAO* algorithm until a time-limit is reached. To solve the original POMDP model, the robot follows the ILAO* algorithm to expand all the belief states on the fringe of the best partial solution graph, and then uses the domain-dependent heuristic function $H(s)$ where $s \in S$ to compute the frontier nodes' heuristic values. To compute the heuristic value of the frontier belief state b which is represented as a list of states and their probabilities, $(state, prob)$, the robot takes the minimum over the heuristic value of each state in the belief state b as shown in Eq. 7.1. For example, in the navigation domain, the heuristic values can be computed by using Euclidean distance, Manhattan distance or Dijkstra distance (considers obstacles). In the restaurant setting, we consider the number of steps until the customers leave the restaurant as our heuristic function.

$$H(b) = H((s_1, p_1), \ldots, (s_k, p_k)) = \min(H(s_1), \ldots, H(s_k)) \tag{7.1}$$

7.3.2 ILAO* on Discrepancy Model

We use the original ILAO* algorithm to solve the augmented POMDP model. The algorithm follows the ILAO* algorithm to expand all the belief states on the fringe of the best partial solution graph, and then uses a modified version of the domain-dependent heuristic function H called $H_{augmented}$ to compute the frontier nodes' heuristic values. Notice that the heuristic function $H(s)$ is initially defined over the state space of the original POMDP model, $s \in S$, and the new

heuristic function $H_{augmented}(s')$ is defined over the state space of the augmented POMDP model, $s' \in S'$, where the state includes both the original state variables and the hypotheses variables, $s' = [s, h]$. Similar to Eq. 7.1, we take the minimum over the heuristic value of each state s' in the belief state. To compute the heuristic value of each state, $[s, h] \in S'$, we exclude the hypotheses variables and use the original heuristic function defined over the original state space to obtain the heuristic values as shown in Eq. 7.2, $H_{augmented}(s') = H(s)$ where $s' = [s, h]$. This naive approach of computing the heuristic values is the baseline approach that we compare against.

$$H_{\text{augmented}}(([s_1, h_1], p_1), \ldots, ([s_k, h_1], p_k), ([s'_1, h_2], p'_1), \ldots, ([s'_m, h_2], p'_m), \ldots, ([s''_1, h_k], p''_1), \ldots, ([s''_n, h_k], p''_n)) = \min(H(s_1), \ldots, H(s_k), \ldots, H(s'_1), \ldots, H(s'_m), \ldots, H(s''_1), \ldots, H(s''_n)) \tag{7.2}$$

7.3.3 ILAO* with Hypothesis Decomposition on Discrepancy Model

The computational complexity of solving the augmented POMDP model depends on the number of hypotheses. As the number of the hypotheses increases, the number of the clarification actions, and consequently the branching factor of the augmented POMDP's graph also increases. For complex problems where there are a lot of potential hypotheses (*e.g.*, where the history of actions and observations should also be considered for diagnosis), it would be very challenging to solve the augmented POMDP model. We address this challenge in this section. Our key idea is to use the lower-bound on the value of the original planning problem under different hypotheses to compute better heuristic values. Our algorithm uses the original ILAO* algorithm on the augmented POMDP model for the expansion, revision and the convergence test steps of the algorithm, but computes better heuristic values to better guide the search. The baseline approach does not take into account the hypotheses when computing the heuristic values. Differently, we use the key idea that the hypotheses can be solved separately independently of one another to compute better lower-bounds on the cost to the goal. As shown in Eq. 7.3, we take the minimum over the lower-bound value of the POMDP model associated with each hypothesis to compute the heuristic values. Variable b denotes a distribution over the states of the original POMDP under hypothesis h_1, *i.e.*, s_1 to s_k. Similarly, b' and b'' represent a distribution over the states under hypothesis h_2 and h_k. $\underline{V}(b, h_1)$ denotes the lower-bound on the value of the belief state b under the hypothesis h_1. Similarly, $\underline{V}(b', h_2)$ and $\underline{V}(b'', h_k)$ denote the lower-bound on the value of the belief state b' and b'' under the hypothesis h_2 and h_k, respectively. Computing the lower-bound on the value of each hypothesis has a similar computational complexity as the original planning problem.

$$H_{\text{augmented}}(([s_1, h_1], p_1), \ldots, ([s_k, h_1], p_k), ([s'_1, h_2], p'_1), \ldots, ([s'_m, h_2], p'_m), \ldots, ([s''_1, h_k], p''_1), \ldots, ([s''_n, h_k], p''_n)) = \min(\underline{V}(b, h_1), \underline{V}(b', h_2), \ldots, \underline{V}(b'', h_k)) \tag{7.3}$$

To compute the heuristic values associated with each hypothesis, one can solve the original POMDP problem associated with each hypothesis, or use approximation methods which require a low computational effort. We generally prefer approximation methods since solving the original POMDP problem has a high computational complexity. In addition, the number of heuristic computations is proportional to the number of frontier nodes of the graph which can exponentially increase as we expand the graph; thus an approach that can compute a lower-bound on the value of the frontier nodes with a low computational effort is preferred. We are particularly interested in approximations that use the underlying MDP to compute lower-bounds on the value of the hypotheses. As we mentioned earlier, the ILAO* algorithm can be used to solve MDPs or POMDPs (belief MDPs). We compute lower-bounds on the value of each hypothesis by applying the ILAO* algorithm to solve the MDP model associated with the hypothesis. The choice between using the POMDP model associated with a particular hypothesis versus using the associated MDP model in general depends on the domain, but in our experiments and in the literature the approximate MDP model has been shown to be more effective [158].

Although it is easier to solve the original planning problem using the MDP approximation compared to solving the augmented POMDP model, but the size of the MDP model still prevents us from fully solving it. To be able to solve the augmented POMDP model in a real-time fashion, *e.g.*, less than 30 seconds planning time for each action execution, we introduce a parameter which limits the time dedicated to computing a heuristic value for each belief state.

To solve the augmented POMDP model, the ILAO* algorithm in Alg. 8 is applied on the augmented POMDP formulation, but rather than using the trivial heuristic computation in Eq. 7.2, our algorithm uses Alg. 9 to compute the heuristic values for the frontier nodes of the augmented POMDP's graph. Alg. 9 also runs the ILAO* algorithm, but on the original POMDP model associated with each hypothesis, and it uses the trivial heuristic computation, $H(s)$, for its frontier nodes. The `beliefs` and `hypotheses` variables are computed from the belief state of the augmented POMDP model (line 2). The states in this belief state can be divided based on which hypothesis they believe is valid. This will divide the belief state into disjoint sets associated with different hypotheses. The probabilities associated with the states in each set is then normalized to form the belief state for each hypothesis.

The algorithm goes over the hypotheses sequentially and runs one iteration of the ILAO*

algorithm which includes all the 3 steps (line 6), namely expansion, cost revision and convergence test. The algorithm keeps iterating over the hypotheses until the heuristic time-limit is reached (line 4). This approach ensures that we dedicate equal time to computing the value of each hypothesis. To compute a lower-bound on the value of the frontier states in each hypothesis' MDP graph (line 6), the algorithm uses the original heuristic function $H(s)$.

Algorithm 9: Compute Heuristic Values for A Belief State $b_{augmented}$ of the Augmented POMDP Model $p_{augmented}$

1 **ComputeHeuristicForAugmenetedTask** ($b_{augmented}$,$p_{augmented}$,htl)
2 (beliefs,hypotheses) $\leftarrow$ DivideHypotheses($b_{augmented}$,$p_{augmented}$)
3 $\underline{V} = \infty$
4 **while** *heuristic time-limit* htl *is not reached* **do**
5 **for** $(b, h) \in$ (*beliefs,hypotheses*) **do**
// ILAO* can be applied on the MDP or POMDP model associated with hypothesis h
6 $\underline{V}_h \leftarrow$ ILAO*(b,h,htl) // for the frontier nodes of this ILAO*, we use the original H function
7 $\underline{V} = \min(\underline{V}, \underline{V}_h)$
8 **return** $\underline{V}$

7.4 Efficient Planning for Achieving Multiple Independents POMDPs

In the multi-task settings such as the restaurant domain, the robot has to solve the agent POMDP model associated with the multiple tasks to attend to them all. When a discrepancy happens, the model associated with one task in augmented, and the Agent POMDP model should be built from this new augmented POMDP model and the rest of the tasks. In the previous chapter, we discussed how we leverage the independent tasks structure and the observation that the number of tasks that the robot can accomplish within a horizon is limited to expedite planning for multiple independent tasks. In this chapter, we focused on solving the augmented POMDP model associated with one task more efficiently. We now discuss how the ideas from the previous chapter and this chapter can be integrated to expedite planning for an agent POMDP that includes augmented POMDP models. In Chapter 6, we used a discounted reward POMDP formulation, but in this chapter we used a goal POMDP representation. It has been shown that goal POMDPs are actually more expressive than discounted reward POMDPs, and any discounted reward POMDP can be converted to a goal POMDP [26]. Thus, we first discuss how the algorithms in Chapter 6 can be modified to be applied on an agent POMDP model built from multiple independents tasks with a goal POMDP representation.

7.4.1 Multi-task Goal POMDP with Adaptive Horizon

Before getting into the details of the algorithm, we discuss how we formalize the client and agent POMDPs with a goal POMDP representation. Different from the discounted reward POMDPs that have either a maximum horizon H or a discount factor γ, the goal POMDPs have a set of goal states G. In addition, in the discounted reward POMDPs, the agent maximizes a reward function, but in the goal POMDPs, the agent minimizes a cost function (all positive costs). The other elements of the two types of POMDPs have the same representation (Chapter 2).

Client and Agent POMDPs

The client POMDP associated with each task is a tuple given by $(S_i, G_i, A_i, Z_i, T_i, O_i, C_i)$, where $S_i = SR \times SC_i$. $G_i = SR \times GC_i$ denotes a non-empty set of goal states where $GC_i \subseteq SC_i$. This defines a goal region for the task specific variables of task i. The goal states $g \in G_i$ are cost-free $C(g, a) = 0$, absorbing $T_i(g, a, g) = 1$ ($\forall a \in A_i$), and fully observable $g \in Z_i$, so that $O_i(s', a, g) = 1$ if $s' = g$ and $O_i(s', a, g) = 0$ otherwise. The agent's objective is to choose actions at each time step to minimize the expected cost to a goal. The other elements of the tuple are similar to the ones we defined for the discounted reward POMDP representation. The agent POMDP created from multiple client POMDPs with a goal POMDP representation for a domain with N tasks is represented by (N, S, G, A, Z, T, O, C) where $G = SR \times GC_1 \times GC_2 \times \ldots \times GC_N$ denotes a non-empty set of goal states for the agent POMDP. The other elements of this POMDP model are the same as the agent POMDP model with a discounted reward representation.

To address the restaurant setting, the robot can build an agent POMDP associated with all the tables in the restaurant and apply the ILAO* algorithm on the agent POMDP to find the next action to execute.

Proposed Multi-Task Method

We use a similar planning and execution framework as in Chapter 6. The main loop of the algorithm is in Alg. 10. As in Chapter 6, P is a set of POMDP models. During the planning phase, the algorithm computes the best action to execute in the current belief state given a time-limit (lines 3-6). In the execution phase, the robot executes the selected action (line 7), updates the belief state (line 8), and replans after each action execution.

Since we do not have access to a finite-horizon, the ideas that we used to solve the discounted reward POMDPs with an infinite-horizon are used in the undiscounted reward POMDP problems. In this section, we discuss how we modify multi-task-AH approach with $H = \infty$ from Chapter 6 and apply it here. Our key ideas are to use the truncated horizon and bound computations.

Algorithm 10: Online Planner with Adaptive Horizon

```
1 MultiTaskAdaptiveHorizonPlanner (env, P)
2     while not AllTasksDone() do
3         tpls ← InitializeTuples(P,2) // we start with pairs of tasks
4         while not timeout do
5             a,tpls,depth ← SelectAction(P,tpls)
6             tpls ← RecomputeTuples(P,depth,tpls) // ILAO* could go one level deeper
7         observations ← Step(env, a)
8         UpdateBeliefs(P,observations)
```

In the multi-task-AH approach, we used the truncated horizon to find the maximum number of tasks that the robot can achieve within the horizon. To solve the agent POMDP with a goal POMDP representation, we use the depth that the ILAO* algorithm reaches to find the maximum number of tasks that the robot can achieve within the depth. Different from the goal POMDP representation where the agent's goal is to minimize the cost function, an agent's goal in a discounted POMDP formulation is to maximize the reward function. Thus the upper-bound computation in a discounted reward POMDP computes a lower-bound in a goal POMDP. However, the lower-bound computation that we used for a discounted reward POMDP cannot be used to compute an upper-bound in a goal POMDP. We will explain the reasons in more details later.

Limited Depth In discounted reward POMDPs with an infinite-horizon, the number of tasks that the robot can attend to within $H = \infty$ is N, $k^* = N$ (Chapter 6.2.2). Similarly in the goal POMDP representation since we do not have access to a maximum horizon, the number of tasks that the robot can attend to is also N, $k^* = N$. Thus, different from the algorithm in Chapter 5 where we could limit the maximum number of tasks that the robot should consider within H to compute an optimal solution, all the N tasks should be considered in a goal POMDP representation. However, since we have access to a truncated depth or horizon, similar to the algorithm in Chapter 6, we leverage the observation that within the truncated depth d, the robot can only consider k tasks ($k \leq N$), and it performs *no ops* on the other tasks. The robot only considers combined models of size k till the truncated depth d while executing *no ops* on the other $N - k$ tasks, rather than a combined model of size N, but computes the lower-bounds using the solutions to the N individual tasks. We use the term depth instead of the horizon here as the depth that the algorithm reaches depends on how the ILAO* algorithm performs its expansion step. If it expands all the frontier nodes in each iteration, the depth increases by 1; however, the algorithm can only expand a subset of the frontier nodes which might not increase the depth by 1.

Alg. 11 shows the multi-task-AH algorithm. The function `InitializeTuples` considers

all possible subsets of size k. Each subset consists of two sets, tpl_c with size k and tpl_l with size $N-k$. The agent applies the ILAO* algorithm on the truncated agent POMDP built from the POMDPs in tpl_c while executing *no ops* on the POMDPs in tpl_l. However, the bound computations for the frontier nodes are done on all POMDPs in $tpl_u = tpl_c \cup tpl_l$ to assure valid lower-bounds on the value of the tuple tpl (`MultiTaskILAO*` function). The `SelectAction` function solves a truncated agent POMDP for each tpl to compute its bounds while executing *no ops* on other POMDPs that are not in tpl (line 7). It then updates the bounds on the value of the full agent POMDP (line 8). The size of the sub-problems is updated as the depth increases. Function `RecomputeTuples` updates the $tpls$ set as the number of tasks that the robot can attend to within the depth increases from k to $k' = k+1$.

Algorithm 11: Multi-task Goal POMDP Solver

1 **InitializeTuples** (P,k)
2 $\quad tpls' \leftarrow \{(tpl_c, tpl_l) : tpl_c \in \mathcal{P}(P), |tpl_c| = k, tpl_l = P \setminus tpl_c\}$
3 $\quad$ **return** $tpls'$
4 **SelectAction** (P,$tpls$)
5 $\quad depth = 0$; $\underline{V}_P = \infty$
6 $\quad$ **for** $tpl \in tpls$ **do**
7 $\quad\quad \underline{V}_{tpl}$,$d_{tpl} \leftarrow$ MultiTaskILAO*(tpl)
8 $\quad\quad \underline{V}_P = \min(\underline{V}_P, \underline{V}_{tpl})$
9 $\quad\quad depth = \max(depth, d_{tpl})$
10 $\quad a_{best} \leftarrow$ action from the tpl with lowest $\underline{V}_{tpl}$
11 $\quad$ **return** a_{best},$tpls$,$depth$
12 **RecomputeTuples** (P,d,$tpls$)
13 $\quad k$,$k' \leftarrow$ the maximum # tasks the robot can attend to within d and $d+1$;
14 $\quad tpls' \leftarrow tpls$
15 $\quad$ **if** $k \neq k'$ **then**
16 $\quad\quad tpls' \leftarrow \{(tpl_c, tpl_l) : tpl_c \in \mathcal{P}(P), |tpl_c| = k', tpl_l = P \setminus tpl_c\}$
17 $\quad$ **return** $tpls'$
18 **ComputeHeuristic** (tpl,htl)
$\quad$ // Variable htl is the time-limit on computing a lower-bound on the value of a task, $\underline{V}_p$
19 $\quad \underline{V} \leftarrow 0$
20 $\quad$ **for** $p \in tpl_u$ **do**
21 $\quad\quad \underline{V} \leftarrow \underline{V} +$ ComputeHeuristicForTask(p,htl)
22 $\quad$ **return** $\underline{V}$

Bound Computation The lower-bound is computed by assuming that the robot can address all the tasks in parallel. Since we only have one robot that sequences the tasks to achieve them all, this lower-bound computation is the minimum cost that the robot will get (`ComputeHeuristic`

function on line 18). We use the original ILAO* algorithm that we discussed in Section 7.3.2 to solve each task separately and compute a lower-bound on its value. Since each task has enough complexity that prevents us from solving it fully, we use the heuristic time-limit parameter to limit the time that is dedicated to computing a lower-bound value for each task (`ComputeHeuristicForTask` function on line 21). To compute a lower-bound on the value of the frontier belief nodes in each task's graph, the ILAO* algorithm uses a trivial heuristic function H that is available for each task.

Since there is no notion of a maximum horizon or a discount factor here, one cannot compute the upper-bound on the cost by selecting the best task to attend to while performing *no ops* on the other tasks as done in Chapter 6. *I.e.*, the blind policy of *no ops* has no notion of a goal. However, there is a computationally intensive method to compute an upper-bound. To compute the upper-bound, the robot could randomly select a sequence of tasks, run each task till its goal is achieved, and then solve the next task in the sequence. After all the tasks in the sequence are done, the sum of the costs from the different tasks is an upper-bound on the cost. Nevertheless, our focus in this chapter is on real-time planning, and the bound computations should require a very low computational effort. Thus, we do not use an upper-bound computation to prune the subsets of tasks as done in Chapter 6.

7.4.2 Solving the Augmented Agent POMDP model

We call the agent POMDP model with the goal POMDP representation with at least one augmented task, an *augmented agent POMDP*. The augmented agent POMDP is built from all the tasks, the augmented ones and the non-augmented ones. The augmented agent POMDP has all the properties of the agent POMDP model. The independence between the tasks is also intact as the augmentation process only affects the augmented tasks and does not affect any other tasks. Thus, Alg. 11 can be applied to the augmented agent POMDP. We integrate Alg. 11 with Alg. 9 to leverage both the task decomposition and the hypothesis decomposition. For this integration we simply replace the `ComputeHeuristicForTask` function (Alg. 11 on line 21) with the `ComputeHeuristicForAugmenetedTask` function (Alg. 9).

7.5 Evaluation

We first provide the proofs on why our approach is guaranteed to reach the goal, *i.e.*, is complete. We then discuss our experimental results on the navigation domain and the restaurant setting.

7.5.1 Properties of Planning on the Discrepancy Model

Definition 1 Given a POMDP model p represented by the tuple (S, G, A, Z, T, O, C), we say a POMDP model q represented by (S, G, A, Z, T', O', C') is a *valid approximation* of the true POMDP p iff the following holds:

1. $\forall s, s' \in S, \forall a \in A : T(s, a, s') > 0 \implies T'(s, a, s') > 0 \wedge T(s, a, s') = 0 \implies T'(s, a, s') = 0$. This condition says that if a particular parameter of T is zero, it should also be zero in T'. If a particular parameter of T is non-zero, it should also be non-zero in T'; however, their values do not need to exactly match. If we were to think about the transition function as a binary matrix where the non-zeros values are 1 and the zero values are 0, POMDPs p and q would have the same binary matrix.
2. $\forall s' \in S, \forall z \in Z, \forall a \in A : O(s', a, z) > 0 \implies O'(s', a, z) > 0 \wedge O(s', a, z) = 0 \implies O'(s', a, z) = 0$. This condition says that if a particular parameter of O is zero, it should also be zero in O'. If a particular parameter of O is non-zero, it should also be non-zero in O'; however, their values do not need to exactly match. Similar to the transition function, the binary matrix representing the observation functions of the POMDPs p and q are the same.
3. $\forall s \in S, a \in A : |C'(s, a) - C(s, a)| < \infty$. The cost function C' differs from C only by a finite cost.
4. For a given initial belief state b, $\forall h \in HT_p$ where HT_p is a set of all possible finite sequences of actions and observations executed (observed) so far for the POMDP p, $V_p^*(h) < \infty$ where ${V_p}^*$ is the optimal cost of reaching the goal from the history h. This says that for a given initial belief state b, for any history of actions and observations executed (observed) so far for the POMDP p, there is a policy that reaches a goal with probability 1.0 (proper policy).

In this section, we assume to have access to a planner that have soundness and completeness guarantees when applied on the true POMDP model p. We prove that when the same planner is applied on a valid approximate POMDP q, it would have soundness and completeness guarantees with respect to the POMDP p.

We use the concept of a policy tree in our proofs. In a POMDP, an agent can base its decisions on the history of its actions and observations. In a policy tree, nodes are actions, and edges are observations. The action at the root is called the root action. An action node has observation edges to actions at the next level. After an action is taken, the next action to take is one of the actions at the next level, depending on what the agent observes. Each policy tree is associated with a value function, which evaluates the long term reward (or cost) of taking the tree (policy).

Given an initial belief state, the optimal policy can be found by going through the set of all useful policy trees and finding the one whose value function is maximal (or minimal if we use a cost function) with respect to the initial belief state. All the sequences of actions and observations in this policy tree reaches a goal with probability 1.0.

Lemma 0) *Under Def. 1, given an initial belief state b and a history of actions and observations executed (observed) so far, a policy tree in the POMDP p also exists in the POMDP q and vice versa.*

Proof: As discussed in Chapter 2, the belief over the states at time t can be computed from an existing belief distribution over the states, b_{t-1}, an action a and observation z as follows where $\Pr(z|b_{t-1}, a)$ is the normalizing constant.

$$b_t(s') = \frac{O(z|s', a) \sum_{s \in S} T(s'|s, a) b_{t-1}(s)}{\Pr(z|b_{t-1}, a)} \tag{7.4}$$

In this equation for the POMDP p, we use T and O as the transition and observation functions respectively and for the POMDP q, we use T' and O' as the transition and observation functions respectively. Given the belief states b_0 and b'_0 associated with the POMDP p and POMDP q respectively, if $\forall s \in S : b_0(s) > 0 \implies b'_0(s) > 0 \wedge b_0(s) = 0 \implies b'_0(s) = 0$, for an action a and an observation z we show that the belief update in Eq. 7.4 maintains the following $\forall s \in S : b_1(s) > 0 \implies b'_1(s) > 0 \wedge b_1(s) = 0 \implies b'_1(s) = 0$ where b_1 is the updated belief state for the POMDP p and b'_1 is the updated belief state for the POMDP q.

Under Def. 1, T has the same binary matrix as T', and O has the same binary matrix as O'. Thus, under Def. 1 and Eq. 7.4, for an action a and an observation z, the following holds:

$$\begin{aligned}
&\forall s, s' \in S : \\
&b_0(s) > 0 \wedge T(s'|s, a) > 0 \wedge O(z|s', a) > 0 \implies b_1(s') > 0 \\
&b_0(s) > 0 \implies b'_0(s) > 0 \\
&T(s'|s, a) > 0 \implies T'(s'|s, a) > 0; O(z|s', a) > 0 \implies O'(z|s', a) > 0 \\
&b'_0(s) > 0 \wedge T'(s'|s, a) > 0 \wedge O'(z|s', a) > 0 \implies b'_1(s') > 0
\end{aligned} \tag{7.5}$$

$$
\begin{aligned}
&\forall s, s' \in S : \\
&b_0(s) = 0 \vee T(s'|s,a) = 0 \vee O(z|s',a) = 0 \implies b_1(s') = 0 \\
&b_0(s) = 0 \implies b'_0(s) = 0 \\
&T(s'|s,a) = 0 \implies T'(s'|s,a) = 0; O(z|s',a) = 0 \implies O'(z|s',a) = 0 \\
&b'_0(s) = 0 \vee T'(s'|s,a) = 0 \vee O'(z|s',a) = 0 \implies b'_1(s') = 0
\end{aligned}
\tag{7.6}
$$

Hence,

$$
\begin{aligned}
&\forall s, s' \in S : \\
&b_0(s) > 0 \wedge T(s'|s,a) > 0 \wedge O(z|s',a) > 0 \implies b_1(s') > 0 \\
&b_0(s) > 0 \wedge T(s'|s,a) > 0 \wedge O(z|s',a) > 0 \implies b'_1(s') > 0
\end{aligned}
\tag{7.7}
$$

$$
\begin{aligned}
&b_0(s) = 0 \vee T(s'|s,a) = 0 \vee O(z|s',a) = 0 \implies b_1(s') = 0 \\
&b_0(s) = 0 \vee T(s'|s,a) = 0 \vee O(z|s',a) = 0 \implies b'_1(s') = 0
\end{aligned}
$$

Thus, $\forall s' \in S : b_1(s') > 0 \implies b'_1(s') > 0 \wedge b_1(s') = 0 \implies b'_1(s') = 0$. This means that after the execution of the action a and getting the observation z, the following holds: $\forall s \in S : b_1(s) > 0 \implies b'_1(s) > 0 \wedge b_1(s) = 0 \implies b'_1(s) = 0$. The two belief states b_1 and b'_1 will have the same states with a non-zero distribution over them and any state that has a zero probability in b_1 will have a zero probability in b'_1. Using the same reasoning one can use a proof by induction method to show that after executing (observing) a sequence of actions and observations of length t, we would have $\forall s \in S : b_t(s) > 0 \implies b'_t(s) > 0 \wedge b_t(s) = 0 \implies b'_t(s) = 0$. Intuitively, this says that under Def. 1 and Eq. 7.4, after executing (observing) a sequence of actions and observations, both the POMDPs have the same set of states with a non-zero distribution over them; thus, they have the same action nodes and observation edges available to them.

