Verified Multi-Robot Planning Under Uncertainty

by

Fatma Faruq
ORCID iD: 0000-0001-6928-0176

A thesis submitted to
the University of Birmingham and the University of Melbourne
for the degree of
Doctor of Philosophy

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
June 2021

Abstract

Multi-robot systems are being increasingly deployed to solve real-world problems, from warehouses to autonomous fleets for logistics, from hospitals to nuclear power plants and emergency search and rescue scenarios. These systems often need to operate in uncertain environments which can lead to robot failure, uncertain action durations or the inability to complete assigned tasks. In many scenarios, the safety or reliability of these systems is critical to their deployment. Therefore there is a need for *robust* multi-robot planning solutions that offer *guarantees* on the performance of the robot team. In this thesis we develop techniques for robust multi-robot task allocation and planning under uncertainty by building on techniques from formal verification.

We present three algorithms that solve the problem of task allocation and planning for a multi-robot team operating under uncertainty. These algorithms are able to calculate the expected maximum number of tasks the multi-robot team can achieve, considering the possibility of robot failure. They are also able to reallocate tasks when robots fail. We formalise the problem of task allocation and robust planning for a multi-robot team using Linear Temporal Logic to specify the team's mission and Markov decision processes to model the robots. Our first solution method is a sampling based approach to simultaneous task allocation and planning. Our second solution method separates task allocation and planning for the same problem using auctioning for the former. Our final solution lies midway between the first two using simultaneous task allocation and planning in a sequential team model. We evaluate all solution approaches extensively using a set of tests inspired by existing benchmarks in related fields.

To Abu and Nano, my stars in heaven

Acknowledgements

There is a long list of people who have supported me through my PhD and who deserve more than just my heartfelt gratitude. I have been extremely lucky to have wonderful supervisors (Dave Parker, Nick Hawes, Bruno Lacerda and Tim Miller) whose kindness and support, I can only hope to pay forward.

Dave was both a mentor and an ally, always ready with advice, support and encouragement. I could not have gotten through these four years without him. Despite moving to Oxford with Nick, Bruno was heavily invested in my research, always welcoming and supportive, helping me navigate through it all. Nick was my first point of contact for this PhD and I do not think I would be here without him. He had the right mix of practicality and conviviality to keep me on track. I would not have been able to visit the University of Melbourne without Tim's support and understanding. I am grateful to Tim for being accommodating and friendly and for introducing me to the lovely people at the Agent Lab in Melbourne and to explainable AI. I don't think I would have been able to navigate the administration processes without Dave, Tim or the PGR admin team (specially Sarah Brookes) and for that I am truly thankful.

I am grateful to my thesis group members Jeremy Wyatt, Mark Lee and Mohan Sridharan for their input, advice and reassurances. I am also indebted to the past and present residents of Room 144, members of the IR Lab and the CS department at large, for their friendship; especially Lenka, Bram, Abol, Nora, Akram, Harish, Iran (Masoumeh) and Brian. I am grateful to my non-CS friends too especially Komal, Sadaf, Tabinda, Beenish, Nida, Asad and Usman.

Of course I could not have done this without the support of my awesome family; especially Nano's love, Ami and Abu's prayers and stories of their scientific (mis)adventures, conversations with Aaniya, Apa, Raahim, Bhai and Ibrahim, and Hassan's invaluable support day in and day out. Lastly, I am grateful to the universe for bringing so many wonderful people (named or otherwise) and things to my PhD journey.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Robots are gradually beginning to enter complex human environments and are therefore expected to be able to deal with the dynamics and uncertainty of the real world. Multi-robot teams are deployed in warehouses and factories performing tasks such as package delivery, product assembly and quality assurance. They are deployed in search and rescue missions and even hospitals for medicine delivery. In fact the COVID pandemic that began in 2019 proved to be a catalyst for such deployments [WW21].

Though robots are able to repeat processes efficiently, they are generally unable to deal with anomalies or unexpected changes in the workspace. For this reason most robot workspaces are well defined with tools to track movement such as lines, QR codes or other markers. If we are to expect multiple robots to enter the real world we must make them robust to uncertainties or at least provide guarantees on their behaviour. An example of such a guarantee could be the probability of success for a particular task, for instance *what is the probability that the robot will deliver a package to a certain location within a deadline?*

The techniques presented in this thesis aim to extend current research by incorporating uncertainty and providing exact guarantees for robots operating in relatively unstructured environments.

1.1 Motivation

One emerging example of a multi-robot setting in the real world is that of self-driving cars on highways. Imagine a fleet of autonomous cars navigating a busy intersection where people make unexpected swerves or pedestrians suddenly decide to cross the road. Such a scenario raises a multitude of questions. For instance, can the autonomous cars work together to avoid any accidents? Are there any guarantees that the cars will not be the cause of an accident? Can the manufacturers say that the probability of a collision between any two self-driving cars is less than a certain threshold?

Similar questions could be asked for a team of robots operating in a warehouse environment. Perhaps one of the many robots in such a setting runs out of battery and is unable to perform the task assigned to it. Can other robots take over? Are warehouse operators aware that such a failure can happen? And if so how can planning algorithms incorporate such failures and keep operators informed?

It is clear that for successful deployment of robots it is imperative to have algorithms that can be used to provide guarantees over the actions of the robots. It is also important that these algorithms can be used to easily determine what went wrong and why. However, most robot planning algorithms do not aim to answer such questions due to the complexity of robot planning itself.

In order to apply robot planning algorithms to real world scenarios, each scenario environment and the robots have to be encoded within a certain framework. Then a planning approach that adheres to that framework can be applied. As a result, robot planning algorithms typically address a constrained set of scenarios in terms of an actual application. For example, some approaches to planning rely on graphs e.g. [Xu11]. Therefore, the robot and its interaction with the environment is modelled as a graph structure. Any tasks the robot must achieve need to be projected onto this structure. Another set of approaches to planning rely on continuous control models e.g. [MS03; SF97]. For these, robots are modelled using equations of motion governed by the robot's physical attributes. Tasks here need to be specified as end points that can be reached through these equations. In general, continuous control approaches to planning are used to create low level plans, i.e. the velocity required to move the robot forward or turn. Graph based approaches are then used to create high level plans, such as the series of steps needed to get a robot to pickup and deliver an item. This thesis focuses on the latter i.e. high level plans. Instead of projecting robot tasks onto the graph structure directly, formal languages such as Linear Temporal Logic (LTL) can be used to specify tasks for the robots. Such languages provide a precise specification for the task and are loosely independent of the models used e.g graphs. Furthermore, these languages can be used to formally verify specific properties of the models. Another advantage of these languages is that they are closer to spoken languages and therefore generally more intuitive, expressive and user friendly. Known techniques can then be used to combine these formal task specifications with the underlying graphs.

1.2 Challenges

In this thesis, we concentrate on multi-robot planning problems under uncertainty of action outcomes including critical robot failure. Our aim is to be able to provide formal guarantees on quantitative properties of our solution such as timeliness or reliability. Of particular interest to us is the application of service robots for intra-logistics, surveillance or stock monitoring. For such scenarios, it is desirable for a collection of *tasks* to be allocated to a team of robots e.g go to locations to pick-up and deliver or monitor objects of interest. The use of a team of robots allows for the tasks to be completed in a way that is robust against failures of individual robots, and allows the number of tasks requested to be dynamic as long as resources exist in the team to service them.

Most existing approaches for solving this class of problems divide the problem into separate *task allocation* (TA) and *planning* processes. TA determines which robot should complete which tasks, and planning determines how each task, or conjunction of tasks, should be completed. This separation is made to reduce the computational complexity of the problem. TA is a combinatorial problem which grows exponentially with the number of tasks and robots. The complexity of path planning is dependent on the models used to represent robots and the algorithms used to generate solutions. One popular modelling paradigm is that of Markov Decision Processes (MDPs). MDPs consist of a set of states (for example, discrete locations in a warehouse) and a set of actions that can be taken in those states. [LDK13] discusses the complexity of solving a particular class of MDPs which can theoretically be solved in polynomial time with respect to the number of states and actions. They show that such MDPs are P-complete. In practice, algorithms used to solve MDPs are hard to characterise with respect to their computational time and space complexity. As a result, solving single-robot path planning problems is considered hard in itself. Solving multi-robot problems increases the complexity of the problem by increasing the number of states and actions.

Another aspect of the real world, is uncertainty. MDPs are able to model uncertainty due to actions, for example, robots may not move as expected and end up in a location some distance away from the expected location. However, modelling uncertainty also comes at a cost. As the uncertainty increases, so does the number of possible states robots can end up in. This compounds the complexity of the problem, as more possible paths mean more potential solutions. Therefore, when it comes to dealing with uncertainties, whether they are due to the environment or the robot's hardware/software, there is still a long way to go. On the one hand, machine learning has provided robots with methods to extract patterns from their environments and build internal models using these patterns, but such algorithms depend on the kind of data/scenarios a robot experiences. On the other hand, models of the environment and a robot (such as MDPs) can provide more formal guarantees but as the number of variables in the model increases so does the complexity of achieving a solution.

As mentioned earlier, providing some quantitative information about the quality of multi-robot plans or guarantees on certain properties of such plans is critical to successful deployment of robot teams. Therefore, along with models of the robot and their interactions with the environment, we also need some formal way of specifying the tasks for these teams. Such a formal specification would reduce human error in specifying tasks, as well as understanding the capabilities of robots. Logics such as LTL can serve such a purpose. This is because they can be seen as being semantically somewhere in the middle of spoken language and mathematical models. For example a task such as *"Always go to the radiation room immediately after visiting the nuclear power plant"* can be expressed as "$\mathtt{G}(powerplant \Rightarrow \mathtt{X}\, room_{radiation})$" where $room_{radiation}$ is a label for the radiation room, *powerplant* is a label for the power plant, `G` translates to always and `X` translates to next. In fact, these languages can be directly translated to mathematical models and are widely used in formal verification. Loosely, formal verification is the process of checking whether a design satisfies some given requirements [Kuk96].

Another benefit of specifying tasks formally is that for complex tasks, the corresponding models can be used to automatically track task progress. A solution to a multi-robot planning problem with formally specified tasks combines robot models with the task specification models. This too can lead to an increase in the number of possible states since each state is now augmented with information about tasks. However, not only does it reduce human error in specifying tasks, it also provides a way to generate guarantees on the properties of the multi-robot plans. A similar approach is used in a branch of formal verification called model-checking, where models are exhaustively searched to see if they conform to specific properties.

In summary, the challenges associated with the problem of multi-robot task allocation and planning under uncertainty with formally specified tasks are those of computational complexity.

1. The complexity of task allocation increases exponentially in the number of robots and tasks.

2. The complexity of path planning, once tasks have been allocated, is heavily dependent on the underlying models. In practice, as the models grow, so does the complexity.

3. Formal task specification reduces error and allows for verification but adds to the complexity of solving path planning problems.

1.3 Outline

In this thesis, we address multi-robot task allocation and planning under uncertainty with a formally specified set of tasks i.e. the team's mission. We aim to generate plans for the multi-robot team that are robust to robot failure and provide an expectation of the number of tasks the team can successfully complete. We only consider uncertainty in action outcomes, e.g. a robot experiences a critical failure while executing an action or is unaware of the state of a door unless it performs the action to check it. Furthermore, with a focus on indoor mobile robots, we consider high-level tasks such as visiting specific locations etc.

We formalise the problem of task allocation and planning for a multi-robot team operating under uncertain conditions. We are able to model uncertainty including the possibility of robot failure using Markov decision processes (MDPs). In order to specify the team's mission precisely, we use Linear Temporal Logic (LTL). The LTL specification also allows for the generation of a task-based reward structure without any user input.

We propose three separate solutions for solving the aforementioned problem; a sampling-based heuristic search approach, an auctioning tasks then planning approach and a simultaneous task allocation and sequential planning approach. A key novelty in all three solution approaches is the use of LTL specifications to automatically generate reward or cost structures and satisfy as much of the mission as possible, even when it is not possible to fully acheive all tasks in the mission.

There is a large body of literature that applies sampling-based heuristic search approaches to planning problems. However, adapting them or the underlying models to solve for cases where there is no way to achieve the entire mission successfully is not trivial. Here, we leverage the use of LTL to generate a novel cost structure relative to the number of tasks in the mission. Through this cost structure and a novel combination of existing techniques we are able to bypass the limitations of these search methods when applied to problems similar to ours.

We also investigate a decoupled task allocation and planning approach to solve the problem. Task allocation determines which robot should complete which tasks, and planning determines how each task, or conjunction of tasks, should be completed. The separation of task allocation and planning is made to reduce the computational complexity of the problem. It allows each robot to plan separately for its own task set, avoiding the need for a joint planning model which is typically exponential in the number of team members. This separation also allows specialised algorithms to be used for the TA and planning parts, increasing the efficiency with which the task-directed behaviour of the team can be generated. Tasks are allocated through a centralised auctioneer. Robots plan for their assigned tasks independent of others. These plans are then combined to ensure that robots can share information about environment states such as doors being open. This allows robots to alter plans if needed. However, this also means that the TA process cannot be informed by the plans of the individual robots, which prevents it from exploiting opportunities, or avoiding hindrances, that are only evident once planning has been performed. For example, if the individual robots plan with time-based models, a task may be much quicker to complete at a particular time of day, but with TA separated from planning, this information cannot be exploited in the allocation process.

We deal with this by using the corresponding models of the LTL specifications to identify such situations and replan for them. Combined with the use of the automatic reward structure, this allows us to extend existing methods by satisfying as much of the mission as possible, even with individual robot failure.

Learning from the above and the sampling based approach, which does not separate task allocation and planning but does not fare well on large models, we propose a hybrid approach. This approach can be viewed as being in the middle of a fully coupled solution method (as in our search based solution) and a fully decoupled one (as in auctioning and planning). Recent work has considered the problem of *simultaneous task allocation and planning* (STAP) [SBD18b; SBD18d; Fat+18], which solves the complete problem in a single process, and can therefore take the plans of each robot (and their costs etc.) into account during the allocation process. STAP assumes that there is no uncertainty while modelling the robots. It also assumes that a single robot can perform a single task and that tasks do not depend on each other. We build on the STAP approach, formalising simultaneous task allocation and planning *under uncertainty* (STAPU) and adapting techniques from formal verification of probabilistic systems to solve the problem. The STAP problem is challenging due to the need to search for solutions (task allocations and associated single-robot plans) over the joint space of possible allocations and multi-robot action choices. The uncertain extension (STAPU) increases this challenge further by introducing uncertainty in the action outcomes of the individual robots.

This can lead to robots failing to achieve tasks they were expected to, for example due to uncertainty that affects the planning problem via action performance (e.g. whether a door is passable in a mobile robot's environment) and uncertainty that affects the robot directly (e.g. a robot experiences a critical failure). To address the uncertainty, we go beyond existing work by formally defining the team objective as *maximising the expected number of achieved tasks*; and proposing an approach for *planning for task reallocations* to increase robustness to failures.

To allow for a better interpretation of the expected team behaviour under uncertainty by a user, our solution to the STAPU problem leverages techniques from probabilistic verification. Specifically, we express individual tasks using the *co-safe* fragment of Linear Temporal Logic (LTL), and a *safe* LTL formula is used to specify safety constraints to be obeyed by all robots as they achieve these tasks. To model how the robots can achieve these specifications, individual robots' capabilities and environments are described using Markov decision processes (MDPs). Building upon techniques and tools for probabilistic model checking, we propose a method that generates multi-robot policies which maximise the expected number of tasks completed before the safety constraint is violated.

1.4 Thesis Organisation

In this section we present a brief summary of each chapter in the thesis.

Chapter 1: We introduce the problem of multi-robot planning under uncertainty using Linear Temporal Logic specifications.

Chapter 2: We introduce the basic formalisms used in this thesis which are also required to understand the related work.

Chapter 3: We survey various approaches to task allocation and planning with a focus on the use of formal specifications.

Chapter 4: We present a formal description of the problem of robust multi-robot task allocation and planning. We use MDPs to model the robots and their interactions with the workspace. We also use LTL to specify the team's mission, with a focus on the individual tasks. We describe the problem objective i.e. the quantitative property of the team plan under consideration. This is the expected number of tasks in the team mission that the multi-robot team can achieve. We then describe the setup used to implement and test all the work presented in this thesis. Finally, we present a naïve solution method, Value Iteration, which does not scale well.

Chapter 5: First, we introduce sampling-based heuristic search and a well-known framework for implementing such approaches. Next, we illustrate the problems associated with maximising the expected number of tasks for a multi-robot team using sampling-based search. Finally, we present and analyse our solution which is able to simultaneously allocate tasks and produce plans for the multi-robot team.

Chapter 6: We introduce a separated task allocation and planning approach to our problem. We begin with an introduction to Sequential Single Item Auctioning for task allocation. We then move on to an explanation of the auctioning and planning approach with task reallocation when robots fail. We end with a focus on the scalability of the proposed solution.

Chapter 7: We present our hybrid approach, an extension of [SBD18b] with the addition of uncertainty and task reallocation. We begin with a formalisation of the various components needed to solve the problem, specially a modified team model. We then show how we incorporate task reallocation and deal with global states. Finally, we compare the performance of this approach with that of sampling and auctioning.

Chapter 8: In our final chapter, we present a summary of our work and discuss its extensions.

1.4.1 Related Publications

Part of the work in Chapter 7 has been published as **Fatma Faruq**, Bruno Lacerda, Nick Hawes, and David Parker. "Simultaneous Task Allocation and Planning Under Uncertainty". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'18).* IEEE, 2018, pp. 3559–3564

We are in the process of submitting an extended version of [Fat+18] to the Robotics and Automation (RAS) journal.

Of relevance to partial mission satisfaction using LTL and MDPs is our work published as Bruno Lacerda, **Fatma Faruq**, David Parker, and Nick Hawes. "Probabilistic planning with formal performance guarantees for mobile service robots". In: *The International Journal of Robotics Research* 38.9 (2019), pp. 1098–1123

Chapter 2

Background

In this chapter, we present some preliminary material that forms the basis of our work. First, we describe the model used to represent robots and their interactions with the environment i.e. Markov decision processes (MDPs). We also introduce relevant concepts such as those of a policy and reward structure. Next, we look at the logic that we use to formally specify robot tasks i.e. Linear Temporal Logic (LTL). We then show how to combine our robot models with the task specification. Finally, we show how to use the combination of robot models and task specifications to generate optimal policies for Markov decision processes.

2.1 Markov Decision Processes (MDPs)

We use *Markov decision processes* (MDPs) to model the evolution of robots and their environment.

Definition 1 (MDP)**.** An *MDP* is a tuple of the form $\mathcal{M} = \langle S, \overline{s}, A, \delta, AP, L \rangle$, where:

- S is a finite set of states;
- $\overline{s} \in S$ is the initial state;
- A is a finite set of actions;

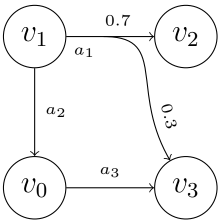


Figure 2.1: A Markov decision process (MDP) with 4 states. The labels of each state are v_0, v_1, v_2, v_3. The actions are a_1, a_2, a_3.

- $\delta : S \times A \times S \to [0, 1]$ is a probabilistic transition function, where $\forall s \in S,\ a \in A : \sum_{s' \in S} \delta(s, a, s') \in \{0, 1\}$;
- AP is a set of atomic propositions;
- $L : S \to 2^{AP}$ is a labelling function, such that $p \in L(s)$ if and only if p is true in $s \in S$.

In each state s of an MDP $\mathcal{M}$, there is a decision between the actions that are *enabled* in s, i.e., those in the set $A_s = \{a \in A \mid \delta(s, a, s') > 0 \text{ for some } s' \in S\}$. If action $a \in A_s$ is chosen in state s, then the probability that the next state is s' is given by $\delta(s, a, s')$. A sequence of such transitions $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ where $\delta(s_i, a_i, s_{i+1}) > 0$ for all $i \in \mathbb{N}$ is an (infinite) *path* through the MDP. A *finite path* $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n$ is a prefix of an infinite path. We denote the sets of all finite and infinite paths of $\mathcal{M}$ starting from state s by $FPath_{\mathcal{M},s}$ and $IPath_{\mathcal{M},s}$.

The choice of action to take at each step of the execution of an MDP $\mathcal{M}$ is made by a *policy*, which can base its decision on the history of $\mathcal{M}$ up to the current state.

Definition 2 (Policy)**.** A *policy* for MDP $\mathcal{M}$ is a function $\pi : FPath_{\mathcal{M},\bar{s}} \to A$ such that, for any finite path ρ ending in state s_n, we have $\pi(\rho) \in A_{s_n}$.

In this work, we will use *memoryless* policies $\pi : S \to A$, which only base their choice of action on the current state, and *finite-memory* policies, which track a finite set of "modes" needed, in conjunction with the current state, to choose an action. For a particular policy π, we can define a probability space $Pr^{\pi}_{\mathcal{M},s}$ over the set of infinite paths $IPath_{\mathcal{M},s}$.

Furthermore, for a measurable function $X : IPath_{\mathcal{M},s} \to \mathbb{R}$, we write $E^{\pi}_{\mathcal{M},s}(X)$ for the expected value of X with respect to $Pr^{\pi}_{\mathcal{M},s}$.

Finally, we define MDP *reward structures.* We use a variant that assigns non-negative values to state-action-state triples.

Definition 3 (Reward structure)**.** A *reward structure* for an MDP $\mathcal{M}$ is a function $r : S \times A \times S \to \mathbb{R}_{\geq 0}$.

Of particular interest in this thesis is the expected amount of reward accumulated up until a target is reached.

Definition 4 (Expected cumulative reward)**.** For reward structure r on MDP $\mathcal{M}$ and target label $b \in AP$, we define the function $cumul^b_r : IPath_{\mathcal{M},s} \to \mathbb{R}_{\geq 0}$ as:

$$cumul^b_r(s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots) = \sum\nolimits_{i=0}^{n_b - 1} r(s_i, a_i, s_{i+1}) \tag{2.1}$$

where n_b is the first index for which $b \in L(s_{n_b})$. For the cases where $b \notin L(s_i) \forall i$, we define $n_b = \infty$. The *expected cumulative reward* under policy π of $\mathcal{M}$ is defined as $E^{\pi}_{\mathcal{M},s}(cumul^b_r)$.

The expected cumulative reward under a policy is also referred to as the value function of the policy. More specifically, we use the value function of the policy to map states to a non-negative value. This value is the expected cumulative reward of following the policy beginning with state s.

Definition 5 (Value function of a policy)**.** For reward structure r on MDP $\mathcal{M}$ and target label $b \in AP$ with $cumul^b_r$, the value function $V^{\pi}_r : S \to \mathbb{R}_{\geq 0}$.

$$V^{\pi}_r(s) = E^{\pi}_{\mathcal{M},s}(cumul^b_r) \tag{2.2}$$

Given a policy, Definition 5 can be used to evaluate a policy, i.e perform *policy evaluation.* Of particular interest to us are *indefinite horizon* MDPs where the path length to a target state is finite but not known beforehand as we will see later in section 4.2.

2.2 Linear Temporal Logic (LTL)

Linear temporal logic (LTL) [Pnu81] is an extension of propositional logic which allows reasoning about infinite sequences of states. The syntax of LTL is as follows.

Definition 6 (LTL syntax)**.** LTL formulas φ over atomic propositions AP are defined using the following grammar:

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathtt{X}\,\varphi \mid \varphi\,\mathtt{U}\,\varphi, \text{ where } p \in AP. \tag{2.3}$$

The $\mathtt{X}$ operator is read "next", meaning that the formula it precedes will be true in the next state. The $\mathtt{U}$ operator is read "until", meaning that its second argument will eventually become true in some state, and the first argument will be continuously true until this point. The other propositional connectives can be derived from the ones above in the usual way. Moreover, other useful LTL operators can be derived from the ones above. Of particular interest for our work are the "eventually" operator $\mathtt{F}\,\varphi$, which requires that φ is satisfied in some future state, and the "always" operator $\mathtt{G}\,\varphi$, which requires φ to be satisfied in all future states: $\mathtt{F}\,\varphi \equiv \mathit{true}\,\mathtt{U}\,\varphi$ and $\mathtt{G}\,\varphi \equiv \neg\,\mathtt{F}\,\neg\varphi$. Given an infinite path σ, we write $\sigma \models \varphi$ to denote that σ satisfies formula φ. Informally, a path is a series of atomic propositions. A path satisfies an LTL formula when the atomic propositions in the path are seen in the order dictated by the formula.

The semantics of full LTL is defined over infinite paths. However, in this work, we are interested in specifying behaviours that occur within finite time. So, we use two well-known subsets of LTL for which properties are meaningful when evaluated over finite paths: *safe* and *co-safe* LTL [KV01]. These are based on the notions of *bad prefix* and *good prefix*. A bad prefix for φ is a finite path that cannot be extended in such a way that φ is satisfied, and a good prefix for φ is a finite path that cannot be extended in such a way that φ is *not* satisfied. To formally define a good prefix we must first define

ω-language. The ω-language of all infinite paths that satisfy φ is defined as:

$$\mathcal{L}(\varphi) = \{\sigma \in (2^{\mathrm{AP}})^\omega \mid \sigma \vDash \varphi\} \tag{2.4}$$

Definition 7 (Good Prefix)**.** Let φ be an LTL formula, $\sigma = \sigma_0\sigma_1 \ldots \in (2^{\mathrm{AP}})^\omega$ such that $\sigma \vDash \varphi$. σ has a *good prefix* for φ if there exists $n \in \mathbb{N}$ for which the truncated finite sequence $\sigma \mid_n = \sigma_0\sigma_1 \ldots \sigma_n$ is such that for every $\sigma' \in (2^{\mathrm{AP}})^\omega$ the concatenation $\sigma \mid_n .\sigma' \vDash \varphi$

Definition 8 (Bad Prefix)**.** Let φ be an LTL formula, $\sigma = \sigma_0\sigma_1 \ldots \in (2^{\mathrm{AP}})^\omega$ such that $\sigma \vDash \varphi$. σ has a *bad prefix* for φ if there exists $n \in \mathbb{N}$ for which the truncated finite sequence $\sigma \mid_n = \sigma_0\sigma_1 \ldots \sigma_n$ is such that for every $\sigma' \in (2^{\mathrm{AP}})^\omega$ the concatenation $\sigma \mid_n .\sigma' \nvDash \varphi$

Safe LTL is defined as the set of LTL formulas for which all non-satisfying infinite paths have a finite bad prefix. Conversely, co-safe LTL is the set of LTL formulas for which all satisfying infinite paths have a finite good prefix.

Definition 9 (Co-safe LTL)**.** Let φ be an LTL formula. We say that φ is a co-safe LTL formula if for all $\sigma \in \mathcal{L}(\varphi)$, σ has a good prefix for φ.

Definition 10 (Safe LTL)**.** Let φ be an LTL formula. We say that φ is a safe LTL formula if for all $\sigma \in \mathcal{L}(\varphi)$, σ has a bad prefix for φ.

If φ is a formula of safe LTL, then its negation $\neg\varphi$ is co-safe.

For simplicity, we assume a *syntactic restriction* for safe and co-safe LTL. We assume that all formulas are in positive normal form (negation can only appear next to atomic propositions). Syntactically safe LTL is the set of formulas for which only the `G` and `X` temporal operators occur, and syntactically co-safe LTL is the set of formulas for which only the `X`, `F` and `U` temporal operators occur.

For any co-safe LTL formula φ written over AP, we can build a deterministic finite automaton (DFA) equivalent to φ.

Definition 11 (DFA Representation)**.** Let φ be a co-safe LTL formula. The DFA representing φ is a tuple $\mathcal{A}_\varphi = \langle Q, \overline{q}, Q_F, 2^{AP}, \delta_\varphi \rangle$, where:

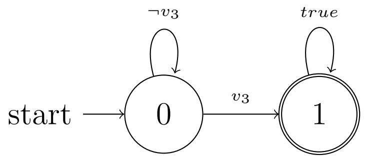


Figure 2.2: The deterministic finite automaton for the formula $\mathtt{F}\, v_3$. There are 2 states with 0 being the initial state and 1 being the only accepting state. v_3 is a transition label (alphabet).

- Q is a finite set of states;
- $\overline{q} \in Q$ is the initial state;
- $Q_F \subseteq Q$ is the set of accepting states;
- 2^{AP} is the alphabet; and
- $\delta_\varphi : Q \times 2^{AP} \to Q$ is a transition function;

such that the language of finite words accepted by $\mathcal{A}_\varphi$ is the set of good prefixes of paths that satisfy φ (or, more precisely, the sequences of state labellings from those paths) [KV01].

Conversely, if φ is a formula in *safe* LTL, then the DFA $\mathcal{A}_{\neg\varphi}$ for its negation represents the *bad prefixes* of φ.

2.3 LTL Specifications for MDPs

Given an MDP $\mathcal{M}$ and an LTL formula φ over the set of atomic propositions AP used to label the MDP, we write $Pr^{\pi}_{\mathcal{M},s}(\varphi)$ for the probability of a path satisfying φ from state s in MDP $\mathcal{M}$ under a policy π.

Definition 12 (Probability of satisfying an LTL formula from a given state)**.** If φ is an LTL formula, s is a state in the MDP $\mathcal{M}$, and π is a policy for the MDP, the probability of satisfying φ under π from s is:

$$Pr^{\pi}_{\mathcal{M},s}(\varphi) = Pr^{\pi}_{\mathcal{M},s}(\{\sigma \in IPath_{\mathcal{M},s} \mid \sigma \models \varphi\})$$

Furthermore, we write $Pr^{\max}_{\mathcal{M},s}(\varphi)$ to denote the maximum probability (over all policies) of satisfying φ from state s.

Another useful property is the expected amount of reward accumulated until a co-safe LTL formula φ is satisfied.

Definition 13 (Expected accumulated reward)**.** For reward structure r, we define the expected amount of reward accumulated using the function:

$$cumul^{\varphi}_{r}(s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots) = \sum_{i=0}^{n_\varphi - 1} r(s_i, a_i, s_{i+1})$$

where n_φ is the first index for which $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_{n_\varphi}$ is a good prefix for φ.

The expected amount of reward r accumulated before φ is satisfied under policy π on $\mathcal{M}$ is then defined as $E^{\pi}_{\mathcal{M},s}(cumul^{\varphi}_{r})$, and we write $E^{\max}_{\mathcal{M},s}(cumul^{\varphi}_{r})$ to denote the maximum expected value over all policies of $\mathcal{M}$.

For a co-safe LTL formula φ, we can compute both the maximum probability $Pr^{\max}_{\mathcal{M},s}(\varphi)$ and the maximum expected reward $E^{\max}_{\mathcal{M},s}(cumul^{\varphi}_{r})$, by building and solving a *product MDP*, which combines the MDP $\mathcal{M}$ with a DFA $\mathcal{A}_\varphi$ for φ.

Definition 14 (Product MDP)**.** If $\mathcal{M} = \langle S, \overline{s}, A, \delta, AP, L\rangle$ is an MDP and φ is a co-safe LTL formula over AP represented by DFA $\mathcal{A}_\varphi = \langle Q, \overline{q}, Q_F, 2^{AP}, \delta_\varphi\rangle$, the *product MDP* is the MDP $\mathcal{M} \otimes \mathcal{A}_\varphi = \langle S_\otimes, \overline{s}_\otimes, A, \delta_\otimes, AP, L_\otimes\rangle$ where:

- $S_\otimes = S \times Q$
- $\overline{s}_\otimes = (\overline{s}, \delta_\varphi(\overline{q}, L(\overline{s})))$
- $\delta_\otimes((s,q), a, (s',q')) = \begin{cases} \delta(s,a,s') & \text{if } q'{=}\delta_\varphi(q, L(s')) \\ 0 & \text{otherwise} \end{cases}$
- $L_\otimes(s,q) = \begin{cases} L(s) \cup \{acc_\varphi\} & \text{if } q \in Q_F \\ L(s) & \text{otherwise} \end{cases}$

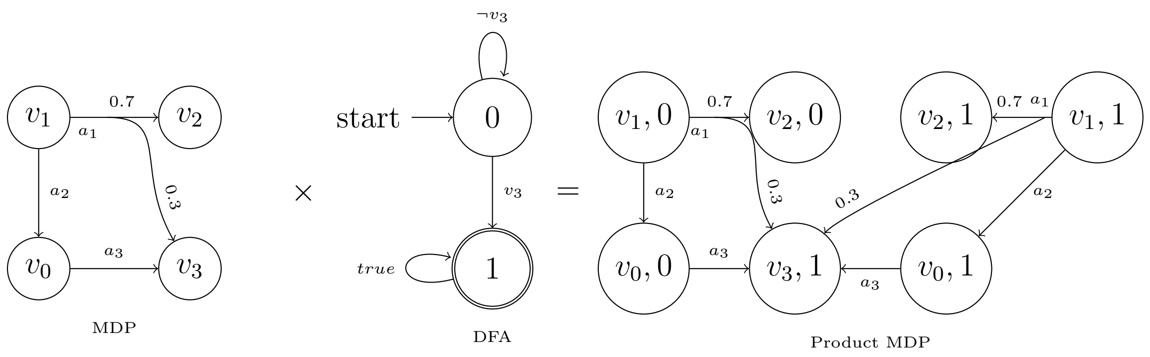


Figure 2.3: The construction of the product MDP for the MDP from Figure 2.1 and LTL formula $\mathtt{F}\, v_3$ i.e. the DFA from Figure 2.2. All MDP and DFA states and transitions are preserved.

The construction of the product MDP $\mathcal{M} \otimes \mathcal{A}_\varphi$ is well known (see, e.g., [BK08]). The product behaves like the original MDP $\mathcal{M}$ (it preserves the probabilities of paths from $\mathcal{M}$) but is augmented with information about the satisfaction of φ. Once a path of $\mathcal{M} \otimes \mathcal{A}_\varphi$ reaches an *accepting state* (i.e., a state of the form (s, q_F) for $q_F \in Q_F$), it is a good prefix for φ and we know that φ is satisfied. This reduces the problem of computing $Pr^{\max}_{\mathcal{M}}(\varphi)$ to finding the maximum probability of reaching an accepting state in the product. Since we label such states with a new atomic proposition acc_φ, we have:

Proposition 1 (Maximum probability of satisfaction in the product MDP)**.**

$$Pr^{\max}_{\mathcal{M}}(\varphi) = Pr^{\max}_{\mathcal{M}\otimes\mathcal{A}_\varphi}(\mathtt{F}\, acc)$$

Furthermore, a (memoryless) optimal policy for reaching acc_φ in $\mathcal{M} \otimes \mathcal{A}_\varphi$ can be converted to an optimal policy for φ in $\mathcal{M}$. The latter is a finite-memory policy whose modes are the DFA states Q. Optimal values and policies can be found using standard techniques over the product such as value iteration [Put94].

In a similar fashion, we can use the product MDP to compute the maximum expected cumulative reward until φ is satisfied, and a corresponding optimal policy:

Proposition 2 (Optimal policy in the product MDP)**.**

$$E^{\max}_{\mathcal{M}}(cumul^{\varphi}_{r}) = E^{\max}_{\mathcal{M}\otimes\mathcal{A}_\varphi}(cumul^{acc_\varphi}_{r_\varphi})$$

where r is a reward structure for $\mathcal{M}$ and r_φ is the corresponding reward structure for $\mathcal{M} \otimes \mathcal{A}_\varphi$, with reward values copied directly to their corresponding transitions in the product.

Lastly, we can extend the value function of a state under a policy such that it uses the product MDP:

Proposition 3 (Value function of a state in the product MDP)**.**

$$V_{\otimes_r}^{\pi}(s_\otimes) = E^{\pi}_{\mathcal{M},s_\otimes}(cumul_r^{\varphi})$$

We will often use superscript notation when referring to product models, e.g., writing $\mathcal{M}^\varphi$ for $\mathcal{M} \otimes \mathcal{A}_\varphi$. For a list $\Phi = \langle \varphi_1, \ldots, \varphi_m \rangle$ of co-safe LTL formulas, we can apply Definition 14 repeatedly to build the product $\mathcal{M} \otimes \mathcal{A}_{\varphi_1} \otimes \cdots \otimes \mathcal{A}_{\varphi_m}$ and we will sometimes use the shorthand $\mathcal{M}^\Phi$ for this. Definition 14 is utilised throughout the thesis and forms the basis for all the solutions presented.

Chapter 3

Related Work

Motion planning for mobile robots has been studied extensively in robotics [GKM10; CS19; Mac+16]. The problem is generally divided into two parts: one dealt with by a *global planner* and the other by a *local planner* [Cai+20]. The local planner generates low level commands to control the speed and direction of the robot and avoid obstacles using algorithms such as the dynamic window approach [FBT97]. The global planner works on an abstraction of the robot's environment such as a topological map [KB91]. It generates a series of high level actions that are required to get the robot from one place to another.

Example 1. Figure 3.1 shows a *topological map*. The space is divided into regions and each region is shown as a node on the map. The edges connecting nodes show that it is possible to travel between these nodes. Assume that a plan for *navigate to* v_6 *from* v_1 needs to be generated. The global planner uses the topological map and generates the plan: $v_1 \;{}^{v_1}\overset{to}{\rightarrow}{}^{v_5}\; v_5 \;{}^{v_5}\overset{to}{\rightarrow}{}^{v_7}\; v_7 \;{}^{v_7}\overset{to}{\rightarrow}{}^{v_6}\; v_6$. The local planner generates the velocity commands for the robot's driving system, needed to get it from one location to another, e.g. v_1 *to* v_5.

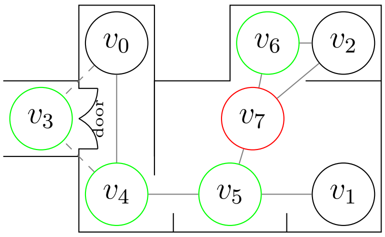


Figure 3.1: A topological map

We focus on approaches for generating global plans with the assumption that a local planner is available for use. The solution approaches to generating a global plan range from search based algorithms such as A* [HNR68] and its variants [Duc+14; Wag18] to sampling based methods such as Rapidly-exploring Random Trees (RRTs) [Lav98] and continuous control based approaches such as potential fields [AI20]. The use of abstraction in global planners reduces the complexity of the problem and the required computational resources. It also makes the solution method easier to understand. For example, it is easier for a human to debug the planning algorithm when the plan is presented as a series of named locations than as a series of velocity commands.

One disadvantage of using abstractions is the loss of accuracy on the guarantees of properties of the resulting plans. In order for these plans to be relied upon in real world scenarios such as warehouses, exact values on quantitative properties of the plans are necessary. These values can be used to inform the end user of the success of the plan or of any errors in the underlying models. However these guarantees are only as accurate as the underlying abstractions. Therefore research has now focused on ways to ensure the correctness of planning algorithms using techniques from the formal verification community [Luc+19].

To this end, formal languages such as Linear Temporal Logic (LTL) have been used to capture complex motion tasks comprising of a sequence of operations or repeating operations e.g. [TD16; Din+14; Smi+11; Leo+17a; Haw+17]. Unlike specifying tasks as nodes on a graph or robot positions, these languages provide a more expressive and intuitive framework for specifying tasks. The use of formal languages in specifying tasks reduces the chance of error and provides mechanisms to verify properties of plans. In fact the use of formal languages such as LTL, combined with models such as MDPs allows for the generation of **correct-by-construction** [MS00] plans e.g. [Ulu+13; TD16; Leo+17b; SBD18b]. As mentioned earlier, the *correctness* of the resulting plans relies heavily on the correctness of the robot models. Algorithms that generate *correct-by-construction* plans for robots borrow heavily from formal verification, specifically **model checking** which *checks* a (system's) model against a given requirement (specification).

In this chapter we survey recent work in robot planning under uncertainty with guarantees on plan properties. Specifically, our focus is to review solutions to task allocation and planning for indoor mobile robots with some degree of uncertainty in the outcome of their actions. Furthermore, we want to be able to generate exact values for quantitative robot team plan properties and in doing so verify the robot team plans. We begin this chapter with a discussion of structures used to model robots focusing on incorporating uncertainty. Next we look at ways to specify tasks for these models, especially those using Linear Temporal Logic. This is followed by a summary of the objectives of task allocation and planning problems. Finally, we describe the various solution methods used to solve these problems. Throughout the chapter, works from the verification community are interleaved with works from the robotics community, with the focus of each section being the myriad of techniques or paradigms used.

3.1 Robot models

There is a variety of ways to model robots and their interaction with the environment. The choice of model depends on the type of problem being solved. The most general of these models is that of a graph, where vertices or nodes denote locations and edges the ability to travel between them. In robot planning, graphs are commonly used in Multi-Agent Path Finding (MAPF) scenarios [MKK17; Foe+17] to find collision free paths for a team of robots. In fact as we will see later, most solution methods for planning adapt graph based algorithms.

Transition systems are another common paradigm used to model robots. Minimally, a transition system consists of a set of states (locations) and a relation of state transitions. Informally this relation tells us whether it is possible to go from one particular state to another. In a deterministic transition system (DTS) this relation is binary, i.e either it is possible to go from one state to another or not. Therefore, robots modelled as deterministic transition systems do not consider uncertainty [Smi+11; VLB20; Lah+15]. Adding weights to the relations in a DTS allows for the modelling of costs, such as the distance between two states or the time taken to travel from one state to another [Smi+11; VLB20].

Unlike DTSs, as we saw in Chapter 2, transitions in MDPs can be used to model uncertainty. MDPs are used to model agents interacting with the environment in a wide range of systems [Whi85], including finance [BR11], operations research [FS12], security and communication protocols [Alt02; Bas+20] and biology [MK20]. MDPs and their extensions such as Partially-Observable MDPs (POMDPs) can be used to model many types of uncertainty, for example uncertain action durations [LS18; Str+20; Hah+17], uncertain action outcomes [Din+14; LPH14; Ash+18; Brá+14; Tre+16; KW12; LPH15a; Lac+19] and uncertain observations [SS04; Ama+13; Omi+17].

Another approach to modelling a robot's interaction with its environment is Petri Nets (PNs) [CL07]. Each state in the environment can be modelled as a *place* in the PN. Robot actions can be modelled as transitions that lead the robot from one state to another. The presence of a robot in a state is denoted by a token in that state. Consequently, PNs provide an intuitive way to monitor task execution in real time. They can also incorporate uncertainty by adding probabilities to the transitions associated with states. However, modelling complex robot actions using PNs is time consuming and not intuitive. This is because complex robot actions need more states and transitions to model. Therefore, the more complex the robot's functionality is the more complex the PN becomes [SS17].

Modelling uncertainty adds to the complexity of the planning problem since there are more potential solutions to search from. Therefore, the choice of a modelling paradigm is dependent on the application domain. Some problem formulations assume that a low level planner may be able to handle uncertainty in motion such as dynamic obstacles e.g. [GMS17] and therefore can use deterministic transition systems or graphs. However, some model uncertainty explicitly using MDPs and their extensions e.g. [Hah+17; Ama+13; Din+14; LPH14; Ash+18]. This uncertainty can be due to obstacles [Hah+17] or robot behaviour such as the possibility of failure [WHL17] or a delay in robot motion [Li+19].

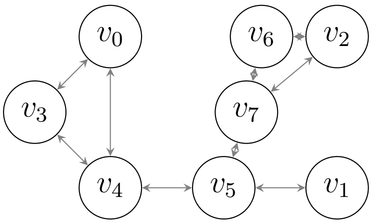


Figure 3.2: A bidirectional graph representing a robot's possible actions in the environment from Figure 3.1. Since the door is always open, it does not need to be modelled.

MDP extensions such as POMDPs add to the complexity of the problem, which may be unnecessary depending on the application. For example, if robots are moving in a warehouse where QR codes can be used to inform them of their location then there is no need for modelling partially observable states.

3.1.1 Multi-robot models

The choice of modelling structures for multi-robot problems is more difficult than that for single-robot problems. This is because multi-robot problems can quickly become intractable. One way to model a team of robots is to generate a cartesian product of all single robot models, for example the cartesian product of MDPs is a Multi-agent MDP (MMDP [Bou96]). Such an approach increases the size of the team model exponentially with the number of robots.

Example 2. Figure 3.2 shows a bidirectional graph modelling a single robot traversing the topological map from Figure 3.1. For simplicity we assume that the door is always open and therefore does not need to be modelled. For one robot the graph will have 8 nodes, one for each location. For a two robot team this will result in a graph of $8^2 = 64$ nodes. As the number of robots increases, the number of nodes in the graph will increase exponentially. The exponential increase also applies to edges in graphs.

This exponential increase affects the solution to the multi-robot problem both in terms of memory space and computation time. Effects on the memory space usage are directly related to the size of the model whereas effects on the computation time usage are not. As we shall see later, the design of the solution method can greatly reduce the computation time. [KZ20; KZ18; Sch+16; SLB20] all operate on the full product team model, generating the relevant parts of the model as needed.

In case the team consists only of homogeneous robots, the product team model can be greatly simplified. For such teams Petri Net (PN) models are a good choice as they do not scale exponentially with the number of robots [KM20; LL19; Man+19]. This is because PNs model states in the environment as *places* and actions from these states as *transitions*. Each robot is simply a token in a *place* or state. As a result, a homogenous robot team does not need extra places or transitions, simply extra tokens. Therefore, PN team models depend more on the number of locations and actions than the number of robots. Modelling robots as tokens also lends itself easily to placing constraints on the number of robots travelling along an edge. When it comes to heterogeneous robot teams, PNs also suffer from an exponential increase. This is because robots will have different actions which will increase the number of transitions. Similarly, the tokens used to represent these robots are no longer interchangeable and that too increases the complexity of the model. Therefore, most multi-robot planning approaches avoid building a joint team model altogether, as in [WC11; MKK17].

One type of modelling we have not discussed so far is where robot behaviours are encoded as a set of constraints [Sah+14; Leo+17b; GMS17]. That is because these encodings themselves rely on some kind of robot model, which may be implicitly described by a set of equations [Sah+14].

There are some works that try to build a team model that is different from the full product team model. For example in [Ulu+13], a team transition system, T is constructed using recursive depth first search. T is not the same as the cartesian product of all robot transition systems, as it considers the robot with the least transition time to the next vertex and then creates an intermediate state called *travelling state* for all other agents that have not reached their next vertex at this time. This pre-processing allows the creation of a more compact team model that can be used to generate a solution. [TD16] also avoids building the full team model by creating multiple small joint models. This is done by grouping robots whose tasks are dependent on each other and creating a joint team model for robots in a particular group only. Another approach to avoiding building the full team model is presented in [SBD18b] which builds a sequential team model. Each robot's model is linked to the other robot's using special transitions originating from states where tasks are completed. [Ulu+13; TD16; SBD18b] all use deterministic transition systems to model the robots.

Adding uncertainty to these team models by using MDPs to model the robots would affect scalability but reduces the assumptions made. For instance, MDPs make it easy to model uncertain action outcomes e.g. a check door action might result in a door being open or closed. Therefore, we choose MDPs to model the robots in our problem formulation (see Chapter 4). Since our focus is on indoor robots where it is possible to determine the robot's location using QR codes etc, we do not consider POMDPs which have the added complexity of partially observable states. Furthermore, MDPs are widely used in planning for mobile robots (e.g [Tei12; Tre+16]), generating correct-by-construction plans (e.g. [Din+14]) and generating plans using model checking techniques (e.g [LPH14]).

3.2 Task Specification

Specifying tasks for robot models can be done explicitly by defining a set of goal states [BBS95]. When using a set of goal states the complexity of the task is dependent on the variables in the robot model. For example, assume an MDP robot model. If the MDP has only one state variable, say the location, then it would be easy to specify getting to a certain location as a task. However, it would not be trivial to specify visiting a set of locations. It might require a change in the model, for example the state of the MDP may need to be modified to include more variables.

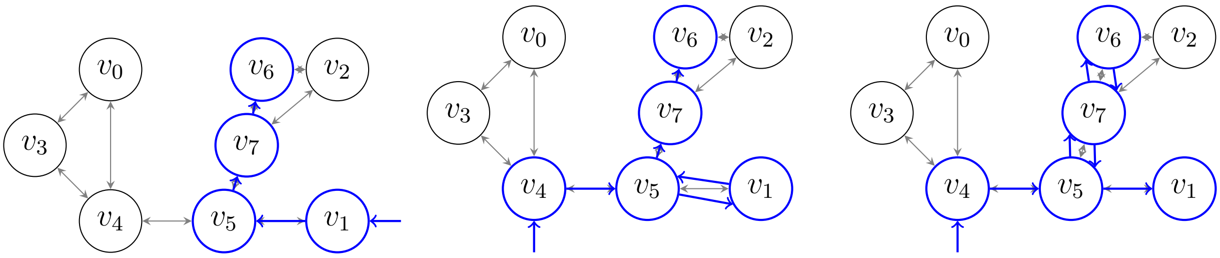
One way to specify complex tasks is through the use of a reward function (see Definition 3) with the objective that the solution maximises the expected cumulative reward (for instance, see Definition 4 for MDPs). In many planning problem formulations, the objective is to minimise the cost function, a dual of the reward function. The cost function can be used to model a variety of properties e.g. the time taken to travel from one state to another, the distance between two states or user preference for a particular action. Reward or cost functions do not have to be strictly positive. Models where tasks are implicitly defined using such functions also need some way to avoid collecting infinite reward or cost. This can be done by limiting the number of steps that the robot can take in terms of states and actions [TD16], adding a discount factor that reduces the reward over time [Boz+20] or combining the reward function with explicitly defined goal states [MM19]. In such a formulation, no more cost or reward is accumulated after the goal state is reached. Limiting the number of steps or adding a discount factor is not a viable option when quantifying properties of the plan such as timeliness.

It is important to note that designing the reward or cost function is a task in itself [GSB17]. For example if the task is to get to a particular state while avoiding another state, the positive reward for getting to the goal state must be balanced by negative reward for ending up in the state to avoid.

3.2.1 LTL Task Specifications

Another way to specify complex tasks for an MDP is through the use of logics such as Linear Temporal Logic (LTL). As we saw in Section 2.3, LTL formulae can be represented as automata which can be combined with the MDP. This is a technique that is widely used by the verification community for verifying probabilistic systems [KP13], particularly for *model-checking* i.e. checking whether a modelled system meets a given specification. The accepting state of the automaton becomes the goal state for the MDP. In fact, in Chapter 4 we use the LTL automata to generate a reward function without any user input. LTL is expressive enough to describe complex tasks and also closer to spoken language than reward functions or MDP states.

Example 3. Continuing with the map from Example 1, assume that the robot is in location v_1. The task, *navigate to* v_6 *from* v_1 can be written as $\mathtt{F}\, v_6$, which is read as



(a) Path for a robot starting in v_1 for $\mathtt{F}\, v_6$

(b) Path for a robot starting in v_4 for $\mathtt{F}(v_1 \wedge \mathtt{F}\, v_6)$

(c) Possible path for a robot starting in v_4 for $\mathtt{F}\, v_1 \wedge \mathtt{F}\, v_6$

Figure 3.3: LTL specifications and possible paths which satisfy them. The underlying graph is from Figure 3.2. The bold states and edges show the path. The initial state is the one with a lone arrow on the outside.

eventually v_6, meaning at some point the robot should get to v_6. If the robot was not in v_1, the task *navigate to* v_6 *from* v_1 could be expressed as $\mathtt{F}(v_1 \wedge \mathtt{F}\, v_6)$. It is read as *eventually* v_1 *and* ***then*** *eventually* v_6. Note the brackets which mean that the robot should eventually get to v_1 and then eventually get to v_6. This is different from *eventually* v_1 *and eventually* v_6 i.e. $\mathtt{F}\, v_1 \wedge \mathtt{F}\, v_6$ which implies that the robot needs to get to v_1 and v_6, irrespective of the ordering. Figure 3.3 illustrates these differences by showing a possible path for each formula.

Recall from Definitions 9 and 10 that an LTL formula can syntactically belong to *co-safe* LTL or *safe* LTL. Safe LTL is generally used to specify tasks that should be repeated infinitely often such as repetitive behaviour [Smi+11; Din+14; GZ18] or safety tasks [Lah+16] such as always avoiding certain locations, actions etc. Repeating tasks are common in information gathering and monitoring scenarios. Co-safe LTL can be used to specify tasks that can be completed in finite time such as *get the coffee from the kitchen* [LPH14; Lah+15]. As mentioned in Section 2.2 the negation of a safe LTL formula is a co-safe LTL formula. This property can be used to avoid the complications associated with infinite paths that arise due to safe LTL formulae [Lah+16].

LTL formulae can also be restricted by disallowing the use of certain operators when specifying a task or by imposing constraints on the form of the formula. For example, in [VLB20] LTL_x i.e. LTL without the next operator is used. [SC20; DCB17; Smi+11; Ulu+13; SBD18b] place restrictions on the form of the task specification. For instance [SBD18b] require that all tasks be specified in disjunctive normal form (DNF). Others require that the task specification formula conforms to a particular form which is used to ensure a certain kind of behaviour. For example [DCB17; Smi+11; Ulu+13] expect tasks to be of the form $\varphi \wedge \mathtt{G}\,\mathtt{F}\,\pi$, which ensures that there is a task π that needs to be repeated continuously.

