THESIS

SOLVING MDPS WITH THRESHOLDED LEXICOGRAPHIC ORDERING USING REINFORCEMENT LEARNING

Submitted by

Alperen Tercan

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2022

Master's Committee:

Advisor: Vinayak S. Prabhu
Co-Advisor: Charles W. Anderson

Edwin K. P. Chong

ABSTRACT

SOLVING MDPS WITH THRESHOLDED LEXICOGRAPHIC ORDERING USING REINFORCEMENT LEARNING

Multiobjective problems with a strict importance order over the objectives occur in many real-life scenarios. While Reinforcement Learning (RL) is a promising approach with a great potential to solve many real-life problems, the RL literature focuses primarily on single-objective tasks, and approaches that can directly address multiobjective with importance order have been scarce. The few proposed approach were noted to be heuristics without theoretical guarantees. However, we found that their practical applicability is very limited as they fail to find a good solution even in very common scenarios. In this work, we first investigate these shortcomings of the existing approaches and propose some solutions that could improve their practical performance. Finally, we propose a completely different approach based on policy optimization using our Lexicographic Projection Optimization (LPO) algorithm and show its performance on some benchmark problems.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Motivation

The need for multi-objective reinforcement learning (MORL) arises from many real-life scenarios and cannot be reduced to single-objective reinforcement learning tasks in general [1]. However, solving them requires overcoming certain inherent difficulties. The main challenge is that having multiple objectives gives rise to solutions that are not comparable without incorporating some user preference information. These are known as *Pareto Optimal* or non-inferior solutions forming a set of solutions where no solution is better than another in terms of all objectives. The user (or Decision Maker) needs to specify the preference to define the desired solution.

Various methods of specifying the user preference have been proposed and evaluated on three main fronts: (a) the expressive power of the specifications, (b) the intuitiveness of writing the specification, and (c) the availability of methods to solve problems with such specifications. For example, writing specifications that result in a partial order of solutions instead of total order makes the specification easier for the user but is not enough to describe a unique preference. Moreover, three main motivating scenarios differing on when the user preference becomes available or used have been studied in the literature. (1) User preference is known beforehand, and it is incorporated into the problem *a priori*. (2) User preference is used *a posteriori*, i.e., firstly a set of representative Pareto Optimal solutions, and the user preference is specified over them. (3) An interactive setting where the user preference is specified gradually during the search and the search is guided accordingly. The first scenario can be considered the most computationally efficient one as it allows narrowing down the search from the beginning. However, this is also the most challenging one for the users, as they need to write the specifications a priori, i.e. without seeing what possible solutions look like. Therefore, in this scenario, it is very important to have a user-friendly and intuitive specification method.

Arguably, the most common specification method for the a priori scenario is *Linear Scalarization* which requires the designer to assign weights to the objectives and make them comparable [2]. The main benefit of this technique is that it allows using many standard algorithms off the shelf as it preserves the additivity of the reward functions. However, expressing user preference with this technique requires significant domain knowledge and preliminary work in most scenarios. While it can be the preferred method when the objectives can be expressed in comparable quantities, e.g. when all objectives have a monetary value, this is not the case most of the time. Usually, the objectives are expressed in incomparable quantities like money, time, and carbon emissions. Furthermore, the simplicity of the specifications which allows using off-the-shelf algorithms limits its expressive power. Even if the user is able to determine an all-encompassing utility function, it is likely to be nonlinear. Trying to capture these nonlinear relations with a linear function limits the set of reachable solutions to a subset of Pareto optimal set.

To address these drawbacks of linear scalarization, several other approaches have been proposed and studied. Nonlinear scalarization methods like Chebyshev [3] are more expressive and can capture all of the solutions in the Pareto optimal set, however, they do not address the user-friendliness requirement. In this paper, we will focus on an alternative specification method that overcomes both limitations of linear scalarization, named *Thresholded Lexicographic Ordering* (TLO). In lexicographic ordering, the user determines an importance order for the objectives, and the less important objectives are only considered if two solutions are the same with respect to the more important objectives. The thresholding part of the technique allows a more generalized definition for being the same w.r.t. an objective. The user provides a threshold for each objective except the last and the objective values are clipped at the corresponding thresholds. This allows the user to specify values beyond which they are indifferent to the optimization of an objective. There is not a threshold for the last objective as it is considered an unconstrained, i.e. open-ended objective. This kind of strict importance order arises in many real-life scenarios. For example, inspired by [4], the autonomous urban driving task can be formulated into three objectives: safety, regulation, comfort. The most important objective is avoiding crashes or more generally minimiz-

ing the risk of crashes below a threshold. Unless this is ensured, cars should not consider things like being on the correct lane when turning. Finally, the comfort of the passenger should affect the decisions only if the first two objectives are satisfied. Note that this is not a simple constrained control task, as there exists an importance order among the constraints and there are many scenarios where constraints cannot be satisfied together. For example, a car needs to wait even if it has the right of way if another car failed to respect the right of way.

Despite the strengths of this specification method, the need for a specialized algorithm to use it in reinforcement learning (RL) has prevented it from being a common technique. The *Thresholded Lexicographic Q-Learning* algorithm was proposed as such an algorithm and has been studied and used in several papers. While it has been noted that this algorithm does not enjoy the convergence guarantees of its origin algorithm Q-Learning, we found that its practical use is limited to an extent that has not been discussed in the literature before. In this paper, we firstly discuss the common modes of failures with this algorithm which we believe is important for future work on this topic. Then, we discuss some Q-Learning variants that can solve these specifications in certain types of problems. Then, we present a Policy Gradient algorithm as a general solution.

1.2 Related Work

Our work is part of the multi-objective RL literature ([5], [6], [7], [8]) and closely related to the works on Constrained Markov Decision Process (CMDPs). More specifically, our work tries to address the same problems with previous work on multi-objective tasks with lexicographic preference ordering while it uses a version of gradient manipulation that has been utilized in some multi-objective RL and CMDP papers. Below, we will first summarize some of the work on multi-objective RL with importance order, then we will share some other papers with different gradient manipulation approaches.

[9] was one of the first papers that investigate the use of RL in multi-objective tasks with preference ordering. After formulating a framework based on abstract dynamic programming (ADP) models, it proposes the use of RL and DP to solve the formulated decision problem. While the

paper introduces Thresholded Lexicographic Q-learning (TLQ) as an RL algorithm to solve the lexicographic ordering, it is not used in the experiments.

[10] empirically compares TLQ with scalarized Q-learning, which uses a weighted sum of the value-vector as its objective function. It analyzes their effectiveness in finding solutions from different parts of the Pareto front for a given task. This is used as a proxy for the performance of the two algorithms in addressing different user preferences. It compares the algorithms in three grid-world tasks with different Pareto front shapes. They show that TLQ significantly outperforms Linear Scalarization (LS) when the Pareto front is globally concave or most of the solutions lie on the concave parts. On the other hand, LS performs better when the rewards are not restricted to terminal states, because TLQ cannot account for the rewards already received earlier in the episode. Later, [11] generalizes this analysis by comparing more approaches using a unifying framework. To our knowledge, [10] is the only previous work that explicitly discusses the shortcomings of TLQ. However, we found that TLQ has other significant issues that occur even outside of the problematic cases they analyze.

[12] introduces a Multi-objective MDP variant called Lexicographic MDP (LMDP) and the corresponding Lexicographic Value Iteration (LVI) algorithm. LMDPs allow having different importance orders in different parts of the state space and defining the thresholds as slack variables which determines how worse than the optimal value is still sufficient. While [12] proves that the algorithm converges to a desired policy if slacks are chosen appropriately, such slacks are generally too tight to allow defining user preferences. This is also observed in [13] which claims that while ignoring the slack bounds proposed in [12] negates the theoretical guarantees, the resulting algorithm still can be useful in practice.

[4] investigates the use of Deep TLQ for urban driving. It shows that the TLQ version proposed in [9] introduces additional bias which is especially problematic in function approximation settings like deep learning. Also, it depends on learning the true Q function, which can not be guaranteed. To overcome these drawbacks, it uses slacks instead of static thresholds in [9] and proposes a different update function.

Finally, [14] uses TLQ in a multi-objective multi-agent setting and shows that this leads to an explosion of the number of thresholds to be set. Thus, they use a dynamic thresholding heuristic based on the previously achieved results during the training process, e.g. setting the threshold to 90% of the achieved highest value.

However, we discovered that these works on using a Q-learning variant with thresholded ordering perform very poorly in most cases due to non-Markovianity of the value function it tries to learn. It is possible to bypass this issue by using policy gradient approach as it does not require learning an optimal value function. In order to handle conflicting gradients, some modifications to the gradient descent algorithm are needed.

Recent work on modified gradient descent algorithms came mostly from Multi Task Learning literature, which could be considered a multiobjective optimization problem. [15] tries to find a common descent direction for all the objectives using their individual gradients. They show that such a direction can be found using the convex hull of the gradients. A significant drawback of the work in our setting is that it does not support user preferences, hence finds an arbitrary point on the Pareto front.

[16] applies [15]'s approach to multi-task learning with a focus on computational efficiency, especially in very high dimensional settings which are common in RL. However, it still finds only an arbitrary point on the Pareto front.

[17] divides the Pareto front into well-spaced regions and poses finding a solution in a given region as a subproblem. Solving all of the subproblems gives a set of well-distributed Pareto optimal solutions. Then, the user preference can be used to pick the best one from this set. While this method uses user preferences, it does not guarantee the exact preference but only gives an approximate solution. While the approximation quality can be increased by increasing the number of subproblems, the number of subproblems needed for the same level of approximation increases exponentially with the number of objectives.

[18] introduces the Exact Pareto Optimal (EPO) algorithm which can find the solutions on the Pareto front which also satisfies a preference-vector. While this technique can achieve any

preference-specific solution on the Pareto front exactly, determining such weights requires knowing the exact shape of the Pareto front and in-depth domain knowledge.

[19] is another work that uses policy gradient algorithms with modified gradients to learn the Pareto front. Their approach consists of two main components: a radial algorithm to find an unspecified single point on the Pareto front and a Pareto-following algorithm to discover the rest of the Pareto front.

[20] proposes the Conflict-Averse Gradient Descent (CAGrad) algorithm which chooses the update direction by maximizing the worst local improvement over objectives and proves its convergence guarantees.