We now prove why given an initial belief state b and a history of actions and observations of length t, a policy tree in the POMDP p also exists in the POMDP q. We just proved that after executing a sequence of actions and observations of length t, we would have $\forall s \in S : b_t(s) > 0 \implies b'_t(s) > 0 \wedge b_t(s) = 0 \implies b'_t(s) = 0$. The belief states b_t and b'_t are the updated belief states in the POMDPs p and q respectively. We now show a policy tree in the POMDP p with the belief state b_t also exists in the POMDP q with the belief state b'_t.

In Def. 1, we assumed that the cost functions in the POMDPs p and q only differ by a finite cost, and the two POMDPs have the same binary matrix for their transition and observation functions. We use a proof by induction method. For a given policy tree of depth 1 in the POMDP p, let's say that the policy tree has a_1 as its root action node, and the set Z_1 as a set of possible observation

edges. Since the following holds: $\forall s \in S : b_t(s) > 0 \implies b'_t(s) > 0 \wedge b_t(s) = 0 \implies b'_t(s) = 0$, there is a policy tree in the POMDP q which has the action a_1 as its root action node and the set Z_1 as a set of possible observation edges. Now, let's assume for a policy tree of depth t in the POMDP p, there exists the same policy tree of depth t in the POMDP q. We will now reason why for a policy tree of depth $t+1$ in the POMDP p, there exists the same policy tree of depth $t+1$ in the POMDP q. As we proved earlier, since the two POMDPs have the same non-zero distribution over their states, each observation edge at depth t in the POMDP p and its corresponding observation edge in the POMDP q have the same actions available to them. Let's call the action associated with one of these observation edges in the POMDP p, a_{t+1}. For this action node in the POMDP p and its corresponding action node in the POMDP q, the same set of possible observations exists in the two POMDPs' policy trees. Thus, given a policy tree of depth $t+1$ in the POMDP p, there exists the same policy tree of depth $t+1$ in the POMDP q. Similarly one can prove that a policy tree in the POMDP q also exists in the POMDP p.

Lemma 1) Soundness: *Under Def. 1, if POMDP q is a valid approximation of the POMDP p, a finite-cost solution from the belief state b'_0 in the POMDP q is also a finite-cost solution from the belief state b_0 in the POMDP p, where* $\forall s \in S : b_0(s) > 0 \implies b'_0(s) > 0 \wedge b_0(s) = 0 \implies b'_0(s) = 0$.

Proof: Without loss of generality, let's assume there is only one goal. If the POMDP q has a finite-cost solution to the goal from the belief state b'_0, there is a finite-cost policy tree in the POMDP q that reaches the goal with probability 1.0. *I.e.*, given the belief state b'_0, any history of actions and observations within this policy tree will always reach the goal. If a history of actions and observations does not reach the goal, the cost of that policy tree cannot be finite. Let's call this policy tree τ. Under Lem. 0, the same policy tree τ also exists in the POMDP p. Since the cost functions of the POMDPs p and q only differ by a finite cost, the policy tree τ also has a finite cost in the POMDP p. Thus, the POMDP p has a finite-cost solution to the goal. Likewise, if a policy tree has an infinite-cost in any of the POMDPs p or q, it will have an infinite-cost according to the other POMDP.

Lemma 2) Completeness: *Under Def. 1, a finite-cost solution from the belief state b_0 in the POMDP p is also a finite-cost solution from the belief state b'_0 in the POMDP q, where* $\forall s \in S : b_0(s) > 0 \implies b'_0(s) > 0 \wedge b_0(s) = 0 \implies b'_0(s) = 0$.

Proof: For every finite-cost solution in the POMDP p, there exits a finite-cost policy tree. Similar

to the proof for Lem. 1, the same policy tree also exists in the POMDP q, and it will have a finite-cost.

Lemma 3) *For a given initial belief state b, $\forall h \in HT_q$ where HT_q is a set of all possible finite sequences of actions and observations executed (observed) so far for the POMDP q, $V_q^*(h) < \infty$ where V_q^* is the optimal cost of reaching the goal from the history h. This says that for a given initial belief state b, for any history of actions and observations executed (observed) so far for the POMDP q, there is a policy that reaches a goal with probability 1.0.*

Proof: Under Lem. 0, given the same initial belief state b for the POMDPs p and q, any history of actions and observations in the POMDP p also exists in the POMDP q and vice versa, thus $HT_p = HT_q$.

Def. 1's condition 4 says that for any sequences of actions and observations in the POMDP p, $\forall h \in HT_p$, *e.g.*, $h = (a_1, z_1, a_2, z_2, \ldots, a_t, z_t)$, there is a finite-cost policy tree that reaches a goal with the probability 1.0. *I.e.*, all the sequences of actions and observations in this policy tree have a finite cost to the goal. After executing a sequence h, the agent will be in the belief state b_t in POMDP p and in the belief state b'_t in POMDP q such that $\forall s \in S : b_t(s) > 0 \implies b'_t(s) > 0 \wedge b_t(s) = 0 \implies b'_t(s) = 0$ (Lem. 0). According to Lem. 2, if there is a finite-cost policy tree in the POMDP p with the belief state b_t, the same policy tree also exists in the POMDP q with the belief state b'_t and will have a finite cost. Thus, for the same sequence of actions and observations h in the POMDP q, there is a finite-cost policy tree where all its sequences of actions and observations reach the goal with the probability 1.0. The same reasoning applies to the other members of HT_q.

Lemma 4) *A POMDP solver with completeness and soundness guarantees on any POMDP would have completeness and soundness guarantees with respect to the POMDP p when applied on the POMDP q.*

Proof: In Lem. 1, we proved that a finite-cost solution to the goal in the POMDP q is sound with respect to the POMDP p. In Lem. 2, we proved that a finite-cost solution to the goal in the POMDP p would have a finite cost in the POMDP q. Hence, if there is a finite-cost solution in the POMDP p, a POMDP planner with soundness and completeness guarantees should be able to find it when applied on the POMDP q.

Lemma 5 *Under Def. 1, planning on the Discrepancy POMDP model with a hypotheses set that*

includes a valid approximation of the true POMDP p, called POMDP q, is sound and complete.

Proof:

- **Soundness**: we use a proof by contradiction method. Let's assume that the planner found a solution to the goal by using the discrepancy POMDP model, but the solution has an infinite cost according to the POMDP p. Since the valid approximate POMDP q will always be a part of the augmented belief state with a non-zero probability associated with it, under Lem. 1, it would assign an infinite cost to the solution. Thus, the planner would not output that infinite-cost solution.
- **Completeness**: in Lem. 0 to 4, we proved that planning on the POMDP q is both sound and complete. Thus, if there exists a finite-cost solution in the POMDP p, applying the POMDP solver on the POMDP q will always find it. Now, if we prove that there will always be a finite-cost path from the augmented start belief state in the discrepancy model to a belief state where only the hypothesis associated with the POMDP q is valid, we can prove that there will always be a finite-cost solution from the augmented start belief state to a goal.

  Given the set of clarification actions with a finite cost, the planner can use them to invalidate all the invalid hypotheses and eventually validate the hypothesis associated with the POMDP q. This is a finite-cost path to a belief state where the POMDP q is only valid. In addition, since the clarification actions act as a *no op* action for the task, under Lem. 3, one can prove that for any history of actions and observations executed (observed), including the clarification actions (or *no ops*), there is a policy for the discrepancy model that reaches a goal. Thus, there will always be a finite-cost solution to a goal in the discrepancy model.

Lemma 6 *Eq. 7.2 and Eq. 7.3 compute admissible heuristic values if the initial heuristic values according to the original heuristic function H are admissible.*

Proof: If the initial heuristic values according to the original heuristic function H are admissible, Eq. 7.2 considers the heuristic value of the state (under any hypothesis) that is closer to the goal as its heuristic value hence it is a lower-bound on the cost to the goal (similarly Eq. 7.1 is an admissible heuristic).

If the initial heuristic values according to the original heuristic function H are admissible, Eq. 7.3 takes a minimum over the heuristic value of the hypotheses. Within each hypothesis, we can use the POMDP solution until a certain time-limit is reached and use the admissible function from Eq. 7.1 to compute the heuristic values of the leaves; this is certainly a lower-bound on the value of the hypothesis. Differently, one can use the MDP approximation until a certain

time-limit is reached and use the admissible function from Eq. 7.1 to compute the heuristic values of the leaves. Using the MDP approximation computes a lower-bound on the value of the goal POMDP associated with the hypothesis [158] (is an upper-bound on the value of a discounted reward POMDP). Since the function from Eq. 7.1 is admissible, overall the approach computes a lower-bound on the value of the hypothesis. Thus, Eq. 7.3 computes admissible heuristic values.

Lemma 7 *Using the ILAO* algorithm [79] to perform planning on the discrepancy POMDP model in an interleaved planning and execution fashion while using Eq. 7.2 or Eq. 7.3 to compute the heuristics is both sound and complete.*

Proof: We know that using the ILAO* to perform planning on any POMDP is sound and complete, and with an admissible heuristic is also optimal [79]. Thus, given Lem. 0 to 6, running the ILAO* planner until convergence on the discrepancy model is sound, complete and optimal.

We run the ILAO* with a time limit that when is reached, the current best solution is returned. We then execute the best action and replan using the updated belief. The ILAO* algorithm with a time-limit will not have the optimality guarantees but will remain sound and complete. This is because the ILAO* never overestimates the heuristic values if the initial heuristic values are admissible (Lem. 6), and repeated problem solving trials cause the heuristic values to converge to their exact values. Thus, under Lem. 5 where there is a finite-cost solution to a goal from any belief state (or history of actions and observations) in the discrepancy model, the ILAO* algorithm with a time limit has the same soundness and completeness guarantees as the original ILAO*.

7.5.2 Efficiency Analysis

We evaluate the performance of our algorithms on the navigation and the restaurant domains.

Navigation Domain

We use a larger version of the 2D navigation domain from Section 7.1 as shown in Fig. 7.4. The 2D domain has 44 states. The only actions available to the robot are "north", "south", "east" and "west". With any of these actions, the agent will take an step in that direction with probability 0.8 and with 0.1 probability in the states that are adjacent to the robot in that direction. If any of the transitions are not possible because of an obstacle, the robot stays in its place with that probability. An exception is that if the robot is in the state 4 and executes north, with the probability 1.0 it ends up in the state 0. Another exception is that if the robot is in the state 36 and executes any action, it can end up in states 3, 4, 5 and 6 each with the probability 0.25. The robot cannot go to the goal through the state 36. The robot gets the following observations regarding where the

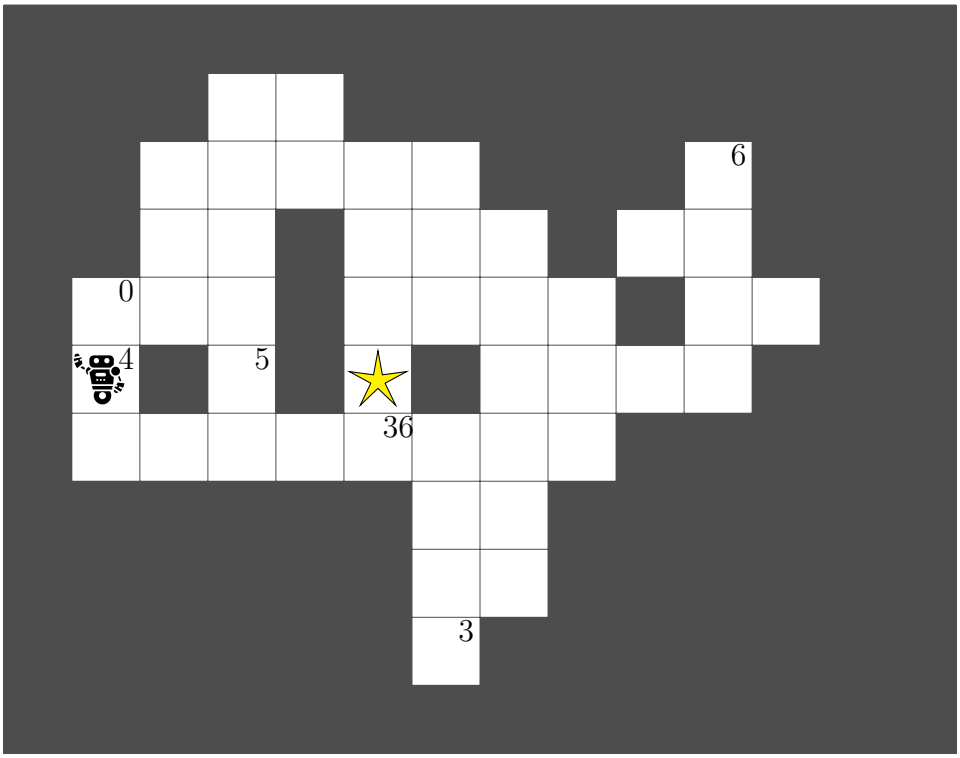


Figure 7.4: A 9×11 grid with the goal shown with a star, and the robot located at the state 4.

obstacles are: "left" (obstacle on the left), "right" (obstacle on the right), "neither" (both adjacent cells are empty), "both" (there are obstacles in both of the adjacent cells), and "goal". The goal is fully observable. The observation function does not depend on the action (only depends on the state) and has a probability of 1.0 for the true observation. The observation function assumes that the robot is pointing to the north always. For example, in the state 4, the robot observes "both", and in the state 0, it observes "left". The robot receives a cost of 0.4 with any action execution except if it executes an action in the goal state. The agent's true initial state is 4, but the initial belief state of the robot is that the robot can be in any state uniformly except the goal, the states 3, 5 and 6, and the adjacent states of the states 3, 5 and 6.

Similar to the small 3×4 scenario, in the planning phase the robot selects the action east and observes "both". Given the observation and the belief state, the robot can only be in the state 4. The planner then selects action north as it is parts of the optimal route to the goal. The robot then observes "both". If the robot is in the state 4 and executes the action north, with the probability 1.0 it should end up in the state 0. In the state 0, the robot can only observe "left". The observation "both" does not match with what is expected by the model, so a discrepancy happens. This example has the 4 hypotheses from the 3×4 grid example. However, the robot can also observe "both" in the states 3, 5 and 6, so it might be the case that the robot is teleported to any

of these states. Each of these states add two more hypotheses since we have to consider both a transition with probability 1.0 to one of these states and a transition with a non-zero and less than 1 probability to one of these states. Thus, in total, we will have 10 hypotheses associated with this discrepancy.

Experiment Setup

We call the whole process from when the robot starts its planning and execution process until it reaches the goal or a maximum number of steps *a planning episode*. We refer to each run of the planner to select an action to execute, *a planning session*. We design the grid environment after the discrepancy such that after a directional action is executed, the robot moves to the state with the highest probability of transitioning to and gets an observation associated with it. Any of the 10 hypotheses might be a valid explanation for the discrepancy. To run the experiments, before the start of the planning episode, we select a hypothesis from the hypotheses set as a valid explanation and call it a ground-truth hypothesis. Given the ground-truth hypothesis, we hard-code how the oracle should answer the robot's questions such that the hypothesis is valid. Note that in real-world, the oracle would answer the questions based on its own ground-truth hypothesis. For each algorithm and each ground-truth hypothesis (out of 10), we run 5 episodes each for 45 actions.

We consider running the algorithms with the following time limits (all in seconds), 1, 3, 5, 7.5, 10, 12.5, 15, 17.5, 20 , 25, 30, and the following clarification action costs, 0.2, 0.5, 1 and 2. For the algorithms that use a heuristic time-limit, we consider 0.005, 0.01 and 0.04 as the heuristic time-limits (all in seconds). The heuristic values are computed by using Dijkstra distance which is basically the cost to the goal in a deterministic setting while considering the obstacles. This is how we define the original heuristic function H that we referred to in the previous sections. We compare the following algorithms against one another:

- **ILAO***: We run the original ILAO* algorithm with a specific time-limit on the discrepancy POMDP model and use the original heuristic function H for heuristic computations (the algorithm in Section 7.3.2).
- **ILAO* with MDP heuristic**: We run the original ILAO* algorithm with a specific time-limit on the discrepancy POMDP model and use the solution to the MDP model associated with the discrepancy model for heuristic computations. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. After the time-limit is reached, for the rest of the heuristic computations, the original heuristic function H is used.
- **ILAO* with decomposed hypotheses and POMDP heuristic**: We run the original ILAO*

algorithm with a specific time-limit on the discrepancy POMDP model. For heuristic computations, we take the minimum over the values of the POMDPs associated with the hypotheses as described in Section 7.3.3. In this algorithm, the lower-bound value $\underline{V}$ for a hypothesis is computed by using the hypothesis' POMDP model. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. After the time-limit is reached, for the rest of the heuristic computations, the original heuristic function H is used.

- **ILAO* with decomposed hypotheses and MDP heuristic (our approach)**: We run the original ILAO* algorithm with a specific time-limit on the discrepancy POMDP model. For heuristic computations, we take the minimum over the values of the MDPs associated with the hypotheses as described in Section 7.3.3. In this algorithm, the lower-bound value $\underline{V}$ for a hypothesis is computed by using the hypothesis' MDP model. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. After the time-limit is reached, for the rest of the heuristic computations, the original heuristic function H is used.

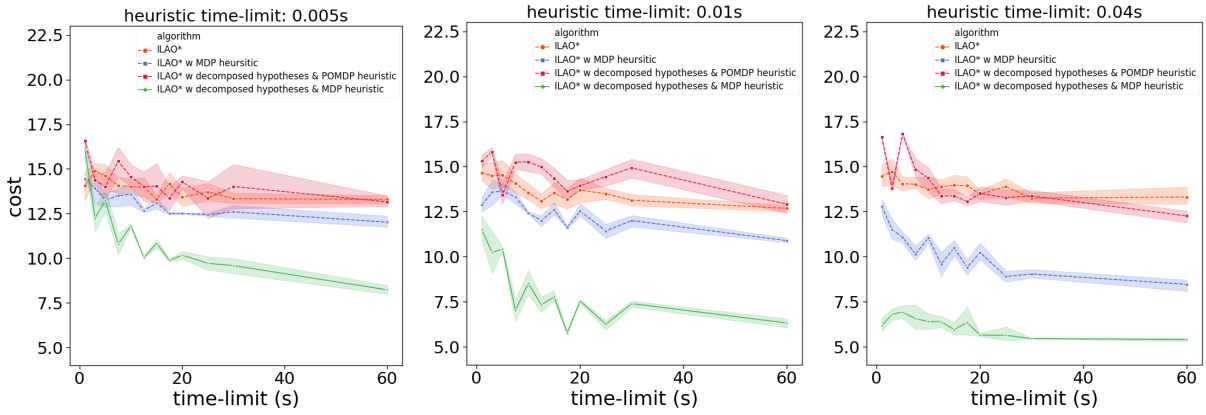
Results

We compare our method with different heuristic time-limits and clarification action costs against the baselines.

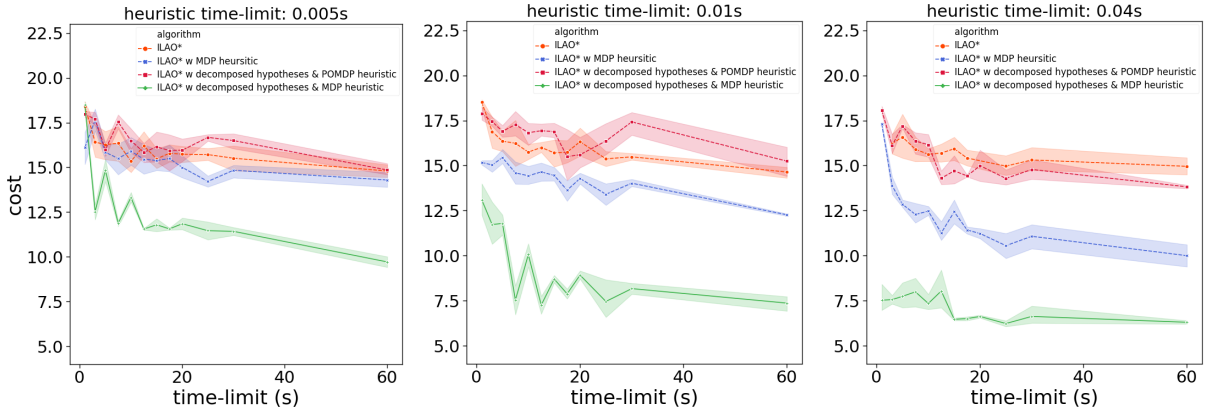
Average Cost We compute the cost of each episode by summing the action costs of each planning session. We then take the mean over the cost of the episodes associated with the 10 different hypotheses. We report the mean and error obtained by each algorithm over the 5 runs. Figs. 7.5a, 7.5b, 7.5c and 7.5d show the results for different clarification action costs. Our method obtains an average cost that is lower than all the baselines. The difference between the performance of our method versus the other methods is larger as the heuristic time-limit increases since there is more time to compute a better heuristic value. However, note that the heuristic time-limit can only increase until a certain point, after that the heuristic computation will prevent the robot from sufficiently searching the large graph associated with the discrepancy model. In our experiments, we noticed that the heuristic time-limit 0.1 does not perform well.

The variant of our approach with a POMDP heuristic computation mostly performs worse than all the algorithms, specifically, with smaller heuristic time-limits and smaller time-limits. This is because the original POMDP associated with each hypothesis has a high complexity, and the planner cannot explore it well enough to compute a good heuristic value. The heuristic computation takes some of the time that the planner could use to explore the large graph associated with the discrepancy model, and if the heuristic value is not very useful, spending computation

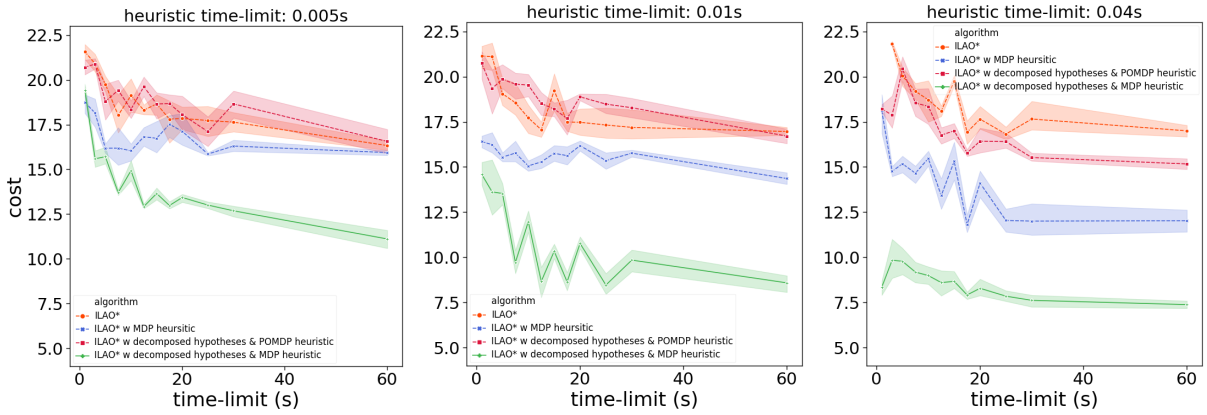
time on it will be worse than having a trivial heuristic function. When there is more time for heuristic computation, *e.g.*, for the heuristic time-limit 0.04 and the time-limits greater than $15s$, the decomposed hypotheses with a POMDP heuristic approach performs better than the original ILAO* approach. The ILAO* algorithm with the MDP heuristic computation mostly performs better than the original ILAO* without the MDP heuristic computation, but its performance significantly increases after we use our decomposed hypotheses approach. When the clarification action cost is larger, *e.g.*, when the cost is 2, the cost of a sequence of 5 directional actions is the same as the cost of one clarification action. Thus, the ILAO* algorithm should go deeper in the graph associated with the discrepancy model to decide if asking a question is beneficial or not. For the clarification action cost of 2 and the heuristic time-limit of $0.01s$, since the heuristic computation takes some of the time that the planner could use to explore the large graph associated with the discrepancy model, the gap between the performance of our method and the ILAO* method is smaller for the time-limits of less than $10s$. This is also the case for the ILAO* with the MDP heuristic approach.