Another advantage of using LTL to specify tasks is the use of the corresponding automata to guide the algorithm towards solutions. This can be done by using reward functions based on automata states [SLB20; Lac+19]. Furthermore, the LTL automata can also be used to track task progress [SBD18a; Lac+19]. In fact [SBD18a] generates sequences of actions called *options* using the notion of task progress.

Not only is LTL expressive, its corresponding automata can enrich the solution method while providing a formal task specification. Therefore, LTL is a good choice for task specification for an algorithm that aims to generate *correct-by-construction* plans. For these reasons, this thesis uses LTL to specify the set of tasks for a multi-robot planning problem, which allows us to borrow techniques from model-checking and be able to verify our solutions by giving quantitative guarantees on the plans. Inspired from real world applications, we use safe LTL to specify safety tasks and co-safe LTL to specify a set of tasks that must be satisfied in finite time.

Remark 1 (On the use of LTL). The discussion above begs the question, why LTL and not some other temporal logic language such as Signal Temporal Logic (STL) [MN04], Metric Temporal Logic (MTL) [Koy90], Probablistic Computational Tree Logic (PCTL) [HJ89]. Both STL and MTL deal with real-time constraints i.e. explicit timing is needed. While this is useful for low-level systems, it is not needed for high-level systems specially in the context of our problem where we can model action durations as costs. PCTL has limited expressivity since it requires single temporal operators and is therefore not useful in expressing high-level missions for robot teams. Since our focus is on non-adversarial robot teams, we do not look to mutli-agent logics such as Dynamic Epistemic Logic (DEL), Coalition logic and BDI logics. For example, in [DB20] DEL is used to enable a robot to reason about other agents' (humans or robots) beliefs (internal states). Such a logic would add to the complexity of our problem and is not needed in the case of non-adversarial teams with independent tasks. We refer the reader to [HW12] for a review of multi-agent logics.

3.2.2 Task Categories

In the previous sections we have surveyed paradigms used to model robots and specify tasks for them. Our focus has been on being able to capture uncertainty without overly complicating the robot model as is suitable for indoor mobile robotics. For task specifications, we have looked at languages that are very loosely coupled to the model, expressive and can aid in model-checking. Since our problem is that of both task allocation and planning, we will now look at the categories robot tasks and multi-robot tasks are divided into. These categories inform the choice of solution.

[Zlo06] looks at using market-based (or auction-based) methods to distribute tasks among a team of robots. It divides tasks into different categories for task allocation to a team. Though the categorisation is for a team of robots, some of it still applies to single-robot planning problems, particularly *atomic tasks* and *simple tasks*. For example, $\mathtt{F}\, v_6$ is an *atomic task* since it can not be decomposed further [Zlo06]. On the other hand, $\mathtt{F}\, v_6 \wedge \mathtt{F}\, v_1$ is a *simple task* since it can be decomposed into the *atomic tasks* $\mathtt{F}\, v_6$ and $\mathtt{F}\, v_1$. Atomic tasks can not be split into further sub-tasks.

Tasks can also be categorised in terms of the robots that must perform them. Atomic tasks generally require only one robot to perform them. [GM04] uses the term *single-robot tasks* to describe such tasks. As in [Zlo06], [GM04] categorises tasks from a multi-robot perspective but the focus is on a complete taxonomy for multi-robot planning. Consequently, [GM04] also categorises task assignment as instantaneous or time-extended. *Instantaneous assignment* refers to situations where tasks are provided instantaneously and therefore must be assigned as they come (sometimes during robot operation). Both [LPH14; VLB20] provide plans for a single robot with instantaneous and *time-extended* assignments.

They modify the plan the robot is currently executing to incorporate the incoming task. In [VLB20] dynamic obstacles are detected and avoiding these is added as an instantaneous task. [Lah+16] also avoids obstacles in a partially known environment. However, instead of adding *avoid detected obstacle* as a task, it directly modifies the product of the LTL automaton and robot model.

Time-extended assignment refers to situations where robots have been given tasks beforehand and have time to plan for them or situations where robots have some model of how tasks are expected to arrive over time. Most single-robot planning problems consider only time-extended assignment [Smi+11; LPH15a; Din+14]. This is also the case for many multi-robot planning problems.

3.2.2.1 Multi-Robot Tasks

One of the key distinctive features of a multi-robot planning problem is the type of tasks and task assignment. In fact task allocation and planning are generally considered as separate problems for robot teams. Multi-robot task allocation (MRTA) is the problem of allocating tasks to a team of robots [GM04]. Multi-robot planning is the problem of finding plans for a team of robots under some constraints where tasks have already been assigned. For example, multi-agent path finding (MAPF) is one instance of multi-robot planning where collision-free paths are found for a team of robots [Ste+19].

When it comes to task allocation for multiple robots there are a variety of problem categories [Nun+17; KSD13; GM04]. [Zlo06] categorises a task that can be decomposed into simple tasks and distributed to multiple robots as a *compound task.* In terms of task allocation, the simple tasks in the compound task can only be allocated to a robot team in *one way.* Compound tasks can be decomposed and be divided amongst multiple robots whereas decomposed simple tasks must be accomplished by a single robot. On the other hand, the simple tasks in a *complex task* can be allocated to a robot team in *multiple ways.* Both complex and compound tasks are categorised as *multi-robot tasks* in [GM04].

For example, consider the task of gathering data and uploading it for a two robot team. If one of the two robots is capable of gathering data only and the other is capable of uploading only, then allocating these tasks is trivial, since there is only one way to allocate this compound task. However, if both of the robots are able to gather data and both are able to upload it, then allocating the tasks becomes complex. The complexity of the problem is increased if say a robot can gather data and upload it at the same time i.e. it is a *multi-task robot* [GM04]. Therefore, these task categorisations also depend on the capabilities of the robots in the team.

Multi-robot planning problems typically involve simple or compound tasks that have been assigned to robots beforehand. These tasks can either be single-robot or multi-robot depending on the capabilities of the robots. Multi-robot tasks as in [TD16] add to the complexity of the problem. This is because it adds another dimension to the coordination required amongst the robots, that of scheduling their parts of the compound tasks with regard to others. [NTD16] uses a combination of single-robot tasks and multi-robot tasks where the single-robot tasks are local to individual robots and may influence the satisfaction of team tasks which are the multi-robot tasks.

When it comes to multi-robot task allocation problems, the mission for the team is generally specified in one of two ways. The first of these is to specify the mission as a set of simple tasks [Cla+17; Tur+14]. The second is to specify the mission as one complex task that must be decomposed [Zlo06; SBD18b]. If a mission is specified as a conjunction of atomic tasks, breaking it into a set of atomic tasks is simple. However, decomposing complex tasks is not trivial as it requires identifying all simple/atomic tasks within the complex tasks. For example, consider a robot team tasked with removing debris. This involves sensing debris, picking it up and transporting it to another location. [Zlo06] use task trees to represent such a task, breaking the complex task down at each level till it consists of atomic tasks. [SBD18b] decompose tasks specified in LTL by looking at the automaton states and their transitions and identifying transitions that can not be omitted in order to reach an accepting state. Despite these methods, both [SBD18b] and [Zlo06] place some limitations on the type of complex tasks. Clearly, decomposing complex tasks adds another layer of complexity to the problem of task allocation and planning.

As both task allocation and planning under uncertainty are difficult problems, in this thesis we do not consider complex tasks, focusing on a mission specification which is a set of atomic or simple tasks in LTL.

3.3 Solution Objectives

Before we discuss the various solution methods employed in multi-robot task allocation and planning, we survey the objectives these solutions try to optimise. The most common of these is that of minimising cost e.g. the average time per cycle for repetitive behaviours [Smi+11; TD16; Tum+13; Ulu+13; GZ18] or the minimum time to complete all tasks [Guo+16; Leo+17b] or the sum of all robot costs [SBD18b]. The dual of minimising cost, i.e. maximising reward, has also been used, e.g. [Sch+16] maximises the joint reward for a particular class of MDPs called Transition Independent MDPs. Others simply aim to find a feasible multi-robot plan [MKK17; VLB20]. In [MKK17] the feasible plan is for a robot team in a multi-agent path finding problem, so the aim is to find a collision free plan while minimising the average cost. In [VLB20] the aim is to find a feasible plan that satisfies the task specification.

Remark 2 (Objectives under Uncertainty). When uncertainty is incorporated into the robot model, the objective is usually to maximise or minimise the *expected* cumulative reward or cost.

In model checking the objective is verifying the model against a certain property, probabilistic or otherwise. The properties specified can be exactly the objectives discussed above. For example, the PRISM model checker [KNP11] allows users to specify properties such as the maximum probability of of reaching a particular state, the minimum reward accumulated when reaching a particular state etc. [Ash+18; Brá+14] also analyse the probability of reaching a particular state for MDPs for model-checking. [SHB16] illustrate the use of various planning techniques for this property as well.

So far, all solution objectives we have looked at, assume that it is possible for the robot team to complete the mission, i.e achieve all tasks. However, this is not always the case. There can be scenarios where not all the tasks in the mission can be completed meaning that the mission can only be *partially satisfied*

3.3.1 Partial Satisfaction

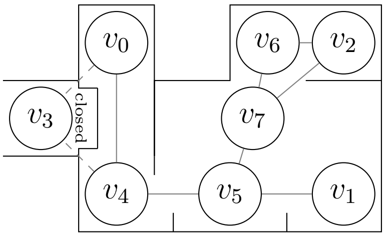


Figure 3.4: Topological map from Figure 3.1 with the door closed.

Example 4. Recall the topological map from Figure 3.1 with the door closed as in Figure 3.4. Assume that the mission is to visit locations v_3 (which is behind a door) and v_5. If the door is closed, then it is not possible to fully satisfy the mission specification. However, in such a case it may still be preferable to visit v_5 instead of giving up altogether.

One way to partially satisfy a mission is to revise the specification to the set of tasks that the robot can achieve [KF14]. Another is to assign rewards to all tasks and then find a solution that optimises these [Tum+13; LK16; Lah+15]. In [Lah+15] safety specifications must not be violated. However, in [Tum+13; LK16] it is possible to violate these tasks. In fact [LK16] balances specification violation with probability of mission satisfaction. In [GZ18] a similar objective is used, that of minimising cost such that the probability of violating the mission specification is below a threshold.

Another way to partially satisfy a mission is to have multiple mission specifications to choose from. These can then be ranked in order of preference by the end user. In [Mei+15] the objective is to start with the most preferred mission specification and check if it is satisfiable. If not, the next preferred specification is checked until a satisfiable specification is found.

When it comes to partial satisfaction, assigning rewards or costs to specifications can be tricky for the end user. This is why [LPH15a] uses the automaton corresponding to the LTL specification to generate a *task progression metric.* It maps each state in the automaton to a value representing how close that state is to reaching an accepting state i.e. one where the LTL formula has been satisfied. Similar to a reward function, this metric can then be used to satisfy as much of the specification as possible. The objective in [LPH15a] is to maximise the probability of satisfying the mission, using the progression metric and cost as tie-breakers. Unlike [LK16; Tum+13], this does not incorporate violating some part of the specification in order to achieve another. Maximising the probability of mission satisfaction is also the objective of works in [Ash+18; Kol+11; SHB16]. In [TTT17; Tei12] the objective is closer to [LPH15a], maximising the probability of satisfaction while minimising cost.

Remark 3 (Quantitative Plan Properties). The work in this thesis lies at the intersection of verification and robotics. Therefore the objective is to be able to provide quantitative values on certain properties of the mutli-robot plan. To this end, [LPH15a; Ash+18; Kol+11; SHB16; TTT17; Tei12] all provide an exact value of the probability of mission satisfaction. While [LK16] does provide an exact value of the probability of mission satisfaction, there is no way to verify the user assigned costs. As we shall see, the approaches in [Ulu+13; TD16; Leo+17b] all aim to generate *correct-by-construction* plans. This means they use formal methods to model robots and specify tasks. The algorithms used are also described formally. To that end, the values they get for their solution objectives i.e cost, can also be seen as quantitative properties of the resulting plans. However, adjusting these plans for unexpected behaviour [Ulu+13] means that these values are no longer admissible.

3.4 Solution Methods

Remark 4 (On traditional AI planning techniques). In the previous sections we discussed various ways to model robots interacting with the environment and how to specify tasks for those models. The focus has been on paradigms that are typically used to model uncertainty in the environment. There is a wealth of AI Planning frameworks which traditionally focused on deterministic systems. These include the popular Planning Domain Definition Language (PDDL) [McD+98], Answer Set Programming (ASP) [MT99; NSS99] and many other symbolic languages such as BC [LLY13]. Each of these frameworks can be used to define a domain (i.e. the robot and its interaction with the environment in our case) and goals (or tasks). [YL04] extended PDDL to express planning domains with probabilistic effects i.e. Probabilistic PDDL. Recently, [EP18] discussed the use of ASP for robotic planning problems and provides solutions to the challenge of modelling partially observable environments and non-deterministic actions. However, unlike MDPs, PDDL and ASP descriptions are cumbersome to write due to the very structured nature of these frameworks. Most off-the-shelf solvers (such as POPF [Col+10] or OPTIC [BCC12]) for problems represented in these frameworks focus on single agent planning. For example [Col+19] which generates plans for a robot in space or [LZ11] which describes a solver for MDPs represented in probabilistic PDDL. Another disadvantage of these frameworks is that temporal tasks are harder to describe since they require time to be encoded explicitly, unlike in LTL where the eventually and next operators do not need any explicit timing. Using languages such as LTL also decouples the task specification from the model itself. As shown in Section 2.3 LTL simply uses the atomic propositions from the model. Nonetheless, the techniques and optimisations used in these solvers are still useful and we discuss some of these here (not in light of these solvers in particular).

In this section we discuss the various solution methods for single-robot planning, multi-robot planning and combined multi-robot task allocation and planning problems. All the solution methods discussed below are dependent on the various aspects discussed in previous sections i.e. the types of robot models used, the way tasks are specified, the types of constraints on those tasks, considerations of uncertainty and the objectives of the problem.

Remark 5. We do not consider solutions to multi-robot task allocation alone e.g. [ZS06] since that is beyond the scope of this work. We also do not look at solutions to multi-robot task scheduling problems e.g. [Pal15] since these aim to provide a schedule of tasks but not a motion plan. Other problem domains similar to ours are vehicle routing [TV02] and multi-travelling salesman problems [LS09].However, these make certain assumptions that we do not and vice versa and so are also beyond the scope of this work.

Remark 6. We also do not look at strategies for individual robot recovery due to hardware or software failures such as those in [CR20; LPM18].

3.4.1 Single-robot Planning

In this section we survey approaches used to solve single-robot planning problems where tasks are specified in LTL or an exact value for a particular plan property is provided or only a part of the task can be completed. Some of these come from the verification community with a focus on model-checking, some are from the robotics community with a focus on *correct-by-construction* plans (inspired from model checking) and others are from the robotics community at large.

We first describe exact methods for single-robot planning. We then look at search based methods, some of which can also be used to provide optimal solutions. Note that in this section we use the term policy more frequently since solutions to MDP based problems are policies.

3.4.1.1 Exact Methods

Recall from Section 3.2.1 that LTL can be syntactically divided into safe and co-safe LTL. Tasks specified in safe LTL include safety tasks such as avoiding a particular location and repeating tasks such as going to a location infinitely often. An LTL formula can be converted to an automaton. As explained in Section 2.3 this can be combined with the robot model (e.g. an MDP). As in [KP13], exact methods for solving these models can then be applied to the combined model for model-checking or to generate solutions with guarantees.

1. **Dijkstra's Algorithm:** Dijkstra's algorithm [Dij59] is a method used to find the shortest path from one node to another in a graph. Dijkstra's algorithm and its variants form part of the group of shortest-path algorithms (see [Mad+17] for a survey of these). Shortest-path algorithms can easily be modified and applied to robot planning problems. For example [Smi+11] uses a modified version of Dijkstra's algorithm to find the shortest cyclic path which satisfies an LTL formula for a repeating task. The work is similar to [Din+14] in that both use a two-part approach to solving for LTL task specifications. They place a constraint on the form of the task specification $\varphi \wedge \mathtt{G}\,\mathtt{F}\,\pi$, read as φ and always π, meaning do φ once and repeatedly do π. Both [Smi+11; Din+14] operate on the product of the robot model and the LTL automaton. [Smi+11] uses a DTS to model the robot, while [Din+14] uses an MDP. The first part of the solution algorithm solves the non-repeating task φ and the second finds the shortest cyclic path which satisfies π.

2. **Policy Iteration:** While [Smi+11] adapts Dijkstra's, [Din+14] finds a feasible solution using pre-existing methods used in model-checking [Bai+14] and then uses policy iteration [How60] to improve the solution. Policy iteration (PI) is an MDP algorithm used to compute the optimal solution (or policy) by starting with an arbitrary policy and iteratively evaluating and improving the policy until an optimum is reached. Policy evaluation uses the value function (see Definition 5) to evaluate the policy. It improves the policy by selecting the action that gives the highest value at each state. For both [Smi+11; Din+14] the inclusion of a repeating task combined with exact methods contributes to the computation time and slows these algorithms down.

3. **Value Iteration:** Similar to policy iteration, value iteration (VI) [Bel03] is an MDP algorithm which uses the value function to iteratively update the value of a state. VI starts at the goal state and percolates to the initial state of the MDP. Once all the values for all states have converged, the solution or plan is generated by

choosing an action for each state based on the state values. While policy iteration searches the policy space, VI operates on the state space [MK12] which makes it more computationally expensive. This is why VI is more suited to scenarios without cycles, e.g. problems with co-safe LTL specifications. In [LPH14], VI is used to generate the optimal policy for a single robot MDP given a set of tasks as the mission specification. The product MDP state includes a variable corresponding to the automaton of each LTL task. Therefore, this approach would scale poorly for a large number of tasks. However, the problem in [LPH14] also looks at adding tasks while the policy is being executed i.e instantaneous assignment. When a new task is received, the automaton for the new task is multiplied with the current product MDP ensuring that all the previous task automata states are preserved. This allows for the generation of optimal policies on the fly considering all unfinished tasks (old and new).

4. **Linear Programming:** Another widely used technique for generating optimal solutions is Linear Programming (LP). It is used to optimise a linear objective function given a set of linear equality and inequality constraints. Since a detailed description is beyond the scope of this thesis, we refer the reader to [Van20] for resources on LP. Similar to [LPH14], the solution objective in [TTT17] is also one of probability maximsation. However, [TTT17] do not specify tasks in LTL but use an LP formulation to find a policy that minimises the cost from the set of policies that maximise the probability of reaching the goal. The goal here can be considered the mission specification, though it is specified as an MDP state. [TTT17] view it as a network flow problem where the objective is to find the policy with the lowest *occupation measure* cost. An *occupation measure* represents the expected number of times an action is executed in a particular state. The LP is solved using i-dual [Tre+16] which borrows ideas from $A*$ search by starting with the LP formulation of the initial state and iteratively exploring the graph until the goal state is reached. The need for an iterative solution is because LPs do not scale well.

3.4.1.2 Exact Methods for Partial Satisfaction

LP approaches have also been used for partial satisfaction of LTL mission specifications. [GZ18] uses a Linear Programming approach encoding the product MDP as a constraint optimisation problem with the objective of minimising cost while keeping the probability of specification violation under a threshold. Similarly VI has been used for partial satisfaction as well. In [LPH15b] VI is modified to include more than one objective, Nested VI [LPH15a], maximising probability of satisfaction using cost and task progression as tie breakers. [Lah+15; LK16] both use VI to partially satisfy a specification using user defined costs for atomic propositions. [LK16] generates policies by converting partial satisfaction to a multi-objective problem, considering the trade-off between the expected distance to satisfaction and the probability of satisfaction. Both the task progression metric and distance to satisfaction metric use the automata states to determine how far the robot is from satisfying the mission. The most important difference between the two is that [LPH15b] does not require user input, therefore, it is able to provide a guarantee on the resulting plan. The use of costs to aid in partial satisfaction is also used in [Lah+16] which uses an iterative planning approach as in [LPH14], adding dynamic obstacles as tasks.

Another approach to partial satisfaction is that of specification revision as in [KF14]. Atomic propositions that make the specification unsatisfiable are iteratively removed using an algorithm based on Dijkstra's shortest path until the specification can be satisfied. Also relevant to partial satisfaction is preference based planning, where task specifications are ranked in order of preference and the aim is to find a task specification that is satisfiable. In [Mei+15] this is done by encoding the problem as a quadratic programming problem and iteratively finding a specification that has a solution.

3.4.1.3 Search Based Methods

The main drawback of all exact methods is that they do not scale well because they consider the entire reachable state space of the model. As models (and LTL automata) grow larger, these methods become intractable. Using LTL to specify tasks adds at least one extra dimension to the input of the planning algorithm. As the number of formulae in the LTL specification increases, the states in the corresponding automata or automaton also increase. Therefore, while LTL is able to provide a succinct and formal way to specify tasks, it does have some overhead. As we will see in the next section, this is why multi-agent planning solutions without formal specifications can outperform those that use them. However, verification of such algorithms is not as easy. Needless to say that exact methods for solving MDPs with or without formal specifications can also become intractable as the model size increases. For this reason many researchers turn to using sampling-based search methods to generate solutions which do not explore the entire state space but sample states as needed. As the number of samples increase, the solutions get closer to the optimal solution. In fact, some methods are able to generate optimal solutions without covering the entire state space.

Before we discuss relevant work, we provide a short overview of the main search methods (see [MK12] Chapter 4 for detail).

Real Time Dynamic Programming (RTDP) and extensions Real Time Dynamic Programming (RTDP) [BBS95] is a sampling-based search algorithm for MDPs with a given initial state and a given goal state. Similar to VI, it uses the value function (see Definition 5) which maps each state to a value. Informally, this value of a particular state tells us how easy it is to reach the goal state from this state. The search starts at the initial state of the MDP, assigning a value to the state based on some heuristic. It then chooses the action in that state based on some action selection method and then samples one of the successors of that action. The process of action selection and successor sampling is repeated until a goal state is reached (called a trial). Once a goal state is reached, a backpropagation (or *backup*) process is initiated where the value of each state seen in the trial is updated. Trials are repeated until the value of the initial state converges i.e. the search terminates. Therefore, RTDP is able to find the optimal solution. RTDP has many variations that aim to improve its convergence such as Bounded RTDP [MLG05] which uses upper and lower bounds on the values of states and Labelled RTDP [BG03] which avoids exploring paths where state values have converged.

Monte Carlo Tree Search (MCTS) Monte Carlo Tree Search [Cou06] is a very popular sampling based heuristic search method with many variants. The core of the algorithm consists of 4 stages: selection, expansion, simulation and backpropagation (or backup). Like RTDP, MCTS too starts at the initial state of the MDP, building a search tree as it goes along.

> *[Selection]* This step depends on previous explorations of the tree. If the tree has not been explored, the selection step is effectively skipped. If the tree has been explored, the selection step starts at the root node (the initial state of the MDP) and continues to select the best child node, until an unexplored node is reached.
>
> *[Expansion]* If this unexplored node is not a goal state (or any other terminal state), this node is expanded i.e. one of its successors is chosen. In terms of the MDP, an action is taken and a successor state is sampled.
>
> *[Simulation]* From this successor state, a trial to a terminal or goal state is simulated. This trial is also called a playout. The assumption is that there exists a simulator which when given a particular successor state will give us a final state with a value attached to it.
>
> *[Backpropagation]* This value is then used to update the selected successor state and all its predecessors that were in the search path.

Note that MCTS is very similar to RTDP with the exception of the simulation step. This is why it is very easy to mix and match components of MCTS and RTDP (and their variants). In fact, [KH13] introduced the Trial-based Heuristic Tree Search (THTS) framework for this purpose, allowing a myriad of sampling-based search approaches to be easily implemented and mixed. For a detailed treatment of MCTS we refer the reader to [Bro+12].

Rapidly-Exploring Random Trees (RRTs) RRTs [Lav98] were originally used to find motion plans for robots in the continuous domain. Similar to the two approaches above, this is a sampling based method. It iteratively builds a tree from the initial state to the goal state. To give an intuition for this, we use some arbitrary notation in the following text. Given a set of states in the continuous domain, the tree is rooted at the initial state. It then samples a random state, s_r from the given set and finds the node, n_c from the tree that is closest to this sampled state. It then adds a new node n_n to the tree which is a state between s_r and the state in n_c. Before adding this new node, the algorithm checks whether or not it is possible to get from s_r to the state in n_c. The state of n_n is sampled along this path. Once the the goal state is added to the tree, the process stops and a path from the root node to the final (goal) node is found using a tree search method such as best first search. RRTs and their variants can be used to find feasible solutions. As the number of samples increase, these methods are able to find optimal solutions as well.

We can now proceed to survey various search based methods employed to find solutions to planning for single robots under uncertainty. Some of these come from the verification community while others from the robotics and AI planning communities.

1. **Optimal Solutions:** In [Ash+18] sampling based heuristic search algorithms are used to *verify* reachability in MDPs i.e. the probability of reaching a particular state in an MDP. While the work comes from the verification community, the authors do not explicitly state the use of any formal languages for task specification. However, they do perform tests on benchmarks from the verification community. The two algorithms used in [Ash+18] are Monte Carlo Tree Search (MCTS [Cou06]) and Bounded Real Time Dynamic Programming (BRTDP [MLG05]). In fact, the authors mix and match different components of these algorithms to create several hybrid

search methods. It is shown that these methods are able to generate solutions much faster than VI while exploring a small part of the state space. As noted in [Brá+14] in order to verify the probability of reaching a state using such methods, zero-reward cycles in the MDP need to be identified and dealt with. The presence of these cycles leads to the algorithms getting stuck in local optima. In order to get to the global optimum, these cycles must be removed. In fact [Ash+18] uses the technique from [Brá+14] to revise the MDP and deal with these cycles. The work in [Brá+14] uses LTL for task specification and verifies MDPs using BRTDP and delayed Q-learning (DQL [Str+06]). Both of these are search based alternatives to VI, with DQL focusing on situations where a simulator is available but the full model may not be available. Both [Ash+18; Brá+14] show that these techniques outperform VI based model checkers.

[Kol+11] solves the same problem from the perspective of the AI planning community i.e. the problem of determining the maximum probability of reaching a particular state in an MDP. It uses the term *MAX-PROB* to define such MDPs. It introduces a general framework for heuristic search for such problems, Find-Revise-Eliminate-Traps (FRET). In a nutshell, the algorithm identifies cycles and assigns state values and state-action values to all states and state-action pairs in the cycle such that the next iteration of the search is able to escape these cycles. These cycles are called *zero-reward* cycles. [Brá+14] also employs a similar method to escape cycles. A detailed study of solving *MAX-PROB* MDPs using heuristic search algorithms can be found in [SHB16] where zero-reward cycles are dealt with using a modified version of FRET.

The algorithms in [Kol+11; SHB16; Brá+14; Ash+18] all focus on generating optimal solutions using heuristic search with a small error threshold. To that end, they can all be used to produce exact guarantees on reachability properties for MDPs. However, all of these approaches require the detection of strongly connected components (cycles) in the MDP. In fact, this process occurs multiple times. Generating such an

MDP where cycles are detected and collapsed is formally referred to as a quotient MDP in the verification community. The time complexity of this is quadratic in the size (states and transitions) of the MDP [De 98; Bai]. Furthermore, there is some extra book-keeping required to keep track of these cycles which too adds to the complexity of the approach.

2. **Feasible Solutions:** Another search based method that has been employed to find solutions to single-robot planning problems using LTL is Rapidly-exploring Random Trees (RRTs). [VLB20] use RRTs and their graph variant Rapidly-exploring Random Graphs (RRGs [Kal13]) to generate plans for a discrete transition system (DTS). RRGs are used to generate a policy for the global specification and RRTs are used to avoid obstacles on the local level. Obstacle avoidance is considered to be a dynamic task, triggered when an obstacle is detected. Instead of generating the full joint product RRTs and RRGs can be used to generate parts of the product using an incremental sampling approach. The RRG in [VLB20] generates a sparse product by making sure that the state to be sampled is a certain distance away form the other states already in the graph. The RRT is used to generate plans between two RRG states. Both RRGs and RRTs are probabilistically complete, i.e. as the number of samples approaches infinity the solution reaches the optimal value. The efficacy of such algorithms depends on the number of samples and the *density* of the underlying graphs. The term density here refers to the degree of connections each vertex in the graph has. For dense graphs, their performance deteriorates again due to the number of samples required to reach a feasible solution.

3.4.1.4 Search Based Methods for Partial Satisfaction

Search based approaches are also used in solutions to partial satisfaction problems. In [Tum+13] depth first search is used to iteratively find cyclic paths that complete as many of the tasks in the mission specification as possible i.e. find the least violating policy. Like [Lah+15] each task is assigned a reward by the end user and violating some task at the cost of satisfying another is allowed. Unlike [Lah+15], tasks in the mission can be overlapping. The use of an iterative depth first search signposts the algorithm's performance. If the plans are very long, then the approach may not scale well, since the better approach would be to do one full depth first search instead of an iterative one.

3.4.2 Multi-robot Task Allocation and Planning

In this section we survey recent works in multi-robot planning which either use LTL for mission specification or model uncertainty. We also look at relevant work in task allocation and planning.

As mentioned in Section 3.1.1 single robot planning solutions can be applied to multi-robot models easily if the full cartesian product of all single robot models is used. This is because the full cartesian product can be treated as a large single robot model. However, single-robot planning solutions also suffer from scalability issues (see Section 3.4.1).

When it comes to solution methods for multi-robot planning problems the trade-off is not only between scalability and optimality but also between various levels of coordination among robots. For example, the full cartesian product (such as an MMDP for MDPs) is an example of a fully coordinated model. At each state of the joint model, each robot is aware of the state and action choices of all other robots. Such a model can also be used to simultaneously allocate tasks and plan. Fully coordinated models generally rely on a centralised planning approach and generally assume fully synchronised actions. Partially coordinated models allow robots to plan without a central decision making approach but with some degree of coordination between the robots. A partially coordinated model is likely to use a semi-centralised or decentralised planning approach. In fact such models are generally referred to as decentralised models.

Finally an uncoordinated model would be one where the robots are not able to coordinate with each other at all. This is the same as solving n single-robot problems where n is the number of robots in the team, which defeats the purpose of multi-robot planning altogether. There is always some degree of coordination built into multi-robot planning algorithms.

The following sections present a summary of the various solution methods grouped on the basis of the ways used to reduce problem complexity or the solution algorithms themselves.

3.4.2.1 Finding Dependencies and Limiting Planning Steps

One way to avoid using the joint model when generating multi-robot plans is to find dependencies between robots and consider joint information only at those states. This is particularly common when the application domain is restricted. The use of domain specific assumptions can be used to simplify the models. For example, in [Sch+16] the maintenance planning domain is considered. Each robot is given a set of independent road maintenance tasks, which it must complete at a minimal cost. Each task may be delayed with a known probability and the overall disruption to traffic is represented as a joint cost. Both these assumptions are incorporated into the joint model which is a transition independent Multi-agent MDP (TI-MMDP). TI-MMDPs assume that agent costs or rewards depend on joint states and actions but each robot's transition probabilities are not affected by other robots and ignore collisions between robots. This results in fewer transitions than when using an MMDP. [Sch+16] introduces conditional return graphs (CRGs) to model the rewards associated with actions of robot i that influence the actions of robot j. For example, consider two robots i and j. The CRG for i at state s_i with action a_i will include information about any changes in the reward due to actions of robot j. Each edge from the vertex for s_i, a_i denotes a different reward. Multiple edges can lead to the same successor state of robot i. These graphs are artificially limited in depth by thresholding the number of time steps allowed (called the time horizon). A branch and bound policy search over these conditional return graphs is used to find the optimal policy for the team. Despite evaluating fewer joint states and actions and an average solution time of a little over 10 seconds, the algorithm does not perform well for more than 7 agents and a time horizon of more than 5. This makes such an approach unsuitable for scenarios where many robots need to be deployed for long periods of time.

Similarly [TD16] introduce an *online receding horizon* approach to plan for multiple robots with local LTL tasks. Online receding approaches aim to provide quick feasible solutions caring only about the next few steps, i.e. the horizon is fixed to a certain number. These approaches are not always optimal but are built for scenarios where it is not possible to compute a plan for a robot before execution. However, it is still possible to compute a solution offline through simulations or if the entire model is available. Some of the LTL tasks in [TD16] are multi-robot tasks i.e they require the help of other robots. Robots whose tasks are interdependent are grouped into *dependency partitions* and a joint product is generated upto to a predefined number of steps or horizon, h. After executing actions for those h steps new dependency partitions may be made and the process is repeated until the specifications are satisfied. To handle synchronous robot actions for tasks that require multiple robots, an event triggered synchronisation policy is used. The states where synchronisation may be needed are determined during planning time.

3.4.2.2 Compact Team Models

[TD16; Sch+16] avoided working on the joint product model by breaking them down or replacing them with more compact models. Similarly, [Ulu+13] uses recursive depth first search to build a team model based on action durations, introducing *travelling states* where at least one robot is ready to perform a new action. As a result, some possible states are pruned. In fact for the fully centralised approach [Ulu+13] also prunes out all *travelling states*. This pruning is dependent on action durations being different, if they are all the same, then it would result in a full joint model. Like [TD16], [Ulu+13] also uses a synchronisation policy to deal with uncertainty.

In [SBD18b] the full joint product model is replaced by a sequential team model with *switch* transitions linking one robot's product model with the next one's. The robot's local product model consists of states corresponding to all tasks in the mission specification and its deterministic transition system states. The switch transitions originate from states where tasks are completed and terminate at the initial state of the next robot, preserving the automaton states.

Joint product models can also be reduced by abstracting sequences of states and actions. For example in [SBD18a] a task allocation and planning approach is presented where robot actions are abstracted as macro-actions called *options* [SPS99]. These options are sequences of actions that allow the robot to progress from one state of the LTL mission automaton to the other. Through these options, each robot bids on a part of or all of a task. Essentially each robot bids to progress towards an accepting state of the LTL specification's automaton. This is done in sequence. An estimate of the cost-to-go i.e. the long-term effect of a task assignment is also incorporated into the algorithm. The resulting approach is an online receding horizon algorithm that allows revision of task allocation.

3.4.2.3 Sampling-based Search

In contrast to simplifying robot models using domain knowledge, sampling based algorithms can be used to avoid building the entire joint product, generating relevant states and actions as required. For example [KZ17] uses an approach inspired by RRT* [Kar+11], an extension of RRT. The robots are modelled as weighted deterministic transition systems and the task specification may include repeating tasks. Each robot is assigned its task set beforehand. The algorithm creates two trees one for prefix (the non repeating task) and one for the suffix (the repeating task). Solutions to repeating tasks are cyclic paths which requires a modification of RRT* since it generally works on trees. The sample space of the algorithm is that of the joint model. This work is extended in [KZ20] to simultaneously allocate tasks and plan for a team of robots. Similarly, [Cla+17] performs simultaneous task allocation and planning using Monte Carlo Tree Search (MCTS). As mentioned earlier, such approaches are asymptotically optimal and therefore the time to generate an optimal solution may be very long.

Related to MCTS is the application of Q-learning [WD92] to generate plans and allocate tasks for a multi-robot team. In [SLB20] a reward function is created using the LTL automaton's states. This is then used to guide the robots towards achieving as much of the task specification as possible. To guarantee convergence a discount factor is used. This is common in many Q-learning approaches. However, it means that tasks achieved much later in the future are considered far less important. Q-learning is not a sampling based search approach but due to its similarities with MCTS and RRT we presented it here.

3.4.2.4 Using Single-robot Plans

[VKM17] introduces a decentralised approach that uses message passing between robots to generate plans. Instead of generating individual initial plans, agents communicate with each other and generate a joint initial plan. Each robot plans its path to the next location using the RRT algorithm. Neighbouring robots create a network, elect a leader and create a tree which is used to compute plans for each robot. If the plans are collision free, the robots synchronise and execute the plan, otherwise they re-plan. The communication overhead in this approach increases more than linearly as the number of robots and state space increases so does the execution time. Therefore the algorithm can not handle large domains or a large number of agents.

Similarly [MKK17] generates collision free plans for a robot team using a conflict based search [Sha+15] approach. Robots plan individually then a high level planner checks if these plans are collision free, if not the plans are revised. Uncertainty is handled using a communication policy which allows robots to communicate at certain points during policy execution. As uncertainty increases, the amount of communication also increases. [Zha+17] also seeds its algorithm with single robot plans. These are then revised by incrementally increasing the *negotiation depth* i.e. how much a certain robot considers other robots. Changing the negotiation depth allows the algorithm to moderate between a centralised and decentralised plan. However, it is unclear how well the algorithm will scale. It may do really well in situations where interactions between robots are sparse since the negotiation depth will be low. Furthermore the model of uncertainty is restricted to a subset of available actions.

$UM*$ in [WC17] also refines single agent plans to solve the problem of multi agent path finding under uncertainty. It is an extension of $M*$ [WC11], an optimal multi-robot path planning algorithm similar to $A*$ search which, considers only a limited set of a robot's neighbours. $UM*$ adds a conflict set and a coupled set to each node in the search. The conflict set consists of robots that may collide with each other with some probability. The coupled set consists of robots that may be able to prevent the violation by taking a different action and is used to choose which nodes to expand. The result is a policy for each agent that guarantees a bound on the combined probability of collision for all robots. The uncertainty in this problem is limited to a change in the robots' speed.

[GD17] also uses a decentralised approach to planning with LTL mission specifications which has the added advantage of allowing robots to swap goals if needed. Each robot generates plans from its initial state to all accepting states of the mission using Dijkstra's algorithm [Dij59]. Robots begin to execute these plans but are governed by a coordination scheme that allows individual robots to request for assistance or assist other robots. Robots are also able to swap goals through this coordination scheme. However, neither the robot models or the coordination scheme model uncertainty.

Recent work in [Car+20] has also used a decentralised approach to multi-robot task allocation and planning. The problem is encoded using PDDL and is aimed at autonomous underwater vehicle (AUV) teams. Tasks are allocated by decomposing the goals geographically using a clustering algorithm, then identifying sets of tasks each robot can reach and finally allocating robots according to the maximum number of tasks they can complete. Tasks are allocated through a vehicle routing and scheduling mechanism which takes into account the time taken to complete a task and the distance between tasks for each robot. Once the tasks are allocated, the temporal planner OPTIC [BCC12] is used to generate plans for each robot. One of the experiments in [Car+20] introduces failure in task execution and allows individual robots to replan. It does not consider task reallocation in case a robot stops functioning entirely e.g. runs out of battery. Since PDDL does not allow modelling uncertainty, there is no way to give an exact quantitative guarantee on the plan.

3.4.2.5 Priority Based Planning

Another way to plan for multiple robots is to use priority based planning algorithms. Robots plan in some specified order and each robot plans considering the plans of the robots before it. Such an approach is used in [Tur+14; Che+17; Str+20]. [Tur+14] considers homogeneous robot teams using the Hungarian algorithm to allocate tasks and then prioritises robots. [Che+17] uses priority based planning to generate provable safe policies in the continuous domain under uncertainty. To plan for each robot it uses the Hamilton Jacobi Value Iteration (HJ VI) which works on a discretisation of the state space.

3.4.2.6 Homogeneous Robot Teams

Planning for homogeneous robot teams means that the team models can be more compact since all robots share the same actions and states.

Petri Nets (PN) are a great choice for such teams since they do not scale exponentially with the number of robots. Infact robots are modelled as tokens, since all robots have the same actions and states. Tasks can also be allocated to robots in conjunction with planning. Another benefit is that of being able to control the number of robots that can travel together, for example the number of cars in a road network. [LL19; Man+19; KM20; HKM20] all model multi-robot problems using PNs with all but [KM20] combining task allocation and planning. PNs can be used to model uncertainty as well e.g. [LL19] model the possibility of robot failure and [Man+19] model uncertainty travel times. [KM20; HKM20] encode PNs as Integer Linear Programming problems to be solved using off-the-shelf solvers whereas [Man+19] converts these to MDPs to solve them.

Constraint-based solutions can also be used to solve multi-robot planning problems with homogeneous teams. They involve optimising objectives subject to a set of constraints. These constraints can be used to model robot actions or collision avoidance or any other assumptions of the environment and team. For example, [Sah+14; Des+17; GMS17] all use Satisfiability Modulo Theory (SMT) solvers to allocate tasks and generate feasible plans. [Leo+17b] uses Optimization Modulo Theory (OMT) to do the same. The core of these solutions is to model robot locations and time as variables and then place constraints on those using linear equations, inequalities and/or inequalities. The extensive encoding means that these solvers do not scale well as the state space increases. Therefore, most of these approaches do not model uncertainty or robots with different capabilities. They assume that a local planner will handle uncertainty, which is possible for dynamic obstacles. However, it is not possible when robot behaviour itself is uncertain. [GMS17] detects robot failure and reallocates tasks when a robot fails. It also models avoiding dynamic obstacles as a task which triggers a replanning cycle. Modelling these uncertainties instead of dealing with them dynamically would improve the team's performance.

[Des+17] is noteworthy because it introduces a provably correct decentralised asynchronous motion planner for distributed robots. An extension of the state machine based programming language P [Des+13] is used to model the robots and specify tasks. The system is verified and a prioritised planning approach is used to generate plans using A* search or an SMT solver. However, the system does not incorporate uncertainty.

3.5 Discussion

The field of multi-robot planning is evolving with the frequent deployment of robot teams in real world scenarios. The research focus in the field is moving from generalised frameworks to application specific ones [Sch+16; MK16; MKK17; Ma+16] as such frameworks are more useful in practice. To this end, there is a variety of multi-robot planning problems. Solutions to these problems balance a trade-off between optimality and scalability. Optimal solution methods attempt to bypass the scalability issue by exploiting assumptions of the problem formulation. For instance [Ulu+13; SBD18b] disregard uncertainty in their solution methods, using deterministic transition systems to model robots and generating *correct-by-construction* plans using techniques from model-checking and verification. [Sch+16; WC17; MKK17] incorporate limited models of uncertainty based on the problem formulation, such as a delay in motion or a joint reward but independent robot models. In the same vein, some solution methods exploit the homogeneity of the robot team [KM20; LL19] modelling all robots as having the same capabilities.

As the complexity of the problem increases, researchers turn to methods for generating feasible solutions instead of optimal ones. For example [KZ20] use a sampling based approach that is asymptotically optimal. Sampling and other search based methods work well on sparse team models, i.e. ones where space of potential solutions is small [Kol+11; KZ20; Ash+18; Brá+14]. Priority-based planning is another technique that forgoes optimality of team performance for scalability [Che+17; Str+20; Tur+14].

The complexity of multi-robot planning problems is exacerbated by the need for formal verification of multi-robot systems [Luc+19]. Providing guarantees on the behaviour of robot teams is imperative to their successful deployment in the real world. For example, this is one of the core reasons that hinders large scale use of autonomous vehicles [PMP20]. In order to verify the behaviour of the robot team, formal methods must be used. As we have seen, [Des+17; GMS17; Leo+17b] both use a variety of formal method techniques to ensure that the system is *correct*. However, their robot models do not consider uncertainty. Furthermore, the methods used are constraint satisfaction based or linear programming based and these do not scale well in practice. Other works such as [Ulu+13; MK16; TD16] deal with the problem of providing guarantees on team behaviour by ensuring that robots communicate with each other during policy execution. While they are not able to provide exact values on team behaviour, they can guarantee that it will follow the specified team plan. Verifying solutions to MDPs through model-checking is a complex process [BHK19]. As we saw in [Ash+18; Brá+14] techniques borrowed from AI search can be applied to provide guarantees on MDPs as traditional model-checking approaches do not scale well. However, it is clear that *correct-by-construction* plans (such as those obtained by combining formal specifications and models) are of importance to the deployment of robots in the real world.

The goal of this thesis is to provide verified solutions to multi-robot task allocation and planning problems that are robust to uncertainties such as the possibility of robot failure. While different elements of these are found in literature, there is no unified solution to this problem. When it comes to multi-robot models that incorporate uncertainty, we have seen that MDPs are used extensively. In most indoor robot deployment scenarios it is possible to fully determine the state of the robot and so we believe that MDPs are a good choice for robot models. In terms of task specification, verified solutions require a formal method to describe these tasks. LTL provides such a framework, that is both intuitive and expressive, capturing a series properties common in real scenarios. It is also worthwhile to investigate whether simultaneous task allocation and planning [SBD18b] with the addition of uncertainty can offer any gains to the team's performance when compared with separated task allocation and planning [Tur+14]. There is indeed a variety of solution methods but those that employ communication schemes during plan execution can not be used to provide exact quantitative values on properties of the plans beforehand. Another hindrance to providing guarantees is that of user defined reward functions since these are neither intuitive nor formally verifiable. Therefore there is a need for the generation of automatic reward functions for example [LPH15a] leverages formal task specifications to generate a reward. Lastly, there has been very little work on high level robot behaviour in the context of multi-robot task allocation and planning e.g. [GMS17] reallocates tasks if robots fail but it does not provide any guarantees on the overall team plan.

Summary This chapter surveyed various approaches to robot planning with a focus on the types of models uses and logic based task specifications. It provided a brief review of the taxonomy used to categorise tasks in robot planning which is intrinsically linked to the scope and complexity of the planning problem. This review was followed by a holistic summary of the various approaches to solve robot planning problems with a focus on those that use some kind of formal methods. The chapter concluded with a discussion on the state of multi-robot planning and task allocation in the context of this thesis.

The literature surveyed in this chapter, showed that MDPs are a suitable modelling paradigm for indoor mobile robots with uncertain action outcomes. For example, they can be used to model critical robot failure as an action outcome. Specifying tasks for robots modelled as MDPs can also be done in a variety of ways. LTL is gaining ground as a task specification language for such robots due to its expressivity and ties with formal verification. In fact with robots as MDPs and tasks as LTL formulae, techniques from probabilistic model checking can be used to generate quantitative guarantees on robot plans. Another key aspect of planning for robot teams with one global mission is that of task allocation, which is computationally demanding. Decomposing a complex global mission into simple tasks that can be allocated to each robot in a team is also computationally expensive. The work in this thesis ignores such decompositions, using a global mission for the robot team that consists of simple tasks. The next chapter describes the problem of verified multi-robot task allocation and planning under uncertainty which is the core focus of the work in this thesis.

Chapter 4

Problem Formalisation

We now formalise the problem of multi-robot planning under uncertainty that we tackle in this thesis. We explain the modelling of each individual robot, the joint multi-robot model, the mission specification for the robots, and then how these are combined to define a problem over an MDP. More specifically, we use these to define the problem of maximising expected number of tasks completed for a multi-robot team under uncertainty. Next, we describe the tools used to implement the algorithms presented in this thesis and the test environments. Finally, we illustrate the limitations of solving the specified problem using value iteration (VI) which is an exact method used to find the optimal solution. VI has been used in both robot planning e.g. [LPH14; LPH15a; LPH15b; Lac+19] and formal verification, particularly model checking MDPs [BHK19; KP13; Bai+14].

4.1 Components

4.1.1 Single-Robot Models

Overall, we assume a set of n robots. The operation of each individual robot i as it attempts to perform tasks is modelled by an MDP $\mathcal{M}_i$. The state space of $\mathcal{M}_i$ comprises both the *local state* of the robot and also the *global state* which is a feature of the environment itself. Therefore, the global state is common to all robots. In our work, the local state

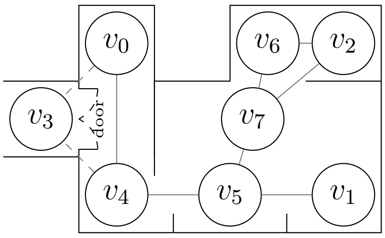


Figure 4.1: Example topological map.

is typically the robot's location within a topological map used to model its environment. The topological map is a graph where nodes represent physical locations of interest and edges represent the robot's ability to move between nodes.

Example 5. Figure 4.1 shows the topological map for a toy example with 8 locations $v_0, \ldots, v_7$. Edges drawn as solid lines indicate that the robot can move freely between a pair of locations. A dashed edge indicates a constraint on the global state, in this case on the status of a door.

We assume that the global state comprises k separate state features. In the example above, there is a single global feature representing the state of the door. The MDP $\mathcal{M}_i$ for robot i has an action set A_i, which is partitioned into actions A_i^l that update the local state (e.g., representing navigation between locations) and action sets A_j^g that update global feature j (e.g., opening or closing a door). The MDP $\mathcal{M}_i$ for robot i is called a *local MDP* and is defined as follows.

Definition 15 (Local MDP)**.** A *local MDP* for robot i is an MDP $\mathcal{M}_i = \langle S_i, \overline{s}_i, A_i, \delta_i, AP, L_i \rangle$ where:

- the state space $S_i = S_i^l \times S_1^g \times ... \times S_k^g$ is the product of the robot's *local* state space S_i^l and the state space S_j^g for k *global* state features;
- actions $A_i = A_i^l \cup A_1^g \cup \cdots \cup A_k^g$ are partitioned into action sets that update the local state and global state features.

Actions from each set update the corresponding part of the state, i.e., for states $s = (s^l, s_1^g, \ldots, s_k^g)$ and $t = (t^l, t_1^g, \ldots, t_k^g)$ in S_i, we require that, if $\delta_{\mathcal{M}_i}(s, a, t) > 0$, then either:

- $a \in A_i^l$ and $s_j^g = t_j^g$ for all j; or

- $a \in A_j^g$, $s^l = t^l$ and $s_{j'}^g = t_{j'}^g$ for all $j' \neq j$.

Remark 7 (Local and Global State Actions). We do not consider actions that can update both the local and global states as this is unlikely to occur in practice. For example, a robot can not move and check if a door is open since it could risk crashing into the door if closed. Moreover, local and global states are always fixed, i.e. a local state can not morph into a global state and vice versa.

Probabilities in the MDP $\mathcal{M}_i$ typically represent either uncertainty regarding the environment or the possibility of failure. Here, we will assume that each robot i has a *designated failure state* $s_i^{fail} \in S_i^l$ which, once entered, cannot be left.

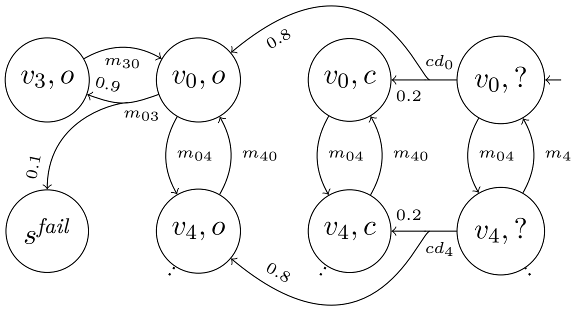


Figure 4.2: Fragment of an example local MDP for robot i corresponding to the map in Figure 4.1, Page 58 (see Example 6).

Example 6. Figure 4.2 shows a fragment of an example local MDP corresponding to the map in Example 5 (see Figure 4.1, Page 58). There is one local state feature, the robot's location, with $S_i^l = \{v_0, \ldots, v_7\}$, and one global state feature, modelling the status of a door (open, closed, unknown) with $S_1^g = \{o, c, ?\}$.

States are of the form (v_j, door) and the initial state is $(v_0, ?)$, where the robot is in location v_0 and the door status is unknown. Actions updating the local state, in the set

A_i^l, are of the form m_{jk} ("move from v_j to v_k"). Actions updating the global state, in the set A_1^g, are of the form cd_j ("check the door status when in v_j"). Some instances of both types of actions exhibit probabilistic behaviour. For example, when checking the door in v_0 (action cd_0), the door is open with probability 0.8; and moving from v_0 to v_3 (action m_{03}) succeeds with probability 0.9, or results in a transition to the failure state with probability 0.1.

4.1.2 Multi-Robot Models

Given MDPs defining the local model for each of the n individual robots, we define the *joint MDP* as an MDP $\mathcal{M}_J$ formed as the product of these. This models the combined execution of the n robots in their environment.

The (joint) state space S_J of the global model includes the local state of each robot i and the state of each global feature j. We start by defining a function that projects joint states onto local robot states.

Definition 16 (Projection Function)**.** Let $S_J = S_1^l \times \cdots \times S_n^l \times S_1^g \times ... \times S_k^g$, and $i \in \{1, \ldots, n\}$. We write $[.]_i : S_J \to S_i$ for the function that projects states of $\mathcal{M}_J$ to the corresponding states of $\mathcal{M}_i$, defined as:

$$[s_1^l, \ldots, s_n^l, s_1^g, \ldots, s_k^g]_i = (s_i^l, s_1^g, \ldots, s_k^g).$$

In each transition of the global model, all robots make transitions simultaneously i.e. the robots move in sync. We therefore require that, in any transition, at most one robot updates each global state feature. This ensures that action outcomes in the joint model are consistent with the single robot models. Since we partition actions according to the parts of the state space that they update, we can enforce this condition simply by constraining the set of joint actions A that are allowed in the global model MDP.

