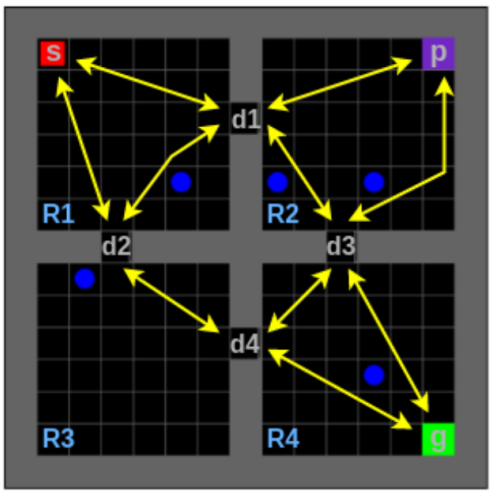
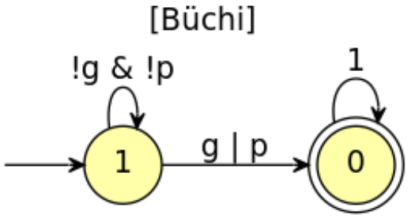
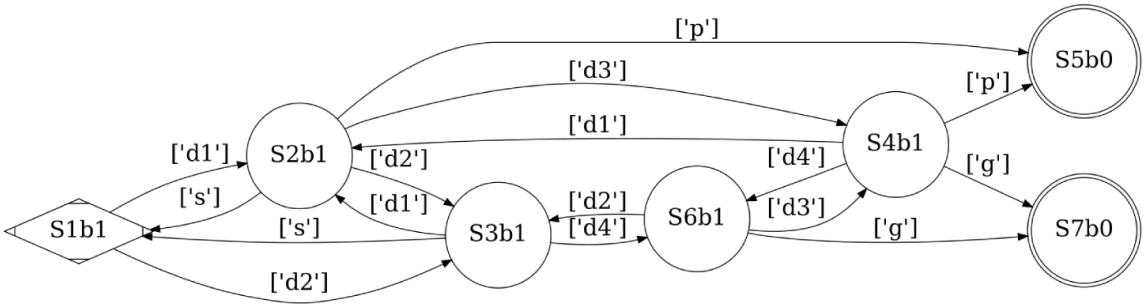
Combining Multi-Objective Planning with Reinforcement Learning to Solve Complex Tasks in Environments with Sparse Rewards

Cas van Rijn

Version of: 7 March, 2023







TU Delft

Combining Multi-Objective Planning with Reinforcement Learning to Solve Complex Tasks in Environments with Sparse Rewards

by

Cas van Rijn

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday March 21, 2023 at 10:15 AM.

Student number: 4690249
Project duration: February 2, 2022 – March 21, 2023
Thesis committee: Dr. A. Lukina, TU Delft, daily supervisor
Dr. M. T. J. Spaan, TU Delft, supervisor
Dr. F. A. Oliehoek, TU Delft

TU Delft

Preface

After a year of hard work, I can proudly present my master thesis. During this thesis, I explored the world of creating control policies for sequential decision-making processes. I have learned many new things and came into contact with parts of computer science that were unknown to me before I started this project. I would like to thank Dr. Anna Lukina, for her guidance, tips, and feedback while working on this project. Also thanks to Dr. Matthijs Spaan and Dr. Frans Oliehoek for agreeing on being part of my thesis committee.

I would like to thank my family who supported me during my thesis. Joop, Safrien, Kiki, Tom, and Norèn, thank you for supporting me and being there for me during the rough times. I'd also like to give a special thanks to Quentin Lee, Martin Li, and Wang Hao Wang, with whom I had the pleasure to share a large portion of my TU Delft career, both during the bachelor and master. Thanks to my long-time friend Tsun Wang Yu, who has been there for the entire journey from HBO to master.

Last but not least, shoutout to the fourth floor (and later on echo) gang for all the useful discussions, unuseful distractions, and all the time we spent together. I thoroughly enjoyed the company!

Cas van Rijn
Delft, March 2023

Abstract

There is a rise in demand for robots that can perform complex tasks such as doing laundry or sending a package from a warehouse to a client. To create a robot that can complete such complex tasks, a control policy is needed that steers the robot in the environment toward its goal. For the creation of such a control policy, the environment is often modeled as a sequential decision-making problem, which are problems where the goal is to find a sequence of actions in an environment that complete a certain task. Three techniques often used to solve sequential decision-making problems are formal methods, planning, and reinforcement learning. A particularly difficult type of sequential decision-making problem to solve is one in which the environment has sparse rewards, a large state space, and where the goal is to complete a complex task. Solely using formal methods, planning, or reinforcement learning often does not suffice in this type of environment since all techniques have a hard time dealing with some of the characteristics of this environment.

This research develops an approach that can solve complex tasks in sequential decision-making processes, i.e. tasks in stochastic environments with sparse rewards, a large state space, and where there are multiple optimization objectives. We do this by combining techniques from the three methods mentioned above (reinforcement learning, formal methods, and planning). Specifically, we create an approach called Multi-Objective Planning with Reinforcement Learning (MOPRL) that optimizes for two objectives (cost and success probability) and provide a lower bound on the probability that the expected success probability and expected average cost are within a certain range of the calculated values. In MOPRL we manually define subtasks in the environment and use reinforcement learning to learn a policy for each subtask. After this, multi-objective planning is used to find Pareto-optimal sequences of subtasks to execute that complete the global task.

We show that with this approach, we are able to outperform two other state-of-the-art reinforcement learning approaches that aim to solve complex tasks in environments, achieving a higher success probability, and having a marginally worse cost. MOPRL is most beneficial when dealing with choice tasks, such as *go to a or go to b*, since it is able to evaluate both options for all objectives. We provide a 90% confidence that the expected success probability is within 5% of the calculated success probability and an 84% confidence that the expected cost is within 5% of the calculated cost.

Acronyms

AI Artificial Intelligence
API Application Programming Interface
ARS Augmented Random Search

DDPG Deep Deterministic Policy Gradient

HLM High-Level Model

LTL Linear Temporal Logic

NN Neural Network

MDP Markov Decision Process
MILP Mixed Integer Linear Programming
MOPRL Multi Objective Planning with Reinforcement Learning

PAC Probably Approximately Correct
PPO Proximal Policy Optimization

RL Reinforcement Learning

Contents

List of Figures

List of Tables

1 Introduction

There is an increasing demand in the real world for robots that can perform complex tasks. According to a research by Harmo, Taipalus, Knuuttila, *et al.* (2005) [1] there is a need for robots that can do household tasks and provide other services for elderly people. This is due to a changing population structure in industrial nations, people are getting older and there will be fewer people available to take care of the elderly when needed. Another sector in which automation is in high demand is agriculture [2], [3], here the use of automated robots can have a high economic benefit, as well as help with the shortage of farm labor. Construction is also a sector in which the use of robots can be beneficial for productivity as well as for the enhancement of working conditions [4], [5]. One of the difficulties of creating such robots is creating a control policy that can steer the robot such that it completes its task in its environment. One of the main challenges of creating such a control policy is the fact that these robots need to operate in dynamic environments in which it is not always known what will happen. For instance, when a robot needs to do household tasks for the elderly, it needs to work in a shared workspace, where there are both humans and robots and contact between them can be expected, this can make the environment unpredictable [6], [7]. The challenge of creating a control policy that can perform complex tasks in an environment is often modeled as a sequential decision-making problem. A sequential decision-making problem is a problem where the goal is to find a sequence of actions that completes a task in an environment. An example of a sequential decision-making process is doing laundry, in which first, the laundry needs to be put in the washing machine, washed, taken out of the washing machine, dried, and maybe even ironed depending on the type of clothes. Another example is sending a package from a warehouse to a client, where one first needs to gather the package from the warehouse, package it, put the address on it, and mail it with the post service to the client.

There are many different techniques to create control policies for sequential decision-making problems. These methods are used to find which action is the best to be performed at which state in the environment in order to complete the task. Three of those methods are reinforcement learning (RL), formal methods, and planning. In RL, an agent learns how to solve the problem via interaction with the environment. The agent interacts with the environment by executing actions, collecting observations, and receiving a reward based on a given reward function. The agent receives a positive reward when it completes the task or a negative reward if it executes an action that has a negative effect. The agent uses these rewards to learn which action to execute at which state of the environment to optimize the overall long-term reward. This process results in a control policy that can be used to interact with the environment and complete the task the agent is trained for. Planning involves finding a sequence of actions that goes from an initial state to a goal state [8]. Most planning techniques use dynamic programming, which takes a problem and recursively breaks it down into subproblems, solving those subproblems and combining them together to find a solution to the main problem. Formal methods are mathematically rigorous techniques used for the specification, design, and verification of systems. Techniques from formal methods can be used to create a controller, given a qualitative specification and a model of the environment. Formal methods create such a controller by exhaustive searching for a sequence of actions that fulfills the specification, no matter what the environment does. A controller is a system that controls a plant or device such that it fulfills the specification in the environment. It

is similar to a control policy; it interacts with an environment in order to complete a certain task (the specification) by selecting actions to execute.

A particularly difficult type of sequential decision-making problem to solve is one in which the environment has sparse rewards and a large state space, and where the goal is to complete a complex task. Washing the laundry is an example of such a complex task since there are many smaller tasks that need to be completed (put clothes in the washing machine, wash the clothes, take it out, etcetera) in order to complete the entire task. The reason that such an environment is difficult to solve is that solely using RL, formal methods and/or planning will not suffice since certain characteristics of this environment are difficult to deal with for each method. RL has difficulties with environments with sparse rewards and learning a policy for complex tasks. Sparse rewards can be reformulated as the likelihood for an agent to discover a positive reward during interaction with the environment. If reaching a positive reward in the environment takes a long sequence of correct actions, and thus has a low likelihood of being discovered, it is possible the agent never discovers the positive reward during training and therefore fails to learn a good-performing policy [9]. Complex tasks are difficult to learn for RL since it is hard to provide the agent with a good reward function to learn a policy that completes the task [10]. Formal methods and planning both have the same difficulty, which is dealing with large state spaces. Both these methods need to explore the entire environment to create a controller/control policy, and therefore scale poorly when applied to environments with large state spaces [11], [12].

Multiple works have proposed different methods of solving complex tasks in environments with sparse rewards. A popular approach is to use Linear Temporal Logic (LTL) to specify tasks for RL [13]. These approaches typically generate a reward function from the task specification and use an RL algorithm to learn a policy. In particular, Li, Vasile, and Belta (2016) [14] proposes TLTL, a version of LTL that focuses on finite-time trajectories. They define quantitative semantics for TLTL and use this to transform temporal logic formulae into real-valued reward functions. Vaezipoor, Li, Icarte, *et al.* (2021) [15] use LTL to describe tasks over a domain-specific language. They label states in the environment and use this to check if the task is completed or whether the agent has made some progression in completing the task and use this to generate rewards. Camacho, Toro Icarte, Klassen, *et al.* (2019) [10] combine a reward machine with reward shaping, they show how LTL specification can be automatically translated into a reward machine, which is a machine that generates reward functions based on the state of the environment and combine this with reward shaping. Other approaches such as HRM by Icarte, Klassen, Valenzano, *et al.* (2022) [16] and HIRO by Nachum, Gu, Lee, *et al.* (2018) [17] use a hierarchical approach and decompose a problem into easier subproblems, and solve those subproblems on a low level using RL. They then use a higher-level policy that selects which subproblems to solve in order to solve the main problem. Other works, as by Francis, Faust, Chiang, *et al.* (2019) [18] and Wohlke, Schmitt, and Hoof (2021) [19] show that complex sparse reward tasks can be solved successfully in environments by combining planning and RL. They use RL to learn a transition model and use planning in this model to complete navigation tasks. Another idea often used to solve complex tasks, is the idea of policy sketches [20], in which a sequence of subtasks is designed in order to achieve a goal or task. Jothimurugan, Bastani, and Alur (2020) [21] and Eysenbach, Salakhutdinov, and Levine (2019) [22] combine policy sketches with RL and high-level planning. They manually define subtasks for which they train agents with RL and use high-level planning to find a sequence of subtasks that achieve a reachability goal.

Most of the above approaches that aim to solve complex tasks in an environment optimize for a single objective such as success probability or the reward received from the environment in order to create a controller or policy that can complete this task. However, in many sequential decision-making processes, there can be multiple objectives that are of importance, such as traffic control systems that should minimize latency and maximize throughput [23] or a robot that needs to do certain tasks in a hotel, where it needs to do them in a reasonable amount of time (cost) but also needs to be efficient with its resource (battery) [24]. Most existing research on solving complex tasks also focuses on deterministic environments, and not much is known about the influence of stochasticity in the environment on solving complex tasks in an environment.

Inspired by the idea of using policy sketches and research by Neary, Verginis, Cubuktepe, *et al.* (2022)

[25] and Jothimurugan, Bastani, and Alur (2020) [21] who divide the environments into smaller subtasks, use RL to solve those subtasks and use planning to find a sequence of subtasks in order to solve simple reachability tasks in environments with sparse rewards. We take a similar approach in which we manually define subtasks and combine techniques from planning, RL, and formal methods together to find a sequence of subtasks that can complete complex tasks in an environment.

1.1. Problem statement

This research's main goal is to create a controller, or control policy that can solve complex tasks which need to be optimized for multiple objectives in stochastic environments with sparse rewards and a large state space by combining techniques from RL, planning, and formal methods. This is a unique setting that has not been researched to our knowledge. When using RL to create control policies, RL optimizes according to its reward function, this may result in a policy that gathers a lot of rewards but has a low probability of completing the main task [26], [27]. Therefore, after we created a controller we want to provide guarantees for all the objectives that are of importance.

In this research, we study a concrete problem setting. We consider a discrete environment with stochastic moving obstacles. The goal is for the agent to complete a certain task while avoiding obstacles in the environment. These stochastic obstacles increase the state space, making it infeasible for planning/formal methods alone. The exact environment used in this research is a grid-world room environment, where the entire environment is divided into rooms. These rooms are connected by narrow doors/passages, see Figure 1.1 as an example of such an environment. This type of environment is often used for research on solving difficult tasks [28].

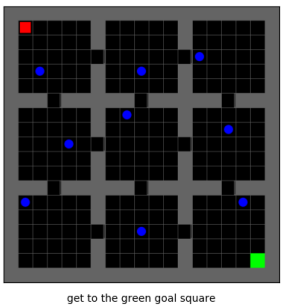


Figure 1.1: Example of a grid-world room environment in which the agent's goal is to get to the green square while avoiding the moving obstacles

The tasks considered are tasks that can be described with finite-LTL. Finite-LTL tasks are tasks that can be completed in a finite amount of time and only need to be completed once. LTL is a formal method often used for defining tasks with modalities referring to time, such as "eventually" and "never". Examples of LTL statements are "eventually it will rain", "there will never be a crash" and "eventually pick up coffee and then eventually bring it to the customer". For the multi-objective setting, this research considers tasks with two objectives, the success probability of completing the task and the cost (number of actions) of completing the task. The goal is to maximize the success probability and minimize the cost.

1.2. Motivational example

As a motivating example, we use the grid in Figure 1.2. This is a discrete environment where the agent (red square) starts at the top left corner and can move either one tile north, east, south, or west each timestep. In this environment, there are stochastic moving obstacles (blue dots) that also move one tile north, east, south, or west each timestep. The environment is divided into four rooms, the agent can move from one room to another through a door, which is exactly one tile in width. The task the agent needs to learn is "go to the green square or go to the purple square".

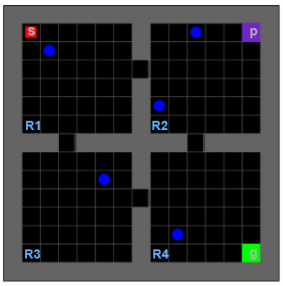


Figure 1.2: Example of an environment in which the agent needs to go to either purple or green while avoiding the moving obstacles.

Due to the random moving obstacles, the state space of this environment is very large since with every added obstacle the state space grows exponentially. Therefore, using solely planning or formal methods is not applicable.

This task is difficult to learn for solely RL since the agent needs to navigate from room to room, this requires the agent to be in front of the door and then move two steps in the same direction to go to the next room, this requires a specific sequence of actions and makes the rewards sparse. it is possible the agent never explores the correct sequence of actions during training and therefore fails to learn to complete the task.

If the agent would go to the green square, it can first go to room 3, then to room 4 and in room 4 navigate to the green square. With this path the agent evades room 2, which has two obstacles, resulting in a policy with a higher success probability. However, if the agent goes to the purple square, it requires fewer steps although this path is less safe, therefore it is beneficial to evaluate the task on multiple objectives.

1.3. Research questions

In order to achieve the main goal of solving LTL-tasks in an environment with sparse rewards while optimizing for both cost and success probability, we define the following main research question:

How can we solve LTL-tasks which need to be optimized for multiple objectives in environments with sparse rewards by combining ideas from RL, planning and formal methods?

To create an approach inspired by the idea of policy sketches [20] and apply it in the environment described in section 1.1, we define the following sub-research questions:

- How can we leverage the idea of policy sketches [20] to solve LTL-tasks in an environment by manually defining subtasks and finding a sequence of subtasks that complete the LTL-task?
- How can we solve LTL-tasks efficiently, optimizing for both success guarantee as well as cost?
- How can we deal with stochasticity in environments, while guaranteeing the global task is not violated?
- How can we give guarantees for both the success probability, as well as the cost of the controller, which metrics should be used?

1.4. Thesis contributions

This research contributes by providing the following:

- An approach, called MOPRL (Multi-Objective Planning with Reinforcement Learning) that can be used to solve LTL-tasks in environments with sparse rewards by manually defining subtasks. RL is used to learn how to complete these subtasks, and planning is used to find sequences of subtasks that can complete the LTL-task.
- Apply this approach in a setting where there are multiple objectives that need to be considered, instead of having only a single objective such as reward or success probability.
- Provide guarantees on the performance of MOPRL for all the objectives that are considered.
- Empirical assessment of what the influence is of stochasticity on completing a complex LTL-task to see if there are cases in which stochasticity can interfere with completing the LTL-task.
- Provide insights on the use of MOPRL compared to other approaches and highlight situations in which MOPRL is more appropriate.

1.5. Outline

This thesis is organized as follows; Chapter 2 contains background information about the techniques used in this thesis. Chapter 3 describes the approach used in this research to solve LTL-tasks in an environment. We go over every sub-question and explain how techniques from RL, planning, and formal methods are combined to be able to solve complex tasks in environments. In chapter 4 we present an overview of works that also aim to solve complex tasks in environments, explain the techniques they use and highlight the difference between their and our approach. Chapter 5 presents the experimental results of MOPRL and a comparison with other approaches, it also describes which libraries are used to implement MOPRL. Chapter 6 goes over the advantages and limitations of MOPRL and critically looks at the evaluation. Chapter 7 contains the conclusions and possible topics for future work.

2

Background

This chapter provides the background knowledge relevant to the techniques used in this thesis. Section 2.1 explains Markov Decision Process (MDP), which is a model commonly used to represent sequential decision-making problems. Section 2.2 is about graphs, which is another technique used to represent environments or abstractions of environments. Section 2.3 is an in-depth description of Linear Temporal Logic (LTL), the logic used to represent tasks in this research. Section 2.4 explains Büchi automata, which is a type of automata used to represent LTL formulae. After introducing the techniques used to represent environments and tasks, we discuss multi-objective pathfinding and the Pareto front in section 2.5. We also discuss Martins algorithm in section 2.6, which is a planning technique used to solve multi-objective pathfinding problems. Section 2.7 provides information about reinforcement learning and some of its variants. Section 2.8 explains Hoeffding's inequality, which is a technique used to provide a confidence bound on the performance of a controller.

2.1. Markov Decision Process

Markov Decision Processes (MDP) is a framework used for modeling stochastic environments and sequential decision-making processes [29]. It defines the actions an agent can make in an environment, the effect of these actions, and the states of the environment.

An MDP is represented by a Tuple (S, A, P, P_0, R) [30], where:

- S: A set of states
- A: A set of actions
- $P(\cdot|s,a)$: A probability transition distribution over states in the next time step, based on the environment being in state s and the agent executing action a
- P_0: A probability distribution according to the initial state
- $R(s,a)$ or $R(s)$: A reward function providing a reward when the agent takes action a in state s or when it reaches state s

One important characteristic of MDPs is the Markov Property. The Markov Property means that the next state at $t+1$ only depends on the state at time t, see Equation 2.1 for the formal definition.

$$Pr(R_{t+1}, S_{t+1}|S_t, A_t) = Pr(R_{t+1}, S_{t+1}|S_0, A_0, S_1, T_1, ..., S_t, A_t) \tag{2.1}$$

An example of an environment, a graphical representation of it, and the formal MDP can be found in Figure 2.1.

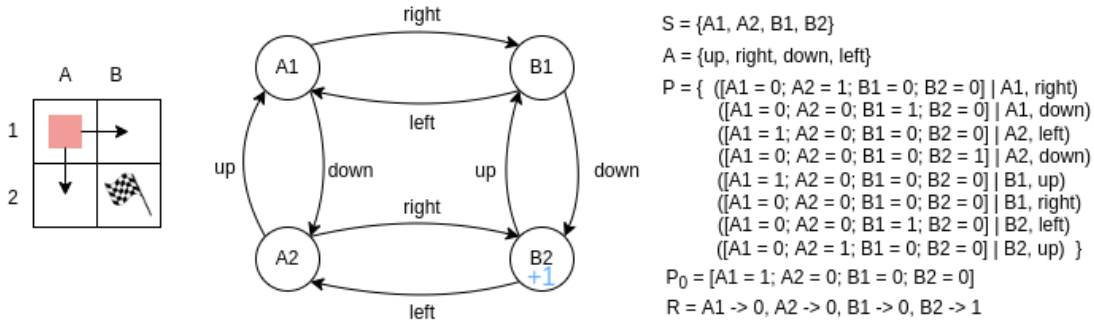


Figure 2.1: Example of an environment, where the red square is the agent and starts in the top left corner (A1), and needs to navigate to the bottom right corner (B2) to get a reward of +1. A graph representation of the environment, and the MDP (S, A, P, P_0, R) representing the environment

Observability
There are two types of observability in environments, fully observable and partially observable. In a fully observable environment, the RL agent has access to all information regarding the current state of the environment. In a partially observable environment, the agent only receives partial information from the environment. For example, it only receives information in a small area around itself instead of the entire environment. When an environment is only partially observable, the MDP is extended with Ω, a set of observations, and O, a set of observation probabilities $P(o|s,a)$ based on the environment being in state s and the agent executing action a.

Labeled Markov Decision Process
in some cases, an MDP is extended with a set of atomic propositions $\prod$ and a labeling function $L(s,a) \rightarrow 2^{\Pi}$ or $L(s) \rightarrow 2^{\Pi}$. These propositions represent properties of an environment that can either be True or False [31]. The labeling function can relate states or state-action pairs to events happening in the environment.

Path
A path in an MDP is a sequence of transitions, $path = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ from state s_0 to s_n where a_1 is the action executed in s_0 to transition to the next state [32].

2.2. Graphs

Graphs are mathematical structures in which points are connected to each other via edges, in graph theory these points are called vertices. Graphs can be used to represent or model different types of maps/networks such as a roadmap or electronic network, see Figure 2.2.

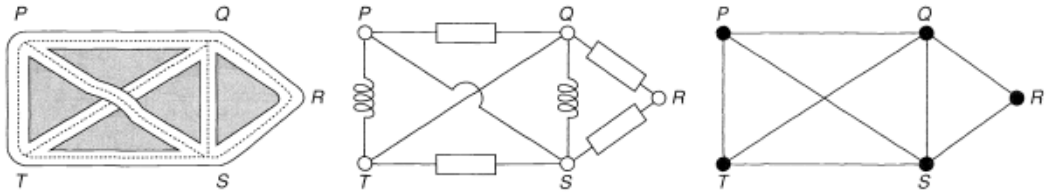


Figure 2.2: Example of a roadmap, an electronic network, and a graph representation of these two scenarios [33]

The formal notation of a graph is $G = (V, E)$ where V is the set of vertices and E is the set of edges. An edge $e \in E$ is a tuple of two vertices $\{x, y\}$ which connects the two vertices with each other, where $x \neq y$ and $x, y \in V$. In Figure 2.2 the graph $G = (V, E)$ exists out of the set $V = (P, Q, R, T, S)$ and $E = (\{P,Q\},\{P,S\},\{P,T\},\{Q,T\},\{Q,R\},\{Q,S\},\{R,S\},\{T,S\})$.

A graph can be directed or undirected, in an undirected graph it is possible to travel both ways over an edge, in a directed graph the edges are represented by an arrow instead of a line, indicating that it is

only possible to traverse the edge in that direction. The graph in Figure 2.2 is an undirected graph and Figure 2.3 is an example of a directed graph.