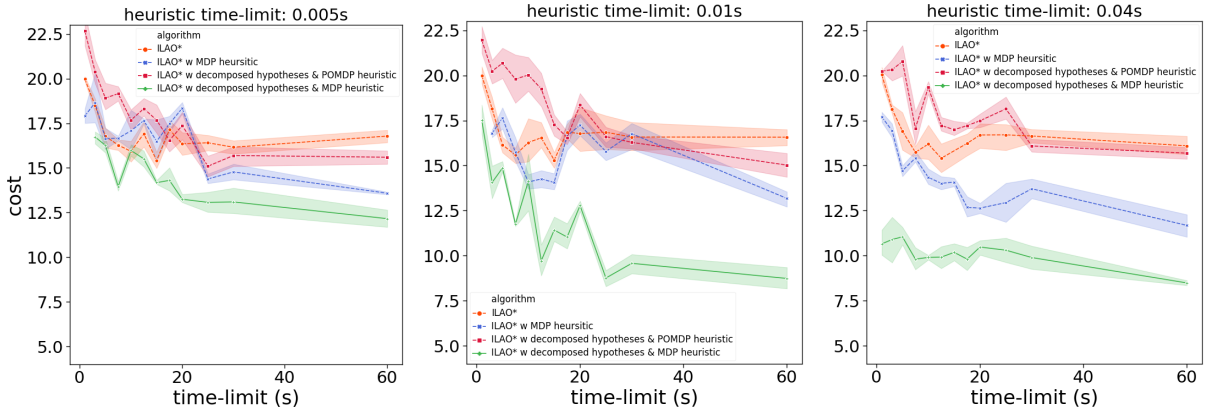
(a) Clarification action cost is 0.2.



(b) Clarification action cost is 0.5.



(c) Clarification action cost is 1.



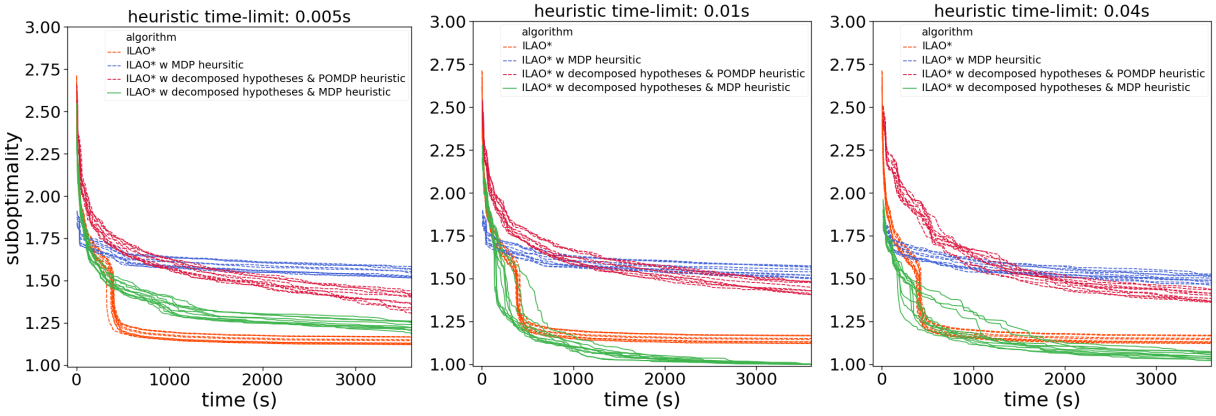
(d) Clarification action cost is 2.

Figure 7.5: Difference between the average cost of our method and the baselines for different time-limits and different heuristic time-limits.

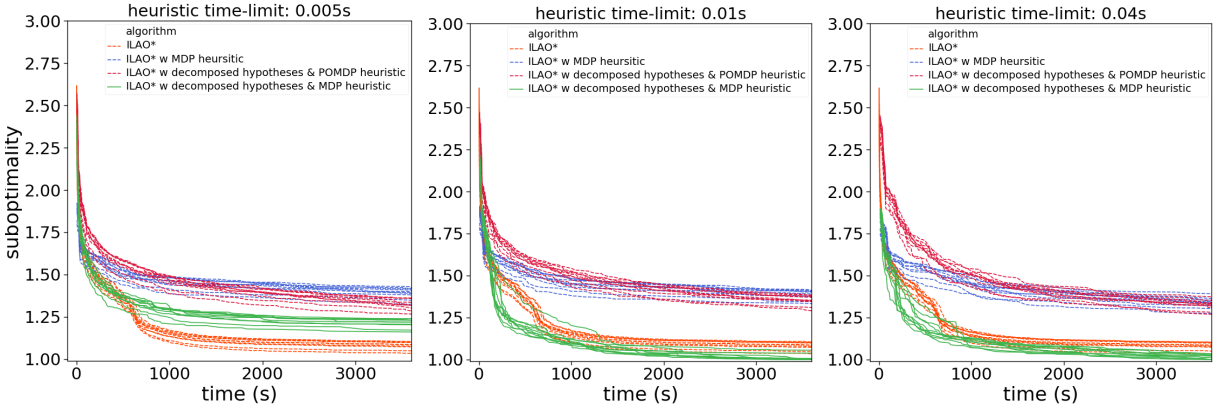
Suboptimality We report how suboptimal the different approaches are as a function of time. We give each planning session a time-limit of 1 hour for these experiments. Right after a discrepancy happens, the different algorithms and the different hypotheses all start from the same belief state. For each hypothesis and each algorithm, we compute the suboptimality at time t ($t < 3600s$) by dividing the optimal cost by the current lower-bound on the optimal cost. The closer this number is to 1, the tighter the lower-bound is to the actual cost to the goal; thus the heuristic value better guides the search. For the first planning step right after the discrepancy, since the belief states for the different algorithms and different hypotheses are all the same, the optimal action that the algorithms should take are the same and have the same cost; thus, the suboptimality cost is comparable across algorithms and hypotheses. However, for the steps after the first step, the different algorithms and the different hypotheses have different belief states, so different optimal costs are considered in computing the suboptimality. Hence, there is more variance in the

suboptimality, and they are not directly comparable. Although, we can still compare their rate of convergence to their optimal costs.

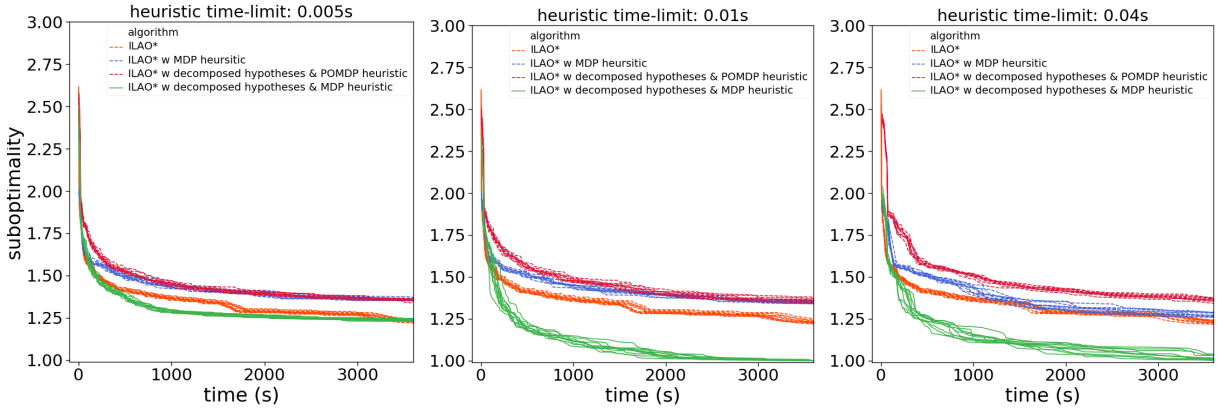
Figs. 7.6a, 7.6b, 7.6c and 7.6d show the results for the first planning session right after a discrepancy happens for different clarification action costs. The different lines with the same color are associated with the 10 different hypotheses. In most of the experiments, specifically, for the heuristic time-limits 0.01 and 0.04, our approach converges faster than the other approaches. However, for this planning step and smaller time-limits of less than $30s$, the ILAO* approach and the ILAO* with the MDP heuristic approach often perform better than our approach. In these cases, when the heuristic time-limit is 0.005 and 0.01, the ILAO* with the MDP heuristic and the ILAO* approach mostly perform better than our method, and when the heuristic time-limit is 0.04, all the three methods sometimes perform better than one another.



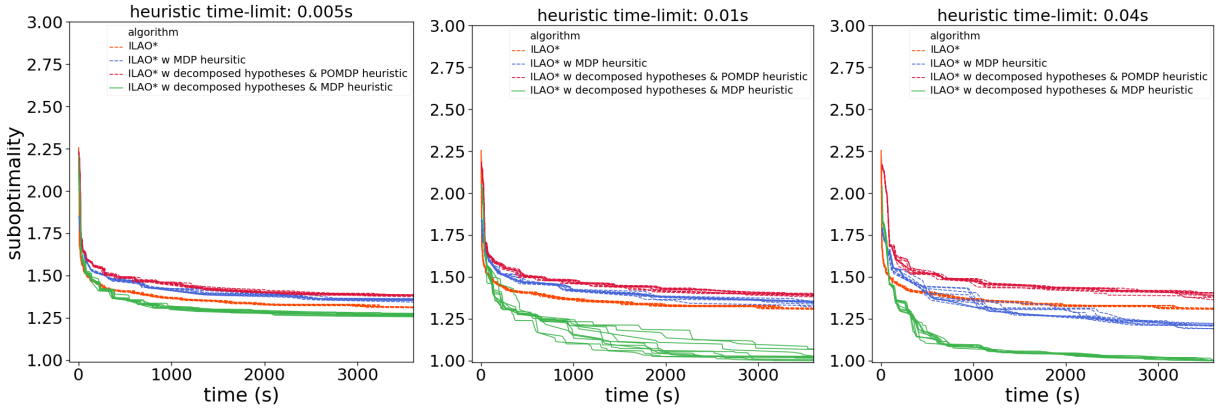
(a) Clarification action cost is 0.2.



(b) Clarification action cost is 0.5.



(c) Clarification action cost is 1.0.

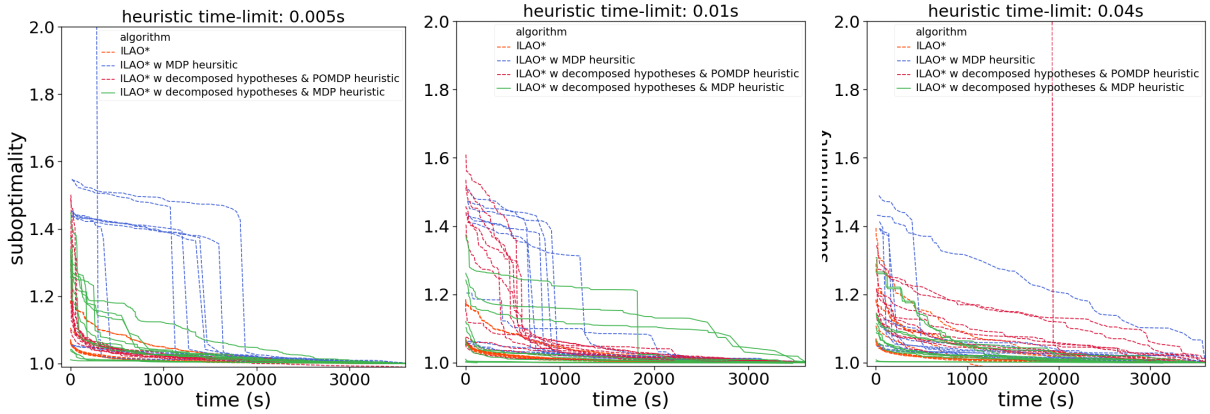


(d) Clarification action cost is 2.0.

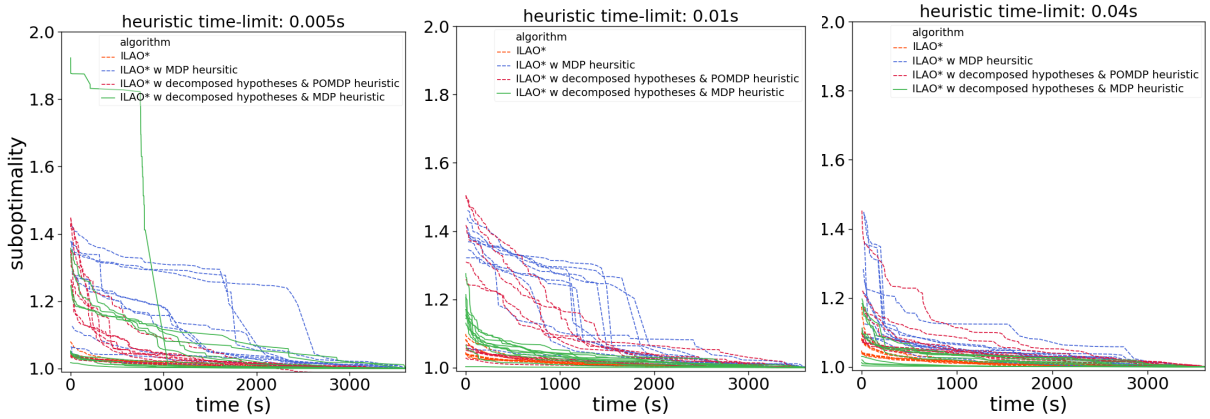
Figure 7.6: Difference between the suboptimality of the different algorithms in the first planning step after the discrepancy.

Figs. 7.7a, 7.7b, 7.7c and 7.7d show the results for the second planning session after a discrepancy happens for different clarification action costs. After the first planning step, depending on the hypothesis, different actions might have been selected by the different algorithms, so each hypothesis and algorithm might start from a different belief state and have different optimal costs. In this planning step, our method mostly performs better than the other methods. After our method, the ILAO* algorithm performs the best. In most of these cases, the ILAO* approach with the MDP heuristic mostly performs worse than the other approaches. We also observe similar results in the third planning step. We believe this is because right after the discrepancy, asking the clarification actions matters a lot in finding a hypothesis that best describes the discrepancy; however, the clarification actions are less important for the steps after. The way we compute the heuristics in our approach does not include the clarification actions, but the heuristic computation in ILAO* with the MDP heuristic includes them. For the steps after the first step, the latter

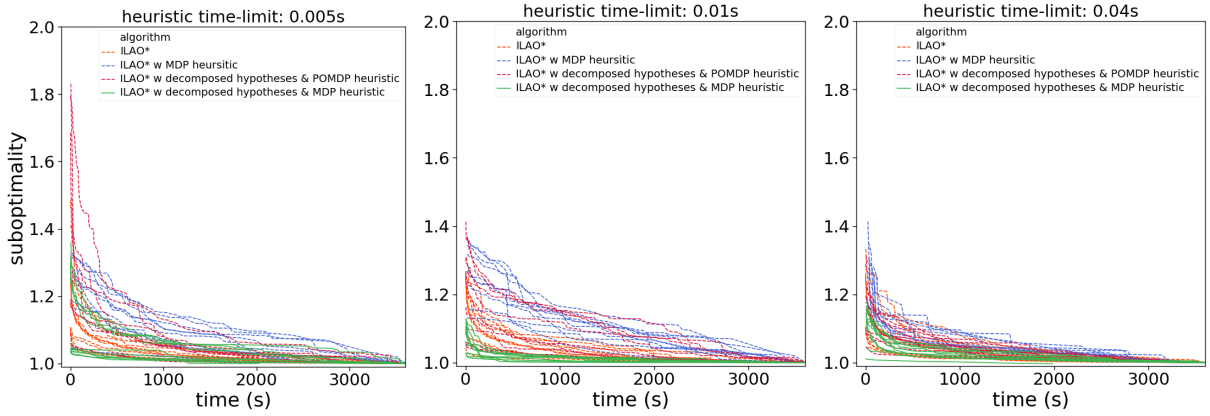
expands a graph with a higher branching factor for heuristic computations (more actions) and computes a less useful heuristic value. Thus, on average over all the planning steps our approach has a better performance than the other approaches.



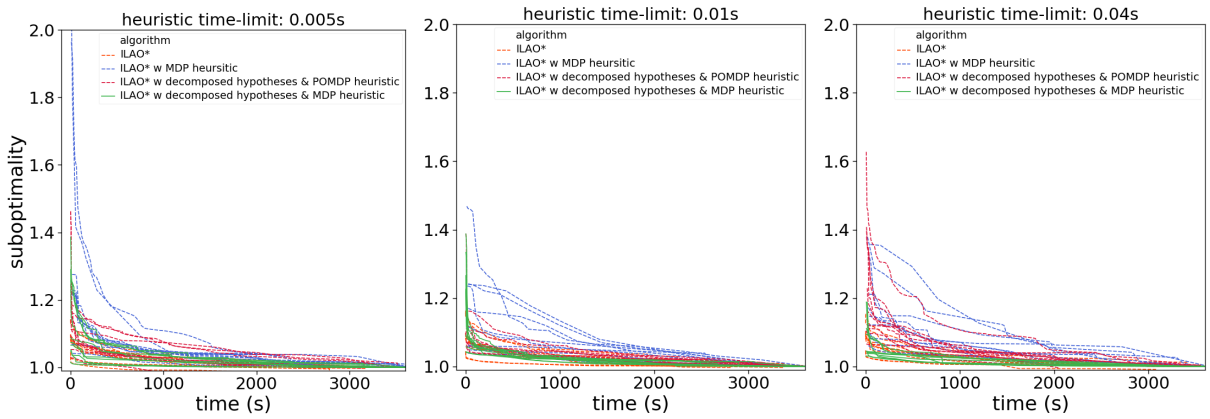
(a) Clarification action cost is 0.2.



(b) Clarification action cost is 0.5.



(c) Clarification action cost is 1.0.



(d) Clarification action cost is 2.0.

Figure 7.7: Difference between the suboptimality of the different algorithms in the second planning step after the discrepancy.

Restaurant Domain

We use a slightly modified version of the restaurant domain from Chapter 3. We provide the differences here. Each table goes through a sequence of states from wanting the menu to paying for the meal and leaving. Any state associated with the current request $= 8$ and hand raise $= 0$ is a goal state. This means that the customers have left, and their table is clean. Similar to the original version of the restaurant setting, the customers level of `satisfaction` is not observable, and there are 6 satisfaction levels, namely `very unsatisfied`, `unsatisfied`, `a bit unsatisfied`, `neutral`, `satisfied`, and `very satisfied`. There are 3 observations associated with the satisfaction level, `happy`, `unhappy` and `neutral`. If the customers are `very unsatisfied`, the robot observes `unhappy` with probability 100%. Similarly, if the customers are `very satisfied`, the robot observes `happy` with probability 100%. If the customers are `unsatisfied`, the robot observes `unhappy` and `neutral` with probability 90% and 10%, respectively. These probabilities differ if the customers are `a bit unsatisfied`. If the customers are `a bit unsatisfied`, the robot observes `unhappy` and `neutral` with probability 70% and 30%, respectively. Differently, if the customers are `neutral`, the robot observes `neutral` and `happy` with probability 70% and 30%, respectively. If the customers are `satisfied`, the robot observes `neutral` and `happy` with probability 30% and 70%, respectively. In the domain from Chapter 3, there was only one serve action that would have different outcomes based on the table's *current request*, but in this domain, the robot has one serve action for each value of the *current request*. Thus, there are 8 serve actions in this domain. Similar to the example in Chapter 7.2.3, we have an additional action called `bring bread`. While the customers are waiting for their food, the robot brings them bread if it does not sees

bread on their table to keep them satisfied (and busy) while waiting. In the model, whenever the customers receive bread their satisfaction level is set to `very satisfied`. The cost function is as follows where the function $action\_cost(a)$ returns the cost of the action a as specified by the $action\_cost\_goto$, $action\_cost\_clarification$, and $action\_cost\_noop$ variables:

$time = min(\text{time since request}, 10)$

$action\_cost\_noop = 1; action\_cost\_goto = 1; action\_cost\_serve = 2; action\_cost\_clarification = 3$

$C(s, \text{serve}) = action\_cost\_serve$

$$C(s, a = \text{other actions}) = action\_cost(a) + \begin{cases} 2^{time} & sat' = 0 \wedge sat' <= sat \\ 1.7^{time} & sat' = 1 \wedge sat' <= sat \\ 1.4^{time} & sat' = 2 \wedge sat' <= sat \\ 2 * (sat_{max} - sat' + 1) & sat' = 3, 4 \wedge sat' < sat \\ 0 & \text{otherwise} \end{cases}$$

Given the scenario from Chapter 7.2.3, the customers are in the `waiting for food` state and the robot believes that the customers are `satisfied` with probability 100%. The robot performs the `bring bread` action and observes that the customers are `unhappy`. As the robot only expected that the customers should be very satisfied, and hence `happy`, a discrepancy occurs. Similar to the previous scenario, there are 4 hypotheses associated with the current discrepancy. However, since the robot can also observe `unhappy` when the customers are `unsatisfied` and `a bit unsatisfied`. The transition to each of these states adds 2 additional hypotheses. In total, there are 8 hypotheses associated with this discrepancy.

Experiment Setup

In the experiments, for a given satisfaction level, we consider the emotion with the highest probability as the observation. We consider the highest satisfaction level in the belief state that matches the ground-truth hypothesis as the customers current satisfaction level. Any of the 8 hypotheses might be a valid explanation for the discrepancy. To run the experiments, before the start of the planning episode, we select a hypothesis from the hypotheses set as a valid explanation and call it a ground-truth hypothesis. Given the ground-truth hypothesis, we hard-code how the oracle should answer the robot's questions such that the ground-truth hypothesis is valid. For each algorithm and each ground-truth hypothesis, we run multiple episodes each for 30 actions. For 1 table, we run 5 episodes. For 2 to 7 tables, we run 15 episodes, and for 8 to 12 tables, we run 35 episodes. We use the same restaurant configuration across the different algorithms and runs.