While these papers use similar ideas with our work, their setting is different than ours as they do not have any explicit importance order. [21] has the most similar setting to ours in gradient-based algorithms. It considers a set of constraints with an unconstrained objective. Then, the gradient of the unconstrained objective is projected onto positive half-space of the active (violated) constraints and adds a correction step to improve the active constraints. When no valid projection is found, the worstly violated constraints are ignored until a valid projection exists. This is one of the main differences with our setting: As we have an explicit importance-order of the objectives, it is not acceptable to ignore a constraint without considering the importance order. Also, we project the gradients onto hypercones instead of hyperplanes, which is a cone with $\pi/2$ vertex angle. Thus, our algorithm allows varying degrees of conservative projections to prevent a decline in the constraints.

While there are many other recent works on CMDPs ([22], [23], [24]), they do not allow having an importance order over the constraint objectives. While the importance order is not needed when all of the constraints can be satisfied simultaneously, this may not be always the case. Therefore, their approaches are not applicable in our setting.

Finally, recently, using RL in tasks with lexicographic ordering has started to attract attention from other communities as well. A notable example is [25], which uses formal methods to construct single objective MDPs when all of the objectives are ω-regular. Their work is based on the previous work [26], which shows that an MDP with single ω-regular objective can be solved by

augmenting the MDP with an automaton that keeps track of the objective. The scalar reward values of this product MDP is constructed in a way to ensure that its solution optimizes the original objective. [25] extends this approach to multiple lexicographic objectives. It combines the automata representing these objectives through a reduction process and augment the original MDP with this newly constructed automaton.

1.3 Contributions

In this paper, we start by summarizing the literature on TLQ and show how the variants proposed in the literature relate to each other. Then, we demonstrate some important shortcomings of these variants using a simple control task (Section 3.4). Most significantly, we describe how they fail to learn reaching to a goal state even in scenarios common in real-life control tasks (Section 3.4.3). Moreover, we show how some of the most natural policies cannot be obtained with these methods when using the same threshold value in all states (Section 3.4.4).

We first share some of our failed attempts to address the issues with TLQ. "TL-SARSA" changes the update function used by TLQ to prevent it from making unrealistically optimistic updates. We explain why this cannot work due to the theoretical limitations of the underlying algorithms. Similarly, we show how "Cyclic Action Selection", a variant that changes how the actions are chosen and tries to prevent unnecessary sacrifices from the primary objective, cannot solve the problem.

Then, we propose two working solutions to solving LMDPs within the framework of value-function methods. Firstly, a TLQ-variant named "Informed Targets" is proposed to modify the update function in a way that better accounts for how action selection mechanisms in TLQ differ from Q-learning. Secondly, an MDP construction method that could solve tasks with two lexicographically ordered objectives that satisfy certain assumptions is described. This construction is based on the state augmentation idea and its validity is proven by showing its optimal policy is the same as the original task. Then, several methods are proposed to extend this method to the many-objectives case.

After completing our analysis of the use of value function based algorithms for TLO, we consider Policy Gradient algorithms. While a popular family of algorithms in RL in general, this family of algorithms has not been considered in the domain of tasks with lexicographic ordering. To our knowledge, our work is the first policy gradient algorithm for TLO.

We start developing our algorithm by creating a general optimization method for multi-objective problems with thresholded lexicographic ordering. The main building block of this algorithm is an iterative projection function that we created. This function iteratively projects gradients to prevent them from conflicting with previous objectives. While such an approach has been used in a few works before, we extend the previous approach by introducing a conservativeness hyperparameter that can be used to make the updates more risk-averse in terms of causing a decline in the previous objectives. This approach can be combined with many first-order optimization methods, although we demonstrate its performance with Gradient Descent.

Finally, we show how this general optimization algorithm can be adapted to RL by proposing a variant of the popular policy gradient algorithm REINFORCE, named Lexicographic REINFORCE.

1.4 Thesis Organisation

In Chapter 2, we start by introducing the Lexicographic Multiobjective MDPs that we use in the work (Section 2.1). Then, we give some general background on Reinforcement Learning methods (Section 2.2), organized under two subsections: value-function algorithms (Section 2.2.1) and policy gradient algorithms (Section 2.2.2).

In Chapter 3, we first analyze the existings value-function based approaches for solving LMDPs. Section 3.1-3.3 explains different variations of TLQ from the literature and Section 3.4 explains the shortcomings of the the existing approaches. Then, in Section 3.5, we introduce our solutions to these shortcomings and LMDPs in general within the framework of value-function algorithms.

In Chapter 4, we discuss our policy-gradient based approach for solving LMDPs. Section 4.1 introduces the necessary notation and mathematical background for this chapter. In Section 4.2, we

first develop a general gradient manipulation algorithm that can be used to solve multi-objective optimization problems with thresholded lexicographic ordering; then, we demonstrate its performance on a benchmark problem. In Section 4.3, we discuss how this approach can be used to adapt policy-gradient algorithms for single-objective MDPs to work in LDMPs and demonstrate its usefulness in several test cases. Finally, Chapter 5 gives a summary of our work.

Chapter 2

Background

2.1 Multiobjective MDP

Definition 1. *A **Multiobjective Markov Decision Process (MOMDP)** is represented by tuple* $\langle S, A, P, \mathbf{R}, \mathbf{\Gamma} \rangle$ *which consists of the following components:*

- S *is the finite set of states with initial state* $s_{init} \in S$ *and set of terminal states* S_F*,*
- A *is a finite set of* m *actions,*
- $P : S \times A \times S \to [0, 1]$ *is a state transition function given by* $P(s, a, s') = \mathbb{P}(s'|s, a)$ *is the probability of transitioning to state* s' *given that the current state is* s *and action* a *is taken.*
- $\mathbf{R} = [R_1, \ldots, R_K]^T$ *is a vector that specifies the reward of transitioning from state* s *to* s' *upon taking action* a *under* K *different reward functions* $R_i : S \times A \times S \to \mathbb{R}$ *for* $i \in \{1, \ldots, K\}$*. For the ease of notation, we will also denote the vector valued function which has* R_i *as individual components as* $\mathbf{R} : S \times A \times S \to \mathbb{R}^K$*.*
- $\mathbf{\Gamma} = \langle \gamma_1, \ldots, \gamma_K \rangle \in \mathbb{R}^K$ *is a tuple of discount factors for each objective.* γ_i *specifies the relative importance of immediate rewards over future rewards with smaller values meaning more importance on immediate rewards. In this work, we will assume that the discount factor is the same for all objectives and it is denoted by* γ*, i.e.* $\gamma_i = \gamma \; \forall i \in \{1, \ldots, K\}$*,*

In such a MOMDP, a *trajectory* $\zeta \in (S \times A)^* \times S$ is a sequence $\zeta = s_0 a_0 s_1 a_1 \ldots a_{T-1} s_T$ where $s_i \in S$, $a_i \in A$ and indices denote the time steps.

To simplify the discussions, MDPs are considered to consist of two components: *agent* and *environment*. The agent is a decision maker which decides what action will be taken at each step. Then, the environment uses this action with functions P and $\mathbf{R}$ to compute the next state of the MDP and the reward to be given.

In this paper, we will use the *episodic* case of MDP where the agent-environment interaction consists of sequences that start in s_{init} and terminates when a state in S_F is visited. Each of these sequences is called an *episode*. The length of the episode is finite but not a fixed number. In MDP literature, this is known as "indefinite-horizon" MDP. We will also assume that the trajectories do not span more than one episode.

We can define an objective $o_i \in \{1, \ldots, K\}$ as maximizing an optimality measure $G_i : Z \to \mathbb{R}$ over set of trajectories with respect to reward function R_i where where Z is the set of all trajectories. G_i is usually called *return* of the trajectory. The most common optimality measure in RL community is the expected discounted cumulative rewards, or formally, $\sum_{t=0}^{T-1} \gamma^t R_i(s_t, a_t, s_{t+1})$. Hence, we will limit our discussion to this as well. Together, every trajectory ζ is assigned a return vector $G(\zeta) = \langle G_1(\zeta), \ldots, G_K(\zeta) \rangle \in \mathbb{R}^K$.

However, since the MOMDP's transition function can be stochastic, the agent does not have control over which trajectory it will end up with. It can only control which action it will take; hence, talking about a *policy* function $\pi : S \times A \to [0, 1]$ that will assign probabilities to state-action pairs instead of individual trajectories is more useful. While a policy in general can be a function of history at time t, i.e., $H_t = A_0, S_1, \ldots, A_{t-1}, S_t$, in an MDP setting, we can find an equally good history-independent policy since both transition and reward functions history-independent [27]

Then, we can talk about a *value* function v^π which does not assign value to the trajectories but the policy that is used to generate them. Moreover, since the generated trajectories depend on the starting state besides the action probabilities, this value function needs to be dependent on the state too. Combining all of these requirements, we can define a value function $v^\pi : S \to \mathbb{R}^K$ such that $v^\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{T-1} \gamma^t R(s_t, a_t, s_{t+1}) | s_0 = s]$. Intuitively, $v^\pi(s)$ is the expected return from following policy π starting from state s. Note that v^π is also referred as *state-value* function as it assigns values to the states.

In order to define overall optimal behavior in a MOMDP, usually, a preference is defined by an order relation over the space of value vectors $v \in \mathbb{R}^K$. Then, we can define an optimal policy π^*

as the policy that is better than or equal to any other policy π. A policy π is defined to be better than or equal to π' if $v^{\pi}(s) \geq v^{\pi'}(s)\ \forall s \in S$.

A Lexicographic MDP (LMDP) is a MOMDP with another component:

- $\tau = \langle \tau_1, \ldots, \tau_{K-1} \rangle \in \mathbb{R}^{K-1}$ is a tuple of threshold values where τ_i indicates the minimum acceptable value for the objective i. The last objective does not require a threshold; hence, there are only $K-1$ values.

In an LMDP, the order over the value vectors uses the threshold values $\tau \in \mathbb{R}^{K-1}$. Then, two value vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^K$ can be compared via the thresholded lexicographic comparison relation $>^{\boldsymbol{\tau}}$ as $\mathbf{u} >^{\boldsymbol{\tau}} \mathbf{v}$ iff there exists $i \leq K$ such that:

- $\forall j < i$ we have $\mathbf{u_j} \geq \min(\mathbf{v_j}, \tau_j)$; and
  - if $i < K$ then $\min(\mathbf{u_i}, \tau_i) > \min(\mathbf{v_i}, \tau_i)$,
  - otherwise if $i = K$ then $\mathbf{u_i} > \mathbf{v_i}$.