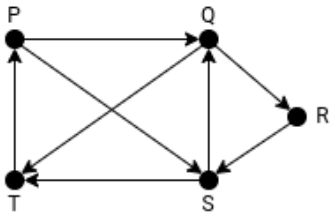


Figure 2.3: Example of a directed graph, edges can only be traversed in the direction of the arrow

Edges can have one or multiple values assigned to them, depending on the scenario the graph is used in. For example, when the graph is a representation of an electronic network, each edge can have a resistance associated with them. When the graph represents a roadmap, each edge can have a value assigned representing the length of the road between the two vertices.

2.3. Linear Temporal Logic

Linear Temporal Logic (LTL) is a modal logic with modalities referring to time, LTL is used to formally describe tasks or requirements for a system. Modal logic is a collection of formal specifications that are used to describe expressions such as *”eventually”*, *”never”* or *”possibly”*. These modalities can be used to formalize statements such as *”eventually it will rain”* or *”the system will never fail”*. These type of statements resolve into either True or False and can be checked given a model of the environment they are used in. LTL plays a big part in model checking of systems by formally defining requirements which can then later be verified [34].

LTL is mostly used to formally describe safety and liveness requirements of a system which then can be verified. Safety requirements are properties that say *”bad thing will never happen”* and liveness properties say *”something good will eventually happen”*. LTL is only used for qualitative properties, this means that plain LTL cannot be used to describe quantitative goals such as ”what is the probability that it will rain in 2 hours” [11]. The LTL syntax [35] can be found in Table 2.1.

LTL	Name	Meaning
f	-	the formula f is True immediately
$X\ f, \circ f$	next	f is True in the next step
$F\ f, \Diamond f$	eventually	f will eventually be True
$G\ f, \Box f$	globally	f is always True
$f\ U\ g$	strong until	f has to be True until g becomes True (and g will become True)
$f\ W\ g$	weak until	f has to be True until g becomes True (f should stay True if g never becomes True)
$f\ M\ g$	strong release	f has to be $True$ until $f \wedge g$ becomes True (and $f \wedge g$ will become True)
$f\ R\ g$	weak release	g has to be True until $f \wedge g$ becomes True (f should stay True if $f \wedge g$ never becomes True)
$f \vDash g$	represented by	f and g are equal and can be represented by each other
$\neg f$	not	f must not be True
$f \wedge g$	and	both f and g must be True
$f \vee g$	or	either f or g must be True
$f \rightarrow g$	implication	if f is True, then g must also be True
$f \leftrightarrow g$	equivalence	f and g must be equal (if f is True, g must also be True and vice versa)
$\top$	True	symbol representing True
$\bot$	False	symbol representing False

Table 2.1: LTL syntax and the meaning of every symbol

An example of LTL is: $\Box\neg crash \wedge \Diamond reach\_goal$. This means globally there never is a crash and eventually, the goal is reached. There is also LTL syntax for historical events, but since this thesis only focuses on future paths, the syntax for these events is excluded.

Standard LTL statements evaluate to either True or False. However, a probabilistic extension of LTL, *probabilistic LTL* exists, which instead of evaluating whether a certain property will be True or False, tries to describe the probability a certain property is True in a probabilistic environment. This is useful for environments that display stochastic behavior and where the LTL might be partially infeasible [36]. In probabilistic LTL the question isn't whether a certain property is $True$ or $False$, but whether it is satisfied with at least a certain probability, i.e. $P_{>0.95}\Box\neg crash \wedge \Diamond reach\_goal$ is the property that requires that in 95% of all cases, there never is a crash and eventually the goal is reached. This example is still an assertion that resolves to either $True$ or $False$, however, it is also possible to describe numerical properties in probabilistic LTL, i.e. $P_{max=?}$ and $P_{min=?}$ corresponding to the maximum and minimum probability a certain property is $True$ [32].

Finite LTL (LTL_f) is a special version of LTL, while regular LTL is used to describe properties that run in infinite time, finite LTL is used for properties with finite traces. The main difference is that in finite LTL the property only needs to be satisfied once, while in regular LTL the property needs to be satisfied indefinitely [37].

2.4. Büchi automata

A Büchi Automata is a theoretical automata that rejects or accepts infinite inputs. Let us consider the Büchi automata in Figure 2.4, state 1 is the start state and state 0 is the accepting state, which needs to be visited infinitely often. This Büchi automata accept inputs in the form of $gg*$ meaning it only accepts inputs where g happens at least once, and only g happens.

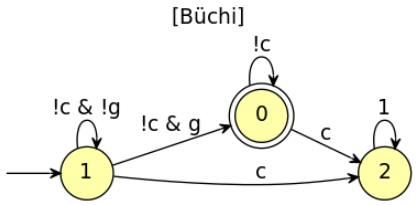


Figure 2.4: Example of a Büchi automata

Figure 2.4 is a complete Büchi automata, a Büchi automata that accepts the same input is represented with the automata in Figure 2.5, which is a simpler automata. The difference between the complete and simple representation is that in the complete representation every possible transition is displayed, so it is always possible to move forward in the automata. In the simple representation only transitions that potentially lead to an accepting state are represented. If a simple Büchi automata is in a state where it receives an input for which it has no transition, the input is rejected. While complete Büchi automata are easier to comprehend, simple Büchi automata have the advantage of having fewer states and transitions.

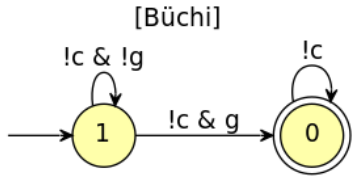


Figure 2.5: Example of a Büchi automata equivalent to the Büchi automata in Figure 2.4, but represented as a simple automata

The formal description of a Büchi automata is the same as that of a standard automata, namely: $A = (Q, \sum, \delta, q_0, F)$, where:

- Q: set containing all states in A
- $\sum$: finite set called the alphabet of A, containing all possible characters in the input of A
- δ: $Q \times \sum \rightarrow Q$: a set of transitions from one state to another
- q_0: the start state of A
- F: set containing all accepting states of A

Büchi Automata can be used to represent LTL properties, since for every LTL there exist a Büchi Automata accepting all sequences that satisfy the specified LTL [38]. Büchi automata representing a certain LTL can be used to check paths in a model and see if they satisfy the LTL property or not. The Büchi automata in Figure 2.4 can be seen as the representation of the LTL $\square\neg crash \wedge \Diamond reach\_goal$, where $c = crash$ and $g = reach\_goal$. So every path that never crashes and eventually reaches the goal is an input that will be accepted by the Büchi automata.

2.5. Multi-objective path finding and Pareto front

Shortest path finding is a problem that has been studied for more than 50 years. The definition of the shortest path problem is: Given a graph $G = (V, E)$, a start location $s \in V$, and a destination $d \in V$, what is the minimum distance path in G from s to d?

The shortest path problem can be solved in polynomial time with the Dijkstra algorithm. In the Dijkstra algorithm, the Bellman optimality principle is used. The Bellman optimality principle states that an initial segment of an optimal path is in itself optimal, so if there is a shortest path from A to C in a graph G, where the path is $A \rightarrow B \rightarrow C$, then following the Bellman optimality principle, $A \rightarrow B$ is also a shortest path, see Figure 2.6 as an example.

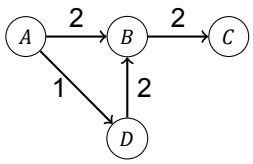
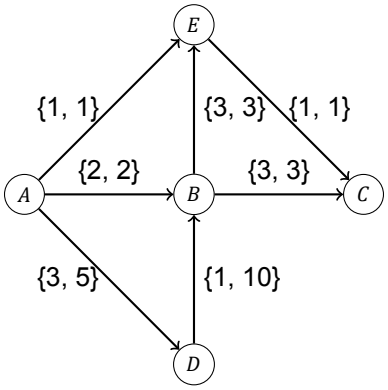
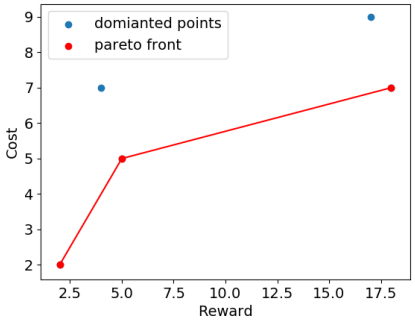


Figure 2.6: The Bellman optimality principle, since $A \rightarrow B \rightarrow C$ is a shortest path, so is $A \rightarrow B$ and $A \rightarrow D \rightarrow B$ is sub-optimal

In single objective path-finding problems, most of the time there is only one optimal path unless there are multiple paths that all achieve the optimal cost. When dealing with multiple objectives, it is possible that there are multiple paths, that are so-called *Pareto optimal*. When dealing with multiple objectives, there would only be one optimal path when that path provides the best value for all objectives compared to all other paths. However, it is possible that one path is optimal for one objective, while another path is optimal for another objective, therefore both paths can be viewed as optimal. Pareto optimality is used to describe optimal solutions in a multi-objective setting. A solution is Pareto optimal if it gives the best value for at least one objective, while not being worse in the other objectives compared to all other solutions. Points that are worse in all objectives compared to any other point are called dominated and are sub-optimal. The set of all non-dominated points is called the *Pareto Front*. See Figure 2.7 as an example of a multi-objective shortest path problem where the start vertex is A and the destination is C. Figure 2.7b contains the Pareto optimal paths for this problem.



(a) A graph with multi-objective edges. each edge has a $\{cost, reward\}$ associated, and the goal is to maximize the reward and minimize the cost



(b) The Pareto front associated with the graph on the left, optimal paths are $A \rightarrow E \rightarrow C$, $A \rightarrow B \rightarrow C$ and $A \rightarrow D \rightarrow B \rightarrow C$

Figure 2.7: (a) A multi-objective graph. (b) values of all possible paths visualized, of which some are dominated (sub-optimal), indicated with blue and the Pareto optimal points are in red

A subclass of the multi-objective setting is the bicriterion problem, the bicriterion problem is the case in which there are only two objectives. The bicriterion problem and most notably the bicriterion shortest path problem has extensively been studied and there are multiple different ways of solving it via approximation or with exact methods [39]–[41].

2.6. Martins algorithm

Martins algorithm [42] is an algorithm designed to solve the multi-objective shortest path problem. Formally, the multi-objective shortest path problem can be described as:

given a graph $G = (V, E)$, where each edge $e \in E$ has n objectives that are assigned to them $e = (O_1, O_2, ..., O_n)$, a staring vertex $s \in V$ and a goal vertex $t \in V$. Find all non-dominated paths from s to t in G if such paths exist.

Martins algorithm main idea is as follows: Each vertex in the graph has a list of permanent and temporary labels. At each iteration, the algorithm selects the minimum lexicographic label from all the sets of temporary labels, converts it to a permanent label, and propagates the information contained in this label to all successors, the successors check if the new label formed from this permanent label is non-

dominated by any other label it knows about and if so, adds it to its temporary label set. The procedure stops when there are no more temporary labels. A label l_j is represented by: $l_j = [z_1, ..., z_n, i, h]$ where $z_1, ..., z_n$ is a vector containing all values for the objectives that need to be optimized for, i is the predecessor vertex from which the label l_j is created and h is the index of the permanent label in i that was used to construct l_j [43].

After Martins algorithm has concluded, each permanent label is a unique Pareto-optimal path. To determine a Pareto-optimal path from the start vertex s to the end vertex t, take a permanent label from t and extract the values i and h from the label. These values indicate the hth label from vertex i that produced the current path, by tracing this path back to s, a Pareto-optimal path from s to t is found. The pseudo-code of Martins algorithm can be found in algorithm 1.

Algorithm 1: Martins algorithm for the multi-objective shortest path problem

Require: graph $G = (V, E)$
Require: C, objective costs for all edges $(i, j) \in E$
Require: start vertex $s \in V$ and goal vertex t$\in E$
Ensure : All Pareto-optimal paths from s to t in G

l_i: is a label of vertex i;
lt_i: is the entire list of temporary labels of vertex i;
lp_i: is the entire list of permanent labels of vertex i;
$z_{q,h}^p$: is the p^{th} objective of a permanent label of vertex q in position h;
Δ: the dominance relation (if $z \Delta z'$ then z is dominated by z')
$--|initialitation$;
$lt_i, lp_i \leftarrow \emptyset$;
$lt_s \leftarrow \{0, ..., 0. \perp, \perp\}$;

$--|iteration$;
while $(\cup_{i \in V} lt_i \neq \emptyset)$ **do**
  $--|find\ minimum\ lexiographic\ label \in lt_i, \forall_i \in V$;
  $l_q \leftarrow minlex\{\cup_{i \in V} lt_i\}$;
  $--|Move\ the\ selected\ label\ from\ the\ temporary\ list\ to\ the\ permanent\ list$;
  $lt_q \leftarrow lt_q \backslash \{l_q\}; lp_q \leftarrow lp_q \cup \{l_q\}$;
  $--|store\ the\ position\ of\ label l_q\ from\ list\ lp_q$;
  $h \leftarrow card(lp_q)$;
  $--|label\ all\ the\ successors\ of\ q$;
  for $j \in V | (q, j) \in E$ **do**
    $--|compute\ l_j,\ the\ current\ label\ of\ vertex\ j$;
    $l_j \leftarrow [z_{q,h}^1 + c^1(q, j), ..., z_{q,h}^k + c^k(q, j), q, h]$
    $--|verify\ that\ there\ is\ no\ label\ of\ vertex\ j\ dominating\ l_j$;
    if $\forall_{j' \in \{lt_j \cup lp_j\}} | l_j' \Delta l_j$ **then**
      $--|store\ the\ label\ l_j\ of\ vertex\ j\ as\ temprary$;
      $lt_j \leftarrow lt_j \cup l_j$;
      $--|delete\ all\ temporary\ labels\ of\ vertex\ j\ dominated\ by\ l_j$;
      $lt_j \leftarrow lt_j \backslash \{l_j' \in lt_j | l_j' \Delta l_j\}$
    end
  end
end

2.7. Reinforcement learning

Reinforcement learning (RL) is an Arificial Intelligence (AI) technique addressing the problem of how an agent can learn to approximate an optimal policy that completes a certain task via interaction with the environment [44]. The agent interacts with the environment for a number of steps, and in each step, the agent executes an action that changes the state of the environment. The agent then observes a

new state and receives a reward, see Figure 2.8 for an example. The agent's behavior should choose actions that optimize the long-term reward it receives. It can learn such behavior over time by trial and error [45].

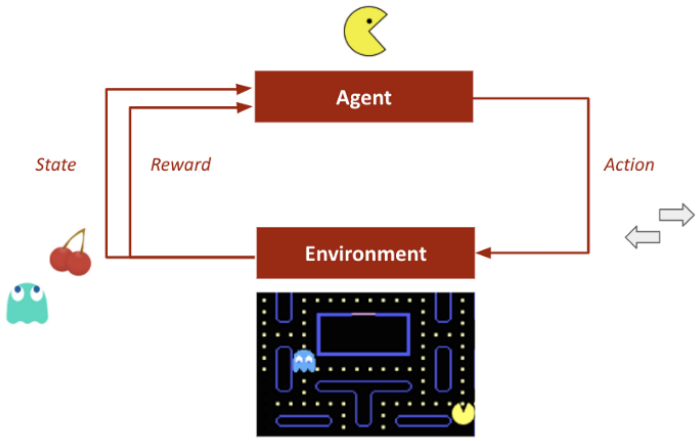


Figure 2.8: RL agent interacting with an environment in the game of Pacman [46]

The RL system exists out of 4 main components; a policy, a reward signal, a value function and a model of the environment [47].

A policy, $\pi : S \rightarrow p(A)$, is what defines the agent's behavior, simply put, the policy is a mapping of incoming states to possible actions. The policy is incrementally improved, learning what the best action is for a certain state. Policies come in many different forms, they can be a simple look-up table mapping all possible states to actions or it may be a neural network, which is a computing system that aims to learn a function that outputs the best action for each possible state.

A reward signal, $R : S \rightarrow \mathbb{R}$, defines the rewards in an environment. In each step, the RL agent receives a single number from the environment, the reward. This reward indicates how good the action the agent made is, it is the agent's purpose to maximize the reward it receives in the long run. Reward signals can both be positive as well as negative, positive rewards indicate an action is good, while negative rewards indicate an action is bad.

The value function, $Q^{\pi}(s)$, is what determines what is good in the long run. Reward signals are immediate rewards of environment states, but an agent needs to learn how to reach a state that can give a positive immediate reward, this is what the value function is for, it calculates the expected cumulative reward. It is possible that an agent first needs to traverse a number of states with no positive reward, in order to be able to reach the state space with positive rewards. The value function makes it so that the agent will know that there are positive rewards in the future if it executes a certain action in that state. An example of a value function that is often used, the Bellman Equation is given in Equation 2.2.

$$Q^{\pi}(s) = max_{a \in A}\left(R(s) + \gamma \sum_{s_{t+1} \in S} [(P(s_{t+1}|a,s)Q^{\pi}(s+1)]\right) \tag{2.2}$$

This equation states that the value in state s is the reward the agent receives in that state plus a discounted future reward. The γ is the discount factor. The higher γ, the more it takes future rewards into account. The future reward is a sum of the probability of reaching state s_{t+1} by taking step a times the value in s_{t+1}.

The model of an environment is something that mimics the behavior of the environment or allows inferences to be made about the environment, it can for example be used to predict the next state of the environment. These models can be used for planning, in which a plan is created, deciding the course of future actions before they are actually experienced. RL methods that use models and planning for

learning are called model-based, RL methods that do not, are called model-free and learn via trial and error.

There are multiple different types of RL, model-based and model-free have been touched on shortly in the previous paragraph, but it is important to highlight the difference between those two approaches and highlight some other RL types.

2.7.1. Model based and model free reinforcement learning

The definition of model-based and model-free RL requires some clarification. In short, model-based RL is the field of combining planning over a model, which is a functional representation of the environment, with learning. This means that in model-based RL, the agent is able to use a model of the environment to reason about what will happen in the environment. There is a lot of grey area between what is considered model-based RL and what is not. Overall, the consensus is that if an approach learns a model from observed data, or uses a model during learning in order to gain insight into the possible future/history of states, it is seen as model-based RL. Model-free RL is seen as a trial-and-error algorithm, where the agent learns a policy directly by interacting with the environment. A division between model-based techniques and model-free techniques when combining planning and RL is made in *model-based Reinforcement Learning; A Survey* [48], they define three categories:

- *Model-based RL with a learned model*, where both a model and policy are learned.
- *Model based RL with a known model*, where there is a known model and planning is used to learn a policy
- *Planning over a learned model*, where a model is learned and planned over, without learning a policy

The last point is not considered model-based RL, since it does not learn a global solution to the problem, but instead is seen as planning-learning integration and falls in the grey area between model-free RL and model-based RL.

2.7.2. On and off policy reinforcement learning

One of the distinctions one can make between different RL methods is on and off-policy methods. In off-policy methods, the algorithm used to evaluate and improve the policy is different from the policy used to select actions during learning. An example of an off-policy RL method is Q-learning, in Q-learning, the agent uses a greedy policy to select the actions during learning, and does not use its own policy. In on-policy methods, the agent does use the current policy in order to select actions during training, an example of an on-policy method is SARSA. In Equation 2.3 and Equation 2.4 the value functions of both Q-learning and SARSA are given, in Q-learning the policy is updated by taking the action that maximizes the post-state Q-function $Q(s_t, a)$. In SARSA the same policy that generated the previous action a_t is used to generate the next action a_{t+1}.

$$Q_{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \, max_{a \in A} \, Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.3}$$

$$Q_{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.4}$$

In both these value functions, α is the learning rate, which indicates how much the newly acquired information should overwrite the old information. γ is the discount factor, determining how important future rewards are compared with the reward of the current state and r_t is the reward received in that state.

2.7.3. Deep reinforcement learning

Deep RL is the field of combining RL with deep learning. It enables RL to solve problems that used to be intractable due to having an extremely large state space. Deep learning is a technique that uses Neural Networks (NN) in order to create approximate functions that can be used to represent a policy. Using plain RL does not scale very well to very large problems, plain RL uses a lookup table to store,

index, and update all possible states and their values. When the environment is very large and therefore has an enormous amount of states, it may be computationally impossible to calculate a value for every state, as well as store the entire table.

Deep RL uses NN to approximate optimal value functions and/or policies. Deep learning NN exists out of a number of nodes that are connected by edges. These nodes are divided into layers, there are three types of layers; an input layer, hidden layers, and an output layer. The input layer receives a state from the environment and a number of neurons will be activated based on the state, these activations will propagate through all hidden layers to the output layer, and the output layer outputs an action based on the neurons activated in the output layer. A deep learning NN must have at least 3 layers, one of each type but can have multiple hidden layers. Figure 2.9 shows a deep RL version of an RL agent, where the agent is represented by a neural network with three layers.

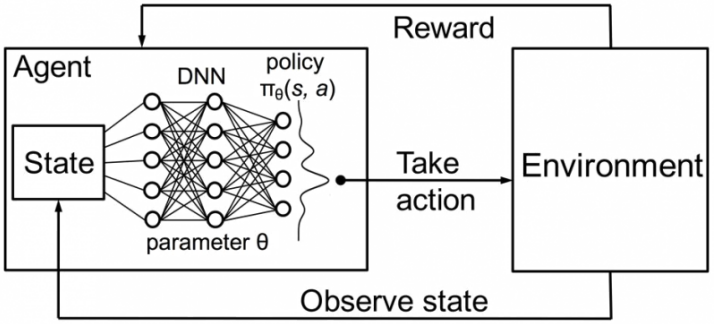


Figure 2.9: example of a deep RL agent [49]

2.8. Hoëffding's inequality

Hoëffding's inequality is a technique from learning theory and probability theory that is used to bound the probability that sums of bounded independent random variables deviate from their expected value ($\mathbb{E}$) more than by a certain amount [50]. When dealing with sums or averages over multiple samples or simulations, Hoëffding's inequality is used to provide a confidence bound that the deviation between the calculated value from the simulation and the expected value is within a certain error margin.

Formally Hoeffding's inequality is defined as:

Let $X_1, ..., X_n$ be independent bounded random variables with $X_i \in [a, b]$ for all i, where $-\infty < a \leq b \leq \infty$ and let $S_n = \sum_{i=1}^{n} X_i$ and an error margin $\epsilon > 0$ then:

$$\mathbb{P}(|S_n - \mathbb{E}[S_n]| \geq \epsilon) \leq \delta = 2e^{-\frac{2n\epsilon^2}{\sum_{i=1}^{n}(b-a)^2}} \tag{2.5}$$

Where n is the number of samples and δ is an upper bound on the probability that the difference between the sum and the expected value is bigger than ϵ [51].

Hoëffdings inequality can be rewritten such that it can be used to calculate the bound on the probability that the average of bounded independent random variables deviates from the expected average with the following equation [52]:

$$\mathbb{P}(|\frac{1}{n}\sum_{i=1}^{n} X_i - \mathbb{E}[X_i]| \geq \epsilon) \leq \delta = 2e^{-\frac{2n\epsilon^2}{(b-a)^2}} \tag{2.6}$$

Hoeffding's inequality can also be used to calculate the number of samples required to gain a probability of at least $1 - \delta$ that the difference between the empirical mean ($\frac{1}{n}\sum_{i=1}^{n} X_i$) and the expected mean ($\mathbb{E}[X_i]$) is at most ϵ, this can be calculated with Equation 2.7 [53].

$$n \geq \frac{(b-a)^2}{2\epsilon^2} \log(\frac{2}{\delta}) \tag{2.7}$$

In the context of RL, Hoëffding's inequality can be used to provide confidence on the average performance of a learned policy by simulating the policy a number of times and use the performance data gathered from each simulation to calculate a bound on the probability that the expected average performance is within a certain range of the average performance calculated from these simulations.

3

Approach

This chapter explains MOPRL, the approach created to achieve the main goal of solving LTL-tasks with multiple objectives in environments with sparse rewards. Each sub-research question will be addressed, explaining the techniques used in order to achieve the main goal. The sub-research questions build on each other, so RQ1 explains the general idea of how we can use the idea of policy sketches to create a controller that can solve LTL-tasks. RQ2 builds on top of that, explaining how we can not just create a controller using the general idea from RQ1, but also optimize this controller for multiple objectives (cost and success probability in this research). RQ3 explains how stochasticity in the environment can potentially violate completing LTL tasks using the ideas from RQ1 and RQ2, and how we add functionality to prevent this from happening. RQ4 explains how we provide guarantees for both the cost and the success probability of the final controller generated by MOPRL. At the end of this chapter, an overview of all the steps taken to generate the controller is presented.