Definition 17 (Joint MDP)**.** Let the model for each robot i be local MDP $\mathcal{M}_i = \langle S_i, \overline{s}_i, A_i, \delta_i, AP, L_i \rangle$, where:

- $S_i = S_i^l \times S_1^g \times \cdots \times S_k^g$;

- $\overline{s}_i = (\overline{s}_i^l, \overline{s}_1^g, \ldots, \overline{s}_k^g)$;

- $A_i = A_i^l \cup A_1^g \cup \cdots \cup A_k^g$;

We assume that the k global state features are the same for all robots (i.e., their state spaces S_j^g, initial values $\overline{s}_j^g$ and action sets A_j^g are the same in each $\mathcal{M}_i$), as is the set AP. The *joint MDP* is an MDP $\mathcal{M}_J = \langle S_J, \overline{s}_J, A_J, \delta_J, AP, L_J \rangle$ where:

- $S_J = S_1^l \times \cdots \times S_n^l \times S_1^g \times ... \times S_k^g$;

- $\overline{s}_J = (\overline{s}_1^l, \ldots, \overline{s}_n^l, \overline{s}_1^g, \ldots, \overline{s}_k^g)$;

- $A_J = \{(a_1, \ldots, a_n) \,|\, a_i \in A_i$ and, for each $1 \leq j \leq k$, we have $a_i \in A_j^g$ for at most one $i\}$.

Then, for states $s, t \in S_J$ and action $a = (a_1, \ldots, a_n) \in A_J$, we define δ_J and L_J as follows:

- $\delta_J(s, a, t) = \prod_{i=1}^n \delta_i([s]_i, a_i, [t]_i)$;

- $L_J(s) = \bigcup_{i=1}^n L_i([s]_i)$.

Our proposed joint MDP model is an instance of a multi-agent MDP [Bou96] (MMDP), where we impose extra structure on local and global state features. Thus, our joint model has similar assumptions as MMDPs, namely that robots have access to the full state space of the team, and that actions are executed in a synchronised fashion, using a common timestep across the robots. We also assume robots are able to navigate around each other efficiently, and do not consider issues related to collisions and obstacle avoidance in the model. Instead, with this model we focus on robustness to (probabilistic) single-robot failures and to uncertainty on the value of global state features, which might yield certain missions achievable by only a subset of the robots.

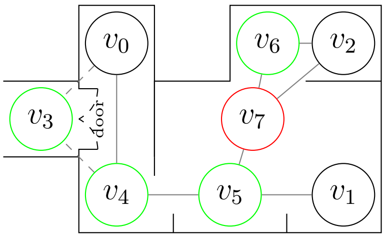


Figure 4.3: A mission specification defined for the example topological map from Page 58, Figure 4.1. Robots start in v_0, v_1, v_2; locations to visit in green and to avoid in red; mission specification $\Phi = \langle \mathtt{F}(v_5 \wedge \mathtt{F}\, v_4), \mathtt{F}\, v_3, \mathtt{F}\, v_6, \mathtt{G}\, \neg v_7 \rangle$.

4.1.3 Mission Specification using LTL

We use a mix of co-safe and safe LTL to specify the robots' mission. A *mission specification* $\Phi = \langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$ consists of a list of co-safe LTL *task specifications* $\varphi_1, ..., \varphi_m$ and a *safety specification* φ_{safe} in safe LTL. We assume the mission to fulfil the *independence* property defined in [SBD18d].

Remark 8. More specifically, independence means that at any given point robots can only contribute to one task. A sufficient condition for this to hold is each task being written over a different set of atomic propositions. As a result, each task is independent of all other tasks and satisfying any task in the mission does not violate any other task in the mission. It also ensures that not satisfying any task in the mission also does not violate any other task in the mission. This means that satisfying all tasks in the mission implies full mission satisfaction. Consequently, it ensures that there is no contention between the tasks and does away with need for any task prioritisation. It allows us to rely on the policy generated by our solution. Unlike [Smi+11; MKK17], we do not monitor task execution since we model the robot-environment interaction as an MDP.

Example 7. We return to the running example. Let v_j be an atomic proposition indicating that a robot is in location v_j. When used to label a state of local MDP $\mathcal{M}_i$, this means that robot i is in v_j. Since the state labelling in the joint MDP takes the union of individual labellings for local MDPs, when v_j labels a state in $\mathcal{M}_J$, it indicates that *some* robot is in v_j.

An example mission specification for this model, shown in Figure 4.3, is $\Phi = \langle \mathtt{F}(v_5 \wedge$

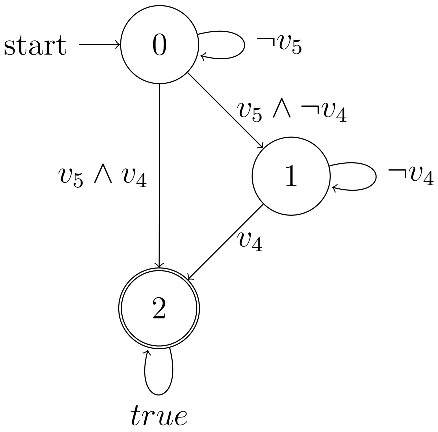


Figure 4.4: The Discrete Finite Automaton for the task $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$. The initial state is 0. The robot stays in this state until it visits v_5. Then it transitions to 1. It stays in this state until it visits v_4. Then it transitions to 2 which is the accepting state (denoted by the double border). The direct transition from 0 to 2 is not possible according to the topological map. This transition is automatically removed during the product construction.

$\mathtt{F}\, v_4), \mathtt{F}\, v_3, \mathtt{F}\, v_6, \mathtt{G}\, \neg v_7\rangle$. Task specification $\varphi_1 = \mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$ requires first v_5 then v_4 to be visited; the other two tasks are to visit v_3 and v_6, respectively. The safety specification $\varphi_{safe} = \mathtt{G}\, \neg v_7$ states that location v_7 must be avoided. Figure 4.4 shows the DFA for the specification $\varphi_1 = \mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$.

4.2 Problem Statement

Given a set of n robots and a mission specification $\Phi = \langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$, our aim is to derive a joint policy for the robots which allows them to *collectively* achieve the tasks $\varphi_1, ..., \varphi_m$ without violating the safety constraint φ_{safe}. This incorporates both the allocation of tasks to robots, and the planning for each robot to achieve its tasks. We also aim to produce *probabilistic guarantees* for these policies which precisely quantify their effectiveness or reliability.

Formally, if the behaviour of each robot i is defined by local MDP $\mathcal{M}_i$, then our goal becomes to synthesise an appropriate *joint policy*, i.e., a policy π_J for the joint MDP $\mathcal{M}_J$. One possibility would be to find a policy that maximises the probability $Pr^{\pi_J}_{\mathcal{M}_J}(\varphi_1 \wedge \cdots \wedge \varphi_m \wedge \varphi_{safe})$ of satisfying *all* LTL formulas in Φ. However, in some scenarios, one or more tasks may become unachievable (for example, if a door is closed, then some locations may become inaccessible for all robots) reducing this probability to zero.

So, instead, we target policies that *maximise the expected number of tasks φ_i completed without violating the safety constraint φ_{safe}*. To formalise this, we use the LTL specifications in two distinct ways. The (co-safe) LTL formulas φ_i are used to construct a reward structure that counts the number of tasks that are completed; and we consider the expected amount of this reward accumulated until the negation of the (safe) LTL formula φ_{safe} is satisfied. For both, we use the product construction described in Section 2.3.

Let $\mathcal{M}'_J = \mathcal{M}_J \otimes \mathcal{A}_{\varphi_1} \otimes \cdots \otimes \mathcal{A}_{\varphi_m}$ be the product of the joint MDP $\mathcal{M}_J$ with DFAs for the m formulas φ_i and $\mathcal{M}^{\Phi}_J = \mathcal{M}'_J \otimes \mathcal{A}_{\neg\varphi_{safe}}$ be the product of $\mathcal{M}_J$ with DFAs for all $m+1$ formulas in the mission specification Φ. We fix a reward structure *tasks* for $\mathcal{M}'_J$ that counts the number of tasks that are completed, i.e., which assigns to each transition $s \xrightarrow{a} s'$ the number of tasks φ_i for which it is a transition into an accepting state for $\mathcal{A}_{\varphi_i}$. Then, our goal is to compute the expected cumulative value of *tasks* up until a point where $\neg\varphi_{safe}$ becomes true, which reduces to computing the expected cumulative reward until reaching accepting states for $\mathcal{A}_{\neg\varphi_{safe}}$ in the product model $\mathcal{M}^{\Phi}_J$:

$$E^{\max}_{\mathcal{M}'_J}(cumul^{\neg\varphi_{safe}}_{tasks}) \;=\; E^{\max}_{\mathcal{M}^{\Phi}_J}(cumul^{acc_{\neg safe}}_{tasks})$$

where, by slight abuse of notation, we use *tasks* to refer to the reward structures for both models.

Note that the cumulative value of *tasks* is always finite, because we consider a finite number of co-safe tasks. Thus, in order to compute $E^{\max}_{\mathcal{M}^{\Phi}_J}(cumul^{acc_{\neg safe}}_{tasks})$, we can make states in $acc_{\neg safe}$ absorbing and compute the total expected reward in the resulting model. In practice, this means that policies will be defined until we reach a state where $\neg\varphi_{safe}$ becomes true (i.e. a state where the safety specification has been broken); or the expected cumulative value of the *tasks* reward structure becomes zero (i.e., a state from where the team cannot achieve more tasks). While task reward maximisation is our main objective, we also incorporate methods to discourage robots from staying idle in order to minimise the number of steps taken to achieve the mission. These methods are specific to the kind of solution approach used and are therefore explained in the chapters corresponding to each solution approach.

Note that the basis of our joint team policy is the joint MMDP (see Section 4.1.2) and therefore we assume that robots are able to communicate with each other at each step and that they move in lock step.

4.3 Implementation

This section describes the tools used to implement and test all the solution methods presented in this thesis.

4.3.1 The PRISM Model Checker

All algorithms presented in this thesis are built on top of the probabilistic model checking tool PRISM [KNP11]. This provides construction of MDPs, from a high-level modelling language, and verification of the MDPs against specifications in LTL, which includes functionality required to implement STAPU such as generating DFAs, building MDP-DFA products, and solving MDPs using a variety of techniques. The implementation builds on PRISM's "explicit" model checking engine, which is written in Java.

4.3.2 Platform used

All experiments described in this thesis were run on an Intel i5 with 16GB of RAM running Linux.

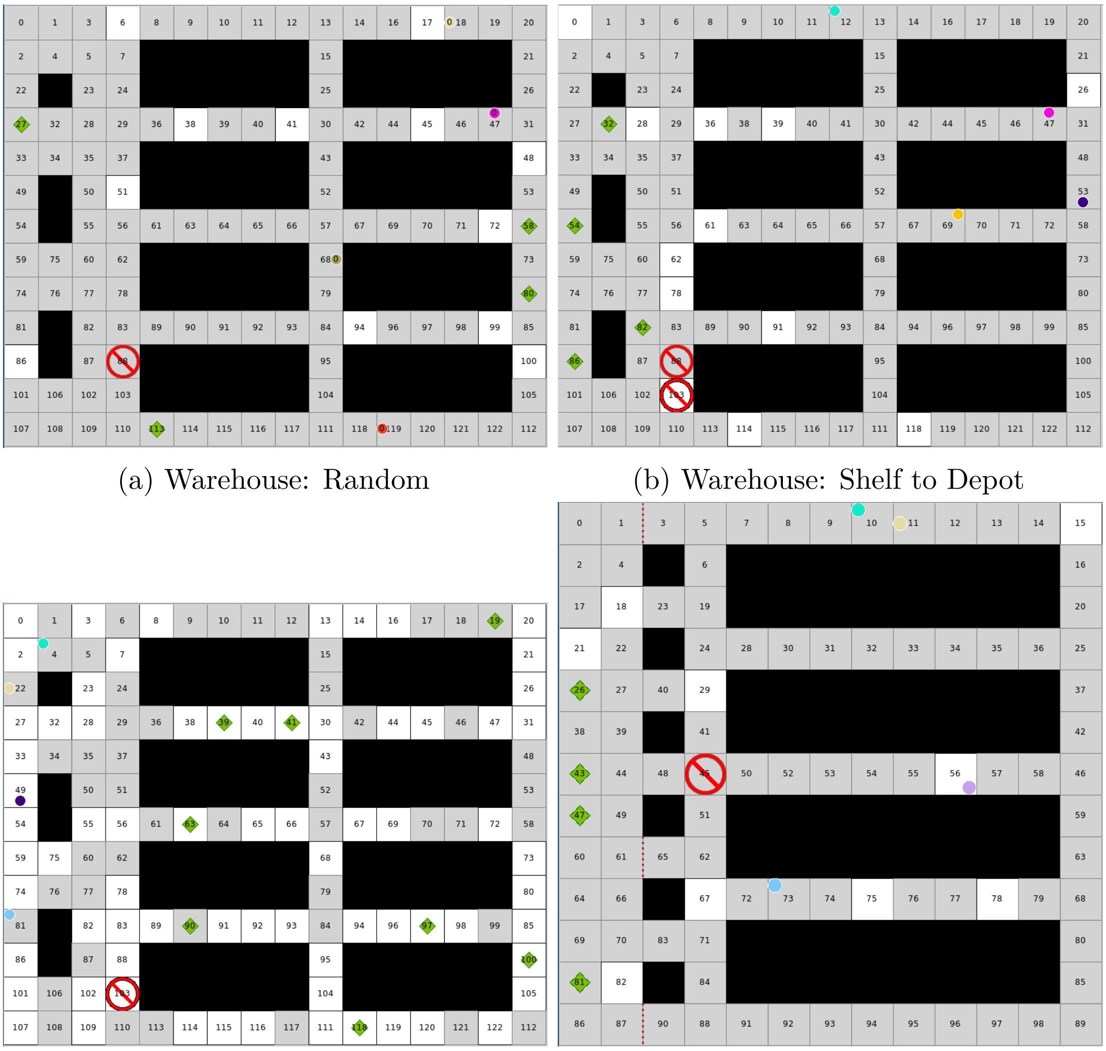
4.3.3 Test Environments

Our test environments are inspired by the benchmarks already used for multi-agent path finding (MAPF) [Ste+19; Geb+18] and multi-agent sequential decision making under uncertainty (MSDM) [Spa]. The need for developing our own benchmarks arose due to the following reasons: 1. MAPF does not consider uncertainty, 2. MSDM is focused on POMDPs, 3. none of these benchmarks considered robot failure, and 4. converting all benchmarks to PRISM models was not trivial.

Our test environments[1] consist of a warehouse, a fully connected grid and an office. We use 3 variations of the warehouse environment: one where robots travel from the shelves to the depot, one where robots travel from the depot to the shelves and one where the initial and task locations are random. Similarly, we use 3 variations of the grid environment: one where robots travel left to right, one where robots travel right to left and one where initial and task locations are random. In order to test the use of global state features, we use variants of the warehouse environment with and without doors. The warehouse has 123 locations, and 100 locations for the version with doors. The office model is slightly smaller, with 60 locations. For most of our experiments, we fix the size of the grid environment to 11×11, but we also vary this size to evaluate scalability with regard to locations.

Missions comprise *tasks to visit various target locations and the safety specification is to avoid a set of locations*. By default, we assume 4 robots and 4 tasks, but we also present results for varying numbers of both. We model individual robot failures by introducing transitions to the designated failure state from some of a robot's local states with a fixed probability (we use 0.2 in our experiments). By default, we assume that most states (90%) can suffer failures, we also experiment with varying this percentage. We use the term *failstates* to refer to locations where robot actions can lead to the designated failure state.

[1] All tests are located in the github repository: https://github.com/fatmaf/generatedTests



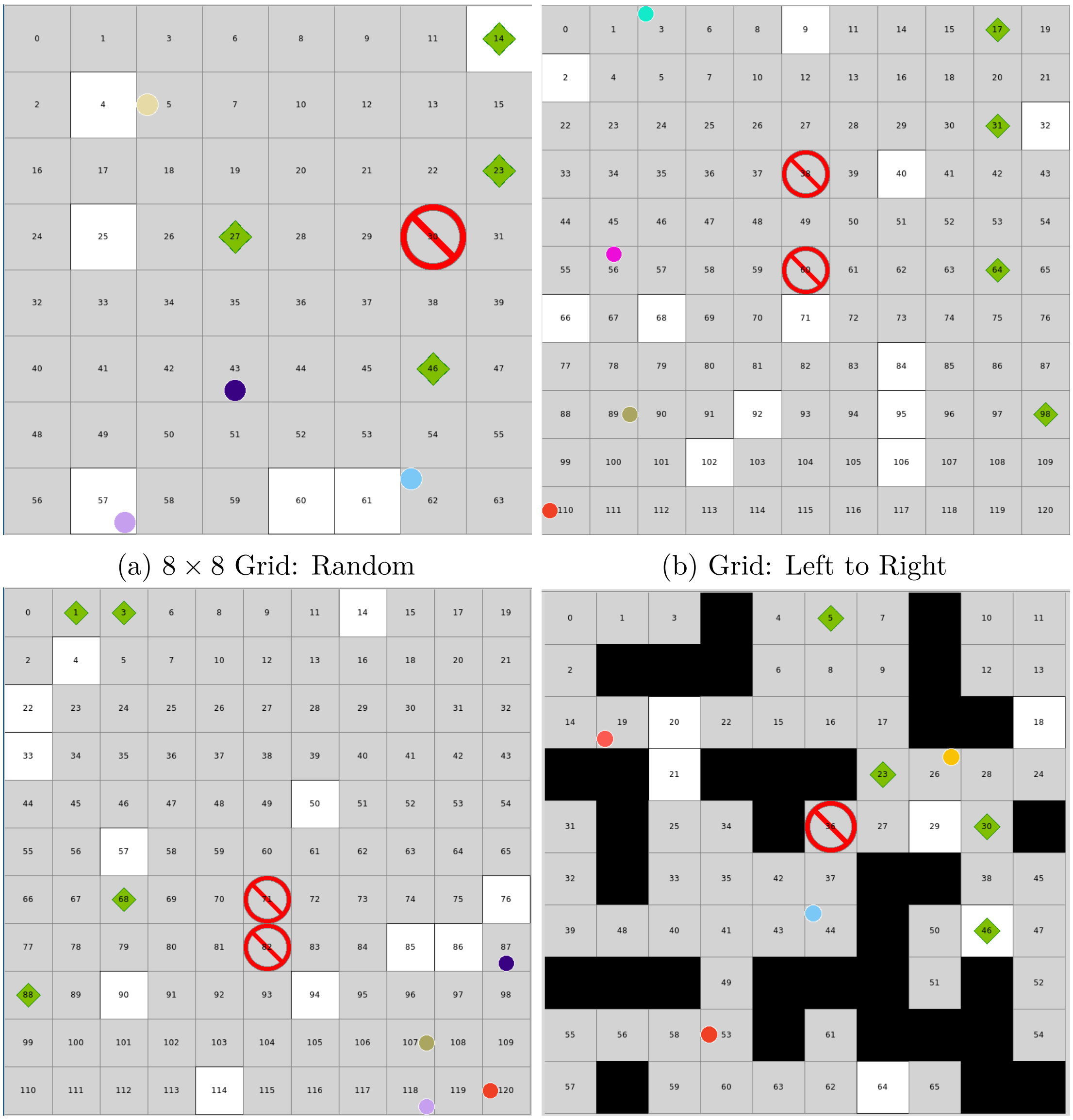
(a) Warehouse: Random

(b) Warehouse: Shelf to Depot

(c) Warehouse: Depot to Shelf

(d) Small Warehouse With 3 Doors

Figure 4.5: Instances of the topological maps used as test environments for benchmarking. Initial locations of robots (4 in these examples) are denoted by coloured circles, task locations are marked by green diamonds and locations to be avoided for the safety specification are shown by a red stop sign. Failure locations (40% of locations in c; 90% elsewhere) are shaded grey. Doors, as used in d, are represented as a dashed brown line between two states.



(a) 8 × 8 Grid: Random

(b) Grid: Left to Right

(c) Grid: Right to Left

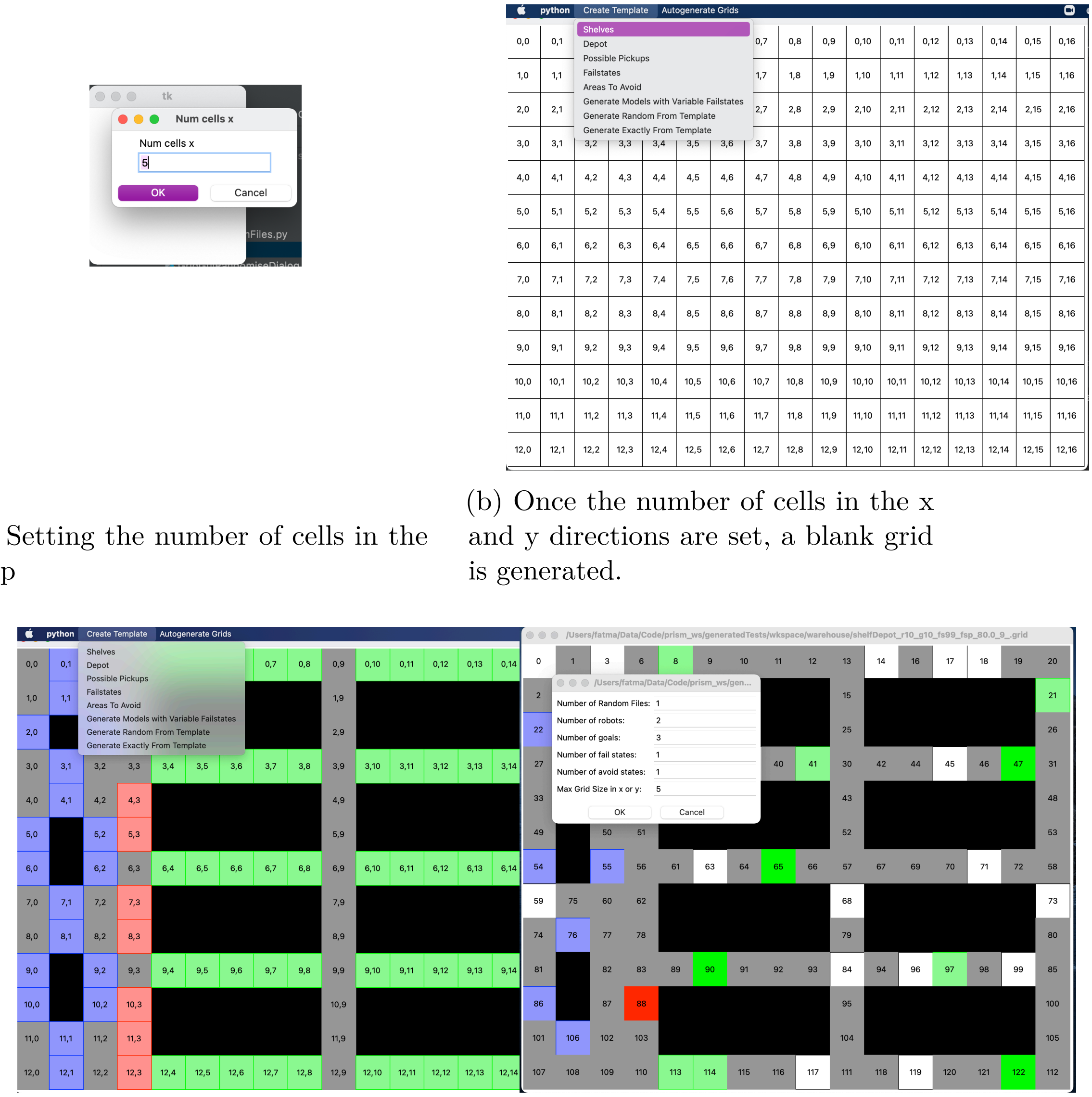
(d) Office

Figure 4.6: Instances of the topological maps used as test environments for benchmarking. Initial locations of robots (4 in these examples) are denoted by coloured circles, task locations are marked by green diamonds and locations to be avoided for the safety specification are shown by a red stop sign. Failure locations (90%) are shaded grey.

4.3.3.1 Generating test environments

The test environments are generated as PRISM models using a *python* script with a *Tkinter* graphical user interface (GUI)[2]. The GUI can be used to specify a map template as in Figure 4.7. The template is then used to generate PRISM models. The GUI can also be used to specify generating grids of varying sizes. Another instance of the GUI allows adding doors to the maps. For each test configuration, we create 10 different variants, in which the initial robot locations, locations for tasks and failstates are chosen randomly. We illustrate a selection of the maps used in Figures 4.5 and 4.6.

[2] https://github.com/fatmaf/generatedTests



Num cells x: 5

(a) Setting the number of cells in the map

0,0	0,1						0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15	0,16
1,0	1,1						1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16
2,0	2,1						2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	7,10	7,11	7,12	7,13	7,14	7,15	7,16
8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	8,10	8,11	8,12	8,13	8,14	8,15	8,16
9,0	9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11	9,12	9,13	9,14	9,15	9,16
10,0	10,1	10,2	10,3	10,4	10,5	10,6	10,7	10,8	10,9	10,10	10,11	10,12	10,13	10,14	10,15	10,16
11,0	11,1	11,2	11,3	11,4	11,5	11,6	11,7	11,8	11,9	11,10	11,11	11,12	11,13	11,14	11,15	11,16
12,0	12,1	12,2	12,3	12,4	12,5	12,6	12,7	12,8	12,9	12,10	12,11	12,12	12,13	12,14	12,15	12,16

(b) Once the number of cells in the x and y directions are set, a blank grid is generated.

0,0	0,1						0,7	0,8	0,9	0,10	0,11	0,12	0,13	0,14
1,0	1,1								1,9					
2,0									2,9					
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14
4,0	4,1	4,2	4,3						4,9					
5,0		5,2	5,3						5,9					
6,0		6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14
7,0	7,1	7,2	7,3						7,9					
8,0	8,1	8,2	8,3						8,9					
9,0		9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11	9,12	9,13	9,14
10,0		10,2	10,3						10,9					
11,0	11,1	11,2	11,3						11,9					
12,0	12,1	12,2	12,3	12,4	12,5	12,6	12,7	12,8	12,9	12,10	12,11	12,12	12,13	12,14

(c) This template can be populated with choices for the possible locations of shelves, depots, failstates, and locations to avoid.

/Users/fatma/Data/Code/prism_ws/generatedTests/wkspace/warehouse/shelfDepot_r10_g10_fs99_fsp_80.0_9_.grid

/Users/fatma/Data/Code/prism_ws/gen...

Number of Random Files: 1
Number of robots: 2
Number of goals: 3
Number of fail states: 1
Number of avoid states: 1
Max Grid Size in x or y: 5

0	1	3	6	8	9	10	11	12	13	14	16	17	18	19	20
2									15						21
22									25						26
27							40	41	30	42	44	45	46	47	31
33									43						48
49		50	51						52						53
54		55	56	61	63	64	65	66	57	67	69	70	71	72	58
59	75	60	62						68						73
74	76	77	78						79						80
81		82	83	89	90	91	92	93	84	94	96	97	98	99	85
86		87	88						95						100
101	106	102	103						104						105
107	108	109	110	113	114	115	116	117	111	118	119	120	121	122	112

(d) The number of robots, goals, failstates etc can then be set in the menu. Finally, the template can then be used to generate PRISM models with initial locations selected from depot locations and goal locations selected from shelf locations. If no template is present, a fully connected grid can be generated automatically by setting the grid size and other variables.

Figure 4.7: Screenshots of the python GUI used to specify template environments and generate corresponding PRISM models.

4.4 Naïve Solution Method: Value Iteration (VI)

As per our problem definition, all robots and DFAs have a fixed initial state. Our objective is to find a policy to absorbing states which maximises the expected cumulative reward.

Remark 9. For a general overview of MDP solution methods from the perspective of the artificial intelligence planning community in computer science we refer the reader to [MK12]. Some of these such as value iteration, policy iteration etc are also summarised in Section 3.4.1. Techniques presented in [MK12] can be combined with automata generated from LTL to give solution methods for MDPs with tasks specifications in LTL. In fact, [KP13], provides an overview of such techniques for automated verification and strategy synthesis. It describes a solution to MDPs with reward structures and LTL specifications as in the PRISM Model Checker [KNP11]. The following text describes this solution in the context of our problem.

A naive approach to obtaining an optimal policy for $\mathcal{M}_J^{\Phi}$ is value iteration (VI) [Bel57]. As the name suggests the core of the algorithm involves iteratively refining the values of states (Definition 5) till the optimal value is reached. The optimal value of all absorbing states is set to 0 since no further reward can be accumulated once such a state is reached. The optimal value of all non-absorbing states is calculated using the *Q-value* of a state action pair.

Definition 18 (Q-value under a value function)**.** The Q-value of a state-action pair (s, a) under a value function V is the one-step lookahead computation of the value of taking a in s under the belief that V is the true expected cost to reach an absorbing state[MK12]:

$$Q^V(s,a) = \sum_{s' \in S} \delta(s,a,s')[r(s,a,s') + V(s')]$$

The optimal value of all non-absorbing states is then

$$V^*(s) = \begin{cases} 0, & \text{if } s \in acc_{\neg safe} \\ \\ \underset{a \in A_J^\Phi}{\arg\max}\; Q^*(s,a), & \text{otherwise} \end{cases} \tag{4.1}$$

$$Q^*(s,a) = \sum_{s' \in S} \delta(s,a,s')[r(s,a,s') + V^*(s')] \tag{4.2}$$

Equations (4.1) and (4.2) are commonly referred to as Bellman Equations.

VI begins with an arbitrary estimate of V^*, V_0 for all states. This estimate is then refined over $n = 1, ..., N$ iterations using the Bellman Equations. Each successive estimate V_n uses values from the previous estimate V_{n-1}:

$$V_n(s) \leftarrow \underset{a \in A_J^\Phi}{\arg\max} \sum_{s' \in S} \delta(s,a,s')[r(s,a,s') + V_{n-1}(s')] \tag{4.3}$$

Equation Equation (4.3) is known as the Bellman update or Bellman backup. As N approaches infinity the value estimate converges to the optimal value, V^* [MK12]. In practice, the convergence criteria for VI is defined using the *residual* of the value function.

$$Res^V(s) = |V(s) - \underset{a \in A_J^\Phi}{\arg\max} \sum_{s' \in S} \delta(s,a,s')[r(s,a,s') + V(s')]| \tag{4.4}$$

Assuming ϵ is the acceptable deviation from the optimal value for each state, a state s is *ϵ-consistent* if $Res^V(s) < \epsilon$. VI terminates when $Res^V < \epsilon$, i.e. all states are ϵ-consistent.

Definition 19. A state $s \in S$ is ϵ-consistent if its residual is less than a given error threshold ϵ i.e. $Res^V(s) < \epsilon$.

Practical implementations of VI limit the maximum number of iterations to avoid VI running forever if it does not converge i.e. if $Res^V \not< \epsilon$.

The policy under the value function returns the action which has the maximum Q-value of all enabled actions in a particular state:

$$\pi^V(s) = \arg\max_{a \in A} Q(s, a) \tag{4.5}$$

This is called the *greedy policy* since it greedily maximises the Q-value for each state.

Algorithm 1 Value Iteration

```
function ValueIteration(MDP $\mathcal{M}$, RewardFunction $r$, $\epsilon$, $N$)
    for all $s \in S$ do
        $V(s) \leftarrow 0$
    end for
    $n = 0$
    while $Res^V > \epsilon$ & $n < N$ do
        for all $s \in S$ do
            Update $V(s)$                                   ▷ using Equation (4.3)
        end for
    end while
    return $\pi^V$
end function
```

The algorithm for Value Iteration from [MK12]

In the case of rewards, VI relies on the assumption that all rewards are positive. Since the objective is to improve the value function, this ensures that the value function improves monotonically. The dual of reward maximisation is cost minimisation; the costs remain positive, but the objective changes to minimising the value function, instead of maximising it. A more detailed treatment of VI and other methods can be found in [MK12].

The Bellman backup for a single state considers all enabled actions in the state and all successor states. Therefore the worst-case runtime of a single Bellman backup is $O(|S||A|)$ time. Each iteration of VI requires $|S|$ Bellman backups, covering all states in the MDP. The runtime of each iteration of VI in the worst-case is then $O(|S|^2|A|)$.

The state and action space of the joint MDP increase exponentially with the number of robots. The state and action spaces are also affected by the number of tasks, since each task is a DFA. This means that as the number of tasks increases, the state and action spaces increase as well. Lastly, as the number of actions that may lead to the designated failure state in the robot model increases, the number of transitions in the MDP increase.

In MDPs where the initial state is given, we are concerned with a policy that starts in this state. Therefore, any states that can not be reached by taking a series of actions from the initial state, can be ignored. VI can then be performed on the states that can be reached starting in the initial state. In a fully connected environment, where all states are reachable from all other states, this modification will not provide any additional gains.

4.4.1 Nested Value Iteration

We demonstrate the scalability of VI on a 4-connected 5 × 5 grid. While our main objective is to maximise the expected number of tasks, we use cost as a tie breaker. Recall from Proposition 2 that $E^{\max}_{\mathcal{M}}(cumul^{\varphi}_r)$ is the maximum expected cumulative reward. Let $E^{\min}_{\mathcal{M}}(cumul^{\varphi}_c)$ be the minimum expected cumulative cost. Let Π be the set of all policies for the MDP $\mathcal{M}$. Let Π^* be the set of all policies that maximise the expected task reward:

$$\Pi^* = \{\pi \in \Pi \mid \pi = \arg\max_{\pi'} E^{\pi'}_{\mathcal{M}}(cumul^{\varphi}_r)\} \tag{4.6}$$

. The updated objective is to find a policy π that maximises the expected task reward using expected cost as a tie breaker i.e. when the task reward for two states is the same, the one with the lower cost is picked:

$$\pi^* = \arg\min_{\pi \in \Pi^*} E^{\pi}_{\mathcal{M}}(cumul^{\varphi}_c) \tag{4.7}$$

Equation (4.7) can be solved using nested value iteration (NVI) [LPH15a], a generalised version of VI which considers multiple objectives in lexicographic order. NVI was originally presented in [LPH15a] with the following three objectives in order of priority:1. maximisation of probability of mission satisfaction 2. maximisation of expected progression reward (see Section 3.4.1) 3. minimisation of expected action costs. In Algorithm 2 we present a generalised version of NVI. Our implementation of Algorithm 2[3] includes statements to check for reward maximisation or cost minimisation. We implemented NVI, building on

[3] Our implementation of NVI can be found here: https://github.com/fatmaf/prism/tree/arm64

existing PRISM code from [LPH15a].

Algorithm 2 Nested Value Iteration

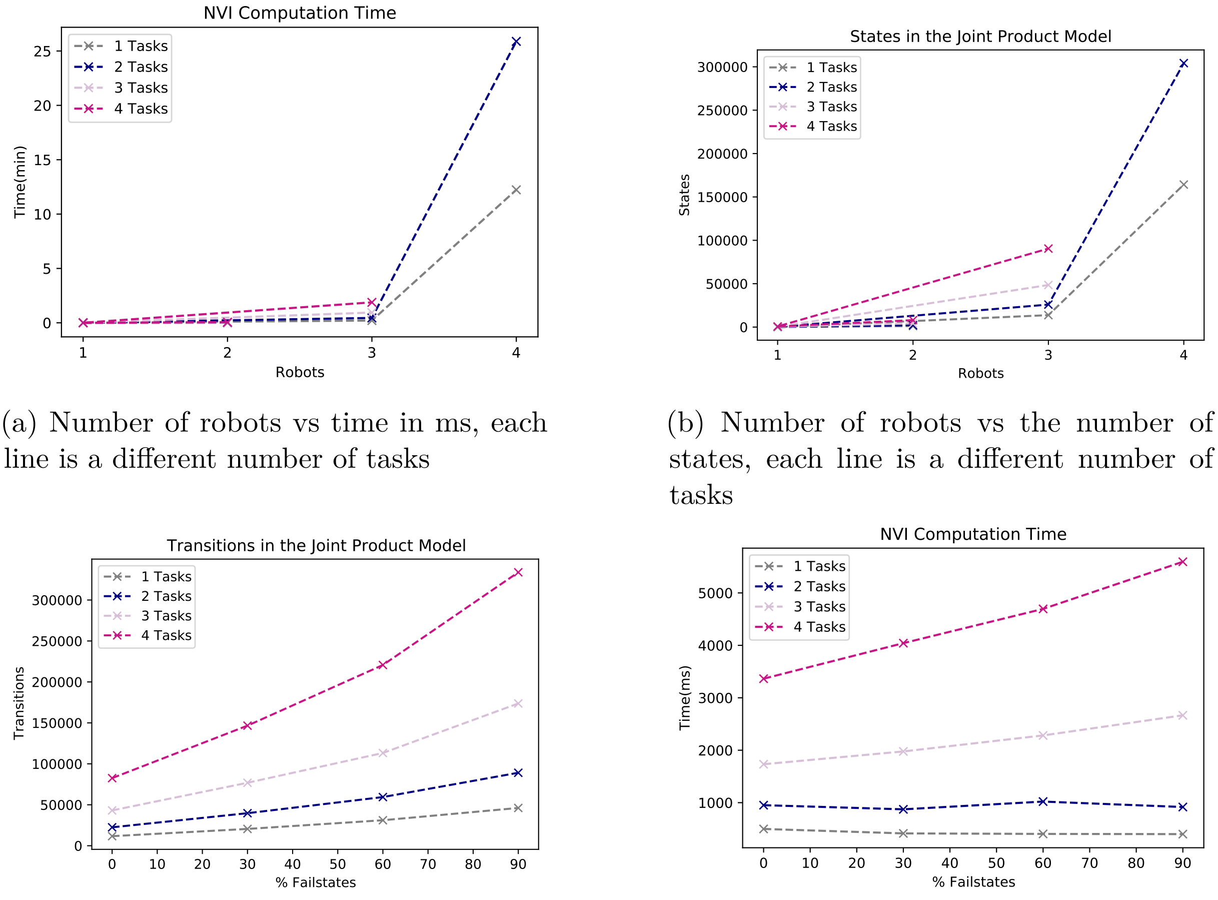
```
function NestedValueIteration(MDP $\mathcal{M}$, A list of reward functions $R$, $\epsilon$, $N$)
    Let $V = \{V^0, V^1, \ldots, V^{|R|}\}$
    for all $s \in S, r \in R$ do
        $V^r(s) \leftarrow 0$
    end for
    $n = 0$
    while $ResidualV(V, \epsilon)$ & $n < N$ do
        for all $s \in S$ do
            Update($V(s)$)
        end for
    end while
    return $\pi^V$
end function

function Update($V(s)$)
    doUpdate← false
    for $i \in \{0, \ldots, |V(s)|\}$ do
        if $Res^{V^i}(s) > \epsilon$ then                    ▷ using Equation (4.4)
            doUpdate← true
            break
        end if
    end for
    if doUpdate then
        for $i \in \{0, \ldots, |V(s)|\}$ do
            Update $V^i(s)$                                  ▷ using Equation (4.3)
        end for
    end if
end function

function ResidualV($V, \epsilon$)
    for $s \in S$ do
        for $i \in \{0, \ldots, |V(s)|\}$ do
            if $Res^{V^i}(s) > \epsilon$ then                ▷ using Equation (4.4)
                return $false$
            end if
        end for
    end for
    return $true$
end function
```

A generalised version of the algorithm for Nested Value Iteration. The original algorithm in [LPH15a] was specific to the objectives being considered in the paper.

For simplicity, Algorithm 2 uses an *ordered* list of reward functions. This means that the primary objective is the first reward function and all subsequent reward functions are used as tie-breakers for their predecessors. In Line 7 the ResidualV function is used to check if *any* of the states are ϵ-inconsistent. If so each state is updated in Line 9 using the Update function which updates a state. The Update function only updates a state if the value functions can be improved in order. The list of reward functions can be replaced by a list of reward and cost functions or a list of cost functions. The value update (Line 24) needs to be adjusted accordingly i.e. for cost functions, state value updates are performed if the new cost is lower.



(a) Number of robots vs time in ms, each line is a different number of tasks

(b) Number of robots vs the number of states, each line is a different number of tasks

(c) Effect of uncertainty on robot actions with the number of robots fixed to 2.

(d) Effect of uncertainty on computation time with the number of robots fixed to 2.

Figure 4.8: The scalability of nested value iteration

4.4.2 Results

For our experiments we vary the number of robots and tasks in the mission specification. Each task is a vistation task of the form $\mathtt{F}\, v_i$ where v_i denotes location i of the map. The mission consists of a series of such tasks and a safety task of the form $\mathtt{G}(\neg v_j)$. An example mission could be $\mathtt{F}\, v_{27}, \mathtt{F}\, v_{46}, \mathtt{G}(\neg v_{30})$.

Robots	Tasks	States	Actions	Time (ms)
2	1	1144	11680	499
2	2	2200	22484	952
2	3	4200	43120	1733
2	4	8000	82544	3364
1	1	96	306	352
1	2	188	600	93
1	3	368	1176	136
1	4	720	2304	185
3	1	13754	447760	12605
3	2	25922	845732	26780
3	3	48492	1591156	56487
3	4	90480	2987240	112843
4	1	164268	17132800	733925
4	2	304236	31792052	1553165
4	3	Out of	Memory	Error

(a) Increase in state-action space with increase in robots and tasks. These models were fully deterministic.

Tasks	% Failstates	States	Actions	Transitions	Time (ms)
1	30	1346	12320	20516	414
2	30	2592	24008	39736	873
3	30	4984	46120	76880	1977
4	30	9568	87568	146640	4042
1	60	1346	12320	31118	404
2	60	2584	23692	59428	1021
3	60	4952	45488	113240	2282
4	60	9536	88672	220672	4695
1	90	1346	12636	46132	400
2	90	2584	24312	89104	917
3	90	4976	47408	173696	2666
4	90	9568	91168	333856	5594

(b) Increase in transitions and computation time with increase in uncertainty. % Failstates refers to the percentage of locations in the topological map that led to the designated failure state.

Table 4.1: The scalability of nested value iteration on the Full Joint Product MDP

Table 4.1a shows the size of the state space as the number of robots and tasks grows. At 4 robots and 3 tasks, there is an out of memory error since the size of the full joint product model is too large. Figure 4.8a shows the time taken to compute the optimal policy using NVI. Figure 4.8b shows the number of states in the resulting joint product multi-robot MDP (product MMDP) as the number of robots and tasks increases. Since NVI considers all reachable states in the product MMDP, the number of states, transitions and actions increases with the number of robots and tasks, resulting in too much memory being used. The size of the product MMDP impacts the time taken to compute a solution since more states and transitions need to be considered.

Figure 4.8c shows the increase in the number of transitions as the number of locations leading to the designated failure state increases. We refer to these locations as *failstates*. From Table 4.1b we can see that even though the number of actions remains the same the computation time increases as the number of transitions grows.

Therefore, in practice, it may not be feasible to find the *optimal* policy for the objective above, but for any policy that we do synthesise, we will also compute the actual expected number of tasks completed without violating the safety constraint, which represents a *probabilistic guarantee* on its performance. We can also compute separate guarantees, for example, the probability with which a particular task φ_i is completed or with which the safety specification φ_{safe} holds.

4.5 Solution Framework

Motivated by the poor scalability of VI, the upcoming chapters of this thesis will look at three different ways of solving the problem in Section 4.2. Each of these relies on MDPs for robot models and LTL for task specification and generating an automatic task reward. We also use the DFAs corresponding to the LTL tasks to track task allocation and task completion among the robots.

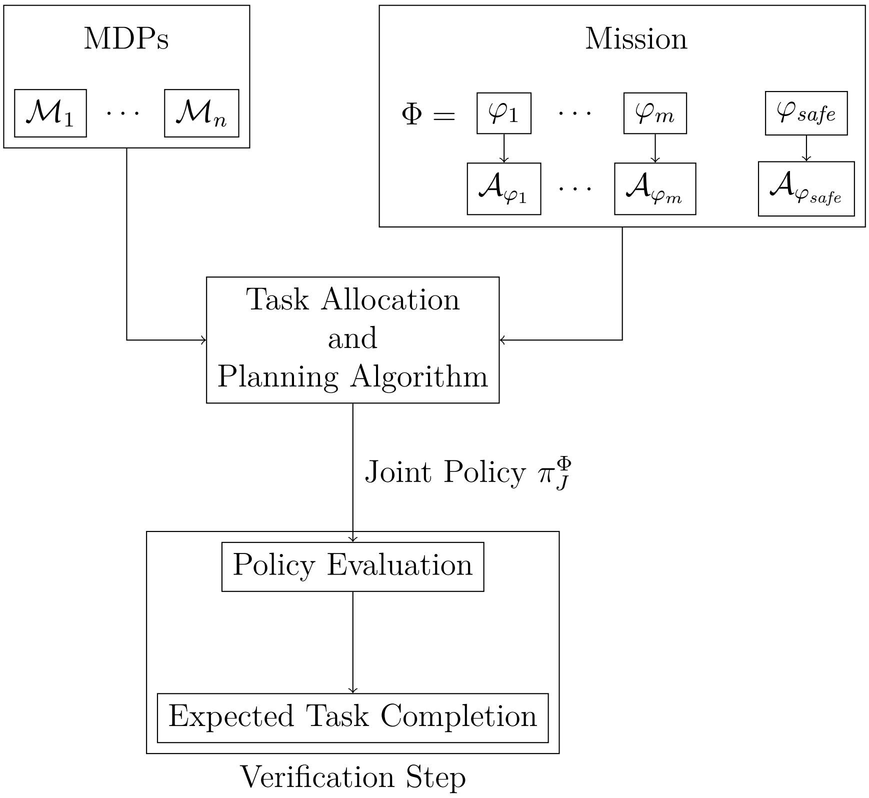

MDPs
$\mathcal{M}_1$ $\cdots$ $\mathcal{M}_n$
Mission
$\Phi =$ φ_1 $\cdots$ φ_m φ_{safe}
$\mathcal{A}_{\varphi_1}$ $\cdots$ $\mathcal{A}_{\varphi_m}$ $\mathcal{A}_{\varphi_{safe}}$
Task Allocation and Planning Algorithm
Joint Policy π_J^Φ
Policy Evaluation
Expected Task Completion
Verification Step

Figure 4.9: Overall architecture of our solution approaches.

Figure 4.9 provides an overview of our solution methodology with the MDPs and DFAs as inputs to each algorithm. Each algorithm outputs a policy π_J^Φ on the joint product MMDP. For each approach we employ a separate verification step where we use the joint policy to provide a guarantee on the expected task completion. This is possible because all methods operate on a product MDP allowing us to automatically verify properties of the MDP using value iteration [BHK19; KP13]. In practice, we perform value iteration on the joint policy which reduces to policy evaluation (see Definition 5) [MK12].

While our work focuses on partial satisfaction through the task reward, exact guarantees on other properties of the policy such as reachability probability etc can also be generated. This can be done by swapping out the expected task completion property for the property under consideration, in the verification step.

All three approaches presented in the following chapters are implemented using the same implementation tools which we described in the previous section. They are all also tested on the same test sets.

Remark 10 (Replanning). Note that using the full joint product for planning means that if a robots fails, other robots can take the out-of-commission robot's tasks. This type of task reallocation may not be possible without the full joint product. Therefore, an important aspect of our approaches, is to consider **task reallocation** when robots in the team **fail**. For some of the algorithms under consideration, this might involve **replanning** from the states where one of the robots fails. Replanning implies generating a new plan (using the same or different algorithm) possibly with the initial state of the MDP modified to reflect the team's state. Replanning is used in many planning applications e.g. [LL18; Cas+19; FKS06].

Chapter 5

Sampling-Based Heuristic Search

As described in Section 4.2 our objective is to find a joint policy for n robots such that they perform all m tasks in the mission specification $\langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$ without violating the safety constraint. Moreover, if it is not possible to achieve *all* tasks in the mission (due to robot failure), our objective is to *achieve as many of the tasks as possible* i.e. partial satisfaction of the mission. In the case of robot failure, this means *reallocating the failed robot's tasks to other robots.* As shown in Section 4.4 a naive solution to this problem can be obtained using value iteration (VI) on the joint multi-robot product MDP,$\mathcal{M}_J$. The size of the joint MDP product, $\mathcal{M}_J$ is exponential in the number of robots and the number of LTL formulae in the mission specification. The increase in size due to the number of robots is unavoidable. The increase in the product size due to the mission specification is also unavoidable as the number of tasks grows. Section 4.4 showed that exact methods, such as VI, are able to generate the optimal policy. However, these methods scale poorly with an increase in the state-action space. This is because they operate on the full joint state-action space which grows exponentially with the number of robots.

In most scenarios, a policy to absorbing states from the initial state, involves a limited set of state-action pairs. Not all the states that can be reached from the initial state are part of this policy. Heuristic search based solutions to MDPs take advantage of this [Brá+14; Ash+18; SHB16; Kol+11]. They use a heuristic function to guide the search.

Definition 20 (Heuristic Function)**.** A heuristic function is a value function $h : S \rightarrow \mathbb{R}_+$ where $h(s)$ is an estimate of the value of state s [MK12].

The heuristic can be used as an initial estimate of the value function. This estimate can then be iteratively updated such that it eventually reaches the optimal value if certain conditions are met. We elaborate on this later in this chapter.

In this chapter, we illustrate the application of a sampling-based heuristic search algorithm to solve the problem described in Section 4.2. First, we present the generalised framework for heuristic search algorithms, Trial Based Tree Search (THTS) [KH13]. Then we elaborate on the aforementioned sampling-based heuristic search algorithm, Labelled Real Time Dynamic Programming (LRTDP) [BG03] under this framework. Recall from Section 3.4.1.3 that LRTDP is an extension of RTDP and can be implemented using the THTS framework. After this background material, we discuss the challenges of applying LRTDP to solve multi-robot planning problems with the possibility of failure and the objective of task reward maximisation. We then show how to adapt the objective of the problem to facilitate the use of LRTDP as a solution method. We modify the exploration strategy in LRTDP incorporating solutions from single-robot planning problems to guide the search in the joint multi-robot model space. Finally, we conclude with results and a discussion on the feasibility of using LRTDP as a solution method to the aforementioned problem.

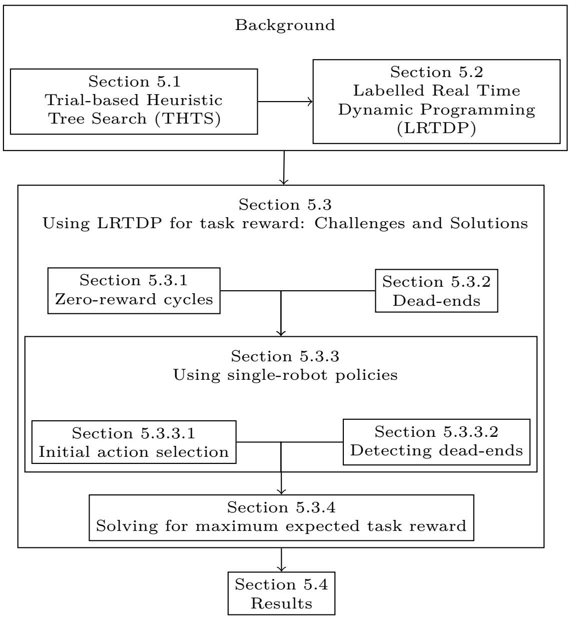


Figure 5.1: An overview of this chapter which begins with some background focused on using Labelled Real Time Dynamic Programming (LRTDP) within the Trial-based Tree Search (THTS) framework. This is followed by a discussion of the challenges of applying LRTDP to the problem formulated in Section 4.2 namely, zero-reward cycles and dead-ends. Existing solutions are utilised for both. Next, the use of single-robot policies to guide the search through initial action selection i.e. a *rollout* policy is explained. It is also shown how to detect deadends using these policies. Finally the solution to task allocation and planning problem is proposed by introducing a novel cost function relative to the number of tasks achieved by the team. The chapter ends with a discussion of the results of applying LRTDP to tests first described in Section 4.3

5.1 Trial-Based Heuristic Tree Search

The Trial-based Heuristic Tree Search (THTS) [KH13] framework presents a unified approach to implementing various heuristic search based algorithms such as Monte-Carlo Tree Search (MCTS) [Bro+12], LAO* [HZ01] and Real Time Dynamic Programming (RTDP) [BBS95]. Some of these have been discussed in Chapter 3 as solution methods to generate verified plans [Brá+14].

The name tree search indicates that the framework supports acyclic graphs or trees. Indeed, in [KH13], the authors convert each MDP to a tree by attaching a step to each state. However, as the authors state, the framework applies to cyclic graphs as well. The use of cyclic graphs avoids duplicate searches if the same state can be reached along different paths[KH13].

This section describes THTS for cyclic graphs, particularly MDPs. It begins with a brief overview of the framework, followed by a more detailed introduction to the framework's components.