Intuitively, we compare $\boldsymbol{u}$ and $\boldsymbol{v}$ starting from the most important objective ($j = 1$); the less important objectives are considered only if the order of higher priority objectives is respected. The relation $\geq^{\boldsymbol{\tau}}$ is defined as $>^{\boldsymbol{\tau}} \vee =$.

While an LMDP definition without the threshold values is possible, its solution is not as interesting. Since the value functions are real-valued, it is unlikely for value vectors of two different policies to have exactly the same value for an objective. Thus, in most cases, an LMDP without threshold values would have the same optimal policy with optimizing only the most important objective.

Note that this relation imposes a total order on the value functions in contrast to partial orders that are allowed by the general MOMDP setting. Therefore, we will assume a total order in this work as our setting is limited to LMDPs.

We should note that the "indefinite-horizon" formulation requires that all episodes must eventually terminate. We can show that our setting satisfies this by observing that it is equivalent to an

Absorbing Markov Chain [28]. A Markov Chain is a stochastic model that is similar to an MDP but that does not have rewards or actions. In fact, an MDP can be converted to a Markov Chain under a stationary policy by ignoring the reward function and combining the policy into the transition function as $P'(s, s') = \sum_{a \in A} P(s, a, s')\pi(s, a)$ where P' is the transition function of Markov Chain and others are defined as above [29]. An Absorbing Markov Chain is a Markov Chain where all walks on the chain are guaranteed to eventually reach an absorbing state that only loops to itself. A Markov Chain is absorbing if it has at least one absorbing state and there is a path from each non-absorbing state to an absorbing state [30]. A terminal state in an MDP can be interpreted as an absorbing state in a Markov Chain where it stays forever and all terminal states are reachable from all non-terminal states in the MDPs we will use in this work. The policies we consider also have non-zero probability for all actions in each state; hence, when the transition function of the corresponding Markov Chain is computed, it will be an Absorbing Markov Chain. The non-zero probability for each action assumption is satisfied through ϵ-greedy approach in Section 3, where ϵ-greedy refers to taking a greedy action with $1-\epsilon$ probability and taking a random action sampled uniformly from A with ϵ probability. That is the best action has $1 - \epsilon + \frac{\epsilon}{|A|}$ probability and the rest have $\frac{\epsilon}{|A|}$ probability each. In Section 4, we assume that this requirement is satisfied by the "structure" of the policy function. For example, the policy function we use in our experiment outputs a probability distribution over action set through a softmax function which results in probability $\frac{e^{f(a_i)}}{\sum_j e^{f(a_j)}}$ for action a_i where $f : A \to \mathbb{R}$. As $e^x > 0 \quad \forall x \in \mathbb{R}$, this function guarantees having non-zero probabilities for all actions by structure.

2.2 Reinforcement Learning

The main goal of RL is finding the optimal policy π^* as defined in the previous section. The learning process, commonly referred to *training*, starts with a suboptimal, usually randomly initialized, policy π and it is iteratively improved based on the experience obtained from interacting with the environment. This usually involves collecting trajectories following a certain behavior policy π^b and using it to improve π, where different algorithms are developed to deal with $\pi^b = \pi$

and $\pi^b \neq \pi$ cases separately. In the episodic case, the trajectory collection is organized as separate episodes which start from initial states and continue until some termination condition is satisfied.

The approaches for policy improvement are considered under two groups: Value-function based methods and Policy Gradient methods. In this work, we will consider approaches from both families, hence we will give some background on both first.

Note that another important distinction in RL is between model-based algorithms which require a model of the underlying MDP and model-free algorithms that do not need an explicit model. All of the algorithms considered in this work are model-free algorithms. See [31] for more information.

2.2.1 Value-function Algorithms

A common method for finding the optimal policy π^* is estimating the optimal value function v^* and constructing π^* using it. v^* is defined as:

$$v^*(s) \triangleq \max_{\pi \in \Pi} v^\pi(s) \quad \forall s \in S \tag{2.1}$$

where Π is the set of all possible policies.

If the transition function of the underlying MDP was known, we could use v^* to choose the action that would lead to the states with the highest v^*. However, in RL the transition function is usually not known.

Therefore, in general, we are interested in learning the *action-value* function $Q^\pi : S \times A \rightarrow \mathbb{R}^K$, $Q^\star$ to be specific. $Q^\pi(s,a)$ denotes the expected return from taking action a in state s and following the policy π after that.

$$Q^\pi(s,a) \triangleq \mathop{\mathbb{E}}_{\pi}[\sum_{t=0}^{T-1} \gamma^t \mathbf{R}(s_t, a_t, s_{t+1}) | s_0 = s, a_0 = a] \tag{2.2}$$

So, similar to Eq. 2.1, we can define $Q^\star$ as:

$$Q^\star(s,a) = \max_{\pi \in \Pi} Q^\pi(s,a) \quad \forall s \in S, \forall a \in A \tag{2.3}$$

Note that for LMDPs, like the comparison of state-value vectors, we use $\geq^{\tau}$ relation when comparing action-value vectors. Hence, the $\max$ operator in both of Eq. 2.1 and Eq. 2.3 use $\geq^{\tau}$ relation.

If we can obtain $Q^{\star}$, we can construct the optimal policy by always picking the best action in every state, that is:

$$\pi^{*}(s,a) = \begin{cases} 1, & a = \arg\max_{a' \in A} Q^{\star}(s,a') \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Note that this policy will never be directly used until the training is done as we will be using an ϵ-greedy version of it. Hence, this does not violate our all actions have non-zero probability assumption. This policy could be introduced to the set of allowed policies after it is learned completely as it can be shown that it reaches the terminal state always.

Now that we know what we need to find, we need to discuss how we can find it. In a single-objective MDP, $Q^{\star}$ needs to satisfy the following equation named Bellman Optimality Equation:

$$Q^{\star}(s,a) = \sum_{s' \in S} P(s,a,s')(R(s,a,s') + \gamma \max_{a' \in A} Q^{\star}(s',a')) \tag{2.5}$$

which can be also written as by replacing probability weighted sum with an expectation:

$$Q^{\star}(s,a) = \mathop{\mathbb{E}}_{s' \sim P}[(R(s,a,s') + \gamma \max_{a' \in A} Q^{\star}(s',a'))] \tag{2.6}$$

where $s' \sim P$ means that the probability distribution for s' is computed using $P(s,a,s')$. Intuitively, the value of an action under the optimal policy should be equal to expectation of a value that combines immediate reward with the discounted value of the optimal action in the next state.

This equation is actually a system of equations, one for each state-action pair and has a unique solution in MDPs with finite state and action spaces [31]. A common approach for solving this system is using dynamic programming. More specifically, starting from a random value for $Q^{\star}$ and

applying (2.5) iteratively results in the unique solution of the system which can be proven by the Banach Fixed Point theorem [32]. Since computing the expectation for each update is expensive, we can use the trajectories obtained from this MDP as an estimation of the true expectation. This gives the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)] \tag{2.7}$$

where (s, a, r, s') is a tuple that experienced from environment. In other words, the subsequence (s, a, s') occurs in a trajectory ζ that is obtained by following a π^b and $R(s, a, s') = r$.

Intuitively, this equation iteratively learns $Q^\star$ by combining the immediate reward with what it believes to be the best return obtainable from the next state. Note that this equation does not involve a policy function as it is an *off-policy* method, meaning the policy being learned is different than the policy used when interacting with the environment and we can learn $Q^\star$ while following any policy that satisfies that allows visiting all state-action pairs infinitely many times. This equation is the crux of *Q-learning* which is widely used for single objective tasks [33]. Trying to generalize it to LMDPs has been the inspiration for the algorithms we will discuss in Section 3.

Note that we will only consider deterministic stationary policies in Section 3 and overload notation as $\pi : S \to A$ to avoid introducing extra notation. We can limit ourselves to deterministic policies without loss of optimality as we are using the indefinite horizon setting with expected cumulative discounted rewards as our optimality measure. Note that this setting can be interpreted as an infinite horizon setting with terminal states replaced with absorbing states. Hence, there is always a deterministic stationary policy $\pi : S \to A$ that is optimal [34].

2.2.2 Policy Gradient Algorithms

Policy gradient algorithms in RL try to learn the policy directly instead of inferring it from the value functions. As the name suggests, they mostly consist of a method to compute or estimate the gradient of the optimality measure w.r.t. policy and a method to carefully update the policy under the light of this potentially imperfect information. We denote the policy parameterized by a vector

of variables, θ as π_θ. The performance of the policy π_θ is denoted as $J(\theta)$ and can be defined as the the expected return from following π_θ starting from s_{init}, i.e. $J(\theta) \triangleq v^{\pi_\theta}(s_{init})$.

Policy gradient algorithms have several advantages over the value-function based methods. Firstly, they allow the policy to get slowly deterministic as the training progresses by making the distribution over actions more deterministic. In contrast, value function based methods are inherently deterministic as they are "greedy" w.r.t. action-values and they rely on additional techniques like ϵ-exploration. Hence, value function based methods require some complex hyperparameter change scheme, such as reducing ϵ carefully during the training so that it is still large enough to explore the currently suboptimal looking actions but also small enough to be able to reach the high value states previously found [31].

Secondly, policy gradient algorithms enable learning policies with arbitrary probabilities for actions. Moreover, the policy could be a much simpler function to represent and learn compared to the action-value function, depending on the task.

Also, while not used in our work, some tasks require having a continuous, hence infinite action space. As the maximization over an infinite space at every step is not possible, most value function based algorithms cannot handle this case. On the other hand, policies can be defined to output a probability distribution over a continuous set, hence policy gradient algorithms naturally enable continuous action spaces.

On the more theoretical side, the policy functions used with policy gradient methods usually output the probabilities for the actions and these probabilities change smoothly with the changing parameters, there are no drastic changes in the policy from a small update to parameters. In contrast, an ϵ-greedy policy could drastically change with small changes in the action-value function if two different actions have close values.

This smooth change property allows stronger convergence properties for policy gradient algorithms (see section 13.2 in [31]). This means that the iterative training process RL is likely to reach an optimal policy. This also enables learning policies by mimicking gradient-based optimization algorithms. However, this operation is not as straightforward as other optimization tasks involving

function approximation, like regression because the performance metrics that we care about are affected by both the action selection and the frequency distribution of the visited states. While the relation between the policy parameters and action selection in a given state is easy to analyze and differentiate, the distribution of the visited states is a function of the policy and the transition function which are typically unknown and hence cannot be differentiated.