3.1. RQ1: How can we leverage the idea of policy sketches to solve LTL-tasks in environments with sparse rewards?

To solve LTL-tasks in environments with sparse rewards, we take inspiration from the idea of policy sketches [20]. In policy sketches, reachability tasks are solved in environments with sparse rewards by defining subtasks that are easier to solve. These subtasks have overlapping start and goal states such that once a subtask is finished, another one can be executed directly after that. Executing a sequence of these subtasks can then solve reachability tasks in the environment. The idea of policy sketches is often combined with RL, where RL is used to learn low-level policies that can transition from the start state of a subtask to its goal state (i.e. complete the subtask). After RL is used to create low-level policies which can complete the subtasks, planning is used on a high level to find a sequence of subtasks to execute in order to complete the global reachability task. An example of this is the paper by Eysenbach, Salakhutdinov, and Levine (2019) [22], where they use RL to learn a policy for every subtask. They then use the subtasks to create a graph, in which each edge represents a subtask, and use Dijkstra's algorithm to find a sequence of subtasks that minimizes the length of the total path. In this research, we leverage the idea of policy sketches, manually defining subtasks and finding sequences of subtasks that can solve LTL-tasks. After defining subtasks, we use RL to train agents that can solve these subtasks on a low level. These subtasks are used to create a graph, which we call the high-level model (HLM), which can be used to find sequences of subtasks for reachability tasks. In contrast to Eysenbach, Salakhutdinov, and Levine (2019) [22], who only find a sequence of subtask for reachability tasks, we use this HLM to solve more complex LTL-tasks. We combine the HLM with the Büchi automata representing the LTL-task into a new graph, called the product graph. Combining the HLM with the Büchi automata converts the problem of finding a sequence of subtasks that can complete the LTL-task to a reachability problem in the product graph. Once the LTL-task is converted to a reachability problem, high-level planning can be used to find a sequence of subtasks that can complete the LTL-task.

The advantage of using the idea of policy sketches is that planning and learning are decoupled. This

makes it possible to plan for different tasks without needing to retrain the subtask controllers, as long as the task can be completed by executing a sequence of already defined subtasks. Decoupling of learning and planning also makes it possible to adjust the learning procedure for subtasks or change the planning procedure separately, making it very flexible. A disadvantage of using the idea of policy sketches is that, since all subtasks are solved only in a local optimal way, the global optimal policy may be lost when they are aggregated back together [54]. In general, hierarchies constrain the policy space and, hence, might prune optimal policies [16]. The flexibility of policy sketches makes it a good option for solving tasks in rapidly changing environments, such as environments where the task that needs to be completed can change since it is not necessary to retrain the subtask controllers if the new task can be completed with a sequence of already existing subtasks. In these situations, it is only required to find a new sequence of subtasks that can complete the new task.

In the next four subsections, we describe how we defined subtasks and use these subtasks to create the HLM and product graph. Subsection 3.1.1 describes how subtasks for reachability tasks are defined. Subsection 3.1.2 explains how these subtasks are used to create an HLM that can be used to find sequences of subtasks that solve reachability tasks. We then explain how we extend this concept in order to find sequences of subtasks that can solve more complex LTL-tasks. Subsection 3.1.3 describes how the subtask definition is extended such that we can combine the HLM with the Büchi automata representing the LTL-task. Subsection 3.1.4 explains how we combine the HLM with the Büchi automata into a product graph, which converts finding a sequence of subtasks that can complete the LTL-task to a reachability problem in the product graph, such that we can use a planning algorithm to find a sequence of subtasks that completes the LTL-task.

3.1.1. Subtasks for reachability tasks

To define the manually crafted subtasks we start with an adjusted version of the option framework [55]. In the option framework, an option is a closed-loop policy for taking actions over a period of time. This option can be seen as executing a subtask, where the closed-loop policy is the agent responsible for completing the subtask. An option exist out of three components: a policy π: $S \times A \rightarrow [0, 1]$, an initiation condition $\mathcal{I} \subseteq S$, and a termination condition $\mathcal{B} : S^+ \rightarrow [0, 1]$. An option is available in state s_t if and only if $s_t \in \mathcal{I}$. If the option is selected, π is used to generate an action, and the environment transitions to state s_{t+1}, where the option terminates with probability $\mathcal{B}(s_{t+1})$, or continues and generates the next action, a_{t+1} and the environment transitions to state s_{t+2} where the option terminates with probability $\mathcal{B}(s_{t+2})$. This will continue until the option terminates, and the agent has the opportunity to select a new option to execute. The subtasks in this research are designed similarly as options. Each subtask has a policy, an entry condition, and a termination condition. When the entry condition of a subtask is met, it can be selected and its policy is used to generate actions until either the subtask is completed or the agent has failed. Formally, in this research subtasks for reachability tasks are defined as $c = (\mathcal{I}_c, \pi_c, \mathcal{F}_c)$, where:

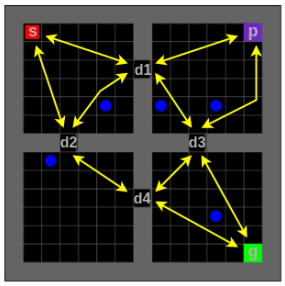
- $\mathcal{I}_c \subseteq S$: A subset of the entire state space of the environment (S) representing the entry conditions of that subtask
- $\pi_c : s \rightarrow a$ the policy used to complete the subtask, it selects an action $a \in A$ to execute in state $s \in S$
- $\mathcal{F}_c \subseteq S$: A subset of the entire state space of the environment (S) representing the goal conditions of that subtask

The main difference between this subtask definition and the option framework is that in our subtasks, F_c represents the states in which the subtask is completed, while in the option framework each state has a probability between 0 and 1 to terminate the current option. it is possible to translate F_c in our subtasks to $\mathcal{B}$ in the option framework by doing the following: every state $s \in F_c$ has $\mathcal{B}(s) = 1$ and every state $ns \notin F_c$ has $\mathcal{B}(ns) = 0$.

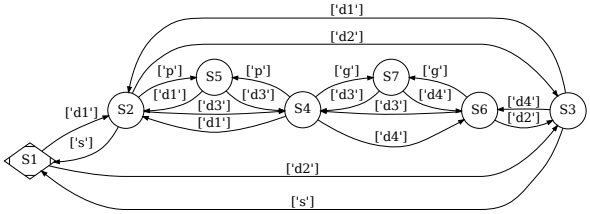
3.1.2. High-level model (HLM)

We use the subtasks to create a model, which we call the high-level model (HLM). This HLM expresses the relation between subtasks and can be used to find sequences of subtasks that complete reachability tasks. The HLM is modeled as a directed graph $G = (V, E)$. To create the HLM, first subtasks are

manually defined in the environment. These subtasks are designed such that they have overlapping entry ($\mathcal{I}$) and goal ($\mathcal{F}$) conditions, such that once a subtask has finished, another one can be started. For each unique entry/goal condition in all subtasks, one unique vertex is created in the HLM. After all vertices are created, edges are added to the HLM. To add an edge between two vertices in the HLM, we use the fact that each subtask navigates from an entry condition to a goal condition, so each subtask is a transition from one vertex in the HLM to another one. We add one directed edge to the HLM for each subtask indicating from which entry condition to which goal condition that subtask navigates the agent. Let us take Figure 3.1 as an example, which contains an example of an environment and its corresponding HLM.



(a) Example of an environment, the green and purple square are goal squares, which can be part of a certain task, the red square is the agent starting location



(b) The HLM, corresponding to the environment in Figure 3.1a, S1 is the starting vertex, corresponding to the starting location in the environment

Figure 3.1: An environment with manually defined subtasks, and the composed HLM

In Figure 3.1a each two-sided arrow represents two subtasks, one subtask for navigating between the two locations in each direction. At the end of each arrow, there is a label representing the goal condition of that subtask. For instance, the agent being at the purple square is the goal condition of a subtask and is labeled *p*.

Figure 3.1b is the HLM corresponding to the subtasks in Figure 3.1a. Each edge is accompanied by the label of that subtask's goal location, so an edge with a p label has as goal condition that the agent is at the purple square. This HLM can be used to find a sequence of subtasks that completes reachability tasks (reach a certain state in the environment). For example, in Figure 3.1b, $S1$ is the start state, and the task is to reach the green square, one can execute the subtasks that traverse the following states: $S1 \rightarrow S2 \rightarrow S5 \rightarrow S4 \rightarrow S7$ to complete the task since when the subtask of edge $S4 \rightarrow S7$ is completed the green square is reached.

To be able to find a sequence of subtasks that can solve LTL-tasks instead of only reachability tasks we convert the problem of solving the LTL-task to a reachability problem. This can be done by combining the Büchi automata representing the LTL-task and the HLM together in a new graph, which we call the product graph. To create this product graph, we first need to extend the subtask definition.

3.1.3. Extending subtask definition

To convert the problem of solving the LTL-task to a reachability problem, every subtask needs to indicate which events it triggers on completion. This is required so that it is possible to check whether an LTL-task will progress, is completed, or is violated when executing a certain sequence of subtasks. Let us take the LTL-task $\Diamond p \vee \Diamond g$ (eventually reach purple or eventually reach green) as an example, the Büchi automata representing this task can be found in Figure 3.2.

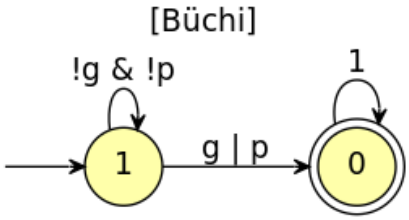


Figure 3.2: Büchi automata representing the task $\Diamond p \vee \Diamond g$

To check if a certain sequence of subtasks would complete the task we need to know whether executing a sequence of subtasks would trigger either p or g, such that we know that the sequence of subtasks would traverse the büchi automata to the accepting state, and therefore is a sequence of subtasks that completes the LTL-task. For this notion, we extend the subtask definition $c = (\mathcal{I}_c, \pi_c, \mathcal{F}_c)$ with a label set L_c. L_c is a label set that indicates which events are triggered when that subtask is completed. So if a subtask ends in the green square, then $L_c = [g]$. L_c can then be used to check what the influence of a subtask is on the Büchi automata of an LTL-task, and whether a certain sequence of subtasks would traverse the Büchi automata to the accepting state.

3.1.4. Converting LTL-task to a reachability problem

Now that we know what events are triggered by subtasks when they are completed, we can combine the HLM and a Büchi automata representing an LTL-task together in a product graph. This reduces the problem of solving the LTL-task to a reachability problem. First, we will explain how we create such a product graph and afterward give an example of how a path in this product graph is a valid sequence of subtasks that complete the LTL-task.

Creating the product graph exists out of three steps:

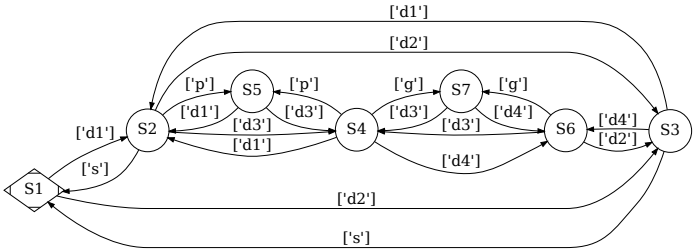
- Create the entire product graph
- Prune unreachable vertices
- Prune states only reachable from a goal vertex

To create the entire product graph $PG = (V_{pg}, E_{pg})$ we take a HLM $H = (V_h, E_h)$ and the Büchi automata $B = (Q, \sum, \delta, q_0, F)$. First, we create a new vertex v in V_{pg} for every vertex $v_h \in V_h$ and every state $b \in Q$ combination, resulting in $|V_h| \cdot |Q|$ vertices in V_{pg}. We call these states $v_i b_i$, so if a vertex v in V_{pg} is created from $v_1 \in V_h$ and $s_1 \in Q$, the new vertex $v \in V_{pg}$ is called $v_1 s_1$.

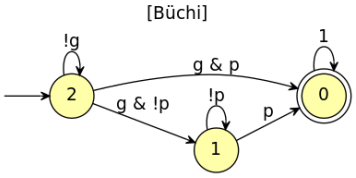
After all the states are created in V_{pg}, the next step is to add edges to E_{pg}, connecting states in V_{pg} with each other. To do this, we check whether executing a certain subtask would traverse the Büchi automata to a new state or not, and if so, a directed edge is added to the product graph, indicating that executing that subtask would traverse the Büchi automata to the next state. So if we are in vertex $v_1 s_1$ in the product graph (v_1 in the HLM, and s_1 in the Büchi automata) and executing the subtask that would go from v_1 to v_2 in the HLM and progress the Büchi automata from s_1 to s_2, a directed edge is added from $v_1 s_1$ to $v_2 s_2$ in the product graph. To see if executing a subtask will traverse the Büchi automata to a new state or not, we use that every edge in E_h is a subtask. As explained in subsection 3.1.3, each subtask c has a set of labels (L_c) associated with them, corresponding to the events that happen when that subtask is completed. Each transition in δ from the Büchi automata contains an atomic proposition that results in either $True$ or $False$, depending on the input. When the atomic proposition evaluates to $True$ given a certain input, the Büchi automata will transition to the next state. We use the set of labels L_c as the input of a transition in δ to see if executing this subtask will transition the Büchi automata to a new state and an edge should be added to the product graph.

Let us clarify this with an example, we take the HLM H from Figure 3.1 and a Büchi automata B corresponding to the LTL-task $\Diamond(g \wedge \Diamond p)$ (eventually reach green and afterward eventually reach purple) in Figure 3.3. Each edge in H is accompanied by L_c, the events that are triggered when the subtask

related to that edge is completed. Each transition in B is accompanied by a logic formula indicating which events need to happen to transition B from one state to another.



(a) The HLM from the environment in Figure 3.1a



(b) A Büchi automata corresponding to the task $\Diamond(g \wedge \Diamond p)$

Figure 3.3: an HLM and a Büchi automata which are combined into a product graph

First, we create new states for every state $v_h \in V_h$ and $b \in Q_B$ combination, resulting in 21 states ($s1b2, s1b1, s1b0, ..., s7b2, s7b1, s7b0$), where state $s1b1$ is created from $S1$ in V_h and state 1 in Q_B. After the states are created, the edges of the product graph are added. To see if a new edge should be added to the product graph, we evaluate the logic formula of each transition in δ_B using the labels from the subtasks in the HLM (L_c). For example, we are in state $s1b2$ in the product graph ($S1$ in H and $b2$ in B). We evaluate each outgoing edge from $b2$ using the outgoing edges from $S1$. $S1$ has two outgoing edges in H, $S1 \rightarrow S2$ and $S1 \rightarrow S3$. $b2$ has three transitions in B, $b2 \rightarrow b2$, $b2 \rightarrow b1$ and $b2 \rightarrow b0$.

The edge $S1 \rightarrow S2$ has $L_c = [d1]$. We evaluate the logic formula of the transition $b2 \rightarrow b2$ ($!g$) using L_c. for $!g$ to be $True$, g must not happen, therefore we check L_c, the set of events that happen when the subtasks related to the edge $S1 \rightarrow S2$ is completed, to see if g happens when $S1 \rightarrow S2$ is traversed in the HLM. L_c does not contain g, so $!g$ evaluates to $True$. Formally we have:

$!g \rightarrow True$, given $L_c = [d1]$

Resulting in an edge being added from $s1b2$ to $s2b2$ in the product graph. The edge $b2 \rightarrow b1$ in B has the logic formula g & $!p$ and evaluates to $False$, since g is not in L_c, formally:

g & $!p \rightarrow False$, given $L_c = [d1]$

So the edge $s1b1 \rightarrow s2b1$ is not added to the product graph. The edge $b2 \rightarrow b0$ in B has the logic formula g & p and evaluates to $False$, since both g and p are not in L_c, formally:

g & $p \rightarrow False$, given $L_c = [d1]$

So the edge $s1b1 \rightarrow s2b0$ is not added to the product graph. For the edge $S1 \rightarrow S3$ in H we have $L_c = [d2]$, resulting into:

$!g \rightarrow True$, given $L_c = [d2]$

g & $!p \rightarrow False$, given $L_c = [d2]$

g & $p \rightarrow False$, given $L_c = [d2]$

So the edge $S1b2 \rightarrow S3b2$ will be added to the product graph since the transition $b2 \rightarrow b2$ ($!g$) evaluates to $True$ and both $S1b2 \rightarrow S3b1$ and $S1b2 \rightarrow S3b0$ will not be added to the product graph since the transitions $b2 \rightarrow b1$ (g & $!p$) and $b2 \rightarrow b0$ (g & p) evaluate to $False$. This process is repeated for every possible edge combination in the product graph and results in the product graph from Figure 3.4. Each vertex in Figure 3.4 has a name in the form of $S_i b_j$, which is created from vertex S_i from H and state j

from B. The start vertex in H is $S1$ and the start state in B is $b1$, so the start vertex in the product graph is $S1b1$. For the goal vertices in the product graph, every vertex in the product graph that is related to an accepting state in B is a goal vertex in the product graph, so every state $S_i b_0$ is a goal state since b_0 is the accepting state in B.

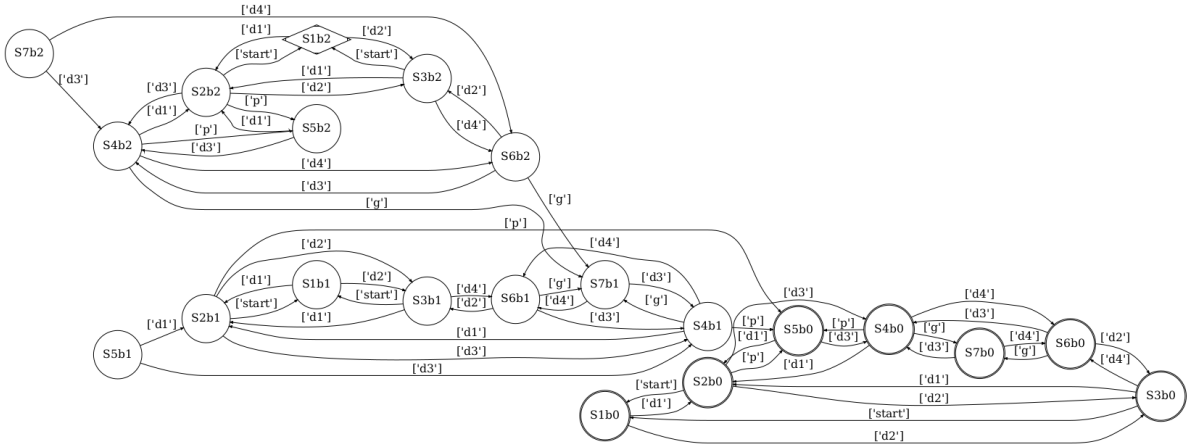


Figure 3.4: Product graph from the HLM and Büchi automata in Figure 3.3

The next step is to prune unnecessary states and edges in the product graph. This will reduce the product graph size, which will speed up finding sequences of subtasks that can solve the LTL-task. First, all states that are unreachable are pruned, in Figure 3.4 state $S5b1$ and $s7b2$ are unreachable, since they are not the starting state and have no incoming edges, so they can be removed.

Second, vertices that are only reachable after a goal vertex is visited can be pruned. This research only evaluates finite LTL-tasks (LTL-tasks that only need to be completed once and in a finite amount of time), so vertices in the product graph that can only be visited after a goal vertex is visited can be pruned since the LTL-task has already been completed once if a goal vertex is visited in the product graph. So in Figure 3.4 the states $S1b0, S2b0, S4b0, S7b0, S6b0$, and $S3b0$ are removed since they can only be visited after $S5b0$ is visited, which is an accepting vertex. After pruning all unnecessary states and edges we gain the final product graph from Figure 3.5

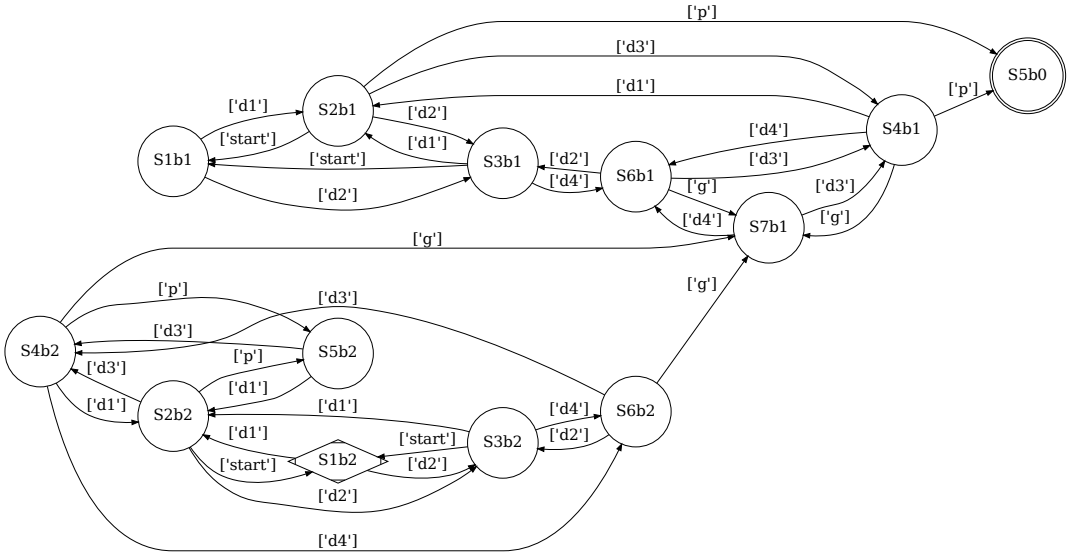


Figure 3.5: Product graph from the HLM and Büchi automata in Figure 3.3 after pruning states and edges

Any path in this product graph from the start vertex to a goal vertex is a sequence of subtasks that complete the LTL-task. Recall that the LTL-task we want to complete is $\Diamond(g \wedge \Diamond p)$ (eventually reach green and afterward eventually reach purple). One path in the product graph in Figure 3.5 that completes the LTL task is $S1b2 \rightarrow S3b2 \rightarrow S6b2 \rightarrow S7b1 \rightarrow S4b1 \rightarrow S5b0$, which are the edges, or subtasks $d2 \rightarrow d4 \rightarrow g \rightarrow d3 \rightarrow p$. If we return to the environment from Figure 3.1a and follow this path of subtasks, highlighted with red arrows in Figure 3.6, we indeed see that this is a valid sequence of subtasks that completes the LTL-task $\Diamond(g \wedge \Diamond p)$.

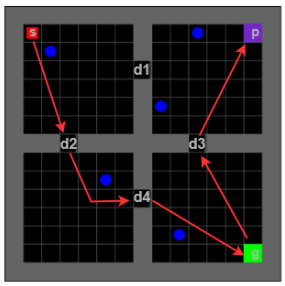


Figure 3.6: Example of an environment, with a sequence of subtasks that first goes to green and afterward goes to purple

3.2. RQ2: How can we solve LTL-tasks efficiently, optimizing for both success probability as well as cost?

With the idea from RQ1, manually defining subtasks and finding sequences of subtasks that complete the LTL-task. The next step is to solve the LTL-task efficiently, optimizing for both success probability as well as cost, we identify two points that need to be addressed:

- How do we learn subtasks, such that we maximize the success probability and minimize the cost?
- How can we use planning to find a sequence of subtasks that optimizes for both cost and probability of success?

3.2.1. How do we learn a policy for subtasks?

In this research, subtasks are defined as: *go from one location in the environment to another while avoiding the moving obstacles*. To learn a policy for subtasks, an RL algorithm is used. To learn how to avoid obstacles as well as efficiently move to the subtask's goal location, accurate rewards need to be given to the agent during training. When the agent is punished too heavily when crashing into an obstacle, it is possible that the agent only tries to avoid obstacles and does not learn how to complete the subtask. When the agent is not punished for taking an extra step, it may learn a policy that is very safe but not very cost-efficient. When the agent is not punished enough for crashing with obstacles, it may learn a policy that is very efficient when it succeeds but often collides with an obstacle.

We initially provided the agent with the following reward function:

$$R(S) = C + E + G \tag{3.1}$$

The three components in Equation 3.1 are defined as:

- C: A negative reward if the agent collides with an obstacle
- E: A small negative reward for each step the agent takes
- G: A positive reward when the agent reaches the goal location

With this reward function the agent is trained to reach the goal state while avoiding the obstacles and the agent also tries to minimize the number of steps it takes to reach the goal state.

After the initial reward function, we researched if the training of an agent could be improved by limiting an agent's operation area. If a subtask is limited to one room, for example, to move from one door in a room to another door in that room, it does not need to explore other rooms in the environment. Therefore it may be beneficial to exclude information about other rooms outside the operation area since those rooms never need to be visited and information from those rooms can make the agent's observations noisy, resulting in subpar performance. We ran some tests to see if it would be beneficial to limit an agent's operation area and what observation we should provide to the agent.