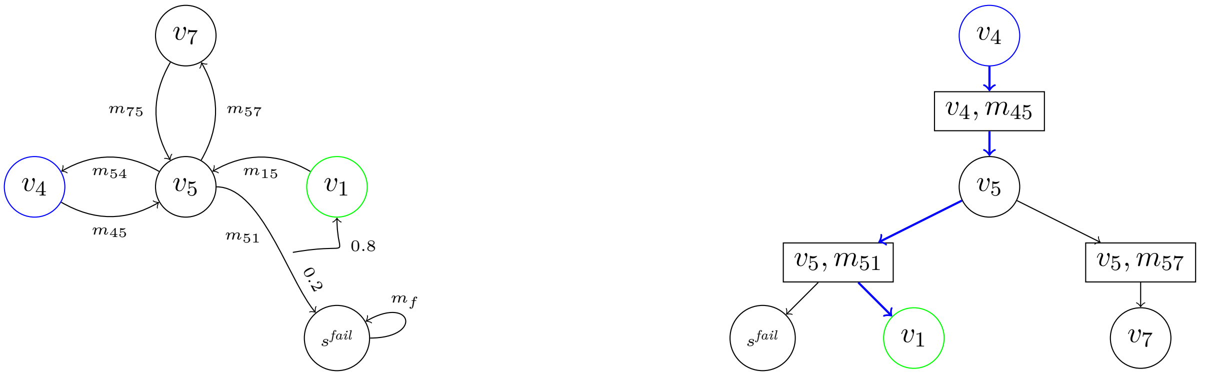
5.1.1 Framework

The THTS framework has five core components: heuristic function, backup function, action selection, outcome selection and trial length. THTS's main advantage is the ability to mix and match these components. The framework alternates operations on two kinds of nodes, decision nodes and chance nodes.

Definition 21 (Decision Node)**.** A *Decision node* is a tuple $n_d = \langle s, V(s) \rangle$ where $s \in S$ is the MDP state and $V(s)$ is the value estimate of state s (see Definition 5 page 13).

Definition 22 (Chance Node)**.** A *Chance node* is a tuple $n_c = \langle s, a, Q(s,a) \rangle$ where $s \in S$ is the MDP state, $a \in A$ is an enabled action for state s and $Q(s,a)$ is the Q-value estimate for that state-action pair (see Definition 18 page 71).

Figure 5.2 illustrates the tree built by THTS for an MDP. Each run of THTS can include a series of trials. A trial is a sequence of state-action pairs, starting in the initial state and ending at some *absorbing* state. We define an absorbing state in the next section but for now let an absorbing state be any state which has no outgoing transitions. This means that once this state is reached no further actions can be taken and no other states can be reached from this state. The root of the tree is in the initial state of the MDP. THTS alternates between visiting decision nodes and chance nodes until it reaches an absorbing state. Not all trials will reach a goal state (see Figure 5.2b). Once a trial terminates, THTS updates the value of all the states and state-action pairs. It then begins a new trial. This process continues until the initial state has been solved or a timeout is triggered.



(a) A simplified fragment of an MDP from the topological map from Figure 4.3. The initial state is v_4 and the task is Fv_1 i.e. the goal state is v_1.

(b) A simplified fragment of the MDP from Figure 5.2a represented as a tree under Trial Based Heuristic Tree Search (THTS).

Figure 5.2: An informal illustration of Trial Based Tree Search. (a) shows a simple MDP. (b) shows a tree built under THTS for this MDP. Circles represent decision nodes i.e states of the MDP. Rectangles represent chance nodes, i.e. state-action pairs of the MDP. The tree is rooted at the initial state. The thick change arrows show one *trial* under THTS. The trial consists of $v_4, \overset{m_{45}}{\rightarrow}, v_5, \overset{m_{51}}{\rightarrow}, v_1$. When the trial terminates the values for all these are backed up. Since the nodes $s^{fail}, v_7, (v_5, m_{57})$ were not visited during the trial, their values are not backed up.

Algorithm 3 reproduced from [KH13] shows the core functions that make up the THTS framework. The input to THTS is the MDP (including the initial state) and a timeout T. T is used to limit the time for a THTS run. If the root node is not solved and the time budget has not been used up, a decision node is visited. When a decision node is visited (Lines 8 to 17) for the first time, it is initialised using the heuristic function. Then an action set is selected based on the action selection component. This action set may contain zero, one or more actions. Most algorithms select just one action. The action selection component returns the set of chance nodes or state-action pairs that need to be visited or executed.

Lines 18 to 24 show the visitChanceNode function. Visiting a chance node involves selecting a set of states or decision nodes which are successors of the chance node. In the case of algorithms such as LAO*, this set may contain all successor states of the state-action pair. However, in sampling based approaches such as RTDP and MCTS, this set typically contains one single state or decision node. The process of visiting decision nodes and then chance nodes continues until a leaf node has been encountered, i.e. an absorbing state has been reached (see Definition 25).

Algorithm 3 The THTS Framework

```
function THTS(MDP $\mathcal{M}$, timeout $T$)
    $n_0 \leftarrow getRootNode(\mathcal{M})$
    while $n_0$.notSolved() & time()$< T$ do
        visitDecisionNode($n_0$)
    end while
    return greedyAction($n_0$)                        ▷ as in Equation (4.5)
end function

function visitDecisionNode(Node $n_d$)
    if $n_d$.notInitialised() then
        initialiseNode($n_d$)
    end if
    $N \leftarrow$selectAction($n_d$)   ▷ see Section 5.1.2.4, typically greedy as in Equation (4.5)
    for $n_c \in N$ do
        visitChanceNode($n_c$)
    end for
    backup($n_d$)              ▷ see Section 5.1.2.3, typically bellman as in Equation (4.3)
end function

function visitChanceNode(Node $n_c$)
    $N \leftarrow$selectOutcome($n_c$)   ▷ see Section 5.1.2.5, typically as per MDP probabilities
    for $n_d \in N$ do
        visitDecisionNode($n_d$)
    end for
    backup($n_c$)              ▷ see Section 5.1.2.3, typically bellman as in Equation (4.3)
end function
```

The THTS Framework reproduced from [KH13]

5.1.2 Components

In THTS, each component plays an important role. Changing the type of any of these components may result in a completely different search algorithm. The following text describes each of the components in THTS. Since [KH13] did not focus on cyclic MDPs, we indicate how these components can aid searching such MDPs.

5.1.2.1 Trial Length

Trial length refers to the length of a path or trajectory. All heuristic search based algorithms have trials that terminate when an absorbing state or a state that is considered to be solved is reached. Some versions of MCTS such as PROST [KE12] introduce a trial length by artificially limiting the number of states explored, therefore, considering a decreased horizon. This can easily be done in Algorithm 3 by incrementing a counter everytime a decision node is visited and modifying line 3 to account for this.

A trial length may also be useful for the case of cyclic MDPs where it is possible for the search to be trapped in cycles or loops in the MDP. Limiting the trial length terminates the trial after the specified number of steps. However, the magnitude of this trial length is very important [MK12]. If the trial length is smaller than the number of states in the MDP, it is not possible to guarantee convergence to the optimal value. This is because some states (including states where the task is completed) may not be reachable within this limit. For very large MDPs, setting the trial length to a number greater than or equal to the number of states in the MDP may be practically infeasible. The reasons for this are twofold: one, that it may result in many wasteful trials and two, that it may use up too much memory.

5.1.2.2 Heuristic Function

The heuristic function is defined in Definition 20 as an initial estimate of the value of a state (Line 10. It is used to guide the search, particularly in the exploration of new states. There is a myriad of ways to compute heuristics. Some heuristics are derived from domain knowledge, for example the Euclidean or Manhattan distance, if the MDP represents a grid world. Other heuristics are domain independent and can be derived by solving abstractions of the MDP. For example, *all-outcome determinisation* assumes that each state-action-state tuple has transition probability 1. The new deterministic model can be solved quickly using classical planning methods and this solution is used as the heuristic.

5.1.2.3 Backup Function

The backup function updates the state-value estimates and action-value estimates (Lines 16 and 23). LAO* and RTDP use a full Bellman backup as in Equation (4.3). MCTS uses a partial backup function based on Monte Carlo backups, estimating transition probabilities using frequency of states seen. Labelled RTDP (LRTDP) [BG03] which is an extension of RTDP, incorporates additional checks in the backup function, aimed at improving convergence.

5.1.2.4 Action Selection

The action selection component chooses an action for a state (Line 12). LAO* and RTDP choose actions greedily based on the Q-value estimates, i.e. in each state, the action with the best Q-value is chosen. The focus of greedy action selection is on exploiting pre-existing knowledge. ϵ-Greedy action selection is a balanced action selection strategy where the greedy action is chosen with probability ϵ and a random action is chosen with probability $1 - \epsilon$. MCTS balances exploration and exploitation using the UCB1 [ACF02] formula. In fact, MCTS uses two different action selection strategies; one for newly encountered states and one for already encountered states.

A *rollout policy* in MCTS refers to the action selection strategy used when a state is encountered for the first time. The most obvious rollout policy or action selection strategy is uniform random action selection. However, more complex strategies that leverage domain knowledge or a heuristic can also be used. It is easy to incorporate these in the THTS framework. There is indeed a wide range of action selection strategies that can be used which makes the THTS framework a good choice for implementing heuristic search algorithms.

5.1.2.5 Outcome Selection

Like action selection, there are multiple outcome selection strategies too. These are used to select the successor state after executing an action (Line 19). Selecting successor states based on their transition probabilities is an obvious outcome selection strategy. Bounded RTDP (BRTDP) [MLG05] uses an outcome selection strategy based on the difference between the upper and lower bounds of the state-value and state-action value estimates.

5.2 Labelled Real Time Dynamic Programming

So far we have looked at a general framework for implementing sampling-based heuristic search algorithms, THTS. Our main motivation for this is that we will be able to vary components to create a search algorithm suited to the multi-agent planning and task allocation and reallocation under robot failure. This section describes and motivates our choice of a trial-based heuristic search algorithm, namely Labelled Real Time Dynamic Programming (LRTDP) [BG03].

5.2.1 Key Definitions

Before we describe LRTDP, we present some definitions we will need to understand the algorithm. The first of these is a property of the heuristic function: admissibility.

Definition 23 (Admissible Heuristic)**.** When the objective is to maximise reward, an *admissible heuristic* does not underestimate the value of any state, therefore $h(s) \geq V^*(s)$ for all $s \in S$. In the case of minimising cost, an *admissible heuristic* does not overestimate the value of any state, therefore $h(s) \leq V^*(s)$ for all $s \in S$.

As a result, admissibility ensures that all promising states are considered during the search. Recall the value function update from Equation (4.3), Section 4.4 where $V_{n+1}(s) \geq V_n(s)$, for all $s \in S$. In words the value function update always improves the value estimate of a state. For this to hold true when using a heuristic function as the initial value estimate, the heuristic function must overestimate the value of a state. This ensures that the improvement is always monotonic.

Another key definition for heuristic search algorithms is that of a *proper policy.* Heuristic search algorithms such as LRTDP, are only able to generate optimal solutions to MDPs if the MDP has at least one proper policy. In order to define a proper policy we must first define goal states and absorbing states.

Definition 24 (Goal States)**.** For an MDP $\mathcal{M} = \langle S, \bar{s}, A, \delta, AP, L\rangle$ (Definition 1) with a reward structure $r(s, a, s)$ (Definition 3), let $T_G \subseteq S$ be the set of *goal* states such that for all $t_g \in T_G$, and for all enabled actions in t_g i.e. $a \in A(t_g)$, $\delta(t_g, a, t_g) = 1$ and $r(t_g, a, t_g) = 0$ i.e. staying in state t_g incurs no further reward. Another way to define these states is by letting $A(t_g) = \{*\}$ where $*$ implies that the robot will continue to stay in this state and collect no further reward.

Definition 25 (Absorbing States)**.** For an MDP $\mathcal{M} = \langle S, \bar{s}, A, \delta, AP, L\rangle$, let $T = T_G \cup T_\perp \subseteq S$ be a set of *absorbing* states for the MDP $\mathcal{M}$. $T_\perp$ is the set of *non-goal* absorbing states such that for all $t_\perp \in T_\perp$, $A(t_\perp) = \{\perp\}$ i.e. there are no outgoing actions from any state in $S_\perp$. Absorbing states are also referred to as *terminal* states.

Definition 26 (Proper Policy)**.** For an MDP $\mathcal{M} = \langle S, \bar{s}, A, \delta, AP, L\rangle$ (Definition 1) with a reward structure $r(s, a, s)$ (Definition 3), let $T = T_G \cup T_\perp \subseteq S$ be a set of *absorbing* states for the MDP $\mathcal{M}$. A policy is *proper* if, from every state s on which the policy is defined, π eventually reaches an absorbing state, $t \in T$ with probability 1 [SHB16].

Remark 11 (Proper Policy Definition)*.* In MDP literature a proper policy is a policy which reaches a *goal* state from every state [MK12]. However in [SHB16] this is modified for the problem of goal probability analysis, replacing *goal* states with *absorbing* states. We use this definition.

These definitions can be extended to the product MDPs from Definition 14 as follows:

Definition 27. (Goal States for a Product MDP) For a product MDP $\mathcal{M} \otimes \mathcal{A}_\varphi = \langle S_\otimes, \bar{s}_\otimes, A, \delta_\otimes, AP, L_\otimes\rangle$ where $\mathcal{A}_\varphi$ is the DFA corresponding to the LTL specification φ (see Definition 11), the set of goal states $T_{\otimes_G} \subseteq S_\otimes$ includes all states $t_g \in T_{\otimes_G} = (s, q)$ where $q \in Q_F$. In words, goal states are states where the mission, Φ has been satisfied. The reward in these states is artificially set to 0 for all actions and once such a state is reached, the choice of action, generally denoted by $*$, does not matter.

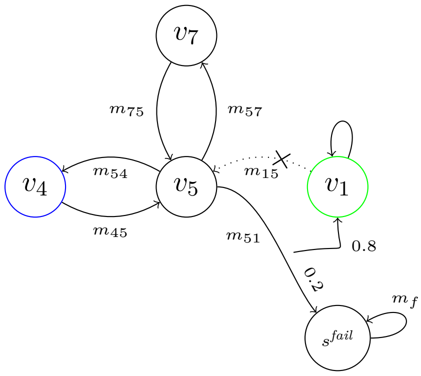


Figure 5.3: Converting an MDP state to a goal state: To convert v_1 to a goal state, the action m_{15} is disabled and a self loop is added to v_1.

Remark 12 (Converting an MDP State to a Goal State or Absorbing State)*.* Any MDP state can be converted to a goal state by disabling all actions in that particular state and in the case of goal states, adding a self looping action (Figure 5.3).

The same can be said for product MDPs. In fact as we saw in Definition 27 after a goal state has been reached, it does not matter what actions are taken. In practice, this can be achieved by not expanding *any* absorbing states (goal or otherwise). Since the DFA state is a feature of the product MDP state, this is easy to do. Whenever a product MDP state's DFA state feature is accepting or absorbing, it is not expanded.

5.2.2 Motivation

There are many heuristic search algorithms. Our choice of algorithm is informed by the availability of robot models and the plan property under consideration. For example, LRTDP assumes that the robot model is known while MCTS requires that a simulator is available to output a (sampled) successor state given a state-action pair as an input. Providing exact values of plan properties is not possible without knowledge of the robot model. Furthermore LRTDP is developed for agents acting in the real world with a short time for computing plans. Therefore, it is able to generate a reasonable action to execute in a state quickly. This makes it an anytime algorithm, since it is able to generate a feasible solution *anytime*. It also makes it an *online* algorithm as the action for a future state can be generated while the agent is executing the algorithm's action for the current state. More specifically, LRTDP is a sampling based heuristic search algorithm. Its ability to generate a reasonable action quickly comes from sampling successor states, instead of visiting all successors. LRTDP uses the knowledge of the underlying agent model to sample states. Since we have an agent model and potentially large MDPs, approaches like LRTDP are best suited to our problem. Not only do they reduce the memory required, but are able to generate a reasonable action, even without solving the MDP completely.

LRTDP builds on the search algorithm Real Time Dynamic Programming (RTDP) [BBS95]. RTDP does not have convergence detection or a stopping criterion, other than a timeout. RTDP trials only terminate when an absorbing state is reached. This means that RTDP trials visit states which may have already converged multiple times.

LRTDP remedies this by adding a stopping criterion other than the timeout. The stopping criterion incorporates convergence detection. As a result, LRTDP ensures that states that can not be improved further and have reached their optimal value are not visited again. This improves the time taken to compute the optimal policy.

LRTDP can be seen as heuristic search alternative to VI (see Section 4.4). It uses Bellman backups (Equation (4.3), page 72) to update the value estimates of states. It also uses the residual (Equation (4.4), page 72) to determine whether a state-value estimate needs to be backed up. LRTDP only updates the value for a state if it is not ϵ-consistent meaning that the residual of the state is greater than the error threshold i.e. $Res^V(s) > \epsilon$ (Definition 19).

For LRTDP to generate an optimal policy the following conditions must be fulfilled:

- The MDP must have at least one proper policy. The presence of a proper policy ensures that all trials eventually terminate and are not stuck in loops.

- The value function, V, improves monotonically. For this to be true, the initial value estimate must be an admissible heuristic. If it is, then the Bellman backup equations ensure that the value estimate always improves.

5.2.3 LRTDP Under THTS

In [BG03] LRTDP is presented using an iterative approach. THTS on the other hand is recursive. Therefore, in order to implement LRTDP under the THTS framework, some modifications to THTS need to be made. Algorithm 4 shows these modifications and presents LRTDP under the THTS framework. The core of the approach is the same as that in Section 5.1.1 with the addition of a *forward backup* for each node and the modification of the *backup* function.

5.2.3.1 Overview

In the following text we walk through LRTDP with the exception of the modified *backup* function which we discuss in the next section. We refer to Algorithm 6 for each of part of the explanation.

Initialisation Given an MDP $\mathcal{M}$ and a timeout T, the algorithm begins by generating the root node i.e. the initial state of the MDP. If the root node is not labelled as solved and there is still time, it is visited. For now we assume that there is some function that allows us to label nodes as solved. We discuss this function later.

Visiting a decision node Since the root node is a decision node (it contains a state but no action), the VisitDecisionNode function is called (Line 4). If the decision node has not been visited before then it is initialised (Lines 11 to 13). This initialisation assigns the value estimate given by the heuristic function to this node. Next a forward Bellman backup is performed i.e. the value estimate is updated using Equation (4.3).

Action selection After this update, the action selection function is called (Line 15). In [BG03] the action selection strategy is greedy action selection. However, under the THTS framework, it can be easily changed. The action selection function returns the corresponding chance nodes. As per greedy action selection, this is just one node.

Visiting a chance node Next the VisitChanceNode function is called (Line 17). The VisitChanceNode function starts from Line 23. First a forward backup of the chance node is performed according to Equation (4.3). Then the outcome selection function is called.

Outcome selection Within the outcome selection function, successor states of this chance node or state-action pair are sampled. LRTDP samples states according to their transition probabilities. For LRTDP the outcome selection function returns just one decision node. Next this decision node is visited and the process is repeated until the trial termination criteria are met.

Trial termination The trial terminates when an absorbing state decision node is reached because absorbing states are labelled as solved using Definition 25. This initiates the *backward* backup process which is different from the aforementioned forward backups as it also includes a mechanism to label states as solved which we discuss next.

Algorithm 4 LRTDP under THTS

```
function THTS(MDP $\mathcal{M}$, timeout $T$)
    $n_0 \leftarrow getRootNode(\mathcal{M})$
    while $n_0$.notSolved() & time()$< T$ do
        visitDecisionNode($n_0$)
    end while
    return greedyAction($n_0$)
end function

function VisitDecisionNode(Node $n_d$)
    backupNode $\leftarrow$ true
    if $n_d$.notSolved() & time()$< T$ then
        if $n_d$.notInitialised() then
            initialiseNode($n_d$)                          ▷ Using $H$ from Definition 20
        end if
        forwardBackup($n_d$)                               ▷  Equation (4.3)
        $N$ $\leftarrow$selectAction($n_d$)                ▷ Typically greedy
        for $n_c \in N$ do
            backupNode$\leftarrow$visitChanceNode($n_c$)
        end for
        backupNode$\leftarrow$backupLRTDP($n_d$,backupNode)
    end if
    return backupNode
end function

function VisitChanceNode(Node $n_c$)
    backupNode $\leftarrow$ true
    forwardBackup($n_c$)                                   ▷  Equation (4.3)
    $N$ $\leftarrow$selectOutcome($n_c$)                   ▷  as per MDP probabilities
    for $n_d \in N$ do
        backupNode$\leftarrow$ backupNode & visitDecisionNode($n_d$)
    end for
    backupNode $\leftarrow$ backup($n_c$,backupNode)
end function
```

Originally LRTDP was presented as an interative algorithm in [BG03]. However, THTS [KH13] is a recursive framework. As a result we adapted LRTDP to fit in the THTS framework as illustrated here.

5.2.3.2 Labelling States as Solved

The main difference between LRTDP and RTDP is due to this backup process. While RTDP simply uses a Bellman backup to backup all nodes in reverse, LRTDP uses a much more involved mechanism, improving the time taken to converge by avoiding states that do not need an update. For any state, it chooses the best action (using the action selection function) and then checks whether all successor states are solved. If so, it labels the current state as solved. So far our explanation of LRTDP has ignored the variable $backupNode$ in Algorithm 4. This variable is a result of modifying LRTDP to fit under the THTS framework, particularly the backward backup process which labels states as solved.

We now describe the process used to label a state as solved. In the original LRTDP formulation all backups are done after the trial has ended. Once the trial has ended a state is backed up and labelled solved if needed. In order to label a parent state as solved, its successor states also need to be marked as solved. Algorithm 5 describes the LRTDP backup which includes the labelling process. The open and closed stacks from Lines 5 to 6 in Algorithm 5 are used to keep track of nodes. The open stack is used to track nodes that are to be visited and the closed stack is used to track nodes that have been visited. The algorithm starts with the current decision node, pushing it to the open stack if it is not solved (Line 7). In Lines 10 to 24 each decision node in the open stack is popped. If the decision node is ϵ-consistent (Line 13) then its successor decision nodes are visited. The decision node itself is pushed onto the close stack.

Note that if a decision node is ϵ-consistent that means that there is no change in the best action for the state the node represents according to the action selection method used. In order to visit the successors of such a state, the action selection method is used to return the chosen chance node (Line 14). Unlike the trial process, there is no outcome selection method here. Instead all successors of the chance node are returned. Each of these is added to the open stack, if it has not already been visited and it has not been labelled as solved. The loop terminates when the open stack is empty i.e. all states on the greedy path from the initial decision node have been explored. Note that if a decision node is not ϵ-consistent, its successors are not visited.

The variable called "allϵ-consistent" in Algorithm 5 is used to flag if any of the states visited were not ϵ-consistent (Line 22). After the loop terminates the labelling process begins. If all states visited were ϵ-consistent, then they are all marked as solved and popped off the closed stack (Lines 26 to 29). However, if any of the visited states were not ϵ-consistent, then they are not marked as solved (Lines 31 to 34). The function returns the flag used to determine if any states were not ϵ-consistent (Lines 36 and 40). This is used to check whether the previous decision node (since this is a recursive algorithm) needs to go through the labelling process (Line 2 in Algorithm 5 and Line 19 in Algorithm 4).

Algorithm 5 Check Solved Function for LRTDP

```
function BACKUPLRTDP(DecisionNode $n_d$,backupNode)
    doPreviousBackups ← false

    if backupNode then
        all$\epsilon$-consistent ← true
        open ← $\emptyset$
        closed ← $\emptyset$
        if $\neg$ $n_d$.solved() then
            open.push($n_d$)
        end if                                  ▷ Visiting successors of current node
        while $\neg$ open.empty() do
            s ← open.pop()
            closed.push(s)
            if residual(s) < $\epsilon$ then
                $n_c$ ← selectAction($n_d$)
                $N_d$ ← successors($n_c$)
                for $n'_d \in N_d$ do
                    if $\neg n'_d$.solved() & $n'_d \notin open \cup closed$ then
                        open.push($n'_d$)
                    end if
                end for
            else
                all$\epsilon$-consistent ← false
            end if
        end while
        if all$\epsilon$-consistent then
            while $\neg$closed.empty() do                ▷ Marking nodes as solved
                $n'_d$ ← closed.pop()
                $n'_d$.setSolved()
            end while
        else
            while $\neg$closed.empty() do
                $n'_d$ ← closed.pop()
                forwardBackup($n'_d$)                    ▷ Equation (4.3)
            end while
        end if
        doPreviousBackups ← all$\epsilon$-consistent
    else
        forwardBackup($n_d$)                             ▷ Equation (4.3)
    end if
    return doPreviousBackups
end function
```

The check solved function from [BG03], slightly modified to fit into the THTS framework.

5.2.3.3 LRTDP run-through

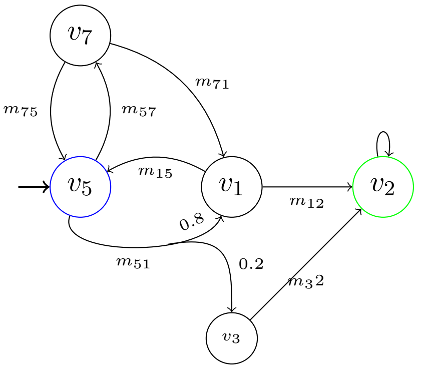


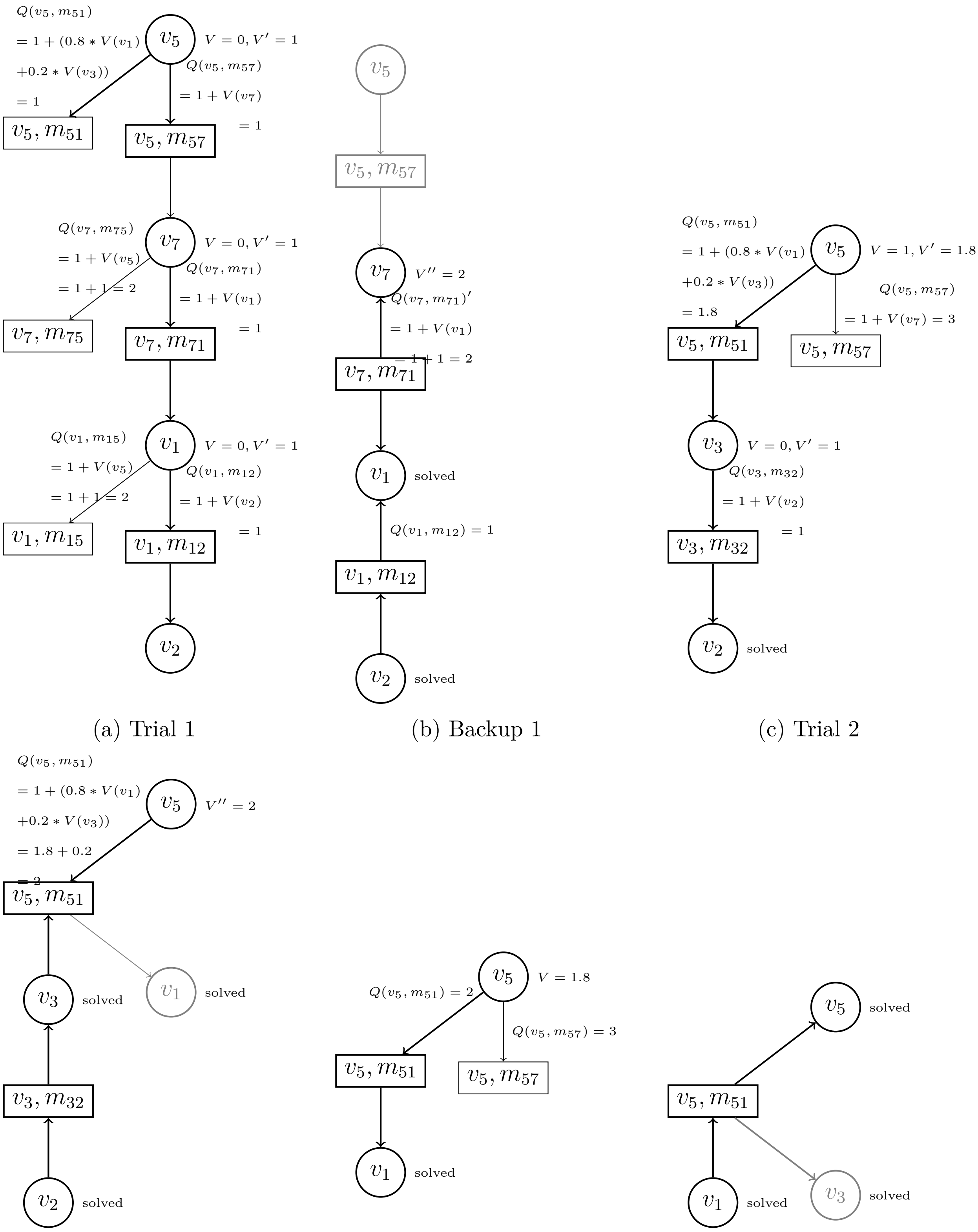
Figure 5.4: An MDP to illustrate LRTDP. Assume the cost of each state-action pair is 1 and the objective is to minimise cost of getting to the goal. The initial state is v_5 and the task is $\mathtt{F}\, v_2$. Since there is only one task, we omit the automaton and show only the MDP.

Example 8 (A run-through of LRTDP)**.** Consider the MDP in Figure 5.4. Let the initial state be v_5, and the task, $\mathtt{F}\, v_2$. Since there is only one task, we do not show the product MDP and omit the automaton for simplicity. The automaton would add one extra state variable with v_2 as an accepting state. So for now we consider v_2 as the goal state.

Let the cost of each state-action pair be 1. The values for all states are set to 0, i.e. our heuristic gives us 0 for all states. Let ϵ be set to 0.01. As a result the value for a state is considered unchanged if the difference between the old and new values is less than 0.01. The goal state v_2 is marked as solved. All other states are marked as unsolved.

Trial 1 The trial starts in v_5, with the following values $V(v_5) = 0$, $Q(v_5, m_{51}) = 1 + (0.8 * V(v_1) + 0.2 * V(v_3)) = 1 + 0 = 1$, and $Q(v_5, m_{57}) = 1 + V(v_7) = 1 + 0 = 1$. Since both actions have the same q-value, m_{57} is chosen arbitrarily (for now). The value is updated, $V(v_5) = 0$. Now v_7 is visited with the following values $V(v_7) = 0$, $Q(v_7, m_{71}) = 1 + V(v_1) = 1 + 0 = 1$, and $Q(v_7, m_{75}) = 1 + V(v_5) = 1 + 1 = 2$. m_{71} is chosen because it has a lower q-value. The value is updated, $V(v_7) = 1$. Now v_1 is visited with the values $V(v_1) = 0$, $Q(v_1, m_{12}) = 1 + V(v_2) = 1 + 0 = 1$, and $Q(v_1, m_{15}) = 1 + V(v_5) = 1 + 1 = 2$. m_12 is chosen because of the lower q-value. The value of v_1 is updated to $V(v_1) = 1$. Since v_2 is the goal, the trial ends here.

Backup for Trial 1 The states visited were v_2, v_1, v_7, v_5. Since v_2 is solved, the BackupLRTDP function returns true. v_1 is not solved. The best action in v_1 is still m_{12} with $Q(v_1, m_{12}) = 1$, therefore there is no change in the value for v_1 and the residual is 0. The successor of m_{12} is v_2 which is solved. Since none of v_1's successors (when taking the best action) are unsolved, v_1 is labelled solved. The BackupLRTDP function returns true. v_7 is not solved. The best action in v_7 is m_{71}, however, its q-value is changed from 1 to 2 i.e. $Q(v_7, m_{71}) = 1 + V(v_1) = 1 + 1 = 2$. Due to this change,no further states are visited, i.e. v_7, m_{71}'s successors are not visited. The variable $all\epsilon - consistent$ in Algorithm 5 is false, so the value for v_7 is updated to the new value i.e. $V(v_7) = 2$. The BackupLRTDP function returns false and the labelling ends. A new trial begins i.e. Trial 2.



(a) Trial 1

(b) Backup 1

(c) Trial 2

(d) Backup 2

(e) Trial 3

(f) Backup 3

Figure 5.5: A run through of LRTDP for the MDP in Figure 5.4 as explained in Example 8

Trial 2 The trial starts in v_5, with the following values $V(v_5) = 1$, $Q(v_5, m_{51}) = 1 + (0.8 * V(v_1) + 0.2 * V(v_3)) = 1 + 0.8 = 1.8$, and $Q(v_5, m_{57}) = 1 + V(v_7) = 1 + 2 = 3$. m_{51} is chosen due to the lower q-value. The value is updated, $V(v_5) = 1.8$. The successors of m_{51} are sampled and v_3 is visited. The values of note here are $V(v_3) = 0$, and $Q(v_3, m_{32}) = 1 + V(v_2) = 1 + 0 = 1$. m_{32} is chosen. The value is updated, $V(v_3) = 1$. Since v_2 is the goal, the trial ends here.

Backup for trial 2 The states visited were v_2, v_3, v_5. v_3 is not solved but once the BackupLRTDP function is called, it marks it as solved, since all its successors are solved and there is no change in its value. Next the BackupLRTDP function for v_5 is called. The q-value for its best action, m_51 has changed from 1.8 to $Q(v_5, m_{51}) = 1+(0.8*V(v_1)+0.2*(v_3)) = 1+(0.8+0.2) = 2$. Therefore, the residual for this state is now 1. This state is not marked as solved and its value is updated to 2. The BackupLRTDP function returns false and the backup process is terminated.

Trial 3 The trial starts in v_5, with the following values $V(v_5) = 2$, $Q(v_5, m_{51}) = 1 + (0.8 * V(v_1) + 0.2 * V(v_3)) = 2$, and $Q(v_5, m_{57}) = 3$. m_{51} is chosen due to the lower q-value. The sampled next state is v_1. v_1 was marked as solved in the first trial, therefore, the trial terminates here.

Backup for trial 3 The states in the trial are v_5, v_1 with v_1 marked as solved. Therefore, the BackupLRTDP function is called for v_5. The best action in v_5 is still m_{51} and its value remains unchanged. All the successors are also marked as solved. Consequently, v_5 is marked as solved. When BackupLRTDP returns true, there are no more nodes that were visited. Therefore, the backup loop terminates.

Now that the initial state is solved, LRTDP terminates. The policy can now be obtained by getting the greedy action for each state.

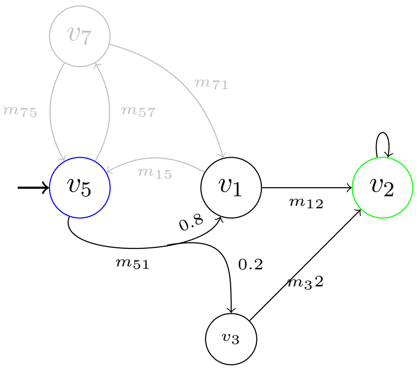


Figure 5.6: The policy obtained for the MDP in Figure 5.4, following Example 8. Only the states and actions in the policy starting in the initial state are shown, all others are greyed out.

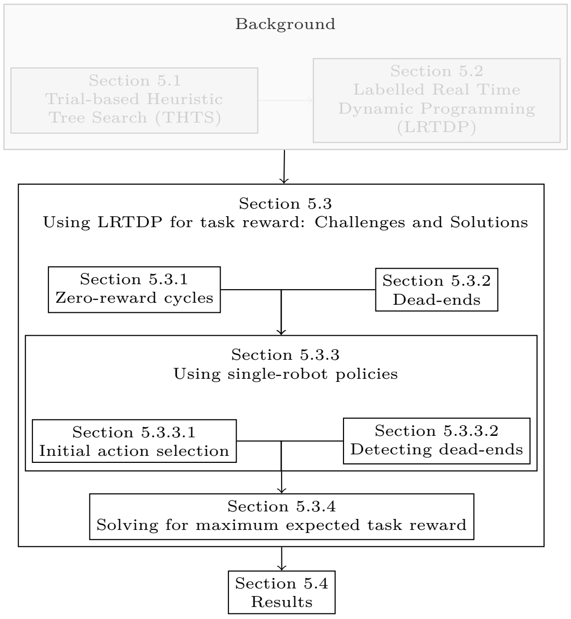


Figure 5.7: The previous sections (grayed out) introduced THTS and LRTDP which was essential background for this chapter. The next sections discuss the challenges of applying LRTDP to the problem formulated in Section 4.2 namely, zero-reward cycles and dead-ends. Existing solutions are utilised for both. Next, the use of single-robot policies to guide the search through initial action selection i.e. a *rollout* policy is explained. It is also shown how to detect deadends using these policies. Finally the solution to task allocation and planning problem is proposed by introducing a novel cost function relative to the number of tasks achieved by the team. The chapter ends with a discussion of the results of applying LRTDP to tests first described in Section 4.3

5.3 LRTDP for Maximising Task Reward with Robot Failures

In this section we show how LRTDP can be applied to the problem in Section 4.2, maximising task reward for a multi-robot scenario with robot failures. Figure 5.8 shows a simplified MDP fragment of the example from Figure 4.2. The robot's initial state is v_4 (marked in change) and the mission is $\mathtt{F}\, v_1$. Since this example only contains one task, we omit the DFA parts of the state. Throughout this section, we will refer to this example to illustrate LRTDP's limitations and our proposed solutions. As we have seen in previous sections, LRTDP can operate on the joint product MDP without exploring the entire state-action space. Therefore, LRTDP can operate on the joint product MDP. However, with the objective of maximising task reward and reallocating tasks on robot failure, the following problems arise:

1. The search can get stuck in zero-reward cycles. These are introduced due to the task reward structure (see Section 4.2) being 0 in certain parts of the product MDP.

2. The search gets stuck in deadends e.g. states where all robots have failed.

5.3.1 Removing Zero-Reward Cycles

Our first problem arises due to the presence of *zero-reward cycles* since our objective is to maximise task reward. Recall from Section 5.1.2 that there are components in THTS that can be varied to change the workings of the algorithm. In this section we illustrate the effect of some of these in attempts to deal with *zero-reward cycles* and present existing solutions.

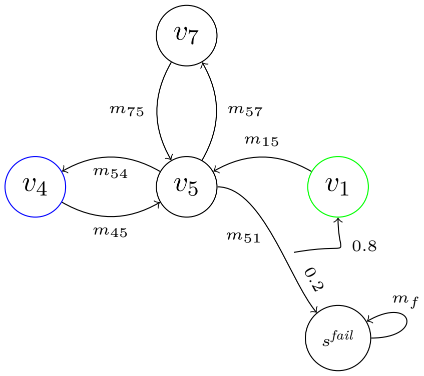


Figure 5.8: Modified fragment of our toy example MDP with the door removed. Assume the robot is in state v_4 and has to get to state v_1.

Example 9 (Zero-Reward Cycles)**.** The MDP in Figure 5.8 has a proper policy if we consider both v_1 and s^{fail} to be absorbing states. Let us look at this example with the objective of maximising task reward, $E^{\max}_{\mathcal{M}}(cumul^{\varphi}_{r})$. The task reward structure is of the form:

$$r(s,a,s') = \begin{cases} 1 & \text{if } s = v_5, a = m_{51}, s' = v_1 \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

The heuristic is constant for all states except the absorbing states:

$$h(s) = \begin{cases} 0 & \text{if } s \in \{v_1, s^{fail}\} \\ 1 & \text{otherwise} \end{cases} \tag{5.2}$$

The heuristic for all state-action pairs is also a constant set to 1. This is an admissible heuristic for the maximum reward since it assumes that the goal can be reached from all states, except the absorbing states. The only state with multiple actions is v_5.

Let the first trial be the path, $v_4 \overset{m_{45}}{\to} v_5 \overset{m_{51}}{\to} v_1$. Once the trial terminates, the backup function will mark v_1,s^{fail} and (v_1, m_{51}) as solved since v_1 and s^{fail} are both absorbing states and m_{51} only leads to these two states. However, state v_5 will not be marked as solved. The two actions in v_5, have the corresponding Q-values, $Q(v_5, m_{57}) = 1$ and $Q(v_5, m_{51}) = 0.8$. Recall from Definition 18 that $Q(s, a)$ is a value estimate for a state-action pair. Specifically, $Q(v_5, m_{51})$ is updated from 1 to 0.8. As a result, the greedy action changes in the backup and its successor, v_7 is not solved. Therefore, v_5 is not marked as solved.

In the next trial, using greedy action selection, the path will be an infinite path $v_4 \overset{m_{45}}{\rightarrow} v_5 \overset{m_{57}}{\rightarrow} v_7 \overset{m_{75}}{\rightarrow} v_5 \overset{m_{57}}{\rightarrow} ...v_7 \overset{m_{75}}{\rightarrow} v_5...$, looping between v_5 and v_7. Since there are no rewards for actions m_{57} and m_{75}, no update takes place and the robot is stuck in a *zero-reward cycle.*

5.3.1.1 Effect of fixing trial length

As mentioned earlier, adding a fixed trial length can be used to stop the search when it is trapped in a loop. However, using a fixed trial length in this example would still result in a policy that has a loop. While the backup function will mark all states as solved, the resulting policy will include the loop $v_5 \overset{m_{57}}{\rightarrow} v_7 \overset{m_{75}}{\rightarrow} v_5....$

5.3.1.2 Effect of an inadmissible heuristic

It is clear that this behaviour is due to the *admissibility* of the heuristic. We can forgo admissibility and initialise all state-action pairs with a lower bound on the task reward e.g. 0. This approach would be greedy and would lead us to the goal in this example. However, this will not be the case for all scenarios. For instance consider a scenario where a certain path to the goal has a task reward of 1, while another has a task reward of 0.8. If the path with the lower task reward is explored first using greedy action selection, the path with a task reward of 1 will never be explored. Replacing the greedy action selection with one that chooses non-greedy actions with some probability is one way to avoid this. However, due to the probabilistic nature of the action selection method, an optimal policy is not guaranteed.

5.3.1.3 Effect of a secondary objective

Another possible solution to the removal of zero-reward cycles might be the use of a secondary objective. Let $c(s, a, s') = 1$ be the cost for all state-action-state tuples. Recall from Definition 4 that $cumul_r^\varphi$ is the cumulative reward starting from the initial state to an absorbing state. Similarly, let $cumul_c^\varphi$ be the cumulative cost of starting in the initial state and terminating in an absorbing state. The updated objective is to find a policy that maximises the expected task reward using expected cost as a tie breaker as explained in Section 4.4, Equations (4.6) and (4.7). The value update is the same as the one defined in Algorithm 2, Line 24. Let Q_r and V_r denote the state-action value and state-value estimates for the task reward. Let Q_c and V_c denote the state-action value and state-value estimates for the cost. The task reward Q-value for action m_{51} in state v_5 is 0.8, whereas $Q_r(v_5, m_{57}) = 1$. Despite the addition of a secondary objective, the value estimates do not change. The estimate of $Q_c(v_5, m_{57})$ continues to increase, without any effect on $Q_r(v_5, m_{57})$. This does not solve the problem of zero-reward cycles.

5.3.1.4 Existing Solutions

From Example 9 we can see that zero-reward cycles *trap* the search in a set a of states. Solutions for MDPs with zero-reward cycles have been presented in both [Kol+11] and [Brá+14]. The common theme for both solutions is to detect cyclic paths in the MDP and then remove the cycles. [Kol+11] introduces Find-Revise-Eliminate-Traps (FRET) which terminates a trial when the residual of a state does not change. Each trial run can be seen as a fragment of the MDP. FRET uses this MDP fragment and identifies any cycles using Tarjan's Algorithm [Tar72]. Tarjan's algorithm is widely used to detect cycles in graphs. Once cycles are identified, FRET essentially collapses all states in a cycle into one super state. If there are no actions that lead out of this super state, then the state's value is set to $-\infty$. If there are actions leading out of the state, then the super state's value is set to that of the best action leading out of the state. The values of all states and state-action pairs inside the super state are the same.

[Brá+14] has a similar approach to FRET, however, cycles are detected during the trial run. Once a cycle is detected, the trial is halted, and the cycle is collapsed. The terminology for cycles here is end components. After collapsing the cycle, the trial begins again.

The methods in [Kol+11] and [Brá+14] are not straightforward to implement. Furthermore the cycle detection has some overhead. As the size of the MDP state-action space increases, this overhead may become a bottleneck.

5.3.1.5 Conclusion

As we have seen zero-reward cycles are due to the reward structure itself i.e. there are multiple state-action-state tuples where there is no reward. Changing the reward for all such state-action-state tuples to a value greater than 0 is one way to remove these cycles. However, such a reward structure can not be used to count the number of completed tasks. Recall that cost minimisation is a dual for reward maximisation. Replacing the reward structure with a cost structure such that each state-action-state tuple has a cost greater than 0 associated with it can be used to avoid zero-reward cycles. However, this too brings its own set of problems due to the possibility of robot failure considered in our MDP models. The next sections illustrate the difficulties with using cost minimisation alone as an objective and their solutions. They also show how to generate a cost structure that is analogous to the task reward structure.

5.3.2 Preemptively Avoiding Unavoidable Dead-ends

By changing the objective of the problem from one of task reward maximisation to one of cost minimisation, we have effectively removed the first of the two aforementioned challenges. However, the challenge of dead-ends still remains. In this section, we discuss the following:

- The challenge of deadends due to the failstate and proposed solutions.
- The challenge of deadends due to partial satisfaction of the mission, i.e. achieving as much of the mission as possible. In fact, this leads to not just deadends but *dead paths* which we can simply treat as deadends identifying the states where these paths

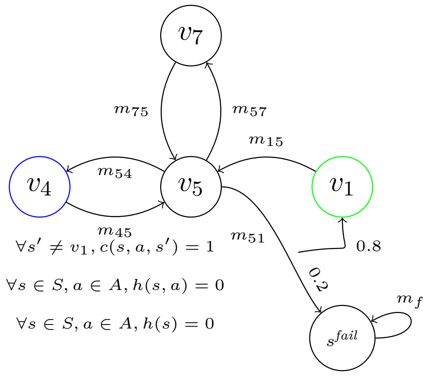

v_7
m_{75}
m_{57}
m_{15}
v_4
m_{54}
v_5
v_1
m_{45}
$\forall s' \neq v_1, c(s,a,s') = 1$
m_{51}
0.8
0.2
$\forall s \in S, a \in A, h(s,a) = 0$
m_f
$\forall s \in S, a \in A, h(s) = 0$
s^{fail}

Figure 5.9: Modified fragment of our toy example MDP with the door removed as in Figure 5.8. Assume the robot is in state v_4 and has to get to state v_1. However, the reward is replaced by a cost function, which changes the heuristic as well.

begin.

Figure 5.9 shows the MDP from Figure 5.8 but with the objective of cost minimisation. This makes the failstate s^{fail} in Figure 5.9 a hindrance to generating an optimal policy. When the search visits this state, it is trapped here forever. In the MDP literature such a state is an *unavoidable dead-end* in the MDP [MK12]. Specifically a dead-end is any state reachable from the initial state such that the probability of reaching a goal state from this dead-end is 0. A dead-end is unavoidable if it is not possible to reach the goal state without taking an action that may lead to the dead-end. This means that there is no *proper* policy for the MDP.

Example 10 (Unavoidable Dead-ends)**.** In this example, we illustrate the problem an unavoidable dead-end poses. Consider the MDP in Figure 5.9 with the objective of minimising the expected cumulative cost, i.e $E^{\min}_{\mathcal{M}}(cumul^{\varphi}_{c})$. Remember that once a robot ends up in a failstate, it stays there indefinitely, taking action m_f every time. Let the cost structure be of the form:

$$c(s,a,s') = \begin{cases} 0 & \text{if } s = v_5, a = m_{51}, s' = v_1 \\ 1 & \text{otherwise} \end{cases} \tag{5.3}$$

Let the heuristic be constant for all states and state-action pairs i.e. 0.

Let the first trial be $v_4 \overset{m_{45}}{\rightarrow} v_5 \overset{m_{51}}{\rightarrow} v_1$. The backup update of the state-values in this trial results in the following: $V(v_1) = 0$ since it is the goal state, $V(v_5) = 0.2 + V(v_1) = 0.2$ and $V(v_4) = 1 + V(v_5) = 1.2$.

In the next trial, the following states are seen as per a greedy action selection strategy: $v_4 \overset{m_{45}}{\rightarrow} v_5 \overset{m_{57}}{\rightarrow} v_7 \overset{m_{75}}{\rightarrow} v_5 \overset{m_{51}}{\rightarrow} s^{fail} \overset{m_f}{\rightarrow} s^{fail} \dots$, i.e. the robot is stuck in the failstate. In v_5, m_{57} is chosen because its q-value is 0 since v_7 has not been expanded. On update this value changes to 1. From v_7 there is no other action but m_{75} so that is the chosen action. The trial is in v_5 again. This time m_{51} is chosen since its q-value is 0.2. If the failstate is sampled again, the value of the failstate updates to 2 (m_f has a cost of 1) and the trial is stuck in the failstate. Adding a trial length to this would eventually make the action m_{51} less attractive, as $0.2 \times V(s^{fail})$ would increase over time. This is because the value of the failstate is updated every time the failstate is visited. As a result both m_{51} and m_{57} would end up having infinite expected accumulated cost. This would permeate to v_4, resulting in no feasible policy.

5.3.2.1 Converting dead-ends to absorbing states

In Example 10, both v_7 and s^{fail} contribute to the increase in accumulated cost. One possible solution to the problem is to artificially convert dead-end states to non-goal absorbing states. To do this, the action m_f needs to be disabled in this state. This means that there is no cost associated with this state. However, there is also no cost associated with the goal state. With the objective of minimising cost the search algorithm may generate a policy to the dead-end instead of the goal state because the accumulated cost to get to the dead-end is lower. Therefore, this is not a viable solution.

The MDP in Figure 5.8 can be classified as a Stochastic Shortest Path MDP [BT91] with unavoidable dead-ends (SSPUDE MDP) [MK12].

Definition 28 (Stochastic Shortest Path MDP with an initial state)**.** From Definition 1, an MDP is a tuple $\mathcal{M} = \langle S, \overline{s}, A, \delta, AP, L \rangle$. Let $c(s, a, s)$ be a cost function such that for all $s \in S$, $a \in A(s)$, $C(s, a, s) \geq 0$.

A Stochastic Shortest Path MDP (SSP MDP) with an initial state is an MDP, $\mathcal{M}$, which satisfies the following conditions [Kol+11]:

- There exists at least one *proper policy*, π.
- For every improper policy π, for every $s \in S$ where π is improper, $V^{\pi}(s) = \infty$.

[MK12] shows that SSPUDE MDPs can be converted to SSP MDPs by using a finite or infinite penalty for dead-ends. It is important to note that the work in [MK12] focuses on minimising costs to the goal.

In the case of an infinite penalty for dead-ends, [MK12] uses FRET to first find all policies that lead to the goal with positive probability. It then uses a heuristic search method to find a policy that minimises cost over the one's that maximise probability of reaching the goal. In the case of a finite penalty, the authors cap the value update function. Specifically if the value of any state is greater than the finite penalty for the dead-end, the value does not change. This can be done by putting a "cap" on the cost of all states during the Bellman backup.

$$V_n(s) \leftarrow \min_{V}(D, \arg\max_{a \in A_J^{\Phi}} \sum_{s' \in S} \delta(s, a, s'))[r(s, a, s') + V_{n-1}(s')] \tag{5.4}$$

where D is the finite penalty.

The value of this finite penalty needs to be set considering the fact that the cost of a path that leads to a dead-end should not be less than the cost of a path that does not.

A loose upper bound on the cost of satisfying one task could be the number of states in the joint MDP, $|S_J| = |S_1^l \times \cdots \times S_n^l \times S_1^g \times ... \times S_k^g|$, where $i \in \{1, \ldots, n\}$. Since each task has its own DFA, and the cost of each action is 1, an upper bound for all tasks would be the number of states in the joint product MDP $\mathcal{M}'_J$.

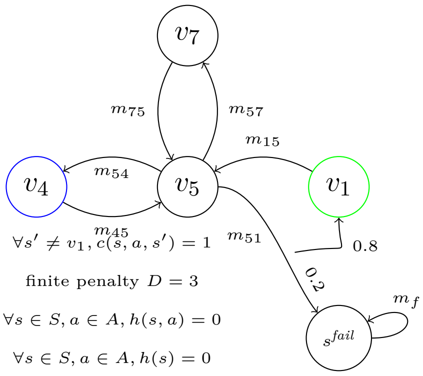


Figure 5.10: Modified fragment of our toy example MDP with the door removed as in Figure 5.8. Assume the robot is in state v_4 and has to get to state v_1. However, the reward is replaced by a cost function, which changes the heuristic as well. Furthermore, there a finite penalty of 3 is used to cap the value function at 3.