The crux of policy gradient algorithms, Policy Gradient Theorem ([35]), offers a closed-form formula for the gradient of the performance w.r.t. the policy parameters which does not require a derivative of the state distribution.

For our setting, i.e. episodic task with expected discounted cumulative reward as optimality measure, the theorem states:

$$\nabla_{(\theta)} J(\theta) \propto \sum_{s} \mu(s) \sum_{a} Q^{\pi_\theta}(s,a) \nabla_{(\theta)} \pi_\theta(a|s) \tag{2.8}$$

where

- $\propto$ means proportional to,
- $\nabla_{(\theta)} J(\theta)$ and $\nabla_{(\theta)} \pi_\theta$ is the gradient of $J(\theta)$ and π_θ at θ, respectively. The gradient of a multi-variable scalar-valued function $F : \mathbb{R}^n \to \mathbb{R}$ is a vector-valued function $\nabla F : \mathbb{R}^n \to \mathbb{R}^n$, where for any point $p \in \mathbb{R}^n$, components of $\nabla F(p)$ are the partial derivatives of F at p.
- $\mu(s)$ is the state distribution under the policy π_θ, i.e. the fraction of time spent in s following π_θ. This can be defined as the ratio between the expected number of time steps spent in s in a single episode and the expected length of a single episode.
- Q^{π_θ} is the action-value function under π_θ.

Intuitively, this means the gradient of the expected return is proportional to gradient of the policy weighted by the action-values. This makes sense as taking higher value actions in more frequently visited states should have more weight than less frequent states.In Section 4.3.1, we will give an example of how $\mu(s)$ and Q^{π_θ} can be estimated.

Once the gradient of the optimality measure w.r.t. the parameters of the policy function, we can use first-order optimization techniques like Gradient Ascent to maximize the optimality measure.

While all based on similar theoretical results, a myriad of policy gradient algorithms have been proposed in the literature [35] [36] [37] [38] [39]. These algorithms mostly differ on how the values that appear on the right-hand side of the equation can be estimated best. We will give an example of these algorithms in Section 4.

Chapter 3

Value Function Based Approaches

So far, the efforts to find exact solutions to the LMDPs has been focused on value-function methods, namely, different variants of Thresholded Lexicographic Q-Learning (TLQ) which is an LMDP adaptation of the Q-learning ([9] [4] [14] [12]). While these methods have been used and investigated in numerous papers, the extent of their weaknesses has not been discussed explicitly. In this work, we will start by introducing TLQ, then we will discuss some important failure scenarios. Then, we will propose some fixes for certain scenarios.

3.1 Thresholded Lexicographic Q-Learning

As mentioned in Section 2.2.1, adapting Eq. 2.7 to work with LMDPs has been the main motivation of the research on TLQ. The challenge is that when the agent has multiple objectives, it is unlikely that an action can be optimal for all of them. Hence, the actions taken when following the optimal policy will not have the maximum action-value at least with respect to some of the objectives. So, simply using the update rule in Eq. 2.7 for each objective and learning the optimal value function of each objective completely independent of the others will not work. This issue is further exacerbated when it is considered that the objectives need to be optimized only up to a threshold value.

Based on this observation, different techniques to allow taking the other objectives into account when computing the action-value functions and using them were developed. We consider all of these techniques under the umbrella of Thresholded Lexicographic Q-Learning (TLQ). As can be seen from the pseudocode in Algorithm 1, TLQ follows the same overall structure as Q-learning ([33]) and it differs in terms of the definition of the value function, and how action selection procedure and update procedure for the value function are defined. Different variations of the algorithm have been proposed in the literature through various instantiations of these components

in Algorithm 1. Note that Q function used in the algorithm is considered a generic one and can be defined in different ways as well.

Algorithm 1: Thresholded Lexicographic Q-Learning

Input: $\langle S, A, \mathbf{R}, \tau, \epsilon \rangle$

Result: Trained Q-function

Initialize action-value function Q with random weights

for *episode* $= 1, M$ **do**

  Initialize state s with initial state s_{init}

  while s *is not terminal* **do**

    $a \leftarrow \texttt{ActionSelection}(s, Q, \epsilon)$

    Take action a, observe state $s' \in S$ and reward vector $\mathbf{r} \in \mathbb{R}^K$

    $Q \leftarrow \text{UpdateRule}(Q, (s, a, s', \mathbf{r}))$

    $s \leftarrow s'$

return Q

We have attempted to modularize this algorithm and divide it into independent components to facilitate its analysis and future work on it; we will talk about them separately in the next sections. However, it should be noted that these components are inherently intertwined due to the nature of the problem. Most importantly, the value functions and acceptable policies are defined recursively where each of them uses the other's last level of recursion. So, due to these inherent circular referencing, the sections will not be completely independent and some combinations of introduced techniques will not work. In the next sections, we will explain which combination each paper proposes.

Another clarification is needed for π notation. As we noted in Section 2.2.1, we are considering ϵ-greedy policies in this section. However, as Q-learning computes the optimal policy directly, the acceptable policies used in the update rules and the greedy part of the action selection can be assumed deterministic. In Algorithm 1, `ActionSelection` function represents the policy function. Algorithm 2 shows that the greedy part of the policy, i.e. $r \geq \epsilon$ case, is deterministic. We can limit ourselves to deterministic policies without loss of optimality as we are using the

indefinite horizon setting with expected cumulative discounted rewards as our optimality measure. Note that this setting can be interpreted as an infinite horizon setting with terminal states replaced with absorbing states. Hence, there is always a deterministic stationary policy $\pi : S \to A$ that is optimal [34].

More intuitively, TLQ attempts to learn the optimal deterministic policy without actually ever using it. This is by the merit of being an *off-policy* algorithm, meaning the policy being learned is different than the policy used when interacting with the environment. See [31] for more details.

Therefore, for the ease of notation, we will overload our π notation as $\pi(s)$ denoting the action for which π assigns probability 1 in state s.

3.2 Value Functions and Update Rules

There are two variants of the TLQ in the literature, in terms of the definition of action-value functions and how they are learned. The starting point for both variants is $Q^\star = \langle Q_1^\star, \ldots, Q_K^\star \rangle$ where each $Q_i^\star : S \times A \to \mathbb{R}$ is defined as in Eq. 2.3 only with the change that the maximization is not done over the set of all policies Π but over a subset of it $\Pi_{i-1} \subseteq \Pi$ which will be defined in Section 3.3.

After this point, [9] proposes learning $\hat{Q}^\star : S \times A \to \mathbb{R}^K$ where each component of $\hat{Q}^\star$ denoted by $\hat{Q}_i^\star$ is defined as:

$$\hat{Q}_i^\star(s,a) \stackrel{\Delta}{=} \min(\tau_i, Q_i^\star(s,a)) \tag{3.1}$$

In other words, it is the rectified version of $Q^\star$. It is proposed to be learned using the following value iteration which is adapted from Eq. 2.5:

$$\hat{Q}_i^\star(s,a) := \min(\tau_i, \sum_{s'} P(s,a,s')(R_i(s,a,s') + \gamma \max_{\pi \in \Pi_{i-1}} \hat{Q}_i^\star(s', \pi(s')))) \tag{3.2}$$

Notice that similar to the definition of $Q_i^\star$, the main change during the adaptation is limiting the domain of $\max$ operator to Π_{i-1} from Π.

On the other hand, [4] shows that estimating $\hat{Q}^\star$ has certain drawbacks. Firstly, Eq. 3.2 is only an approximate value iteration when learning $\hat{Q}^\star$ because the threshold should be applied on $Q^\star$ not $\hat{Q}^\star$ as can be seen in Eq. 3.1. So, this value iteration can fail to learn the true $\hat{Q}^\star$. Moreover, they show that the $\min$ operator introduces bias when the update targets are computed where the update target refers to the value we will use to update the value of Q-function in value iteration. This means while we try to estimate the probability weighted sum in the update equation by the obtained samples, the expectation of the mean of obtained samples is not the same as the value we are trying to estimate. Therefore, it is proposed that we estimate $Q^\star$ instead and use it when the actions are being picked. More formally, we can use the following value iteration to estimate $Q^\star$:

$$Q_i^\star(s,a) = \sum_{s' \in S} P(s,a,s')(R_i(s,a,s') + \gamma \max_{\pi \in \Pi_{i-1}} Q_i^\star(s', \pi(s')) \tag{3.3}$$

Notice that the thresholding is incorporated in the set of policies the $\max$ is applied on.

3.3 Acceptable Policies and Action Selection

The second important part of TLQ is the definition of "Acceptable Policies", Π_i, which is likely to be different for each objective. These policies will be the ones that satisfy the requirements of the first i objectives. This definition will be used for both to decide the domain of $\max$ operator in the previous section and to choose the action taken using the `ActionSelection` function. The pseudocode of this function can be seen in Algorithm 2.

This algorithm starts with the set of all actions and iterates over the objectives while limiting the set of actions to the ones acceptable for all objectives up to that point. If branch in the for loop computes A_o from A_{o-1} using Q_o. `AcceptableActs` function is assumed to have the thresholding information embedded. The else branch handles the corner case where $o = K$ or A_{o-1} contains a single action. Firstly, if $o = K$, the algorithm should just pick the best action w.r.t. Q_o from A_{o-1}. If this branch was executed because A_{o-1} contains a single action, the $\arg\max$ is just a dummy operation that picks that single action. Note that `AcceptableActs` never returns an empty set; hence, A_o cannot be empty.

Algorithm 2: ActionSelection

Function `ActionSelection`$(s, Q, \epsilon|A)$**:**
  $r \sim U(0,1)$
  if $r < \epsilon$ **then**
    a is picked randomly from A
  else
    $A_0 \leftarrow A$
    for $o = 1, K$ **do**
      if $|A_{o-1}| > 1$ **and** $o < K$ **then**
        $A_o \leftarrow$ `AcceptableActs`(s, Q_o, A_{o-1})
      else
        $a \leftarrow \arg\max_{a' \in A_{o-1}} Q_o(s, a')$
        break
  return a

Note that the structure of this function is the same across different variations in the literature and different instantiations of the function differ in how the `AcceptableActs` subroutine is implemented. `AcceptableActs` takes the current state s, the Q-function to be used, and the actions acceptable to the objectives up to the last one and outputs the actions acceptable to the objectives up to and including the current one.