We tested four different types of observations (two global observations and two local observations) in a big environment (Figure 3.7 in which the subtask was to go from the starting location (red square) to door 1(green square).

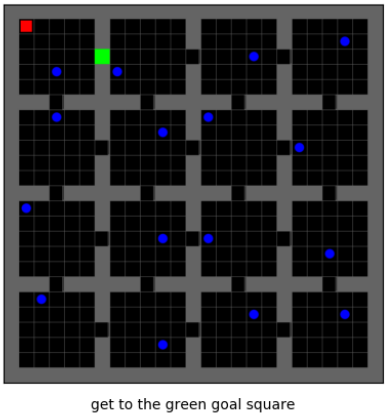


Figure 3.7: Environment used for testing different types of operation areas and observations, the agent (red) needs to navigate to the green square while avoiding the stochastic moving obstacle (blue)

The four different types of observations tested are:

1. Using the entire grid of the environment as an observation (global)
2. Using the operation area, containing only tiles that are in the operation area (local)
3. Extracting the location of the agent and all obstacles from the environment (global)
4. Extracting the location of the agent and obstacles in the operation area (local)

For the results of the tests, see Table 3.1.

Experiment	Observation size	Average steps	Succes probability
entire environment (global)	625	7.48	0.76
operation area (local)	49	7.00	0.90
agent and obstacle locations (global)	34	6.48	0.78
agents and obstacle locations, (local)	4	6.75	0.83

Table 3.1: Results from testing different types of observations for completing a subtask

These tests show that it is beneficial to limit an agent's operation area and provide observations containing the tiles in that operation area since this resulted in the highest success probability of 90%. To limit the agent's operation area when learning a subtask we extend the subtask definition $c = (\mathcal{I}_c, \pi_c, \mathcal{F}_c, L_c)$ by adding O_c, the agent's operation area, indicating which parts of the environment are relevant in order to learn the subtask. For example in Figure 3.8, we limit the operation area (O_c) of the subtask to the 8x8 squares, highlighted with light blue.

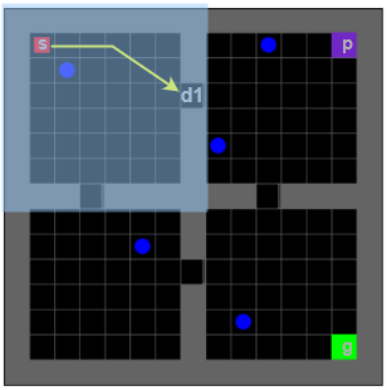


Figure 3.8: The operation area of the subtask "go from the start location to door 1"

We also add a new variable to the reward function since the agent never needs to leave the operation area of the subtask in order to learn it. We provide the agent with a negative reward L when it leaves the operation area, resulting in the final reward function:

$$R(S) = C + E + G + \boldsymbol{L} \tag{3.2}$$

We assign C, the reward when colliding with an obstacle the value of -1. E, the reward of taking a step is -0.03. G, the reward when an agent successfully completed the subtask is the value of 1. L, the reward when the agent leaves its operation area is -1. These values were chosen after testing different configurations. When the (negative) reward given to the agent for hitting an obstacle or leaving its operation area was too large in comparison with the reward given to the agent when the subtask is completed, the agent would get stuck in a local optimum, prioritizing avoiding the obstacles and points near the operation area border over completing the subtask.

3.2.2. How can we use planning to find an optimal sequence of subtasks?

Now that it is possible to learn policies for subtasks and find sequences of subtasks that can complete an LTL-task using the product graph, the next step is to find optimal sequences of subtasks. In this research, we want to optimize for both the cost as well as the success probability, meaning it is a multi-objective setting. In order to be able to find an optimal sequence of subtasks in the product graph we use Pareto optimality, see section 2.5, which is a technique used to find efficient solutions in a multi-objective setting. To be able to find Pareto-optimal sequences of subtasks, we need to know the success probability and cost of completing each subtask in the product graph. To gain this success probability and cost of completing a subtask c, we train an agent for that subtask and use its policy π_c to run 600 simulations. From these simulations, the success probability and average cost of completing subtask c are estimated. The estimated success probability $\mathbb{P}_c$ is calculated with Equation 3.3.

$$\mathbb{P}_c = \frac{amount\ of\ successful\ runs}{600} \tag{3.3}$$

For the cost, we use all successful runs from the 600 simulations used for calculating the success probability. Unsuccessful runs are excluded from this calculation since we want to know the average cost of completing the subtask and not the average cost of running the subtask. Equation 3.4 is used to calculate the cost (C_c), where $n = \#\ of\ successful\ runs$.

$$C_c = \frac{1}{n}\sum_{1}^{n} steps \tag{3.4}$$

After knowing the average success probability and cost of completing each subtask, we can calculate the estimated success probability and cost of a sequence of subtasks (S) with:

$$\mathbb{P}(S) = \prod_{\forall s \in S} \mathbb{P}(s) \tag{3.5}$$

$$C(S) = \sum_{\forall s \in S} C(s) \tag{3.6}$$

Now that we can calculate $\mathbb{P}(S)$ and $C(S)$ we can find all Pareto optimal paths in a product graph. To find all Pareto optimal paths, we define the dominate function as follows; given two paths in the product graph (sequences of subtasks), $S1$ and $S2$ we say $S1$ dominates $S2$ ($S1 \succ S2$) if:

$$\mathbb{P}(S1) > \mathbb{P}(S2)\ and\ C(S1) < C(S2) \tag{3.7}$$

where:

- $\mathbb{P}(x)$: success probability of x
- $C(x)$: cost of x

With the dominate function we can use Martins algorithm (section 2.6) to find all Pareto optimal sequences of subtasks in the product graph from the start state to a goal state. It is possible that a product graph has multiple goal states, for example in cases when the LTL-task is a choice task such as $\Diamond g \vee \Diamond p$ (go to green or go to purple). See Figure 3.9 for the product graph created from the LTL-task $\Diamond g\ or\ \Diamond p$ and the HLM in Figure 3.1b.

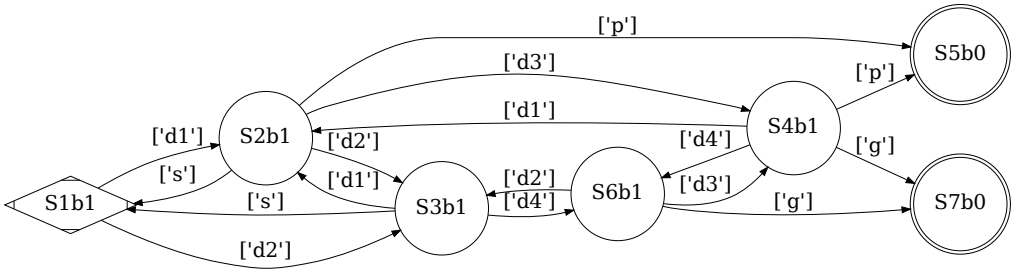


Figure 3.9: Product graph from the HLM in Figure 3.1b and the LTL-task $\Diamond g \vee \Diamond p$

If there are multiple goal vertices in a product graph, Martins algorithm is used to find Pareto-optimal paths for every possible start vertex - goal vertex combination in the product graph. When there are multiple goal vertices in a product graph it is possible that a Pareto-optimal path to one goal vertex is dominated by a Pareto-optimal path to another goal vertex. If we look at Figure 3.10, which is the same product graph as in Figure 3.9, but with the success probability and average cost of completing each subtask displayed above the edges. There are two Pareto-optimal paths from the start vertex ($S1b1$) to the goal vertex $S5b0$, one such path is $S1b1 \rightarrow S2b1 \rightarrow S5b0$, resulting in a success probability of 0.73 and a cost of 18, the other is $S1b1 \rightarrow S2b1 \rightarrow S4b1 \rightarrow S5b0$, resulting into a success probability of 0.74 and a cost of 27. From the start vertex to the goal state $S7b0$ there is one Pareto-optimal path ($S1b1 \rightarrow S3b1 \rightarrow S6b1 \rightarrow S7b0$) with a success probability of 0.86 and a cost of 26. This means that the path $S1b1 \rightarrow S2b1 \rightarrow S4b1 \rightarrow S5b0$ is dominated by a path leading to another goal state since the path $S1b1 \rightarrow S3b1 \rightarrow S6b1 \rightarrow S7b0$ has a higher success probability (0.86 over 0.74) and a lower cost (26 over 27), therefore $S1b1 \rightarrow S2b1 \rightarrow S4b1 \rightarrow S5b0$ is no longer deemed to be Pareto-optimal.

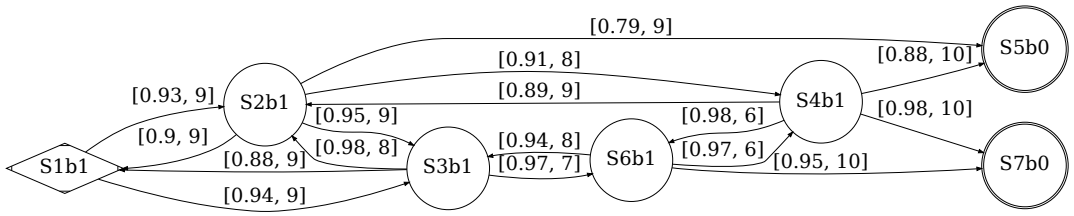


Figure 3.10: Product graph from the HLM in Figure 3.1b and the LTL-task $\Diamond g \vee \Diamond p$, displaying the success probability and cost of every subtask

So in order to prevent sub-optimal paths from being found all permanent labels in all goal states are filtered. All permanent goal state labels that are dominated by a permanent label in another goal state are removed, such that only Pareto optimal paths remain.

3.3. RQ3: How can we deal with stochasticity in environments, while guaranteeing the LTL-task is not violated?

In order to prevent stochasticity in the environment from violating the specification, let us first clarify in which cases stochasticity can violate the LTL-task. If we take Figure 3.11 as an example, the task that needs to be completed is $\Diamond\ green\ \wedge \Box\ \neg\ purple$, or in words, eventually reach green while always avoiding purple. The defined subtasks are presented with the yellow arrows and the agent is currently at *d2* (red square). The fastest way towards $\Diamond green$ is to use the subtask to go from *d2* to *d1*, and then from *d1* to *green* in order to complete it. However, the agent is only trained to avoid the obstacles and

not to avoid *purple*, so if the subtask to reach *d1* is executed, there is a high chance that the agent will try to avoid the obstacles by going up and left, in which case the agent goes over *purple* and violates the global task.

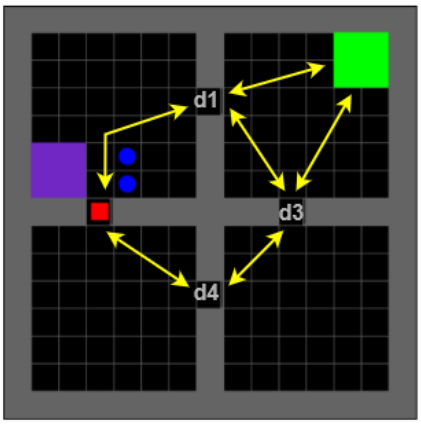


Figure 3.11: An example of a situation in which a global LTL-task can be violated, in this example the task is $\Diamond green \wedge \Box \neg purple$

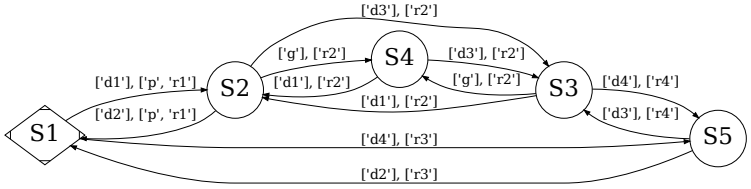
In order to prevent agents from violating the global LTL-task in case of reach-avoid tasks, there are two options:

- Train the subtasks with the global task in mind such that agents are trained to avoid states that violate the global task
- Ensure that all subtasks that potentially violate the global task are excluded during the creation of the product graph so that they will never be selected when trying to find a sequence of subtasks that complete the global task

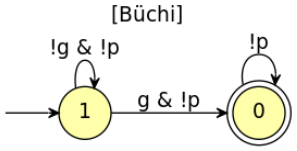
For this research, the choice was made to exclude all subtasks that may potentially violate the LTL-task during the creation of the product graph. When an LTL-task includes an avoid predicate ($\Box \neg x$), all subtasks that potentially trigger x cannot be used, since they can potentially violate the LTL-task. There were two main reasons we chose to avoid using subtasks that may potentially violate the LTL-task over training the subtasks with the global task in mind. First, when training each subtask with the LTL-task in mind, reward functions can become overly complex and new reward functions need to be created for every different LTL-task. Adjusting the subtask description does not have this issue. The second downside of training the subtask with the LTL-task in mind is that it requires all subtask controllers to be retrained when there is a new LTL-task, so it becomes less flexible. When each subtask indicates what events it may potentially trigger, it is possible to synthesize a controller for a new LTL-task, without the need of retraining the subtask controllers, and instead, it is only required to generate a new product graph and plan a new sequence of subtasks. A downside of excluding subtasks during the creation of the product graph is that possible solutions may be pruned from the solution space since you eliminate entire subtasks from being utilized.

To ensure that subtasks that potentially violate the global task are excluded during the creation of the product graph, we extend the subtask definition $c = (\mathcal{I}_c, \pi_c, \mathcal{F}_c, O_c, L_c)$ with the field LM_c. LM_c is a set that contains all events that can potentially trigger during the execution of a subtask.

During the creation of the product graph, the set LM_c is used to exclude subtasks that can potentially violate the LTL-task, so that they can never be selected during planning. Let us take the HLM of Figure 3.11 and the Büchi automata from the LTL-task $\Diamond g \wedge \Box \neg p$ (eventually reach green while never visiting purple) as an example.



(a) The HLM from the environment in Figure 3.11



(b) A Büchi automata corresponding to the task $\Diamond g \wedge \Box \neg p$)

Figure 3.12: an HLM and a Büchi automata of the task $\Diamond g \wedge \Box \neg p$ which are combined into a product graph

In the HLM in Figure 3.12a each edge is accompanied by two sets, the first set is L_c, the set with events that happen when the subtask is completed. The second set is LM_c, the set containing events that the subtask potentially triggers.

Since the LTL-task is a reach-avoid task, in which certain states need to be avoided, during the creation of the product graph LM_c is used to check whether a subtask could potentially violate the avoid part of the task and if so, it is not added to the product graph so that it can never be selected during planning. To create the product graph, first, all the states ($s1b1, s1b0, ..., s5b1, s5b0$) are created. After creating all states, L_c and LM_c are used to evaluate the logic formula of each edge in the Büchi automata to see if an edge can be added to the product graph. For example, we are in state $s1b1$ in the product graph ($S1$ in HLM and $b1$ in $Büchi$) and evaluate each outgoing transition from $b1$ using the outgoing edges from $S1$. $S1$ has two outgoing edges in HLM, $S1 \rightarrow S2$ and $S1 \rightarrow S3$. $b1$ has two outgoing transitions in $Büchi$, $b1 \rightarrow b1$ and $b1 \rightarrow b0$. The edge $S1 \rightarrow S2$ has $L_c = [d1]$ and $LM_c = [p, r1]$. We evaluate the logic formula of the transition $b1 \rightarrow b1$ ($!g$ & $!p$) using L_c and LM_c. for $!g$ to be $True$, g must never happen, therefore we check both L_c and LM_c, to see if g will happen or potentially can happen when $S1 \rightarrow S2$ is traversed in the HLM. both L_c and LM_c do not contain g, so $!g$ evaluates to $True$. $!p$, is a negation, so p must never happen in order for this to be $True$. We once again check both L_c and LM_c. Since LM_c contains p, indicating p can potentially happen when executing this subtask, $!p$ evaluates to $False$. So formally we have:

$!g$ & $!p \rightarrow False$, given $L_c = [d1]$ and $LM_c = [p, r1]$

Resulting in no edge being added from $s1b1$ to $s2b1$ in the product graph. If we repeat this process for all edge combinations and prune unnecessary states and edges, we gain the product graph in Figure 3.13.

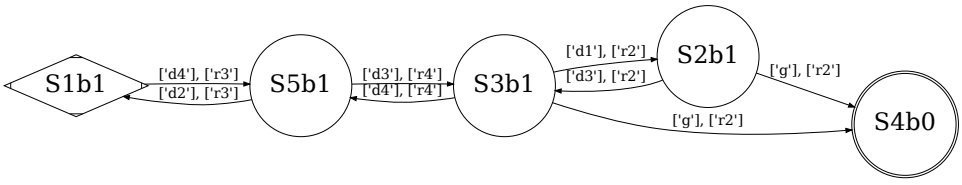


Figure 3.13: An example of a product graph in which the LTL-task was a reach-avoid task and some subtasks can not be used

Subtasks that have the p event in their LM_c set are excluded from the product graph since they can potentially violate the avoidance part of the LTL-task ($!p$). Since they are not in the product graph, they cannot be selected by the planning algorithm when searching for a sequence of subtasks that can complete the LTL-task. This guarantees that the avoid part of the LTL-task is never violated by any sequence of subtasks found in the product graph.

3.4. RQ4: How can we give guarantees for both the success probability, as well as the cost of the controller, which metrics should be used?

In order to provide guarantees for both the success probability and the cost of the controller, the choice was made to calculate probably approximately correct (PAC) guarantees using Hoëffding's inequality (section 2.8). PAC-guarantees provide an upper bound to the probability (the probably part) that the difference between the empirical mean of a number of variables is within a certain error (the approximate part) of the expected mean. PAC-guarantees avoid the state explosion problem [56], which is a phenomenon in which the state space grows exponentially in size with the number of environment variables. Since in our environment the state space grows exponentially with the number of obstacles, PAC-guarantees are very suited. PAC-guarantees are often used for mathematical analysis of AI systems and are also used in statistical model checking to provide a bound on the probability that the difference between the calculated mean and the expected mean is within a certain range.

To calculate the PAC-guarantees for both the cost and success probability, the following variables are required: the error margin (ϵ), a lower (a) and upper (b) bound on the values the random independent variables can take and the number of simulations used (n). When n, a, b, and ϵ are known, Equation 2.6 is used to calculate the probability that the difference between the calculated values and the expected values for the success probability and cost are within the error margin. Since the success probability and cost have different values assigned to n, a, b, and ϵ, they are discussed in separate subsections.

3.4.1. PAC-guarantees for success probability

For the success probability, the lower bound (a) and upper bound (b) are known, the lower bound is 0 (failed to complete the task), and the upper bound is 1 (the task is successfully completed). To gain the number of simulations required to get a certain confidence ($1-\delta$) and error margin (ϵ) we use Equation 2.7. Based on the calculation we chose to run 600 simulations since this results in a 90% confidence ($1-\delta = 0.9$) that the calculated mean deviates less than 0.05 (ϵ) from the expected mean, see Equation 3.8.

$$n \geq \frac{(b-a)^2}{2\epsilon^2}\log(\frac{2}{\delta}) = \frac{(1-0)^2}{2\cdot 0.05^2}\log(\frac{2}{0.1}) = 599.15 \tag{3.8}$$

This means that once a controller is created that can complete the LTL-task, we simulate it 600 times. Let us say 480 of those simulations are successful, so a success probability of $\frac{480}{600}\cdot 100 = 80\%$, then the probability ($\delta$) that the difference between the calculated success probability and the expected success probability is bigger than ϵ (0.05) is 0.099, see Equation 3.9.

$$\mathbb{P}(|\frac{1}{n}\sum_{i=1}^{n} X_i - \mathbb{E}[X_i]| \geq \epsilon) \leq 2e^{-\frac{2n\epsilon^2}{(b-a)^2}} = 2e^{-\frac{2\cdot 600\cdot 0.05^2}{(1-0)^2}} = 0.099 \tag{3.9}$$

With this we can provide a 90% confidence ($1-0.099 \approx 0.9$) that the expected success probability will be within 0.05 of the estimated success probability (80%), so within the range of 75 till 85%.

3.4.2. PAC-guarantees for cost

To calculate the PAC-guarantees for the cost we use all successful runs out of the 600 simulations used to calculate the PAC-guarantees for the success probability, since then we can calculate the PAC-guarantees for both the cost and success probability with a fixed number of simulations. For the cost, we first need to define a lower bound (a), an upper bound (b), and an error margin (ϵ) in order to calculate the PAC-guarantees. In the case of calculating the success probability, the lower and upper bound are fixed for every task, 0 if the controller failed to complete the task and 1 if the controller successfully completed the task. However, the lower and upper bound of the cost are dependent on the type of task that needs to be solved and the type of environment. Some tasks in an environment will have a higher cost than others since they require more actions to be completed. For example, the

task $\Diamond g \wedge \Diamond p$ in Figure 3.14 has a higher cost than the task $\Diamond p$ in all cases, since it is impossible to go to green and purple with a lower cost (fewer steps) than solely going to purple.

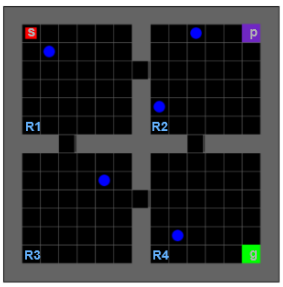


Figure 3.14: Example of an environment in which the task $\Diamond g \wedge \Diamond p$ has a higher cost than the task $\Diamond p$

To find the lower bound on the values the cost can be, we use the fact that if we remove the stochastic moving obstacles in the environment, the task becomes a shortest path problem, which can be solved by the Dijkstra algorithm. In Figure 3.15 an environment with no obstacles is presented, and a minimum cost path for the task $\Diamond g \wedge \Diamond p$ is given, which takes a total of 34 steps. It is impossible to have a cost lower than the minimum cost path found in the environment without obstacles, no matter if there are obstacles present or not, therefore this minimum cost can be used as a lower bound (a) for calculating the PAC-guarantees.

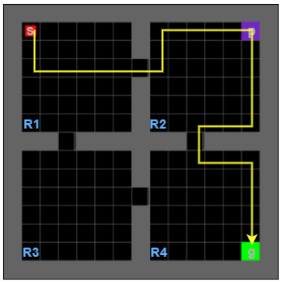


Figure 3.15: Example of an environment without obstacles in which the task is $\Diamond g \wedge \Diamond p$ and a minimal cost path that completes this task

To determine an upper bound, two things need to be considered. First, we do not want to set the upper bound too high because a high upper bound will weaken the guarantees we can provide. A higher upper bound results in a lower probability $(1 - \delta)$ that the difference between the calculated average cost and the expected average cost is within the error margin (ϵ) and therefore provides a lower confidence. Secondly, we do not want to set the upper bound too low. Since the cost may not exceed the upper bound for the PAC-guarantees, when a simulation takes more than $upper\ bound$ steps, the simulation is terminated and it is assumed the controller has failed to complete the task. Therefore, if we set the upper bound too low it is possible that many simulations used to calculate the PAC-guarantees will fail due to reaching the upper bound amount of steps, which will weaken the PAC-guarantees. To find an upper bound that gives us the best possible guarantees for an LTL-task we execute trial-and-error experiments. In the first experiment, we start with an upper bound of $2 \cdot lower\ bound$ and run 600 experiments to see how many of those experiments fail due to reaching the cost upper bound. If too many experiments failed due to reaching the upper bound cost, we increase the upper bound to $3 \cdot lower\ bound$ and run 600 simulations again. We repeat this process n times until the difference in the number of failed experiments due to reaching the upper bound cost between experiment $n-1$ and experiment n is negligibly small and then use $n - 1 \cdot lower\ bound$ as the upper bound.

We set the error bound (ϵ) to $0.05 \cdot upper\ bound$, since 5% of the upper bound is also the error margin used to calculate the success probability PAC-guarantees ($1 \cdot 0.05 = 0.05$).

3.5. Overview of approach

The techniques used in RQ1 till RQ4 are combined in one process which exists out of 8 steps, see algorithm 2. This process takes as input the environment, the manually crafted subtasks, and an LTL-task and outputs a controller that can complete the LTL-task in the environment.

In this section, we reiterate how subtasks are defined, with all extensions included. We also explain a little more in-depth how we create a controller from a sequence of subtasks (step 7 in algorithm 2) since this has not been discussed yet in section 3.1 till section 3.4. As last we summarize MORPL in its entirety.