Example 11 (Unavoidable Dead-ends with a Finite Penalty)**.** We now revisit Example 10, modifying it to add a finite penalty $D = 3$ as shown in Figure 5.10. Let the first trial be $v_4 \overset{m_{45}}{\to} v_5 \overset{m_{51}}{\to} v_1$. The backup would update the state-values as follows: $V(v_4) = 1.2, V(v_5) = 0.2 \times 1 = 0.2, V(v_1) = 0$. In the next trial we will see $v_4 \overset{m_{45}}{\to} v_5 \overset{m_{57}}{\to} v_7 \overset{m_{75}}{v}_5 \overset{m_{51}}{\to} s^{fail} \overset{m_f}{\to} s^{fail} \overset{m_f}{\to} s^{fail} \overset{m_f}{\to} s^{fail}$. This is similar to the loop in Example 10 except that m_f is only executed thrice due to the finite penalty. Once the value estimate of s^{fail} is 3, s^{fail} will be marked as solved since its residual will be 0. This will terminate the trial and update value estimates to $V(v_4) = 1 + V(v_5) = 1.6, V(v_7) = 1, V(v_5) = 0.2 \times 3 = 0.6, V(v_1) = 0$. While $Q(v_5, m_{57}) = 1$, $Q(v_5, m_{51}) = 0.6$ due to the finite penalty. From this we can see that the state-action pair (v_5, m_{51}) is already more attractive than m_{57}. If the probability of reaching the goal with (v_5, m_{51}) was 0.1, then we would see the trial get stuck in the loop $v_5, \ldots, v_7 \ldots$ till the value of (v_5, m_{57}) crossed $0.9 \times 3 = 2.7$. After that (v_5, m_{51}) would again become more attractive and we would have the correct policy.

5.3.2.2 Partial Satisfaction

So far we have been looking at an example with only one task. If the mission consisted of more than one tasks, where it was impossible to complete one of the tasks, we would run into a similar problem.

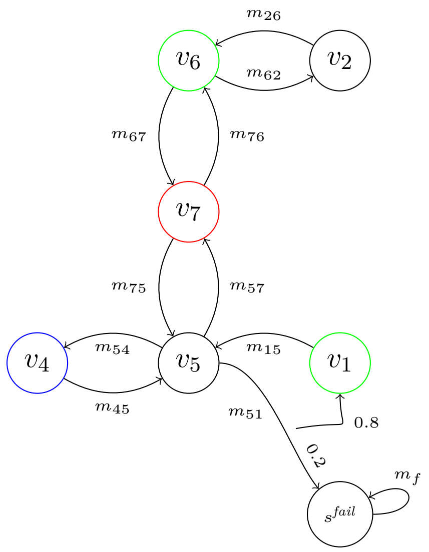


Figure 5.11: Modified fragment of our toy example MDP with the door removed. Assume the robot is in state v_4 and has the mission $\mathtt{F}\, v_1, \mathtt{F}\, v_6, \mathtt{G}\, \neg v_7$. The DFA portion is omitted for compactness.

Example 12 (Partially Satisfiable Missions)**.** Figure 5.11 extends the scenario in Figure 5.8 by adding more tasks to the mission. In particular, the robot now has to visit states v_1 and v_6 while avoiding v_7. The robot can only satisfy task $\mathtt{F}\, v_6$ through v_7. The safety task v_7 is a hard constraint that can not be violated. Therefore, the robot can only achieve part of the mission, i.e. $\mathtt{F}\, v_1$. Consequently, we run into a similar problem, as the one in Example 10. The accepting state here is one where both v_1 and v_6 have been visited. As mentioned above, this is impossible in our example.Let v_7 be an absorbing state whose value is fixed as the finite penalty D ,i.e. $V(v_7) = D$. As a result, all states v_4, v_5, v_1, s^{fail} will eventually have the value D as well, since there is no other absorbing state. Therefore, the search will not be able to generate a feasible policy.

5.3.2.3 Proposed Solution

One way to solve the problem in Example 12 is by introducing a penalty D' which is relative to the number of tasks completed in a state:

$$D'(s) = D\frac{m - m'(s)}{m} \tag{5.5}$$

where m is the total number of tasks, $m'(s)$ is the number of completed tasks in state s and D is the maximum penalty. $m'(s)$ can be determined using the DFA state variables in the product MDP state. Recall that the product MDP is a tuple $\mathcal{M} \otimes \mathcal{A}_\varphi = \langle S_\otimes, \overline{s}_\otimes, A, \delta_\otimes, AP, L_\otimes \rangle$ where $\mathcal{A}_\varphi$ is the DFA corresponding to the LTL specification φ (see Definition 11). Each state of the product MDP consists of the MDP state and the corresponding DFA state i.e. $s_\otimes = (s, q) \in S_\otimes$. For the mission $\Phi = \langle \mathtt{F}\, v_1, \mathtt{F}\, v_6, \mathtt{G}\, \neg v_7 \rangle$, the joint product MDP is $\mathcal{M} \otimes \mathcal{A}_{\varphi_0} \otimes \mathcal{A}_{\varphi_1} \otimes \mathcal{A}_{\varphi_{\neg safe}}$, where $\varphi_0 = \mathtt{F}\, v_1, \varphi_1 = \mathtt{F}\, v_6, \varphi_{safe} = \mathtt{G}\, \neg v_7$. Therefore a state in the product MDP has the form $s_\otimes = (s, q_{\varphi_0}, q_{\varphi_1}, q_{\varphi_{\neg safe}})$. The number of tasks completed in a state $s_\otimes$ is equal to the number of DFA states in $s_\otimes$ which belong to the set of accepting states Q_{F_φ} of the corresponding DFA excluding the one for $\varphi_{\neg safe}$.

The cost function D' penalises states where fewer tasks are achieved more. As a result it is analogous to the task reward function. The task reward structure counts the number of tasks completed as a result of executing action a in state s and ending up in state s'. Similarly, the cost function D' reduces the penalty for an absorbing state according to the number of tasks completed in that state. Note that the cost function D' is not a dual of the task reward function. Nonetheless this cost function can be used to generate a feasible policy for our problem.

From Example 12 it is clear that detecting states from which no further tasks can be completed is not trivial. If these states are not detected then it is not possible to generate a feasible policy that maximises the expected number of tasks. Therefore a mechanism for detecting such states in required. In the next section we look at such a mechanism for a multi-robot planning problem.

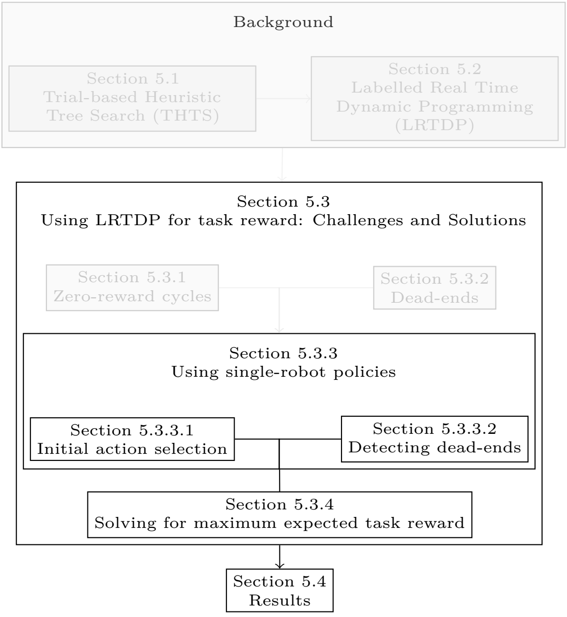


Figure 5.12: The previous sections (grayed out) essential background (THTS and LRTDP) and discussed solutions to the challenges of applying LRTDP to the problem formulated in Section 4.2 namely, zero-reward cycles and dead-ends. The upcomping sections show the use of single-robot policies to guide the search through initial action selection i.e. a *rollout* policy. They also show how to use detect dead-ends using these solutions. Finally the solution to task allocation and planning problem is proposed by introducing a novel cost function relative to the number of tasks achieved by the team. The chapter ends with a discussion of the results of applying LRTDP to tests first described in Section 4.3

5.3.3 Using Single-robot Policies

So far we have been looking at single robot examples. We have shown how the challenges associated with using LRTDP for the problem of partial task satisfaction can be solved. Solving a single robot's MDP is much cheaper than solving a multi-robot MDP. This means we can generate solutions to single robot MDPs quickly. These solutions can be used to provide a heuristic for task reward and cost, to detect dead-ends and to generate a rollout policy. We generate single robot policies under the assumption that there is no task allocation and each robot must maximise expected task reward for the entire mission on its own.

As shown in Section 4.2, these policies can be obtained via nested value iteration (Algorithm 2) over the product of the robot's MDP with the mission DFA, using task reward maximisation as the main objective and cost minimisation as the tie-breaking objective.

Recall the definition of the robot MDP from Definition 15. Given a robot MDP $\mathcal{M}_i = \langle S_i, \overline{s}_i, A_i, \delta_i, AP, L_i \rangle$ for the i^{th} robot in the team with the mission specification $\Phi = \langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$, the local product MDP is $\mathcal{M}_i^\Phi = \mathcal{M}_i \otimes \mathcal{A}_{\varphi_1} \otimes \cdots \otimes \mathcal{A}_{\varphi_m} \otimes \mathcal{A}_{\varphi_{\neg safe}}$. Let *tasks* be a reward function that counts the number of completed tasks after action a has been executed in state s and the robot has reached state s'. Let *cost* be a cost function $c(s, a, s') = 1$ for all state-action-state tuples. The policy for robot i can be generated as

$$\pi_i^\Phi = \textsc{NestedValueIteration}(\mathcal{M}_i, [\mathit{tasks}, \mathit{cost}]) \tag{5.6}$$

5.3.3.1 Initial Actions from Single-robot Policies

Let $\mathcal{M}_J$ be the joint multi-robot MDP (Definition 17) and $\mathcal{M}_J^\Phi$ (see Section 4.2) be the joint product of the multi-robot MDP and all DFAs of the mission Φ. The projection function (Definition 16) maps states of the joint MDP to those of the individual robot MDPs. The number of robots in the team is n.

With slight abuse of notation, we say that the same projection function can be used to map states of the joint product MDP to those of the joint local product MDP. The rollout policy maps states in the joint product model to actions using the policy π_i^Φ i.e.

$$\pi_r(s_J^\Phi) = \left(\pi_i^\Phi([s_J^\Phi]_1), \ldots, \pi_i^\Phi([s_J^\Phi]_n)\right) \tag{5.7}$$

This policy can then be used to explore states in the joint product MDP. As mentioned earlier, a *rollout policy* is commonly used in Monte-Carlo Tree Search to guide exploration of new states. In a similar vein, we use this rollout policy to generate actions for states being visited for the first time. This ensures that our search is guided towards promising states. Note that this rollout policy can not be used to generate an *admissible* heuristic function for the cost. This is because it considers single agent solutions and can therefore overestimate the cost of completing the entire mission.

Instead of using the rollout policy for the heuristic function, we use it to guide the search through action selection i.e. select actions for states that have not been visited before. Once a state has been visited, we then use an ϵ-greedy action selection strategy to choose actions.

5.3.3.2 Detecting Dead-ends

The single-robot policies are also useful for detecting dead-end states where the expected accumulated task reward is 0. To this end we save the values generated when solving the single-robot problems in Equation (5.6). We then use these to determine if a certain state is a dead-end. The intuition behind this is that if none of the robots are able to gather more task reward in a certain joint state individually, then it is not possible for the team to gather further reward in that state either. Let $V^{\pi_i^\Phi}$ be the value function obtained from solving single-robot problems using NVI. A state $s_J^\Phi \in S_J^\Phi$ is marked as a dead-end if

$$\sum_{i=1}^{n} V^{\pi_i^\Phi}([s_J^\Phi]_i) = 0 \tag{5.8}$$

Therefore, this joint state can be marked as a dead-end and assigned the penalty D'. In fact each dead-end state is converted to an absorbing state by disabling any actions in that state.

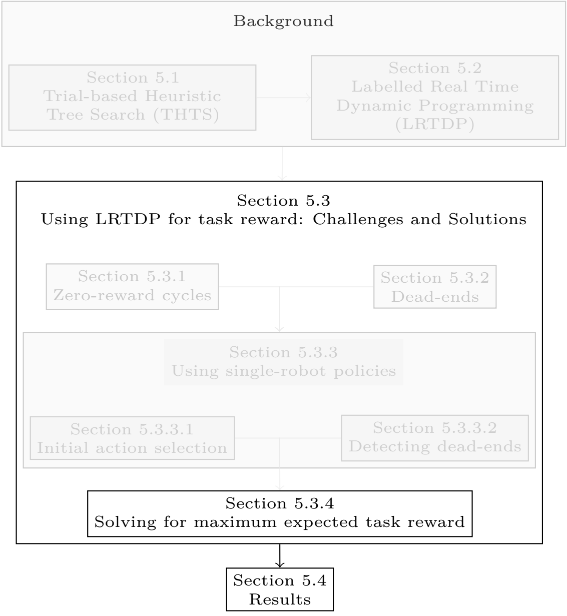


Figure 5.13: The previous sections (grayed out) presented essential background (THTS and LRTDP) and discussed solutions to the challenges of applying LRTDP for task reward maximisation. They also showed how to use single-robot policies to guide the search and detect dead-ends. The upcoming section combines these, presenting an updated MDP formulation to solve the problem in Section 4.2. The chapter ends with a discussion of the results of applying LRTDP to tests first described in Section 4.3

5.3.4 Solving for Maximum Expected Task Reward

In order to avoid zero-reward cycles, and partially satisfy a mission, we use a relative cost structure D' (Equation (5.5)) for all absorbing states, including dead-ends. Using this information, we modify the problem in Section 4.2 for LRTDP as follows. Given a set of n robots and a mission specification $\langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$, our aim is to derive a joint policy for the robots which minimises cost without violating the safety constraint φ_{safe}. Though this formulation incorporates task allocation, it does not solve for maximum task reward. We are using the cost structure D' as a proxy for the task reward. In order to calculate the expected accumulated task reward, we can perform VI on the resulting joint policy.

Given the set of local MDPs (Definition 15) we can create the joint MDP $\mathcal{M}_J$ using Definition 17. This is combined with the mission specification $\langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$ using the product construction described in Section 2.3 to give us $\mathcal{M}_J^{\Phi}$. We can now use $\mathcal{M}_J^{\Phi}$ in a

Stochastic Shortest Path (SSP) problem.

Definition 29. The Multi-robot Task Allocation and Planning Stochastic Shortest Path (SSP) problem is a tuple $SSP(\mathcal{M}_J^{\Phi}) = \langle \mathcal{M}_J^{\Phi}, C_J^{\Phi}, T_J^{\Phi}, D_J^{\Phi} \rangle$ where:

- $\mathcal{M}_J^{\Phi}$ is the multi-robot joint product MDP,
- T_J^{Φ} is the set of absorbing states including accepting states and those detected as dead-ends using single agent solutions,
- $C_J^{\Phi} : S_J^{\Phi} \setminus T_J^{\Phi} \times A_J^{\Phi} \times S_J^{\Phi} \to \mathbb{R}_{\geq 0}$ is a cost structure that assigns cost to state-action-state tuples excluding actions in terminal states,
- $D_J^{\Phi} : T_J^{\Phi} \to \mathbb{R}_{\geq 0}$ is the one-time terminal cost of reaching an absorbing state.

The one-time terminal cost is calculated as follows:

$$D_J^{\Phi}(s_J^{\Phi}) = \begin{cases} D & \text{if } s_J^{\Phi} \in acc_{\neg safe} \\ D\frac{m-m'}{m} & \text{otherwise} \end{cases} \tag{5.9}$$

where $D = |\mathcal{M}_J^{\Phi}|$ is a fixed penalty and equal to the number of states in the joint product model, m is the total number of tasks in the mission and m' is the number of tasks that have not been completed. m' is a reformulation of the *tasks* reward structure in Section 4.2. Lastly, Equation (5.4) is modified to use D_J^{Φ} as the cap:

$$V_n(s_J^{\Phi}) \leftarrow \min(D_J^{\Phi}(s_J^{\Phi}), \arg\min_{a \in A_J^{\Phi}} \sum_{s' \in S_J^{\Phi}} \delta(s_J^{\Phi}, a, s')[c(s_J^{\Phi}, a, s') + V_{n-1}(s')]) \tag{5.10}$$

where cost replaces the reward and the maximisation is replaced by a minimisation.

In words, $D_J^{\Phi}(s_J^{\Phi})$ is D for all states that violate the safety specification regardless of the number of tasks. In all other states $D_J^{\Phi}(s_J^{\Phi})$ the cost is relative to the number of completed tasks. These states include accepting states of $\mathcal{M}_J^{\Phi}$ (where $m' = m$ and therefore $D_J^\Phi(s_J^\Phi) = 0)$, dead-end states detected using single-robot solutions and states where all robots are in the designated failure state.

Minimising cost instead of maximising task reward incorporates the cost of actions in the objective. This encourages robots to distribute tasks evenly so as to minimise the sum of all robot action costs. If the objective was to simply maximise task reward, each robot might choose to do as many of the tasks as possible until it failed. Once it failed, the next robot could continue to do the remaining tasks, passing execution to subsequent robots when it failed. This is because the objective of maximising task reward does not take into account the cost of robot actions. However, the cost structures C_J^Φ and D_J^Φ can be used together to encourage task reward maximisation and team cost minimisation.

We can use LRTDP to generate a solution for $SSP(\mathcal{M}_J^\Phi)$ such that we minimise total cost including C_J^Φ and D_J^Φ i.e. $E_{\pi_J^\Phi}^{\min}(cumul_{C_J^\Phi \cup D_J^\Phi}^{acc_{\neg safe}})$. This generates a *feasible* policy for the problem defined in Section 4.2 i.e. a multi-robot task allocation and planning problem that maximises task reward. However, it does not generate an *optimal* policy. In fact the use of a cost structure instead of the tasks reward structure means that an extra model checking step is required in order to provide an exact expectation on the number of completed tasks following this joint policy.

Algorithm 6 THTS with Model Checking

```
function THTS(SSP MDP $SSP(\mathcal{M}_J^\Phi)$, timeout $T$)
    ▷ Initial action selection using Equation (5.7)
    ▷ Subsequent action selection using greedy action selection
    ▷ Dead-ends detected using Equation (5.8)
    ▷ Terminal states penalised using Equation (5.9)
    ▷ Backups using Equation (5.10)

    $n_0 \leftarrow getRootNode(SSP(\mathcal{M}_J^\Phi))$
    while $n_0$.notSolved() & time()$< T$ do
        visitDecisionNode($n_0$)
    end while

    ▷ Extracting Policies using Depth First Search (DFS)

    $\pi_g \leftarrow DFS(n_0, as_{greedy})$    ▷ greedy action selection
    $\pi_{mv} \leftarrow DFS(n_0, as_{mv})$    ▷ most visited action selection

    ▷ Calculate expected number of completed tasks under each policy
    $V^{\pi_g}(\overline{s}_J^\Phi) = E_{\mathcal{M}_J^\Phi}^{\pi_g}(cumul_{tasks}^{acc_{\neg safe}})$
    $V^{\pi_{mv}}(\overline{s}_J^\Phi) = E_{\mathcal{M}_J^\Phi}^{\pi_{mv}}(cumul_{tasks}^{acc_{\neg safe}})$
end function
```

Algorithm 6 shows the outermost function of the THTS modified to include this model checking step. Note that extracting the policy from the LRTDP search graph can be done using various action selection strategies. Traditionally the action selection strategy for extracting the policy is the same as the one used in the search. However, for scenarios where the search algorithm has not converged, other action selection strategies can be used.

Borrowing from Monte-Carlo Tree Search action selection strategies [Bro+12], we use the *most visited* action selection strategy. It has been shown empirically that when the search has not converged choosing the most visited action is usually better [Bro+12; Hua11]. We also use the greedy action selection strategy to extract the policy. It selects actions greedily based on their Q-values. This is the action selection strategy used during the search as well. The algorithm used to extract these policies is a simple depth first search starting at the root node, using the chosen action selection strategy to generate children.

Once the polices have been extracted policy evaluation (see Definition 5) is used to compute the expected number of completed tasks for each policy.

Remark 13 (Dealing with unexplored states). In scenarios where the search has not converged, not all states in the extracted policy have been visited or explored. Our aim is to provide a quantitative value for the expected number of tasks that can be completed by this policy. However, for unexplored states, it is not possible to provide such a guarantee because the algorithm itself has not been able to generate a policy for these states. Since LRTDP is an *anytime* algorithm, it is possible to replan from these unexplored states. However, since we plan offline within a given time limit i.e. we generate the policy before execution, this is not a viable option. Therefore, when such a state is seen during policy evaluation, it is treated as an absorbing state. This makes the guarantee conservative.

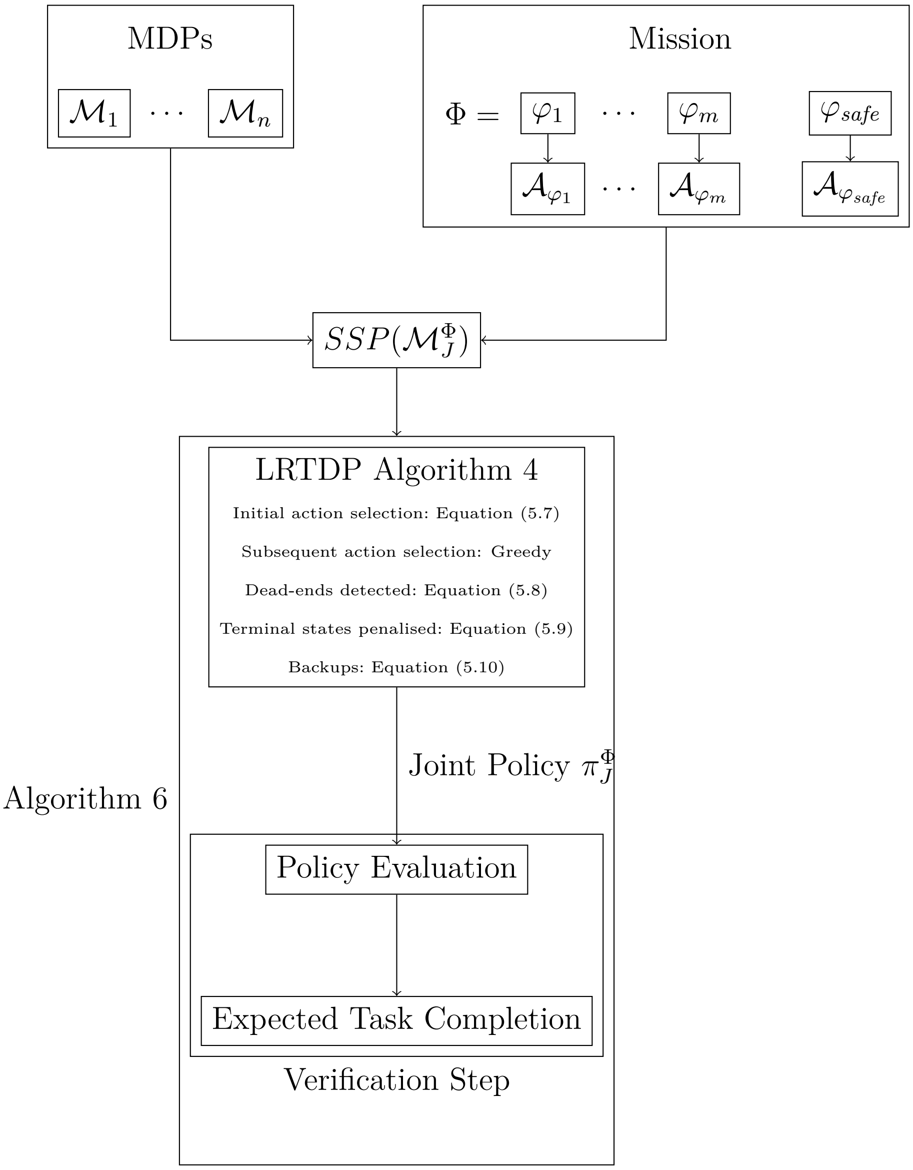
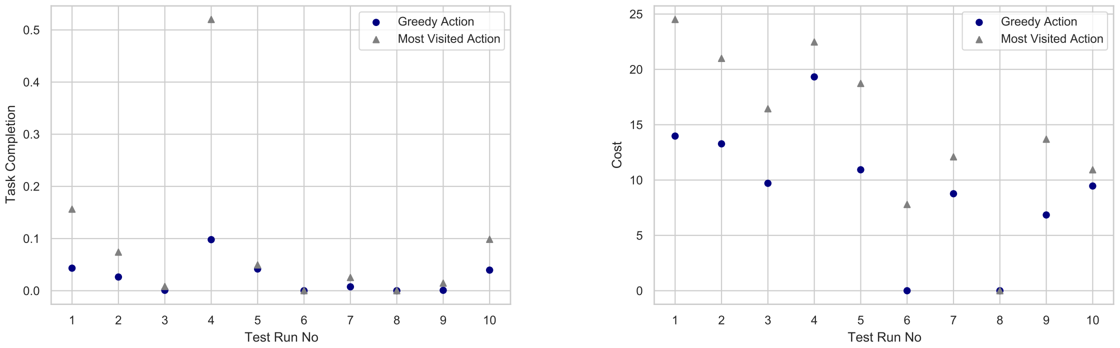

MDPs
$\mathcal{M}_1$ ··· $\mathcal{M}_n$
Mission
$\Phi =$ φ_1 ··· φ_m φ_{safe}
$\mathcal{A}_{\varphi_1}$ ··· $\mathcal{A}_{\varphi_m}$ $\mathcal{A}_{\varphi_{safe}}$
$SSP(\mathcal{M}_J^\Phi)$
LRTDP Algorithm 4
Initial action selection: Equation (5.7)
Subsequent action selection: Greedy
Dead-ends detected: Equation (5.8)
Terminal states penalised: Equation (5.9)
Backups: Equation (5.10)
Joint Policy π_J^Φ
Algorithm 6
Policy Evaluation
Expected Task Completion
Verification Step

Figure 5.14: An overview of LRTDP. Robot models and LTL automata are combined to create a joint product model. This is converted to an SSP (see Definition 29). The SSP is fed to Algorithm 6. Two different action selection methods are used. Equation (5.7) is used to select actions for the first time. Greedy action selection is used at all other times. The backup function used (Equation (5.10)) includes the relative penalty for terminal states (Equation (5.9)). The timeout is set to 2 hours. Once the joint policy is generated using depth first search (see Algorithm 6), a verification step takes place which gives us an exact guarantee on the team plan.

5.4 Results

We evaluated LRTDP for our SSP MDP on a grid-like warehouse environment which is described in Section 4.3.3, Figure 4.5c. We ran the algorithm 10 times with the number of robots set to 4, the number of tasks set to 4 and all actions in 90 percent of the locations led to the designated failure state for each robot. In fact the experimental setup[1] is the same as that described in Section 4.3, i.e. the PRISM Model Checker was extended to include Algorithm 6 and the test environments were the same as in Section 4.3.3. For LRTDP we set a timeout of 2 hours. As a result we do not show any results on the time taken to compute a policy since LRTDP did not converge within 2 hours for any of our experiments. We refer to this time as the computation time (for the policy) in future chapters.



(a) Expected task reward

(b) Expected cost

Figure 5.15: Expected task reward and cost for policies generated using LRTDP with a 2 hour timeout for 10 randomised scenarios with 4 tasks and 4 robots and 90% failstates. The policy was extracted using two different action selection strategies: greedy action selection and most visited action selection. The expected cumulative task reward and cost guarantee of both policies is shown.

Figure 5.15 shows the expected cumulative task completion and cost values output by evaluating the policy from LRTDP with a 2-hour limit. This warehouse environment had 123 locations. With 4 robots, 4 tasks, 1 safety task the number of states in the joint product MDP was $123^4 \times 2^5 \approx 7 \times 10^9$ since the DFA for each task had 2 states. Since

[1] The implementation can be found here:https://github.com/fatmaf/prism/tree/arm64

the grid is fully connected there are many potential solutions to the problem which is why the search does not terminate even after 2 hours. For this reason, the policy extracted using the most visited action selection strategy is able to provide a better performance guarantee (Figure 5.15). This is in line with the empirical evidence in [Hua11] which sees the most visited action selection strategy outperform the greedy action selection strategy when search has not converged.

Figure 5.15b shows the expected accumulated cost corresponding to each run from figure Figure 5.15a and is presented for completeness. Notice that the difference between the expected task reward for the greedy action selection and the most visited action selection does not mean that the difference between the expected cost for these will follow a similar trend. For example for the second run, the expected task rewards are close to each other but the expected cost is further apart. For the fourth run, the expected task rewards are much further but the expected costs are much closer. This can be explained by the randomisation of tests. Since all tests are randomised, no two tests have exactly the same initial locations for robots, tasks or locations that lead to the failure state. Therefore, different scenarios have different costs and expected task reward.

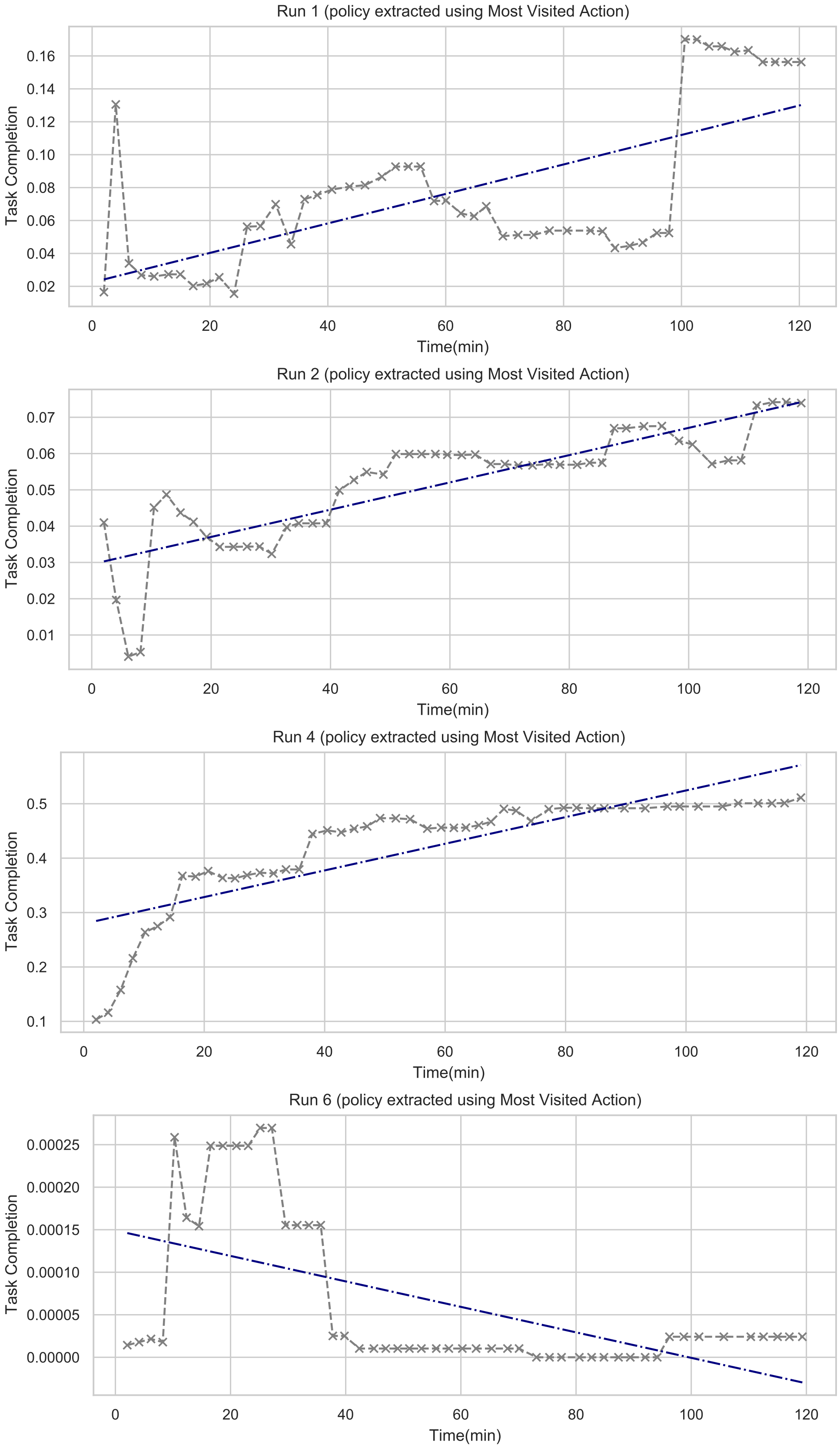


Figure 5.16: Expected task reward at various intervals for a selection of the tests in Figure 5.15. Since task reward is not the objective optimised in LRTDP, it does not improve monotonically. The change line shows the trend of the task completion reward.

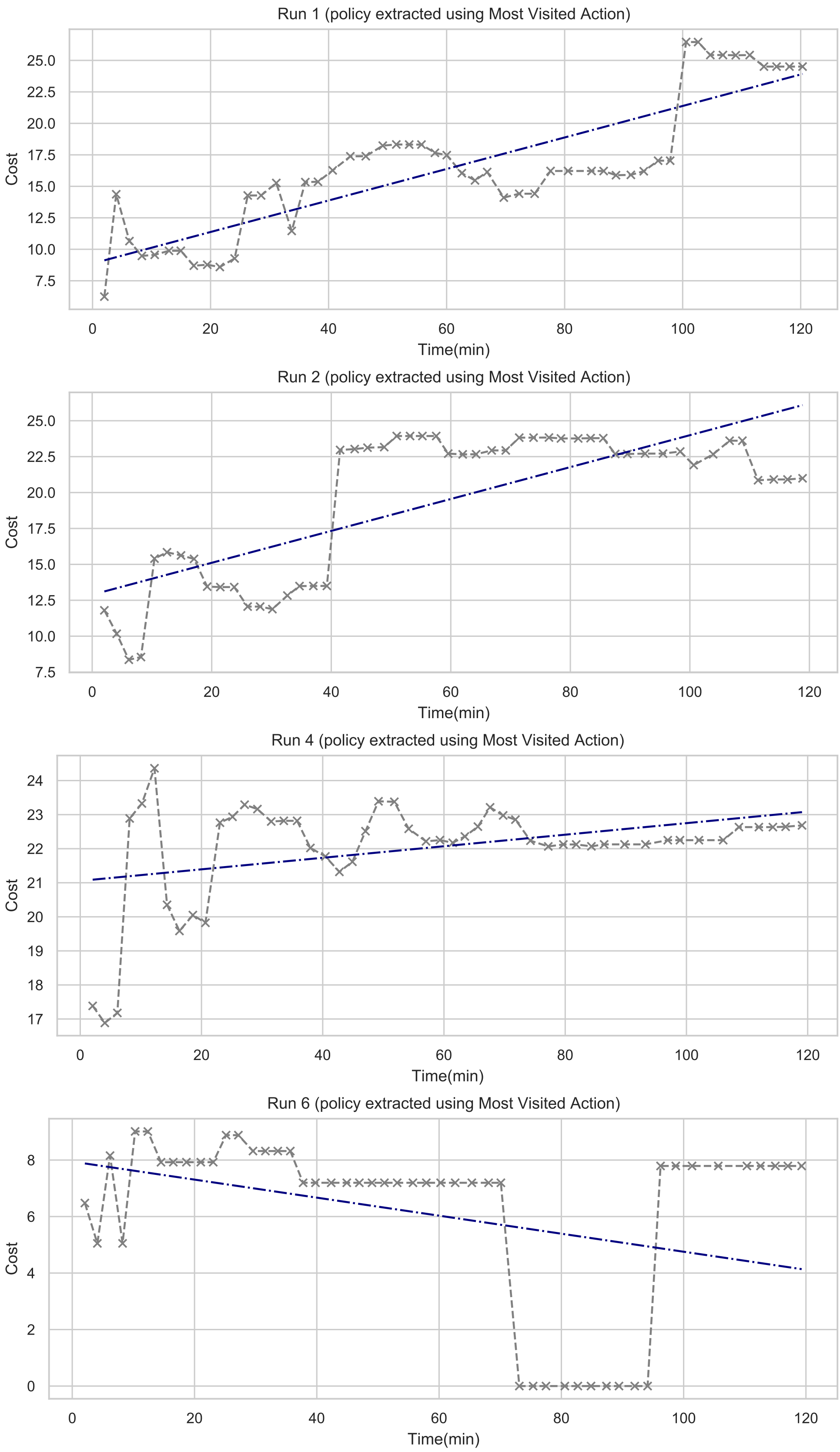


Figure 5.17: Expected cost at various intervals for a selection of the tests in Figure 5.15. The cost is dependent on the task reward. The change line shows the trend of the cost.

This is explained further by Figure 5.16 which shows the expected task reward at various intervals. The policy is extracted using greedy action selection and evaluated using policy evaluation. As mentioned earlier there is no direct correlation between the cost function in Definition 29 and that of the task reward. However, there is a correlation between the trends for expected task reward and the expected cost for a particular run. Figure 5.17 shows the expected cost at various time intervals. The trends in cost match those in the task reward. As the task reward increases, so does the cost. This can be explained by the fact that as the team completes more tasks, it visits more locations which increases the cost.

Furthermore the use of cost function instead of a task reward in the SSP MDP, $SSP(\mathcal{M}_J^{\Phi})$ means that LRTDP does not strictly improve the expected task reward. This behaviour is expected as the objectives for policy evaluation differ from the objectives of the SSP MDP. After extracting the policy at a certain time interval, we evaluate the policy for expected task reward and expected cost. The cost structure used during policy evaluation does not penalise absorbing states because the primary objective is that of maximising task reward which assigns a reward of 0 to non-goal absorbing states. In fact, our policy evaluation method matches NVI Algorithm 2.

Summary

- In this chapter, we extended Labelled Real Time Dynamic Programming (LRTDP), a sampling-based heuristic search algorithm to the problem of maximising expected number of tasks for a multi-robot team under uncertainty.

- We used single-robot solutions to create a rollout policy which we used for selecting actions in new states. Such an approach has not been applied to LRTDP to the best of our knowledge.

- We showed that it is possible to simultaneously allocate tasks and generate plans using such an approach, bypassing cycle detection due to zero-reward cycles which may have a large overhead. In order to apply LRTDP, we modified our MMDP formulation from one of maximising expected task reward to one of minimising cost, using a novel cost function that penalises states based on the LTL automata.

- Our approach extracts a guarantee on the expected number of completed tasks as a quantitative property of the joint policy.
- As expected, our results demonstrated that when the search has not converged choosing the most visited action to extract the policy outperforms choosing the greedy action to extract the policy.
- Indirectly our results also indicated that for large fully connected environments, with our formulation of the MMDP for LRTDP, it is not possible to expect anytime behaviour, which is one of LRTDP's strengths.

In the next chapter we use a separated task allocation and planning approach to solve the same problem and evaluate its performance with LRTDP.

Chapter 6

Auction-Based Task Allocation and Planning

As we saw in Chapter 5 even when using sampling-based approaches simultaneous task allocation and planning is computationally expensive. Breaking down the problem reduces the complexity and allows for a more tractable solution. Task allocation itself takes up an increasing amount of resources as the number of robots and tasks grows. The number of ways of allocating m tasks to n robots is a Stirling number of the second kind, exponential in the number of robots and tasks [Zlo06]:

$$S(m,n) = \frac{1}{n!}\sum_{j=0}^{n}(-1)^{(n-j)}\begin{pmatrix} m \\ n \end{pmatrix} j^{m}$$

In this chapter, we decouple task allocation from task planning and consider the use of auction-based methods for tasks allocation. Once tasks have been allocated, each robot must plan for itself. This takes up more resources. Furthermore, task allocation in such scenarios should be such that the robots are able to optimise for their given objectives during planning. In fact many auctioning-based task allocation approaches use planning to inform the auction itself e.g. [HK13]. Others use an approximate plan to inform the auction e.g. [Lag+04; KM06; Car+20]. Therefore, task allocation and planning are intrinsically linked.

This chapter begins by discussing some task allocation techniques which are generally used in conjunction with planning. It then describes sequential single-item auctioning. This auctioning mechanism is then applied to the problem of allocating tasks to a multi-robot team with a LTL mission specification which we formalised in Section 4.2. Finally we show results of this decoupled approach using sequential single-item auctioning for task allocation and value iteration (NVI) for planning. This approach is compared to LRTDP from Chapter 5.

6.1 Auction-Based Task Allocation

The problem in Section 4.2 can be categorised as a single-task robot, single-robot task, time-extended assignment problem using the taxonomy in [GM04]. The authors in [GM04] state that the problems in this category are strongly NP-hard. This means that the exponential space of combinations of task allocations renders an enumerative solution intractable. Therefore, optimal solution approaches such as those that use linear programming e.g. [Kuh55] are not feasible for such problems. On the other hand, auctioning-based approaches to task allocation have shown near optimal performance with far fewer resources [Koe+06]. Note that such approaches are also referred to as market-based since each robot can be seen as purchasing a task considering its resources.

Though auction-based task allocation requires a centralised auctioneer, i.e. a system that decides which bid to accept, it can be used to provide a semi-decentralised approach to task allocation in a multi-robot system. Each robot can bid on a task, calculating the bid separately from other robots and speeding up the task allocation process.

[Sch+15] compares four different mechanisms for task allocation of which three are auction-based: Round-robin, Ordered single-item auction, sequential single-item auction and Parallel single-item auction. Round-robin allocation, allocates tasks to robots in a round robin fashion. Ordered single-item auctioning offers the same task to all robots and chooses the one with the best bid for that task. This process is repeated until all tasks have been allocated. sequential single-item auctioning allows all robots to bid on any of the tasks in the task set. In each round one task is allocated to a single robot, based on the best bid. Like sequential single-item auctions, Parallel single-item auctions, allow all robots to bid on any of the tasks in the task set. However, all tasks are allocated in one single round, with each task going to the robot that placed the best bid for it. [Sch+15] shows that plans obtained after task allocation using sequential single-item auctions generally have lower costs than those using other mechanisms. However, in scenarios where robots are spread out such that task allocation is not complex, sequential single-item auctions take up more computation resources than other mechanisms. This is because sequential single-item auctions, auction one task at a time and in these scenarios are similar to a round-robin scheme.

[Koe+06] analyses sequential single-item auctions for task allocation problems with the objective of minimising the team's cost. The team's cost is considered to be the sum of individual robot costs. It shows that when optimising for the sum of costs, sequential single-item auctioning is at most two times the optimal cost. For small models, the optimal cost can be obtained by using a simultaneous task allocation and planning algorithm similar to LRTDP in Chapter 5. [Koe+06] also shows that the total number of bids made using sequential single-item auctions is bounded by $|m||n|$, where m is the number of tasks and n is the number of robots. While other auction methods are able to generate solutions that are closer to the optimal, the number of bids and runtime of these algorithms is higher than sequential single-item auctions [Koe+06]. Therefore, sequential single-item auctions are able to scale with the number of robots and tasks, avoiding an exponential increase in the state-action space. This makes them a viable choice for a decoupled auctioning and planning approach. Furthermore, as shown in [Sch+15], sequential single-item auctions perform well in most situations which also makes them suitable for our diverse set of test environments (see Section 4.3.3). As formalised in Section 4.2, our aim is to maximise the task reward, providing an exact quantitative value on the expected task completion of a multi-robot plan. To this end, we choose sequential single-item auctions since these are able to generate close to optimal task allocations, are suitable for many scenarios and are easy to implement compared to other auctioning techniques. The next section describes a sequential single-item auction [Koe+06] applied to the problem of task reward maximisation in more detail. It also shows how plans can be generated once tasks have been auctioned.

6.2 Sequential single-item Auctioning and Planning

In summary, a sequential single-item auction (Algorithm 7) begins with all tasks waiting to be allocated to robots (Line 2). Each robot bids on an unallocated task (Line 10). Each robot's bid is the maximal improvement it can offer to the team's objective if it were to perform that particular task in addition to the tasks it was previously allocated. The robot with the overall best bid is allocated the corresponding task (Line 13). This concludes one round of the auction. The process is repeated in subsequent rounds until all tasks are allocated. Therefore, sequential single-item auctions ensure that a robot is not idle if it is able to improve the team's performance.

In Section 4.2 we formalised the problem of multi-robot task allocation and planning with the objective of maximising the expected number of tasks completed by the team. In the following text, we introduce notation relevant to the application of a sequential single-item auction to the aforementioned problem for the purpose of task allocation and planning.

6.2.1 Single-robot Policies

Let $Set(\Phi) = \{\varphi_1, \ldots, \varphi_m, \varphi_{safe}\}$ be a function that takes the mission tuple and outputs the set of tasks in the mission.The set of tasks to allocate $T = Set(\Phi) \setminus \{\varphi_{safe}\}$. We exclude the safety task because all robots must not violate this task. As a result it is automatically assigned to each robot. With slight abuse of notation, we denote the set of

Algorithm 7 Sequential single-item Auctioning and Planning

```
function SSIPLAN($Set(\Phi) = \{\varphi_1, \ldots, \varphi_m\, \varphi_{safe}\}, \mathcal{M}_0, \ldots, \mathcal{M}_n$)
    $T \leftarrow \{\varphi_1, \ldots, \varphi_m\}$
                                            ▷ set of unallocated tasks $T = Set(\Phi) \setminus \varphi_{safe}$
    for $i \in N$ do
        $\Phi^i = \{\varphi_{safe}\}$
                                            ▷ assign safe to each robot
    end for
    while $T \neq \emptyset$ do
        for $i \in N$ do
            $bid(i) = \max_{\varphi \in T} bestbid(i, \varphi)$
                            ▷ find the best bid for a robot using Equations (6.5) and (6.6)
        end for
        $winningbid = \max_{i \in N} bestbid(i)$
                                            ▷ find the winning bid
        $r \leftarrow \arg\max_{i \in N} bestbid(i)$
                                            ▷ find the robot with the winning bid
        $\varphi \leftarrow$ task with winning bid
        $\Phi^r \leftarrow \Phi^r \cup \varphi$
                            ▷ add that corresponding task to that robot's task set
        $T \leftarrow T \setminus \varphi$
                            ▷ update the set of tasks to exclude the new task
    end while
    for $i \in N$ do
        $\pi^{\Phi i} \leftarrow$ NestedValueIteration($\mathcal{M}_i{}^{\Phi i}$, [$tasks$, $costs$])
                ▷ for each robot, generate the policy for its task set Algorithm 2
    end for
end function
```

This algorithm chains task allocation through auctioning with policy generation using Nested Value Iteration. Here, the sequential single-time auctioning algorithm from [Koe+06] is adapted to the problem of task reward maximisation using costs as tie-breakers. Then NVI (see Algorithm 2) is used to generate individual robot plans.

tasks allocated to a robot i as

$$\Phi^i \subseteq Set(\Phi) = \{\varphi \mid \varphi \in T\} \cup \{\varphi_{safe}\}. \tag{6.1}$$

Informally, we say that Φ^i is robot i's mission specification i.e. Φ^i contains the set of tasks that are allocated to robot i. Φ^i is a subset of the set of tasks in the original mission Φ. We also ensure that for any robot i, Φ^i always includes the safety task φ_{safe}. This is because as stated in Section 4.2 the safety task applies to all robots. Recall from Definition 15 that the MDP $\mathcal{M}_i$ is used to model robot i. Given $\mathcal{M}_i$ and Φ^i, the local product MDP of robot i is:

$$\mathcal{M}_i{}^{\Phi^i} = \mathcal{M}_i \otimes \mathcal{A}_{\varphi_1} \otimes \cdots \otimes \mathcal{A}_{\varphi_k} \otimes \mathcal{A}_{\varphi_{\neg safe}} \mid \{\varphi_1, \ldots, \varphi_k, \varphi_{safe}\} = \Phi^i \tag{6.2}$$

Robot i generates its policy for Φ^i by applying value iteration (NVI, see Algorithm 2) to its local product MDP $\mathcal{M}_i{}^{\Phi^i}$ as explained in Definition 14 i.e.

$$\pi^{\Phi^i} = NestedValueIteration(\mathcal{M}_i{}^{\Phi^i}, [\mathit{tasks}, \mathit{costs}]), \tag{6.3}$$

where *tasks* is the task reward function which counts the number of tasks completed using the corresponding LTL automata states and *costs* is a cost function. Recall that the objective of NVI is to maximise the expected task reward using cost as a tie breaker. The expected number of tasks completed by robot i for Φ^i under the policy π^{Φ^i} can be calculated as in Definition 5 using the reward structure *tasks*. We use $E^{\Phi^i}_{\mathit{tasks}}$ to denote this value. Similarly, we use $E^{\Phi^i}_{\mathit{cost}}$ to denote the expected cost under the same policy. Note that both $E^{\Phi^i}_{\mathit{cost}}$ and $E^{\Phi^i}_{\mathit{tasks}}$ can be obtained by using the policy,π^{Φ^i} generated by NVI over the local product MDP, $\mathcal{M}_i{}^{\Phi^i}$ with the initial state $\overline{s^{\Phi^i}}$.

6.2.2 Robot Bids

The first step in a sequential single-item auction is that of each robot bidding on one of the tasks in the task set T (Algorithm 7, Line 10). In order to bid for a task φ, robot i uses NVI to generate a policy for its assigned tasks and the new task i.e. $\Phi^i \cup \varphi$. It can then use this policy to calculate the expected task reward and expected cost. Robot i's bid for task φ is then a column vector:

$$robotbid(i, \varphi) \in \mathbb{R}^{2\times 1} = \begin{bmatrix} robotbid(i, \varphi)_{tasks} \\ robotbid(i, \varphi)_{cost} \end{bmatrix} = \begin{bmatrix} E^{\Phi^i \cup \varphi}_{tasks} - E^{\Phi^i}_{tasks} \\ E^{\Phi^i \cup \varphi}_{cost} - E^{\Phi^i}_{cost} \end{bmatrix} \tag{6.4}$$

Each robot chooses a task $\varphi \in T$ such that φ offers the maximal improvement to the team's objective among all unallocated tasks in T. In line with our formulation of NVI (Equations (4.6) and (4.7), Section 4.4) we choose the task that offers maximum improvement in the task reward, using cost as a tie breaker.

Let RB_i denote the set of bids that robot i can place. Let the set of bids with the highest task reward generated by robot i from the set of unallocated tasks T, be

$$RB_i^{tasks} = \{robotbid(i, \varphi') \in RB_i \mid \varphi' = \arg\max_{\varphi \in T} robotbid(i, \varphi)_{tasks}\} \tag{6.5}$$

Then the robot's best bid is the one from the above mentioned set that has the lowest cost i.e.

$$bestbid(i) = robotbid(i, \varphi') \in RB_i^{tasks} \mid \varphi' = \arg\min_{\varphi \in T} (robotbid(i, \varphi)_{cost}). \tag{6.6}$$

To reiterate, each robot i, generates a bid for each task φ in the task set. The bid a robot makes is a column vector consisting of two values, one for the improvement in the team's task reward $robotbid(i, \varphi)_{tasks}$ and one for the improvement to the team's cost $robotbid(i, \varphi)_{cost}$. To choose its best bid, each robot i, looks at the values $robotbid(i, \varphi)$ for all the tasks in the task set. It chooses the task which provides the highest increase in the team's task reward. There may be multiple tasks that provide the same highest increase in task reward, i.e. the set RB_i^{tasks} has more than one element. In these scenarios the robot uses cost as a tie-breaker, i.e. from the bids that provide the highest increase in task reward, the robot chooses the task which gives the lowest increase in cost, $\arg\min_{\varphi \in T} (robotbid(i, \varphi)_{cost})$. To simplify all this notation, in Algorithm 7 this is shown as $bestbid(i) = max_{\varphi \in T} bid(i, \varphi)$ in Line 10.

6.2.3 Winning Bid

The auctioneer, which in our case is Algorithm 7 chooses the winning bid from the set of bids placed by all robots using the same criterion as in Equations (6.5) and (6.6), adapting it to the set of bids each robot places. In words, the winning bid is the bid that offers the most improvement to the team's task reward. If multiple bids offer the same task reward improvement, then the improvement to the cost is used as a tie breaker. In Algorithm 7 this is shown as $bid = max_{i \in N} bestbid(i)$ in Line 13 where N is set of robot indices. Once the winning bid is chosen, the corresponding task is added to the winning robot's task set, Φ^i and removed from the set of unallocated tasks T (Line 20). Once all the tasks are allocated the auction is complete.

This allows the robots to generate a policy for their task sets using value iteration as described in Section 6.2.1, Equation (5.6) (Lines 23 to 26).

Remark 14. The policy generation step can be skipped with a little book keeping. As we saw in Section 6.2.2 each robot uses NVI to generate its bid. Instead of regenerating the policy in Lines 23 to 26, the policy generated during the robot's bid calculation can be used.

Example 13 (Task Allocation using sequential single-item auctions)**.** Consider the scenario from Figure 4.3 reproduced here. The set of unallocated tasks is $T = \{\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4), \mathtt{F}\, v_3, \mathtt{F}\, v_6\}$. The safety task $\mathtt{G}\, \neg v_7$ is assigned to all robots i.e. $\Phi^1 = \{\mathtt{G}\, \neg v_7\}$, $\Phi^2 = \{\mathtt{G}\, \neg v_7\}$ and $\Phi^3 = \{\mathtt{G}\, \neg v_7\}$. The MDP for this example deviates from the MDP fragment shown in Figure 4.2 on page 59 for the sake of simplicity.

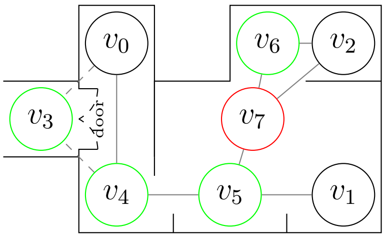


Figure 4.3: (repeated from page 62) Example topological map. Robots start in v_0, v_1, v_2; locations to visit in green and to avoid in red; mission specification $\Phi = \langle \mathtt{F}(v_5 \wedge \mathtt{F}\, v_4), \mathtt{F}\, v_3, \mathtt{F}\, v_6, \mathtt{G} \neg v_7 \rangle$.