Now, we will describe how Π_i can be defined by three thresholding approaches. Note that we use the term "thresholding" as an umbrella term for all techniques to define acceptable policies and it only reflects the history of the domain which started with using threshold values for acceptable policies in [9]. Also, note that these techniques operate over the set of policies and `AcceptableActs` operates over the set of actions. However, limiting these techniques for a single state s instead of $\forall s \in S$ will show that these techniques can be all directly adapted to `AcceptableActs`.

1. **Absolute Thresholding:** This is the approach proposed by [9] where the actions with values higher than a real number are considered acceptable. Formally,

$$\Pi_i \triangleq \{\pi_i \in \Pi_{i-1} \mid \hat{Q}_i^\star(s, \pi_i(s)) = \max_{a \in \{\pi_{i-1}(s) | \pi_{i-1} \in \Pi_{i-1}\}} \hat{Q}_i^\star(s, a), \forall s \in \mathcal{S}\} \tag{3.4}$$

2. **Absolute Slacking:** This is the approach taken by [4] where a slack from the optimal value in that state is determined and each action within that slack is considered acceptable.

$$\Pi_i \triangleq \{\pi_i \in \Pi_{i-1} \mid Q_i^\star(s, \pi_i(s)) \geq \max_{a \in \{\pi_{i-1}(s) | \pi_{i-1} \in \Pi_{i-1}\}} Q_i^\star(s, a) - \delta_i, \forall s \in \mathcal{S}\} \tag{3.5}$$

Notice that this thresholding scheme is not directly compatible with our definition of LMDPs in Section 2. However, they are both used to simply introduce some relaxation in policy selection; hence, there is not a fundamental difference between the two formulations of LMDPs. See [12] and [13] for a definition based on slacks.

3. **Relative Slacking:** In this approach, slacks are defined as ratios $\eta \in (0, 1]$ rather than absolute values. Then, any action with value greater than $(1 - \eta)$ times the optimal value is considered acceptable. Formally,

$$\Pi_i \triangleq \{\pi_i \in \Pi_{i-1} \mid Q_i^\star(s, \pi_i(s)) \geq (1 - \eta) \max_{a \in \{\pi_{i-1}(s) | \pi_{i-1} \in \Pi_{i-1}\}} Q_i^\star(s, a), \forall s \in \mathcal{S}\} \tag{3.6}$$

While has not been proposed in any previous work, we included this for the sake of completeness. Notice that "Relative Thresholding" would be essentially the same technique, only with different parameters.

In the next section, we will discuss the shortcomings of the TLQ approach.

3.4 Issues

The benefits and drawbacks of TLQ depend largely on the type of task at hand. Different variants of TLQ are able to solve different types of tasks. Thus, in order to discuss the issues with TLQ, we need to give a taxonomy of the tasks.

3.4.1 Taxonomy

Firstly, we should introduce some additional terminology to facilitate the discussion.

- **Contsrained/Unconstrained Objective:** Constrained objectives are bounded in quality by their threshold values above which all values are considered the same. An unconstrained objective does not have a such threshold value. In an LMDP setting, all objectives but the last one are constrained objectives. Note that the concept of constrained/unconstrained objectives also exists in the slack formulation where constrained objectives are defined to be the ones where a slack from the optimal action-value is allowed.

- **Terminating Objective:** An objective that either implicitly or explicitly pushes the agent to go to a terminal state of the MDP. More formally, this means that discounted cumulative reward of $\zeta^1[0:t]$ is higher than discounted cumulative reward of $\zeta^2[0:t']$ if $s_t^1 \in S_F$ and $s_{t'}^2 \notin S_F$ where $\zeta^1[0:t]$ and $\zeta^2[0:t']$ denote length t and t' prefixes of two different trajectories ζ^1 and ζ^2.

- **(Quantitative) Reachability Objective:** A reachability objective is represented by a unit-reward in the terminal states of the MDP and zero rewards elsewhere. This can be generalized to a quantitative version when different terminal states can have different reward values. We will call an objective that has non-zero rewards in at least one non-terminal state a non-reachability objective. Notice that a quantitative reachability objective would be a terminating objective if its rewards are positive and non-terminating if its rewards are negative.

This taxonomy also helps to see what are the problematic cases and what could be the solutions. In the literature, some case studies have been done on problems that fall into different parts of the problem space. While these case studies give some empirical evidence that TLQ can be used in these parts of the problem space, none of them have any claims of its applicability in general instances of these cases. Hence, the issue of lack of theoretical guarantees for TLQ remains in these cases too. We believe that TLQ would fail to find the optimal policies in many problems that fall under these cases. We give an example of this in Section 3.4.4.

We will summarize the empirical results from the literature about TLQ's applicability below.

1. *TLQ is shown to work empirically on a case study*:

   (a) [10] shows a case study where the constrained objective is reachability, the unconstrained objective is terminating, and TLQ works.

   (b) [4] shows a case study where the constrained objective is a non-terminating reachability constraint, i.e. non-positive reward reachability objective and TLQ works.

2. *TLQ does not work*:

   (a) [10] shows that TLQ does not work when the constrained objective is non-reachability.

   (b) In this work, we show that TLQ also does not work when the constrained objective is a terminating reachability objective, i.e. positive reward reachability, but the unconstrained one is non-terminating.

3.4.2 Benchmark

In the next sections, we will describe some scenarios that are common in control tasks yet TLQ fails to solve. To illustrate different issues caused by TLQ, we need an adaptable multiobjective task. Moreover, in order to simplify these illustrations, we should use a finite state and action space example where the tabular version of TLQ can be used. The tabular version refers to the case where the Q-function is represented by a piecewise function (table) that maps each state-action pair to a real number independently of other values. The alternative to this would be function approximation where the value for a state-action pair cannot be set independent of others. Our MAZE environment which will be introduced in this section satisfies all of our requirements. Figure 3.1 shows an example MAZE.

The state space of a MAZE task consists of the cells of the grid. Each cell is a state and it is represented by a two-tuple with its column as the first element and its row as the second element. The action space consists of four actions: $Up, Down, Left, Right$. These actions move the agent in the maze and any action which would cause the agent to leave the grid will leave its position unchanged.

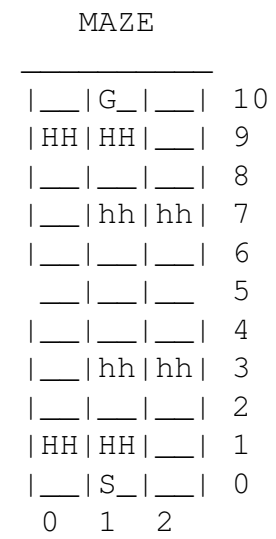

Figure 3.1: An example MAZE which can be used to demonstrate the issues with uniform thresholding for TLQ.

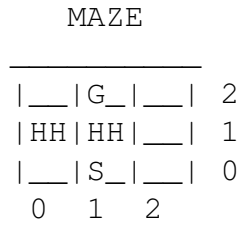

Figure 3.2: A simple maze that can be used to demonstrate how TLQ fails to reach the goal state.

In all MAZE instantiations, each episode starts in S and G is the terminal state. There are also two types of bad tiles with different costs. The tiles marked with Hs show the high penalty bad tiles whereas the ones with hs show the low penalty ones. In this work, we will use -5 as the high penalty and -4 as the low penalty. But we consider the penalty amounts parameters in the design of a maze; so, they could change. There are two high level objectives: Reach G and avoid bad tiles. Ordering of these objectives and exact definition of them results in different tasks. We use these different tasks to cover different parts of the problem space as described in Section 3.4.1.

In the next sections, we will only present the shape of the maze and objectives but the dynamics of it will be as described here.

3.4.3 Failing to Reach the Goal

A common scenario in control tasks is having a primary objective to reach a goal state and a secondary objective that evaluates the path taken to there. However, TLQ either needs to ignore the secondary objective or fails to satisfy the primary objective in such cases. Formally, this is a scenario where the primary objective is a reachability objective. All of the thresholding methods from Section 3.3 fail to guarantee reaching the goal in this setting when used threshold/slack values are uniform throughout state space. The maze in Figure 3.2 can be used to observe this phenomenon. Assume that our primary objective is to reach G and we encode this with a reward function that is 0 everywhere but G where it is R. And our secondary objective is avoiding the bad tiles. There are two Pareto optimal policies in this maze, indicated by the state trajectories:

1. $(1,0) \rightarrow (1,1) \rightarrow (1,2)$
2. $(1,0) \rightarrow (2,0) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (1,2)$

We can assume that the designer might be interested in both of them. Now, we will show how the second one is unattainable by TLQ.

Since the reward is given only in the goal state, $\hat{Q}^\star$ can be equal to τ only for the state-action pairs that lead to the goal state. All others will be discounted from these; hence, all are less than τ. This means the agent will always ignore the secondary when using absolute thresholding of [9]. Alternatively, we can consider using absolute thresholding with $Q^\star$ and applying the threshold only for action selection, similar to absolute slacking. However, this kind of absolute thresholding creates a circular region of acceptable "states" whose center is the terminal state and diameter is a function of actions, discounting, and the threshold. Once the agent gets into this region, there is not any reason to actually reach the terminal state. A natural question here would be whether the diameter could be small enough to limit the acceptable states to the goal state. While this is technically possible, setting such a conservative threshold means the secondary objective will never be considered. An alternative would be having threshold values increase as they get closer to the terminal state; however, this would require significant effort from the user and take the

intuitiveness of thresholding away. If the secondary objective happens to be a terminating one, the terminal state will be reached because the agent will be pushed there by the secondary objective. We believe this is why [10] has not observed this issue as in their experiments with Deep Sea Treasure. As their secondary objective, minimizing the number of steps before the terminal state, is a terminating objective.

To illustrate this issue, consider again the Maze in Figure 3.2. $Q_1^\star((2,2), \textit{Left}) = R$ and $Q_1^\star((2,2), \textit{Right}) = \gamma R$. Hence, if we want to agent to choose *Left* over *Right* in state $(2,2)$, τ_1 should be set such that it is greater than γR. The problem is, all of the actions in any state that is further than one step from G have action-values smaller than or equal to γR, meaning the agent has to ignore the secondary objective completely.

Similarly, Relative Slacking determines the maximum detour. If a non-optimal action delays reaching the goal for k steps, this can be allowed only by defining $\eta > \gamma^k$. However, this detour can be taken repeatedly, preventing actually reaching the goal.