Algorithm 2: steps of MOPRL

input : Environment E
input : A set of subtasks S
input : LTL-task L
output: A controller which selects subtasks in order to complete the LTL-task L
1. $Train\ agents\ for\ each\ subtask\ in\ S$;
2. $Collect\ success\ probability\ and\ cost\ of\ trained\ agents$;
3. $HLM \leftarrow Construct\ High\ Level\ Model\ from\ S$;
4. $Büchi \leftarrow Construct\ Büchi\ automata\ from\ L$;
5. $Graph \leftarrow Construct\ Product\ Graph\ from\ HLM\ and\ Büchi$;
6. $Paths \leftarrow Find\ Pareto\ optimal\ paths\ in\ Graph$;
7. $Controller \leftarrow generate\ controller\ from\ a\ path,\ where\ path \in Paths$;
8. $Simulate\ Controller\ in\ environment\ to\ provide\ PAC\ guarantees$;

3.5.1. Subtask definition

To reiterate, with all extensions included, a subtask is defined as $c = (\mathcal{I}_c, \mathcal{F}_c, \pi_c, O_c, L_c, LM_c)$ where:

- $\mathcal{I}_c \subseteq S$: A subset of the entire state pace of the environment (S) representing the entry conditions of the subtask
- $\mathcal{F}_c \subseteq S$: A subset of the entire state space of the environment (S) representing the goal conditions of the subtask
- $\pi_c : s \rightarrow a$ the policy used to complete the subtask, it selects an action $a \in A$ to execute in state $s \in S$
- O_c: the operation area; the area of the environment the agent can use to learn this subtask
- L_c: set containing labels of events triggered when c is completed
- LM_c: set containing labels of events that can possibly happen when c is executed

The goal of a subtask is to navigate from a state in the entry conditions $\mathcal{I}_c$ to a state in the goal conditions $\mathcal{F}_c$ while avoiding the obstacles in the environment.

3.5.2. Controller generation from a sequence of subtasks

After we use Martins algorithm to find all Pareto optimal paths (sequence of subtasks) in the product graph, we select one of these paths and use it to generate a controller. This path can be chosen based on whether we want to have the highest possible success probability, the lowest cost, or a balance between the two. Once a path has been chosen, a controller is generated. The controller works the following way: it takes an observation from the environment as input and outputs an action based on this observation and the currently loaded subtask. During the initialization of the controller, the first

subtask in the path is loaded, this subtask's policy is used for generating actions. Once the current subtask is completed, the next subtask in the sequence is loaded and this is repeated until the last subtask is completed, resulting in the completion of the LTL-task, Figure 3.16 shows an overview of how the controller functions.

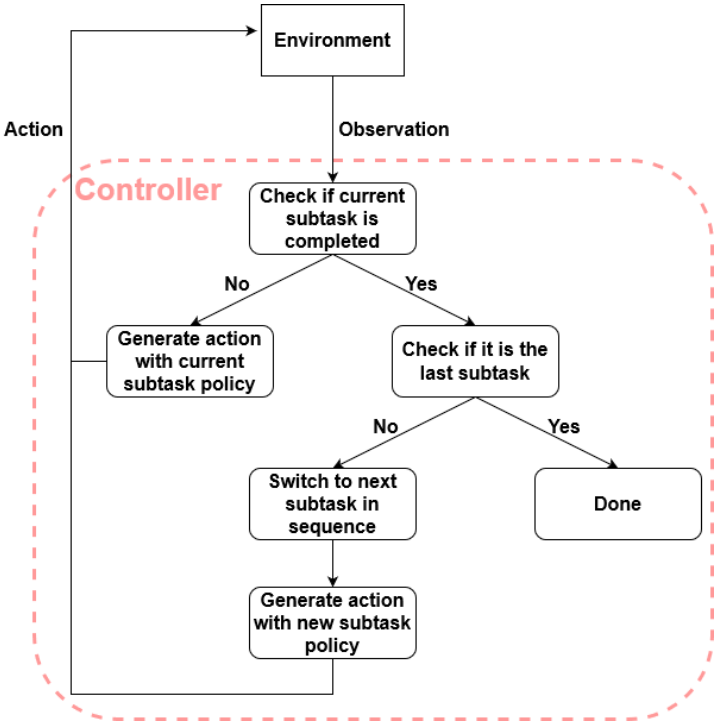


Figure 3.16: Overview of controller functionality

3.5.3. MOPRL summary

To summarize MOPRL, or the approach taken to solve LTL-tasks in stochastic environments: given an LTL-task that needs to be solved in the environment and manually defined subtasks in the environment of which a sequence of those subtasks can solve the LTL-task in the environment. We train agents for every subtask using RL and simulate the policy of every subtask to estimate the success probability and cost of every subtask. The subtasks are combined together in a graph, called the high-level model (HLM) in which sequences of subtasks can be found that solve reachability tasks in the environment. To be able to find sequences of subtasks for the LTL-task, the LTL-task is converted to a Büchi automata representing this LTL-task. The HLM and the Buchi automata are combined together into a new graph called the product graph, in which a sequence of subtasks can be found that complete the LTL-task. In this product graph, we use the estimated success probability and cost of every subtask to find Pareto-optimal sequences of subtasks that complete the LTL-task with Martins algorithm. One of these Pareto-optimal paths is chosen to generate a controller. This controller is then simulated and from these simulations, PAC-guarantees are calculated for both the success probability and average cost of using this controller, providing a bound on the probability that the expected performance is within a certain error margin of the estimated performance.

4

Related work

In this chapter, we highlight current research on the topic of solving complex tasks in sequential decision-making processes. These researches are divided into three types; reward machine, task decomposition, and hierarchical RL. In each section, we explain the basics of each of these types and discuss some of the most relevant approaches of each type. These relevant approaches are used by other researches as comparison during evaluation [13], [21], [54] or focus on LTL-tasks specifically [10], [15]. At the end of this chapter, we provide an overview of all approaches discussed and the type of environment and tasks they focus on in Table 4.1.

4.1. Reward machine

One of the approaches often used to learn complex tasks is reward machines in combination with RL. To learn optimal behavior, an RL agent interacts with the environment and learns from its experience. To learn whether a certain action has a positive or negative impact on completing the task, the agent is provided with a reward generated by a reward function. These reward functions receive information about the agent's current situation and provide the agent with a suitable reward. When dealing with simple tasks such as reachability tasks, it is easy to provide a suitable reward function to the agent. However, when dealing with more complex tasks where not only the current situation of the agent is important, but also which part of the task have yet to be completed, it becomes more difficult to provide reward functions that can be used by the agent to successfully learn the task.

A reward machine is a system that provides an agent with a suited reward function based on the state of the task, the state of the environment, and possibly other variables. Reward machines allow for composing different reward functions in flexible ways, including loops, conditional rules, and other techniques. As an agent acts in the environment, moving from state to state, the reward machine can also move to a different state, depending on whether certain events have happened in the environment or not. After every transition, the reward machine outputs the reward function the agent should use at that time. For example, an agent needs to learn the task ”deliver coffee and mail to the office”. When the agent has no coffee, the agent needs to learn how to get a coffee, and once it has a coffee, how to deliver it to the office, the same idea applies to the mail. It may be beneficial to provide the agent with different reward functions, depending on whether the agent has acquired a coffee and/or mail or not [16].

Reward machines are often designed as finite state machines, they take an abstracted description of the environment as input, and output a reward function. This allows for the agent to be rewarded differently based on the state of the reward machine. If we go back to the example of ”deliver coffee and mail to an office”, an example of a reward machine that represents this task is in Figure 4.1. In this reward machine, the agent can only receive a positive reward from the reward machine by visiting the office after both the mail and the coffee have been collected.

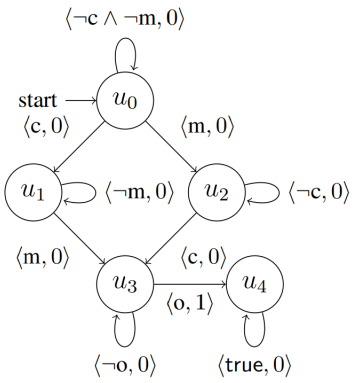


Figure 4.1: Example of a reward machine for the task "deliver coffee and mail to an office". c means coffee is collected, m stands for mail is collected and o means that the office is reached [57]

Compared to the approach used in this paper, reward machines have the benefit that they are guaranteed to converge to an optimal policy in the case of tabular reinforcement learning [57], our approach provides no such guarantees. A downside of reward machines is that they do scale poorly to complex tasks that require high-level planning [13]. Therefore, reward machines are the most suited for small to decently-sized environments and tasks.

4.1.1. Relevant reward machine approaches

A Composable Specification Language for Reinforcement Learning Tasks [58] proposes a task specification language called SPECTRL. This language can be used to define complex tasks, which contains the following syntax:

$$\phi ::= achieve\ b \mid \phi_1 ensuring\ b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

A specification in this language is a logic formula that can be used to evaluate whether a certain sequence of actions in an environment $S = s_0 \xrightarrow{a_0} ... \xrightarrow{a_t} s_t$ accomplished the desired task. They provide an algorithm that takes an environment and task specification as input and uses this to generate a task monitor, which is a finite-state machine that tracks which parts of the task are completed and which constraints have been violated. It uses this task monitor to provide an agent with partial rewards, based on the degree to which the specification was satisfied during a run.

Reinforcement Learning With Temporal Logic Rewards [14] proposes a specification language for robotic applications, called TLTL. TLTL operates over finite-time trajectories of an agent's state. TLTL formulas are evaluated against finite time sequences of events. These events are produced by the environment. They define quantitative semantics for TLTL, which they call robustness degree. The robustness degree exists out of real-valued reward functions. These reward functions take a TLTL-task and state trajectory as input and outputs a value indicating how far from satisfying or violating the specification the trajectory was. These rewards are used by an agent to learn a policy for how to complete the task.

LTL2Action: Generalizing LTL Instructions for Multi-Task RL [15] uses a labeling function to assign propositions to states in the environment MDP, then during training, these propositions are used to check whether the LTL-task is satisfied. When the LTL-task is satisfied the agents get a reward of 1, and when the LTL-task is falsified, the agent gets a reward of -1. To deal with the time modalities of LTL, they use LTL progression, LTL progression is a semantics-preserving rewriting procedure that takes an LTL formula and the current labeled state as input and returns a formula that identifies aspects of the task that remain to be addressed.

LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning [10] translates specifications of various formal languages into reward machines. They use a labeling function to relate $S \times A \times S \rightarrow 2^p$, i.e. they relate state transitions to propositions over the alphabet

p. They propose a reward machine based on mealy machines, this mealy machine takes specification propositions over 2^p as input and outputs an appropriate reward function. In order to create the mealy machine, they require the deterministic finite automata(s) representing the specification(s).

PlanGAN: Model-based Planning With Sparse Rewards and Multiple Goals [59] aims to solve environments with multiple tasks and sparse rewards. They use a model-based RL method in which any goal observed during a given run can be used as an example of how to achieve that goal from states that occurred earlier on in that same run. They generate rewards based on how far the run was from reaching one of the multiple goals. They use this information to train an ensemble of networks (GANs) which learn to generate plausible future trajectories conditioned on achieving a particular goal instead of directly learning a goal-conditioned policy. After they have this ensemble of networks they use it in combination with a planning algorithm to achieve the main goal in as few steps as possible.

Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning [57] create a finite state machine that takes an abstract description of the environment as an input and outputs a reward function. In this abstract description, only high-level events in the environment can be detected by the agent. With this, the agent will be rewarded differently at different times, enabling rewards for temporally extended tasks. The reward machine is defined over a set of propositions related to the high-level events that can happen in the environment. The reward machine traverses from one state to another if a certain event happens in the environment. They decompose the reward machine into multiple subtasks, one subtask per state in the reward machine, and learn a policy for every subtask.

4.2. Task decomposition

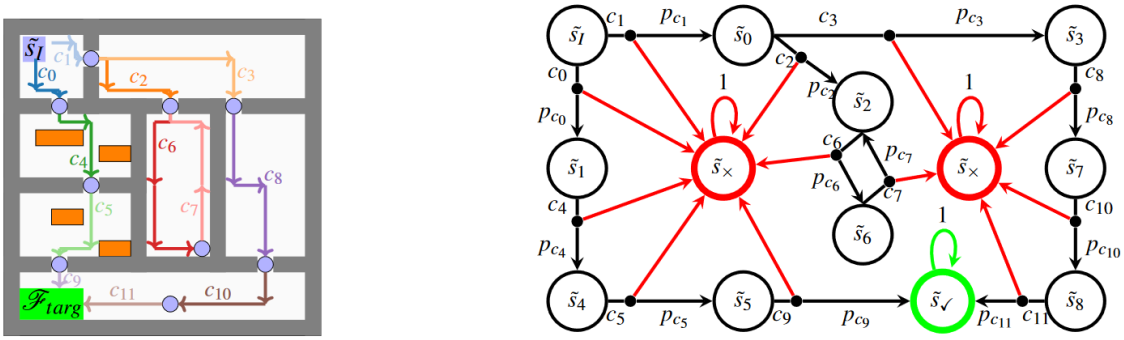
Task decomposition is the type of approach most related to the approach taken in this research. In task decomposition, a global task is seen as a sequence or composition of subtasks. When these subtasks are executed successfully, the global task is completed. Subtasks provide information about high-level structural relationships among them, but not how to implement each subtask or which subtasks need to be executed to complete the main task. Task decomposition approaches define two levels, a low level at which the policies for the subtasks function, and a high level, at which a planning algorithm is used to make use of the structural relationships between subtasks and find the subtasks that need to be completed and the order in which they need to be completed [13].

To create the subtasks, there are two options: design the subtasks manually or automatically generate subtasks from the global task. When subtasks are manually designed, the subtasks are unrelated to the global task, and the main challenge is to find which subtasks to complete and the order in which they need to be completed. When subtasks are automatically generated from the global task, the task specification structure is leveraged to directly create subtasks that need to be completed, and there is almost none or very little planning to be done on a high level since all subtasks already have some relation to the global task.

To learn a low-level policy for subtasks, RL is often used. Most subtasks are ”easy” tasks and therefore do not require a complex reward function to learn. Some approaches first train agents for all subtasks and after training, use the performance of every subtask’s policy to find the best sequence of subtasks to execute in order to complete the global task [13]. Others interleave planning and training, using planning to first find a sequence of subtasks to train, and then only train the subtasks in that sequence, if RL fails to train a policy for one of the subtasks in the sequence, it plans for a new sequence and trains the subtasks in the new sequence until it successfully found a sequence of subtasks it can train agents for [25], [60].

To find a sequence of subtasks the structural relationship between subtasks needs to be modeled in some sort of framework. To create such a model one can use a parametric MDP (pMDP) [25], a graph, as done in *Compositional Reinforcement Learning from Logical Specifications* [13] and this research or another type of framework. After such a model is created, a planning algorithm can be used to find a composition of subtasks to execute that completes the global task. Figure 4.2 is an example

of an environment in which a navigation task is divided into multiple smaller subtasks and planning is used to find which sequence of subtasks to execute.



(a) The labyrinth task environment.

(b) The HLM corresponding to the labyrinth example.

Figure 4.2: Example of task decomposition, in which there are twelve manually defined subtasks. Each subtask in the environment is represented by a colored path and entry. Exit conditions for the subtasks are shown as blue circles. In the HLM, each subtask causes a transition to its successor state with probability P_c. Otherwise, the HLM will transition to the failure state $\tilde{s}_\times$ with probability $1 - P_c$, visualized with red transitions [25]

Task decomposition approaches use the same main idea as this research, finding a sequence of subtasks to execute to complete a global task. However, existing task decomposition approaches focus on settings with one objective, such as success probability or reward instead of a setting with multiple objectives.

4.2.1. Relevant task decomposition approaches

Verifiable and Compositional Reinforcement Learning Systems [25] proposes a framework that divides an environment into subsystems and presents the relation between the subsystems as a parametric MDP. A parametric MDP is an MDP in which the transition probabilities represent the likelihood of the outcomes that could occur when the subsystem is executed. This allows for the decomposition of a task specification into smaller tasks for the subsystems, which can then be trained and tested in order to achieve the specification. They consider qualitative specification, such as *there is at least a 95% chance the goal is reached*. They use Mixed Integer Linear Programming (MILP) to find which subtasks need to be completed and the required success probability for each of these subtasks in order to satisfy the specification and then train the selected subsystems until they achieve this success probability.

Compositional Reinforcement Learning from Logical Specifications [13] aims to learn a control policy for complex tasks given by a logical specification and use a compositional approach, called DiRL. They reduce the problem for a given MDP M and a specification ϕ to an abstract reachability problem for M by constructing an abstract graph G_ϕ inductively from ϕ. Each predicate p in the specification will be associated with a subgoal region in M, where p is True in that region. It then uses RL to find policies for each edge in G_ϕ, which can transit the system from one vertex in G_ϕ to the next. To find trajectories that satisfy the reachability problem in G_ϕ they use Dijkstra and the success probabilities of the edges in G_ϕ to find a path. A path then correlates with the subtasks that need to be executed to complete the global task.

Teaching Multiple Tasks to an RL Agent using LTL [54] describes a framework, called LPOPL, that uses LTL progression to divide a set of tasks provided in co-safe LTL, a subset of LTL for which truth of a formula can be assured in a finite number of timesteps, into multiple subtasks and then learn a policy for those subtasks. They support the following LTL operators:

$\phi ::= p | \neg\phi | \phi_1 \wedge \phi_2 | \circ \phi | \phi_1 \cup \phi_2$ with $p \in P$

LTL progression uses the fact that an LTL formula can progress along a sequence of truth assignments, at each time step the progression operator can be applied to update the task formula to reflect which parts are satisfied or unsatisfied. They subtract subtasks with LTL progression for each task in

the provided set of tasks and then learn a policy for each subtask.

4.3. Hierarchal reinforcement learning

Hierarchical RL is a mechanism that allows for RL to learn a challenging long-horizon task by decomposing the task into simpler subtasks using a hierarchy of policies. The highest level policy chooses the subtasks to complete and learns how to complete the main task by selecting the correct subtask to complete given the current state. The low-level policies are trained to successfully be able to perform the subtasks [61].

Subtasks in Hierarchical RL are formally defined as $\omega = [\pi, (r, g), (I, \beta)]$

- A policy π, the policy of the subtask, mapping states to primitive actions
- The objective component (r, g)
  - r, the reward used to train π
  - g, the subgoal or set of subgoals associated with ω
- the execution component (I, β)
  - I, the initial condition of ω, a set of states of which execution of ω may start
  - β termination conditions, such as a time limit or when the subgoal is reached

There are two approaches to learning a hierarchical RL policy, one with subtask discovery and one in which the subtasks are provided manually. When the subtask space needs to be discovered, it can either be done simultaneously with learning the hierarchical policy or before training the hierarchical RL agent. When the subtasks are manually crafted beforehand and supplied as input for the hierarchical RL agent, it only needs to learn policies for solving the subtasks as well as learn a higher-level policy that selects which subtask to execute when.

A hierarchical RL policy needs to have at least two hierarchies, one lower hierarchy, which are the lowest subtask controllers that directly output actions that can be executed in the environment, and a top-level policy that dictates which subtasks to execute. It is possible to have more than two hierarchies, resulting into a tree-like structure of policies. Figure 4.3 gives an example of a hierarchical RL agent and its corresponding environment. In this example, the agent has 3 policy levels, the environment passes the state to the agent's top-level policy, which selects a policy one level lower, and passes the observation to the selected policy, this propagates until one of the lowest-level subtask policies is selected, which generates an action that can be executed in the environment.

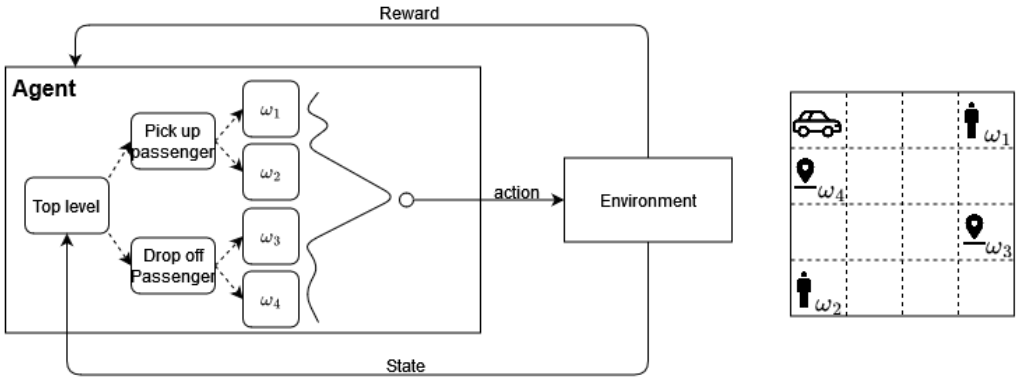


Figure 4.3: example of a hierarchical RL agent, in which an agent needs to pick up passengers and drop them off at a certain location

An example of a hierarchical RL process is given in Figure 4.4. In this example, a hierarchical RL agent decomposes and performs the complex task "Going to Hawaii" (GTH). The hierarchical RL agent is

composed of a hierarchy of policies, with a total of three levels. The task policy π_{GTH} (highest level) decomposes the original task GTH into the two subtasks "Book Tickets" (BT) and "Go To Airport" (GTA). π_{GTH} initially chooses the subtask BT. Then, BT is executed until its termination occurs at time $T3$. During this period, the policy of the subtask BT, π_{BT}, chooses different even easier subtasks to execute on a level one below itself (lowest level). These even easier subtasks are "Open Booking Website" (OBW) and "Enter Flight Information" (EFI). After BT terminates at $T3$, π_{GTH} chooses to execute GTA, which itself chooses the subtask "Go to Taxi Stand" (GTS), one level below itself. During each timestep, a primitive action is chosen by the lowest level subtask policy, e.g. π_{OBW}, π_{EWI} or π_{GTS}. The hierarchical RL agent receives a task reward r_{GTH} in response to the primitive action, which is accumulated and given at different time scales to different levels. The task reward may be optional for the policies below the highest level, which can be trained using subtask-related rewards. In this manner, the hierarchical RL process continues in time until GTH finishes. Figure 4.5 shows the hierarchy all subtasks are divided in.

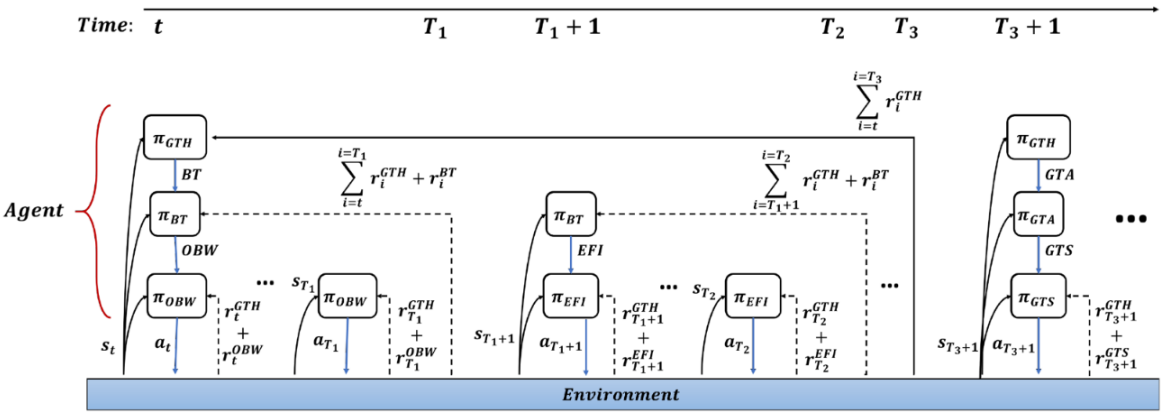


Figure 4.4: Example of a hierarchical RL process, the task is "Go to Hawaii" (GTH), which is divided into two subtasks, "book tickets" (BT) and "Go to Airport" (GTA), which themselves have other even smaller subtasks beneath them [61]

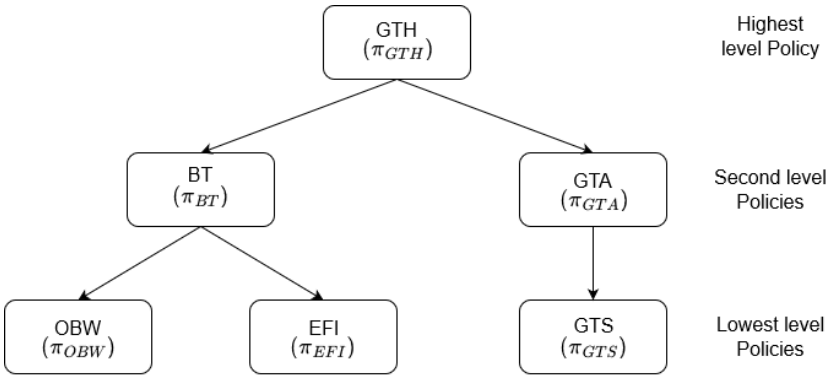


Figure 4.5: The hierarchy in which all policies from Figure 4.4 are divided

Most existing hierarchical RL approaches only work in discrete domains, or require pre-trained low-level controllers. In standard hierarchical RL, the type of tasks that can be solved are sequential tasks [62]. The difference between hierarchical RL methods and the approach used in this research is that hierarchical RL methods only optimize for one objective, instead of multiple. Our approach also is able to create policies for LTL tasks, while most hierarchical RL approaches only support sequential tasks.