We consider running the algorithms with the following time limits (all in seconds), 5, 10, 15, 20, 25, 30, 40, 50, 75 , 100, 125, 150, 175 and 200. For the algorithms that use a heuristic time-limit, we consider $0.01s$ as the heuristic time-limit (in seconds); *i.e.*, the heuristic computation for each task can only take $0.01s$. In all the algorithms, we compute the value of a leaf belief node by summing over the lower-bounds on the values of the tasks. We consider the number of steps until the customers leave the restaurant as our heuristic function. Notice that in presence of multiple tables, this trivial heuristic function is a lower-bound on the cost to a goal since it ignores having multiple tables to attend to. We compare the following algorithms against one another:

- **Agent ILAO***: We run the original ILAO* algorithm with a specific time-limit on the agent POMDP model built from the discrepancy POMDP model and all the other POMDP models. We use the original heuristic function H for computing the heuristic values for each task as described in Section 7.3.2 (Eq. 7.2).
- **Agent ILAO* with MDP heuristic**: We run the original ILAO* algorithm with a specific time-limit on the agent POMDP model. We use the solution to the MDP model associated with each task for heuristic computations. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. For the rest of the heuristic computations beyond the heuristic time-limit, the heuristic function H is used.
- **Agent ILAO* with decomposed hypotheses and MDP heuristic**: We run the original ILAO* algorithm on the agent POMDP model. To compute the heuristic value for each task, we take the minimum over lower-bound on the value of the MDP associated with each hypothesis as described in Section 7.3.3 (Eq. 7.3). For the rest of the heuristic computations beyond the heuristic time-limit, the original heuristic function H is used.
- **Multi-task ILAO***: We run Alg. 11 and use the original heuristic function H for computing the heuristic values for each task as described in Section 7.3.2 (Eq. 7.2).
- **Multi-task ILAO* with MDP heuristic**: We run Alg. 11 and use the solution to the MDP model associated with each task for heuristic computations. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. The heuristic computations are performed until a time-limit is reached as specified by the heuristic time-limit parameter. For the rest of the heuristic computations beyond the heuristic time-limit, the original heuristic function H is used.
- **Multi-task ILAO* with decomposed hypotheses and MDP heuristic (our approach)**: We run Alg. 11 and take the minimum over the lower-bound on the value of the MDP associated with each hypothesis for heuristic computations as described in Section 7.3.3 (Eq. 7.3). The heuristic computations are performed until a time-limit is reached as specified

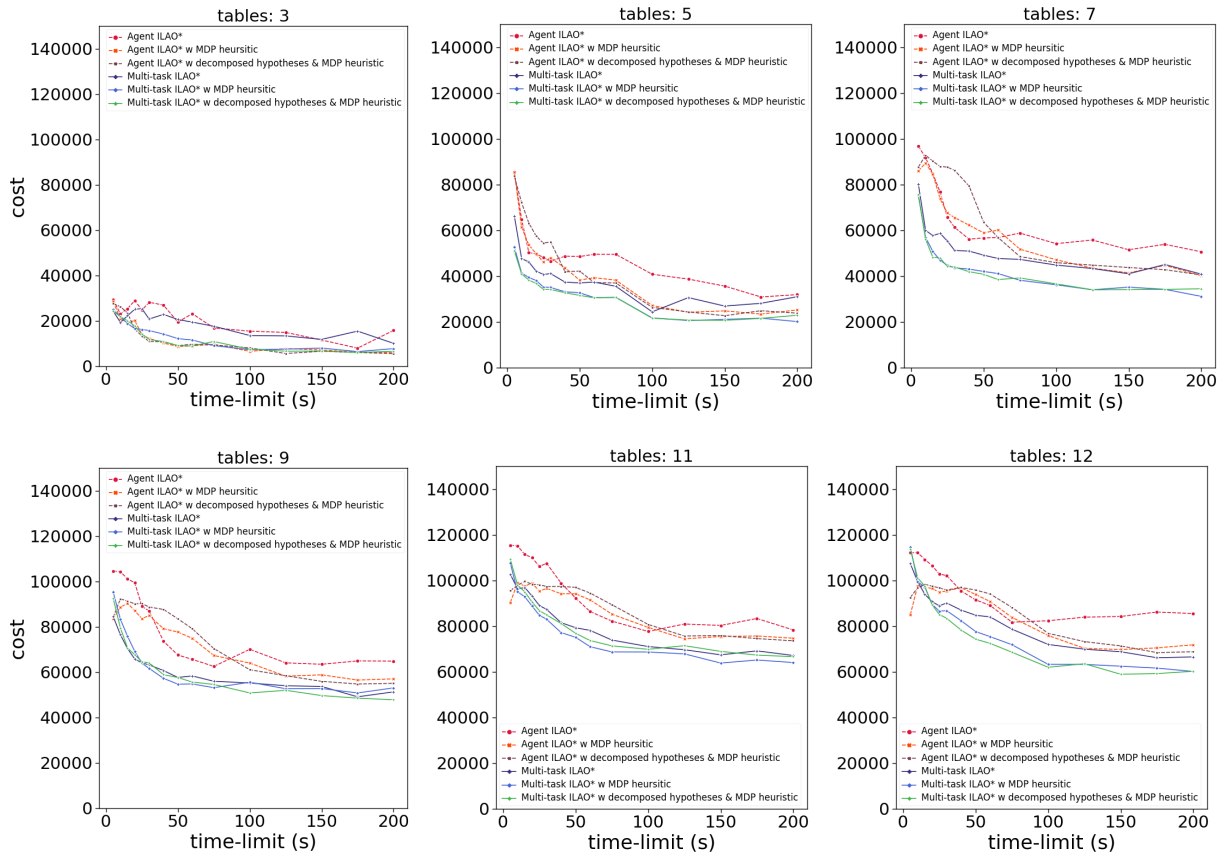


Figure 7.8: Difference between the average cost of the different algorithms in a restaurant with different number of tables.

by the heuristic time-limit parameter. For the rest of the heuristic computations beyond the heuristic time-limit, the original heuristic function H is used.

Results

We compare our method with different number of tables against the baselines. The average cost for an algorithm in one episode is computed as follows $c_{avg} = \sum_{t=0}^{30} \sum_{i=0}^{N} \frac{\sum_{s \in S_i} b_{i,t}(s) C_i(s, a_t)}{|a_t|}$ where $|a_t|$ is the length of the selected action at time t and N is the number of tasks. We then take the mean over the cost of the episodes associated with the 8 hypotheses. We report the mean and error obtained by each algorithm over the multiple runs. Fig. 7.8 shows the average cost that the different algorithms obtain for different number of tables as we increases the time-limit. The top 3 agent ILAO* approaches are plotted with a warm color and the multi-task approaches that use Alg. 11 are plotted with a cool color. For readability of the plots, we only plot the mean cost over the multiple runs.

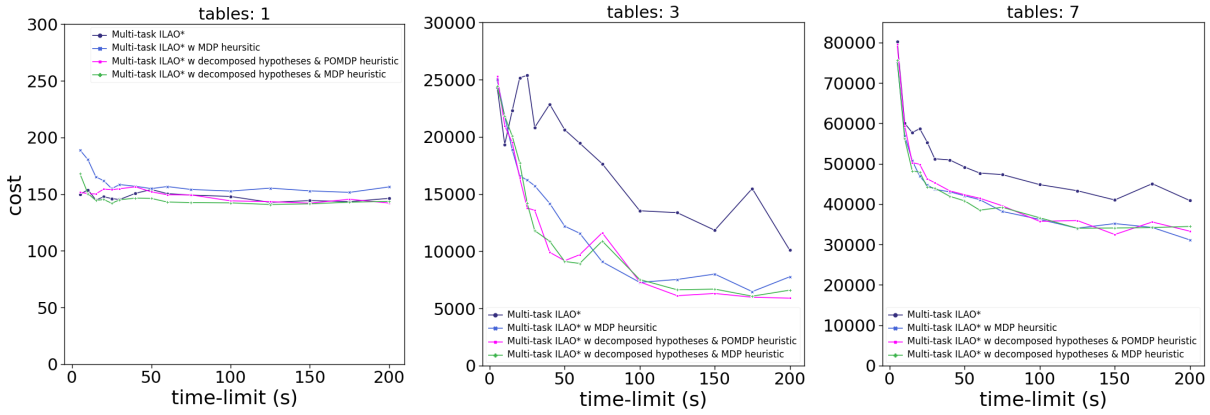


Figure 7.9: Difference between the average cost of the different multi-task algorithms in a restaurant with 1, 3 and 7 tables.

For 3 tables, the performance of the multi-task ILAO* is slightly better than the performance of the agent ILAO*, but as the number of tables increases, the multi-task ILAO* performs much better than the agent ILAO* approach. For larger number of tables, the multi-task ILAO* approach even performs better than the agent ILAO* approaches that use the MDP heuristic. This signifies the importance of using the independent tasks structure and the limited depth.

Within the different variants of the agent ILAO* algorithm, for 3 tables, the algorithms that use the MDP heuristic perform much better than the agent ILAO* algorithm. The agent ILAO* with MDP heuristic performs better than the other two algorithms for smaller number of tables ($tables < 6$) and smaller time-limits ($tl < 50s$). The agent ILAO* with decomposed hypotheses and MDP heuristic performs better than the other two algorithms for smaller number of tables ($tables < 6$) and larger time-limits ($tl > 50s$). The two algorithms that use an MDP heuristic perform the same for larger number of tables ($tables > 6$) and larger time-limits, and they perform better than the agent ILAO* algorithm in those settings.

Within the different variants of the multi-task ILAO* algorithm, the two algorithms that use an MDP heuristic mostly perform better than the multi-task ILAO* algorithm. The difference between the performance of these two algorithms and the multi-task ILAO* algorithm is larger for smaller number of tables ($tables < 8$), but for larger number of tables this gap becomes smaller. We believe this is because for larger number of tables the two algorithms take more time to compute the heuristic values for all the tables, and the algorithms spend less time on the main search. The performance of the two approaches are very similar in a restaurant setting with different number of tables and different time-limits. We believe this is because 6 out of the 8 hypotheses claim that the customers become unsatisfied as a result of the bring bread action, so in all those cases, the robot's policy is to not bring bread for the customers. Thus, solving all those 8

individual hypotheses to compute a heuristic value is unnecessary and not that helpful compared to solving one MDP model associated with all of them.

Note that in a restaurant with a large number of tables ($tables \geq 8$), other than the augmented table, there are more than 7 other tables that the robot should attend to. In such settings, attending to that one augmented table might not even be on the robot's list of priorities. Thus, it does make sense if the two multi-task algorithms that use an MDP heuristic mostly perform the same. Fig. 7.9 supports this hypothesis. For one augmented table, the performance of the ILAO* with decomposed hypotheses and MDP heuristic is better than the ILAO* with MDP heuristic algorithm, but as the number of tables increases, the two algorithm become very similar in performance. The performance of the ILAO* with decomposed hypotheses and POMDP heuristic is also very similar to our approach since reasoning about the observations do not modify the robot's policy drastically in this domain.

7.6 Conclusion and Discussion

In this chapter, we discuss how we formulate the discrepancy between the robot's observation and its model as a planning problem over an augmented model. The augmented model includes a set of hypotheses regarding the discrepancy and a set of clarification actions targeted at an oracle to invalidate the hypotheses. We then provide an approach that solves the augmented POMDP model more efficiently by using the key idea that the hypotheses can be solved separately independently of one another to compute better heuristic values. Our evaluation on a navigation domain shows that our approach significantly performs better than the other approaches. Finally, we integrate this approach with the efficient planning approaches that we developed in Chapter 6 to expedite task planning for multi-task settings while addressing the discrepancies effectively. We evaluate our algorithms on the restaurant setting and observe that the multi-task approaches perform better than the approaches that solve the combined model. Within the multi-task approaches, the approaches that use an MDP heuristic perform better. In this domain, our multi-task approach with the decomposed hypotheses mostly performs similarly to the multi-task approach without the decomposed hypotheses. We believe this is because the different hypotheses do not have significantly different robot policies associated with them in the restaurant domain.

Our focus in this work is mostly on the discrepancy reasoning and recovery; future work could involve studying the discrepancy detection and diagnosis in more details. Our approach assumes to have access to an oracle, *e.g.*, a restaurant waiter, that can address the clarification questions. Future work could focus on designing more appropriate questions that can be asked from the customers when an oracle is not available. Another area that we do not focus on in this work is how (and if) the planning model should be updated with the new discrepancy information.

Chapter 8

Related Work

Autonomous robots that face a diversity of environments, a variety of tasks and a range of interactions cannot be pre-programmed by foreseeing at the design stage all possible courses of actions they may require. Especially, in dynamic and changing environments with semantically rich tasks and human interactions such as the restaurant domain, robots with explicit deliberation are needed. Even for applications in well-structured environments with a reduced range of tasks, where engineered robotics operations are feasible, deployment and adaptation costs can be reduced if the robot is equipped with deliberation capabilities. These deliberation functions include: planning, acting, observing, monitoring, and learning [86]. What design choices a roboticist make regarding each of these functionalities depend on the domain that the robot is operating on and the resources it has access to. In this work, we focus on two deliberative functionalities, namely planning and monitoring, for domains such as the restaurant domain where the robot should achieve multiple independent tasks. As part of the planning functionality, we discuss the related work on the formalization of the waiting tables task as a planning problem and efficient decision-making algorithms to achieve multiple independent tasks. As part of the monitoring functionality, we discuss the related work on addressing the discrepancies that arise during execution.

In this work, we focus on problems where a robot is required to accomplish a set of prespecified tasks. In many robotics applications this assumption is valid since the tasks can be modeled in a decomposed fashion. Examples of these domains are given in Chapter 1. If the underlying tasks are not given beforehand, our work can be combined with the existing approaches that decompose a huge model into multiple task models to generate the tasks.

We start by discussing approaches that expedite planning and execution without using the independence structure. We then discuss methods that do leverage the independence structure to speed up planning and execution. We provide a discussion on the approaches that decompose a

huge model into multiple tasks. Finally, we expand on the approaches that enable a robot to handle the discrepancy between the robot's model and its observations through learning and planning.

8.1 Formalization of the Restaurant Domain

We first discuss the task representations that our robot could use to perform planning. We then explain how the task representations have been used in relevant works to model the restaurant setting and provide a discussion on their drawbacks.

8.1.1 Task Representation for Planning

A lot of robotics applications consist of a set of tasks where the way the robot sequences the tasks and is moving between them significantly affects the overall performance. The naive way is to look at this problem as a navigation problem and represent it as a Traveling Salesman Problem (TSP) where the goal is to find a minimal-cost cyclic tour through a set of points such that every point is visited once [5]. However, to apply these approaches in real-world, multiple additional factors of the robotic site have to be considered, e.g., obstacle avoidance, partial order of the sequence, complicated objective functions, *e.g.*, time or energy, a set of possible robot base locations, or even temporal constraints. There has been a lot of works on modeling such problems [3]. In multi-goal path planning problems, the goal is to determine a cost-efficient collision-free path to visit a set of locations (goals) and return to a starting location [65, 205]. Differently, task sequencing or scheduling approaches can have complicated objective functions and may consider obstacles or may not [2, 102, 207]. There has also been a lot of work on scheduling with temporal constraints [53, 138] and even combining temporal reasoning and spatial reasoning [63, 106]. All these representations focus on representing tasks in some way and sequencing them to optimize some objective function while satisfying spatial and/or temporal constraints. Although the objective of our work is the same, we focus on applications such as waiting tables in a restaurant that focuses on achieving a set of tasks with internal states that evolve over time, *e.g.*, the people become dissatisfied after a few time steps, and may be partially observable, *e.g.*, the customers' request might not be directly observable. Different from the path planning literature, our focus is on task planning. Our work also differs from the task scheduling literature since in the domains that we target there are no explicit temporal constraints, the robot should figure out in what order it should attend the tasks by modeling how each task evolves as the robot performs actions. These types of problems can be represented as sequential decision-making models under uncertainty. In particular, models that can represent stochastic actions and partially observable state spaces are used.

The most common representations for sequential decision models in decision-theoretic planning under uncertainty are Markov decision processes (MDPs). MDPs provide a mathematical framework for modeling sequential decision-making in situations where outcomes of actions are uncertain. The objective of an agent is to maximize the accumulated reward over its lifetime. The solution for an MDP is a policy that decides the best action for each state in the MDP, known as the optimal policy. MDPs have proven to be useful in a variety of sequential planning applications where it is crucial to account for uncertainty in the action execution. These applications include inventory management, highway pavement maintenance, quality control, modeling medical treatment, and of course robotics [27, 67, 97, 99, 198]. We use the MDP representation to model the tasks in our mobile robot domain in Chapter 4.

The partially observable MDP model (POMDP) generalizes the MDP model to allow for even more forms of uncertainty to be accounted for in the process. In POMDPs, the true state of the system is not directly observable by the decision-maker. Instead, the decision-maker receives observations from the environment. POMDPs have been shown useful in formalizing a variety of different application including machine maintenance, interplanetary rovers, machine vision, network troubleshooting, search and rescue, education, and medical diagnosis [36, 37, 152, 175]. Especially, in the context of human-robot interaction, models inspired by POMDPs' ability to represent unobservable mental states in people and reason based on beliefs have been used to model social behaviors in agents models [33, 91, 134, 136, 152]. POMDPs have enabled robotic teammates to coordinate through communication [14], software agents to infer the intention of human players in game AI applications [116], and autonomous driving or robot navigation where the robot interacts with pedestrians and human drivers [12, 13, 33].

Some works show the advantages of using the POMDP representation on real-world problems over more naive approaches that do not model the human's mental state. Examples of these are the following. Some work formulates robot-student tutoring help action selection problem as Assistive Tutor POMDP (AT-POMDP) by maintaining a belief over the student's mastery of the material and engagement with the task [152]. Their evaluation demonstrated the effectiveness of using the AT-POMDP to help students with a long division math task. The students who received help from the AT-POMDP policy showed improved learning gains when compared to students who received help from a fixed policy. In another work the intentions of the humans are represented as hidden state in POMDP, and the time-dependence of action outcomes are explicitly modeled for both the humans and the robot. State aggregation over the time dimension of the state space is used to trade-off between the quality of the representation of time and the model's size in order to find sufficiently expressive models that can also be solved tractably. They show that the policies for time-dependent POMDP models with human intention as hidden state outperformed

the policies of the less expressive models such as time-dependent MDP models [33]. In [91], the authors formulate the problem of shared autonomy as a POMDP with uncertainty over the user's goal. As solving the POMDP to select the optimal action is intractable, they use hindsight optimization to approximate the POMDP solution. In a user study, they compare their method to a standard predict-then-blend approach, *i.e.*, predict a single goal, then assist for that goal, and find that their method enables users to accomplish tasks more quickly while utilizing less input. In our work, we use the POMDP representation to model the tasks in the restaurant domain. We consider the customers' internal state as partially observable. As most POMDP algorithms are interactable when applied on real-world problems, some of these works also focus on speeding up POMDP planning by using approximation methods [91] and the assumptions that works for a specific domain, *e.g.*, the mixed observability assumption in [135]. We use a similar approach as the latter works for the restaurant setting and show how our work in Chapters 5 and 6 leverages the independence between the multiple POMDP tasks to expedite planning.

We have already introduced the Markov Decision Process (MDP). Then, the Partially Observable Markov Decision Process (POMDP) was presented which was the focus of most of this thesis. Chapter 2 provides more details on these two representations.

8.1.2 Formalization of the Waiting Tables Task

Waiter robots have been deployed in restaurants to assist waiters by carrying food to the tables [163]. There has been work on robot localization and navigation in a restaurant [149, 206]. Different from these works, we use existing localization and navigation algorithms [21], and we focus on task planning rather than path planning. Task planning is another area of research that has been studied in the service robot domain [94, 129]. These works either do not model the customers' satisfaction or model it as an observable variable and use it to prioritize the next task. Differently, we are interested in how the robot's sequence of actions maximizes the customers' long-term satisfaction which is not directly observable. A key aspect of our formalization is that the tables evolve over time; this is not modeled in any of the previous works. Research has been done on predicting the customer's state in a restaurant or a bar [39]. This work focuses on inferring the customers' internal state and using that to select a robot behavior. In contrast, we focus on how the robot's sequence of actions impacts the customers' long-term satisfaction. Our formalization does not only enable a robot to consider what action it should execute immediately, but also what sequence of actions will be performed in the future to increase the customers' satisfaction.

8.2 Robot Planning for Achieving Multiple Tasks

We focus on relevant works where the robot (or agent) should achieve multiple tasks. In this section, we target problems were the smaller models or tasks are given, and we only focus on expediting planning and execution for these problems. Our work in Chapter 4 uses the MDP representation and the work in Chapters 5 and 6 use the POMDP representation. Thus, we compare the former with the MDP literature and the latter with the POMDP literature.

8.2.1 Combining the Tasks and Solving Large Models Efficiently

As mentioned earlier, a conventional planning approach for domains represented as multiple tasks is to combine all the tasks' states and actions into one large model and compute the optimal policy for the combined model at each time step. However, this approach is impractical if the number of tasks are large. In this section, we provide relevant work on how to solve large MDP and POMDP models faster.

Speeding Up MDP Planning

State abstraction (or state aggregation) has been extensively studied in artificial intelligence for making learning and planning algorithms practical in large, real-world problems. The robot finds solutions in the abstract state space much faster by treating groups of states as one state instead of working in the original state space. Some work provides a unified treatment of state abstraction for Markov decision processes [113]. They study five abstraction schemes and analyze their usability for planning and learning. Another work investigates approximate state abstractions, which treats nearly-identical situations as equivalent. They present theoretical guarantees of the quality of behaviors derived from four types of approximate abstractions and empirically demonstrate their effectiveness [1].

An object-oriented representation, which is an extension of the MDP formalism where the state is represented as a set of objects each of which is composed of a set of features or attributes has been proposed in [58]. They show orders of magnitude faster learning compared with state-of-the-art algorithms that use standard representations not based on objects. A similar formalism, relational MDPs (RMDPs), was introduced by [77] as a way to generalize plans to new environments as well as generalizing plans from smaller tractable environments to significantly larger ones. Another representation called factored MDPs allows very large, complex MDPs to be represented compactly. In factored MDPs the framework of dynamic Bayesian network (DBN) describes a compact representation of the transition model and the reward function. Different

algorithms have been proposed to solve these problems [11, 31, 32, 54].

Temporal abstractions have also been successfully used to increase the speed of planning and learning. Hierarchical methods, such as MAXQ [57], allow learners to exploit a task that is decomposed into different sub-tasks. The decomposition enables an agent to learn each subtask relatively quickly and then combine them, resulting in an overall learning speed improvement (compared to methods that do not leverage such a subtask hierarchy). In reinforcement learning (RL), options [179] provide a general framework for defining temporally abstract courses of action to speed up learning and planning. Hierarchical Reinforcement Learning (HRL) and planning approaches have also been proposed to learn the internal policies of options (or subtasks) and their termination conditions, in tandem with the policy over options [10, 15, 103, 105].

All the above approaches expedite planning or learning in some way for MDP models by using state and temporal abstraction techniques, or assuming some object-oriented, hierarchical, factored structure. Different from these approaches, our work in Chapter 4 focuses on speeding up task execution rather than task planning by planning less often. The robot focuses on one task at a time and only solves the large model when necessary. This both speeds up task execution and also removes the necessity of processing all the sensory variables at each time step. One can integrate our work with the above approaches to expedite both task planning and execution.

Speeding Up POMDP Planning

POMDPs provide a general framework for planning in partially observable stochastic environments. However, due to their computational complexity, for most real-world problems with large state and observation spaces, POMDP planning is computationally intractable. The challenges arise for the following two reasons. First, as the state is not fully observable, the agent must reason about probability distributions over the states (or beliefs). This challenge is called the ”curse of dimensionality". In other words, the agent needs to reason over the complete history of actions and observations up to the current time, in order to decide which action to perform. The second major challenge is the “curse of history". In a planning task, a robot often needs to take many actions to reach the goal, resulting in a long planning horizon. For POMDPs, the complexity of planning often grows exponentially with the horizon. Thus, together, a long planning horizon and a high-dimensional belief space compound the difficulty of planning under uncertainty. Research on this topic has focused on solving POMDPs more efficiently by addressing either or both of these challenges.

There is extensive research on speeding up POMDP solvers using different variations of the point-based value iteration method [144]. These methods dramatically speed up solving POMDPs and generate approximate policies for large domains by using probabilistic sampling [169],

specifically the probabilistic selection of the belief space subset and the order of value function updates of the belief space. They use a small number of sampled beliefs, thus are also able to plan for longer horizons. It has been shown that these approaches generate good, approximate policies for large domains. Similarly, Monte-Carlo-based approaches deal with the challenges of POMDP planning by sampling from the belief state and generating random simulations to estimate long-term reward [173].

Other methods for scaling up POMDPs leverage factored representations in the form of decision trees [30] or graphs [11, 168], specifically Algebraic Decision Diagrams (ADDs). Even though ADDs expedite planning by utilizing the limited dependencies between the state variables, they fail to compactly represent the POMDP when the policy is dependent on all the variables [166].

Some research for scaling up POMDPs compress the state space [147, 160] by mapping high-dimensional belief state into low-dimensional compressed belief or by bounding the number of non-zero values within each belief point [204]. In [114], the authors propose an approach to cluster belief states and combine it with belief compression to further improve POMDP tractability.

Research on hierarchical POMDP planning (HPOMDP) includes learning how to perform a set of subtasks independently, and then learning a high-level policy to sequence the subtasks [181]. A method [69] focuses on clustering the belief space to decompose a flat POMDP into an HPOMDP that has coarser discretization at higher levels for both state and action space and then solving the HPOMDP. Another method uses hierarchical finite-state controllers to scale-up planning and to provide theoretical guarantees on the quality of the computed policy [78]. Other work gets insights from the hierarchical approaches and provides a point-based approach that automatically generates long action sequences and uses the sequences rather than the primitive actions to guide sampling in the belief space and reduce the effective planning horizon [108]. Different from all these approaches, the tasks in our domains are independent and the robot can execute them in any order, *i.e.*, no task provides a precondition for another task. Our approach not only considers executing the tasks in a sequence but also interleaving the tasks' execution when more rewarding.

In the POMDP literature, there is not much research done on the impact of planning horizon on a POMDP's solution; however, research on fully observable POMDPs (MDPs) has focused on theoretical aspects of shallow versus long horizon planning. Some work assumes an accurate model and identifies a set of properties of MDP that determines the loss due to shallow planning [93]. Another work focuses on how approximating a value function in MDPs becomes increasingly difficult if the horizon increases [110]. Differently, it has been shown that planning for smaller horizons can be beneficial when approximate POMDP solvers are used or the model is not accurate [40, 92]. Another work expedites planning by planning up to a truncated horizon and

using an estimate of the true optimal value, learned by solving similar MDPs for the full horizon, as the terminal value [66].

All the above approaches expedite POMDP planning in some way by using approximation methods, using compression or clustering techniques, or assuming some factored or hierarchical structure in the domain. Although, we share common insights with these methods, *i.e.*, computing bounds [107, 174] and planning up to a truncated horizon, we leverage the structure in the class of problems with multiple independent tasks to expedite planning for both short and long horizons. There have been methods that use the structure in these problems, but they do not address optimal POMDP planning for long horizons [166, 200]. In Chapters 5 and 6, we consider a structure in the domain, namely the independence among the tasks, that differs from the assumptions made in the previous methods. Our approaches can leverage the above methods to further expedite planning when finding solutions for subsets of tasks.