Seeing how Absolute Slacking fails to overcome this problem is a little trickier. It requires a closer inspection of action-values. Since we want that the agent to go left in $(2,2)$, the following should be true:

$$
\begin{aligned}
& Q_1^\star((2,2), \textit{Right}) < Q_1^\star((2,2), \textit{Left}) - \delta_1 \\
\implies & \gamma R < R - \delta_1 \\
\implies & \delta_1 < R(1-\gamma)
\end{aligned}
$$

However, allowing the agent to pick *Right*, instead of *Up* in $(1,0)$ requires:

$$
\begin{aligned}
& Q_1^\star((1,0), \textit{Right}) \geq Q_1^\star((2,2), \textit{Up}) - \delta_1 \\
\implies & \gamma^3 R \geq \gamma R - \delta_1 \\
\implies & \delta_1 \geq R\gamma(1-\gamma^2)
\end{aligned}
$$

Combining these two requirements implies that:

$$
\begin{aligned}
& R(1-\gamma) > R\gamma(1-\gamma^2) \\
\implies & 1 > \gamma(1+\gamma) \\
\implies & 0 > \gamma^2 + \gamma - 1 && \text{Solving the quadratic equation} \\
\implies & 0.62 > \gamma
\end{aligned}
$$

This shows that to reach the desired policy, not only δ but γ needs to be adjusted too. However, the γ parameter is assumed to be an environment constant and traditionally set to values close to 1. Moreover, there is no real way to find the correct γ value apart from computing the action-value function, the very thing we are trying to compute.

Also, a similar analysis shows that small tricks like replacing the primary reward function with

$$
R_1'(s,a,s') = \begin{cases} 0, \text{if } s' = G \\ -1, \text{otherise} \end{cases} \tag{3.7}
$$

with or without discounting do not solve this problem.

3.4.4 Failure to Sacrifice Early and Late

Even if the problem above is avoided because the secondary objective happened to be a terminating one, there are other, more subtle, issues with TLQ. An important one is the failure to sacrifice early and late in the episode.

Consider the maze shown in Figure 3.1. There are bad tiles in four rows and avoiding any of the rows of bad tiles takes two steps. For example, compare the following two paths:

1. $(1,0) \to (1,1) \to (1,2)$
2. $(1,0) \to (2,0) \to (2,1) \to (2,2) \to (1,2)$

Path 2 avoids the bad tiles but it takes 4 steps to get to $(1, 2)$ from $(1, 0)$ compared to only 2 steps of Path 1. Since avoiding any tiles costs the same number of extra steps, a natural policy in this maze would be avoiding HH tiles and ignoring hh tiles. However, this is not possible with either thresholding method. Now, we will discuss how each thresholding method fails this achieving this Pareto optimal policy.

As discussed in the previous section, absolute thresholding described in Section 3.3 results in only considering the first objective. Hence, we will consider learning $Q^\star$ and applying thresholding to it. Similar to the previous section, the action-values show the reward in G discounted by the length of the shortest path to G from the cell this state-action pair leads to. For example, $Q^\star((1, 8), \textit{Right}) = R\gamma^3$ as it takes 3 steps to get to G from $(2, 8)$. Hence, the action-values increase as the agent gets closer to the goal. Assume that the agent is in $(1, 0)$, the action that we need to take is *Right*, meaning τ_1 should be set smaller than or equal to $\gamma^{11}R$ in Absolute Thresholding. However, since the action values will be larger than this in the states closer to G, it will mean that the primary objective will be ignored for the rest of the episode. Hence, the agent will avoid h tiles too and the desired policy is unattainable.

Similarly, since Relative Thresholding effectively limits the length of detours and detours for avoiding hs are of the same length as the ones for Hs, this cannot give a policy that only goes through hs.

Absolute Slacking will cause this problem in the reverse, meaning the late episode detours requires detours during the whole episode. Assume the agent is in cell $(1, 8)$, then we need

$$
\begin{aligned}
& Q_1^\star((1, 8), R) > Q_1^\star((1, 8), U) - \delta_1 \\
\implies & \gamma^3 R > \gamma R - \delta_1 \\
\implies & \delta_1 > R\gamma(1 - \gamma^2)
\end{aligned}
$$

Then, if going left instead of up is not allowed in cell $(1, 6)$:

$$
\begin{aligned}
& Q_1^\star((1,6), R) < Q_1^\star((1,6), U) - \delta_1 \\
\implies & \gamma^5 R < \gamma^3 R - \delta_1 \\
\implies & \delta_1 < R\gamma^3(1-\gamma^2)
\end{aligned}
$$

Combining these two requirements gives $R\gamma^3(1-\gamma^2) > R\gamma(1-\gamma^2)$, which requires $\gamma > 1$ which is false.

3.5 Variations to TLQ and Some Alternatives

In this section, we will try to address the problems with TLQ within the framework of value function algorithms. We will start by briefly talking about two of our failed attempts to develop working TLQ-variants to show the breadth of the problems and our work. Moreover, we believe these ideas are quite natural and can look promising; so, we would like to share our experience to help people working on TLQ algorithms.

Then, we will describe three of our working solutions. While these solutions are limited either in terms of convergence or applicable domains, they provide either a good solution to a sizeable subset of common tasks or give a good alternative to TLQ in the general case.

3.5.1 Failed Attempts

In this section, we will give our not very successful attempts to improve the performance of TLQ.

TL-SARSA

Our first failed attempt was switching to an on-policy learning framework that could solve agents getting stuck problem in Section 3.4.3. An important reason for this issue was the agents' optimistic expectation that they would be following the optimal behavior after each action. So, we

considered that an on-policy agent which uses its realistic behavior to learn would not suffer from these issues.

So, we modified our update functions from Section 3.2 to mimic SARSA ([31]) instead of Q-Learning. This would mean replacing the max operators with the actual action. For example, Equation 3.3 (the update function from [4]) will become:

$$Q_i^\star(s,a) = \sum_{s' \in S} P(s,a,s')(R_i(s,a,s') + \gamma Q_i^\star(s',a') \tag{3.8}$$

where $a' = \pi(s')$.

However, this naive attempt failed due to some theoretical limitations of SARSA. [40] states that the convergence of SARSA is guaranteed under the condition that the policy is greedy in limit. However, our policies are not necessarily greedy with respect to $Q^\star$ in limit. Thresholding means that sometimes actions suboptimal w.r.t. $Q^\star$ are chosen. For example, if $Q_1^\star(s,a_1) > Q_1^\star(s,a_2) > \tau_1$ and $Q_2^\star(s,a_2) > Q_2^\star(s,a_1)$ for a state s in a two objective task, the policy we want to learn is not greedy w.r.t. $Q_1^\star$. This manifested itself as constant oscillations in the policy in our experiments.

Cyclic Action Selection

Our second half-failed attempt was modifying the action selection mechanism to solve the phenomenon described in Section 3.4.3. It was based on the intuition that the reason for this issue was unnecessary sacrifices in the primary objective, which is a constrained objective, that is not required by the secondary objective. For example, if we consider the maze in Figure 3.2, going left or right in cell $(2,2)$ is the same w.r.t. secondary objective, hence the agent should not sacrifice from the primary objective irrespective of the thresholds/slacks. Using this intuition, we developed a cyclic action selection algorithm. In Algorithm 3, we show a two-objectives version of it for simplicity. While it can be generalized to K objectives, we do not believe it would be of interest considering its failure to completely address our problems.

Algorithm 3: ActionSelection

Function `CyclicActionSelection`$(s, Q|A)$**:**
  $A_0 \leftarrow A$
  $A_1 \leftarrow$ `AcceptableActs`(s, Q_1, A_0)
  if $|A_1| \leq 1$ **then** `// If there are not more than one option`
    return $\arg\max_{a \in A_0} Q_1(s, a)$
  $A_2 \leftarrow$ `AcceptableActs`(s, Q_2, A_1)
  if $|A_2| \leq 1$ **then**
    return $\arg\max_{a \in A_1} Q_2(s, a)$
  return $\arg\max_{a \in A_2} Q_1(s, a)$; `// Max over` A_2 `but wrt` Q_1

As the pseudocode shows, the idea is to assign the unconstrained function a threshold/slack which would be used to return the action selection right back to the primary objective but after applying this new threshold. This can be seen from the Q-functions used with $\arg\max$ and `AcceptableActs` throughout the algorithm. It starts with using Q_1, then uses Q_2 if there are multiple acceptable actions w.r.t. Q_1. Finally, it uses Q_1 again if there is more than one action acceptable w.r.t. Q_2. Notice that there is a `AcceptableActs` call using Q_2 which is different than Algorithm 2. This requires having a threshold/slacking for the unconstrained objective which is basically a hack. It can be used with any thresholding function from Section 3.3.

However, there are several problems with this approach. Firstly, having a threshold for the unconstrained objective removes one of the important supposed benefits of TLQ, namely its intuitiveness. Especially the cyclic nature of this makes two different threshold values, one for each objective, to be coupled in a complex way when deciding which detours will be taken. This can lead to a blind hyperparameter search. Our experiments show that the success of the policy is highly sensitive to the choices of these two hyperparameters.

Also, the problem described in Section 3.4.4 persists, which means some of the very natural policies cannot be found with this technique.

3.5.2 Informed Targets

While using the SARSA variant has failed as seen in Section 3.5.1, we believe that our intuition about the root of the issues described in Section 3.4.3 was correct. Hence, we decided that an approach that could better align the update target with the "actual policy" could still solve the problem with short-sighted sacrifices. One such way could be accounting for the possibility of actions not being taken according to the given objective. Here, we will present the approach for two objectives. Its generalization to K objectives is not necessarily straightforward and we regard it as a future research direction. To illustrate the idea, assume that $Q_1^\star(s', a_1) > Q_1^\star(s', a_2) > \tau_1$ and $Q_1^\star(s', a_2) > Q_2^\star(s', a_1)$ for a state s' in a task with only two actions. Eq. 3.3 uses $R_1(s, a, s') + \gamma_1(s', a_1)$ when computing update target for $Q_1^\star(s, a)$ as a_1 maximizes $Q_1^\star$ in state s'. However, this is misleading as a_1 will never be chosen in state s'. Instead $Q_1^\star(s', a_2)$ should be used as a_2 maximizes $Q_2^\star$ in s'. Notice that this is still different than TL-SARSA as we may be following a completely different policy. In other words, a_2 is used not because it is actually the action taken but it would be the action taken in the optimal case. We call this "informed targets" as value functions make "informed" updates, knowing what would be the actual action taken. More formally, this means modifying the update function to:

$$Q_1^\star(s, a) = \sum_{s' \in S} (R_1(s, a, s') + Q_1^\star(s', \arg\max_{\pi \in \Pi_1} Q_2^\star(s', \pi(s')) P(s, a, s') \tag{3.9}$$

Notice that the target for the objective 1 is computed by choosing the optimal action with respect to 2. This prevents optimistic updates that happen due to targets computed with actions that never would be taken. It should be noted that these updates were the reason for the failure mode discussed in Section 3.4.3. Preventing them solves this issue but brings a different problem: Instability in update targets. Consider the scenario in Section 3.4.3. If the current policy is going to left in state $(2, 2)$, the value of going right would be γR. Assuming that the threshold is smaller than γR, at some point the value of going right would pass the threshold and both going right and left would be equally good. Once this happens, the update target for going right will become

$\gamma Q_1^\star(s, Right)$, hence it will start to decrease until it is smaller than the threshold. Then, the target will go back to its original value, hence result in an endless cycle. While it is possible to introduce some buffer in these updates such that the oscillations do not affect the policy that is being followed, the optimality of the resulting policy will depend on the initialization.