4.3.1. Relevant hierarchical reinforcement learning approaches

Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning [16] combines ideas of reward machines with hierarchical RL to create a reward machine, called HRM to solve complex tasks in an environment. They base their implementation on the options framework specifically, in

which the problem is decomposed into multiple subproblems with $(\mathbb{I}, \pi, \beta)$ where $\mathbb{I}$ is the initial condition, π is the policy for solving the subproblem and β gives the probability that the option will terminate in every state. They use the option framework on a cross-MDP of the environment and a reward machine to learn how to move from one state in the reward machine to another. A higher-level policy is then used to learn how to select among those options in order to collect rewards. With this approach, they aim to learn a policy for every edge in the reward machine and use the higher-level policy to select which edge to traverse in the reward machine.

Data-Efficient Hierarchical Reinforcement Learning [17] presents an approach called HIRO to learn highly complex behaviors for simulated robots, such as pushing objects and utilizing them to reach target locations. HIRO uses a two-layer structure, with a lower level policy μ^{lo} and a higher level policy μ^{hi}. The higher-level policy works at a coarser layer and sets goals for the lower-level policy. There can potentially be an infinite number of low-level policies and each is trained to match its observed state to the desired goal. The high-level policy receives an observation from the environment and produces a high-level action, which is then given to the low-level policy and translated into actions for the environment, the environment provides the high-level policy with a reward, and the high-level policy uses this reward to provide the lower-level policies with a reward using a fixed parameterized reward function.

Abstract Value Iteration for Hierarchical Reinforcement Learning [21] is a hierarchical approach using model-based RL for continuous state and action spaces. The user provides a set of subgoal regions and their approach learns options that serve as transitions between these subgoal regions, the writers create A-avi, an algorithm that alternates between learning a policy for each subtask and estimating the transition probabilities and the expected reward, and finally uses value iteration to compute an optimal option policy (which options to complete).

Deep reinforcement learning with temporal logics [62] creates policies for tasks described with LTL in continuous environments. The LTL property acts as a high-level exploration guide for the agent, where low-level planning is handled by a deep RL architecture. They convert the LTL task to an automaton and create a product between the automaton and the MDP representing the environment in order to synchronize them. They then use automatic task decomposition to create subtasks in the product MDP by considering every state in the automaton a ”task divider”, so every transition in the automaton is considered a subtask. After this, Modular Deep Deterministic Policy Gradient (Modular DDPG) is used to learn a policy in the product MDP. In modular DDPG there are multiple neural networks, one for each subtask, to learn a policy in the product MDP. The set of neural nets acts as a global modular actor-critic deep RL architecture, which allows the agent to jump from one sub-task to another by just switching between the set of neural nets.

Solution	Task specification	Type of objective	Approach	Planning algorithm	Randomness	Environment type
SPECTRL [58]	SPECTRL	Single	Reward machine	-	Determinisitc	Continuous
TLTL [14]	truncated LTL	Single	Reward machine	-	Deterministic	Continuous
LTL2action [15]	LTL	Single	Reward machine	-	Deterministic	Discrete & Continuous
LTL and Beyond [10]	LTL	Single	Reward machine	-	Deterministic	Discrete & Continuous
PlanGAN [59]	not formally specified	Single	Reward machine	-	Deterministic	Continuous
QRM [57]	not formally specified	Single	Reward machine	-	Deterministic	Discrete & Continuous
Verifiable & Compositional RL [25]	Reachability objectives	Single	Task decomposition and planning	MILP	Stochastic	Discrete & Continuous
DiRL [13]	SPECTRL	Single	Task decomposition and planning	Dijkstra	Deterministic	Continuous
LPOPL [54]	co-safe LTL	Single	Task decomposition and planning	LTL progression	Deterministic	Discrete
HRM [16]	not formally specified	Single	Hierarchical RL	-	Deterministic	Discrete & Continuous
HIRO [17]	not formally specified	Single	Hierarchical RL	-	Deterministic	Continuous
A-avi [21]	not formally specified	Single	Hierarchical RL	-	Deterministic	Continuous
DDPG with LTL [21]	LTL	Single	Hierarchical RL	-	Deterministic	Continuous

Table 4.1: Overview of approaches that try to solve tasks in environments with sparse rewards

5

Implementation and evaluation

In this chapter, we discuss how the environment and MOPRL are implemented and evaluate the use of MOPRL. The code of this thesis can be found at `https://github.com/casoku/thesis`.

5.1. Implementation

In this section, we go over the libraries used to implement MOPRL and the environment. We first go over the libraries used to create the environment, followed by the libraries used to convert LTL-tasks to Büchi-automata and the last subsection is about the RL algorithm used to learn policies for the subtasks.

5.1.1. Environment

To implement the environment, OpenAI gym [63] and Minigrid [64] are used. OpenAI gym is a python library that provides an application programming interface (API) between a learning algorithm and the environment. Minigrid is a python package that comes with a few standard discrete grid environments and allows for creating discrete grid environments that are compatible with OpenAI gym.

5.1.2. LTL-tasks and Büchi automata

The LTL-tasks are implemented with Spot automata [35]. Spot is a library that can convert LTL formulas into Büchi automata. Spot support the following LTL operators:

- True : 1
- False : 0
- not : !
- and : &
- or : |
- implies : ->
- equivalence : <->
- xor : ∧
- next : X
- eventually : F
- globally : G
- weak until : W
- strong until : U
- weak release: V, R
- strong release: M

More information about LTL can be found in section 2.3, and more information about Büchi automata can be found in section 2.4. The sign after the colon is the symbol that spot uses to represent that operator.

Any LTL-task that exists out of these operators can be converted by Spot to a Büchi automata, which is then used to create the product graph. We use the simplified representation of the Büchi automata since we are only interested in viable paths. The simple representation removes edges and states that would lead to states in which the LTL becomes unsatisfiable, using the simple representation leads to smaller product graphs.

5.1.3. Training agents

In order to train the agents on how to complete the subtasks, we use Stable-Baselines3 [65] Proximal Policy Optimization (PPO) [66] implementation. PPO is a policy gradient method, so it optimizes its policy with respect to the expected return (long-term cumulative reward) and supports both discrete action and state spaces. PPO is known to have a good balance between sample complexity and simplicity, and has an overall good performance, outperforming other state-of-the-art RL algorithms in many different environments.

5.2. Evaluation

The evaluation of MOPRL exists of three parts, in the first part MOPRL is compared to two other state-of-the-art approaches. To be able to make the comparison, we had to make two adjustments to their implementation. The first adjustment was made in order to make the other approaches compatible with discrete environments. The second modification was made since both state-of-the-art approaches only optimize for a single objective, the success probability. We added extra functionality so that we are also able to collect the cost when using these approaches. The approaches MOPRL is compared against are DiRL [13] and SPECTRL[58]. For each of these approaches, there will be a section explaining what adjustments were made. The second part exists out of analyzing MOPRL when applied in a more challenging environment. We analyze the use of multi-objective planning, how the planning procedure scales, and we provide PAC-guarantees for a number of tasks. In the third part, we use the results of the first and the second part to provide some insights about the use of MOPRL.

5.2.1. Comparison

The comparison is performed in an environment with four rooms, see Figure 5.1. In this environment, the agent starts at the top left position and there are two goal locations, the purple square, and the green square. The blue dots are moving obstacles the agent needs to evade.

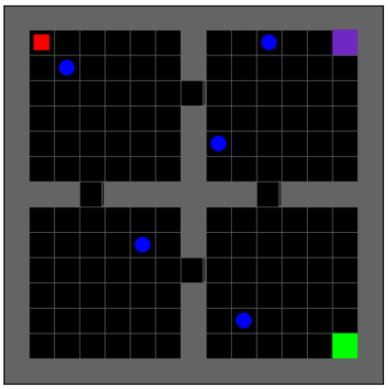


Figure 5.1: Environment used for comparison with different approaches

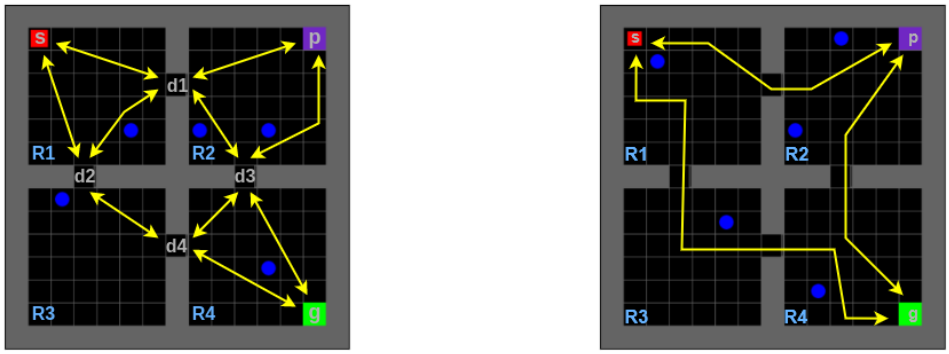
We test five different LTL-tasks to see how MOPRL compares to SPECTRL and DiRL, these five LTL-tasks are described in Table 5.1.

Task	Type	Description
$\Diamond g \vee \Diamond P$	Choice	Eventually reach green or eventually reach purple
$\Diamond(g \wedge \Diamond p)$	Sequential	Eventually reach green and next eventually reach purple
$\Diamond g$	Reachability	Eventually reach green
$\Diamond g \wedge \Box \neg r3$	Reach-avoid	Eventually reach green while always avoiding room 3 (bottom left room)
$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	Difficult	Eventually reach green, then eventually reach purple, then eventually reach green, and then eventually reach start

Table 5.1: Tasks used for comparison

Each task is trained for a total of 1,000,000 steps, so if an approach uses five subtasks, the number of steps used to learn each subtask is $\frac{1{,}000{,}000}{5} = 200{,}000$. When training is completed for a subtask, it is simulated 600 times and the success probability and average cost are calculated.

For MOPRL, we evaluated two different types of subtasks, see Figure 5.2. In Figure 5.2a we use small subtasks, which resulted in a total of 20 subtasks. In Figure 5.2b we use big subtasks, resulting in a total of 6 subtasks.



(a) Small subtasks used for the evaluation

(b) Big subtasks used for the evaluation

Figure 5.2: Different subtask sizes used for evaluation

DiRL

DiRL is designed for deterministic continuous environments and not stochastic discrete. They automatically decompose the main task into subtasks instead of manually defining subtasks in the environment, which we do for this research. They use RL algorithms suited specifically for continuous environments to learn a policy for those subtasks. We added support for the PPO algorithm from Stable-Baselines 3 used in this research so we can train subtasks in discrete environments.

The subtasks created by DiRL can potentially require the entire environment in order to be completed, therefore we supply the agent with global observations of the environment. The observation exists out of the position of the agent and the location of all moving obstacles in the environment.

When training an agent, DiRL provides rewards solely based on the subtask, and not on the state of the environment, so when dealing with moving obstacles plain DiRL would not get a negative reward when colliding with an obstacle unless it is a part of the subtask specification. We change this such that 50% of the reward is based on the subtask and 50% is based on the state of the environment so that when the agent crashes with an object, it still receives a negative reward.

SPECTRL

SPECTRL is also designed for deterministic continuous environments, just like DiRL. SPECTRL uses a task specification and the environment MDP and combines these together into a task monitor. The task monitor tracks how far the task has progressed and provides the agent with rewards based on this progression. For example, if the task is ”eventually go to green and then go to purple”, once green has been reached SPECTRL will provide the agent with a reward and then if the agent reaches purple afterward, SPECTRL will provide the agent with another reward.

SPECTRL itself is very intertwined with the learning algorithm (ARS) they use. However, ARS is only suited for continuous actions and state spaces, and in this thesis, the environment has discrete action and state spaces. They also use trajectory-based learning, but openAI gym environments use step-based rewards. In order to be able to apply SPECTRL on the discrete room environment a wrapper was created. This wrapper has two functions; first, it is to decouple SPECTRL from the learning algorithm. We want to be able to use the PPO algorithm since our environment is discrete. The second function of the wrapper is that it keeps track of the current trajectory of the agent and the trajectory in the previous step. SPECTRL calculates the reward based on these trajectories, and if the reward of the current trajectory is higher than the reward of the trajectory of the previous step, we reward the agent with a positive reward. With this change it is possible to use learning algorithms for step-based rewards, this enables the use of the PPO algorithm or any general learning algorithm for openAI gym environments.

Comparison results

For every task, a controller is generated and simulated. We repeated this three times and from these three experiments, we calculate an average success probability and cost which are reported in Figure 5.3 and Figure 5.4. All data used for the comparison can be found in Appendix A.

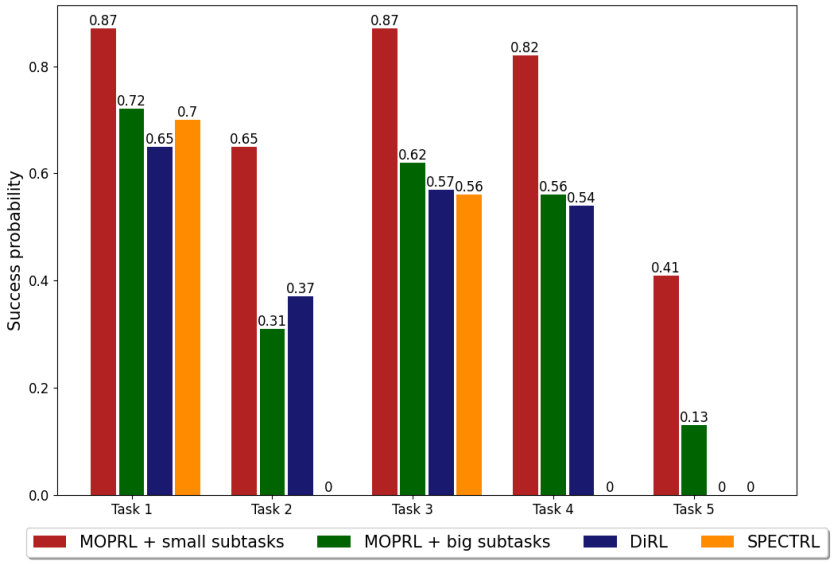


Figure 5.3: Success probability comparison of different approaches

Figure 5.3 displays the average success probability of the tested approaches (higher is better). MOPRL + small subtasks has the highest success probability for all five tasks. DiRL, MOPRL + big subtasks, and SPECTRL are fairly close in performance for task 1 and task 3, with MOPRL + big subtasks being marginally better. SPECTRL was unable to solve task 2, task 4, and task 5. These tasks were the most difficult to learn since they either are sequential (task 2 and task 5) or need the agent to avoid a certain region (task 4). It seems that SPECTRL is unable to create a controller for challenging tasks in environments with sparse rewards. This is probably because SPECTRL does not have any built-in functionality to deal with sparse rewards. It uses a task monitor to track how much of a task is

completed, but if it is already difficult to learn how to do the initial part of a task, it cannot learn how to do the entire task with this approach. DiRL is unable to create a controller for task 5, this is because DiRL creates 6 big subtasks for task 5, which gives each subtask a budget of $\frac{1000000}{6} = 166666$ steps, which was not enough to be able to learn a policy for the subtasks defined by DiRL.

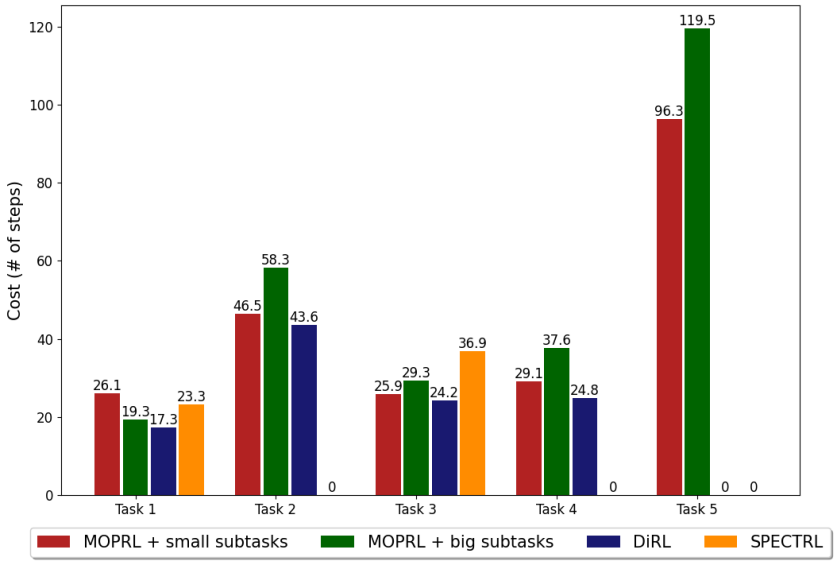
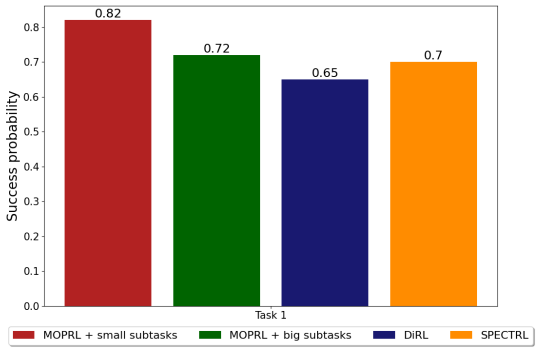
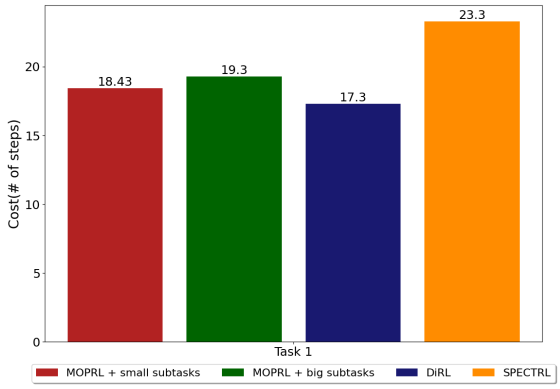


Figure 5.4: Cost comparison of different approaches

Figure 5.4 displays the average cost of the tested approaches (lower is better). DiRL has the lowest cost for tasks 1, 2, 3, and 4. After DiRL, MOPRL + small subtasks has the second lowest cost for tasks 2, 3, and 4. We'd like to highlight the difference in cost between DiRL and MOPRL + small subtasks for Task 1 ($\Diamond g \vee \Diamond p$). DiRL achieves a considerably better cost for this task. This is because for the controller of MOPRL + small subtasks the path (sequence of subtasks) in the product graph with the highest success probability is used. However, MOPRL + small subtasks also found another path that completes the task at a lower cost. In Figure 5.5 we made the comparison again for task 1, but in this case, we used the path with the lowest cost instead of the path with the highest success probability to generate the controller for MOPRL + small subtasks to see if the cost of MOPRL + small subtask would be closer to the cost of DiRL.



(a) Success probability comparison for task 1, when we use the path (sequence of subtasks) with the lowest cost to create a controller



(b) Cost comparison for task 1, when we use the path (sequence of subtasks) with the lowest cost to create a controller

Figure 5.5: Comparing different approaches for task 1, when we use the path (sequence of subtasks) with the lowest cost to create a controller

When using the path with the lowest cost for the generation of the controller for MOPRL + small subtasks, we still get the highest success probability (Figure 5.5a) of all the approaches, but now the cost is better than MOPRL + big subtasks and SPECTRL, and closer to DiRL compared to when we used the path with the highest success probability (Figure 5.3 and Figure 5.4). This is because when we optimize for success probability, MOPRL chooses a sequence of subtasks that steers the agent toward the green square via the bottom left room, since this path is deemed safer and results in a higher success probability. When we optimize for cost, MOPRL sends the agent to the purple square, which is closer to the starting location than the green square, but more challenging to reach. DiRL always steers the agent toward the purple square.

Summary of comparison
We compared two versions of MOPRL, one in which we defined big subtasks and one in which we defined small subtasks with two other approaches, DiRL and SPECTRL in an environment with four rooms for five different tasks. MOPRL in combination with small subtasks has the highest success probability for all five tasks, both when we optimize the sequence of subtasks for cost or for success probability. DiRL has the lowest cost for all tasks, except task 5, since DiRL is unable to create a controller that can complete that task. After DiRL, MOPRL with small subtasks achieves the lowest cost for all tasks.

5.2.2. Performance in challenging environment
To analyze the performance of MOPRL, we apply it in a bigger environment with different types of rooms, see Figure 5.6. A total of 68 subtasks were defined in this environment and for each subtask an agent was trained for 40000 steps, resulting in a total of $40,000 * 68 = 2,720,000$ training steps. In this environment and the environment used for the comparison (Figure 5.2a) a number of tasks will be tested in order to gain insight into the following:

- Generate Pareto paths; Find Pareto optimal paths for different tasks and compare the probability and cost estimated during planning to the success probability and cost calculated from simulations
- Scalability of planning; Create different sizes of product graphs and measure how long it takes to generate the product graph and find all Pareto optimal paths
- Calculate PAC-guarantees; calculate the PAC-guarantees for all tested tasks and see how strict the bounds are it provides

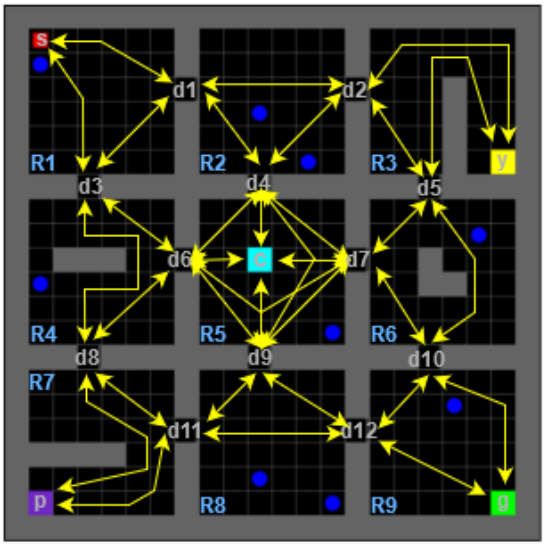


Figure 5.6: Environment used for performance evaluation, each yellow arrow is a subtask

Generated Pareto paths

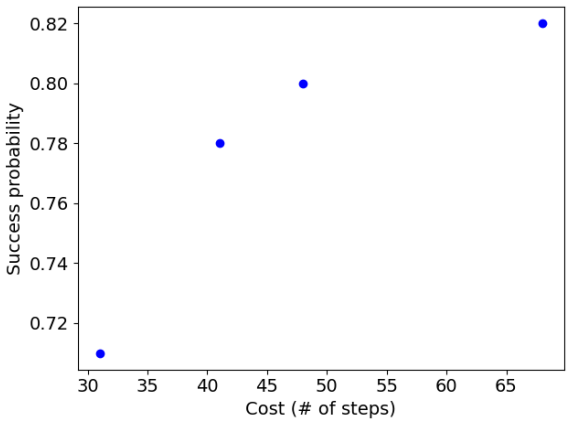
To evaluate the use of multi-objective planning for finding paths (sequence of subtasks) in the product graph, we run three different tasks in the environment in Figure 5.6. These tasks are designed to have multiple possible paths leading toward completing the LTL-task. We plot the calculated Pareto optimal paths found and compare the performance estimated during planning to the performance found from simulating the entire controller. The tasks we tested can be found in Table 5.2.

Task	Type	Description
$\Diamond\, g \vee \Diamond\, y$	Choice	Eventually reach green or eventually reach yellow
$\Diamond(\neg R4\ U\ p \wedge \neg R7\ U\ g)$	Strong until	Eventually reach purple while never entering room 4 and eventually reach green while never entering room 7
$\Diamond y \wedge \Diamond c \wedge \Diamond p \wedge \Diamond g$	Sequential choice	Eventually reach yellow, cyan, purple and green in any order

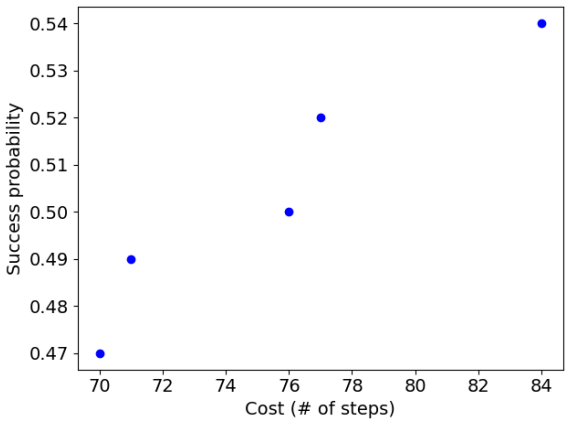
Table 5.2: Tasks used for generating Pareto paths

The first task is a simple choice task, which requires less than 7 subtasks to be completed. The second task is an until task, in this task the agent cannot enter certain rooms until a condition is met, and takes around 10 subtasks to complete. The third task is a sequential choice task in which the agent needs to complete a number of goals, but is free to choose the order in which it completes these goals. The third task takes more than 13 subtasks to be completed.