8.2.2 Merging the Solutions to the Individual Tasks

In the literature, there is a large amount of research on merging the solutions to multiple MDPs instead of solving the combined MDP model. We provide the pros and cons of each class of approaches. Some of these works can be also applied on the POMDP models. However, we are not aware of any approaches that apply them on the POMDP models. We could only find one recent and relevant work that focuses on combining the solutions to multiple POMDP models.

Merging MDP Solutions

Goal management is an area of research that has some similarities to our work [187]. Some work permits arbitration between current goals based on priority values that are dynamically computed from predefined conditions and rules [43]. Another work evaluates all possible goals for the agent using a set of predefined fitness functions and selects the goal with the best combined score [128]. However, in our domains, it is not feasible to hard-code the conditions or fitness functions. In some other work, Q-learning is used to learn a goal selection policy over all the state variables [89]. We take a more reactive approach that only focuses on one task at a time and switches between the tasks when it is triggered rather than considering all the tasks at each time step.

Behavior-based control systems (BBC) provide algorithms to select and activate the appropriate behaviors given the robot's observations [118, 146]. In these approaches, the behaviors are selected and executed concurrently to collectively achieve the desired system-level behavior [133]. They take a decentralized approach to decide what action to execute, whereas we take a centralized approach to decide what task to perform. Another work predefines the conditions and

learns a switching policy over them [117, 151]. However, we do not predefine the conditions for the switching policy.

The restless multi-arm bandit problem (RMAB) [197, 200] concerns the optimal allocation of resources over time among a collection of bandits (or tasks) which are in competition. At each time step, an algorithm should decide which bandits should be active, *i.e.*, follow their optimal policy, and which bandits should be passive, *i.e.*, evolve to a new state. The optimization is to find a policy for the sequential selection of active bandits. These problems can be modeled as Markov Decision Process (MDP), are intractable [75], and have shown to have near-optimal heuristic solutions on real-world problems [56, 115]. Another relevant planning approach focuses on sequential stochastic resource allocation problems where multiple MDPs are weakly coupled by resource constraints [29, 123]. In [123], the authors describe heuristic techniques for dealing with several classes of constraints that use the solutions for individual MDPs to construct an approximate global solution. Different from these approaches, our approach in Chapter 4 uses a learning approach to decide when the robot should switch to another task rather than planning at each time step.

As mentioned earlier, some of the above approaches can also be applied on the POMDP representation, although we are not aware of any works that apply them on the POMDPs. We now briefly mention how the algorithms in Chapters 5 and 6 are different from the above approaches. Our algorithms use a POMDP representation, instead of an MDP representation, and propose planning approaches (instead of learning approaches) to address the multi-task domains. The approaches are different since they 1) take into account the partial observability of the state space, 2) are fast and provably optimal for both short and long horizons, and) do not limit the robot's action to be passive or active; the robot's action set is a union of all POMDPs' action sets.

Merging POMDP Solutions

The only work that we are aware of and the closest work to ours is by [166], which developed an algorithm to decompose a factored POMDP into a set of restricted POMDPs (or tasks). They solve each identified task separately, create a set of models with all possible combinations of the subsets of the tasks, randomly sample the subsets and solve them, and combine the policies of the smaller sampled subsets into a policy for the complete model. This work mostly focuses on decomposing a huge factored POMDP into multiple POMDPs (or tasks) and provides a sampling approach to select the subsets. Differently, our work provides rigorous approaches to select the subsets and solve them to find a provably globally optimal solution. We assume that the robot has a predefined set of tasks, efficiently remove subsets of tasks that have a low solution quality, and then create a set of smaller models from the remaining subsets and solve them. We compare

against this method [166] in Chapters 5 and 6.

8.2.3 Discussion on How to Decompose a Large Model Into Multiple Tasks

In the previous section, we assumed that the multiple tasks are given and discussed approaches that speed up planning. In the case that the multiple tasks are not given, one may use the following approaches to decompose the large model into multiple tasks or sub-goals. Most of these approaches focus on hierarchical decomposition of large MDP or POMDP models.

In the MDP literature, decomposition techniques try to solve the global MDP by solving small local MDPs [52, 109]. These works present a general framework to decompose MDPs into smaller ones. The local solutions are then pieced together using a hierarchical policy construction approach to obtain a global solution. More recent hierarchical RL approaches are capable of learning both the internal policies of options (subtasks) in tandem with the policy over options [10, 103, 105]. Other approaches have also been proposed to perform hierarchical decomposition of factored Markov decision processes [95]. To be able to represent multiple levels of hierarchy, the MAXQ hierarchical representation has been used. In [121], they present an algorithm for automatic discovery and transfer of MAXQ hierarchies.

In the POMDP literature, some hierarchical POMDP planning (HPOMDP) approaches investigate the problem of automatically discovering the hierarchy. Some work models the search for a good hierarchical policy in POMDP problems as a non-convex optimization problem with variables corresponding to the hierarchy and policy parameters [38]. Another work addresses the computational difficulty of solving the optimization problem by combining the factored encoding of hierarchical structures into a dynamic Bayesian network (DBN) with a maximum-likelihood estimation technique for policy learning [183]. Some of these approaches may not be directly applicable on the domains that we focus on since the tasks in our domains are independent and no task provides a precondition for another task; however, they can be applied on other robotics domains.

As mentioned earlier, some work developed an algorithm to decompose a factored POMDP into a set of restricted POMDPs (or tasks) and then combine their solutions to find a policy for the complete model [166]. They explain the process of identifying variables that correspond to tasks, and how to create a model restricted to a single task, or to a subset of tasks. Our work can be combined with this approach to first decompose a factored POMDP into smaller POMDPs and then merge their solutions to find an optimal solution.

8.3 Discrepancy between Observations and Planning Model

When executing plans, the world may evolve differently than predicted resulting in discrepancies between predicted and observed states of the world. These discrepancies can be caused by noisy sensors, unanticipated exogenous actions, or by inaccuracies in the predictive model used to generate the plan in the first place. Regardless of the cause, when a discrepancy is detected, it brings into question whether the plan being executed remains valid (i.e., projected to reach the goal) and where relevant, optimal with respect to some prescribed metric. The available literature for execution monitoring is mostly concerned with the problem of monitoring plan validity [68]. This is the problem of deciding whether an executing plan will still reach the goal after unexpected events have happened. There are also works on more complex problem of monitoring optimality [72]. This thesis is concerned with monitoring plan validity.

This line of research is related to a broad category of approaches that focus on execution monitoring. There are several surveys that classify different execution monitoring approaches applied to robotics and as a general problem [4, 84, 85, 143]. In this section, we provide an overview of this field and discuss relevant works that pertain to our work on addressing discrepancies in POMDP planning. For more details regarding the different execution monitoring approaches, please refer to the surveys.

In order to operate in a changing and partially unpredictable environment, robots need the ability to detect when the execution does not proceed as planned, and to correctly identify the causes of the failure. An execution monitoring system is a system that allows the robot to compare what is predicted regarding the robot activities to what is observed in the world. It detects and interprets the discrepancies, performs diagnosis and triggers recovery actions when needed [24]. Reasoning about uncertainty as part of the planning problem can be done and is necessary in order to weight different viable plans; however, the reasoning do not necessarily increase robustness of the execution, *e.g.*, if there is missing information. Thus, in order to act robustly in a partially unknown and dynamic world, the system must also tolerate uncertainty (failing execution). A monitoring system could be divided into the following 3 functionalities: 1) fault detection that indicates that something is going wrong in the monitored system, 2) fault diagnosis that makes a classification of what is going wrong, and 3) fault recovery that takes relevant actions to handle the situation [143]. The execution monitoring approaches are classified into: analytical, data-driven, and knowledge-based approaches [143]. This classification is mainly inspired from the field of industrial control.

Analytical approaches (or model-based methods) rely on planning and acting models, but also control theory models and filtering techniques for low-level action monitoring. These methods use

a model to predict the state of a system and compare this prediction with the observation of the current state. Data-driven approaches rely on statistical clustering methods for analyzing training data of normal and failures cases, and pattern recognition techniques for diagnosis. Knowledge-based approaches exploit specific knowledge in different representations (*e.g.*, rules), which is given or acquired for the purpose of monitoring and diagnosis. Our approach to discrepancy detection, isolation and recovery can be classified as a model-based approach.

Model-based methods can be separated into approaches that use qualitative models of the world, and approaches that use quantitative models of the world. The modeling formalisms call for very different approaches. In quantitative models, the model is usually developed based on some fundamental understanding of the physics of the process. In quantitative models, this understanding is expressed in terms of mathematical functional relationships between the inputs and outputs of the system. In contrast, in qualitative models these relationships are expressed in terms of qualitative functions such as logical elements. Qualitative model-based methods are often used for high-level plan monitoring, since they are based on logic mostly [48, 51]. A common approach is to monitor the execution of plans using preconditions and effects of actions to check that the plan is still consistent [62, 68, 189, 191] while considering temporal plan constraints [59, 112]. Quantitative model-based approaches usually rely on the generation and analysis of numerical residuals, rather than on logic [84, 122]. Residuals are differences between model-generated estimated values and the values observed during execution. These residuals should ideally be zero (or zero mean) under no-fault conditions. In order to be useful in practical applications, they should be insensitive to noise, disturbances, and model uncertainties while maximally sensitive to faults. Quantitative model-based methods are generally used to monitor stochastic and continuous systems. There are also research efforts on integrating these approaches into a joint framework [48, 201].

We say a discrepancy has happened when the robot's observation of the world does not belong to the set of observations that the robot could receive from the environment by reasoning on its planning model. Under the assumption that the robot's sensors are not defective, the planning model should be the reason for erroneous expectation, and the different elements of the POMDP model should be examined for their inaccuracies. Similar to many other planning and execution monitoring approaches (even learning), we assume that the state and observations spaces of the planning model are complete. Thus, the transition and observation functions of the POMDP are the main cause of the discrepancy. Hence, in this work, we will focus on the discrepancies that arise due to the inaccuracies in the predictive model, and we will present related work in the field as it pertains to our work on discrepancies in POMDP planning. Among the discrepancy detection, diagnosis and recovery functionalities, our main focus is to use the original inaccurate planning

model for discrepancy recovery, so the remaining of this section will mostly focus on planning under model inaccuracies.

8.3.1 State Estimation, Plan Repair and Replanning

Previous work has looked at the problem of fault detection and diagnosis as a state estimation problem; given some observation that gives rise to the suspicion that the actual current state is not the one that is expected, one would like to identify the actual state [196]. Particle filters have been extensively used for Bayesian state estimation in non-linear systems with noisy measurements. They approximate the probability distribution with a set of samples or particles. These algorithms are robust to modelling errors including unmodelled movements and systematic errors. An approach within this body of literature called Sensor Resetting Localization (SRL) addresses the sensor estimation problem by inserting additional hypotheses generated from sensing in the belief state when the robot is uncertain of its position [111]. Some other work combines particle filters with POMDPs for controlling a system [192]. A policy for the POMDP is computed offline while particle filters are used online to track the belief state. They present multiple algorithms that focus on faults that cannot directly be detected from current sensor values but require inference from a sequence of time-varying sensor values. In [120], the author formulates the hybrid monitoring and diagnosis task as a Bayesian model tracking and selection problem. Following research on particle filters, they use a factored sampling technique to sample and represent the multi-modal posterior distribution of the state (models) given the observations. One major problem with the use of particle filters for diagnosis is that they focus on the most likely models, that is the nominal behavior, while fault modes are unlikely and can therefore slip the attention of the filter. In [120], the author overcomes this problem by biasing the samples towards the results of a separate, qualitative diagnosis.

Some works recover from the faults by generating a universal plan [164]– one that covers any possible situation the system may encounter, even when errors are made. In fact, when modeling the problem as a Markov decision process (MDP), a policy that covers the entire state space, such as the one generated by the value iteration or policy iteration algorithms, is a universal plan. Even the LAO* and RTDP algorithms exploit reachability analysis to converge on optimal universal plans that include only states that are reachable during the course of plan execution. In most of these works a fault is associated with the execution of a faulty action, either mistakenly performing a different normal action than the one specified by the plan (e.g., missing an exit in driving) or performing a special action that represents some faulty condition (e.g., having a flat tire). Since they map every state to an action, an agent executing the policy always knows what to do next and this choice will be optimal and so no execution monitoring for dealing with run-time

discrepancies is required, except for state estimation in the case of POMDPs. Planning can itself be made robust through replanning and plan repair [70]. Replanning can be activated when a plan has failed; it consists of stopping the plan execution, eventually positioning the system in a safe state, and developing a new plan from the current situation and the remaining objectives. Plan repair can be activated when part of a plan has failed, before replanning occurs; it can consist of modifying the remainder of the current plan or developing a new plan from the failed one by backtracking and eliminating the failed and impossible actions [76]. Some approaches consider replanning from scratch rather than repairing a failed plan as it can be more efficient in certain situations [131]; however, several people have found plan modification to be more efficient in practice [72, 100].

Uncertainty in planning problems is sometimes handled by modeling the problem deterministically - enabling classical planning techniques to be used - but using methods for execution monitoring and replanning to handle situations that arise when the plan fails (*e.g.*, when a precondition at some point fails to hold). According to [28], two extreme approaches can be adopted: The first requires monitoring all preconditions required by future actions in the plan (once they are established); when one fails replanning is invoked. Refinements of this scheme are, of course, possible. The second, and much more common, approach simply monitors the current state and should an unanticipated state be reached replanning is invoked. The former approach is very costly, and the latter approach is not able to anticipate failures in advance. Thus, the decision of whether to monitor a plan precondition, and when to monitor it, involves balancing the cost of monitoring and the value of monitoring information. In [28], the author addresses the cost versus value of monitoring trade-off by formulating the problem as a POMDP. They then develop approximation methods to solve the POMDP faster such that it scales with plan size. Some other work uses execution monitoring to address the challenges of POMDP solving. They propose an approach to approximately solve quasi-deterministic POMDPs by converting them into contingency planning problems or MDPs. They build plans assuming both the actions and the observations are reliable, then monitor the execution of the plan and use a value of information calculation to add information gathering actions online [193]. Similar to [28], the authors focus on classical planning plus execution monitoring to solve problems that could be represented as POMDPs and use value of information to measure whether monitoring is worthwhile.

Another relevant work [130] presents an MDP-based integrated formulation for implementing spacecraft goal-based mission planning, fault detection, and fault reconfiguration that takes into account both logic-based compositional and continuous-valued models. The MDP includes state variables and actions associated with the fault detection step (state variables such as fault flags and processed measurements, and actions such as diagnostic actions) and the reconfiguration step

(state variables such as current faults, ongoing mission-related actions, and mission status) as well as planning. They then manage the complexity of solving the huge MDP by decomposition of the integrated MDP into three separate MDPs for planning, fault detection, and reconfiguration thereby reducing the computational effort required to generate the policies and the memory space required to store them. In [145], the authors formally define error models that characterize the likelihood of various faults and consider the problem of fault-tolerant planning, which optimizes performance given an error model. They introduce an approach to plan for a bounded number of faults and analyze its theoretical properties. All these approaches have a model of the error or failure that might happen in the world and address it by including it in the planning model in advance.

Depending on the way the system is modeled and on the available observations, diagnosis may not be required or can be trivial. This is for instance the case when the state can be sensed completely, or when it can be sensed partially and the observations coincide exactly with the predictions of the model. Since then there is no discrepancy, there is no reason to believe that the actual state is any different from the predicted one. In any case, the situation-dependent need and requirements for diagnosis should be guided by its purpose, that is, in the case of execution monitoring, it should be determined with respect to the state evaluation and replanning. If, for instance, state evaluation is able to specify a subset of states in all of which the current plan should be continued, then there is no need to disambiguate between two candidate diagnoses when both candidates belong to this subset. This is for instance achieved by the explicit annotation of a sufficient and necessary condition for the continued plan validity in [72].

In all approaches that we described above the premise is that planning from scratch in the new, unexpected situation would produce a plan that is valid and optimal. But what if the discrepancy that occurred is due to a systematic error and will thus repeat itself? This is for instance the case when the agent applies an incorrect model of its own actions during planning. Consider the following example from [72] of a soccer robot equipped with a kicking device: The user provided the robot with a model describing that kicking will make the ball travel in a straight line until it hits an obstacle. Unfortunately, during the game a fuse blows, causing the kicking device to fail entirely. Assume the robot has intercepted the ball and is in a good position to score a goal by either kicking or pushing the ball into the goal. Since kicking is usually faster this is the preferred option, as it has a higher probability of success. Hence the robot triggers a kick action, but nothing happens, because of the hardware defect. The robot realizes that something went wrong when observing that the ball is still right in front of it, as this observation is inconsistent with the model. But planning again in the new situation will not do any good since the best plan will still be to kick instead of pushing the ball – according to the erroneous model. None of the approaches we have

described so far would ever get out of this loop and the robot would miss its chance of scoring. What is missing is a model adjustment step before replanning to account for modeling faults. Thus, the appropriate approach that a robot should take depends on the type of the discrepancy. For example, some work [24] distinguishes between two sources for discrepancies: exogenous actions (EA) and violation of ontological assumptions (VOA). The paper assumes that the system is always able to tell whether an action has been executed completely or not by reading internal sensors. This is used to determine whether an EA or a VOA has caused a discrepancy: if no action has been executed but a discrepancy occurs it is assumed to be due to an EA, otherwise it is due to a VOA. When an EA occurs there is no need to adjust the model of the dynamics, instead only the knowledge about the current situation is modified. If the discrepancy is deemed to be due to a VOA, they use the truth value of the fluents that do not match the expectations to extend the successor state axioms. Some other work [49] proposes to precede replanning with a model-adjustment step to alter the planning operators as necessary to accommodate for this kind of discrepancies. Not doing this implicitly assumes that a failure is never due to a systematic fault and this ignorance can lead to the infinite repetition of such a failure. Another work applies to path-planning problems where one needs to find shortest paths repeatedly as edges or vertices are added or deleted, or the costs of edges are changed, for example, because the cost of planning operators, their preconditions, or their effects change from one path-planning problem to the next [101]. They develop an algorithm and present analytical results that demonstrate its similarity to A*. They also present experimental results that demonstrate its potential advantage if the path-planning problems change only slightly and the changes are close to the goal.

Replanning, plan repair, and state estimation approaches suffice if the discrepancy is not due to a fundamental change in the environment and as a consequence to the robot's planning model. When there remains uncertainty about the model applied in planning, one can explore the environment in order to improve the model through reinforcement learning approaches. We will discuss the approaches that address POMDP planning where the model is inaccurate or uncertain in the next section.

The relevant works on this topic mostly divide into two categories. Some research focuses on learning the POMDP model's parameters. Some other works assume a good initial, but incomplete POMDP model and address the incompleteness using planning approaches. Our proposed contribution falls under the second category. Since our proposed approach uses human input to resolve the discrepancies, we end this section by discussing approaches that use human input to learn new skills or decrease robot's uncertainty for planning.

8.3.2 Learning and Refining the Model Parameters

Papers in this area focus on learning a policy directly from interactions with the environment. A well-known model-free approach [119] resolves perceptual aliasing in POMDPs by using variable length short-term memory. They use a test to determine when a distinction is relevant to a task to add it to its short-term memory. The Baum-Welch algorithm is used to tune the parameters of the model. Another work uses the same short-memory approach and proposes a model-free algorithm for learning finite-state controllers of a given size by reducing the search to policies representable as finite policy graphs [124].

Learning a model from the environment and then solving it using a model-based approach or interleaving these two steps has also been studied. Some work presents a model-based algorithm that learns a POMDP model and its solution in conjunction, avoiding the slow computation of the Baum-Welch algorithm [167]. They augment the USM algorithm originally proposed in [119] to learn a POMDP model using a predefined sensor model. Some works use Bayes-adaptive POMDPs (BAPOMDPs) which are POMDPs characterized by a prior distribution over their underlying hidden Markov model (HMM) parameters [50, 156, 159, 186, 194]. Rather than fitting a single HMM to training data—which could introduce significant modeling error into subsequent analysis—the defining characteristic of BAPOMDPs is the use of a prior distribution to capture model uncertainty. The optimal BAPOMDP policies make decisions under uncertainty in both the system state and model parameters. A recent work is motivated by that the optimal policy to any finite horizon POMDP can be captured by a finite-state controller (FSC) [186]. They extend an expectation-maximization (EM) algorithm for solving Bayes-adaptive MDPs (BAMDPs) via FSC optimization to the more general BAPOMDP setting.

To decrease the amount of data needed, some work presents an algorithm called MEDUSA which incrementally learns a POMDP model using oracle queries and heuristics to select actions that will improve the model, while still optimizing a reward function [90]. A similar approach is taken in [61] which operates on a finite number of POMDPs sampled from an unconstrained model posterior. Given a history of actions and observations, the next action is chosen by minimizing the Bayes risk over the set of all possible actions. They take an active-learning approach in which the agent asks an expert for the correct action to take if the agent deems that model uncertainty may cause it to take undue risks. These queries both limit the amount of training required and allow the agent to infer the potential consequences of an action without executing it. The convergence properties of this method require that the agent can query an oracle (at a cost) to reveal the optimal action at any time [61]. Another method learns the observation function of a POMDP using oracle queries [9]. They associate a Dirichlet distribution over possible observations to each state. Similar to the previous approach, a set of candidate models is sampled and updated whenever new

oracle information about an optimal action becomes available. The optimal action that is inquired from an oracle is compared with the actions that each POMDP suggests. If the actions agree, the Dirichlet parameters are updated.

Intrinsic motivation can also be used to enable the agent to learn a useful model of the environment that is likely to help it learn its eventual tasks more efficiently [71, 170]. As in [82], the intrinsic reward can be used to drive the agent to where the model is uncertain in its predictions, or to acquire novel experiences that its model has not been trained on. In some work, the agent improves its model of the environment through probabilistic inference, and learning progress is measured in terms of Shannon's information gain. They provide a theoretically sound foundation for designing more effective exploration strategies [177].

8.3.3 Solving the Models in Presence of Model Uncertainty

Optimal solutions to Markov Decision Problems can be sensitive with respect to the state transition probabilities as in some practical problems, the estimation of those probabilities is not accurate [55, 199]. This problem was first proposed in the context of MDPs. Bounded Markov Decision Processes (BMDPs) [74] in particular specify the uncertainty over the transitions using uncertainty intervals and solve the BMDPs using optimistic or pessimistic assumptions [88, 137]. Towards more effectively modeling problems with unpredictable events, some work uses a factored MDP model [31] and develops a hybrid framework that explicitly distinguishes decision factors whose conditional probability tables (CPTs) are not assigned precisely while still representing known probability components using conventional principled MDP transitions [202]. They describe two solution approaches for modeling such events. The first approach augments the conventional model with additional features that effectively render the events predictable. In the second approach, they devise a model wherein unpredictable events are explicitly treated as special factors whose CPTs are not assigned precisely, and provide a formal method for recasting the problem as a BMDP. In a similar vein, some work introduces a novel class of problems called Configurable Markov Decision Processes (CMDPs) where the robot is allowed to explicitly reason about different transition functions. If changing the transition function is deemed more rewarding, the robot can ask an external entity, *e.g.*, a human user, to change the environment accordingly. They show the effectiveness of their approach from theoretical and algorithmic perspectives [172] as well as its performance in multiple problems [171].

The BMDP formulation and algorithms have also been applied to POMDPs [87, 132, 139]. In [132], they assume that the parameters of POMDPs are imprecise but bounded and formulate the framework of Bounded-parameter Partially Observable Markov Decision Processes (BPOMDPs). They propose a modified value iteration as a basic strategy for dealing with parameter imprecision

in a BPOMDP. They then introduce an anytime algorithm based on computing lower and upper-bounds for solving BPOMDPs. One of the first works that studied parameter imprecision of general POMDPs is by [87] where they formulate a new framework, POMDPs with imprecise parameters (POMDPIPs), introduce a new optimality criterion by adopting beliefs in the imprecisely specified state transition functions and observation functions, present an algorithm to solve them, and provide its theoretical analysis. Another work builds upon [87, 132] and the BMDP literature and introduces a new optimality criterion to solve BPOMDPs and a policy iteration algorithm that converges to an ϵ-optimal policy under the proposed optimality criterion.

We are also taking a planning approach to handle the model imperfection. Different from the above papers, we do not consider the uncertainty in the model definition explicitly. We resolve the discrepancy between the robot's observations and the model definition as it arises by querying an oracle.

8.3.4 Learning and Planning Using Human Input

The idea of querying an expert to obtain labels for unlabeled training examples has been explored in machine learning research. In this context, Active Learning (AL) has been used to minimize labeling costs while considering the value of obtaining correct labels [44, 45]. This problem has also been explored in more complex domains where there are multiple oracles, each of which may be reluctant to answer, incorrectly answer, and have data point-sensitive costs subject to a fixed budget [60, 203].