The update function with buffer hyperparameter b can be obtained by replacing Π_i in Section 3.9 with $\hat{\Pi}_i$ which is defined as:

$$\hat{\Pi}_i \triangleq \{\pi_i \in \hat{\Pi}_{i-1} | Q_i^\star(s, \pi_i(s)) \geq \max_{a \in \{\pi_{i-1}(s) | \pi_{i-1} \in \Pi_{i-1}\}} Q_i^\star(s, a) - \delta_i - b, \forall s \in \mathcal{S}\} \tag{3.10}$$

Notice that this will lead to a smaller oscillation zone which in turn is going to prevent the policy from oscillating as it still uses Π_i. Also, note that the problems in Section 3.4.4 still persists.

3.5.3 Solution to Non-reachability Constrained Objective Case

In this section, we will show how a problem where constrained objectives are non-reachability can be solved by augmenting the state space. This idea of state augmentation has been used before with slightly different or narrower purposes [41].

Single Constrained Objective

When the constrained objective is a non-reachability objective, this can be solved by using state augmentation that keeps track of obtained cost/reward for the constrained objective so far. In this section, we will use a different MDP that is inspired by a real-life scenario to also give a more intuitive example and show the real use of LMDPs.

An example: A car travels across the country using highways. It starts the journey in the city s_0 and tries to go to a city $s_F \in S_F$. Once he reaches a city in the set s_F, he will stop traveling. The highway toll for the highway from the city s to s' is represented by the function $h(s, s')$ where $h : S \times S \to \mathbb{R}$. The driver has a budget of B dollars and tries to have the best trip within this budget. His cost within this budget will be reimbursed by his company, so he has no incentive to

spend less as long as he is within the budget. His pleasure from arriving in the city s is given by $p(s)$ where $p : S \to \mathbb{R}$ and $p(s) = 0, \forall s \notin S_F$.

Formally, we have two objectives: minimizing the tolls and maximizing the pleasure. Minimizing tolls is constrained/thresholded by the budget B. Maximizing the pleasure is unconstrained. Following our formulation in Section 2:

- $R_1(s, a, s') = -h(s, s')$ and $\tau_1 = -B$. Notice again that we expressed the threshold without discounting. Since we will not be using TLQ, we do not need to find the corresponding discounted threshold. Also, notice that the corresponding discounted threshold actually depends on the trajectory.
- $R_2(s, a, s') = p(s')$.

We can express this two-objective task and preserve the preferences by constructing the following single-objective task:

- Set of states: $\hat{S} = S \times \mathbb{R}$ where (s, c) means the driver is in the city s and so far the driver has spent $B - c$ dollars on tolls. Augmented initial state $\hat{s}_0 = (s_0, B)$.
- Set of actions: $\hat{A} = A$
- Transition function $\hat{P} : \hat{S} \times A \times \hat{S} \to [0, 1]$ where

$$\hat{P}((s, c), a, (s', c')) = \begin{cases} P(s, a, s'), \text{ if } c' = c - h(s, s') \\ 0, \text{otherwise} \end{cases} \tag{3.11}$$

- Reward function $\hat{R} : \hat{S} \times A \times \hat{S} \to \mathbb{R}$

$$\hat{R}((s, c), a, (s', c')) = \begin{cases} 0, \text{if } s' \notin S_F \\ p(s'), \text{else if } c' \geq 0 \\ \lambda c', \text{otherwise} \end{cases} \tag{3.12}$$

Note that a non-zero reward will be given only when the car reaches a final destination. If the driver has stayed within the budget, he gets his pleasure value as the reward. If he has exceeded the budget, he is penalized accordingly with a multiplier λ. Implicitly, we assume that $p(s) > 0, \forall s \in S_F$.

Also, note that while this reward function specifies the optimal policy correctly, it may not be a good reward function for learning and exploration purposes. For instance, until the agent learns how to stay within the budget, all the terminal states will have negative values and non-terminal states will have higher values. Hence, the agent can get stuck here by trying to avoid terminal states. Realizing that it can get positive rewards may require a good exploration policy.

This has the following advantages:

- Different thresholds are supported
- Thresholding is intuitive
- Convergence proofs exist.

Optimality of New MDP

We can easily show that this single-objective task has the same ordering of trajectories as the original task. More formally, $\zeta^1 = s_0^1, a_0^1, s_1^1, a_1^1, \ldots, s_{n^1}^1$ is better than $\zeta^2 = s_0^2, a_0^2, s_1^2, a_1^2, \ldots, s_{n^2}^2$ under the original task if and only if the augmented trajectory $\hat{\zeta^1}$ is also better than augmented trajectory $\hat{\zeta^2}$ under this single-objective task. Here, we will use the cumulative reward as the optimality metric when comparing trajectories. For the original task trajectories, we use the thresholded lexicographic comparison relation defined in Section 2. Note that subscripts n^1 and n^2 denotes the indexes of ζ^1 and ζ^2, not the polynomials.

Proof: $\zeta^1 \geq \zeta^2$ under the original task if and only if one of the following must be true:

1. $\sum_{\zeta^1} R_1(s,a,s'), \sum_{\zeta^2} R_1(s,a,s') \geq \tau_1$ and $\sum_{\zeta^1} R_2(s,a,s') \geq \sum_{\zeta^2} R_2(s,a,s')$
2. $\sum_{\zeta^1} R_1(s,a,s') \geq \tau_1 > \sum_{\zeta^2} R_1(s,a,s')$

3. $\tau_1 > \sum_{\zeta^1} R_1(s,a,s') \geq \sum_{\zeta^2} R_1(s,a,s')$

We can show that each of these statements implies that the same ordering holds for $\hat{\zeta^1} \geq \hat{\zeta^2}$ under the single objective task. Firstly observe that:

$$\begin{aligned}
\sum_{\hat{\zeta}} \hat{R}(\hat{s},\hat{a},\hat{s}') = p(\hat{s}_n(s)) &\iff \sum_{\hat{\zeta}} \hat{R}(\hat{s},\hat{a},\hat{s}') > 0 \\
&\iff \hat{s}_n(s) \in S_F \wedge \hat{s}_n(c) \geq 0 \\
&\iff \sum_{\hat{\zeta}} h(s,s') \geq B \iff \sum_{\zeta} R_1(s,a,s') \geq \tau_1
\end{aligned}$$

Then, for the first case:

$$\begin{aligned}
\sum_{\zeta^1} R_1(s,a,s'), \sum_{\zeta^2} R_1(s,a,s') \geq \tau_1 \implies &\sum_{\hat{\zeta^1}} \hat{R}(\hat{s},\hat{a},\hat{s}') = p(\hat{s}_{n^1}(s)) \wedge \\
&\sum_{\hat{\zeta^2}} \hat{R}(\hat{s},\hat{a},\hat{s}') = p(\hat{s}_{n^2}(s))
\end{aligned}$$

Also,

$$\begin{aligned}
\sum_{\zeta^1} R_2(s,a,s') \geq \sum_{\zeta^2} R_2(s,a,s') &\implies p(\hat{s}_{n^1}(s)) \geq p(\hat{s}_{n^1}(s)) \\
&\implies \sum_{\hat{\zeta^1}} \hat{R}(\hat{s},\hat{a},\hat{s}') > \sum_{\hat{\zeta^2}} \hat{R}(\hat{s},\hat{a},\hat{s}') \\
&\implies \hat{\zeta^1} \geq \hat{\zeta^2}
\end{aligned}$$

For the second case,

$$
\begin{aligned}
&\sum_{\zeta^1} R_1(s,a,s') \geq \tau_1 > \sum_{\zeta^2} R_1(s,a,s') \\
\implies &\sum_{\hat{\zeta}^1} \hat{R}(\hat{s},\hat{a},\hat{s}') > 0 \wedge \sum_{\hat{\zeta}^2} \hat{R}(\hat{s},\hat{a},\hat{s}') \leq 0 \\
\implies &\sum_{\hat{\zeta}^1} \hat{R}(\hat{s},\hat{a},\hat{s}') > \sum_{\hat{\zeta}^2} \hat{R}(\hat{s},\hat{a},\hat{s}') \\
\implies &\hat{\zeta}^1 \geq \hat{\zeta}^2
\end{aligned}
$$

For the third case, we can observe that:

$$
\begin{aligned}
&\tau_1 > \sum_{\zeta^1} R_1(s,a,s') \geq \sum_{\zeta^2} R_1(s,a,s') \\
\implies &B > \sum_{\hat{\zeta}^1} h(s,s') \geq \sum_{\hat{\zeta}^2} h(s,s') \\
\implies &\hat{s}_{n^1}(c) \geq \hat{s}_{n^2}(s) \\
\implies &\lambda \hat{s}_{n^1}(c) \geq \lambda \hat{s}_{n^2}(s) \\
\implies &\sum_{\hat{\zeta}^1} \hat{R}(\hat{s},\hat{a},\hat{s}') > \sum_{\hat{\zeta}^2} \hat{R}(\hat{s},\hat{a},\hat{s}') \\
\implies &\hat{\zeta}^1 \geq \hat{\zeta}^2
\end{aligned}
$$

□

Note that we've specified the unconstrained objective as a quantitative reachability objective, ie. it is non-zero only in the terminal states. Now, we will remove the restriction over p.

Alternative 1: First option is to extend the state space again to keep track of p as well. So, the MDP will be:

- State Space: $\hat{\hat{S}} = S \times \mathbb{R} \times \mathbb{R}$ where the state $(s, c, \bar{p})$ corresponds to accumulating $\bar{p}\, p(s)$ so far.