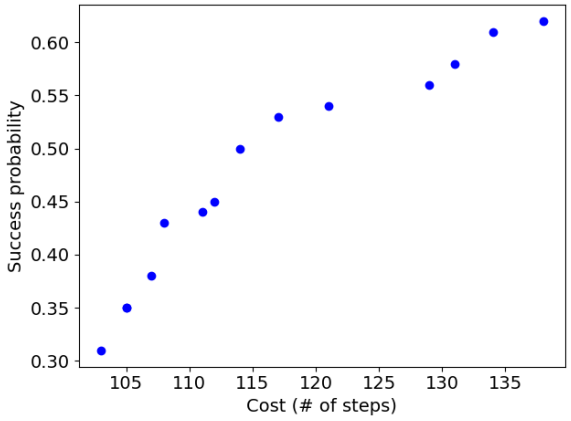
Figure 5.7 displays all the points on the Pareto front found by MOPRL for each task in Table 5.2. Task 1 (Figure 5.7a) has a total of four Pareto optimal paths, of which the highest estimated success probability is 0.82 and the lowest is 0.71. The highest estimated cost is 68 and the lowest is 31. Task 2 (Figure 5.7b) has a total of five Pareto optimal paths, the highest estimated success probability is 0.54 and the lowest is 0.47. The highest estimated cost is 84 and the lowest is 70. The third task (Figure 5.7c) has thirteen Pareto optimal paths, the highest estimated success probability is 0.62 and the lowest is 0.31. The highest estimated cost is 138 and the lowest is 103.



(a) All 4 Pareto optimal paths of the task $\Diamond F\ g \vee F\ y$



(b) All 5 Pareto optimal paths of the task $\Diamond(\neg R4\ U\ p \wedge \neg R7\ U\ g)$



(c) All 13 Pareto optimal of the task $\Diamond y \wedge \Diamond c \wedge \Diamond p \wedge \Diamond g$

Figure 5.7: Pareto optimal paths found by martins algorithm in the product graph

In Figure 5.7a the four Pareto optimal points are spread out, meaning there is a notable difference between each possible path. For example, if we compare the point $(0.82,\ 68)$ and the point $(0.8,\ 48)$, we can see that with a drop of just 0.02 in success probability it is possible to improve the estimated cost by 20. Figure 5.7b has five Pareto optimal points, however, the difference between points for this task is not as big, for cost the biggest difference between two points is 7, $(0.54,\ 84)$ and $(0.52,\ 77)$. Figure 5.7c has a total of thirteen points, with the difference between each point being fairly small. However, the difference between the points with the highest and lowest success probability, $(0.62,\ 138)$ and $(0.31,\ 103)$ is 0.3, which is a notable difference.

The success probabilities of each point in Figure 5.7 is an estimation calculated by taking the product of the success probabilities of all subtasks in the path related to that point. The cost is estimated by taking the sum of the estimated cost of the subtasks in the path. Each of these points has a certain margin of error since the cost and success probability of a subtask is an estimation. In Table 5.3, Table 5.4, and Table 5.5 we compare the estimated success probability and cost with the measured success probability and cost for the three tasks. In these tables, the difference between the estimation and measurement is noted in either green, red or black. If the difference is green, the measured success probability or cost is better than the estimated value, if the difference is red, the measured value is worse and if it is black they are equal. For every measured value, we also provide the standard deviation (the number behind the ±). To gain the measured success probability the entire controller related to a Pareto optimal point is simulated 600 times to calculate the success probability, from these 600 simulations, all successful runs are used to calculate the measured cost. The success probability and cost are calculated with the formulas from subsection 3.2.2.

Pareto optimal path	Estimated success probability	Measured success probability	Difference success probability	Estimated cost	Measured cost	Difference cost
Path 1	0.82	0.826 ±0.015	+0.006	68	72 ±29.7	-4
Path 2	0.8	0.78 ±0.017	-0.02	48	47.7 ±3.5	+0.3
Path 3	0.78	0.788 ±0.017	+0.008	41	40.8 ±2.4	+0.2
Path 4	0.71	0.698 ±0.019	-0.012	31	30.3- ±3.6	+0.7

Table 5.3: Comparing the estimated success probability and cost to a measured success probability and cost for the task $\Diamond\ g \vee \Diamond\ y$

Pareto optimal path	Estimated success probability	Measured success probability	Difference success probability	Estimated cost	Measured cost	Difference cost
Path 1	0.54	0.56 ±0.02	+0.02	84	84.6 ±5	-0.4
Path 2	0.52	0.49 ±0.02	-0.03	77	76.7 ±5.9	+0.3
Path 3	0.5	0.47 ±0.02	-0.03	76	76.5 ±6.3	-0.5
Path 4	0.49	0.49 ±0.02	0	71	69.8 ±4.7	+1.2
Path 5	0.47	0.476 ±0.02	+0.006	70	70.7 ±6	-0.7

Table 5.4: Comparing the estimated success probability and cost to a measured success probability and cost for the task $\Diamond(\neg R4\ U\ p \wedge \neg R7\ U\ g)$

Pareto optimal path	Estimated success probability	Measured success probability	Difference success probability	Estimated cost	Measured cost	Difference cost
Path 1	0.62	0.611 ±0.02	-0.009	138	142.3 ±22.9	-4.3
Path 2	0.61	0.583 ±0.02	-0.027	134	138.4 ±23.4	-4.4
Path 3	0.58	0.553 ±0.02	-0.027	131	134.7 ±24.7	-3.7
Path 4	0.56	0.54 ±0.02	-0.02	129	134.5 ±27.3	-5.5
Path 5	0.54	0.53 ±0.02	-0.01	121	121.3 ±7.2	-0.3
Path 6	0.53	0.5 ±0.02	-0.03	117	117.9 ±5.3	-0.9
Path 7	0.5	0.496 ±0.02	-0.004	114	114.1 ±5.8	-0.1
Path 8	0.45	0.453 ±0.02	+0.003	112	113 ±6.6	-1
Path 9	0.44	0.416 ±0.02	-0.024	111	112.5 ±6.3	-1.5
Path 10	0.43	0.38 ±0.02	-0.05	108	108.8 ±6	-0.8
Path 11	0.38	0.4 ±0.02	+0.02	107	107 ±6.8	0
Path 12	0.35	0.33 ±0.019	-0.02	105	105.4 ±6.7	-0.4
Path 13	0.31	0.3 ±0.019	-0.01	103	103.3 ±6.9	-0.3

Table 5.5: Comparing the estimated success probability and cost to a measured success probability and cost for the task $\Diamond y \wedge \Diamond c \wedge \Diamond p \wedge \Diamond g$

In these tables we can see that the measured success probability is lower than the estimated success probability in most cases. The largest difference in success probability is path 10 in Table 5.5, where the difference is 0.05 (5%). For all other paths, the difference between the measured and estimated success probability is 0.03 or lower. For the cost, there is not a single path for which the difference is outside of expectation, since the standard deviation is larger than the difference for every path. The paths of the third task (Table 5.5) have a bigger difference between the estimated cost and measured cost than the other two tasks, this is because this task requires more steps to be completed than tasks 1 and 2, and thus has a bigger margin of error. When we look at the standard deviation of the cost, path 1 of Table 5.3 has a considerable standard deviation (29.7) for the measured cost compared to the other paths of this task. Path 1, path 2, and path 3 in Table 5.5 also have large standard deviations, but since these paths also require more steps it is less surprising than path 1 of Table 5.3.

The difference between the estimated and measured success probability can go up to 5% and the difference between the estimated and measured cost can go up to 5.5, so when Pareto optimal points are close to each other, it may be possible that a point which is Pareto optimal during planning, is dominated when we look at the measured values, an example of this is path 3 and path 4 in Table 5.4. Path 3 has a higher estimated success probability (0.5 against 0.49) and a higher cost (76 against 71) during planning, but during the measurements, path 4 has a higher success probability (0.49 vs 0.47) and a lower cost (69.8 vs 76.5) resulting in path 4 dominating path 3 when we look at the measured values.

Scalability of planning
In order to measure the scalability of planning we use both the environment from Figure 5.2a and Figure 5.6 and multiple LTL-tasks in order to generate product graphs of different sizes. We measure how long it takes for each product graph to be produced and how long it takes to find the Pareto optimal paths in these product graphs. All experiments are executed on an HP Zbook Power G9 with an Intel core I7-12700H and 16 GB RAM.

A total of ten tests were done, in Table 5.6 we display the size of the graph of the HLM, the size of the Büchi automata used to create the product graph, and the size of the final product graph of each test. The sizes in this table are displayed as (number of states x number of edges).

Test	HLM Size	Task size	Product graph size
Test 1	(7 x 20)	(2 x 3)	(7 x 18)
Test 2	(7 x 20)	(3 x 6)	(13 x 36)
Test 3	(7 x 20)	(5 x 14)	(25 x 72)
Test 4	(7 x 20)	(32 x 243)	(143 x 422)
Test 5	(7 x 20)	(128 x 2187)	(449 x 1282)
Test 6	(17 x 68)	(4 x 9)	(46 x 128)
Test 7	(17 x 68)	(16 x 81)	(227 x 796)
Test 8	(17 x 68)	(19 x 100)	(276 x 1033)
Test 9	(17 x 68)	(32 x 243)	(452 x 1596)
Test 10	(17 x 68)	(64 x 729)	(885 x 3118)

Table 5.6: Different sizes of environments and tasks tested for scalability

Each test was run 50 times to collect data on the time needed for creating the product graph and on the time it took to find all Pareto optimal paths in the product graph. This data is used to create boxplot graphs displaying the computation time of each of these two steps.

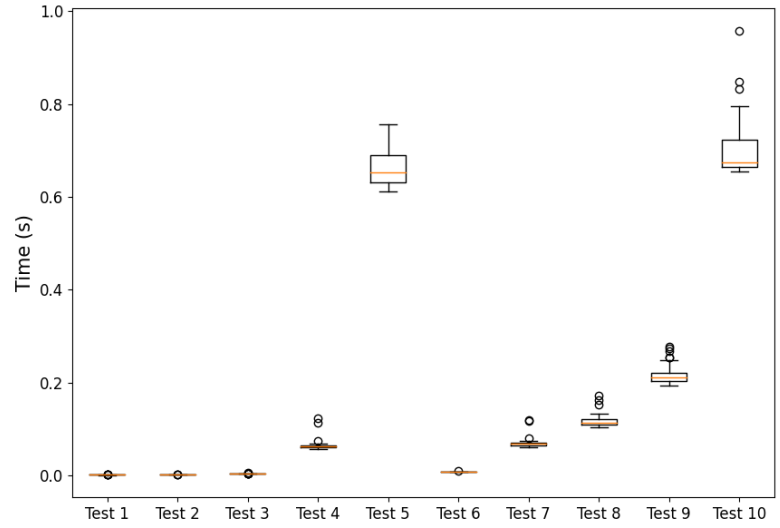


Figure 5.8: Graph representing the time it takes to create different sizes of product graphs

Figure 5.8 shows the time it takes to create the product graph for different environments and task sizes. Most of the graph is in line with the expectation, the bigger the size of the product graph, the longer it takes to generate. All product graphs were created in under one second, so for an HLM consisting of 68 subtasks, we are able to generate a product graph for tasks of size (64 x 729) in a reasonable amount of time. Interesting is Test 5, Test 5 has roughly the same product graph size as Test 9 but it takes about twice as long to create this product graph. This indicates that task size has the most influence on the time it takes to create the product graph. If we compare Test 4 and Test 7, we see the same behavior. Test 4 and Test 7 take roughly the same time to create the product graph, but the product graph of Test 7 (227 x 796) is about 1.8 times the size of Test 4 (143 x 422). Test 4 does however have a bigger task size (32 x 243) compared to Test 7 (16 x 81), so again the task size seems the most influential factor for the time it takes to create the product graph.

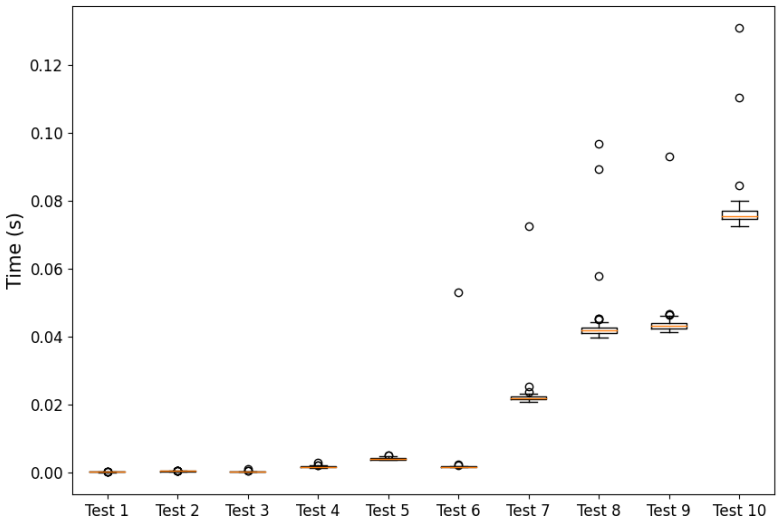


Figure 5.9: Graph representing the time it takes to find all Pareto optimal paths in the product graph

Figure 5.9 shows the time it takes to find all Pareto optimal paths in product graphs of different sizes. There is a clear relation between the size of the product graph and the time it takes to find all paths. We are able to find all Pareto optimal paths in less than 0.15 seconds for every tested product graph size. Interesting is that Test 7 takes longer than Test 5, even though Test 5 has a bigger product graph. A possible explanation for this is the number of Pareto optimal paths in the product graph, if we look at Table 5.7 we can see that test 7 has fourteen Pareto optimal paths, while test 5 only has two. Test 8 and test 5 also show this, the product graph of test 5 is larger than the product graph of test 8, but test 8 requires more time to find all Pareto optimal paths since test 8 has thirteen Pareto optimal paths and test 5 has only two.

Test	Paths found
Test 1	1
Test 2	3
Test 3	1
Test 4	2
Test 5	2
Test 6	4
Test 7	14
Test 8	13
Test 9	11
Test 10	10

Table 5.7: Number of Pareto optimal paths found for each test

PAC-guarantees
In order to provide guarantees for the entire controller we calculate PAC-guarantees. PAC-guarantees provide an upper bound on the probability that the expected performance is within a certain error margin of the estimated performance, see section 3.4.

We calculate the PAC-guarantees for all the tasks used for the comparison and evaluating the use of multi-objective planning. Table 5.8 provides an overview of these tasks and the environment they are used in (either four rooms, Figure 5.2a or nine rooms, Figure 5.6). For every task, we chose the sequence of subtasks that results in both a good success probability as well as a good cost. For test 1 and test 6, we use the sequence of subtasks with the second highest estimated success probability since these tasks are choice tasks, and using the path with the second highest success probability results in a substantial improvement in cost. For all other tests, we use the path with the highest success probability, since these tasks do not gain a big cost improvement from using a path with a lower estimated success probability.

Test	Task	Environment
Test 1	$\Diamond g \vee \Diamond P$	Four rooms
Test 2	$\Diamond(g \wedge \Diamond p)$	Four rooms
Test 3	$\Diamond g$	Four rooms
Test 4	$\Diamond g \wedge \Box\neg r3$	Four rooms
Test 5	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	Four rooms
Test 6	$\Diamond\ g \vee \Diamond\ y$	Nine rooms
Test 7	$\Diamond(\neg R4\ U\ p \wedge \neg R7\ U\ g)$	Nine rooms
Test 8	$\Diamond y \wedge \Diamond c \wedge \Diamond p \wedge \Diamond g$	Nine rooms

Table 5.8: Tasks and environments used for calculating PAC-guarantees

To calculate the PAC-guarantees for the success probability and the cost of each task we use the following equation, which can also be found in section 2.8:

$$\mathbb{P}(|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}[Z]| \geq \epsilon) \leq \delta = 2e^{-\frac{2n\epsilon^2}{(b-a)^2}} \tag{5.1}$$

Where:

- n: Number of samples used
- $Z_i, ..., Z_n$: Bounded random variables with $Z_i \in [a, b]$ for all i and $-\infty < a \leq b \leq \infty$
- a: lower bound on the values the random variables can take
- b: upper bound on the values the random variables can take
- $\mathbb{E}[Z]$: expected mean
- ϵ: Error margin between the expected mean and the estimated mean
- δ: The bound on the probability that the difference between the expected mean and the estimated mean is bigger than ϵ

To calculate the PAC-guarantees for success probability, the lower bound (a) and upper bound (b) are 0 and 1 since when a simulation is successful, it results in 1 and if a simulation fails the task, it results into a 0. We use a total of 600 simulations in order to estimate the success probability of a controller, so $n = 600$, for the error margin (e) we take the value of 0.05, so the error margin between the expected success probability and estimated success probability is at most 5%. These values result in a δ of 0.1, see Equation 5.2.

$$\mathbb{P}(|\frac{1}{n}\sum_{i=1}^{n} Z_i - \mathbb{E}[Z_i]| \geq \epsilon) \leq \delta = 2e^{-\frac{2\cdot 600\cdot 0.05^2}{(1-0)^2}} = 0.1 \tag{5.2}$$

So there is an upper bound (δ) of 0.1 (10%) on the probability that the error between the estimated success probability and expected success probability is greater than 0.05. This holds for the success probability of every task since the upper bound, lower bound, the number of simulations and the error margin are always the same. In Table 5.9 we present the estimated success probability of each test, the lower and upper bound for the expected success probability, and the probability the expected success probability is actually within these two bounds.

Test	**Estimated success probability ($\mathbb{P}_t$)**	**Lower bound success probability ($\mathbb{P}_t - \epsilon$)**	**Upper bound success probability ($\mathbb{P}_t + \epsilon$)**	**Probability expected success probability is within bounds ($1 - \delta$)**
Test 1	0.81	0.76	0.86	0.9
Test 2	0.63	0.58	0.68	0.9
Test 3	0.84	0.79	0.79	0.9
Test 4	0.79	0.74	0.84	0.9
Test 5	0.39	0.34	0.44	0.9
Test 6	0.77	0.72	0.83	0.9
Test 7	0.53	0.48	0.58	0.9
Test 8	0.61	0.56	0.66	0.9

Table 5.9: Estimated success probability for different tasks and their PAC-guarantees

The PAC-guarantees give us a 90% confidence that the expected success probability is within the range given in the third and fourth column in Table 5.9.

For the cost, we take the minimal amount of steps it takes to solve the task when there are no moving obstacles in the environment as the lower bound (a) since it is impossible to have a path with a lower number of steps than the minimum cost path when there are no moving obstacles. For the upper bound we set the value to $3 \cdot lower\ bound$. The reason we chose $3 \cdot lowerbound$ for the upper bound is that if we chose a lower value ($2 \cdot lower\ bound$), multiple simulations would fail due to reaching the upper bound on the cost, and therefore lower the success probability. Using a higher value would weaken the PAC-guarantees we can provide. Table 5.10 shows the three upper bounds tested and the number of simulations that failed due to reaching this upper bound.

Test	**Simulations reaching $2 \cdot lower\ bound$ upper bound**	**Simulations reaching $3 \cdot lower\ bound$ upper bound**	**Simulations reaching $4 \cdot lower\ bound$ upper bound**
Test 1	2	1	0
Test 2	0	0	0
Test 3	0	0	0
Test 4	5	1	0
Test 5	0	0	0
Test 6	35	5	4
Test 7	0	1	0
Test 8	27	6	4

Table 5.10: Number of simulations that fail due to reaching the upper bound limit on the allowed number of steps

If we would use $2 \cdot lower\ bound$ as the upper bound, test 6 and test 8 have a substantial number of tasks failing due to reaching the upper bound. The difference between using an upper bound of $3 \cdot lower\ bound$ and $4 \cdot lower\ bound$ is negligible if we look at the number of failed simulations, so $3 \cdot lower\ bound$ was chosen as upper bound.

The error margin (ϵ) for each task is $0.05 \cdot upper\ bound$ since this is also the same error margin percentage-wise used for the success probability ($0.05 \cdot 1 = 0.05$). Table 5.11 shows the cost lower bound, upper bound, and error margin used for each test. For the number of samples (n) we use all successful runs from the 600 simulations used to calculate the success probability PAC-guarantees, so if we have 500 successful simulations out of all 600 runs, then $n = 500$.

Test	**Lower bound cost**	**Upper bound cost**	**Error margin**
Test 1	16	48	2.4
Test 2	42	126	6.3
Test 3	24	72	3.6
Test 4	24	72	3.6
Test 5	84	252	12.6
Test 6	26	78	3.9
Test 7	63	189	9.45
Test 8	92	276	13.8

Table 5.11: Lower bound, upper bound and error margin for the cost used to calculate the PAC-guarantees

Table 5.12 shows the calculated PAC-guarantees for all the tests. We are able to give $\geq 90\%$ confidence that the expected average cost is within the range given in the fourth and fifth column for 7 out of 8 tests. Only in test 5 did we gain a confidence of $< 90\%$, namely, 84%. This is because test 5 has the lowest amount of successful runs, weakening the bound we can provide.

Test	Number of samples (n)	Estimated cost (C_t)	Lower bound cost ($C_t - \epsilon$)	Upper bound cost ($C_t + \epsilon$)	Probability expected average cost is within bounds ($1 - \delta$)
Test 1	459	18.2	15.8	20.6	0.99
Test 2	385	46.4	40.1	52.7	0.97
Test 3	517	26.7	23.1	30.3	0.99
Test 4	502	26.5	22.9	30.1	0.99
Test 5	222	94.5	81.9	107.1	0.84
Test 6	457	47.7	43.8	51.6	0.99
Test 7	306	84	74.6	93.5	0.94
Test 8	337	143	129.2	156.8	0.96

Table 5.12: Estimated cost for different tasks and their PAC-guarantees

Summary of performance
We can find multiple Pareto optimal paths (sequence of subtasks) in the product graph. For the choice-type tasks, the spread in success probability and cost is the largest. There is a difference between the estimated performance of a path and the measured performance of a path, the largest difference between the estimated and measured success probability is 5% and the largest difference between the estimated and the measured cost is 5.5. When two Pareto optimal points are close to each other, one point may get dominated when we look at the measured value, while deemed Pareto optimal during planning.

We generate a product graph from an HLM existing out of 68 subtasks and LTL-tasks, whose Büchi automata exist out of 64 states and 729 edges in under a second. The size of the Büchi automata seems to have the biggest influence on the time it takes to create the product graph. For product graphs existing out of 885 states and 3118 edges and smaller, we find all Pareto optimal paths in under 0.15s, next to the size of the product graph the number of Pareto optimal paths has an influence on the time it takes to find all paths.

For the success probability, we provide PAC-guarantees $\geq 90\%$ that the expected success probability is within 0.05 of the measured success probability. For the cost we provide PAC-guarantees of $\geq 90\%$ that the expected average cost is within $0.05 \cdot 3 \cdot minimum\ cost$ of the measured cost, except for one task, for which the number of samples was too low and we only achieved a PAC-guarantee of 84%.

5.2.3. Insights gained from evaluation

From the evaluation, we gain a number of insights on the use of MOPRL and when it can be beneficial to use.

Insight 1: MOPRL outperforms other approaches in learning LTL-tasks in discrete stochastic environments with sparse rewards

We outperform 2 other approaches in five different tasks in a discrete stochastic environment, being able to solve more difficult tasks and have a better success probability for all tasks while the cost is only slightly worse compared to the other approaches.

Insight 2: The size of subtasks is important for the performance of MOPRL

Using smaller subtasks with a small training budget outperforms using big subtasks with a large training budget in environments with sparse rewards, having both a higher success probability as well as a lower cost.

Insight 3: Using multi-objective planning to find a sequence of subtasks is the most beneficial in choice tasks

When using multi-objective planning to find Pareto optimal sequences of subtasks it is best if all points on the Pareto front are spread out. When two points are close to each other it is possible that during planning one point is seen as Pareto optimal, but during simulation is dominated by another point near it. Choice tasks have the most spread out Pareto front.

Insight 4: MOPRL is able to find a sequence of subtasks for models with 68 subtasks and large LTL-tasks within seconds

MOPRL is able to create a product graph and find all Pareto optimal paths within seconds for an HLM consisting of 68 subtasks and LTL-tasks consisting of 64 states and 729 edges or smaller. The size of the LTL-task is the most influential on the time it takes to create the product graph. When there are more Pareto optimal paths in a product graph, it takes longer to find all those paths compared to product graphs of the same size with fewer paths.