In robotics research, active learning has been used in robot learning from demonstration research area which aims at learning new skills from human demonstrations. Active learning has been used to maximize the generalizability of a learned skill to unseen situations while efficiently using the human teacher's limited time [41]. In this context, some work presents a supervised learning technique that uses measures of similarity to past examples and classification confidence of the underlying supervised learning algorithm to determine when the robot should act autonomously or request a demonstration [42]. Another work investigates whether robots can supplement their questions with information about their state in a manner that increases the accuracy of human responses [154]. In [34], the authors identify three types of questions (label, demonstration and feature queries) and discuss how a robot can use these while learning new skills. In label queries category, the learner selects an unlabeled instance and requests a label for it. In demonstration queries category, the learner finds a configuration of the environment that its model does not cover, and asks for a demonstration. Feature queries are divided into subcategories; the robot can ask about a particular feature's relevance for the task, its variance or invariance, whether a feature can have a certain value, or what values a feature is allowed to have. The paper presents

two experiments; one experiment characterizes the use of these queries in human task learning and the other evaluates them in a human-robot interaction setting. They find that feature queries are the most common in human learning and are perceived as the smartest when used by the robot. Most of these works focus on learning new skills from demonstration using active learning while also investigating how the incorporation of AL methods in LfD impacts a robot's interaction with its user. Our focus is not on the interaction between the robot and the customers or the oracle, but we get inspiration from the above works for generating the questions. As mentioned previously, in POMDP literature, some works learn the POMDP parameters by inquiring a human expert [9, 61, 90]. Different from all these works, we focus on developing planning and execution algorithms that leverage queries to resolve the discrepancy between the robot's observations and its model.

A relevant work [7, 8] presents a novel type of POMDPs called Oracular POMDPs (OPOMDPs) which rather than standard observations include an "oracle". This representation allows the robot to query an "oracle" for the full state information at a fixed cost. They show that this class of POMDPs are easier to solve than regular POMDPs. In [7], they introduce an efficient heuristic algorithm that utilizes the solution of the underlying MDP and weighs the value of consulting the oracle against the value of taking a state-modifying action. To address the optimality issues of [7], [8] presents an approximate, anytime algorithm that converges to the optimal value function and thus the optimal policy. This work assumes that the robot has access to an oracle at any point during planning. This approach expedites POMDP planning by querying an oracle for the full state information. Some work extends the previous work [153]. They explore the use of humans who are already in the environment as information providers in a real-world navigation scenario. They introduce a Human Observation Provider POMDP framework (HOP-POMDP) that models humans' availability and costs of interruption to determine when to query the humans during navigation. They contribute new algorithms for planning with HOP-POMDPs. Another work introduces a model for planning robot manipulation tasks under uncertainty in a human environment [185]. The planning agent is enabled to request human help in case of insufficient information about the current world state, including situations associated with a high risk. Furthermore, human users are provided with the opportunity to express preferences about the manipulation task's outcome. In all these works, the questions help with the state estimation problem, but do not address the model adjustment step that is required to prevent the robot from repeating failures.

Chapter 9

Conclusion and Future Work

In this chapter, we review the contributions of this thesis. We then discuss future avenues for future work. Finally, we summarize the thesis document.

9.1 Contributions

The key contributions of our work are the following.

Formalization of the class of problems with multiple independent tasks that evolve over time We introduce a novel class of problems with one robot attending to N independent tasks that evolve over time. We model each task, the robot's state and the actions that can be applied to the task as a POMDP and call it *client POMDP*. We then discuss what assumptions are necessary for the N client POMDPs to be independent. Finally, we discuss how the N client POMDPs are combined into one large POMDP model called an *agent POMDP*.

Formalization of robot waiting in a restaurant We formalize the waiting tables task as a planning problem with one robot and N independent tables (or N independent POMDP tasks). We provide the assumptions under which the restaurant domain can be an instance of the aforementioned class of problems.

Efficient task execution algorithm for multi-task problems We provide an approach for efficient *task execution* by reducing the need to perform planning at each time step. Our approach learns what state features have the largest impact on the switching behavior of the robot and uses them as triggers to stop the execution of the current task. The replanning step on all the tasks is only performed often when one of those triggers are active, thus, it speeds up the task execution.

Efficient short and long-horizon planning algorithms for multi-task problems with the independence structure We expedite task planning for the aforementioned class of problems. We leverage the independence structure in these problems to solve the agent POMDP more efficiently. Our key ideas include decomposing the agent POMDP with N tasks into a series of much smaller planning problems with k tasks ($k < N$) and computing lower and upper-bounds on the value of an optimal solution for variable horizons which allow us to terminate the search early while guaranteeing optimality. We analyze the algorithms theoretical properties and demonstrate its efficiency on the waiting tables domain.

An algorithmic framework for addressing the discrepancy between the robot's observations and its model We address the challenges of planning for real-world applications such as the restaurant domain where having an exact and comprehensive model that works for all the tables is infeasible. As a result, unexpected situations could arise that prevent the robot from attending to all the tables. We refer to these unexpected situation where the robot's observation differs from what is expected to be observed given the robot's model as discrepancies. We formulate the discrepancy between the robot's observations and its model as an augmented planning problem with a set of hypotheses that explain the discrepancy and a set of clarification actions that can invalidate the hypotheses. The goal of our formulation is to enable the robot to achieve the task irrespective of the existing discrepancy, and only diagnose the exact explanation for the discrepancy and resolve it if it helps with achieving the goal.

Efficient planning algorithms for solving the discrepancy model in multi-task problems We solve the discrepancy POMDP model more efficiently by using the lower-bound on the value of the individual hypothesis to compute better heuristic values for the frontier nodes in the graph associated with the discrepancy model. We then integrate this approach with the efficient long-horizon planning algorithms that we developed for multi-task problems. Since the discrepancy model has a goal POMDP representation that differs from the discounted sum POMDP representation that we used in the multi-task planning algorithms, we first describe what modifications should be made on the multi-task planning algorithms to enable them to be applied on goal POMDPs. We then describe how the heuristic computation for these multi-task planning algorithms changes to leverage the lower-bound on the value of each individual hypothesis.

Experimental results on the waiting tables in the restaurant domain We demonstrate the performance of our efficient short and long-horizon planning algorithms on a simulated restaurant setting with a large number of tables. We observe significant speedup when the robot uses our algorithms to decompose the large planning problem into smaller sub-problems while guaranteeing

an optimal solution. We then evaluate the efficiency of our planning algorithms that solve the discrepancy model in a single-task grid-world environment and observe a higher quality of solutions that our planning algorithms compute. Finally, we consider a task with a discrepancy model and multiple other tasks that use the original planning model and evaluate our planning algorithms in such multi-task setting. Our results show the benefits of decomposing the large planning problem into smaller planning problems.

9.2 Discussion

In this section, we discuss the challenges of running our algorithms on a real robot in a real restaurant setting. Although we mostly focus on the restaurant setting, the challenges extend to most multi-task settings. The next section will focus on the many avenues for future work.

Autonomous robots that face a diversity of environments, a variety of tasks and a range of interactions cannot be accurately modeled in simulation and pre-programmed by foreseeing at the design stage all possible courses of actions they may require. Especially, in dynamic and changing environments with semantically rich tasks and human interactions such as the restaurant domain, robots with explicit deliberation functions such as planning, acting, observing, monitoring, and learning are needed [86]. What design choices a roboticist make regarding each of these functionalities depend on the domain that the robot is operating on and the resources it has access to. In this thesis, we focus on two deliberative functionalities, namely planning and monitoring, for domains such as the restaurant domain where the robot should achieve multiple independent tasks. In this section, we will first focus on the challenges that the planning and monitoring modules should be able to tackle in a real restaurant setting. More specifically, we will go through the real-world situations where our assumptions do not hold. We will then discuss the general challenges of running our planning and monitoring modules on the real robot alongside the acting, observing, and learning modules.

9.2.1 Going Beyond the Assumptions

In the formalization of the restaurant setting, we assume that the N tables (or tasks) are independent. This assumption helps us in addressing some of the computational challenges of planning for such multi-task domains. However, in a real restaurant setting, the independence assumption might not hold. If the independence assumption does not hold for certain tables, one can combine the POMDPs associated with those dependent tables and solve it along the other tables. In this case, as long as some of the tables are independent, one can leverage our algorithms to expedite

planning. In the worst case, however, if all the tables are dependent on one another, our approach performs the same as the baseline approaches.

The definition of the independence structure could also serve as a limitation of our work. We assume that the robot's state is fully observable, and the robot's observation function depends on the task's state only. Thus, under this assumption, one cannot perform both active localization and task planning jointly in our POMDP planner. We basically decouple the task planning aspect of the multi-task domains from their path planning aspect and mostly focus on task planning. Although this decoupling of task and path planning is a common approach in robotics applications, it might limit the types of domains where one can use our approaches.

For the multi-task settings, we develop provably optimal algorithms that expedite planning for short and long horizon planning problems. Although our provably optimal approaches perform better than the baseline approaches, for long horizon problems, their high planning times prevent them from being practical in a real restaurant setting with many tables. For these problems suboptimal anytime approaches might be preferred (similar to the approach in Chapter 7). We will explain how our algorithms can be extended to further expedite planning for real-time applications in the next section.

In dynamic and changing environments with semantically rich tasks and human interactions such as the real restaurant domain, the assumption of having a perfect model of the domain might not always hold for all the tasks. We develop algorithms that effectively address the unexpected situations that arise due to having an imperfect model for multi-task domains by querying an oracle. Under the assumption that the robot has access to an oracle, *e.g.*, human waiter, these algorithms to some extend enable the robot to deal with the unpredictability of a real restaurant domain with diverse customers (*i.e.*, they might not follow the same model that the robot follows). However, the assumption that the robot has access to an oracle might become invalid in some domains, *e.g.*, the human waiter is not available when the robot needs him. For these type of domains or situations where an oracle is not available, the robot can still guarantee to reach the goal if all the diagnosed hypotheses agree on an alternative finite-cost route to the goal. In this case, the robot can just take the alternative finite-cost route to reach the goal without asking any clarification questions. However, if the hypotheses cannot agree on one finite-cost route to the goal, our algorithms fail to reach the goal. This signifies the importance of developing effective diagnosis strategies to come up with a suitable set of hypotheses.

The algorithms that we describe in this work are limited to a particular discrepancy detection approach, and they use domain knowledge for diagnosis. As we will describe in the next section, the discrepancy detection and diagnosis approaches can be extended to better address real robot settings where different detection and diagnosis strategies might be needed. For example, a soccer

robot that is pushed by another soccer robot could just take an state estimation and replanning approach to address the discrepancy rather than adjusting its planning model [111]. However, a robot with a hardware defect would need to adjust its model to replan effectively [122]. In real-world settings, the robot should be equipped with different detection and diagnosis strategies to be able to handle the different discrepancy scenarios.

9.2.2 Simulation to Real World

To some extend, we address some of the challenges of a real world setting by using a POMDP representation. The POMDP representation is more powerful than deterministic planning and the MDP representation. The POMDP representation enables the robot to reason about both uncertainty in the action execution and observation in a systematic manner, but at the cost of more computation effort. Thus, if the complex real world settings can be modeled accurately, meaning that all the elements of the POMDP including the states, the actions, the observations, and the transition and observation probabilities are given, the POMDP representation would be able to address the uncertainty of the real world. The POMDP representation assumes complete certainty over all its elements. However, as we discussed in Chapter 7, it is not always possible to model the real world perfectly. It is in fact very challenging to define in advance all the states, all the actions, all the observations, and the true transition and observation probabilities for the POMDP. The algorithms in Chapter 7 are designed to address some of the challenges associated with having an inaccurate model of the real world to some extend. *I.e.*, they address the challenges of having inaccurate transition and observation functions to guarantee that the robot will eventually reach its goal under the assumption that all the states, all the actions, and all the observations are given. However, even given the assumptions, there is still so much research to be done to have a system that can tackle different types of discrepancies and failures. Furthermore, the assumptions regarding the states, actions and observations might also be invalid in the real world.

In this thesis, we use a hard-coded restaurant simulator. We also assume that the robot is equipped with all the necessary actions and observations. However, in unstructured environments with semantically rich tasks and human interactions such as the restaurant domain, no matter how well we design the simulator, there will be differences between the simulation and the real world. The gap between the simulation and the reality could be due to the differences in physical modeling or perception modeling and might result in failed executions or errors in observations. The observation errors could include errors in localization, activity recognition, the detection of the table's current request, etc. There are techniques to increase the system's robustness to execution failures and perception errors. Differences such as physical modeling errors can sometimes be lessened via appropriate choices of the state-action space [165]. Differences such as perception

errors can also be lessened via appropriate filtering techniques or by adding extra sensors for more robustness, *e.g.*, by adding extra sensors on each table in the restaurant setting rather than only using top-view cameras or the sensors on the robot. In this work, we did not discuss the sensor requirements or the different methods for getting reliable and robust observations; however, having robust observations affects both the execution (or acting) and planning modules and is a crucial step to successfully deploy the robots in a real restaurant.

In addition to the previous challenges, many simulations, unlike the real-world, act completely synchronously with the policy decisions, meaning no changes in the environment happen while the agent is deciding the next action and there is no real-time constraint on the agents decision making. Such differences could be addressed by designing simulations that act asynchronously or having other strategies that appropriately deal with the changes in the environment while the robot is planning the next action.

In robotics, a lot of effort has been put into exploring training in simulation and then executing policies on real robot hardware. This approach is commonly known as Sim2Real [162]. There are two commonly used techniques which do bring in an element of transfer learning from simulation to the real world: domain randomization during training [182] and additional training after simulation in the real-world [162]. This signifies the importance of learning in the real restaurant setting for a successful transfer from the simulation to the real world.

9.3 Future Work

We discuss the many avenues for future work that we consider interesting and worth pursuing. We consider future work on topics that are more specific to this thesis as well as future work on topics that are related to the challenges in deploying the robot in a real restaurant setting.

Expediting planning further Many other approached have also been proposed to speed up POMDP solvers by using point-based methods [169], hierarchical planning [181], clustering and compression of belief space [160, 175], factored representation [168], and online POMDP approaches [158]. Our approaches can leverage the above methods to solve the subsets of tasks faster. In most of this work, we focused on optimal planning for multi-task settings. Another direction for expediting planning for multi-task settings is to focus on practical anytime algorithms that use both the independent tasks and the adaptive horizon (which inherently is anytime) and integrate them with the practicality of anytime sampling-based approaches such as [176] which randomly samples a subset of scenarios to speed up planning.

There are also techniques that could trivially be integrated with our methods to expedite

planning further such as: 1) solving the individual tasks offline, 2) solving the sub-problems in parallel (the sub-problems lend themselves well to multi-threading), and 3) using the lower and upper-bounds to prune the nodes that are known to be sub-optimal in the search tree [140].

Furthermore, in a specific restaurant, a waiter's procedures might follow a specific routine that could correspond to only exploring a certain part of the robot's state, action, and observation space. All the planning experiences that the robot obtains as it plans and executes action could be learnt into a model that could be used later on for fast replanning.

Frequency of replanning In our approaches in Chapters 5 and 6, a robot would reevaluate the environment after each action execution and replan accordingly. A drawback of this approach is that if the actions take a lot of time steps to execute, or a high-level action that consists of multiple primitive actions is executed, the robot could miss some key events. Differently, our approach in Chapter 4 focus on a more reactive approach to switch from one task to another using the stimuli. Since planning with a high frequency can be very inefficient, future work could involve integrating these two approaches such that the robot could replan after each action execution and when something interesting happens in the environment.

Discrepancy detection The appropriate approach that a robot should take to address a discrepancy depends on the type of the discrepancy. Some discrepancies might be due to exogenous events rather than as a result of an action execution. To handle these type of discrepancies, it might not be needed to adjust the transition or observation functions, instead only the knowledge about the current state could be modified; thus, the state estimation and replanning approaches might suffice [111, 120]. However, if the discrepancy occurs as a result of an action execution, replanning with a model-adjustment step to alter the transition or observation functions are required to accommodate for this kind of discrepancy. In this work, we only tackle the latter type of discrepancy where the model should be adjusted. However, a more in-depth study of the different types of discrepancies for a given domain might be necessary to choose the right strategy.

In this work, our focus is mostly on the discrepancy recovery rather than the discrepancy detection and diagnosis. We only focus on discrepancies where an observation is impossible given the robot's model. We do not consider cases where an observation has a low probability of being observed according to the model. Future work could involve detecting these low-probability discrepancies as done in [111].

Discrepancy diagnosis We generate a set of hypotheses by modifying the transition and observation functions. In general, coming up with a set of hypotheses that includes an approximation of the correct model could be domain-dependent and very challenging. Coming up with different

hypotheses generation methods that are appropriate for different types of domains could be an interesting future work.

We used domain-knowledge to decide what parts of the transition and observation functions should get affected when a discrepancy occurs. For example, in the grid-world, when a discrepancy occurs, it affects all the transitions that involve the current and next states irrespective of the action. For example, if the robot executes "north" from the state 4 and expects to end up in the state 0, and a discrepancy happens. This discrepancy involves a change in the transition from the state 4 to the state 0 with any action. If we could previously go from the state 4 to the state 0 with another action, *e.g.*, action "left", under the hypothesis that this transition is impossible, we cannot go from the state 4 to the state 0 with the action "left". Differently, in the restaurant domain, if the robot believes that the customers are "satisfied", and they become "unsatisfied" with the action "bring bread", they will also become "unsatisfied" if the robot believes that the customers are "neutral" and executes "bring bread". In this domain, when a discrepancy occurs, it affects all the transitions that involve the action and the next state irrespective of the current state.

In this work, we asked the clarification questions from an oracle waiter. Differently, some questions can also be asked from the customers themselves. Future work could involve having multiple types of oracles with different confidence on their answers and multiple types of questions with different information gain as done in [60, 203].

Updating the discrepancy model The focus of this thesis was not on permanently updating the model with the new discrepancy information. However, depending on the domain and the types of discrepancies, the discrepancy information could be useful to include in the model. In situations where the discrepancy persists throughout task executions, one might use learning approaches as discussed in Chapter 8.3.2 to update the planning model for later use [61, 90].

Efficient planning with multiple robots In this work, we focus on efficient task-planning for a robot in a multi-task setting. If we were to use our algorithms in a multi-robot restaurant, we could first assign the tasks to the different robot waiters and then plan for each robot individually. However, in presence of multiple robots, the task and motion planning aspects of the problem need to be integrated. This is because in a multi-robot setting, the assumption that the tasks are independent of one another might become invalid as going to a table by a robot might be blocked by a different robot going to another table. In addition to figuring out which robot has a priority over the other robot to pass first given the tasks' status [126], future work could focus on comping up with strategies to preserve the independence between the tasks.

In a real restaurant setting, the multiple waiters are able to share tasks depending on how

needy their assigned tables are. For example, if one robot is idle, it could take up an action that another robot should have performed for better overall performance of the restaurant. This could also be an interesting direction for future work.

Performing sidework tasks Waiters typically do work in the restaurant in addition to their main serving tasks. Waiter sidework could include duties such as cleaning service areas, refilling table condiments, tidying menus, restocking beverage and server stations. A robot's idle time could be used to perform these sidework tasks. One strategy to attend to these sidework tasks could be to include these sidework tasks in the robot's set of main tasks; however, this approach would make the combined model larger and more challenging to solve. A direction for future work could involve coming up with planning strategies to attend to these sidework tasks while effectively addressing the main tasks.

Deploying the robot in a real restaurant setting The ultimate goal of our research is to enable a service robot to perform tasks in a real restaurant by taking care of an ongoing stream of requests efficiently and respond to the customers appropriately. Our research does not focus on inferring the customers state from sensory observations. Our future efforts could involve integrating our planner with models that infer some of those observable variables, *e.g.*, the current request/activity, from data [180].

Our example implementation of the restaurant domain might be insufficient for a real restaurant setting with tasks and human interactions. In this work, we make the following simplifications. First, we assume that the information from different customers is integrated, and we only consider having access to the information for a given table. In a real restaurant setting, making decisions based on aggregated information might not be preferred. Second, we only focus on task-planning for a restaurant setting and assume that we have access to a path to the tables. In general, the task and path planning aspects of the problem might need to be integrated to account for low-level information (*e.g.*, customers blocking the robot's path) that could help with making better high-level decisions. Finally, there are also other challenges that come up when we include perception and execution modules in our robotic system such as receiving inconsistent observations or action failures. In this thesis, we mostly evaluated our algorithms in a simulated restaurant setting. A more in-depth study is needed to analyze the complexity of the problems that our robotic system, including the planning, perception, and execution modules, can address.

9.4 Summary

In this thesis, we contribute novel algorithms for domains where a robot should accomplish multiple tasks by switching between them. A conventional approach to deal with these multi-task problems is to combine all the tasks' states and robot actions into one large model and compute an optimal policy for this combined model. For the problems that we are interested in, the number of tasks can be large making this planning approach computationally impractical and challenging. We provide an algorithm that expedites task execution by solving the combined model associated with all the tasks less often. We then formalize a general class of problems where a robot is required to accomplish a set of tasks that are partially observable and evolve independently of each other according to their dynamics. We formalize the restaurant domain and define the assumptions under which the restaurant domain can be an instance of this class of problems. We present methods that exploit the structure found in these problems, namely the independence between the tasks, to optimally and efficiently plan for short and long planning horizons. Our key ideas include decomposing the problem into a series of much smaller planning problems, and computing lower and upper-bounds on the value of an optimal solution for variable horizons. We then focus on real-world applications such as the restaurant domain where having an exact and comprehensive model that works for all the tables is infeasible. As a result, discrepancies could arise that prevent the robot from attending to all the tables. We discuss how we formulate the discrepancies as a robot planning problem. We then explain how we tackle the discrepancies by augmenting the planning problem's state and action space with a set of hypotheses and questions regarding the discrepancies that aim at finding where the potential inaccuracies in the original planning model lie. Finally, we provide algorithms to solve the augmented planning problem more efficiently for single-task and multi-task settings. We present the algorithms, analyze their theoretical properties, and demonstrate their effectiveness on a grid environment and the waiting tables domain.

Appendix A

Robot Experiments

In this appendix, we give an overview of the CoBot robots and the tasks they can perform with a focus on the restaurant setting (Fig. A.1). The CoBot robots are developed in the CORAL laboratory as part of the research of many past students, including: Mike Licitra, who physically built the robots; Joydeep Biswas [20], Brian Coltin [46], and Stephanie Rosenthal [155], who laid the foundation for the complete navigation, task execution, and symbiotic autonomy of the robots.

The CoBot robots navigate smoothly and quickly due to their omnidirectional bases. All the computation is done on an onboard tablet or laptop computer. They have LIDAR and Kinect sensors to localize and detect obstacles, a touchscreen to interact with humans, and speakers for voice interactions. For sensing, the robots use a combination of planar LIDAR and an RGB-D camera. The CoBots autonomously localize and navigate using depth-camera and LIDAR-based localization and navigation algorithms [23]. The CoBots autonomously avoid obstacles by moving to the side of the hallway, but if they cannot avoid an obstacle, they stop and say "Please excuse me." until the obstacle is moved. For the things the robots cannot do, the robots ask humans in the building for help [155]. For example, the CoBots do not have arms, so they ask humans to press the elevator buttons for them. These functionalities were developed to enable the CoBot robots to perform user-requested tasks in office buildings. In this work, we will discuss the functionalities that we used in our robot experiments to enable the CoBot robots to service the tables in a restaurant setting.

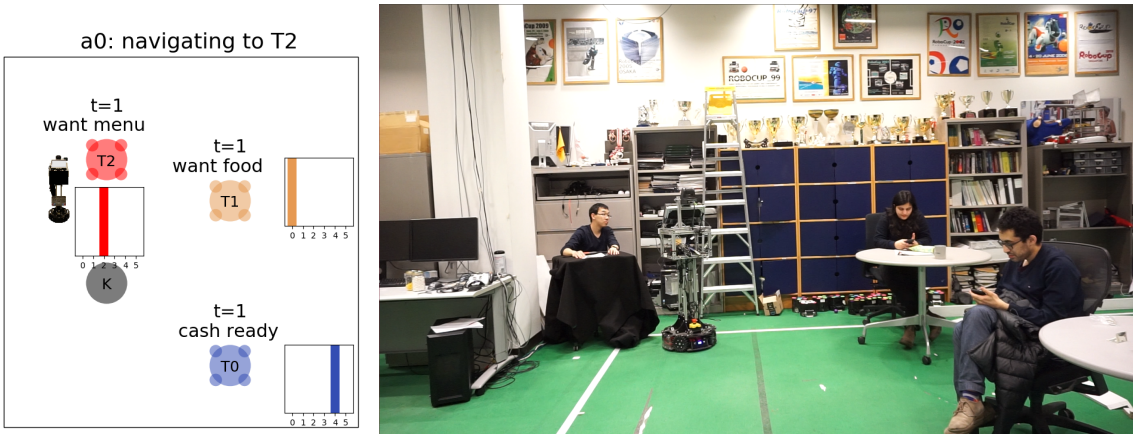
The software architecture on the CoBots is designed to be modular and easily extensible. The CoBot robots have multiple software nodes, including the robot hardware drivers, localization, navigation, the graphical user interface, the task planner, the server interface, and the scheduling server [20]. The software nodes communicate with each other using ROS, and specifically through topics and services [150]. In this thesis, we use the localization and navigation software modules, and we will discuss them in more details later.