Each action a robot takes leads to the designated failure state with probability 0.2. The only exceptions to this are the checkdoor action in states v_0 and v_4 and the actions that lead to the state v_3 if the door is open. The probability of the door being open is 0.8. The cost of all actions is 1.

Auction Round 1 In the first round of the auction, each robot bids on one of the tasks in T. Each robot chooses the task to bid on based on the criterion in Equations (6.5) and (6.6). For example, robot 1 in location v_0 can get a maximum expected task reward of 0.8 for $\mathtt{F}\, v_3$, 0.64 for $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$ and 0 for $\mathtt{F}\, v_6$. Robot 1's best bid is for $\mathtt{F}\, v_3$. Robot 2 in location v_1 can get a maximum expected task reward of 0.64 for $\mathtt{F}\, v_3$, 0.64 for $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$ and 0 for $\mathtt{F}\, v_6$. The cost for $\mathtt{F}\, v_3$ is 2.952 while the cost for $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$ is 2.44. Therefore, its best bid is for $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$. Robot 3 can get a maximum expected task reward of 0.8 for $\mathtt{F}\, v_6$ and 0 for the others. The individual robot bids are a column vector with the first value corresponding to the task reward and the second to the cost:

$$bestbid(1) = \begin{bmatrix} 0.8 \\ 1.8 \end{bmatrix}, bestbid(2) = \begin{bmatrix} 0.64 \\ 2.44 \end{bmatrix}, bestbid(3) = \begin{bmatrix} 0.8 \\ 1 \end{bmatrix}$$

The bids for robots 1 and 3 are tied with regard to the task reward, so the cost is used as a tie-breaker. Therefore the winning bid is the bid by robot 3. Its task assignment is updated to $\Phi^3 = \{\mathtt{F}\, v_6, \mathtt{G} \neg v_7\}$. This task is removed from the set of unallocated tasks. The team's expected task reward is 0.8 with an expected cost of 1.

Auction Round 2 In the second round all robots bid on the remaining tasks $T = \{\mathtt{F}(v_5 \wedge \mathtt{F}\,v_4), \mathtt{F}\,v_3\}$. For this round robot 3 can not bid on either of the tasks in T since this means violating the specification i.e. visiting v_7. Therefore, robot 1's bid of 0.8 for $\mathtt{F}\,v_3$ is the winning bid. Its task assignment is updated to $\Phi^1 = \{\mathtt{F}\,v_3, \mathtt{G}\,\neg v_7\}$. The team's expected task reward is 1.6 with an expected cost of 2.8.

Auction Round 3 In the third round there is only one remaining task i.e. $T = \{\mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)\}$. The best bids from each robot are:

$$bestbid(1) = \begin{bmatrix} 0.4096 - 0.8 \\ 3.6 - 1.8 \end{bmatrix} = \begin{bmatrix} -0.3904 \\ 1.8 \end{bmatrix}, bestbid(2) = \begin{bmatrix} 0.64 \\ 2.44 \end{bmatrix}, bestbid(3) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that robot 1's bid for $\mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)$ is the increase it offers to the team's overall expected task reward. To do this we save the expected task reward for robot 1's previous task allocation.

Robot 2 has not been allocated any tasks and so its bid for the task is 0.64. Robot 2 is assigned the task with the final expected task reward for the team as 2.24. Note that when a robot is not able to complete any tasks e.g. robot 3, our implementation disregards its bid.

As we saw in Example 13, the team's overall task reward is in fact the sum of the task rewards of each robot. The team's cost is the sum of individual robot costs. This formulation combined with the auctioning procedure discourages robots from being idle if they are able to perform tasks.

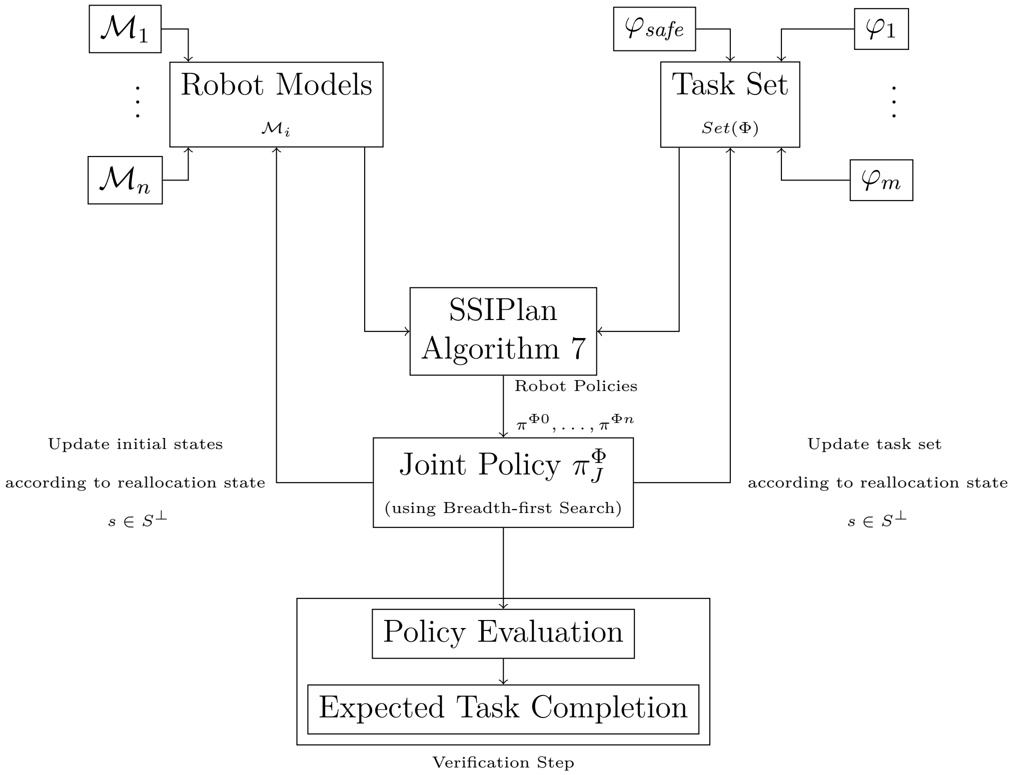


Figure 6.1: Outline of the overall approach. Tasks φ_i are sent to Algorithm 7 along with the robot MDPs, $\mathcal{M}_i$ for each robot r_i. Algorithm 7 allocates tasks in the task set T to robots using sequential single-item auctioning. Then policies, $\pi^{\Phi i}$ for each robot r_i are generated using value iteration (Algorithm 2). These policies are used to build a synchronised joint policy π_J^{Φ}. A reallocation state is chosen from π_J^{Φ}, the initial state of the robot MDPs are updated to represent the chosen reallocation state. The task set is also updated to remove any tasks that have been completed up until the chosen reallocation state. These are then sent to Algorithm 7 until no more reallocation states exist or the procedure is interrupted, at which point the current joint policy π_J^{Φ} is returned, along with the expected number of completed tasks.

6.3 Building a Joint Policy

Figure 6.1 provides an overview of the process of using Algorithm 7, described in the previous section, to iteratively generate a full joint team policy. Once tasks are allocated and individual robot policies generated as in Algorithm 7, a joint team policy must be created. This means that individual robot policies must be mapped to a joint team policy. Furthermore, there may be scenarios where individual robots fail. Since Algorithm 7 does not allow robots to communicate progress with each other, there needs to be a way to identify states where a robot fails and reallocate tasks in those states.

The following sections show how a joint policy is built for individual policies and how task reallocation takes place.

Using NVI we are able to get an action for each state that is reachable from the initial state. This results in a local policy $\pi^{\Phi^i} : S^{\Phi^i} \to A^{\Phi^i}$ where $s^{\Phi^i} = (s_i, q^{\Phi^i})$. The set of states in the joint product model is $S^{\Phi}_J = S^{\Phi^0} \times S^{\Phi^1} \times \dots S^{\Phi^n}$. The joint policy $\pi^{\Phi}_J : S^{\Phi}_J \to A^{\Phi}_J$ maps states in the joint product model to joint actions. Recall the projection function Definition 16 that projects a state from a joint MDP to the local MDP of robot i. Let ${[.]_i}^{\Phi^i} : S^{\Phi}_J \to S^{\Phi^i}$ be a projection function that maps states in the joint product MDP to the local product MDPs for each robot.

The joint action, a^{Φ}_J for the state $s^{\Phi}_J \in S^{\Phi}_J$ is:

$$a^{\Phi}_J = (\pi^{\Phi^1}({[s^{\Phi}_J]_1}^{\Phi^1}), \pi^{\Phi^2}({[s^{\Phi}_J]_2}^{\Phi^2}), \dots, \pi^{\Phi^n}({[s^{\Phi}_J]_n}^{\Phi^n})) \tag{6.7}$$

Remark 15 (Actions Modifying Global State Features). Recall from Section 4.2 that in our formulation we disallow any actions where more than one robot attempts to influence a global state feature.

Remark 16 (Communicating Global State Features). The joint policy operates on the multi-robot MDP as in Section 4.1.2. In order to build the joint policy, we assume that robots are able to communicate the values of global state features. This is inline

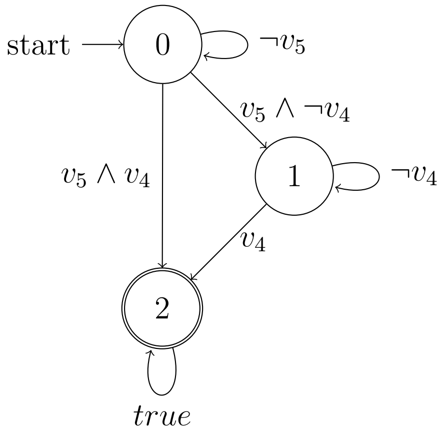
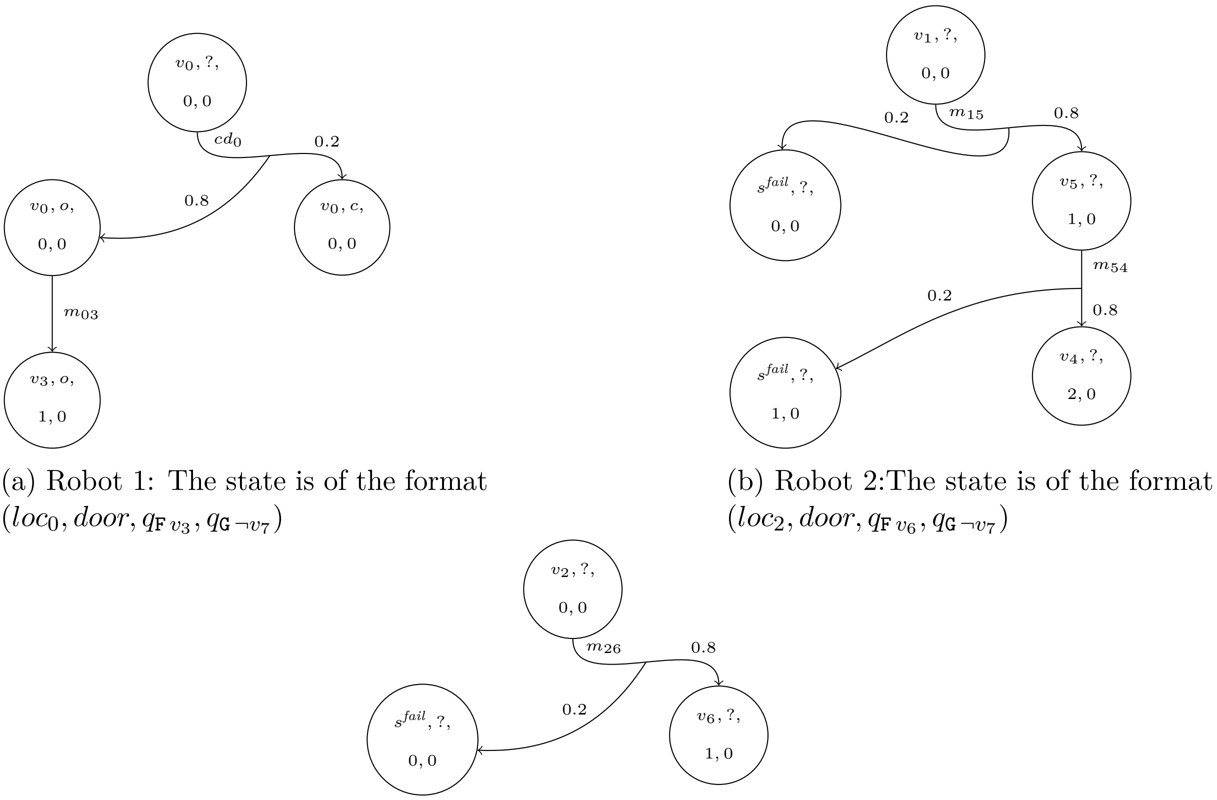


Figure 4.4: (repeated from page 63) The Deterministic Finite Automaton for the task $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$. The initial state is 0. The robot stays in this state until it visits v_5. Then it transitions to 1. It stays in this state until it visits v_4. Then it transitions to 2 which is the accepting state (denoted by the double border). The direct transition from 0 to 2 is not possible according to the topological map. This transition is automatically removed during the product construction.

with the assumptions in Section 4.2. In fact when a certain joint action is generated using Equation (6.7) this is done implicitly by generating the successors of this joint action in the joint state.

Example 14 (Joint Policy)**.** Figure 4.4, reproduced here, shows the Deterministic Finite Automaton (DFA) of the task $\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)$. The initial state of the DFA is 0 while the final state is 2. The DFAs of the other tasks are omitted since they only contain two states, the initial state 0 and the final state 1.

Figure 6.2 shows fragments of the policies for all three robots generated in Example 13. These are fragments because in our implementation we perform NVI not just from the initial state but from all states. The reasons for this will become clear in this example.



(a) Robot 1: The state is of the format $(loc_0, door, q_{\mathtt{F}\, v_3}, q_{\mathtt{G}\, \neg v_7})$

(b) Robot 2:The state is of the format $(loc_2, door, q_{\mathtt{F}\, v_6}, q_{\mathtt{G}\, \neg v_7})$

(c) Robot 3: The state is of the format $(loc_2, door, q_{\mathtt{F}(v_5 \wedge \mathtt{F}\, v_4)}, q_{\mathtt{G}\, \neg v_7})$

Figure 6.2: Fragments of policies for all three robots.

Each robot's state includes its local state and the state of the door (which is a global state). ? means the state of the door is unknown, o means the door is open and c means it is closed. It also includes the state corresponding to the DFA of the task it was allocated and the state of the safety task DFA.

The initial joint state is $(v_0, v_1, v_2, ?, 0, 0, 0, 0)$. According to Equation (6.7) the joint action is

$$\begin{aligned} a_J^\Phi &= (\pi^{\Phi^1}([s_J^\Phi]_1{}^{\Phi^1}), \pi^{\Phi^2}([s_J^\Phi]_2{}^{\Phi^2}), \pi^{\Phi^3}([s_J^\Phi]_3{}^{\Phi^3})) \\ &= (\pi^{\Phi^1}(v_0, ?, 0, 0), \pi^{\Phi^2}(v_1, ?, 0, 0), \pi^{\Phi^3}(v_2, ?, 0, 0)) \\ &= (cd_0, m_{26}, m_{15}) \end{aligned}$$

The successors of this joint state are shown in Figure 6.3. In fact we can extract the joint policy using breadth first search or depth first search with joint actions as described in Equation (6.7). The search backtracks when the policy has no further actions for a state and it terminates after each encountered state has been visited once.

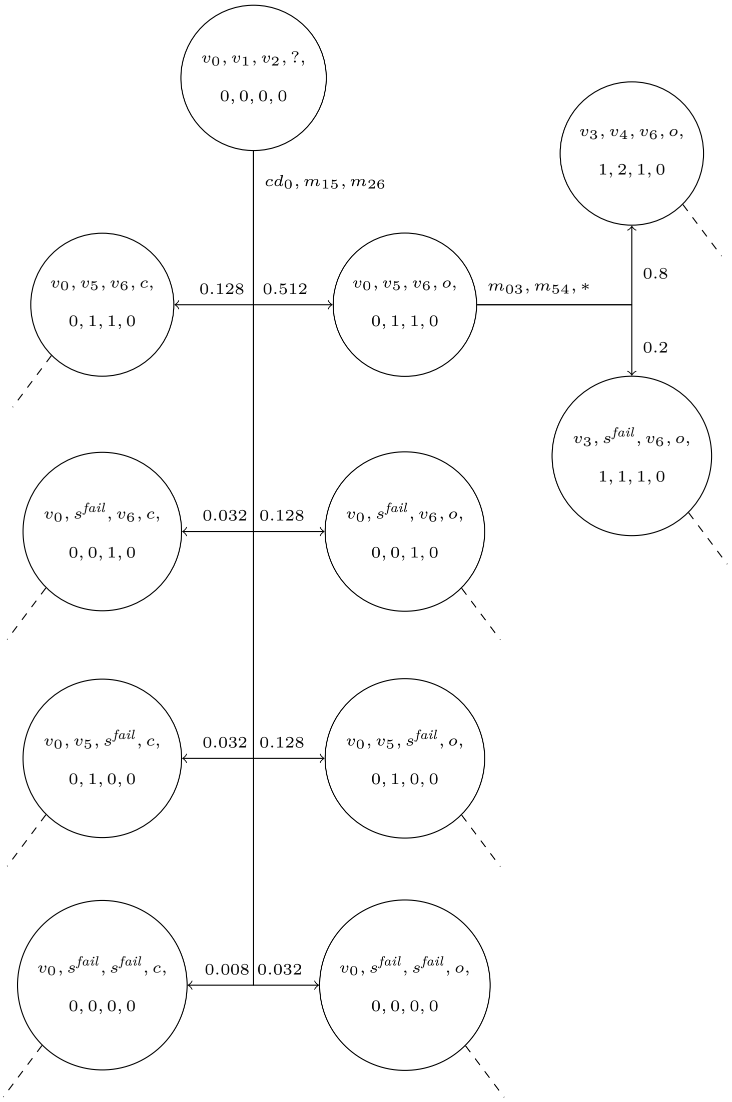


Figure 6.3: A fragment of the joint policy

Note that the action for state $(v_0, v_5, v_6, o, 0, 1, 1, 0)$ contains a $*$ which is an arbitrary action because robot 2 has completed its task i.e. it has reached an accepting state (see Definition 24, page 90). However, the other robots have not reached an accepting state and have possible actions. As a result, we continue the search.

The policy fragments shown in Figure 6.2 do not show what action robots 2 and 3 should choose when the door is open, i.e. the value of the corresponding state variable is o. This is why our implementation of NVI considers all possible locations as initial states. The complete policy has an action for robots 2 and 3 when the door is open. Therefore, we are able to generate a joint action for this state. This action leads to two states. State $(v_3, v_4, v_6, o, 1, 2, 1, 0)$ is an accepting state because all robots have completed their task sets. Therefore, the search backtracks from this state as there is nothing else to explore. State $(v_3, s^{fail}, v_6, o, 1, 1, 1, 0)$ is not an accepting state, because robot 2 failed midway through its task. In this state, none of the robots have any associated actions, therefore the search also back tracks from this state. However, it makes a note of this state and the probability of reaching this state from the initial state. We see how to deal with states like this one in the next section.

6.4 Reallocation through Replanning

Recall from Sections 4.2 and 4.5, that our objective is to generate a *full* joint policy for the team, including the reallocation of tasks when a robot fails. In the MDP model described in Definition 15, when a single robot fails, it can no longer perform any actions. Therefore, any allocated tasks this robot failed to do, need to be reallocated. With the help of the joint policy, we can identify these reallocation states as shown in Figure 6.1. Example 14 mentioned the use of breath first search or depth first search to extract the joint policy. During this extraction, states where more than one robot has failed can be isolated.

Example 15 (Reallocation States)**.** Continuing Example 14, one state where a robot failed was, $(v_3, s^{fail}, v_6, o, 1, 1, 1, 0)$. In this state, none of the robots had any actions but an accepting state was not reached. Therefore, this state can be isolated as a reallocation state.

There may be other states where some robots do have actions e.g. $(v_0, s^{fail}, v_6, o, 0, 1, 1, 0)$. Since the door is open robot 1's policy maps this state to the action m_{03} but robot 2 has failed and therefore, its task is incomplete. As a result, $(v_0, s^{fail}, v_6, o, 0, 1, 1, 0)$ is also a reallocation state.

Definition 30 (Reallocation states)**.** We define a set of reallocation states as

$$S^{\perp} = \{s_J^{\Phi} \in S_J^{\Phi} \mid \pi^{\Phi^i}([s_J^{\Phi}]_i^{\Phi^i}) = \perp \text{ for any } 1 \leq i \leq n\}$$

In words the set of reallocation states is all joint states where *any* robot does not have an action.

Remark 17. According to Definition 30, reallocation states are not restricted to designated failure states. In fact, the set of reallocation states includes all joint states where any robot is in a *non-goal absorbing* state (see Definition 25, page 90). As a result, tasks assigned to robots in such states can be reassigned and no robot is left idle if it is able to do more tasks. This definition also applies to states where the safety specification has been violated as well as states where all robots have failed. A simple check for such states can be used to exclude them from this set of states. In fact in such states tasks will not be reallocated since none of the robots are able to move.

In order to reallocate tasks in a reallocation state, the initial states of all robot models are updated to reflect the chosen reallocation state. After this a new auction is initiated i.e. Algorithm 7 is called again. The unallocated tasks in this auction is the set of incomplete tasks including those that have been previously assigned.

Example 16 (Reallocation)**.** Continuing with Example 14, in state $(v_0, s^{fail}, v_6, o, 0, 1, 1, 0)$, robot 1 has failed, robots 0 and 2 have not. Therefore $(v_0, s^{fail}, v_6, o, 0, 1, 1, 0)$ is a reallocation state. The initial states of the robot MDPs are updated as (v_0, o) for robot 0, (s^{fail}, o) for robot 1 and (v_6, o) for robot 2. Robot 2 has completed its assigned task therefore the set of unassigned tasks does not include this task and is $T = \{\mathtt{F}\,v_3, \mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)\}$. Note that robot 1 failed midway through the task, therefore the initial state of DFA corresponding to this task must be updated to reflect this. Figure 4.4 reproduced in Example 14, Section 6.3 shows the DFA for this task. Its initial state is changed from 0 to 1.

Auction Round 1 In the first round of the auction, each robot bids on one of the tasks in $T = \{\mathtt{F}\,v_3, \mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)\}$. Each robot chooses the task to bid on based on the criterion in Equations (6.5) and (6.6). Note that robot 3 can not bid on any of the tasks in T because the safety specification $\mathtt{G}\,\neg v_7$ would have to be violated for it to achieve any of these tasks. Robot 2 can not bid on any of these tasks since it is in its designated failures state s^{fail}. Robot 1's best bid is for $\mathtt{F}\,v_3$ which it can complete with a task reward of 1 and a cost of 1. Therefore robot 1 is assigned this task. The set of unallocated tasks is updated to $T = \{\mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)\}$.

Auction Round 2 In this round, there is only one task and one robot that is able to achieve this task. Therefore, robot 1 is assigned this task. Note that the DFA state corresponding to this task is updated to 1, meaning that robot 0 will only need to visit v_4 in order to complete this task.

Policy As noted earlier, robots use NVI to generate the expected reward and cost for a task. The policy from this can be reused (skipping the policy generation step in Algorithm 7).

In Example 16 the reallocation state was chosen arbitrarily to illustrate how Algorithm 7 is used to reallocate and replan. However, a more informed choice of reallocation states to replan for would allow for the generation of a partial joint plan in scenarios when time is limited. For this purpose we choose to order reallocation states by the probability of reaching such a state from the initial state in the joint policy,π^{Φ}_{J} i.e. we order states $s^{\Phi}_{J} \in S^{\perp}$ in decreasing order of $Pr^{\pi^{\Phi}_{J}}_{\mathcal{M}^{\Phi}_{J},\bar{s}^{\Phi}_{J}}(\mathtt{F}\,s^{\Phi}_{J})$. In practice this is achieved by adding the reallocation states to a priority queue. The probability of reaching a reallocation state can be calculated during the joint policy building step. As a result of this prioritisation we can tradeoff a complete joint policy in favour of reduced computation time, excluding reallocation states that have a reachability probability below a certain threshold.

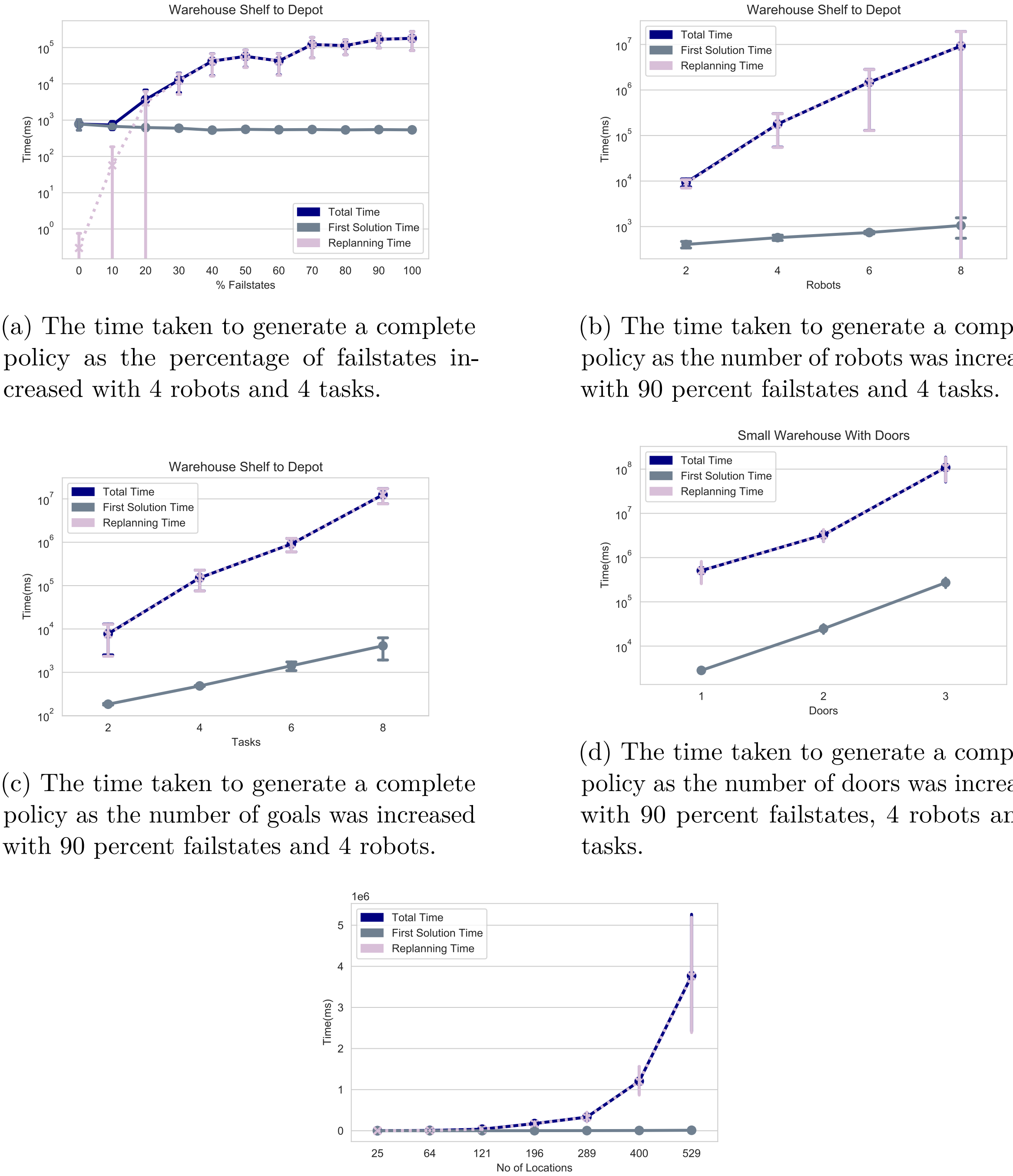
6.5 Results

In this section we present the results of auctioning and planning followed by a comparison with LRTDP from Chapter 5. We tested the approach on the warehouse scenario described in Section 4.3.3, Figure 4.5c and the grid scenario in Figure 4.6a. The experimental setup[1] used has already been described in Section 4.3 and was used to test NVI on the joint multi-robot MDP as well as LRTDP from the previous chapter. The difference here is that while we put a 2 hour limit on the computation time for LRTDP, we did not do so here, since as we will see later, the computation time was on average less than 2 hours.

For each test, we randomised the initial positions of robots, goals and locations that led to the designated failure state with a probability of 0.2. For each test set, we ran 10 tests. In the rest of this text we refer to locations which have transitions to the designated failure state as *failstates*.

Figure 6.4 shows the computation time in milliseconds as the number of robots, tasks, doors, locations and percentage of failstates are varied. As expected, the general trend is an increase in the computation time as each variable increases. Figure 6.5 shows boxplots depicting the increase in the number of replanning attempts as the percentage of failstates increases or the total number of locations (in the topological map) increases. It also shows the decrease in task completion with the increase in percentage of failstates.

[1] link to the code for auctioning here:https://github.com/fatmaf/prism-school-pc



(a) The time taken to generate a complete policy as the percentage of failstates increased with 4 robots and 4 tasks.

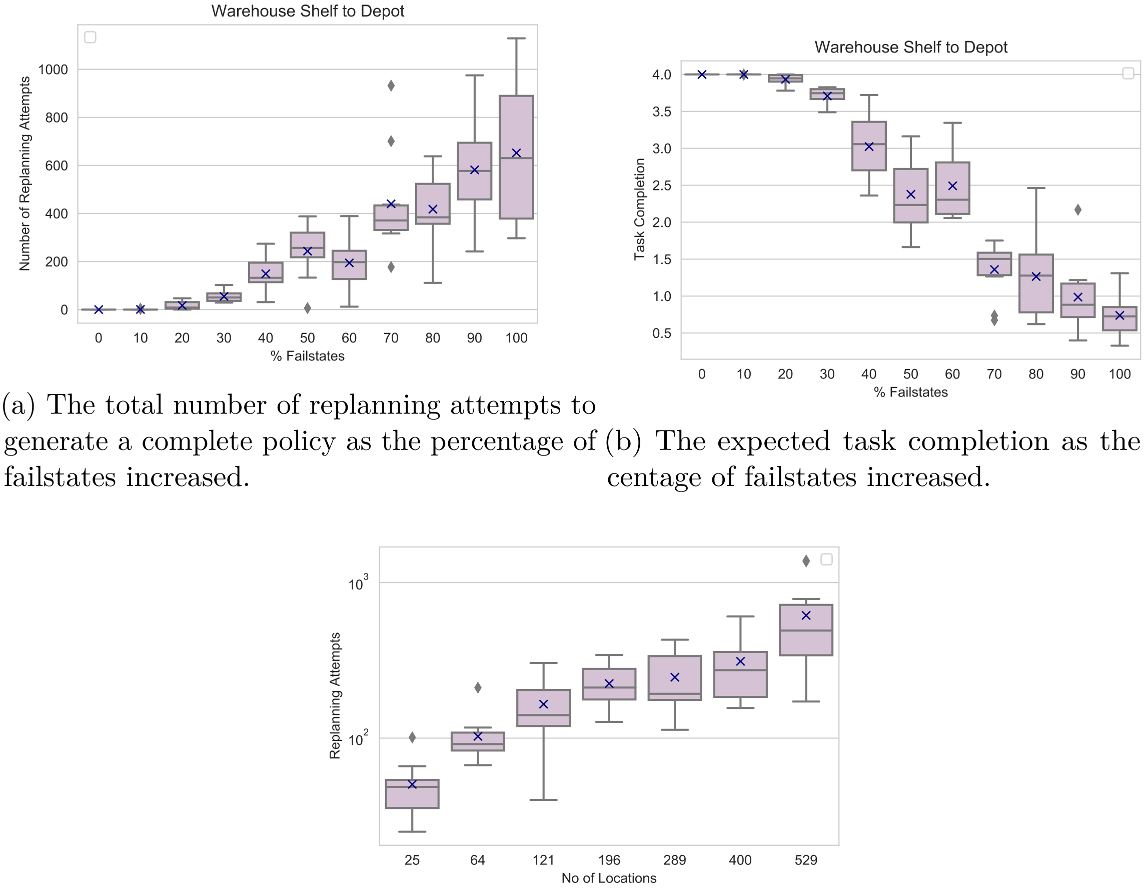
(b) The time taken to generate a complete policy as the number of robots was increased with 90 percent failstates and 4 tasks.

(c) The time taken to generate a complete policy as the number of goals was increased with 90 percent failstates and 4 robots.

(d) The time taken to generate a complete policy as the number of doors was increased with 90 percent failstates, 4 robots and 4 tasks.

(e) The time taken to generate a complete policy as the number of locations was increased with 90 percent failstates, 4 robots and 4 tasks.

Figure 6.4: This figure shows the trend of the computation time for auctioning and planning as certain elements are varied. The figures presented are all pointplots where the vertical bars indicate the standard deviation and the points indicate the means. The number of robots was set to 4 (unless varied), the number of tasks was set to 5 (unless varied) and the percentage of failstates was set to 90 percent (unless varied). The number of doors i.e. global states was 0 except for the tests in Figure 6.4d.



(a) The total number of replanning attempts to generate a complete policy as the percentage of failstates increased.

(b) The expected task completion as the percentage of failstates increased.

(c) The total number of replanning attempts made as the number of locations was increased.

Figure 6.5: Task completion and number of replanning attempts: Boxplots, where the crosses indicate the mean values and whiskers show the interquartile range and dots show outliers. A randomised set of 10 scenarios were used to generate these results. The number of robots was set to 4 and the number of tasks was set to 5. The number of doors i.e. global states was 0.

In Figure 6.4a, the increase in the percentage of failstates means that more transitions lead to the designated failure state for each robot. This in turn, increases the number of times robots fail and replan as shown in Figure 6.5a. As robots fail more often, the expected task completion reduces (Figure 6.5b). In fact, Figure 6.4a shows that as the percentage of failstates increases, more time is spent in replanning. The replanning time for up to twenty percent failstates is considerably lower than the time taken to generate the first solution. This is because a low percentage of failstates means fewer transitions in the MDP. As expected, smaller MDPs are solved more quickly.

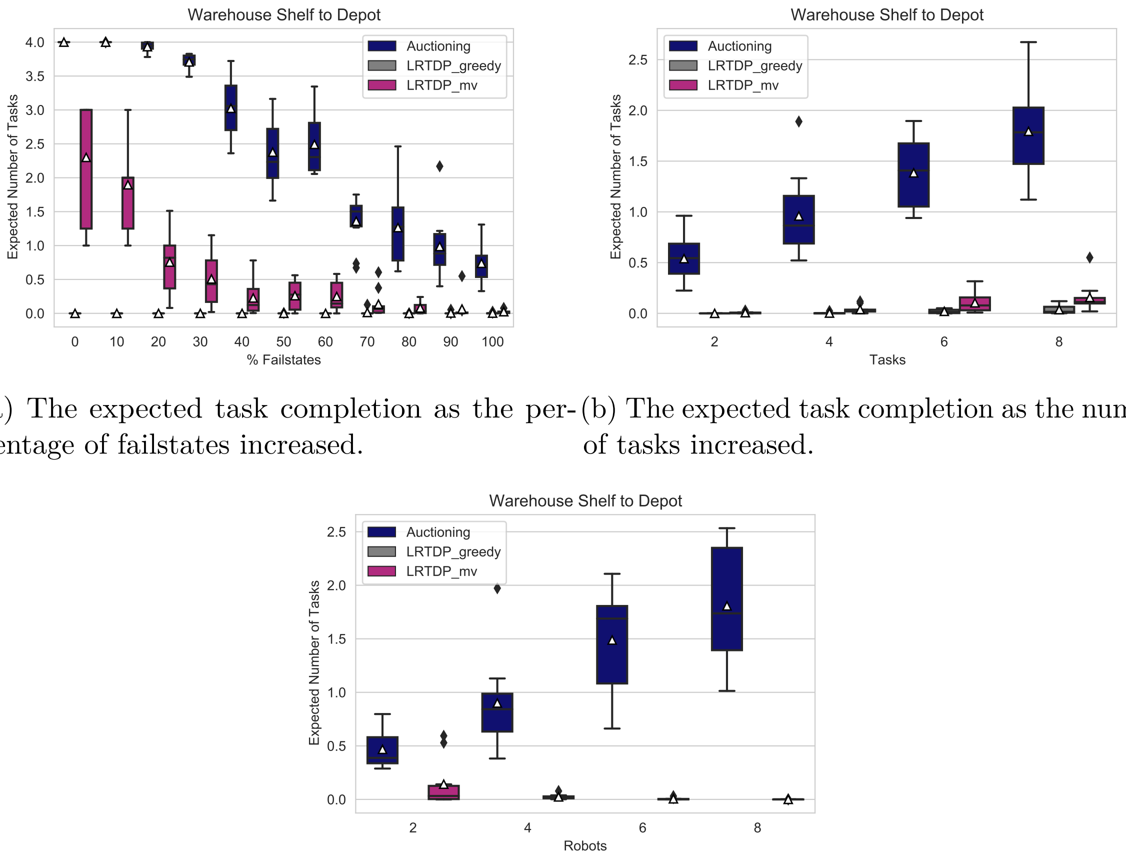
In Figures 6.4b to 6.4e the percentage of failstates is fixed at 90 percent for all tests. We can see that the time to generate the first solution is always considerably less than the time to generate subsequent solutions for reallocation states. The close to exponential increase in the computation time as the number of robots, tasks, doors and locations increases is due to the increase in the size of the underlying product MDPs. As the number of robots increase, more MDPs need to be solved and more reallocation states are discovered. Similarly, as the number of tasks increases, the size of the product MDPs increases exponentially. The size of the policies increases since the robots must travel to more states in order to achieve the increasing tasks. This again increases the number of reallocation states.

Increasing the number of global states, doors in our case, also increases the size of the underlying MDPs (Figure 6.4d). A door can take on three states and therefore each location in the map now has three different configurations; one with the door's state not known, one with the door open and one with the door closed. The increase in the model states due to this can not be avoided. Furthermore planning for each robot separately means that this increase in state applies to all robot models. In fact each robot generates policies during the auctioning process and this increase in state affects the computation time too. The evidence for this can seen by comparing the time for the marker for 90% failstates in Figure 6.4a and the time for the marker for 1 door in Figure 6.4d.

The increase in the number of locations in the underlying MDPs happens more slowly than that in Figures 6.4a to 6.4d. The increase in the number of tasks and global states causes the state space of the MDPs to grow much faster. This is because each task added results in a DFA with at least 2 states. The resulting product MDP is at least twice the size. The global states have a similar effect. However, an increase in the number of locations, increases the size of the product MDP by the number of states in the task DFAs and any global states. This explains the slower increase in computation time.

6.5.1 Comparison with LRTDP

We also compared the expected task completion from LRTDP (Chapter 5) to that from auctioning and planning. Since LRTDP has a built in time out, we simply set the time out to match the time taken to solve the same scenario using either auctioning and planning or the algorithm presented in the next chapter. We set the time out equal to the bigger duration of the two. Figure 6.6 shows the results of this set of experiments as the number of robots, tasks and percentage of failstates is varied. The time-out for LRTDP was either equal to or greater than the time the auctioning approach took. The experiments were performed using the warehouse Shelf to Depot environment with 4 robots, 4 tasks and 90 percent failstates, unless they were varied. Compared to LRTDP, auctioning and planning is able to achieve a higher task completion. This is because the size of the joint multi-robot product MDP is much bigger than the smaller MDPs that need to be solved with auctioning and planning. A large number of states means that more states need to be visited in order to get complete the mission or a task. Therefore, the algorithm does not perform as well as auctioning and planning. The differences between $LRTDP_{mv}$ and $LRTDP_{greedy}$ were explained in Chapter 5.



(a) The expected task completion as the percentage of failstates increased.

(b) The expected task completion as the number of tasks increased.

(c) The expected task completion as the number of robots increased.

Figure 6.6: Boxplots, where the triangles indicate the mean values and whiskers show the interquartile range and dots show outliers. A randomised set of 10 scenarios were used to generate these results. The number of robots was set to 4 and the number of tasks was set to 5. The number of doors i.e. global states was 0. $LRTDP_{greedy}$ is the policy extracted by using greedy action selection. $LRTDP_{mv}$ is the policy extracted by selection the most visited action in each state of the search tree.

Summary

- In this chapter, we demonstrated a decoupled approach to solving the problem of maximising expected task reward for a multi-robot team. Tasks were allocated using sequential single-item auctions. In order to bid on a task, each robot calculated the expected reward for the task using the policy generated by value iteration. This policy was then reused, removing the need for an extra planning step.
- We also demonstrated how to build a joint policy and use it to identify states where reallocation of tasks and therefore replanning must be done.
- We showed that the auctioning and planning approach takes less time to compute a feasible solution than LRTDP from Chapter 5.

In the next chapter we present a method that does not decouple task allocation from planning, as done in this chapter, but also does not work on the full joint multi-robot product MDP as in Chapter 5.

Chapter 7

Simultaneous Task Allocation and Planning

The problem of determining a joint robot policy, as formulated in Section 4.2, can be tackled in a number of ways. In Chapter 5 we presented a sampling-based approach where task allocation and planning were performed on the joint multi-robot model. In Chapter 6 we presented an auctioning-based approach where task allocation and planning were decoupled and a joint multi-robot plan was generated afterwards. In this chapter, we propose an approach called *simultaneous task allocation and planning under uncertainty (STAPU)*, which combines the processes of task allocation and task planning in order to exploit information about individual robot plans into the allocation process. STAPU lies between the methods in the previous chapters. It allows for simultaneous task allocation and planning but does not use the full joint multi-robot model.

This chapter begins with a formal description of the various components used in STAPU. It then describes the task reallocation mechanism. The chapter ends with an evaluation of STAPU in relation to the *sampling-based search* i.e. LRTDP (Chapter 5) and *Auctioning and planning* (Chapter 6) approaches.

7.1 Methodology

Inspired by the (non-probabilistic) approach of [SBD18b], we propose a method that initially plans on a *sequential* model of the robots, avoiding the construction of the fully synchronised joint model. This exploits the assumption that tasks have no interdependencies and can each be completed by a single robot. We consider each robot independently in turn, allowing it to (simultaneously) choose tasks to undertake and decide how best to complete them. To achieve this, we build and solve a *team MDP*, which can be viewed as a sequence of models for each individual robot. More precisely, each individual model for robot i is a *local product MDP*, encoding the dynamics of the robot (from local MDP $\mathcal{M}_i$), and the definitions of the tasks, along with the extent to which they have been completed so far (using DFAs $\mathcal{A}_\varphi$ for the formulas φ in the mission specification Φ).

The local product MDPs are joined using *switch transitions*, which represent changes in the team model from robot i to $i + 1$ (the next robot in the sequential model). These transitions are added in every state of robot i's model where no task has yet been started, or one has just been completed, as determined by whether the DFAs for each task are in their initial or accepting states. When a switch transition occurs, robot $i + 1$ begins in its initial state. The state of the DFAs remains the same, so that information about task completion is preserved. Considered sequentially, this model allows each robot a choice, before or after executing any task, as to whether it or the subsequent robots should tackle the unallocated tasks.

We find a sequential policy, by computing an optimal policy for the team MDP that maximises the expected number of tasks completed without violating the safety constraint. We then convert this sequential policy into a joint policy, where the robots execute concurrently. In situations where an action for this policy cannot be generated (e.g., because of robot failure), we perform a *replanning* process from that joint state. These steps are described in more detail in the following sections.

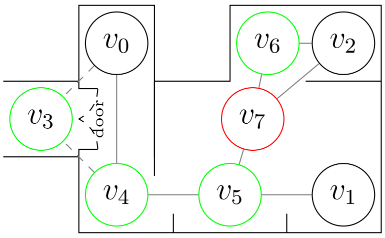


Figure 4.3: (repeated from page 62) Example topological map. Robots start in v_0, v_1, v_2; locations to visit in green and to avoid in red; mission specification $\Phi = \langle \mathtt{F}(v_5 \wedge \mathtt{F}\, v_4), \mathtt{F}\, v_3, \mathtt{F}\, v_6, \mathtt{G}\, \neg v_7 \rangle$.

7.1.1 Team MDP and Sequential Policy

The *team MDP* $\mathcal{M}_T^\Phi$ is built as the *union* of MDPs for the n robots, and models their joint behaviour in a *sequential* fashion. The MDP for each individual robot i is the local product MDP $\mathcal{M}_i^\Phi = \mathcal{M}_i \otimes \mathcal{A}_{\varphi_1} \otimes \cdots \otimes \mathcal{A}_{\varphi_m} \otimes \mathcal{A}_{\varphi_{\neg safe}}$, combining the local model $\mathcal{M}_i$ of robot i's behaviour with DFAs representing the satisfaction of the formulas in the mission $\Phi = \langle \varphi_1, \ldots, \varphi_m, \varphi_{safe} \rangle$.

States of the team MDP $\mathcal{M}_T^\Phi$ take the form (i, s_i^Φ) where i denotes the robot currently executing tasks and s_i^Φ is a state of the local product MDP $\mathcal{M}_i^\Phi$. So, $s_i^\Phi = (s_i, q_1, \ldots, q_m, q_{\neg safe})$ comprises a state $s_i \in S_i$ from the local MDP $\mathcal{M}_i$ for robot i and one for each of the DFAs $\mathcal{A}_{\varphi_1}, \ldots, \mathcal{A}_{\varphi_m}, \mathcal{A}_{\varphi_{\neg safe}}$.

Example 17 (Local Product MDP)**.** Consider the scenario from Figure 4.3 reproduced here. A fragment of the local MDP for one robot is shown in Figure 7.1.

The local product MDP for each robot contains the states of the robot, the state of the door and the states for the task. A fragment of the local product MDP is shown in Figure 7.2. If the robot starts in state $(v_0, o, 0, 0, 0, 0)$ and chooses action m_{03}, it moves to state $(v_3, o, 0, 1, 0, 0, 0)$ completing the task $\mathtt{F}\, v_3$. The DFA for $\mathtt{F}\, v_3$ has two states 0 which is the initial state and 1 which is the accepting state. The robot can not go back to a state where the task is not complete as seen by the actions available in $(v_3, o, 0, 1, 0, 0, 0)$, all of which lead to a state where the corresponding DFA variable matches the value of the accepting state.

Definition 31 (Team MDP)**.** Given a mission Φ and the local product MDPs $\mathcal{M}_i^\Phi =$

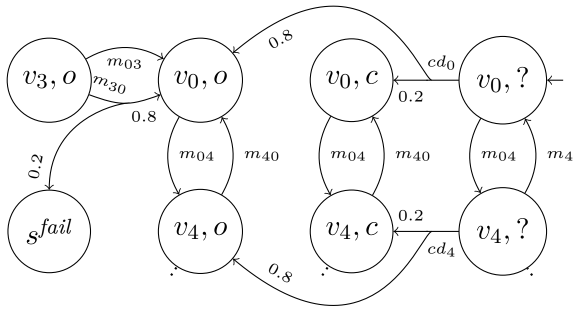


Figure 7.1: Fragment of an example local MDP for robot i corresponding to the map in Figure 4.3 (see Example 13). Each action leads to the designated failure state with probability 0.2 with the exception of the check door actions (cd_0, cd_4) and the actions $m_{04}, m_{40}, m_{03}, m_{43}$. The probability of the door being open is 0.8. The cost of all actions is 1.

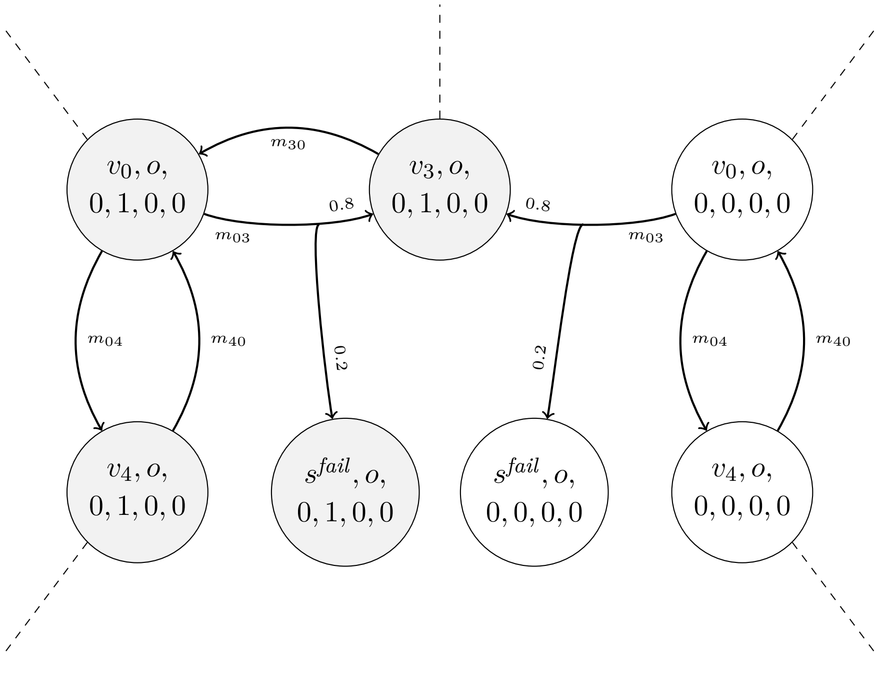


Figure 7.2: Fragment of the local product MDP for robot i. For simplicity, we show the a very small part which includes four states from the local MDP in Figure 7.1. Each state has the following variables $(v_i, door, q_{\mathtt{F}\,v_6}, q_{\mathtt{F}\,v_3}, q_{\mathtt{F}(v_5 \wedge \mathtt{F}\,v_4)}, q_{\neg(\mathtt{G}\,\neg v_7)})$ where v_i denotes the robot's position on the topological map and *door* denotes the state variable of the door. The shaded states are the ones where one of the tasks in the mission, $\mathtt{F}\,v_3$ has been completed.

$\langle S_i^\Phi, \overline{s}_i^\Phi, A_i, \delta_i^\Phi, AP^\Phi, Lab_i^\Phi \rangle$ for each robot i, the *team MDP* is defined as the MDP $\mathcal{M}_T^\Phi = \langle S_T^\Phi, \overline{s}_T^\Phi, A_T, \delta_T^\Phi, AP^\Phi, Lab_T^\Phi \rangle$ where:

- $S_T^\Phi = \bigcup_{i=1}^n (\{i\} \times S_i^\Phi)$
- $\overline{s}_T^\Phi = (1, \overline{s}_1^\Phi)$
- $Lab_T^\Phi(i, s_i^\Phi) = Lab_i^\Phi(s_i^\Phi)$
- $A_T = \{\zeta\} \cup \bigcup_{i=1}^n A_i$, where ζ labels a *switch transition;*
- δ_T^Φ is defined as follows. For action $a_i \in A_i$ of robot i, the team MDP mirrors the local product MDP:

$$\delta_T^\Phi((i, s_i^\Phi), a_i, (i, t_i^\Phi)) = \delta_i^\Phi(s_i^\Phi, a_i, t_i^\Phi), \tag{7.1}$$

where $s_i^\Phi, t_i^\Phi \in S_i^\Phi$ are states of the local product MDP for robot i. For switch transitions:

$$\delta_T^\Phi((i, s_i^\Phi), \zeta, (j, t_j^\Phi)) = 1 \tag{7.2}$$

if all of the following conditions hold:

(i) $i < n$ and $j = i + 1$;

(ii) local product MDP states $s_i^\Phi{=}(s_i, q_1, \ldots, q_m, q_{safe})$ and $t_j^\Phi{=}(t_j, q_1, \ldots, q_m, q_{safe})$ have the same DFA state components $q_1, \ldots, q_m, q_{safe}$;

(iii) either all DFA states $q_1, \ldots, q_m, q_{safe}$ are initial states, or at least one of them is a "new" accepting state (i.e., is an accepting state and has an incoming transition in $\mathcal{M}_i^\Phi$ from a state where it is not accepting);

(iv) t_j is the initial state in robot j's local MDP $\mathcal{M}_j$ i.e. $t_j = \overline{s_j} = \overline{(t^l, s_1^g, \ldots, s_k^g)}$ (from Definition 15) where s^l is the local state feature for robot j and $s_1^g, \ldots, s_k^g$ are its global state features (see Example 19).

For all other pair of states, $\delta_T^\Phi((i, s_i^\Phi), \zeta, (j, t_j^\Phi)) = 0$.