Insight 5: MOPRL provides PAC-guarantees, which gives a confidence of at least 90% that the expected success probability is within a certain error margin of the estimated success probability and a confidence greater or equal to 80% for the cost

We simulate the entire controller to calculate a success probability and average cost and use PAC-guarantees to provide a bound on the probability that the expected success probability and expected average cost are within a certain error bound of the calculated values. These bounds are $\geq 90\%$ for the success probability with an error margin of 5% for all tested tasks and $\geq 80\%$ for the cost with an error margin of $0.05 \cdot 3 \cdot best\ possible\ cost$.

6

Discussion

This chapter discusses the approach used and the results of this research. We discuss the advantages and limitations of using MOPRL and also critically look at the evaluation.

6.1. Advantages and limitations of MOPRL

6.1.1. Advantages

The first advantage of MOPRL is that it decouples learning and planning in two stages. This makes it possible to change the learning algorithm, so it can be applied in other environments that benefit from using a different learning method than the one used in this research. It is also possible to use a different planning method, which may be desired if it is not sequences of subtasks optimized for multiple objectives that need to be found but instead the task needs to be solved in another context.

A second advantage is that MOPRL uses subtasks that are created independently of the LTL-task in order to create a controller. This makes it possible to solve different LTL-tasks without the need to retrain agents for subtasks and only requires re-planning to find a new sequence of subtasks that can complete the LTL-task, given that the LTL-task can be solved by a sequence of the already defined subtasks. This is beneficial in an environment where a controller needs to be able to do many different tasks.

A third advantage is that MOPRL optimizes for multiple objectives, in some environments, there are more objectives than just reward, success probability, and/or cost, using MOPRL accounts for these circumstances.

6.1.2. Limitations

The first limitation of using MOPRL is that it is very tailored toward the environment used in this research, the rooms make it easy to design subtasks and it is easy to see what good subtasks are. It is difficult to say how MOPRL will perform in environments with different characteristics.

A second limitation is that MOPRL requires manually defined subtasks. In some environments, it may not be possible or difficult to determine subtasks that can efficiently be learned by RL, adding an extra engineering step to applying MOPRL.

A third limitation is that MOPRL prunes the solution space. Every subtask is solved locally, which means it may be possible that optimal solutions may be pruned. One step that prunes the solution space is that we exclude subtasks that potentially may violate the global LTL-task when creating the product graph, by removing these subtasks, a large part of the solution space is removed.

6.2. Shortcomings evaluation

For the comparison, it can be argued that the comparison between MOPRL, DiRL, and SPECTRL is not executed optimally since DiRL and SPECTRL are originally not designed for the type of environment used in this research, and adjustments needed to be made in order for them to be applicable in discrete environments. It may be possible with some extra engineering to boost the performance of DiRL or SPECTRL in discrete stochastic environments with sparse rewards.

For the scalability of planning, only two different environment sizes are used, to gain a better insight into the influence of the size of the HLM it would be better to have used more HLMs with different sizes.

The PAC-guarantees provide a bound on the probability that the expected success probability and expected average cost are within a certain error margin of the estimated values. However, we are only able to provide a 90% confidence for the success probability, and for the cost, there is one measurement that only gives an 84% confidence, in some settings a higher confidence may be required, and a different method of calculating the confidence should be used or more simulations should be run to gain a higher confidence.

7

Conclusion and future work

7.1. Conclusion

Sequential decision-making problems are problems where the goal is to find a sequence of actions that complete a task in an environment. A particularly difficult type of sequential decision-making problem to solve is one in which the environment has sparse rewards, a large state space, and where the goal is to complete a complex task. This thesis aims to create a controller that can be used to solve these types of environments in cases where the task needs to be optimized for multiple objectives. We create MOPRL, an approach that combines techniques from planning, formal methods, and reinforcement learning to synthesize such a controller. From formal methods, we use LTL to formally define complex tasks. We use the idea of policy sketches and manually define subtasks in the environment, use reinforcement learning to learn a policy for each of these subtasks, and combine these subtasks together in a graph called a higher-level model. The higher-level model is combined with the LTL-task into a new graph, called the product graph. Afterward, Martins algorithm is used to find all sequences of subtasks in the product graph that complete the LTL-task and have a Pareto optimal success probability and cost. Finally, we provide PAC-guarantees on the performance of the entire controller.

MOPRL is able to outperform two other state-of-the-art approaches that aim to solve complex tasks when applied in a stochastic discrete environment with sparse rewards and large state space by achieving a higher success probability and only a marginal drop in cost. MOPRL is also able to complete difficult tasks, in which the other two approaches fail. We are able to find multiple solutions that optimize for success probability and cost. Sometimes, a solution has a marginally lower success probability but a big improvement in cost compared to the solution with the highest success probability. When only optimizing for success probability, this solution may not be found. The planning procedure of our approach scales well to bigger tasks, being able to find solutions for large tasks in a matter of seconds.

An advantage of MOPRL is that it separates learning and planning, enabling one to easily switch to a different learning method or planning algorithm. It also allows completing multiple different tasks without the need of relearning subtasks and only needing to replan the sequence of subtasks that need to be completed. Limitations of MOPRL are that, first, it is very tailored toward the environment used in this research, meaning it might not be applicable to different types of environments. Second, MOPRL also requires the subtasks to be defined manually, in some environments it may be difficult to determine what good subtasks would be. Lastly, MOPRL prunes the solution space, removing possible solutions from being explored.

7.2. Future work

There are many interesting topics that can be researched in the future, we highlight some of the general research topics that can be of interest as well as some suggestions for direct improvements to the approach used in this research.

Both the reinforcement learning as well as the planning step have room for improvement, reducing

the time it takes to generate the controller. The subtask agents can be trained in parallel and the planning procedure copies data structures in some parts, which possibly can be removed, improving execution time.

Another point for further research is to generalize the implementation more. Currently, the implementation is very tailored towards the current environment. The implementation should be made more general so it can easily be used in other environments.

The approach used in this research can be combined with ideas from other approaches. For example, to deal with stochasticity we only use subtasks that are guaranteed to never violate the LTL, it may be beneficial to use a reward machine and train agents for the subtask with the global task in mind.

Research the use of multi-objective planning, it seems that multi-objective planning has the most benefits of being used for choice-type tasks, but this should be researched more thoroughly to see what type of planning would be the best to use in which instances.

A more general topic from which our approach, but also other types of research can benefit is researching what feasible subtasks are that can be learned by reinforcement learning. If there is some way to automatically find feasible subtasks in an environment, it removes one of the downsides of using this approach, needing to manually supply the subtasks. Other hierarchical solutions can also benefit from finding feasible subtasks.

A

Evaluation data comparison

A.1. Average of all experiments

Method	Task	Best Proba-bility	Cost	Probability	Highest Cost
SPECTRL	$\Diamond g \vee \Diamond P$	0.70	23.30		
	$\Diamond(g \wedge \Diamond p)$	0.00	0.00		
	$\Diamond g$	0.56	36.90		
	$\Diamond g \wedge \Box\neg r3$	0.00	0.00		
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.00	0.00		
DiRL	$\Diamond g \vee \Diamond P$	0.65	17.30		
	$\Diamond(g \wedge \Diamond p)$	0.37	43.60		
	$\Diamond g$	0.57	24.15		
	$\Diamond g \wedge \Box\neg r3$	0.54	24.80		
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.00	0.00		
OM - small subtasks	$\Diamond g \vee \Diamond P$	0.87	26.13	0. 82	18.43
	$\Diamond(g \wedge \Diamond p)$	0.65	46.53		
	$\Diamond g$	0.87	25.93		
	$\Diamond g \wedge \Box\neg r3$	0.82	29.07		
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.41	96.27		
OM - big subtasks	$\Diamond g \vee \Diamond P$	0.72	19.27		
	$\Diamond(g \wedge \Diamond p)$	0.31	58.26		
	$\Diamond g$	0.62	29.29		
	$\Diamond g \wedge \Box\neg r3$	0.56	37.58		
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.13	119.50		

A.2. Experiment 1

Method	Task	Best Probability	Cost	Probability	Highest Cost	# of sub-tasks	Steps per Subtask	Total steps
SPECTRL	$\Diamond g \vee \Diamond P$	0.7	24.9			1	1000000	1000000
	$\Diamond(g \wedge \Diamond p)$	0	0			1	1000000	1000000
	$\Diamond g$	0.56	36.4			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0	0			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			1	1000000	1000000
DiRL	$\Diamond g \vee \Diamond P$	0.65	16.3			2	500000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.39	44			2	500000	1000000
	$\Diamond g$	0.53	24			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.54	24.8			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			4	250000	1000000
OM - small subtasks	$\Diamond g \vee \Diamond P$	0.86	26	0.85	19	20	50000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.63	46.5			20	50000	1000000
	$\Diamond g$	0.85	25.9			20	50000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.79	33.4			20	50000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.37	99.8			20	50000	1000000
OM - big subtasks	$\Diamond g \vee \Diamond P$	0.77	21			6	166666	999996
	$\Diamond(g \wedge \Diamond p)$	0.4	57.8			6	166666	999996
	$\Diamond g$	0.65	32			6	166666	999996
	$\Diamond g \wedge \Box\neg r3$	0.57	43.6			6	166666	999996
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.19	116.8			6	166666	999996

A.3. Experiment 2

Method	Task	Best Probability	Cost	Probability	Highest Cost	# of sub-tasks	Steps per Subtask	Total steps
SPECTRL	$\Diamond g \vee \Diamond P$	0.69	21.7			1	1000000	1000000
	$\Diamond(g \wedge \Diamond p)$	0	0			1	1000000	1000000
	$\Diamond g$	0.56	37.4			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0	0			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			1	1000000	1000000
DiRL	$\Diamond g \vee \Diamond P$	0.64	18.3			2	500000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.35	43.2			2	500000	1000000
	$\Diamond g$	0.61	24.3			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.54	24.8			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			4	250000	1000000
OM - small subtasks	$\Diamond g \vee \Diamond P$	0.89	26.4	0.81	18	20	50000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.66	46.7			20	50000	1000000
	$\Diamond g$	0.87	26.3			20	50000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.85	27			20	50000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.45	94			20	50000	1000000
OM - big subtasks	$\Diamond g \vee \Diamond P$	0.78	20.6			6	166666	999996
	$\Diamond(g \wedge \Diamond p)$	0.39	63.4			6	166666	999996
	$\Diamond g$	0.61	27.7			6	166666	999996
	$\Diamond g \wedge \Box\neg r3$	0.54	43.4			6	166666	999996
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.19	122.2			6	166666	999996

A.4. Experiment 3

Method	Task	Best Probability	Cost	Probability	Highest Cost	# of sub-tasks	Steps per Subtask	Total steps
SPECTRL	$\Diamond g \vee \Diamond P$	0.69	21.3			1	1000000	1000000
	$\Diamond(g \wedge \Diamond p)$	0	0			1	1000000	1000000
	$\Diamond g$	0.54	35.7			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0	0			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			1	1000000	1000000
DiRL	$\Diamond g \vee \Diamond P$	0.6	16.2			2	500000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.15	53.59			2	500000	1000000
	$\Diamond g$	0.59	28.18			1	1000000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.58	25.74			1	1000000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0	0			4	250000	1000000
OM - small subtasks	$\Diamond g \vee \Diamond P$	0.85	26	0.82	18.3	20	50000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.65	46.4			20	50000	1000000
	$\Diamond g$	0.9	25.6			20	50000	1000000
	$\Diamond g \wedge \Box\neg r3$	0.82	26.8			20	50000	1000000
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.4	95			20	50000	1000000
OM - big subtasks	$\Diamond g \vee \Diamond P$	0.74	32.6			20	50000	1000000
	$\Diamond(g \wedge \Diamond p)$	0.8	21.2			6	166666	999996
	$\Diamond g$	0.42	62			6	166666	999996
	$\Diamond g \wedge \Box\neg r3$	0.54	45			6	166666	999996
	$\Diamond(g \wedge \Diamond(p \wedge \Diamond(g \wedge \Diamond s)))$	0.62	44.8			6	166666	999996
		0.18	131			6	166666	999996

Bibliography

[1] P. Harmo, T. Taipalus, J. Knuuttila, J. Vallet, and A. Halme, “Needs and solutions - home automation and service robots for the elderly and disabled,” in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 3201–3206. DOI: 10.1109/IROS.2005.1545387.

[2] J. Hollingum, “Robots in agriculture,” *Industrial Robot: An International Journal*, 1999.

[3] J. Lowenberg-DeBoer, I. Y. Huang, V. Grigoriadis, and S. Blackmore, “Economics of robots and automation in field crop production,” *Precision Agriculture*, vol. 21, no. 2, pp. 278–299, 2020.

[4] P. Vähä, T. Heikkilä, P. Kilpeläinen, M. Järviluoma, and E. Gambao, “Extending automation of building construction — survey on potential sensor technologies and robotic applications,” *Automation in Construction*, vol. 36, pp. 168–178, 2013, ISSN: 0926-5805. DOI: https://doi.org/10.1016/j.autcon.2013.08.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0926580513001167.

[5] C. Balaguer and M. Abderrahim, *Robotics and automation in construction*. BoD–Books on Demand, 2008.

[6] V. Azimirad, M. T. Ramezanlou, S. V. Sotubadi, and F. Janabi-Sharifi, “A consecutive hybrid spiking-convolutional (chsc) neural controller for sequential decision making in robots,” *Neurocomputing*, vol. 490, pp. 319–336, 2022, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.11.097. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221018075.

[7] R. Pietrantuono and S. Russo, “Robotics software engineering and certification: Issues and challenges,” in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 308–312. DOI: 10.1109/ISSREW.2018.00023.

[8] X. Gandibleux, F. Beugnies, and S. Randriamasy, “Martins’ algorithm revisited for multi-objective shortest path problems with a maxmin cost function,” *4OR*, vol. 4, pp. 47–59, Mar. 2006. DOI: 10.1007/s10288-005-0074-x.

[9] J. Hare, “Dealing with sparse rewards in reinforcement learning,” *arXiv preprint arXiv:1910.09281*, 2019.

[10] A. Camacho, R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Ltl and beyond: Formal languages for reward function specification in reinforcement learning,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2019, pp. 6065–6073. DOI: 10.24963/ijcai.2019/840. [Online]. Available: https://doi.org/10.24963/ijcai.2019/840.

[11] R. Drechsler, *Formal system verification*. Springer, 2018.

[12] C.-I. Vasile, *Motion planning and control: A formal methods approach*, 2016. [Online]. Available: https://open.bu.edu/handle/2144/17081.

[13] K. Jothimurugan, S. Bansal, O. Bastani, and R. Alur, “Compositional reinforcement learning from logical specifications,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 10 026–10 039, 2021.

[14] X. Li, C.-I. Vasile, and C. Belta, *Reinforcement learning with temporal logic rewards*, 2016. DOI: 10.48550/ARXIV.1612.03471. [Online]. Available: https://arxiv.org/abs/1612.03471.

[15] P. Vaezipoor, A. C. Li, R. A. T. Icarte, and S. A. McIlraith, “Ltl2action: Generalizing ltl instructions for multi-task rl,” in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 10 497–10 508. [Online]. Available: https://proceedings.mlr.press/v139/vaezipoor21a.html.

[16] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Reward machines: Exploiting reward function structure in reinforcement learning,” *Journal of Artificial Intelligence Research*, vol. 73, pp. 173–208, Jan. 2022. DOI: 10.1613/jair.1.12440. [Online]. Available: https://doi.org/10.1613%2Fjair.1.12440.

[17] O. Nachum, S. S. Gu, H. Lee, and S. Levine, “Data-efficient hierarchical reinforcement learning,” *Advances in neural information processing systems*, vol. 31, 2018.

[18] A. Francis, A. Faust, H.-T. L. Chiang, *et al.*, *Long-range indoor navigation with prm-rl*, 2019. DOI: 10.48550/ARXIV.1902.09458. [Online]. Available: https://arxiv.org/abs/1902.09458.

[19] J. Wohlke, F. Schmitt, and H. van Hoof, “Hierarchies of planning and reinforcement learning for robot navigation,” in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, May 2021. DOI: 10.1109/icra48506.2021.9561151. [Online]. Available: https://doi.org/10.1109%2Ficra48506.2021.9561151.

[20] J. Andreas, D. Klein, and S. Levine, *Modular multitask reinforcement learning with policy sketches*, 2016. DOI: 10.48550/ARXIV.1611.01796. [Online]. Available: https://arxiv.org/abs/1611.01796.

[21] K. Jothimurugan, O. Bastani, and R. Alur, “Abstract value iteration for hierarchical reinforcement learning,” 2020. DOI: 10.48550/ARXIV.2010.15638. [Online]. Available: https://arxiv.org/abs/2010.15638.

[22] B. Eysenbach, R. Salakhutdinov, and S. Levine, *Search on the replay buffer: Bridging planning and reinforcement learning*, 2019. DOI: 10.48550/ARXIV.1906.05253. [Online]. Available: https://arxiv.org/abs/1906.05253.

[23] M. L. Littman, *Algorithms for sequential decision-making*. Brown University, 1996.

[24] P. Schillinger, M. Bürger, and D. V. Dimarogonas, “Multi-objective search for optimal multi-robot planning with finite ltl specifications and resource constraints,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 768–774.

[25] C. Neary, C. Verginis, M. Cubuktepe, and U. Topcu, “Verifiable and compositional reinforcement learning systems,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 32, 2022, pp. 615–623.

[26] D. A. Jack Clark. “Faulty reward functions in the wild.” (2016), [Online]. Available: https://openai.com/blog/faulty-reward-functions/ (visited on 09/30/2022).

[27] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, *Concrete problems in ai safety*, 2016. DOI: 10.48550/ARXIV.1606.06565. [Online]. Available: https://arxiv.org/abs/1606.06565.

[28] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, “Supplementary material for: Hierarchical reinforcement learning: A comprehensive survey,” *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–35, 2021.

[29] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender, “Complexity of finite-horizon markov decision process problems,” *J. ACM*, vol. 47, no. 4, pp. 681–720, Jul. 2000, ISSN: 0004-5411. DOI: 10.1145/347476.347480. [Online]. Available: https://doi.org/10.1145/347476.347480.

[30] M. Ghavamzadeh, S. Mannor, J. Pineau, A. Tamar, *et al.*, “Bayesian reinforcement learning: A survey,” *Foundations and Trends® in Machine Learning*, vol. 8, no. 5-6, pp. 359–483, 2015.

[31] M. Lahijanian, J. Wasniewski, S. B. Andersson, and C. Belta, “Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees,” in *2010 IEEE International Conference on Robotics and Automation*, 2010, pp. 3227–3232. DOI: 10.1109/ROBOT.2010.5509686.

[32] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, “Automated verification techniques for probabilistic systems,” in *Formal Methods for Eternal Networked Software Systems (SFM’11)*, M. Bernardo and V. Issarny, Eds., ser. LNCS, vol. 6659, Springer, 2011, pp. 53–113.

[33] R. J. Wilson, *Introduction to graph theory*. Pearson Education India, 1979.

[34] S. Arzaghi, “An introduction to linear temporal logic (ltl),” Feb. 2021.

[35] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, “Spot 2.0 — a framework for LTL and ω-automata manipulation,” in *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA’16)*, ser. Lecture Notes in Computer Science, vol. 9938, Springer, Oct. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8.

[36] M. Cai, H. Peng, Z. Li, and Z. Kan, “Learning-based probabilistic ltl motion planning with environment and motion uncertainties,” *IEEE Transactions on Automatic Control*, vol. 66, no. 5, pp. 2386–2392, 2021. DOI: 10.1109/TAC.2020.3006967.

[37] A. Camacho, J. A. Baier, C. Muise, and S. A. McIlraith, “Finite ltl synthesis as planning,” in *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[38] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[39] P. Hansen, “Bicriterion path problems,” in *Multiple Criteria Decision Making Theory and Application*, G. Fandel and T. Gal, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 109–127, ISBN: 978-3-642-48782-8.

[40] P. Halffmann, S. Ruzika, C. Thielen, and D. Willems, “A general approximation method for bicriteria minimization problems,” *Theoretical Computer Science*, vol. 695, pp. 1–15, 2017, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2017.07.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397517305418.

[41] A. Skriver and K. Andersen, “A label correcting approach for solving bicriterion shortest-path problems,” *Computers & Operations Research*, vol. 27, no. 6, pp. 507–524, 2000, ISSN: 0305-0548. DOI: https://doi.org/10.1016/S0305-0548(99)00037-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054899000374.

[42] E. Q. V. Martins, “On a multicriteria shortest path problem,” *European Journal of Operational Research*, vol. 16, no. 2, pp. 236–245, 1984.

[43] X. Gandibleux, F. Beugnies, and S. Randriamasy, “Martins’ algorithm revisited for multi-objective shortest path problems with a maxmin cost function,” *4OR*, vol. 4, pp. 47–59, Mar. 2006. DOI: 10.1007/s10288-005-0074-x.

[44] A. G. Barto and S. Mahadevan, “Recent advances in hierarchical reinforcement learning,” *Discrete Event Dynamic Systems*, vol. 13, no. 1, pp. 41–77, Jan. 2003, ISSN: 1573-7594. DOI: 10.1023/A:1022140919877. [Online]. Available: https://doi.org/10.1023/A:1022140919877.

[45] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[46] L. Faik, *Deep q network: Combining deep & reinforcement learning*, Mar. 2021. [Online]. Available: https://towardsdatascience.com/deep-q-network-combining-deep-reinforcement-learning-a5616bcfc207.

[47] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[48] T. M. Moerland, J. Broekens, and C. M. Jonker, “Model-based reinforcement learning: A survey,” *arXiv preprint arXiv:2006.16712*, 2020.

[49] V. V. PV, *Deep reinforcement learning: Value functions, dqn, actor-critic method, back-propagation through stochastic functions*, Aug. 2020. [Online]. Available: https://medium.com/@vishnuvijayanpv/deep-reinforcement-learning-value-functions-dqn-actor-critic-method-backpropagation-through-83a277d8c38d (visited on 08/15/2022).

[50] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963, ISSN: 01621459. [Online]. Available: http://www.jstor.org/stable/2282952 (visited on 12/20/2022).

[51] R. Castro, “Elen6887 lecture 7: Pac bounds and concentration of measure,” 2010.

[52] J. Duchi, *Cs229 supplemental lecture notes hoeffding’s inequality*, 2017.

[53] J. Domke, *Learning theory*, 2010.

[54] R. Toro Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, “Teaching multiple tasks to an rl agent using ltl,” in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 2018, pp. 452–461.

[55] R. S. Sutton, D. Precup, and S. Singh, “Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, vol. 112, no. 1, pp. 181–211, 1999, ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00052-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370299000521.

[56] P. Ashok, J. Kř etínský, and M. Weininger, “PAC statistical model checking for markov decision processes and stochastic games,” in *Computer Aided Verification*, Springer International Publishing, 2019, pp. 497–519. DOI: 10.1007/978-3-030-25540-4_29. [Online]. Available: https://doi.org/10.1007%2F978-3-030-25540-4_29.

[57] R. T. Icarte, T. Klassen, R. Valenzano, and S. McIlraith, “Using reward machines for high-level task specification and decomposition in reinforcement learning,” in *International Conference on Machine Learning*, PMLR, 2018, pp. 2107–2116.

[58] K. Jothimurugan, R. Alur, and O. Bastani, “A composable specification language for reinforcement learning tasks,” 2020. DOI: 10.48550/ARXIV.2008.09293. [Online]. Available: https://arxiv.org/abs/2008.09293.

[59] H. Charlesworth and G. Montana, “Plangan: Model-based planning with sparse rewards and multiple goals,” in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 8532–8542. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/6101903146e4bbf4999c449d78441606-Paper.pdf.

[60] M. Lahijanian, M. R. Maly, D. Fried, L. E. Kavraki, H. Kress-Gazit, and M. Y. Vardi, “Iterative temporal planning in uncertain environments with partial satisfaction guarantees,” *IEEE Transactions on Robotics*, vol. 32, no. 3, pp. 583–599, 2016. DOI: 10.1109/TRO.2016.2544339.

[61] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, “Hierarchical reinforcement learning: A comprehensive survey,” *ACM Comput. Surv.*, vol. 54, no. 5, Jun. 2021, ISSN: 0360-0300. DOI: 10.1145/3453160. [Online]. Available: https://doi.org/10.1145/3453160.

[62] M. Hasanbeig, D. Kroening, and A. Abate, “Deep reinforcement learning with temporal logics,” in *International Conference on Formal Modeling and Analysis of Timed Systems*, Springer, 2020, pp. 1–22.

[63] G. Brockman, V. Cheung, L. Pettersson, *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[64] M. Chevalier-Boisvert, L. Willems, and S. Pal, *Minimalistic gridworld environment for openai gym*, https://github.com/maximecb/gym-minigrid, 2018.

[65] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[66] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. DOI: 10.48550/ARXIV.1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347.