Figure A.1: CoBot mobile service robots.

The CoBot robots offer multiple tasks to their users. A task refers to a function that the robot can execute and takes a fixed number of arguments as its input. The CoBot robots offer their users four tasks: `GoTo`, `PickUpAndDelivery`, `Escort`, and `MessageDelivery`. Each of the tasks that the CoBot robots can execute involves driving to one or more locations. A full description of the tasks is in [142]. In this thesis, we will go through the `GoTo` task which is used to go to the different tables in the restaurant. `GoTo` requires a single location argument as the robot's destination. To execute this task, the robot drives from its current position to the specified destination. We use the POMDP planning approaches that we describe in this thesis to decide what action the robot should perform next. The `GoTo` task is leveraged to navigate between the multiple tables. We will explain the different components of our robot experiments, namely, the restaurant model, perception module, task planning module and execution module next.

A.1 Restaurant Model

Our restaurant setup has three tables with one person on each table as shown in Fig. A.2 and one robot. In our restaurant setting, the positions that the robot navigates to, the kitchen and the tables, are hard-coded. Each table follows the POMDP model from Chapter 3. We added two new actions to each POMDP model, *go to the kitchen* and *pick up food for a table*. The kitchen is shown with k in Fig. A.2a. We also added a new state variable *food_pickedup* to each POMDP model that shows if the food or drink is picked up by the robot. We create a robot simulator with the POMDP models associated with the 3 tables.



(a) Simulation (b) Real-world

Figure A.2: A restaurant setting with 3 tables ($T0$, $T1$ and $T2$) and one robot.

A.2 Perception

The perception module has two components, a component for sensing the tables' states and another component for sensing the state of the robot.

We use the robot simulator which uses the POMDP model for each table to generate the observations associated with the state of the table. After each action execution, the simulator considers a set of possible observations given the robot's current belief state and the executed action. The robot then selects a random observation from this set. For the example scenario that we show later (based on the POMDP model in Chapter 3), we assume that all the following state variables: food, water, cooking status, current request, hand raise, time since food or water has been served, time since food is ready, and time since request are fully observable, and the satisfaction level is the only hidden state variable. There is no stochasticity in terms of the 8 observable state variables. Thus, after each action execution, the simulator just outputs the updated values for those 8 state variables as the observation. Our future efforts could involve inferring some of those observable variables, *e.g.*, the current request/activity, from input data [180].

The CoBot robots use a Vector Map to localize. The Vector Map is a representation of the layout of the building stored in vector form; that is, each constituent segment of the map is stored, using a pair of $2D$ points described by their (x, y) coordinates. The blue lines in Fig. A.3 show a plotting of the vectors that are stored in the Vector Map file. The robot uses this Vector Map to localize correctly. The robot uses the reading from its sensors (Lidar and Kinect) to find planar surfaces, matches these planar surfaces to walls described in the Vector Map and continuously updates its position [23].

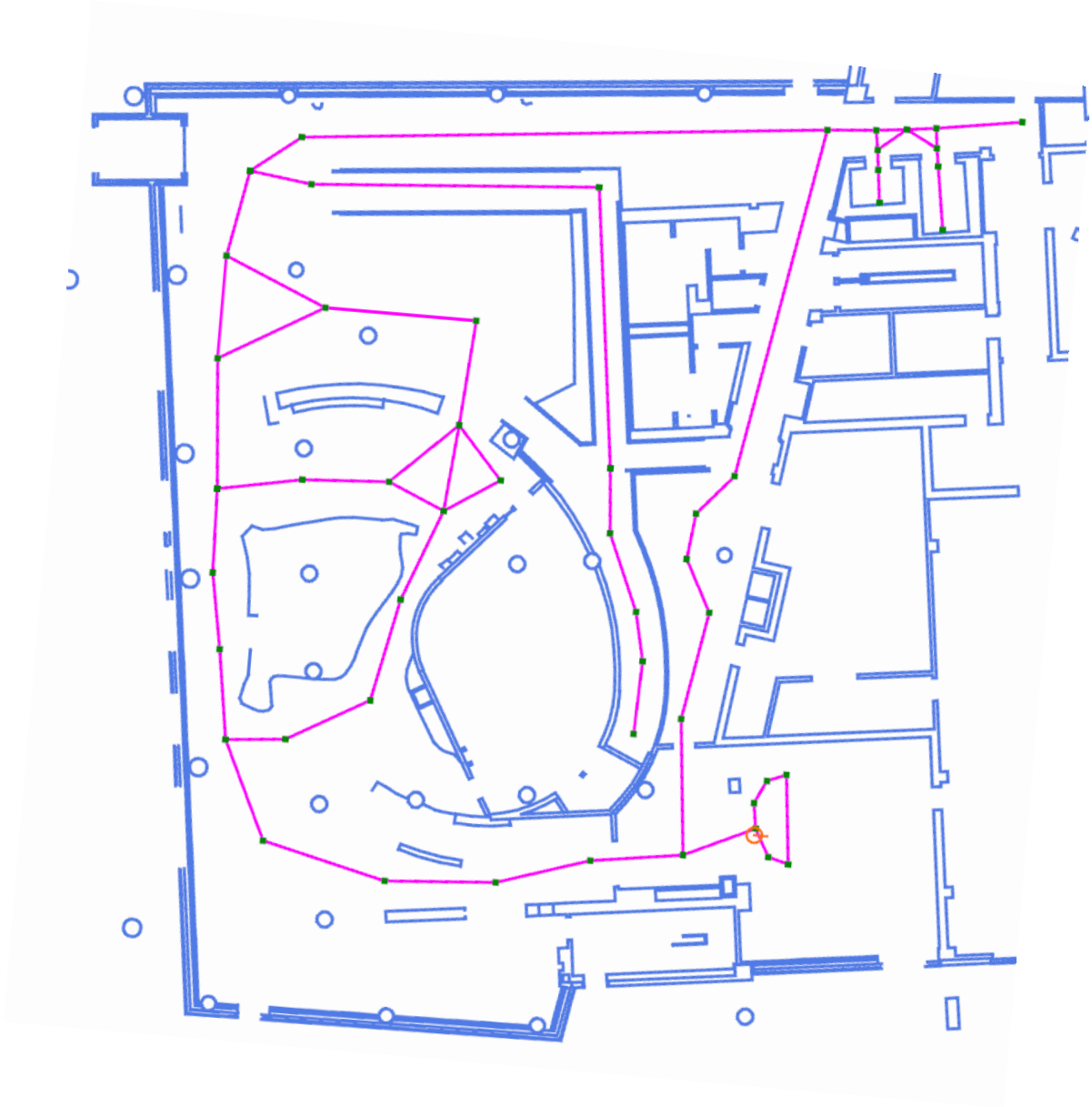


Figure A.3: The Navigation Map of the Gates Hillman Center's 3rd floor used by the CoBot robots.

A.3 Task Planning

We use the POMDP solver from Chapter 5 and apply it on the restaurant model. The POMDP planner and the CoBot communicate through ROS topics. Whenever the POMDP planner selects an action to be executed by the robot, it publishes the action on a topic where the CoBot receives it. The robot then checks if the action is a navigation action, a service action, or a communication action and executes it. The planners from Chapter 6 and Chapter 7 could also be easily replaced with this planner to solve longer horizon problems and address the discrepancies that arise.

A.4 Execution

There are 3 types of actions that the robot can execute, navigation action, service action, or communication action. For the navigation actions, the CoBot robots use a Navigation Graph to navigate around the building. The Navigation Graph stores the information the robot needs to move around the environment described by the Vector Map. The vertices of the Navigation Graph are points on the map, identified by their (x, y) coordinates, and edges are straight lines connecting them. The navigation graph is stored by the robot as a binary file. Fig. A.3 shows the plotting of the Navigation Graph for the Gates Hillman Center's 3rd floor with green vertexes and pink edges overlaid on the Vector Map. The robot is shown with an orange circle, and its direction is shown with an orange line. The robots use the Navigation Graph when they must move to a given destination. To reach a location (x, y), the robot first identifies and drives to the closest point on the Navigation Graph, and then, the robot computes a path, via the graph, to the point closest to its destination. Finally, the robot drives, in a straight line, from the point on the Navigation Graph to its desired position. Given a destination location, the navigation planner first finds the projected destination location that lies on one of the edges in the navigation graph. This projected destination location is then used to compute a topological policy using Dijkstra's algorithm for the entire graph. The navigation planner projects the current location onto the graph and then executes the topological policy until the robot reaches the edge on which the destination lies, and then drives straight to the destination location [22]. We use the navigation graph to navigate between the multiple tables in the restaurant. Fig. A.4 shows the plotting of the Navigation Graph for the room where we set up the restaurant setting with 3 tables.

CoBots take a conservative approach to navigation. In particular, the obstacle-avoidance algorithm uses a local greedy planner, which assumes that paths on the navigation graph will always be navigable and can be blocked only by humans. As a result, the planner will not consider an alternative route if a path is blocked, but will stop before the humans and ask to be

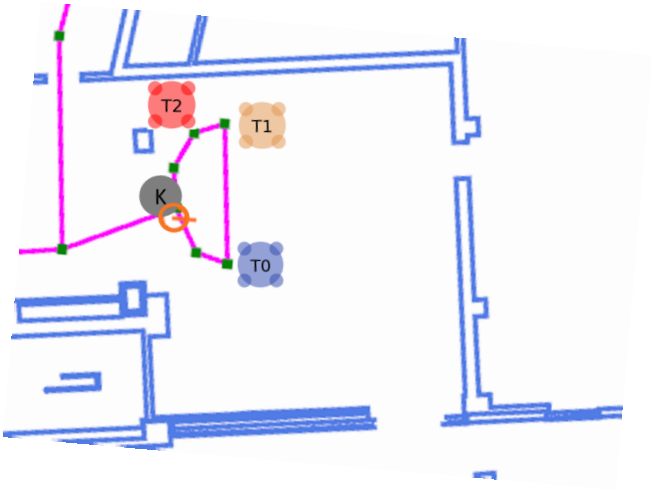


Figure A.4: The Navigation Map of the room where we set up the restaurant setting with the tables and the kitchen overlaid on it.

excused. When the robot finds an obstacle on its path, the robot stops, and requests passage, saying "Please excuse me.". Upon arrival at its destination, the robot announces that it has completed its navigation to the POMDP planner. The robot's position is updated as the robot moves in the environment using its localization module.

If the action is a communication action, the robot uses its speaker to say the message associated with the communication action, *e.g.*, "your food is not ready". Since CoBot does not have arms to manipulate the world, it relies on human help for some of its capabilities, *e.g.*, placing the food on the table or handing over the menu. If the action is a service action, the robot asks for help from the humans to perform the action, *e.g.*, "please take the menu" and "please take your food".

A.5 Example Scenario

Here[1] is a video of one example scenario using our setup. As mentioned earlier, in our restaurant setting, we hard-code the coordinates (x, y, θ) for the kitchen and the tables. The image on the top-left of the video is a top-view image of the restaurant with tables $T0$, $T1$ and $T2$. The status of the tables, wait time, current request and the belief over the satisfaction level is shown in the top-left image. The robot attends to the tables by executing the actions that the POMDP planner selects until all the customers leave the restaurant. As the robot services the tables, the tables' satisfaction level increases. As the customers wait to be served, their satisfaction level decreases.

The robot takes an interleaved planning and execution approach where it plans for a fixed short

[1] https://youtu.be/cq2TpFoPc60

finite horizon of 4, executes the action and replans. The robot's action is shown on the bottom of the video. Each table goes through a sequence of 8 requests from `want menu` to `clean table`. The customers' hand raises in the video is just for the sake of the video, and the robot is not sensing any of the hand raises. In the model, the customers need to be attended to by the robot at all times unless they are eating their food or drinking. *I.e.*, if the tables' current request is not `eating` or `drinking`, the customers need a service from the robot.

Fig. A.5 shows snapshots of the video. We randomly initialize the status of the tables. The customer on $T0$ has his money ready for the robot to pick up, and the robot believes that this customer is `satisfied` with probability 1.0. The customer on $T1$ is waiting for her food, and the robot believes that this customer is `very unsatisfied` with probability 1.0. The customer on $T2$ wants the menu, and the robot believes that this customer is `slightly unsatisfied` with probability 1.0. All the wait times are 0 at the beginning. The belief state of the robot includes the status of all the tables.

Given the current belief state, the POMDP planner selects action `Go to Table 2` as shown in Fig. A.5a. The robot then uses its navigation graph and localization module to go to $T2$. The robot then plans its next action (Fig. A.5b). The robot asks the customer to take the menu (not shown) and then asks what his order is (Fig. A.5c). Consequently, the satisfaction level of the customer on $T2$ increases. Since the customers on $T0$ and $T1$ are waiting, their satisfaction level decreases. The robot then decides to go to $T1$ and perform the communication action "your food is not ready" to keep the customer informed and increase her satisfaction level (Fig. A.5d). While the robot is attending to $T1$, the satisfaction level of the customers on $T0$ and $T2$ decreases. The robot then goes to $T0$ to obtain the money from that customer (Fig. A.5e and Fig. A.5f).

As mentioned earlier, the customers' satisfaction level depends on their wait time, and the reward function depends on the customers' satisfaction level and how long they have been waiting. Thus, the robot attends to the multiple tables to increase their satisfaction levels and reset their wait times. The robot keeps switching between the multiple tables to attend to them until all the customers leave.

(a) The robot goes to $T2$.

(b) The robot is planning its next action.

(c) The robot asks what the customer's order is.

(d) The robot then goes to $T1$ and performs the communication action `your food is not ready`.

(e) The robot goes to $T0$.

(f) The robot asks the customer to place the money in the basket.

Figure A.5: Snapshots of the robot video.

Bibliography

[1] David Abel, D Ellis Hershkowitz, and Michael L Littman. Near optimal behavior via approximate state abstraction. *arXiv preprint arXiv:1701.04113*, 2017. 8.2.1

[2] Sergey Alatartsev, Vera Mersheeva, Marcus Augustine, and Frank Ortmeier. On optimizing a sequence of robotic tasks. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 217–223. IEEE, 2013. 8.1.1

[3] Sergey Alatartsev, Sebastian Stellmacher, and Frank Ortmeier. Robotic task sequencing problem: A survey. *Journal of intelligent & robotic systems*, 80(2):279–298, 2015. 8.1.1

[4] Gianluca Antonelli. A survey of fault detection/tolerance strategies for auvs and rovs. In *Fault diagnosis and fault tolerance for mechatronic systems: Recent advances*, pages 109–127. Springer, 2003. 8.3

[5] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006. 8.1.1

[6] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. 7.1

[7] Nicholas Armstrong-Crews and Manuela Veloso. Oracular partially observable markov decision processes: A very special case. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2477–2482. IEEE, 2007. 8.3.4

[8] Nicholas Armstrong-Crews and Manuela Veloso. An approximate algorithm for solving oracular pomdps. In *2008 IEEE International Conference on Robotics and Automation*, pages 3346–3352. IEEE, 2008. 1.3, 8.3.4

[9] Amin Atrash and Joelle Pineau. A bayesian method for learning pomdp observation parameters for robot interaction management systems. In *The POMDP practitioners workshop*, 2010. 8.3.2, 8.3.4

[10] P. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *AAAI*, 2017. 1.2.1, 8.2.1, 8.2.3

[11] R Iris Bahar, Erica A Frohm, Charles M Gaona, Gary D Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal methods in system design*, 10(2-3):171–206, 1997. 8.2.1, 8.2.1

[12] Haoyu Bai, Shaojun Cai, Nan Ye, David Hsu, and Wee Sun Lee. Intention-aware online pomdp planning for autonomous driving in a crowd. In *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2015. 3.1, 8.1.1

[13] Tirthankar Bandyopadhyay, Kok Sung Won, Emilio Frazzoli, David Hsu, Wee Sun Lee, and Daniela Rus. Intention-aware motion planning. In *Algorithmic foundations of robotics X*, pages 475–491. Springer, 2013. 8.1.1

[14] Samuel Barrett, Noa Agmon, Noam Hazon, Sarit Kraus, and Peter Stone. Communicating with unknown teammates. In *ECAI*, pages 45–50, 2014. 8.1.1

[15] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 2003. 1.2.1, 8.2.1

[16] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial intelligence*, 72(1-2):81–138, 1995. 2.2.2

[17] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957. 2.1

[18] Dimitri P Bertsekas and John N Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16(3):580–595, 1991. 2.2.2

[19] B. Bethke, J. P. How, and A. Ozdaglar. Approximate dynamic programming using support vector regression. In *CDC*, 2008. 4.3.4

[20] Joydeep Biswas. Vector map-based, non-markov localization for long-term deployment of autonomous mobile robots. 2014. A, A

[21] Joydeep Biswas and Manuela Veloso. Episodic non-markov localization: Reasoning about short-term and long-term features. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3969–3974. IEEE, 2014. 8.1.2

[22] Joydeep Biswas and Manuela M Veloso. Localization and navigation of the cobots over long-term deployments. *The International Journal of Robotics Research*, 32(14):1679–1694, 2013. 5.4.4, A.4

[23] Joydeep Biswas and Manuela M Veloso. Episodic non-markov localization. *Robotics and Autonomous Systems*, 87:162–176, 2017. 5.4.4, 5.4.4, A, A.2

[24] Marcus Bjäreland. Model-based execution monitoring. In *Linköping Studies in Science and Technology, Dissertation No 688, available at http://www. ida. liu. se/labs/kplab/people/marbj*. Citeseer, 2001. 8.3, 8.3.1

[25] Blai Bonet and Héctor Geffner. Learning in depth-first search: A unified approach to heuristic search in deterministic, non-deterministic, probabilistic, and game tree settings. Technical report, Technical report, Universidad Simon Bolivar, 2005. Available at https . . . , 2005. 2.2.2

[26] Blai Bonet and Hector Geffner. Solving pomdps: Rtdp-bel vs. point-based algorithms. In *IJCAI*, pages 1641–1646. Pasadena CA, 2009. 2.2.2, 7.2.1, 7.4

[27] Richard J Boucherie and Nico M Van Dijk. *Markov decision processes in practice*. Springer, 2017. 8.1.1

[28] Craig Boutilier. Approximately optimal monitoring of plan preconditions. *arXiv preprint arXiv:1301.3839*, 2013. 8.3.1

[29] Craig Boutilier and Tyler Lu. Budget allocation using weakly coupled, constrained markov

decision processes. 2016. 8.2.2

[30] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1168–1175. Citeseer, 1996. 8.2.1

[31] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11: 1–94, 1999. 8.2.1, 8.3.3

[32] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial intelligence*, 121(1-2):49–107, 2000. 8.2.1

[33] Frank Broz et al. *Planning for human-robot interaction: representing time and human intention*. PhD thesis, Carnegie Mellon University, The Robotics Institute, 2008. 8.1.1

[34] Maya Cakmak and Andrea L Thomaz. Designing robot learners that ask good questions. In *2012 7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 17–24. IEEE, 2012. 8.3.4

[35] Anthony R. Cassandra. Russell and norvig's 4x3 maze. [Maze domain definition]. URL `http://cs.brown.edu/research/ai/pomdp/examples/4x3.95.POMDP`. 7.1

[36] Anthony R Cassandra. A survey of pomdp applications. In *Working notes of AAAI 1998 fall symposium on planning with partially observable Markov decision processes*, 1998. 3.1, 5.1, 8.1.1

[37] Anthony Rocco Cassandra. Exact and approximate algorithms for partially observable markov decision processes. 1998. 8.1.1

[38] Laurent Charlin, Pascal Poupart, and Romy Shioda. Automated hierarchy discovery for planning in partially observable environments. In *Advances in Neural Information Processing Systems*, pages 225–232, 2007. 8.2.3

[39] Luefeng Chen, Min Wu, Mengtian Zhou, Jinhua She, Fangyan Dong, and Kaoru Hirota. Information-driven multirobot behavior adaptation to emotional intention in human-robot interaction. *IEEE Transactions on Cognitive and Developmental Systems*, 2018. 3.1, 8.1.2

[40] YiChun Chen, Mykel J Kochenderfer, and Matthijs TJ Spaan. Improving offline value-function approximations for pomdps by reducing discount factors. In *IROS*, 2018. 8.2.1

[41] Sonia Chernova and Andrea L Thomaz. Robot learning from human teachers. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(3):1–121, 2014. 1.3, 8.3.4

[42] Sonia Chernova and Manuela Veloso. Interactive policy learning through confidence-based autonomy. *Journal of Artificial Intelligence Research*, 34:1–25, 2009. 8.3.4

[43] D. Choi. Reactive goal management in a cognitive architecture. *Cognitive Systems Research*, 2011. 8.2.2

[44] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine learning*, 15(2):201–221, 1994. 8.3.4

[45] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical

models. *Journal of artificial intelligence research*, 4:129–145, 1996. 8.3.4

[46] Brian Coltin. Multi-agent pickup and delivery planning with transfers. 2014. A

[47] Brian Coltin and Manuela Veloso. Multi-observation sensor resetting localization with ambiguous landmarks. *Autonomous robots*, 35(2):221–237, 2013. 1.3, 7.1

[48] M-O Cordier, Philippe Dague, François Lévy, Jacky Montmain, Marcel Staroswiecki, and Louise Travé-Massuyès. Conflicts versus analytical redundancy relations: a comparative analysis of the model based diagnosis approach from the artificial intelligence and automatic control perspectives. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(5):2163–2177, 2004. 8.3

[49] William Cushing and Subbarao Kambhampati. Replanning: A new perspective. *Proceedings of the International Confer-ence on Automated Planning and Scheduling. Monterey, USA*, pages 13–16, 2005. 8.3.1

[50] Patrick Dallaire, Camille Besse, Stephane Ross, and Brahim Chaib-draa. Bayesian reinforcement learning in continuous pomdps with gaussian processes. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2604–2609. IEEE, 2009. 8.3.2

[51] Johan De Kleer, Alan K Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial intelligence*, 56(2-3):197–222, 1992. 8.3

[52] Thomas Dean and Shieu-Hong Lin. Decomposition techniques for planning in stochastic domains. In *IJCAI*, volume 2, page 3. Citeseer, 1995. 8.2.3

[53] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1-3):61–95, 1991. 8.1.1

[54] Thomas Degris, Olivier Sigaud, and Pierre-Henri Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *Proceedings of the 23rd international conference on Machine learning*, pages 257–264, 2006. 8.2.1

[55] Karina V Delgado, Leliane N De Barros, Daniel B Dias, and Scott Sanner. Real-time dynamic programming for markov decision processes with imprecise probabilities. *Artificial Intelligence*, 230:192–223, 2016. 2.2.2, 8.3.3

[56] Sarang Deo, Seyed Iravani, Tingting Jiang, Karen Smilowitz, and Stephen Samuelson. Improving health outcomes through better capacity allocation in a community-based chronic care model. *Operations Research*, 61(6):1277–1294, 2013. 8.2.2

[57] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 2000. 8.2.1

[58] C. Diuk, A. Cohen, and M. L. Littman. An object-oriented representation for efficient reinforcement learning. In *ICML*, 2008. 8.2.1

[59] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009. 8.3

[60] Pinar Donmez and Jaime G Carbonell. Proactive learning: cost-sensitive active learning

with multiple imperfect oracles. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 619–628, 2008. 8.3.4, 9.3

[61] Finale Doshi, Joelle Pineau, and Nicholas Roy. Reinforcement learning with limited reinforcement: Using bayes risk for active learning in pomdps. In *Proceedings of the 25th international conference on Machine learning*, pages 256–263, 2008. 1.3, 7.1, 8.3.2, 8.3.4, 9.3

[62] Richard J Doyle, David J Atkinson, and Rajkumar S Doshi. Generating perception requests and expectations to verify the execution of plans. In *AAAI*, pages 81–88, 1986. 8.3

[63] Stefan Edelkamp, Morteza Lahijanian, Daniele Magazzeni, and Erion Plaku. Integrating temporal reasoning and sampling-based motion planning for multigoal problems with dynamics and time windows. *IEEE Robotics and Automation Letters*, 3(4):3473–3480, 2018. 8.1.1

[64] Sean P Engelson and Drew V McDermott. Error correction in mobile robot map learning. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pages 2555–2556. IEEE Computer Society, 1992. 7.1