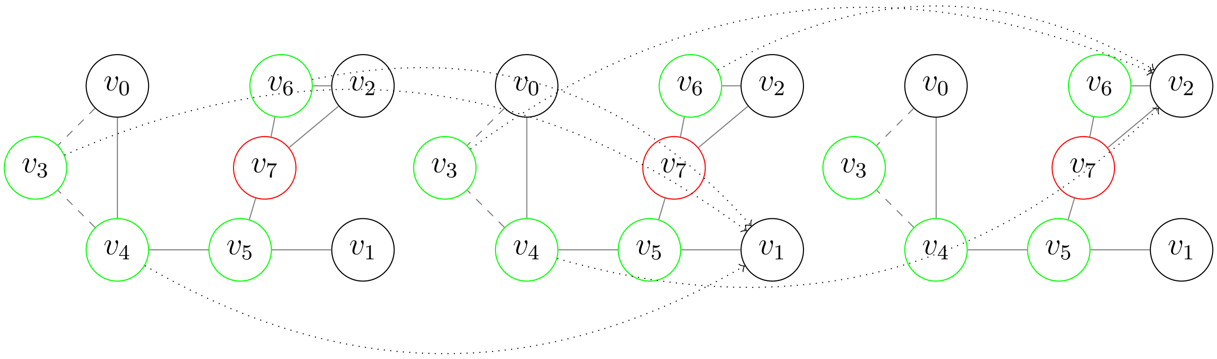


Figure 7.3: A simplified view of the team MDP using the topological map. The dotted edges represent switch transitions ζ. The automata state variables and global state variables have been removed. Robot 1 starts in v_0, robot 2 starts in v_1 and robot 3 starts in v_2. Note that for v_5 does not have an outgoing switch transition. This is because the corresponding task is not completed at v_5.

Example 18 (Team MDP)**.** Figure 7.3 shows a simplified view of the team MDP using the topological map. Since there are three robots there are three copies of the map. Each copy is only linked to the next copy. This link is through the switch transitions ζ that originate from states where tasks have been completed. Figure 7.4 shows a more detailed view of a fragment of the team MDP to explain this. Note that the only state with the switch transition is the one where $\mathtt{F}\, v_3$ has just been completed. Also note that there are no switch transitions from the designated failure state or the state where the safety task is violated.

Remark 18 (Switch Transitions and Failure States)*.* The primary objective of our problem is maximising the expected task reward. One way to achieve this is to have each robot attempt to satisfy the mission specification in turn. If the first robot fails, the second robot continues from then on. If the second robot fails, the next robot continues and so on. If the first robot does not fail, it completes the entire mission while the other robots in the team do nothing. This method does not allocate tasks to the team. It also does not minimise the number of steps in the plan. In fact, it defeats the purpose of a *robot team* altogether. If switch transitions originate from the designated failure states (s^{fail} from Section 4.1.1) then it is possible for robot 1 to attempt as many of the tasks as it

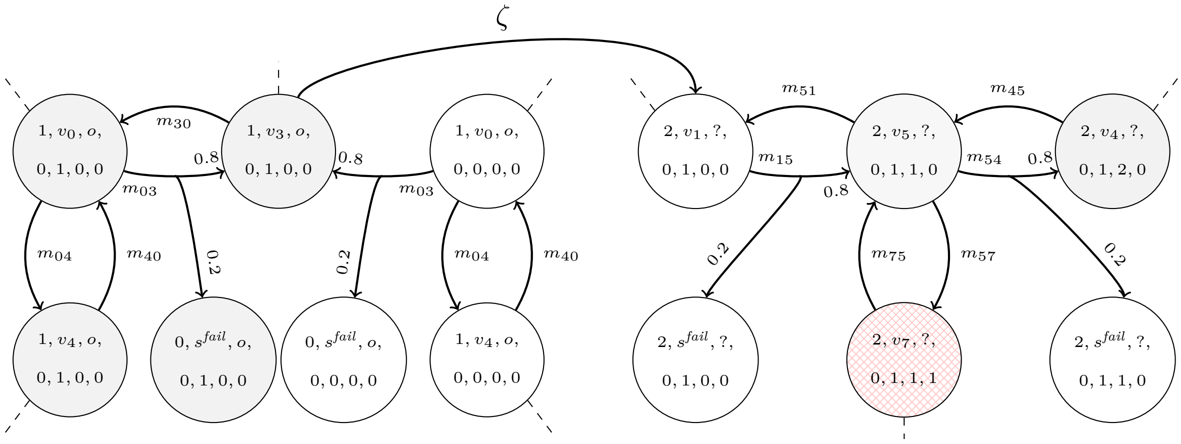

ζ
1, v_0, o, 0, 1, 0, 0
1, v_3, o, 0, 1, 0, 0
1, v_0, o, 0, 0, 0, 0
1, v_4, o, 0, 1, 0, 0
0, s^{fail}, o, 0, 1, 0, 0
0, s^{fail}, o, 0, 0, 0, 0
1, v_4, o, 0, 0, 0, 0
2, v_1, ?, 0, 1, 0, 0
2, v_5, ?, 0, 1, 1, 0
2, v_4, ?, 0, 1, 2, 0
2, s^{fail}, ?, 0, 1, 0, 0
2, v_7, ?, 0, 1, 1, 1
2, s^{fail}, ?, 0, 1, 1, 0
m_{30}
m_{03}
m_{04}
m_{40}
m_{51}
m_{15}
m_{45}
m_{54}
m_{75}
m_{57}
0.8
0.2

Figure 7.4: A fragment of the team MDP showing a switch transition. The states on the left belong to robot 1 whereas the states on the right belong to robot 2. The red state must be avoided as per the safety task.

can until it fails and then pass execution over to robot 2 and so on. Not adding switch transitions to these designated failure states ensures that this behaviour does not arise. Therefore, in our formulation we do not add switch transitions to designated failure states.

The team MDP encodes the execution of the tasks by the team, whilst avoiding the construction of the joint MDP. Note that, while the number of states in a joint MDP is exponential in the number of robots, for the team MDP it is linear. This will allow us to scale to much larger models, as will be highlighted in Section 7.2.

Our first step towards creating a joint policy is to construct a *sequential policy* π_T^Φ by solving the team MDP $\mathcal{M}_T^\Phi$. Since $\mathcal{M}_T^\Phi$ already includes the DFAs for the LTL formulas in the mission specification Φ, we generate the policy π_T^Φ using the same approach formalised in Section 4.2, i.e., finding an optimal policy that maximises the expected number of tasks completed without violating the safety condition:

$$E^{\max}_{\mathcal{M}_T^\Phi}(cumul^{acc_{\neg safe}}_{tasks}) \tag{7.3}$$

The atomic proposition $acc_{\neg safe}$ labels states where the DFA for $\neg\varphi_{safe}$ is in an accepting states; these are transferred to $\mathcal{M}_T^\Phi$ when copying the labelling from the local product MDPs

$\mathcal{M}_i^{\Phi}$ (see Definition 31) The reward structure *tasks* is defined exactly as in Section 4.2, by counting the number of tasks that are completed when executing a given transition. In practice, we solve this using Nested Value Iteration as described in Section 4.4 with a cost structure as a tie-breaker. In fact, this is the same method used to generate policies in the auctioning and planning approach from Chapter 6.

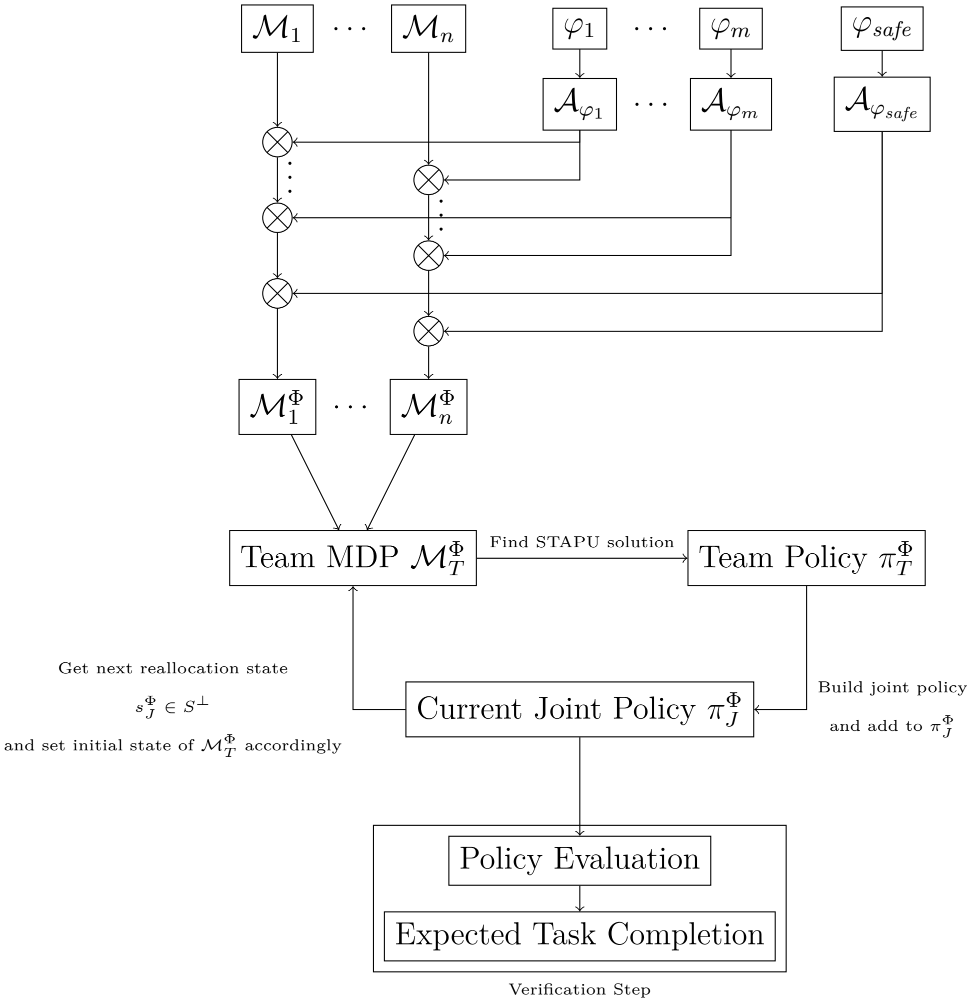

$\mathcal{M}_1$
$\cdots$
$\mathcal{M}_n$
φ_1
$\cdots$
φ_m
φ_{safe}
$\mathcal{A}_{\varphi_1}$
$\cdots$
$\mathcal{A}_{\varphi_m}$
$\mathcal{A}_{\varphi_{safe}}$
$\mathcal{M}_1^{\Phi}$
$\cdots$
$\mathcal{M}_n^{\Phi}$
Team MDP $\mathcal{M}_T^{\Phi}$
Find STAPU solution
Team Policy π_T^{Φ}
Get next reallocation state
$s_J^{\Phi} \in S^{\perp}$
and set initial state of $\mathcal{M}_T^{\Phi}$ accordingly
Current Joint Policy π_J^{Φ}
Build joint policy
and add to π_J^{Φ}
Policy Evaluation
Expected Task Completion
Verification Step

Figure 7.5: Outline of the overall approach. The mission MDP $\mathcal{M}_i^{\Phi}$ for each robot r_i is built as the product of the robot MDP and the specification DFAs. Then, we build the team MDP $\mathcal{M}_T^{\Phi}$ and solve a STAPU for the initial state of the robots. The obtained sequential policies are then used to build a synchronised joint policy π_J^{Φ}. A reallocation state is chosen from π_J^{Φ}, the initial state of the team MDP, along with its switch transitions, are updated to represent the chosen reallocation state, and a new STAPU is solved. We keep choosing new reallocation states and solving new STAPUs until no more reallocation states exist, or the procedure is interrupted, at which point the current joint policy π_J^{Φ} is returned, along with the expected number of completed tasks.

In this section we have seen how to create and solve a team MDP that is not exponential with respect to the number of robots. The upcoming sections show how to generate a joint robot plan with exact guarantees from the solution of the team MDP. Figure 7.5 summaries the overall approach. From the sequential team MDP solution, we first build a joint policy which allows us to identify reallocation states. We can then generate policies from these states in order to incrementally build a full joint policy. Finally, we use value iteration to generate an exact guarantee on the maximum expected task reward on the full joint policy, which is a common technique used in model-checking MDPs [BHK19].

7.1.2 Building a Joint Policy

The previous section described how to build and solve the team MDP $\mathcal{M}_T^\Phi$ for maximum expected task reward. In this section, we show how to create a joint policy from this. We start with a (memoryless) optimal policy $\pi_T^\Phi : S_T^\Phi \to A_T$ for the team MDP $\mathcal{M}_T^\Phi$. This policy is *sequential* in nature, i.e., it starts by prescribing actions for robot 1, then a switch transition occurs and actions are prescribed for robot 2, and so on. However, our goal is for these policies to be executed *concurrently*. In this section, we describe how we use this sequential policy to construct a joint policy π_J. This joint policy can then be used for execution, and also for providing more accurate performance guarantees over the joint model. Note that, whilst enumerating all states of the joint model is typically unfeasible, many times the states visited *under a policy* is just a small fraction of the full joint model state space. Thus providing guarantees of joint execution is feasible.

Example 19 (Switch Transitions and Global State Features)**.** Note that in Figure 7.4 the switch transition originates from $(1, v_3, o, 0, 1, 0, 0)$ and terminates in $(2, v_1, ?, 0, 1, 0, 0)$ not $(2, v_1, o, 0, 1, 0, 0)$ i.e. the value of the global state feature is not preserved. This design decision is a consequence of the uncertainty in the model. For example, consider the scenario in Figure 7.6 which shows part of a hypothetical office space with a reception area. There is some probability that the reception area is closed meaning that both doors are closed. Imagine a team of 2 robots. Let robot 1's initial position be v_1 and robot 2's initial position be v_8. The teams mission is to visit locations v_0, v_2 and v_7. The team

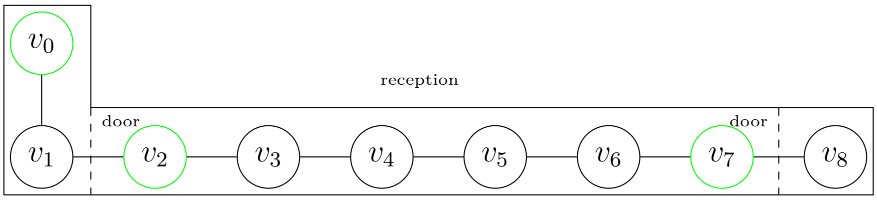


Figure 7.6: Example: Office with Reception with the mission $\langle \mathtt{F}\, v_0, \mathtt{F}\, v_2, \mathtt{F}\, v_7 \rangle$.

model $\mathcal{M}_T^\Phi$ is a sequential model where robot 1 chooses its set of tasks and plans for them and then hands over execution to robot 2. Robot 1 chooses to check the state of the door first, visit v_2 if the door is open and then visit v_0. It then hands over execution to robot 2 sharing the state of the door in the switch transition. If the door is open, robot 2 proceeds to go to v_7. Therefore one path of the joint policy is for robot 1 to check the door and for robot 2 to visit v_7 i.e. the first joint action is cd_1, m_{87}. However, at this point the state of the door is unknown and therefore the action m_{87} is illegal. To avoid such scenarios we do not consider the value of global state features for switch transitions. However, with or without this design decision, the uncertainty in the model coupled with its sequential structure necessitates the need for a joint policy construction step to make sure that no illegal actions are taken.

Since we are working with LTL specifications, we will in fact construct a memoryless joint policy π_J^Φ which applies to the product of the joint model $\mathcal{M}_J$ and the DFAs for the LTL formulas in the mission specification Φ. This can easily be converted into a finite-memory policy π_J for $\mathcal{M}_J$. In constructing the joint policy π_J^Φ, we will assume that robots communicate with each other after executing each action.

We start by defining the *projection* of a team MDP policy π_T^Φ onto a policy over each local product MDP $\mathcal{M}_i^\Phi$.

Definition 32 (Policy projection)**.** Let $\pi_T^\Phi : S_T^\Phi \to A_T$ be a policy for team MDP $\mathcal{M}_T^\Phi$. Its projection onto $\mathcal{M}_i^\Phi$ is the policy $[\pi_T^\Phi]_i : S_i^\Phi \to A_i$, defined by:

$$[\pi_T^\Phi]_i(s) = \begin{cases} \pi_T^\Phi(i,s) & \text{if } \pi_T^\Phi(i,s) \text{ is defined and } \pi_T^\Phi(i,s) \neq \zeta \\ \perp & \text{otherwise.} \end{cases} \tag{7.4}$$

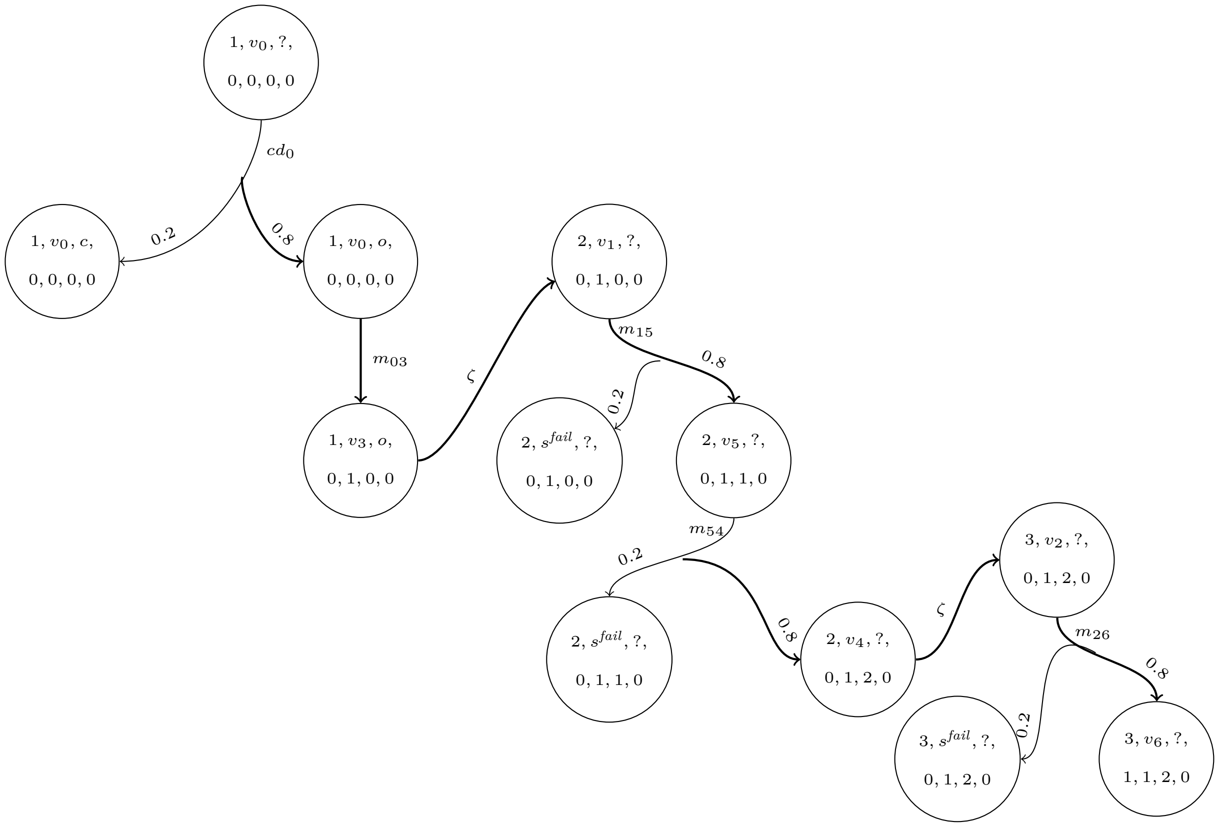


Figure 7.7: A fragment of the team policy. The thick lines show the most likely path in this fragment.

We want to use π_T^Φ to select an action a to take in joint product state $s_J^\Phi = (s_J, q^\Phi)$ where $s_J = (s_1^l, \ldots, s_n^l, s_1^g, \ldots, s_k^g)$ and $q^\Phi = (q_1, \ldots, q_m, q_{safe})$. A first approach one might think of to do so is to choose the joint action:

$$a = ([\pi_T^\Phi]_1([s_J]_1, q^\Phi), \ldots, [\pi_T^\Phi]_n([s_J]_n, q^\Phi)) \tag{7.5}$$

However, doing so ignores the sequential nature of π_T^Φ, which is synthesised over a model that assumes robots act in order. This means that, in action a above, robot i does not consider the tasks that previous robots intend to complete when choosing its action. This can lead to lack of coordination, with multiple robots attempting to complete the same task.

To avoid this issue, the policy action for robot i is chosen using an updated version of q^Φ that assumes the previous robots' execution will be according to the path with highest probability. To define the update to q^Φ, we start by considering the notion of *most probable terminal state* reached in $\mathcal{M}_i^\Phi$ under $[\pi_T^\Phi]_i$. Terminal states are those where no policy action is available, either because a switch transition occurred in the team MDP or because the robot failed.

Definition 33 (Most probable terminal state)**.** Let $\mathit{term}([\pi_T^\Phi]_i) = \{s \in \mathcal{M}_i^\Phi \mid [\pi_T^\Phi]_i(s) = \perp\}$, and $s \in S_i^\Phi$. The most probable terminal state for robot i, starting in s and under policy $[\pi_T^\Phi]_i$ is defined as:

$$s^* = \underset{s' \in \mathit{term}([\pi_T^\Phi]_i)}{\arg\max} \; Pr_{\mathcal{M}_i^\Phi, s}^{[\pi_T^\Phi]_i}(\mathtt{F}\, s'), \tag{7.6}$$

where we slightly abuse LTL notation and write $\mathtt{F}\, s'$ to denote "eventually reach state s'".

Example 20 (Most probable terminal state)**.** Figure 7.7 shows a fragment of the team policy. The most probable terminal state for robot 1 from $(1, v_0, ?, 0, 0, 0, 0)$ is $(1, v_3, o, 0, 1, 0, 0)$. The most probable terminal state for robot 2 from $(2, v_1, ?, 0, 1, 0, 0)$ is $(2, v_4, 0, 1, 2, 0)$. The most probable terminal state for robot 3 from $(3, v_2, ?, 0, 1, 2, 0)$ is $(3, v_6, ?, 1, 1, 2, 0)$.

Using the notion of most probable terminal state, we can define the notion of a *concurrency-aware projection* of a joint product MDP state to a local product state for i.

Definition 34 (Concurrency-aware projection)**.** Let $s_J^\Phi = (s_J, q^\Phi) \in S_J^\Phi$ with $s_J = (s_1^l, \ldots, s_n^l, s_1^g, \ldots, s_k^g)$ and $q^\Phi = (q_1, \ldots, q_m, q_{safe})$ The *concurrency-aware projection* of s_J^Φ onto the local product state space S_i^Φ of robot i is defined recursively as follows:

$$[s_J^\Phi]_1^{||} = (s_1^l, s_1^g, \ldots, s_k^g, q_1, \ldots, q_m, q_{safe}), \tag{7.7}$$

$$[s_J^\Phi]_{i+1}^{||} = (s_{i+1}^l, s_1^g, \ldots, s_k^g, q_1^{i*}, \ldots, q_m^{i*}, q_{safe}^{i*}), \tag{7.8}$$

where $(q_1^{i*}, \ldots, q_m^{i*}, q_{safe}^{i*})$ are the DFA components of the most probable terminal state for robot i, starting in $[s_J^\Phi]_i^{||}$ and under $[\pi_T^\Phi]_i$.

Example 21 (Concurrency-aware projection)**.** The concurrency-aware projection for the initial state of the joint policy $\overline{s}_J^\Phi = (v_0, v_1, v_2, ?, 0, 0, 0, 0)$ for each robot takes into account the most probable terminal state from Definition 33.

$$[\overline{s}_J^\Phi]_1^{||} = (v_0, ?, 0, 0, 0, 0)$$
$$[\overline{s}_J^\Phi]_2^{||} = (v_1, ?, 0, 1, 0, 0)$$
$$[\overline{s}_J^\Phi]_3^{||} = (v_2, ?, 0, 1, 2, 0)$$

Broadly speaking, we project the joint MDP states to the corresponding local state, as in Definition 16, but also update the DFA states taking into account the most probable task execution of previous robots.

We can now define the joint policy obtained from π_T^Φ.

Definition 35 (Joint team policy)**.** The *joint team policy* $\pi_J^\Phi : S_J^\Phi \to A_J$ is such that:

$$\pi_J^\Phi(s_J^\Phi) = \left([\pi_T^\Phi]_1\left([s_J^\Phi]_1^{||}\right), \ldots, [\pi_T^\Phi]_n\left([s_J^\Phi]_n^{||}\right)\right). \tag{7.9}$$

The joint team policy can be executed by the team of robots in a synchronised fashion. Note that, to compute π_J^Φ at each joint product state, we need to compute the concurrency-aware projection of that state to each local product. This requires us to compute the most probable terminal state for each local policy, which can be computationally expensive, as it requires the computation of reachability probabilities. However, this can be avoided by maintaining the results of the reachability probability computations for all states reachable under $[\pi_T^\Phi]_i$, for each robot i. In practice, these reachability probablities can be computed at the beginning of the joint team policy construction step and revised as needed during the computation of the most probable terminal state if that state does not exist in the list.

Example 22 (Joint team policy)**.** Continuing Example 21 the joint action in the initial state $\overline{s}_J^\Phi = (v_0, v_1, v_2, ?, 0, 0, 0, 0)$ is:

$$\begin{aligned}
&\pi_J^\Phi(v_0, v_1, v_2, ?, 0, 0, 0, 0) \\
&\qquad = \left([\pi_T^\Phi]_1\,(1, v_0, ?, 0, 0, 0, 0)\,, [\pi_T^\Phi]_2\,(2, v_1, ?, 0, 1, 0, 0)\,, [\pi_T^\Phi]_3\,(3, v_2, ?, 0, 1, 2, 0)\right) \\
&\qquad =(cd_0, m_{15}, m_{26})
\end{aligned}$$

This state-action pair has 8 successors. We refer the reader to Figure 6.3 (page 145), which shows all successors of this state-action pair. This set of states includes states where the door is either open or closed and robots 2 and 3 have either failed or moved to the next state. One of these is $(v_0, v_5, v_6, o, 0, 1, 1, 0)$ where the door is open and both robots 2 and 3 have succeeded. Note that the door is now open. This means that the states for each robot are now:

$$\begin{aligned}
s_{1T}^\Phi &= (1, v_0, o, 0, 1, 1, 0) \\
s_{2T}^\Phi &= (2, v_5, o, 0, 1, 1, 0) \\
s_{3T}^\Phi &= (3, v_6, o, 0, 1, 1, 0)
\end{aligned}$$

These states are not shown in Figure 7.7 due to space considerations and for the sake of simplicity. However, the actions in these states are the same as those of their counterparts shown in Figure 7.7. The joint action is constructed as follows:

- We begin with the current state of robot 1 $(1, v_0, o, 0, 1, 1, 0)$.
- The action for robot 1 according to the team policy is $[\pi_T^\Phi]_1((1, v_0, o, 0, 1, 1, 0)) = m_{03}$.
- The most probable terminal state for robot 1 in state ${s_1}_T^\Phi = (1, v_0, o, 0, 1, 1, 0)$ is $(1, v_3, o, 1, 1, 1, 0)$.
- The concurrency-aware projection of robot 2's state ${s_2}_T^\Phi = (2, v_5, o, 0, 1, 1, 0)$ is then $[(2, v_5, o, 0, 1, 1, 0)]_2^{\|} = (2, v_5, o, 1, 1, 1, 0)$.
- The action for robot 2 according to the team policy is $[\pi_T^\Phi]_2((2, v_5, o, 0, 1, 1, 0)) = m_{54}$.
- The most probable terminal state for robot 2 in state $(2, v_5, o, 1, 1, 2, 0)$.
- The concurrency-aware projection of robot 3's state ${s_3}_T^\Phi = (3, v_6, o, 0, 1, 1, 0)$ is then $[(3, v_6, o, 0, 1, 1, 0)]_3^{\|} = (3, v_6, o, 1, 1, 2, 0)$.
- The action for robot 3 according to the team policy is $[\pi_T^\Phi]_2((3, v_6, o, 1, 1, 2, 0)) = *$. Recall from Definition 25 (page 90) that $*$ is a *don't care action* when goal states (Definition 27) have been reached.
- Therefore, the most probable terminal state for robot 3 is its current state $(3, v_6, o, 1, 1, 2, 0)$.
- The joint action is then $(m_{03}, m_{54}, *)$ where $*$ indicates that robot 3 remains in its current state.

In this section, we demonstrated the construction of a joint team policy from a sequential team policy. Recall from Figure 7.5 that this is the second step of the overall approach. In the following section, we explain the next step i.e. how to identify states where tasks may need to be reallocated in this joint policy and how to trigger a new planning cycle from these states (replan).

7.1.3 Reallocation through Replanning

As described in Definition 31, switch actions are only added to states where a task has been completed. This allows the team model to consider robots in a sequential fashion, but also introduces *reallocation states* where the joint team policy is not defined. In those cases, our approach is to build another instance of the team MDP, where the initial state is set to the reallocation state. In this subsection, we formally define the reallocation states, and propose an approach for replanning from those states that takes into account how likely they are to occur. By first considering the most probable reallocation states, our algorithm can be viewed as *anytime*, as it can incrementally build a more complete solution, with more accurate performance guarantees, but if interrupted early, it can still provide a partial solution and under-estimate of the guarantee.

Definition 36 (Reallocation states)**.** We define the set of reallocation states as:

$$S^{\perp} = \{s_J^{\Phi} \in S_J^{\Phi} \mid [\pi_T^{\Phi}]_i \left([s_J^{\Phi}]_i^{\|}\right) = \perp \text{ for all } 1 \leq i \leq n\}. \tag{7.10}$$

The definition of reallocation states here differs from the definition of reallocation states in the auctioning and planning approach from Chapter 6. In the auctioning and planning approach there is no team model which can be used to reallocate tasks if a robot has no actions. Therefore the reallocation states in Chapter 6 are those where *any* robot has no actions. In STAPU however, switch transitions allow exchange of task information across the team. As a result, reallocation states in STAPU are those where *all* robots have no actions.

In order to replan for a reallocation state $s_J^{\Phi} \in S^{\perp}$, we simply set the initial states of each local MDP $\mathcal{M}_i$ to corresponding projection $[s_J^{\Phi}]_i$, and reconstruct a team MDP according to Definition 31.

We consider more likely reallocation states first, by ordering the set $S^{\perp}$ in decreasing order of reachability probability under π_J^{Φ}, i.e. we order states $s_J^{\Phi} \in S^{\perp}$ in decreasing order of $Pr^{\pi_J^{\Phi}}_{\mathcal{M}_J^{\Phi}, \overline{s}_J^{\Phi}}(\mathtt{F}\, s_J^{\Phi})$.

Example 23. Continuing Example 22 the joint action for the state $(v_0, v_5, v_6, o, 0, 1, 1, 0)$ is $(m_{03}, m_{54}, *)$. This action leads to two states. State $(v_3, v_4, v_6, o, 1, 2, 1, 0)$ is an accepting state because all robots have completed their task sets and does not need to be expanded further. State $(v_3, s^{fail}, v_6, o, 1, 1, 1, 0)$ is not an accepting state, because robot 2 failed midway through its task. In this state, none of the robots have any associated actions. This is a reallocation state and it is added to a priority queue. In this queue the probability of reaching a certain state from the initial state of the joint policy is used to order states. For example, the probability of reaching state $(v_3, s^{fail}, v_6, o, 1, 1, 1, 0)$ from state $\overline{s}^{\Phi}_{J} = (v_0, v_1, v_2, ?, 0, 0, 0, 0)$ is 0.1024 while the probability of reaching $(v_0, s^{fail}, v_6, c, 0, 0, 1, 0)$ is 0.032. Therefore, $(v_0, s^{fail}, v_6, c, 0, 0, 1, 0)$ is chosen for reallocation and planning first. In order to reallocate, the the team model's initial state is updated to $\overline{s}^{\Phi}_{T} = (1, v_0, c, 0, 0, 1, 0)$. Previously added switch transitions are removed because the initial states of all robots have changed. The new switch transitions use the updated states of all robots. Once the team model is updated, NVI is used to generate a team policy.

7.1.4 Generation of guarantees

The final step in our approach is the generation of the maximum expected task reward when following this policy. This is computed using policy evaluation (see Definition 5) on the final joint policy, using task reward as objective and cost as a tie-breaker. The process is the same as that in Chapter 6 and Chapter 4.

Remark 19 (On the correctness of STAPU). Throughout this chapter we have provided formal definitions of the various elements of the STAPU approach. The team MDP construction follows [SBD18b]. The solution to the team MDP is generated using VI which is widely used in model-checking MDPs. The joint policy construction ensures that the states of the automata evolve as per the transitions described in the automata themselves. Furthermore, while the sequential team policy is used to get the joint action, this action is executed on the joint team MDP. This ensures that no illegal actions are executed and the state transitions of the joint team MDP are preserved. Finally, in order to generate a correct guarantee on the maximum expected task reward, we perform VI on the final joint team policy which preserves DFA and joint team MDP state transitions as explained above. This ensures that the exact value of the maximum expected task reward is correct and that a feasible plan is generated as per the mission specification and robot models.

7.2 Results

A comparison of STAPU against the naive approach of building and solving the full joint MDP is impractical since this has very limited scalability. Instead, for a more competitive baseline comparison, we use the *auctioning and planning* approach presented in Chapter 6.

A key benefit of STAPU is its application of probabilistic model checking techniques, which use LTL formulas for task and safety specifications, and compute probabilistic guarantees on safety or efficiency for the policies that are generated. So, for a fair comparison, we adopt similar techniques for the baseline implementation based on auctioning.

As in Chapter 6 we compare this approach with LRTDP from Chapter 5. For these comparisons LRTDP has a time out which we set to the maximum of the time taken for STAPU or auctioning and planning.

The experimental setup[1] used has already been described in Section 4.3 and was used to generate results for tests in Chapters 4 to 6. The set of tests is also the same, specifically those introduced in Section 4.3.3; namely warehouse scenario described in Section 4.3.3, without doors Figure 4.5c and with doors Figure 4.5d and the grid scenarios e.g. Figure 4.6a.

7.2.1 Comparison with Auctioning and Planning

We evaluate STAPU, focusing on scalability and the quality of generated policies compared to auctioning and planning from Chapter 6 and sampling-based search from Chapter 5. As we saw in Chapter 6, the auctioning and planning approach outperformed the sampling-

[1] link to the code: https://github.com/fatmaf/prism-school-pc

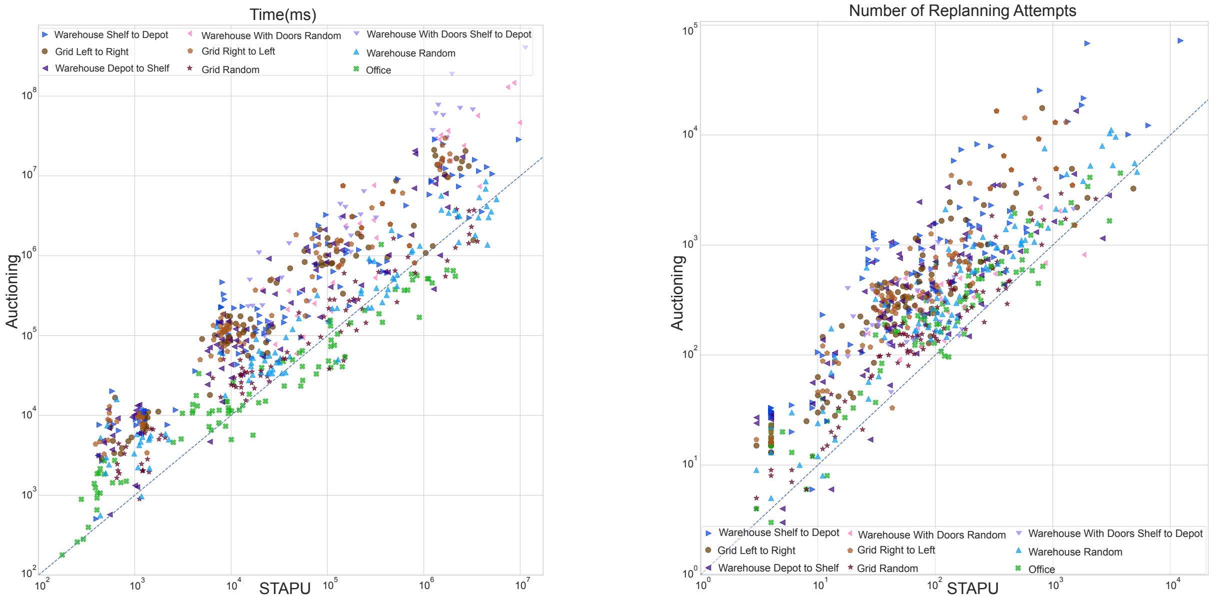
based search algorithm, LRTDP. Further, auctioning and planning approaches are widely used for multi-robot task allocation and planning. Therefore, we use this approach as our baseline.

All three approaches consider the efficiency of policies by adding a cost based on distance to each action in the local models and minimising expected cumulative cost as a secondary objective.

First, Figure 7.8a shows the total time required for policy generation across the full set of test environments. We use a scatter plot, with STAPU on the x-axis and auctioning on the y-axis, so points above the dotted indicate better performance (shorter times) for STAPU. We see that STAPU generally takes less time than auctioning. An exception is the office environment, where the model is smaller and auctioning is usually faster. STAPU solves instances of the team MDP, which is larger and slower to solve than the local robot MDPs solved during auctioning. However, the decoupling of task allocation and planning in the latter leads to more occurrences of replanning in order to reallocate tasks after a robot fails (and therefore a larger number of MDPs need to be built and solved). The number of times that replanning occurs is shown with a similar scatter plot in Figure 7.8b and shows a correlation with the timing results.

To check that faster performance by STAPU is not at the expense of generating policies of poorer quality, we also compare the probabilistic guarantees offered by the policies that STAPU and auctioning generate. Figure 7.9a plots the expected number of tasks for each approach, as computed from the generated joint policy. We see that these values are generally very similar for STAPU and auctioning. For completeness, we also present the expected cost for STAPU and auctioning in Figure 7.9b. Here too, the values are generally very similar for both approaches.

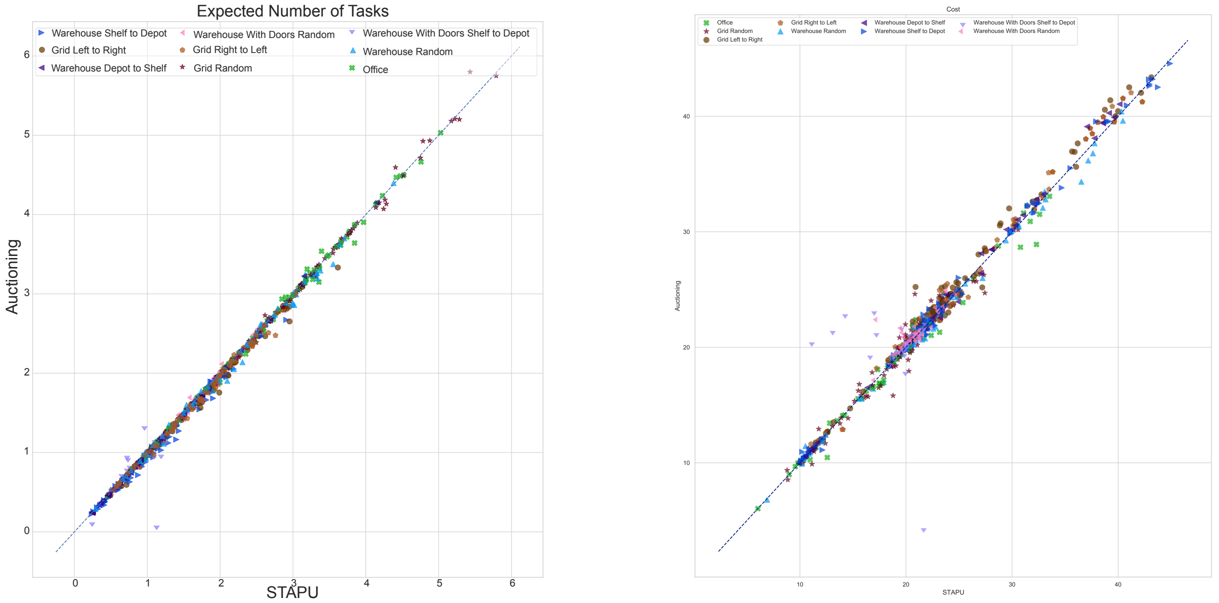
Next, we consider how the time required by STAPU and auctioning is affected by various aspects of the test environments. For clarity of presentation we consider a fixed environment in each case, but we have observed similar patterns on the other benchmarks. Figure 7.10 shows the results. Since there are multiple random instances for each test



(a) Total computation time (in milliseconds)

(b) Number of times that replanning occurs

Figure 7.8: Scatter plots showing performance for STAPU (x-axis) vs. auctioning (y-axis) across the full set of test environments.



(a) Expected number of completed tasks.

(b) Expected cost.

Figure 7.9: Scatter plots of the expected number of completed tasks and expected cost for STAPU (x-axis) vs. auctioning (y-axis) across the full set of test environments.

environment, we use boxplots that show the mean and median values of each set and their range. The box shows the interquartile range and the ends indicate the outer-quartiles. The mean is represented by a triangle, outliers by diamonds and the median is the horizontal bar enclosed in the box.

Figure 7.10a shows times as we vary the *percentage of states in which failures can occur*. Larger numbers of failures lead to more replanning and thus higher times. The rate of increase in time is less steep for STAPU. For a clearer comparison of the difference in effect on STAPU and auctioning, Figure 7.10b shows relative times: the ratio of times for auctioning and STAPU (i.e., the speedup factor obtained with STAPU). As the percentage of failure states increase, we see that STAPU does better, up to 10 times faster in cases. This is because the sequential team model, allows reallocation of tasks after failures and reduces the amount of replanning required. STAPU is also able to modify the existing team MDP for each replan, where as auctioning has to build and solve a new one each time. However, the team MDP is larger and slower to solve. For lower numbers of failure states, the additional overhead of solving larger MDPs in STAPU outweighs the gain and auctioning is faster.

Figure 7.10c shows the effect on time as the *number of robots* is increased. This increases the size of the team MDP for STAPU, but also the amount of replanning needed, so STAPU remains faster in general. Similar trends are seen in Figure 7.10d, which shows the times as the *number of tasks* increases, and in Figure 7.10e, where we vary the *number of map locations* (and therefore the size of each robot's local MDP). For the latter, we fully connected square grids with increasing sizes.

Finally, we evaluate the performance of the two methods as the *number of global states features* increases. For this, we use a variation of the warehouse environment where the status of doors are modelled as global states. The default state of each door is unknown; and in order to go through a door, a robot must check whether the door is open or closed. STAPU's team model allows robots to transfer knowledge of this value. As seen in Figure 7.10f STAPU is significantly faster than the auctioning approach as the number of shared states increase.

7.2.2 Comparison with LRTDP

We saw in Chapter 6 that LRTDP does not solve the problem in the time STAPU or *auctioning and planning* can. The policies computed using LRTDP with a time out to match STAPU and auctioning are sub par. This is because LRTDP searches the entire joint model for a solution which takes much longer. This is reflected in Figure 7.11 which adds results for STAPU on to Figure 6.6. From Figure 7.11 it is clear that the quality of policies (solutions) produced by LRTDP within the 2 hour time limit is always worse than those produced by STAPU or *auctioning and planning.* LRTDP consistently produces a lower value for the expected number of tasks for each test set.

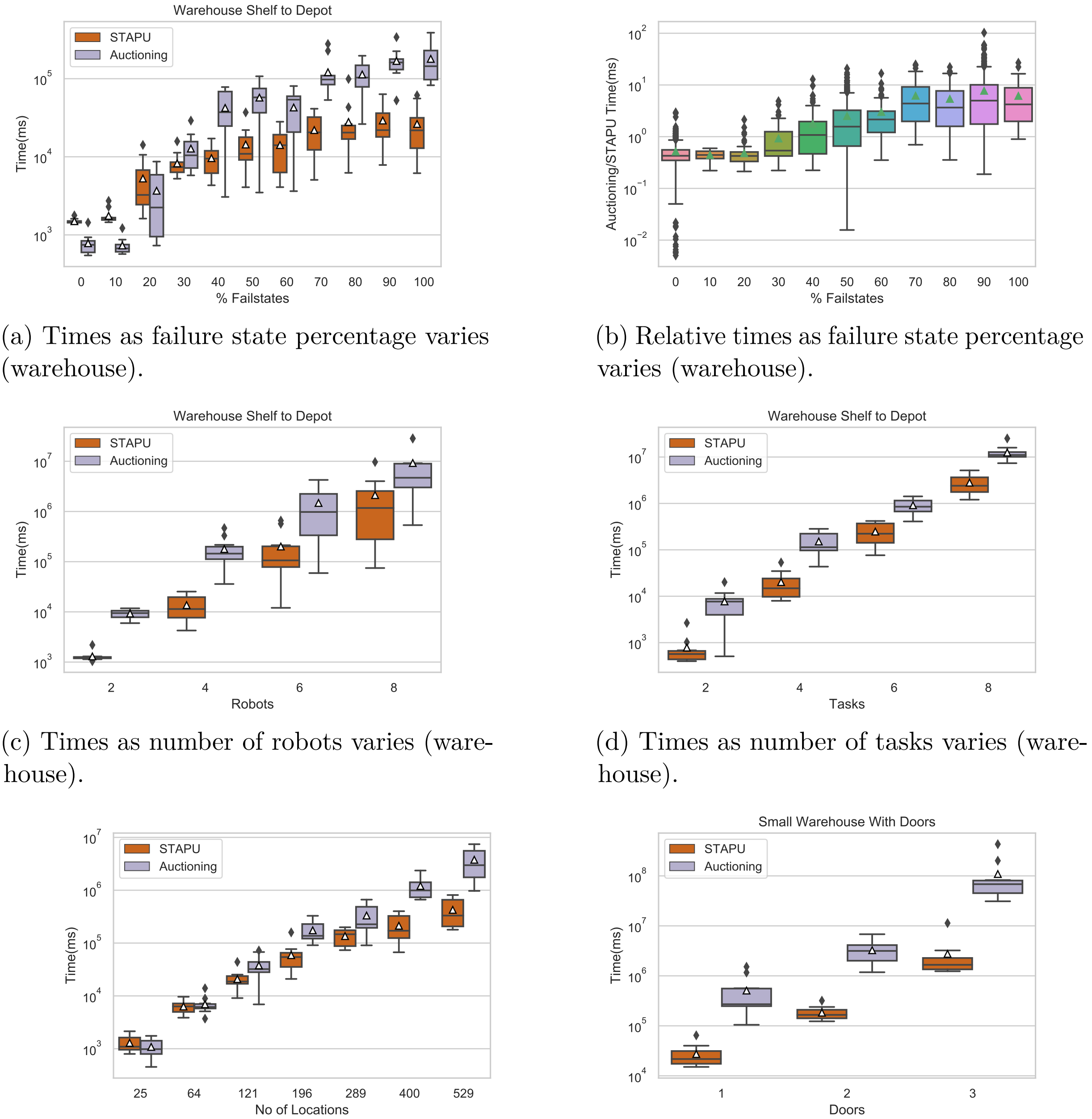
Summary

- In this chapter, we presented a solution to maximise expected task reward for a multi-robot team, STAPU.

- The use of a sequential team model allowed STAPU to allocate tasks and plan simultaneously, as in the sampling approach in Chapter 5. It also reduced the space complexity of the problem, avoiding the exponential state-space explosion with the number of robots.

- An extra step was needed to create the joint policy and identify reallocation states, similar to what was proposed in the auctioning approach (Chapter 6). In fact, the only difference between the joint policy construction for both approaches was the identification of reallocation states.

- With regard to computation time, STAPU was able to outperform auctioning. As we saw, STAPU is feasible for larger models. For smaller models, the overhead of building the team model, impacts STAPU's performance negatively.



(a) Times as failure state percentage varies (warehouse).

(b) Relative times as failure state percentage varies (warehouse).

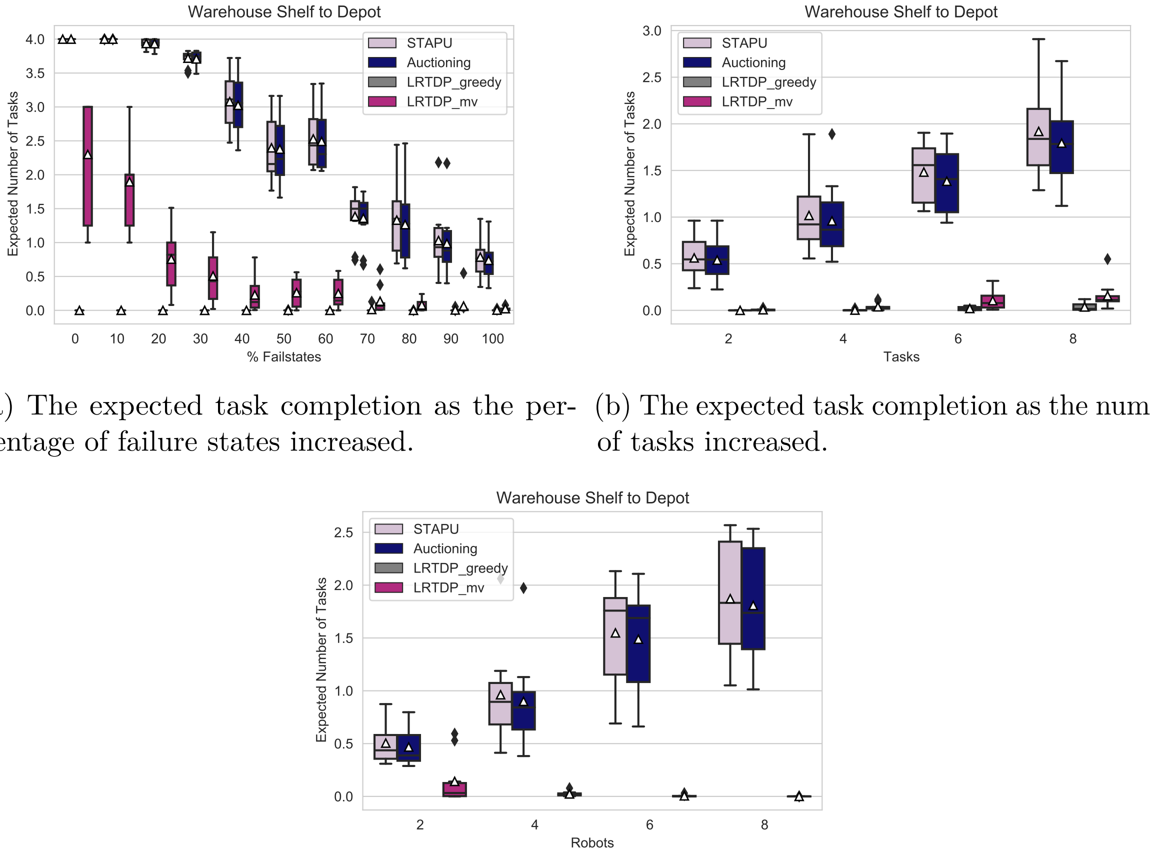
(c) Times as number of robots varies (warehouse).

(d) Times as number of tasks varies (warehouse).

(e) Times as number of map locations varies (grid).

(f) Times as num. global features varies (warehouse+doors).

Figure 7.10: Box plots comparing the computation time for STAPU and auctioning as different aspects of the benchmark models are varied.



(a) The expected task completion as the percentage of failure states increased.

(b) The expected task completion as the number of tasks increased.

(c) The expected task completion as the number of robots increased.

Figure 7.11: Boxplots, where the triangles indicate the mean values and whiskers show the interquartile range and dots show outliers. A randomised set of 10 scenarios were used to generate these results. The number of robots was set to 4 and the number of tasks was set to 5. The number of doors i.e. global states was 0. LRTDP_greedy is the policy extracted by using greedy action selection. LRTDP_mv is the policy extracted by selection the most visited action in each state of the search tree.

- With regard to number of tasks completed by the team, STAPU produced policies that were comparable to those produced by the auctioning and planning approach. It is possible that with further optimisation STAPU is able to do even better in terms of this guarantee.

Chapter 8

Conclusions

8.1 Summary

The work in this thesis combines techniques from formal verification with techniques from robot planning to allocate tasks and generate policies with a probabilistic guarantee on the number of tasks completed by the joint policy. In this section we present a summary of the work highlighting our findings and contributions.

8.1.1 Mission Specification using LTL

We formalised the problem of robust multi-robot task allocation and planning under uncertainty with a formal mission specification using a fragment of Linear Temporal Logic (LTL) evaluated on finite sequences. To the best of our knowledge, this formulation is *unique* because it uses the automata of the LTL specification to generate a reward function in the context of a multi-robot system with *partial mission satisfaction.* All solution methods to this problem that have been presented in this thesis operated on a product MDP, which combined the LTL automata with the robot model. Generating policies using this product MDP allowed us to track task progress and automatically determine tasks in the mission that were completed. Another advantage of these automata was enabling algorithms to determine task allocations and reallocations without the need for any external input.