[65] Jan Faigl, Vojtech Vonásek, and Libor Preucil. A multi-goal path planning for goal regions in the polygonal domain. In *ECMR*, pages 171–176, 2011. 8.1.1

[66] Amirmassoud Farahmand, Daniel Nikolaev Nikovski, Yuji Igarashi, and Hiroki Konaka. Truncated approximate dynamic programming with task-dependent terminal value. In *AAAI*, 2016. 8.2.1

[67] Eugene A Feinberg and Adam Shwartz. *Handbook of Markov decision processes: methods and applications*, volume 40. Springer Science & Business Media, 2012. 8.1.1

[68] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972. 8.3

[69] Amalia Foka and Panos Trahanias. Real-time hierarchical pomdps for autonomous robot navigation. *Robotics and Autonomous Systems*, 55(7):561–571, 2007. 8.2.1

[70] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In *ICAPS*, volume 6, pages 212–221, 2006. 8.3.1

[71] Mikhail Frank, Jürgen Leitner, Marijn Stollenga, Alexander Förster, and Jürgen Schmidhuber. Curiosity driven reinforcement learning for motion planning on humanoids. *Frontiers in neurorobotics*, 7:25, 2014. 8.3.2

[72] Christian Wilhelm Fritz. *Monitoring the generation and execution of optimal plans*. PhD thesis, 2009. 7.1, 8.3, 8.3.1

[73] P. Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine learning*, 2006. 4.3.2

[74] Robert Givan, Sonia Leach, and Thomas Dean. Bounded parameter markov decision processes. In *European Conference on Planning*, pages 234–246. Springer, 1997. 8.3.3

[75] Kevin D Glazebrook, Diego Ruiz-Hernandez, and Christopher Kirkbride. Some indexable families of restless bandit problems. *Advances in Applied Probability*, 38(3):643–672,

2006. 8.2.2

[76] Keith Golden, Oren Etzioni, and Daniel Weld. Planning with execution and incomplete information. Technical report, Technical report, Dept of Computer Science, University of Washington, TR96-01-09, 1996. 8.3.1

[77] Carlos Guestrin, Daphne Koller, Chris Gearhart, and Neal Kanodia. Generalizing plans to new environments in relational mdps. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1003–1010, 2003. 8.2.1

[78] Eric A Hansen and Rong Zhou. Synthesis of hierarchical finite-state controllers for pomdps. In *ICAPS*, pages 113–122, 2003. 8.2.1

[79] Eric A Hansen and Shlomo Zilberstein. Lao*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001. 2.2.2, 7.2.3, 7.2.3, 7.3.1, 4, 8, 7.5.1

[80] Milos Hauskrecht. Value-function approximations for partially observable markov decision processes. *Journal of artificial intelligence research*, 13:33–94, 2000. 2.2.1

[81] Carlos Hernández and Pedro Meseguer. Lrta*(k). In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 1238–1243, 2005. 2.2.2

[82] Todd Hester and Peter Stone. Intrinsically motivated model learning for a developing curious agent. In *2012 IEEE international conference on development and learning and epigenetic robotics (ICDL)*, pages 1–6. IEEE, 2012. 8.3.2

[83] M. Humphrys. Action selection methods using reinforcement learning. *From Animals to Animats*, 1996. 4.2.1

[84] Inseok Hwang, Sungwan Kim, Youdan Kim, and Chze Eng Seah. A survey of fault detection, isolation, and reconfiguration methods. *IEEE transactions on control systems technology*, 18(3):636–653, 2009. 8.3

[85] Félix Ingrand and Malik Ghallab. Robotics and artificial intelligence: A perspective on deliberation functions. *Ai Communications*, 27(1):63–80, 2014. 8.3

[86] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017. 8, 9.2

[87] Hideaki Itoh and Kiyohiko Nakamura. Partially observable markov decision processes with imprecise parameters. *Artificial Intelligence*, 171(8-9):453–490, 2007. 8.3.3

[88] Garud N Iyengar. Robust dynamic programming. *Mathematics of Operations Research*, 30 (2):257–280, 2005. 8.3.3

[89] U. Jaidee, H. Muñoz-Avila, and D. W. Aha. Learning and reusing goal-specific policies for goal-driven autonomy. In *ICCBR*, 2012. 1.2.1, 8.2.2

[90] Robin Jaulmes, Joelle Pineau, and Doina Precup. Probabilistic robot planning under model uncertainty: an active learning approach. In *NIPS Workshop on Machine Learning Based Robotics in Unstructured Environments*, 2005. 1.3, 7.1, 8.3.2, 8.3.4, 9.3

[91] Shervin Javdani, Siddhartha S Srinivasa, and J Andrew Bagnell. Shared autonomy via hindsight optimization. *Robotics science and systems: online proceedings*, 2015, 2015.

8.1.1

[92] Nan Jiang, Alex Kulesza, Satinder Singh, and Richard Lewis. The dependence of effective planning horizon on model accuracy. In *AAMAS*, 2015. 8.2.1

[93] Nan Jiang, Satinder P Singh, and Ambuj Tewari. On structural properties of mdps that bound loss due to shallow planning. In *IJCAI*, 2016. 8.2.1

[94] Hyun-Wook Jo, Jae-Ho Ahn, Jun-Sang Park, Jun-Han Oh, and Jong-Tae Lim. Task planning for service robots with optimal supervisory control. In *2010 IEEE Conference on Robotics, Automation and Mechatronics*. IEEE, 2010. 3.1, 8.1.2

[95] Anders Jonsson and Andrew Barto. Causal graph based decomposition of factored mdps. *Journal of Machine Learning Research*, 7(Nov):2259–2301, 2006. 8.2.3

[96] J. Karlsson. *Learning to solve multiple goals*. PhD thesis, University of Rochester, 1997. 4.2.1

[97] Seyedeh N. Khatami and Chaitra Gopalappa. A reinforcement learning model to inform optimal decision paths for hiv elimination1. *medRxiv*, 2021. doi: 10.1101/2021.07.11.21260328. URL https://www.medrxiv.org/content/early/2021/07/14/2021.07.11.21260328. 8.1.1

[98] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 4.3.1

[99] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 2013. 4.2.1, 8.1.1

[100] Sven Koenig, David Furcy, and Colin Bauer. Heuristic search-based replanning. In *AIPS*, pages 294–301, 2002. 8.3.1

[101] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artificial Intelligence*, 155(1-2):93–146, 2004. 8.3.1

[102] Ewa Kolakowska, Stephen F Smith, and Morten Kristiansen. Constraint optimization model of a scheduling problem for a robotic arm in automatic systems. *Robotics and Autonomous Systems*, 62(2):267–280, 2014. 8.1.1

[103] G. Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *IJCAI*, 2016. 1.2.1, 8.2.1, 8.2.3

[104] Richard E Korf. Real-time heuristic search. *Artificial intelligence*, 42(2-3):189–211, 1990. 2.2.2

[105] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *NIPS*, 2016. 1.2.1, 8.2.1, 8.2.3

[106] TK Satish Kumar, Marcello Cirillo, and Sven Koenig. On the traveling salesman problem with simple temporal constraints. In *Tenth Symposium of Abstraction, Reformulation, and Approximation*, 2013. 8.1.1

[107] Hanna Kurniawati, David Hsu, and Wee Sun Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *RSS*, 2008. 8.2.1

[108] Hanna Kurniawati, Yanzhu Du, David Hsu, and Wee Sun Lee. Motion planning under uncertainty for robotic tasks with long time horizons. *IJRR*, 2011. 8.2.1

[109] Pierre Laroche, Yann Boniface, and René Schott. A new decomposition technique for solving markov decision processes. In *Proceedings of the 2001 ACM symposium on applied computing*, pages 12–16, 2001. 8.2.3

[110] Lucas Lehnert, Romain Laroche, and Harm van Seijen. On value function representation of long horizon problems. In *AAAI*, 2018. 8.2.1

[111] Scott Lenser and Manuela Veloso. Sensor resetting localization for poorly modelled mobile robots. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 1225–1232. IEEE, 2000. 1.3, 7.1, 7.1, 8.3.1, 9.2.1, 9.3

[112] Steven James Levine. *Monitoring the execution of temporal plans for robotic systems*. PhD thesis, Massachusetts Institute of Technology, 2012. 8.3

[113] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. In *ISAIM*, 2006. 8.2.1

[114] Xin Li, William K Cheung, and Jiming Liu. Improving pomdp tractability via belief compression and clustering. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 40(1):125–136, 2009. 8.2.1

[115] Keqin Liu and Qing Zhao. Indexability of restless bandit problems and optimality of whittle index for dynamic multichannel access. *IEEE Transactions on Information Theory*, 56(11): 5547–5567, 2010. 8.2.2

[116] Owen Macindoe, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Pomcop: Belief space planning for sidekicks in cooperative games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012. 8.1.1

[117] E. Martinson, A. Stoytchev, and R. C. Arkin. Robot behavioral selection using q-learning. Technical report, Georgia Institute of Technology, 2001. 1.2.1, 8.2.2

[118] M. J. Matarić and F. Michaud. Behavior-based systems. In *Springer Handbook of Robotics*. 2008. 1.2.1, 8.2.2

[119] R McCallum. Reinforcement learning with selective perception and hidden state. 1997. 1.3, 7.1, 8.3.2

[120] Sheila McIlraith. Diagnosing hybrid systems: A bayesian model selection approach. In *Proceedings of the Eleventh International Workshop on Principles of Diagnosis (DX'00)*, pages 140–146, 2000. 8.3.1, 9.3

[121] Neville Mehta, Soumya Ray, Prasad Tadepalli, and Thomas Dietterich. Automatic discovery and transfer of maxq hierarchies. In *Proceedings of the 25th international conference on Machine learning*, pages 648–655, 2008. 8.2.3

[122] Juan Pablo Mendoza. Regions of inaccurate modeling for robot anomaly detection and model correction. 2017. 8.3, 9.2.1

[123] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. P. Kaelbling, T. L. Dean, and

C. Boutilier. Solving very large weakly coupled markov decision processes. In *AAAI/IAAI*, 1998. 8.2.2

[124] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. *arXiv preprint arXiv:1301.6721*, 1999. 8.3.2

[125] V. Mnih, K. Kavukcuoglu, and D. Silver et al. Human-level control through deep reinforcement learning. *Nature*, 2015. 2.1, 4.2.1, 4.3.1

[126] Anahita Mohseni-Kabir, David Isele, and Kikuo Fujimura. Interaction-aware multi-agent reinforcement learning for mobile agents with individual goals. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3370–3376. IEEE, 2019. 9.3

[127] Anahita Mohseni-Kabir, Manuela Veloso, and Maxim Likhachev. Efficient robot planning for achieving multiple independent partially observable tasks that evolve over time. In *ICAPS*, 2020. 6.3.2, 6.3.2, 6.3.2, 6.4, 2

[128] H. Muñoz-Avila, M. A. Wilson, and D. W. Aha. Guiding the ass with goal motivation weights. In *Goal Reasoning: Papers from the ACS Workshop*, 2015. 1.2.1, 8.2.2

[129] Lu Na and Lu Fei. Robot multi-tasks optimization using improved jshop2 planner. *International Journal of Control and Automation*, 2015. 3.1, 8.1.2

[130] Ali Nasir. *Comprehensive Fault Tolerance and Science-Optimal Attitude Planning for Spacecraft Applications.* PhD thesis, 2012. 8.3.1

[131] Bernhard Nebel and Jana Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial intelligence*, 76(1-2):427–454, 1995. 8.3.1

[132] Yaodong Ni and Zhi-Qiang Liu. Bounded-parameter partially observable markov decision processes. In *ICAPS*, pages 240–247, 2008. 1.3, 8.3.3

[133] M. Nicolescu, O. C. Jenkins, and A. Olenderski. Learning behavior fusion estimation from demonstration. In *ROMAN*, 2006. 8.2.2

[134] Stefanos Nikolaidis and Julie Shah. Human-robot teaming using shared mental models. *ACM/IEEE HRI*, 2012. 8.1.1

[135] Stefanos Nikolaidis, Yu Xiang Zhu, David Hsu, and Siddhartha Srinivasa. Human-robot mutual adaptation in shared autonomy. In *2017 12th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 2017. 3.1, 8.1.1

[136] Stefanos Nikolaidis, Minae Kwon, Jodi Forlizzi, and Siddhartha Srinivasa. Planning with verbal communication for human-robot collaboration. *ACM Transactions on Human-Robot Interaction (THRI)*, 7(3):1–21, 2018. 8.1.1

[137] Arnab Nilim and Laurent El Ghaoui. Robustness in markov decision problems with uncertain transition matrices. In *Advances in neural information processing systems*, pages 839–846, 2004. 8.3.3

[138] Ernesto Nunes and Maria Gini. Multi-robot auctions for allocation of tasks with temporal constraints. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. 8.1.1

[139] Takayuki Osogami. Robust partially observable markov decision process. In *International*

Conference on Machine Learning, pages 106–115, 2015. 8.3.3

[140] Sébastien Paquet, Brahim Chaib-draa, and Stéphane Ross. Hybrid pomdp algorithms. In *Proceedings of The Workshop on MSDM*, 2006. 9.3

[141] F. Pedregosa, G. Varoquaux, and A. Gramfort et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011. 2

[142] Vittorio Perera. *Language-Based Bidirectional Human and Robot Interaction Learning for Mobile Service Robots*. PhD thesis, University of Science and Technology of China, 2018. A

[143] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005. 8.3

[144] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. Point-based value iteration: An anytime algorithm for pomdps. In *IJCAI*, volume 3, pages 1025–1032, 2003. 2.2.1, 8.2.1

[145] Luis Enrique Pineda, Yi Lu, Shlomo Zilberstein, and Claudia V Goldman. Fault-tolerant planning under uncertainty. In *Twenty-Third International Joint Conference on Artificial Intelligence*. Citeseer, 2013. 8.3.1

[146] P. Pirjanian. Behavior coordination mechanisms-state-of-the-art. Technical report, University of Southern California, 1999. 1.2.1, 8.2.2

[147] Pascal Poupart and Craig Boutilier. Value-directed compression of pomdps. In *Advances in neural information processing systems*, pages 1579–1586, 2003. 8.2.1

[148] ML Puterman et al. Discrete stochastic dynamic programming, 1994. 2.1

[149] Yu Qing-xiao, Yuan Can, Fu Zhuang, and Zhao Yan-zheng. Research of the localization of restaurant service robot. *International Journal of Advanced Robotic Systems*, 2010. 3.1, 8.1.2

[150] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009. A

[151] C. Raïevsky and F. Michaud. Improving situated agents adaptability using interruption theory of emotions. In *SAB*, 2008. 1.2.1, 8.2.2

[152] Aditi Ramachandran, Sarah Strohkorb Sebo, and Brian Scassellati. Personalized robot tutoring using the assistive tutor pomdp (at-pomdp). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 8050–8057, 2019. 8.1.1

[153] Stephanie Rosenthal and Manuela Veloso. Modeling humans as observation providers using pomdps. In *2011 RO-MAN*, pages 53–58. IEEE, 2011. 1.3, 8.3.4

[154] Stephanie Rosenthal, Anind K Dey, and Manuela Veloso. How robots' questions affect the accuracy of the human responses. In *RO-MAN 2009-The 18th IEEE International Symposium on Robot and Human Interactive Communication*, pages 1137–1142. IEEE, 2009. 8.3.4

[155] Stephanie L Rosenthal. *Human-Centered Planning for Effective Tast Autonomy*. PhD thesis, Carnegie Mellon University, 2012. A, A

[156] Stephane Ross, Brahim Chaib-draa, and Joelle Pineau. Bayes-adaptive pomdps. In *Advances in neural information processing systems*, pages 1225–1232, 2008. 1.3, 7.1, 8.3.2

[157] Stephane Ross, Joelle Pineau, and Brahim Chaib-draa. Theoretical analysis of heuristic search methods for online pomdps. In *Advances in neural information processing systems*, pages 1233–1240, 2008. 2.2.1

[158] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-Draa. Online planning algorithms for pomdps. *Journal of Artificial Intelligence Research*, 32:663–704, 2008. 1.2.2, 2.2.1, 5.1, 7.3.3, 7.5.1, 9.3

[159] Stéphane Ross, Joelle Pineau, Brahim Chaib-draa, and Pierre Kreitmann. A bayesian approach for learning and planning in partially observable markov decision processes. *Journal of Machine Learning Research*, 12(May):1729–1770, 2011. 8.3.2

[160] Nicholas Roy, Geoffrey Gordon, and Sebastian Thrun. Finding approximate pomdp solutions through belief compression. *Journal of artificial intelligence research*, 23:1–40, 2005. 1.2.2, 5.1, 8.2.1, 9.3

[161] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice Hall Upper Saddle River, NJ, USA:, 2002. 7.1

[162] Andrei A Rusu, Matej Večerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *Conference on Robot Learning*, pages 262–270. PMLR, 2017. 9.2.2

[163] Sakari, Pieska, Van, Spiz, Juhana, Jauhiainen, Mika, and Luimula. Social service robots in wellness and restaurant applications. 2013. 8.1.2

[164] Marcel Schoppers. Universal plans for reactive robots in unpredictable environments. In *IJCAI*, volume 87, pages 1039–1046. Citeseer, 1987. 7.1, 8.3.1

[165] Devin Schwab. *Robot Deep Reinforcement Learning: Tensor State-Action Spaces and Auxiliary Task Learning with Multiple State Representations*. PhD thesis, Carnegie Mellon University, 2020. 9.2.2

[166] Guy Shani. Task-based decomposition of factored pomdps. *IEEE transactions on cybernetics*, 44(2):208–216, 2013. 5.1, 5.4, 6.4, 8.2.1, 8.2.2, 8.2.3

[167] Guy Shani, Ronen I Brafman, and Solomon E Shimony. Model-based online learning of pomdps. In *European Conference on Machine Learning*, pages 353–364. Springer, 2005. 8.3.2

[168] Guy Shani, Pascal Poupart, Ronen I Brafman, and Solomon Eyal Shimony. Efficient add operations for point-based algorithms. In *ICAPS*, pages 330–337, 2008. 1.2.2, 5.1, 8.2.1, 9.3

[169] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based pomdp solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, 2013. 1.2.2, 5.1, 8.2.1, 9.3

[170] Pranav Shyam, Wojciech Jaśkowski, and Faustino Gomez. Model-based active exploration. *arXiv preprint arXiv:1810.12162*, 2018. 8.3.2

[171] Rui Silva, Francisco S Melo, and Manuela Veloso. What if the world were different?

gradient-based exploration for new optimal policies. In *GCAI*, pages 229–242, 2018. 8.3.3

[172] Rui Silva, Gabriele Farina, Francisco S Melo, and Manuela Veloso. A theoretical and algorithmic analysis of configurable mdps. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 455–463, 2019. 8.3.3

[173] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, pages 2164–2172, 2010. 8.2.1

[174] Trey Smith and Reid Simmons. Heuristic search value iteration for pomdps. *Proc. Uncertainty in Artificial Intelligence*, 2004. 8.2.1

[175] Trey Smith, David R Thompson, and David Wettergreen. Generating exponentially smaller pomdp models using conditionally irrelevant variable abstraction. In *ICAPS*, pages 304–311, 2007. 1.2.2, 5.1, 8.1.1, 9.3

[176] Adhiraj Somani, Nan Ye, David Hsu, and Wee Sun Lee. Despot: Online pomdp planning with regularization. In *Advances in neural information processing systems*, pages 1772–1780, 2013. 6.5, 9.3

[177] Yi Sun, Faustino Gomez, and Jürgen Schmidhuber. Planning to be surprised: Optimal bayesian exploration in dynamic environments. In *International Conference on Artificial General Intelligence*, pages 41–51. Springer, 2011. 8.3.2

[178] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998. 2.1, 4.1, 4.3.4

[179] R. S. Sutton, S. P. Singh, D. Precup, and B. Ravindran. Improved switching among temporally abstract actions. In *NIPS*, 1999. 4.2.1, 8.2.1

[180] A. Taylor, R. Kaufman, S. Girdhar, and H. Admoni. Modeling human need for attention and interruptibility in restaurant scenarios. In *Proceedings of In AI x Food Workshop at IJCAI '19*, August 2019. 9.3, A.2

[181] Georgios Theocharous and Leslie P Kaelbling. Approximate planning in pomdps with macro-actions. In *Advances in Neural Information Processing Systems*, 2004. 1.2.2, 5.1, 8.2.1, 9.3

[182] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017. 9.2.2

[183] Marc Toussaint, Laurent Charlin, and Pascal Poupart. Hierarchical pomdp controller optimization by likelihood maximization. In *UAI*, volume 24, pages 562–570, 2008. 8.2.3

[184] J. N. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, 1997. 2.1

[185] Interaktives Planen unter unsicheren Bedingungen. Interactive planning under uncertainty. 2017. 8.3.4

[186] Erik P Vargo and Randy Cogill. Expectation-maximization for bayes-adaptive pomdps. *Journal of the Operational Research Society*, 66(10):1605–1623, 2015. 8.3.2

[187] S. Vattam, M. Klenk, M. Molineaux, and D. W. Aha. Breadth of approaches to goal reasoning: A research survey. Technical report, Naval Research Lab Washington DC, 2013. 1.2.1, 8.2.2

[188] M. Veloso, J. Biswas, B. Coltin, and S. Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *IJCAI*, 2015. 4.1

[189] Manuela M Veloso, Martha E Pollack, and Michael T Cox. Rationale-based monitoring for planning in dynamic environments. In *AIPS*, volume 171, page 180, 1998. 8.3

[190] Manuela M Veloso, Joydeep Biswas, Brian Coltin, Stephanie Rosenthal, Susana Brandao, Tekin Mericli, and Rodrigo Ventura. Symbiotic-autonomous service robots for user-requested tasks in a multi-floor building. 2012. 5.4.4

[191] Venkat Venkatasubramanian, Raghunathan Rengaswamy, and Surya N Kavuri. A review of process fault detection and diagnosis: Part ii: Qualitative models and search strategies. *Computers & chemical engineering*, 27(3):313–326, 2003. 8.3

[192] Vandi Verma, Geoff Gordon, Reid Simmons, and Sebastian Thrun. Real-time fault diagnosis [robot fault diagnosis]. *IEEE Robotics & Automation Magazine*, 11(2):56–66, 2004. 1.3, 7.1, 8.3.1

[193] Minlue Wang, Sebastien Canu, and Richard Dearden. Improving robot plans for information gathering tasks through execution monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5285–5291. IEEE, 2013. 8.3.1

[194] Yi Wang, Kok Sung Won, David Hsu, and Wee Sun Lee. Monte carlo bayesian reinforcement learning. *arXiv preprint arXiv:1206.6449*, 2012. 8.3.2

[195] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015. 2.1, 4.1, 4.2.1

[196] Richard Washington. On-board real-time state and fault identification for rovers. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 2, pages 1175–1181. IEEE, 2000. 8.3.1

[197] Richard R Weber and Gideon Weiss. On an index policy for restless bandits. *Journal of Applied Probability*, 27(3):637–648, 1990. 8.2.2

[198] Douglas J White. Further real applications of markov decision processes. *Interfaces*, 18(5): 55–61, 1988. 8.1.1

[199] Chelsea C White III and Hany K Eldeib. Markov decision processes with imprecise transition probabilities. *Operations Research*, 42(4):739–749, 1994. 1.3, 8.3.3

[200] Peter Whittle. Restless bandits: Activity allocation in a changing world. *Journal of applied probability*, 25(A):287–298, 1988. 8.2.1, 8.2.2

[201] Cristina M Wilcox and Brian C Williams. Runtime verification of stochastic, faulty systems. In *International Conference on Runtime Verification*, pages 452–459. Springer, 2010. 8.3

[202] Stefan Witwicki, Francisco Melo, Jesús Capitán, and Matthijs Spaan. A flexible approach

to modeling unpredictable events in mdps. In *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013. 8.3.3

[203] Kyle Hollins Wray and Shlomo Zilberstein. A pomdp formulation of proactive learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016. 8.3.4, 9.3

[204] Kyle Hollins Wray and Shlomo Zilberstein. Approximating reachable belief points in pomdps. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 117–122. IEEE, 2017. 8.2.1

[205] Christian Wurll and Dominik Henrich. Point-to-point and multi-goal path planning for industrial robots. *Journal of Robotic Systems*, 18(8):445–461, 2001. 8.1.1

[206] Qingxiao Yu, Can Yuan, Zhuang Fu, and Yanzheng Zhao. An autonomous restaurant service robot with high positioning accuracy. *Industrial Robot: An International Journal*, 2012. 3.1, 8.1.2

[207] Paraskevi Th Zacharia, Elias K Xidias, and Nikos A Aspragathos. Task scheduling and motion planning for an industrial manipulator. *Robotics and computer-integrated manufacturing*, 29(6):449–462, 2013. 8.1.1