The use of LTL for mission specification also provided the multi-robot team and the solution methods used with a formal description of the mission. This allowed for the generation of a probabilistic guarantee on the number of expected tasks completed when following a given policy. Each approach presented in the thesis had a separate verification step that operated on the joint product policy leveraging the LTL DFAs combined with the robot MDP model. In fact, the use of LTL allowed for the specification of rich behaviour in an intuitive manner. A consequence of this is the contribution of the corresponding LTL automata to the increase in size of the product model. However, there is no non-LTL approach that is cheaper and allows for the same level of richness of behaviour specification.

8.1.2 Solution Approaches

We presented three separate solution methods for robust multi-robot task allocation and planning, which provided a task completion guarantee on the multi-robot plans. Note that the addition of a *guarantee* on the plans in the form of maximum expected task reward means that a discounted reward can not be used. The use of a discount factor guarantees convergence of MDP solution algorithms [MK12].

All three approaches were implemented using the PRISM [KNP11] model checker API and evaluated on the same test sets. To this end, a python GUI was created to generate test environments and their corresponding PRISM models.

8.1.2.1 Sampling-based Heuristic Search

The first of these solution methods employed Labelled Real Time Dynamic Programming (LRTDP), a trial-based heuristic search method. Such methods are able to generate policies using the full joint product model but limiting the search space to state-action pairs that seem promising according to the heuristic function. Therefore, these methods are able to generate feasible or optimal policies utilising fewer computational resources by not building the full joint product model. Using LRTDP to solve the problem of reward maximisation for an MDP with zero-reward cycles requires cycle detection [Kol+11; Brá+14; Ash+18], which has considerable overheads. To remedy this, we demonstrated the use of a cost structure based on the automata of the LTL task set, which penalised states relative to the number of tasks completed. This changed the optimisation objective from reward maximisation to cost minimisation. Using this cost structure we assigned a finite penalty [KW12] to states in the MDP from which a goal state was not reachable. As a result we were able to generate a joint policy which provided a feasible solution to our multi-robot task allocation and planning under uncertainty problem.

In order to improve the solution quality we used a rollout policy generated from single robot solutions to guide the initial search. Furthermore, we implemented a dead-end detection mechanism using these single robot solutions, in order to detect states where no further tasks would be completed. Our automata based cost structure was feasible for the problem and was able to push the search towards viable solutions. However, our results showed that LRTDP does not scale well for large, fully connected models e.g. grid-based scaenarios used in this thesis. Our experiments also confirmed that when LRTDP did not converge in the given time, using the most visited action to extract the policy offers better performance than using the greedy best action [Hua11; Bro+12], as is the case with Monte-Carlo Tree Search (MCTS).

8.1.2.2 Auctioning and Planning

Our second solution method separated the task allocation and planning processes using an auction-based approach. Auctioning-based approaches to task allocation and planning are able to generate close to optimal solutions [Koe+06]. This requires that the choice of objectives used to judge bids in the auction phase is aligned with those used in the planning phase. One advantage of auctioning and planning is the reduction in the size of the MDPs that must be solved, which greatly reduces planning time.

Our results showed that auctioning and planning is a viable option for our problem of robust multi-robot task allocation and planning. It was able to avoid the exponential increase in the joint state action space as the number of robots grows. The separation of task allocation and planning resulted in single robot models that were solved more quickly when compared to the sampling-based search method. We also showed that this approach greatly outperforms the sampling-based approach in terms of the expected number of tasks completed following the joint policy and the computation time itself. Therefore, planning in the full joint model space is sub-par when compared to auctioning and planning.

Handling Uncertainty Our formulation used MDPs to model uncertainty, explicitly including robot failure. This allowed for the generation of robust multi-robot policies which incorporated task reallocation. The sampling approach operated on the joint model and so tasks could be reallocated when robots failed without any special mechanisms to do so.

Unlike the sampling approach, auctioning worked on individual robot models and was therefore unable to detect uncertainty in the team's behaviour. This was remedied by the joint policy construction step that allowed for the identification of reallocation states. In order to generate a full joint policy, the auctioning and planning cycle was repeated for each of these states. Our results also demonstrated that as the uncertainty in the model increased and more robots failed, more auctioning and planning cycles were required to build the full joint policy, which increased the overall computation time. This was partly due to the separation between task allocation and planning which we remedied in our final solution method, STAPU.

8.1.2.3 Simultaneous Task Allocation and Planning

Our final solution, the *simultaneous task allocation and planning under uncertainty* algorithm (STAPU), used a sequential team model to generate a task allocation and policy for a multi-robot team, which was robust to the possibility of robot failure. In the sequential team model, single robot models were connected with transitions at all states where tasks were completed. These transitions enabled the algorithm (STAPU) to simultaneously allocate tasks and generate plans for the robots. However, like the auctioning and planning approach, information about common environment states was not shared in this team model. This combined with the uncertainty in robot behaviour e.g. the possibility of robot failure, necessitated the construction of a joint policy, as in the auctioning and planning approach.

Our results showed that due to the sharing of task related information during the planning phase, the number of reallocation states was greatly reduced when compared to auctioning and planning. This in turn reduced the overall computation time. The process for task reallocation here was similar to that of the auctioning and planning approach, in that a new planning cycle needed to be triggered. We showed that STAPU was able to generate plans more quickly than auctioning and planning with policies of comparable quality. Furthermore, our results showed that for smaller environments, the overhead of the sequential team model contributed to the higher computation time when STAPU was compared with auctioning and planning. However, for larger environments this was not the case and STAPU was able to outperform auctioning and planning in terms of computation time. Our results also demonstrated that the quality of the multi-robot policies generated through STAPU were comparable to those generated by the auctioning and planning approach.

8.1.3 Contributions

In summary, our contributions are:

- Formalisation of a robust multi-robot task allocation and planning problem considering robot failure and using LTL specifications to generate a task reward structure. Combined with the objective of satisfying as much of the mission as possible, this problem formulation is unique and novel to the best of our knowledge. While there are works that look at partial satisfaction for single robots with LTL specifications, we are not aware of work that attempts this for a team of robots with LTL without any user driven rewards.

- A sampling-based heuristic search solution method to the problem without any cycle detection using a novel task cost structure that penalised states based on the LTL automata. Through this cost structure, our approach is able to generate a feasible (albeit poor) solution to the problem.

- An auctioning and planning approach with a reward structure able to count the number of completed tasks using the LTL automata and satisfy as much of the mission as possible. The closest to this work are [Car+20; SBD18a; SBD18c], however, they does not consider task reallocation to other robots and are not able to provide a guarantee on the number of tasks completed by the team.

- A technique to build a joint policy from single robot policies able to isolate states where robots failed and/or those where tasks needed to be reallocated.

- The *simultaneous task allocation and planning under uncertainty* algorithm (STAPU) which used a sequential team model to simultaneously allocate tasks and generate policies. This approach extends the work in [SBD18d] to MDPs. However, the addition of uncertainty makes the two works incomparable.

- Experimental analysis of all three approaches on a set of test scenarios and environments. These were inspired by existing benchmarks in the field of multi-agent path finding and were generated as PRISM [KNP11] models with LTL specifications.

- A GUI that generates PRISM models from user specified environments or computer generated random 4-connected grid environments.

Therefore, the work in this thesis has illustrated that the complexity of combined task allocation and planning under uncertainty can be reduced by generating feasible policies using auctioning and STAPU. The use of LTL not only allows for formal task specification but the generation of guarantees. Furthermore, it may be beneficial to verify policies against specifications *instead* of using traditional model checking techniques to generate policies for specifications. Finally, it may not always be possible to generate a full joint policy, specifically when individual robots experience critical failure. The work in this thesis illustrated the use of replanning to deal with such scenarios, iteratively building a full joint policy i.e. a policy that is robust to uncertainty in action outcomes including individual robot failure.

8.2 Future Work

In this section we discuss possible extensions of our work.

User Preference for Tasks We formulated the task reward structure such that each task has equal reward. However, in some scenarios certain tasks may be more important than others. Incorporating user preference in our task reward structure is fairly straightforward. We would simply need an interface to allow users to assign values to individual tasks.

Another possible addition is that of soft constraints. Our formulation assumed that the safety constraint in the mission specification is a hard constraint. This meant that if the safety constraint is violated, the entire mission fails. However, as in [Lah+15; TD16], in some scenarios there may be constraints that can be violated at a cost. These are soft constraints. Incorporating these into our work could also be done through a reward structure, perhaps a negative reward for violating each constraint. This would of course affect the solution approaches used to solve the problem, particularly the sampling-based approach. Competing reward functions would result in the formulation of a constrained MDP as in [TTT17] or a multi-objective optimisation problem as in [Lah+16].

Collisions and Robot Behaviour Our approaches did not consider robots colliding with each other in the environment. Taking inspiration from Conflict Based Search [Sha+15], robot collisions or conflicts can be detected during the joint policy construction phase in STAPU and auctioning and planning. In fact such an approach has already been used in Multi-agent Path Finding as in [MK16; Fel+17]. Once these collisions are detected, a new cycle can be triggered.

This solution can also be applied to the sampling-based search solution in Chapter 5. However, as we have demonstrated, for large fully connected environments sampling-based search does not perform well. Therefore, repeating the search multiple times to reduce conflicts would increase the computation time. Instead of replanning, the sampling-based search algorithm itself can be modified to disable actions that lead robots to the same states.

Our problem formulation considered the possibility of robot failure or uncertainty in global states such as doors, however there may be other high-level robot behaviours that could add uncertainty to the problem. These behaviours could be influenced by the physical limitations of the robots or the workspace/environment itself. An example of robot related behaviours is slower speeds due to reduced charge. Examples of environment related behaviours include slower speeds due to crowds or human interaction. Therefore our approaches would benefit from richer robot models.

Complex Tasks Though we presented general approaches to task allocation and reallocation, it remains to be seen how these approaches tackle more complex LTL task specifications. For example, what would happen if a robot failed midway through a pick up and delivery task? Or what would happen if a robot failed midway through a task that enforced a strict sequence such as "Do task A immediately after task B". In the case of pick up and delivery tasks, an extra global state for the object to be picked up could be added. This would allow the planning algorithm to track the location of the object if a robot fails. However, as the number of objects would grow, so would the size of the state space and therefore, such an approach would not scale well.

The solution methods presented in this thesis do not place constraints on reallocating strictly sequential tasks. For example, if a robot fails midway through a task, that task is reallocated to another robot. The other robot simply continues the task from then on. This may be impractical for certain scenarios for example, when a data gathering robot fails to get to the upload location or a robot contaminated with radiation fails to get to the safe room. [SBD18d] is able to decompose mission specifications with sequential tasks efficiently, however it does not take into account any uncertainty.

Stemming from this, another interesting avenue to explore is that of complex tasks that require multiple robots like those in [TD16; NTD16; Sch+16]. In such scenarios, a sum of costs objective may not be feasible and therefore more complex solution objectives might need to be explored.

Finally, all solution approaches presented in this thesis apply to heterogeneous robots. Therefore, it would be interesting to see their application to such a team, specifically when only certain robots can perform certain tasks.

Action Durations and Asynchronous Behaviour We assume that all robots execute actions at the same speed, i.e. they move in lock step. In practice, this is not the case. One way to model action durations is using costs. In fact in all our work, we assume that each action has cost 1 and use this as a tie-breaker. Making the cost for actions non-uniform can be done easily through this cost function. However, it does not solve the problem of robots moving in lock step. The most obvious but expensive approach to solve this would be to introduce a state variable for time. Similar to [Ulu+13], *travelling* states where at least one robot has finished executing its action, can be introduced. The sampling approach would be the best candidate for such a solution, since it generates states on the fly.

The auctioning and planning and STAPU solution methods use VI to produce solutions. Therefore, introducing such states in those approaches is not trivial. The introduction of the time state variable in these approaches would result in a large increase in the number of states. This increase would be dependent on the granularity of the time variable itself. However, it would make the problem intractable. For this reason, perhaps, it would be prudent to introduce a time variable in the joint policy construction step instead. However, it is not clear how this can be incorporated in replanning cycles. Other ways to model uncertain action durations will influence the planning approach e.g. [Str+20] uses priority-based planning or the choice of models e.g. [Man+19] looks at homogeneous robots using Petri Nets. The work in [Str+20] could be used in conjunction with auctioning based approaches for task allocation to generate feasible plans.

Lastly, as in [Haw+17], the action durations could be learnt through repeated exploration of the workspace using approaches like [Kra+17].

Algorithmic Improvements The solutions presented in this thesis do not exploit parallelism. The sampling approach could be spread over multiple processors with a consolidation step, as in [KZ18]. In the auctioning and planning approach, all robots could place their bids in parallel and plan in parallel as well [ZSP08]. Similarly, in STAPU all local product MDPs could be built in parallel. For large models, the gains in computation time due to parallelism may be well worth the overheads. However, for smaller models this may not be the case.

The performance of the auctioning and planning approach can also be improved by using approximations when generating robot bids like [Lag+04]. Unlike our approach in Chapter 6, this would mean that robot policies are not generated in the auctioning phase. However, the approximations could be used to guide the planning process or provide an initial policy that can be further refined, as in [WC17; Din+14].

The sampling approach presented in this thesis did not use any cycle detection. It would be worthwhile to investigate the performance of a sampling approach that implements cycle detection, similar to [Kol+11; SHB16; Brá+14; Ash+18]. Perhaps, it is possible to improve on cycle detection for a multi-robot planning problem.

Lastly, we have not considered the use of linear programming or constraint satisfaction based methods for solving the robust multi-robot task allocation and planning problem under uncertainty. It is known that these do not scale well, however, they have still been employed to generate solutions by making assumptions such as deterministic robots [Leo+17b; GMS17] with a separate component dealing with uncertainty [GMS17;

Des+17]. Furthermore, similar approaches have been used for MDPs [TTT17; BR06; DFR08] as well. They too do not scale well in comparison with other exact methods such as Value Iteration [BR06]. Nonetheless, the resulting policies are either optimal or possibly at least closer to the optimal than the approaches presented here. Perhaps, there is room for some abstractions that allow these algorithms to scale well while being able to provide probabilistic guarantees on the resulting feasible policies.

8.3 Recommendations

The problem set out in the beginning (Section 4.2) of this thesis was that of generating a joint multi-robot policy for a team of robots with uncertain action outcomes including robot failure. A key feature of the problem was the ability to *verify* the policy by generating a guarantee i.e. an exact value on the expected number of tasks completed by the team. This meant that the joint policy would attempt to partially satisfy the mission, if full satisfaction was not possible. Another feature was that of reallocating tasks to functioning robots in the team when one or more robots failed.

To this end, the results from the algorithms presented in this thesis show that the use of sequential team model and replanning as in STAPU i.e. Chapter 7 is able to generate verified joint policies in reasonable time for robot teams. It is also able to relay task information in the planning phase. This would be beneficial in scenarios where robots experience critical failure often and tasks need to be reallocated. However, in scenarios where the robot models are small, an auctioning and planning approach like in Chapter 6 may be more computationally feasible, particularly when robots do not fail often. While sampling-based approaches generally fare well, LRTDP as applied in Chapter 5 is not well suited to the problem of producing verified policies for large multi-robot MDPs.

The common thread in all the approaches presented in this thesis was the use of LTL to generate task rewards or costs, track task progress and verify the resulting joint policies by producing a guarantee on the team's task completion. Generating automatic reward or cost structures does away with the need of manual tuning of said rewards or costs to get the expected solution and is, in my opinion, well worth the (computational) expense. Furthermore, as motivated in Chapter 1, a guarantee on the joint policy in terms of an exact value on a quantitative property, is not only useful but crucial to successful deployment. Tracking task progress too is very useful for replanning as well as informing users of the team's progress and satisfying as much of the mission as possible. In fact, partial mission satisfaction is a more realistic and plausible objective, specially when robots are known to fail. Therefore, the benefits of using LTL (or similar logics) in planning outweigh the disadvantage of computational expense.

In conclusion, this thesis presented a unique multi-robot problem formulation inspired from real life situations where robots might fail. It also illustrated the use of various solution methods to solve this problem providing guarantees on team behaviour. To that end, the work in this thesis is a step towards making multi-robot system deployments more reliable and robust. However, as evidenced by the variety of possible extensions, there is a lot more still to be done.

Bibliography

[ACF02] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2 (2002), pp. 235–256.

[AI20] Muhammad Zulfaqar Azmi and Toshio Ito. "Artificial Potential Field with Discrete Map Transformation for Feasible Indoor Path Planning". In: *Applied Sciences* 10.24 (Dec. 2020), p. 8987. ISSN: 2076-3417. DOI: 10.3390/app10248987. URL: http://dx.doi.org/10.3390/app10248987.

[Alt02] Eitan Altman. "Applications of Markov decision processes in communication networks". In: *Handbook of Markov decision processes*. Springer, 2002, pp. 489–536.

[Ama+13] Christopher Amato, Girish Chowdhary, Alborz Geramifard, N. Kemal üre, and Mykel J. Kochenderfer. "Decentralized control of partially observable Markov decision processes". In: *52nd IEEE Conference on Decision and Control*. IEEE. 2013, pp. 2398–2405.

[Ash+18] Pranav Ashok, Tomáš Brázdil, Jan Křetínský, and Ondřej Slámečka. "Monte carlo tree search for verifying reachability in markov decision processes". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 11245 LNCS. 2018, pp. 322–335. ISBN: 9783030034207. URL: https://arxiv.org/pdf/1809.03299.pdf.

[Bai] Christel Baier. “Tutorial: Probabilistic Model Checking”. In: (). URL: `https://resources.mpi-inf.mpg.de/departments/rg1/conferences/vtsa17/slides/baier-vsta17-MDP-final.pdf`.

[Bai+14] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Steffen Märcker. “Computing conditional probabilities in Markovian models efficiently”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 515–530.

[Bas+20] Abul Bashar, Shahabuddin Muhammad, Nazeeruddin Mohammad, and Majid Khan. “Modeling and Analysis of MDP-based Security Risk Assessment System for Smart Grids”. In: *2020 Fourth International Conference on Inventive Systems and Control (ICISC)*. IEEE. 2020, pp. 25–30.

[BBS95] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. “Learning to act using real-time dynamic programming”. In: *Artificial Intelligence* 72.1-2 (1995), pp. 81–138. ISSN: 0004-3702. DOI: `10.1016/0004-3702(94)00011-O`.

[BCC12] J. Benton, Amanda Coles, and Andrew Coles. “Temporal planning with preferences and time-dependent continuous costs”. In: *Twenty-Second International Conference on Automated Planning and Scheduling*. 2012.

[Bel03] Richard Ernest Bellman. *Dynamic Programming*. USA: Dover Publications, Inc., 2003. ISBN: 0486428095.

[Bel57] Richard Bellman. “A Markovian decision process”. In: *Journal of mathematics and mechanics* 6.5 (1957), pp. 679–684.

[BG03] Blai Bonet and Héctor Hector Geffner. “Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming”. In: *ICAPS’03 Proceedings of the Thirteenth International Conference on International Conference on Automated Planning and Scheduling*. 2003. ISBN: 1-57735-187-8. URL: `www.aaai.org`.

[BHK19] Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. “The 10,000 Facets of MDP Model Checking”. In: *Computing and Software Science*. Ed. by Bernhard Steffen and Gerhard Woeginger. Lecture Notes in Computer Science. Netherlands: Springer, 2019, pp. 420–451. ISBN: 978-3-319-91907-2. DOI: 10.1007/978-3-319-91908-9_21.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*. Vol. 950. Cambridge, Mass: MIT Press, 2008. ISBN: 9780262026499.

[Bou96] Craig Boutilier. “Planning, Learning and Coordination in Multiagent Decision Processes”. In: TARK ’96 (1996), pp. 195–210. URL: http://dl.acm.org/citation.cfm?id=1029693.1029710.

[Boz+20] Alper Kamil Bozkurt, Yu Wang, Michael M. Zavlanos, and Miroslav Pajic. “Control synthesis from linear temporal logic specifications using model-free reinforcement learning”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 10349–10355.

[BR06] Diego Bello and German Riano. “Linear programming solvers for Markov decision processes”. In: *2006 IEEE Systems and Information Engineering Design Symposium*. IEEE. 2006, pp. 90–95.

[BR11] Nicole Bäuerle and Ulrich Rieder. *Markov decision processes with applications to finance*. Springer Science & Business Media, 2011.

[Brá+14] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. “Verification of Markov decision processes using learning algorithms”. In: (2014), pp. 98–114.

[Bro+12] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Senior Member, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and*

AI in Games 4.1 (2012), pp. 1–43. ISSN: 1943-068X. DOI: `10.1109/tciaig.2012.2186810`. URL: `http://mcts.ai/pubs/mcts-survey-master.pdf`.

[BT91] Dimitri P. Bertsekas and John N. Tsitsiklis. “An analysis of stochastic shortest path problems”. In: *Mathematics of Operations Research* 16.3 (1991), pp. 580–595.

[Cai+20] Kuanqi Cai, Chaoqun Wang, Jiyu Cheng, Clarence W. De Silva, and Max Q.-H. Meng. “Mobile Robot Path Planning in Dynamic Environments: A Survey”. In: *arXiv preprint arXiv:2006.14195* (2020).

[Car+20] Yaniel Carreno, Èric Pairet, Yvan Petillot, and Ronald P. A. Petrick. “A decentralised strategy for heterogeneous auv missions via goal distribution and temporal planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 431–439.

[Cas+19] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. “Replanning for situated robots”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 29. 2019, pp. 665–673.

[Che+17] Mo Chen, Somil Bansal, Ken Tanabe, and Claire J. Tomlin. “Provably Safe and Robust Drone Routing via Sequential Path Planning: A Case Study in San Francisco and the Bay Area”. In: *CoRR* abs/1705.04585 (2017). arXiv: `1705.04585`. URL: `http://arxiv.org/abs/1705.04585`.

[CL07] Hugo Costelha and Pedro Lima. “Modelling, analysis and execution of robotic tasks using petri nets”. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2007, pp. 1449–1454.

[Cla+17] Daniel Claes, Frans Oliehoek, Hendrik Baier, and Karl Tuyls. “Decentralised Online Planning for Multi-Robot Warehouse Commissioning”. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. AAMAS ’17. 2017.

[Col+10] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. “Forward-chaining partial-order planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 20. 1. 2010.

[Col+19] Amanda Jane Coles, Andrew Ian Coles, Moises Martinez Munoz, Okkes Emre Savas, Thomas Keller, Florian Pommerening, and Malte Helmert. “On-board Planning for Robotic Space Missions using Temporal PDDL”. In: *11th International Workshop on Planning and Scheduling for Space (IWPSS)*. 2019.

[Cou06] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[CR20] George Council and Shai Revzen. “Fast Recovery of Robot Behaviors”. In: *arXiv preprint arXiv:2005.00506* (2020).

[CS19] Márcia M. Costa and Manuel F. Silva. “A survey on path planning algorithms for mobile robots”. In: *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. IEEE. 2019, pp. 1–7.

[DB20] Lasse Dissing and Thomas Bolander. “Implementing Theory of Mind on a Robot Using Dynamic Epistemic Logic”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Ed. by Christian Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, July 2020, pp. 1615–1621. DOI: 10.24963/ijcai.2020/224. URL: https://doi.org/10.24963/ijcai.2020/224.

[DCB17] Kun Deng, Yushan Chen, and Calin Belta. “An Approximate Dynamic Programming Approach to Multiagent Persistent Monitoring in Stochastic Environments With Temporal Logic Constraints”. In: *IEEE Transactions on Automatic Control* 62.9 (2017), pp. 4549–4563. DOI: 10.1109/TAC.2017.2678920.

[De 98] Luca De Alfaro. *Formal verification of probabilistic systems*. stanford university, 1998.

[Des+13] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. "P: safe asynchronous event-driven programming". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 321–332.

[Des+17] Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. "DRONA: A Framework for Safe Distributed Mobile Robotics". In: *Proceedings of the 8th International Conference on Cyber-Physical Systems.* 2017, pp. 239–248.

[DFR08] Eric V. Denardo, Eugene A. Feinberg, and Uriel G. Rothblum. "On occupation measures for total-reward MDPs". In: *2008 47th IEEE Conference on Decision and Control.* IEEE. 2008, pp. 4460–4465.

[Dij59] Edsger W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[Din+14] Xuchu Ding, Stephen L. Smith, Calin Belta, and Daniela Rus. "Optimal control of Markov decision processes with linear temporal logic constraints". In: *IEEE Transactions on Automatic Control* 59 (5 2014), pp. 1244–1257. DOI: 10.1109/TAC.2014.2298143.

[Duc+14] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. "Path planning with modified a star algorithm for a mobile robot". In: *Procedia Engineering* 96 (2014), pp. 59–69.

[EP18] Esra Erdem and Volkan Patoglu. "Applications of ASP in robotics". In: *KI-Künstliche Intelligenz* 32.2 (2018), pp. 143–149. DOI: 10.1007/s13218-018-0544-x.

[Fat+18] **Fatma Faruq**, Bruno Lacerda, Nick Hawes, and David Parker. "Simultaneous Task Allocation and Planning Under Uncertainty". In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'18).* IEEE, 2018, pp. 3559–3564.

[FBT97] D. Fox, W. Burgard, and S. Thrun. “The dynamic window approach to collision avoidance”. In: *IEEE Robotics Automation Magazine* 4.1 (1997), pp. 23–33. DOI: 10.1109/100.580977.

[Fel+17] Ariel Felner, Roni Stern Solomon, Eyal Shimony, Israel Eli Boyarski, Israel Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Glenn Wagner, and Pavel Surynek. “Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges”. In: *SoCS*. 2017.

[FKS06] Dave Ferguson, Nidhi Kalra, and Anthony Stentz. “Replanning with rrts”. In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* IEEE. 2006, pp. 1243–1248.

[Foe+17] Klaus-Tycho Foerster, Linus Groner, Torsten Hoefler, Michael Koenig, Sascha Schmid, and Roger Wattenhofer. “Multi-agent pathfinding with n agents on graphs with n vertices: Combinatorial classification and tight algorithmic bounds”. In: *International Conference on Algorithms and Complexity*. Springer. 2017, pp. 247–259.

[FS12] Eugene A. Feinberg and Adam Shwartz. *Handbook of Markov decision processes: methods and applications*. Vol. 40. Springer Science & Business Media, 2012.

[GD17] Meng Guo and Dimos V. Dimarogonas. “Task and Motion Coordination for Heterogeneous Multiagent Systems With Loosely Coupled Local Tasks”. In: *IEEE Transactions on Automation Science and Engineering* 14.2 (2017), pp. 797–808. DOI: 10.1109/TASE.2016.2628389.

[Geb+18] Martin Gebser, Philipp Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen, and Tran Cao Son. “Experimenting with robotic intralogistics domains”. In: *CoRR* abs/1804.10247 (2018). DOI: 10.29007/crx7.

[GKM10] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. “A survey of motion planning algorithms from the perspective of autonomous UAV guidance”. In: *Journal of Intelligent and Robotic Systems* 57.1 (2010), pp. 65–100.

[GM04] Brian P. Gerkey and Maja J. Matarić. “A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems”. In: *The International Journal of Robotics Research* 23.9 (Sept. 2004), pp. 939–954. ISSN: 0278-3649. DOI: 10.1177/0278364904045564.

[GMS17] Ivan Gavran, Rupak Majumdar, and Indranil Saha. “Antlab: A Multi-Robot Task Server”. In: *ACM TECS* 16.5s (2017), pp. 1–19.

[GSB17] Ricardo Grunitzki, Bruno Castro da Silva, and Ana L. C. Bazzan. “A Flexible Approach for Designing Optimal Reward Functions.” In: *AAMAS*. 2017, pp. 1559–1561.

[Guo+16] Meng Guo, Charalampos P. Bechlioulis, Kostas J. Kyriakopoulos, and Dimos V. Dimarogonas. “Hybrid Control of Multi-agent Systems with Contingent Temporal Tasks and Prescribed Formation Constraints”. In: *IEEE Transactions on Control of Network Systems* 5870.c (2016), pp. 1–1.

[GZ18] Meng Guo and Michael M Zavlanos. “Probabilistic Motion Planning under Temporal Tasks and Soft Constraints”. In: *IEEE Transactions on Automatic Control* 63.12 (2018), pp. 4051–4066. ISSN: 15582523. DOI: 10.1109/TAC.2018.2799561.

[Hah+17] Ernst Moritz Hahn, Vahid Hashemi, Holger Hermanns, Morteza Lahijanian, and Andrea Turrini. “Multi-objective robust strategy synthesis for interval Markov decision processes”. In: *International Conference on Quantitative Evaluation of Systems*. Springer. 2017, pp. 207–223.

[Haw+17] Nick Hawes, Christopher Burbridge, Ferdian Jovan, Lars Kunze, Bruno Lacerda, Lenka Mudrova, Jay Young, Jeremy Wyatt, Denise Hebesberger, Tobias Kortner, et al. “The strands project: Long-term autonomy in everyday environments”. In: *IEEE Robotics & Automation Magazine* 24.3 (2017), pp. 146–156.

[HJ89] Hans Hansson and Bengt Jonsson. “A framework for reasoning about time and reliability”. In: *1989 Real-Time Systems Symposium.* IEEE Computer Society. 1989, pp. 102–103.

[HK13] Ahmed Hussein and Alaa Khamis. “Market-based approach to multi-robot task allocation”. In: *2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR).* IEEE. 2013, pp. 69–74.

[HKM20] Ioana Hustiu, Marius Kloetzer, and Cristian Mahulea. “Distributed Path Planning of Mobile Robots with LTL Specifications”. In: *2020 24th International Conference on System Theory, Control and Computing (ICSTCC).* IEEE. 2020, pp. 60–65.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[How60] Ronald A. Howard. “Dynamic programming and markov processes.” In: (1960). DOI: 10.2307/1266484.

[Hua11] Shih-Chieh Huang. “New Heuristics for Monte Carlo Tree Search Applied to the Game of Go”. Ph.D. dissertation. Nat. Taiwan Normal Univ., Taipei, 2011, p. 106.

[HW12] Wiebe van der Hoek and Michael Wooldridge. “Logics for multiagent systems”. In: *Ai Magazine* 33.3 (2012), pp. 92–92. DOI: 10.1609/aimag.v33i3.2427.

[HZ01] Eric A. Hansen and Shlomo Zilberstein. “LAO^*: A heuristic search algorithm that finds solutions with loops”. In: *Artificial Intelligence* 129.1-2 (2001), pp. 35–62.

[Kal13] Rahul Kala. “Rapidly exploring random graphs: motion planning of multiple mobile robots”. In: *Advanced Robotics* 27.14 (2013), pp. 1113–1122.

[Kar+11] Sertac Karaman, Matthew R. Walter, Alejandro Perez, Emilio Frazzoli, and Seth Teller. “Anytime motion planning using the RRT”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1478–1483.

[KB91] Benjamin Kuipers and Yung-Tai Byun. “A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations”. In: *Robotics and autonomous systems* 8.1-2 (1991), pp. 47–63.

[KE12] Thomas Keller and Patrick Eyerich. “PROST: Probabilistic Planning Based on UCT”. In: *ICAPS*. 2012, pp. 119–127. URL: `www.aaai.org`.

[KF14] Kangjin Kim and Georgios Fainekos. “Revision of Specification Automata under Quantitative Preferences”. In: *CoRR* (2014).

[KH13] Thomas Keller and Malte Helmert. “Trial-based heuristic tree search for finite horizon MDPs”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 23. 2013, pp. 135–143.

[KM06] Nidhi Kalra and Alcherio Martinoli. “Comparative study of market-based and threshold-based task allocation”. In: *Distributed autonomous robotic systems 7*. Springer, 2006, pp. 91–101.

[KM20] Marius Kloetzer and Cristian Mahulea. “Path planning for robotic teams based on LTL specifications and Petri net models”. In: *Discrete Event Dynamic Systems* 30.1 (2020), pp. 55–79.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[Koe+06] S. Koenig, C. Tovey, M. Lagoudakis, V. Markakis, and D. Kempe. "The power of sequential single-item auctions for agent coordination". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI).* Vol. 2. AAAI'06. Boston, Massachusetts: AAAI Press, 2006, pp. 1625–1629. ISBN: 978-1-57735-281-5. URL: http://dl.acm.org/citation.cfm?id=1597348.1597457.

[Kol+11] Andrey Kolobov, Mausam Mausam, Daniel S Weld, and Hector Geffner. "Heuristic search for generalized stochastic shortest path MDPs". In: *Twenty-First International Conference on Automated Planning and Scheduling.* 2011.

[Koy90] Ron Koymans. "Specifying real-time properties with metric temporal logic". In: *Real-time systems* 2.4 (1990), pp. 255–299. DOI: 10.1007/bf01995674.

[KP13] M. Kwiatkowska and D. Parker. "Automated Verification and Strategy Synthesis for Probabilistic Systems". In: *ATVA.* 2013.

[Kra+17] Tomáš Krajník, Jaime P. Fentanes, Joao M. Santos, and Tom Duckett. "Fremen: Frequency map enhancement for long-term mobile robot autonomy in changing environments". In: *IEEE Transactions on Robotics* 33.4 (2017), pp. 964–977.

[KSD13] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. "A comprehensive taxonomy for multi-robot task allocation". In: *The International Journal of Robotics Research* 32.12 (2013), pp. 1495–1512.

[Kuh55] Harold W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[Kuk96] Yuji Kukimoto. *Introduction to Formal Verification.* Accessed: 6-Jan-21. 1996. URL: https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html.

[KV01] Orna Kupferman and Moshe Y. Vardi. "Model checking of safety properties". In: *Formal Methods in System Design* 19.3 (2001).

[KW12] Andrey Kolobov and Daniel S. Weld. “A theory of goal-oriented MDPs with dead ends”. In: *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*. 2012, pp. 438–447.

[KZ17] Yiannis Kantaros and Michael M. Zavlanos. “Sampling-based control synthesis for multi-robot systems under global temporal specifications”. In: *ICCPS*. 2017.

[KZ18] Yiannis Kantaros and Michael M. Zavlanos. “Distributed Optimal Control Synthesis for Multi-Robot Systems under Global Temporal Tasks”. In: *Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018* (2018), pp. 162–173. ISSN: 00189286. DOI: 10.1109/ICCPS.2018.00024.

[KZ20] Yiannis Kantaros and Michael M. Zavlanos. “Stylus*: A temporal logic optimal control synthesis algorithm for large-scale multi-robot systems”. In: *The International Journal of Robotics Research* 39.7 (2020), pp. 812–836.

[Lac+19] Bruno Lacerda, **Fatma Faruq**, David Parker, and Nick Hawes. “Probabilistic planning with formal performance guarantees for mobile service robots”. In: *The International Journal of Robotics Research* 38.9 (2019), pp. 1098–1123.

[Lag+04] Michail G. Lagoudakis, Marc Berhault, Sven Koenig, Pinar Keskinocak, and Anton J. Kleywegt. “Simple auctions with performance guarantees for multi-robot task allocation”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 1. IEEE. 2004, pp. 698–705.

[Lah+15] Morteza Lahijanian, Shaull Almagor, Dror Fried, Lydia E. Kavraki, and Moshe Y. Vardi. “This Time the Robot Settles for a Cost: A Quantitative Approach to Temporal Logic Planning with Partial Satisfaction”. In: *The Twenty-Ninth AAAI Conference (AAAI-15)* Aaai (2015), pp. 3664–3671.

[Lah+16] Morteza Lahijanian, Matthew R. Maly, Dror Fried, Lydia E. Kavraki, Hadas Kress-Gazit, and Moshe Y. Vardi. “Iterative temporal planning in uncertain environments with partial satisfaction guarantees". In: *IEEE Transactions on Robotics* 32.3 (2016), pp. 583–599.

[Lav98] Steven M. Lavalle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning.* Tech. rep. 1998.

[LDK13] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. "On the complexity of solving Markov decision problems". In: *arXiv preprint arXiv:1302.4971* (2013).

[Leo+17a] Francesco Leofante, Erika ábrahám, Tim Niemueller, Gerhard Lakemeyer, and Armando Tacchella. "On the Synthesis of Guaranteed-Quality Plans for Robot Fleets in Logistics Scenarios via Optimization Modulo Theories". In: *CoRR* abs/1711.04259 (2017).

[Leo+17b] Francesco Leofante, Erika ábrahám, Tim Niemueller, Gerhard Lakemeyer, and Armando Tacchella. "On the synthesis of guaranteed-quality plans for robot fleets in logistics scenarios via optimization modulo theories". In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE. 2017, pp. 403–410.

[Li+19] Minglong Li, Wenjing Yang, Zhongxuan Cai, Shaowu Yang, and Ji Wang. "Integrating Decision Sharing with Prediction in Decentralized Planning for Multi-Agent Coordination under Uncertainty." In: *IJCAI*. 2019, pp. 450–456.

[LK16] M. Lahijanian and M. Kwiatkowska. "Specification Revision for Markov Decision Processes with Optimal Trade-off". In: *IEEE Conference on Decision and Control (CDC)* Cdc (2016), pp. 7411–7418.

[LL18] An T. Le and Than D. Le. "Search-Based Planning and Replanning in Robotics and Autonomous Systems". In: *Advanced Path Planning for Mobile Entities* (2018), p. 63. DOI: `10.5772/intechopen.71663`.

[LL19] Bruno Lacerda and Pedro U. Lima. “Petri net based multi-robot task coordination from temporal logic specifications”. In: *Robotics and Autonomous Systems* 122 (2019), p. 103289.

[LLY13] Joohyung Lee, Vladimir Lifschitz, and Fangkai Yang. “Action language BC: Preliminary report”. In: *Twenty-Third International Joint Conference on Artificial Intelligence*. 2013.

[LPH14] Bruno Lacerda, David Parker, and Nick Hawes. “Optimal and dynamic planning for Markov decision processes with co-safe LTL specifications”. In: *IEEE International Conference on Intelligent Robots and Systems* (2014), pp. 1511–1516.

[LPH15a] Bruno Lacerda, David Parker, and Nick Hawes. “Nested Value Iteration for Partially Satisfiable Co-Safe LTL Specifications”. In: *2015 AAAI Fall Symposium Series*. 2015.

[LPH15b] Bruno Lacerda, David Parker, and Nick Hawes. “Optimal policy generation for partially satisfiable co-safe LTL specifications”. In: *IJCAI International Joint Conference on Artificial Intelligence* 2015-Janua (2015), pp. 1587–1593.

[LPM18] André Leite, Andry Pinto, and Aníbal Matos. “A Safety Monitoring Model for a Faulty Mobile Robot”. In: *Robotics* 7.3 (2018). ISSN: 2218-6581. DOI: `10.3390/robotics7030032`. URL: `https://www.mdpi.com/2218-6581/7/3/32`.

[LS09] Jan Karel Lenstra and David Shmoys. *The Traveling Salesman Problem: A Computational Study*. 2009.

[LS18] Lantao Liu and Gaurav S. Sukhatme. “A Solution to Time-Varying Markov Decision Processes”. In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1631–1638. DOI: `10.1109/LRA.2018.2801479`.

[Luc+19] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. “Formal specification and verification of autonomous robotic systems: A survey”. In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–41.

[LZ11] Boris Lesner and Bruno Zanuttini. “Efficient policy construction for MDPs represented in probabilistic PDDL”. In: *Twenty-First International Conference on Automated Planning and Scheduling*. 2011.

[Ma+16] Hang Ma, Sven Koenig, Nora Ayanian, Liron Cohen, Wolfgang Hoenig, T.K. Satish Kumar, Tansel Uras, Hong Xu, Craig Tovey, and Guni Sharon. “Overview: Generalizations of Multi-Agent Path Finding to Real-World Scenarios”. In: (2016).

[Mac+16] Thi Thoa Mac, Cosmin Copot, Duc Trung Tran, and Robin De Keyser. “Heuristic approaches in robot path planning: A survey”. In: *Robotics and Autonomous Systems* 86 (2016), pp. 13–28.

[Mad+17] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. “A survey of shortest-path algorithms”. In: *arXiv preprint arXiv:1705.02044* (2017).

[Man+19] Masoumeh Mansouri, Bruno Lacerda, Nick Hawes, and Federico Pecora. “Multi-robot planning under uncertain travel times and safety constraints”. In: *The 28th International Joint Conference on Artificial Intelligence (IJCAI19), August 10-16, Macao, China*. 2019, pp. 478–484.

[McD+98] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL - The Planning Domain Definition Language*. Tech Report CVC TR098-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[Mei+15] Li Meilun, Zhikun She, Andrea Turrini, and Lijun Zhang. “Preference Planning for Markov Decision Processes”. In: *AAAI Conference on Artificial Intelligence* (2015), pp. 3313–3319.

[MK12] Mausam and Andrey Kolobov. *Planning with Markov Decision Processes*. 2012. ISBN: 9781608458868.

[MK16] H. Ma and S. Koenig. “Optimal Target Assignment and Path Finding for Teams of Agents”. In: *AAMAS*. 2016.

[MK20] Kelsey Maass and Minsun Kim. “A Markov decision process approach to optimizing cancer therapy using multiple modalities”. In: *Mathematical medicine and biology: a journal of the IMA* 37.1 (2020), pp. 22–39.

[MKK17] Hang Ma, T. K. Satish Kumar, and Sven Koenig. “Multi-agent path finding with delay probabilities”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

[MLG05] H Brendan McMahan, Maxim Likhachev, and Geoffrey J Gordon. “Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees”. In: *Proceedings of the 22nd international conference on Machine learning*. 2005, pp. 569–576.

[MM19] Arjun Muralidharan and Yasamin Mostofi. “Path Planning for Minimizing the Expected Cost Until Success”. In: *IEEE Transactions on Robotics* 35.2 (2019), pp. 466–481. DOI: 10.1109/TRO.2018.2883829.

[MN04] Oded Maler and Dejan Nickovic. “Monitoring temporal properties of continuous signals”. In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166. DOI: 10.1007/978-3-540-30206-3_12.

[MS00] Mila Majster-Cederbaum and Frank Salger. “Correctness by construction: towards verification in hierarchical system development”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2000, pp. 163–180. DOI: 10.1007/10722468_10.

[MS03] Ian M. Mitchell and Shankar Sastry. “Continuous path planning with multiple constraints”. In: *42nd IEEE International Conference on Decision and Control (IEEE Cat. No. 03CH37475)*. Vol. 5. IEEE. 2003, pp. 5502–5507.

[MT99] Victor W. Marek and Miroslaw Truszczyński. “Stable models and an alternative logic programming paradigm”. In: *The Logic Programming Paradigm.* Springer, 1999, pp. 375–398. DOI: 10.1007/978-3-642-60085-2_17.

[NSS99] Ilkka Niemela, Patrik Simons, and Timo Soininen. “Stable model semantics of weight constraint rules”. In: *International Conference on Logic Programming and Nonmonotonic Reasoning.* Springer. 1999, pp. 317–331.

[NTD16] Alexandros Nikou, Jana Tumova, and Dimos V. Dimarogonas. “Cooperative task planning of multi-agent systems under timed temporal specifications”. In: *2016 American Control Conference (ACC).* IEEE. 2016, pp. 7104–7109.

[Nun+17] Ernesto Nunes, Marie Manner, Hakim Mitiche, and Maria Gini. “A taxonomy for task allocation problems with temporal and ordering constraints”. In: *Robotics and Autonomous Systems* 90 (2017), pp. 55–70.

[Omi+17] Shayegan Omidshafiei, Shih-Yuan Liu, Michael Everett, Brett T. Lopez, Christopher Amato, Miao Liu, Jonathan P. How, and John Vian. “Semantic-level decentralized multi-robot decision-making using probabilistic macro-observations”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA).* IEEE. 2017, pp. 871–878.

[Pal15] Andrew W. Palmer. “Belief Space Scheduling”. Doctor of Philosophy Ph.D. University of Sydney; Faculty of Engineering, IT; School of Aerospace, Mechanical, and Mechatronic Engineering, July 2015. URL: http://hdl.handle.net/2123/14280.

[PMP20] Đorđe Petrović, Radomir Mijailović, and Dalibor Pešić. “Traffic accidents with autonomous vehicles: type of collisions, manoeuvres and errors of conventional vehicles’ drivers”. In: *Transportation research procedia* 45 (2020), pp. 161–168.

[Pnu81] A. Pnueli. “The Temporal Semantics of Concurrent Programs”. In: *Theoretical Computer Science* 13 (1981).

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. First. New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN: 0471619779.

[Sah+14] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J. Pappas, and Sanjit A. Seshia. "Automated composition of motion primitives for multi-robot systems from safe LTL specifications". In: *IROS*. 2014.

[SBD18a] Philipp Schillinger, Mathias Bürger, and Dimos V. Dimarogonas. "Auctioning over probabilistic options for temporal logic-based multi-robot cooperation under uncertainty". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 7330–7337.

[SBD18b] Philipp Schillinger, Mathias Bürger, and Dimos V. Dimarogonas. "Decomposition of finite LTL specifications for efficient multi-agent planning". In: *Distributed Autonomous Robotic Systems*. Springer, 2018, pp. 253–267.

[SBD18c] Philipp Schillinger, Mathias Bürger, and Dimos V. Dimarogonas. "Improving Multi-Robot Behavior Using Learning-Based Receding Horizon Task Allocation". In: *Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018*. 2018. DOI: 10.15607/RSS.2018.XIV.031. URL: http://www.roboticsproceedings.org/rss14/p31.html.

[SBD18d] Philipp Schillinger, Mathias Bürger, and Dimos V. Dimarogonas. "Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems". In: *The International Journal of Robotics Research* 37.7 (2018), pp. 818–838. DOI: 10.1177/0278364918774135.

[SC20] Mohit Srinivasan and Samuel Coogan. "Control of mobile robots using barrier functions under temporal logic specifications". In: *IEEE Transactions on Robotics* (2020).

[Sch+15] Eric Schneider, Elizabeth I. Sklar, Simon Parsons, and A. Tuna Özgelen. “Auction-based task allocation for multi-robot teams in dynamic environments”. In: *Conference Towards Autonomous Robotic Systems*. Springer. 2015, pp. 246–257.

[Sch+16] Joris Scharpff, Diederik M. Roijers, Frans A. Oliehoek, Matthijs T. J. Spaan, and Mathijs M. De Weerdt. “Solving Transition-Independent Multi-agent MDPs with Sparse Interactions (Extended version) *”. In: *Proceedings of the 30th Conference on Artificial Intelligence (AAAI 2016)* (2016), pp. 3174–3180.

[SF97] Alexis Scheuer and Th Fraichard. “Continuous-curvature path planning for car-like vehicles”. In: *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS’97*. Vol. 2. IEEE. 1997, pp. 997–1003.

[Sha+15] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. “Conflict-based search for optimal multi-agent pathfinding”. In: *Artificial Intelligence* 219 (2015), pp. 40–66.

[SHB16] Marcel Steinmetz, Jörg Hoffmann, and Olivier Buffet. “Goal probability analysis in MDP probabilistic planning: Exploring and enhancing the state of the art”. In: *Journal of Artificial Intelligence Research* 57 (Oct. 2016), pp. 229–271. DOI: 10.1613/jair.5153.

[SLB20] Chuangchuang Sun, Xiao Li, and Calin Belta. “Automata Guided Semi-Decentralized Multi-Agent Reinforcement Learning”. In: *2020 American Control Conference (ACC)*. IEEE. 2020, pp. 3900–3905.

[Smi+11] Stephen L. Smith, Jana Tůmová, Calin Belta, and Daniela Rus. “Optimal path planning for surveillance with temporal-logic constraints”. In: *IJRR* 30.14 (2011).

[Spa] Matthijis Spaan. *MASPlan*. URL: http://masplan.org/problem_domains.

[SPS99] Richard S. Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[SS04] Matthijs T. J. Spaan and N. Spaan. "A point-based POMDP algorithm for robot planning". In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004.* Vol. 3. IEEE. 2004, pp. 2399–2404.

[SS17] E. V. Sorokin and A. V. Senkov. "Application of growing nested Petri nets for modeling robotic systems operating under risk". In: *IOP Conference Series: Earth and Environmental Science*. Vol. 87. 8. IOP Publishing. 2017, p. 082046.

[Ste+19] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks". In: (2019). URL: `http://arxiv.org/abs/1906.08291`.

[Str+06] Alexander L. Strehl, Lihong Li, Eric Wiewiora, John Langford, and Michael L. Littman. "PAC model-free reinforcement learning". In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 881–888.

[Str+20] Charlie Street, Bruno Lacerda, Manuel Mühlig, and Nick Hawes. "Multi-Robot Planning Under Uncertainty with Congestion-Aware Models". In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '20. Auckland, New Zealand: International Foundation for Autonomous Agents and Multiagent Systems, 2020, pp. 1314–1322. ISBN: 9781450375184.

[Tar72] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[TD16] Jana Tumova and Dimos V. Dimarogonas. "Multi-agent planning under local LTL specifications and event-based synchronization". In: *Automatica* 70.October (2016), pp. 239–248.

[Tei12] F. Teichteil-Königsbuch. "Stochastic Safest and Shortest Path Problems." In: *Association for the Advancement of Artificial Intelligence* (2012), pp. 1825–1831. URL: `www.aaai.org`.

[Tre+16] Felipe Trevizan, Sylvie Thiébaux, Pedro Santana, and Brian Williams. "Heuristic Search in Dual Space for Constrained Stochastic Shortest Path Problems". In: *International Conference on Automated Planning and Scheduling* Icaps (2016), pp. 326–334. URL: `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/13179/12694`.

[TTT17] Felipe W. Trevizan, Florent Teichteil-Königsbuch, and Sylvie Thiébaux. "Efficient solutions for Stochastic Shortest Path Problems with Dead Ends." In: *UAI*. 2017.

[Tum+13] Jana Tumova, Luis I. Reyes Castro, Sertac Karaman, Emilio Frazzoli, and Daniela Rus. "Minimum-violation LTL planning with conflicting specifications". In: *2013 American Control Conference* (2013), pp. 200–205.

[Tur+14] Matthew Turpin, Kartik Mohta, Nathan Michael, and Vijay Kumar. "Goal assignment and trajectory planning for large teams of interchangeable robots". In: *Autonomous Robots* 37.4 (2014), pp. 401–415.

[TV02] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[Ulu+13] Alphan Ulusoy, Stephen L. Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. "Optimality and Robustness in Multi-Robot Path Planning with Temporal Logic Constraints". In: *The International Journal of Robotics Research* 32.8 (July 2013), pp. 889–911.

[Van20] Robert J. Vanderbei. *Linear programming: foundations and extensions*. Vol. 285. Springer Nature, 2020. DOI: `10.1007/978-3-030-39415-8`.

[VKM17] Alberto Viseras, Valentina Karolj, and Luis Merino. "An asynchronous distributed constraint optimization approach to multi-robot path planning with complex constraints". In: *SAC*. 2017.

[VLB20] Cristian Ioan Vasile, Xiao Li, and Calin Belta. "Reactive sampling-based path planning with temporal logic specifications". In: *The International Journal of Robotics Research* 39.8 (8 June 2020), pp. 1002–1028. ISSN: 0278-3649. DOI: `10.1177/0278364920918919`.

[Wag18] Glenn Wagner. *Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning*. July 2018. DOI: `10.1184/R1/6723329.v1`.

[WC11] Glenn Wagner and Howie Choset. "M*: A complete multirobot path planning algorithm with performance bounds". In: *2011 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2011, pp. 3260–3267.

[WC17] Glenn Wagner and Howie Choset. "Path planning for multiple agents under uncertainty". In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 27. 1. 2017.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[Whi85] Douglas J. White. "Real applications of Markov decision processes". In: *Interfaces* 15.6 (1985), pp. 73–83.

[WHL17] Bo Wu, Bin Hu, and Hai Lin. "A learning based optimal human robot collaboration with linear temporal logic constraints". In: *arXiv preprint arXiv:1706.00007* (2017).

[WW21] Xi Vincent Wang and Lihui Wang. "A literature survey of the robotic technologies during the COVID-19 pandemic". In: *Journal of Manufacturing Systems* 60 (2021), pp. 823–836. ISSN: 0278-6125. DOI: `https://doi.org/10.1016/j.jmsy.2021.02.005`.

[Xu11] Ling Xu. *Graph planning for environmental coverage*. Carnegie Mellon University, 2011.

[YL04] Håkan L. S. Younes and Michael L. Littman. “PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects”. In: *Techn. Rep. CMU-CS-04-162* 2 (2004), p. 99.

[Zha+17] Shiqi Zhang, Yuqian Jiang, Guni Sharon, and Peter Stone. “Multirobot Symbolic Planning under Temporal Uncertainty”. In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. AAMAS ’17. São Paulo, Brazil: International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 501–510.

[Zlo06] Robert Michael Zlot. “An auction-based approach to complex task allocation for multirobot teams”. PhD thesis. Citeseer, 2006.

[ZS06] Robert Zlot and Anthony Stentz. “Market-based complex task allocation for multirobot teams”. In: *Transformational Science And Technology For The Current And Future Force: (With CD-ROM)*. World Scientific, 2006, pp. 169–176.

[ZSP08] Michael M. Zavlanos, Leonid Spesivtsev, and George J. Pappas. “A distributed auction algorithm for the assignment problem”. In: *2008 47th IEEE Conference on Decision and Control*. IEEE. 2008, pp. 1212–1217.