- Transition function: $\hat{\hat{P}} : \hat{\hat{S}} \times A \times \hat{\hat{S}} \to [0, 1]$ where

$$\hat{\hat{P}}((s, c, \bar{p}), a, (s', c', \bar{p}')) = \begin{cases} \hat{P}((s, c), a, (s', c')), \bar{p}' = \bar{p} + p(s') \\ 0, \text{otherwise} \end{cases} \tag{3.13}$$

- Reward function: $\hat{\hat{R}} : \hat{\hat{S}} \times A \times \hat{\hat{S}} \to \mathbb{R}$ where

$$\hat{\hat{R}}((s, c, \bar{p}), a, (s', c', \bar{p}')) = \begin{cases} 0, \text{if } s' \notin S_F \\ \bar{p}', \text{else if } c' \geq 0 \\ \lambda c', \text{otherwise} \end{cases} \tag{3.14}$$

Alternative 2: Extending the state space is not always optimal, as it increases the complexity. Instead, we can try to directly modify Equation 3.12. With this, we will still use $\hat{S}$ and $\hat{P}$ as the state space and transition function, respectively.

- Most simply, we can start giving $p(s')$ reward in the non-terminal states. Then, we can guarantee the lexicographic ordering by subtracting a large value C_l that is guaranteed to be larger than $\sum_t p(s')$ from $\lambda c'$.

$$\hat{\hat{R}}((s, c), a, (s', c')) = \begin{cases} p(s'), \text{if } s' \notin S_F \\ p(s'), \text{else if } c' \geq 0 \\ \lambda c' - C_l, \text{otherwise} \end{cases} \tag{3.15}$$

Optimality proofs of these new MDPs are very similar to our proof in Section 3.5.3. So, we leave it to the reader to avoid repeating it.

Multiple Constrained Objectives Case

Our analysis above assumes that there are only two objectives: a constrained primary objective and an unconstrained secondary objective. However, in that setting, many CMDP algorithms are readily applicable. Therefore, we are more interested in when we have multiple constraints that need to be solved in the lexicographic order. Yet, extending the approach above to this setting is not straightforward. To apply it, we need to know which constraints can be satisfied together. More formally, if the constrained objectives are $1, \ldots, (k-1)$, we need to find the maximum i such that there exists a policy that satisfies objectives $1, \ldots, (i-1)$, i.e. can reach a state $(s, c_1, c_2, \ldots, c_{i-1})$ such that $s \in S_F$ and $c_1, c_2, \ldots, c_{i-1} \geq 0$. We identified three different approaches that could be used for this but we believe future work is needed to develop more efficient methods.

One-by-one The simplest method to solve tasks with multiple constrained objectives is reminiscent of linear search algorithm. We can start with the first (most important) constraint and see if we can find a policy that satisfies it, i.e. can reach (s, c_1) such that $s \in S_F$ and $c_1 \geq 0$ from s_{init}. If such a policy exists, we can introduce the second constraint to see if a policy that satisfies both of them simultaneously exists. Continuing in this fashion, it can be found up to which objective the agent can satisfy simultaneously. However, this method can be prohibitively expensive as it requires solving $O(k)$ subproblems. More importantly, it is very hard if not impossible to know whether a subproblem is not solvable or just taking too long to learn.

For this method, we can construct the following reward function for each i value in different ways. An approach would be maximizing the worstly violated constraint:

$$\hat{R}((s, c_1, \ldots, c_{i-1}), a, (s', c'_1, \ldots, c'_{i-1})) = \begin{cases} R(s, a, s'), \text{if } s' \notin S_F \\ R(s, a, s'), \text{else if } c'_j \geq 0 \ \ \forall j < i \\ \lambda \min_j c'_j - C_l, \text{otherwise} \end{cases} \tag{3.16}$$

Where R is the reward function of the unconstrained objective in the original MDP and C_l is an upper bound on the unconstrained reward that can be collected during an episode.

Binary Search As the name suggests, this method is inspired by binary search algorithm. Assuming the constrained objectives are $1, \ldots, (k-1)$, we can start by trying to solve constraints $1, \ldots, \lfloor \frac{k}{2} \rfloor$, then we can try $1, \ldots, \lfloor \frac{3k}{4} \rfloor$ or $1, \ldots, \lfloor \frac{k}{4} \rfloor$ depending on whether it was solvable or not, respectively. While this method is faster than one-by-one, it still suffers from the same halting problem. We can use Eq. 3.16 for this approach too.

Dynamic Search This method is not concretized and is intended mostly as an idea for future research. [14] presents an approach to set the threshold values for TLQ dynamically, depending on the attainable performance up to that point in the training. Similarly, we can introduce and remove constraints dynamically during the training without waiting for the algorithm to successfully converge for a subproblem.

Chapter 4

Policy Gradient Approach

In this section, we will introduce our policy gradient approach that utilizes consecutive gradient projections to solve LMDPs. We will start by giving the relevant linear algebra and geometry background for vector projections onto certain sets. Then, we will introduce our Lexicographic Projection Algorithm (LPA), a multi-objective optimization algorithm that lexicographically optimizes objectives by iteratively projecting the gradients of each objective. After demonstrating the usefulness of LPA on basic optimization tasks, we will show how it can be combined with policy gradient algorithms to solve LMDPs.

4.1 Background

In this section, we will start by showing how the projection equations we use in our algorithm are derived. For the sake of completeness, we will start with some simpler and well-known projections and move on to explain the projections that are actually used in our algorithm.

4.1.1 Orthogonal Projection onto a Hyperplane

One of the most well-known projection tasks is projecting a vector $\boldsymbol{y} \in \mathbb{R}^n$ onto a hyperplane $H_{\boldsymbol{a}}$ that passes through the origin, specified by its normal vector $\boldsymbol{a} \in \mathbb{R}^n$ as $H_{\boldsymbol{a}} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \langle \boldsymbol{x}, \boldsymbol{a} \rangle = 0\}$ where $\langle\rangle$ denotes the dot product defined as $\boldsymbol{v}^T\boldsymbol{a} = \sum_i \boldsymbol{v}_i\boldsymbol{a}_i$. Projection of $\boldsymbol{y}$ onto $H_{\boldsymbol{a}}$ is notated as $\boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y})$ and defined as $\boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y}) = \arg\min_{\boldsymbol{v} \in H_{\boldsymbol{a}}} \|\boldsymbol{v} - \boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y})\|$. $\|\|$ denotes the L2 norm defined as

$$\|\boldsymbol{v}\| = \sqrt{\boldsymbol{v}^T\boldsymbol{v}} = \sqrt{\sum_i \boldsymbol{v}_i^2}$$

$\boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y})$ can be found easily by using well-known result $\boldsymbol{y} - \boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y}) \parallel \boldsymbol{a}$, i.e. the projection error is parallel to the normal vector of the hyperplane. Then, there is a $c \in \mathbb{R}$ such that $\boldsymbol{y} - \boldsymbol{P}_{\boldsymbol{a}}^H(\boldsymbol{y}) = c\boldsymbol{a}$.

$$
\begin{aligned}
& \boldsymbol{y} - \boldsymbol{P}_{\boldsymbol{a}}^{H}(\boldsymbol{y}) \parallel \boldsymbol{a} \\
\implies & \boldsymbol{P}_{\boldsymbol{a}}^{H}(\boldsymbol{y}) = \boldsymbol{y} - c\boldsymbol{a} \\
\implies & \langle \boldsymbol{P}_{\boldsymbol{a}}^{H}(\boldsymbol{y}), \boldsymbol{a} \rangle = \langle \boldsymbol{y}, \boldsymbol{a} \rangle - c\langle \boldsymbol{a}, \boldsymbol{a} \rangle \\
\implies & 0 = \langle \boldsymbol{y}, \boldsymbol{a} \rangle - c\|\boldsymbol{a}\|^2 \\
\implies & c = \frac{\langle \boldsymbol{y}, \boldsymbol{a} \rangle}{\|\boldsymbol{a}\|^2} \\
\implies & \boldsymbol{P}_{\boldsymbol{a}}^{H}(\boldsymbol{y}) = \boldsymbol{y} - \frac{\langle \boldsymbol{y}, \boldsymbol{a} \rangle}{\|\boldsymbol{a}\|^2}\boldsymbol{a}
\end{aligned}
$$

4.1.2 Projection onto a Halfspace

In many cases, we may want to not project a vector that is already on one side of the hyperplane. For example, if we want to project a vector onto a feasible set, the vector that is already in the feasible set should not be projected. This idea can be formalized by extending the definition above to halfspaces. A positive halfspace $S_{\boldsymbol{a}}^{+}$ is defined as $S_{\boldsymbol{a}}^{+} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \langle \boldsymbol{x}, \boldsymbol{a} \rangle \geq 0\}$. This can be thought as the set of vectors with which $\boldsymbol{a}$ makes an angle less than or equal to $\frac{\pi}{2}$. We can define the projection $\boldsymbol{y}$ onto $S_{\boldsymbol{a}}^{+}$ as follows:

$$
\boldsymbol{P}_{\boldsymbol{y}}^{S^+}(\boldsymbol{a}) = \begin{cases} \boldsymbol{y}, & \boldsymbol{y} \in S_{\boldsymbol{a}}^{+} \\ \boldsymbol{P}_{\boldsymbol{y}}^{H}(\boldsymbol{a}), & \text{otherwise} \end{cases} \tag{4.1}
$$

Note that the piecewise function handles $\boldsymbol{y} \in S_{\boldsymbol{a}}^{+}$ and $\boldsymbol{y} \notin S_{\boldsymbol{a}}^{+}$ cases separately.

4.1.3 Projecting a vector onto a cone

While halfspaces are one of the most common sets in practice, they can be limiting in many cases. A natural extension to this idea would be limiting the set to vectors with which $\boldsymbol{a}$ makes an angle $\frac{\pi}{2} - \Delta$ for some $0 \leq \Delta \leq \frac{\pi}{2}$. The angle between two vectors is defined the same way it is in plane geometry as two vectors in n-dimensional space still form a plane, i.e. a 2-dimensional

subspace. The angle can be computed using dot product: The angle between two vectors is defined using dot product:

$$\langle v, u \rangle = \cos \angle v, u \|\boldsymbol{a}\| \|\boldsymbol{x}\|$$

Note that since $\cos(\Delta) = \cos 2\pi - \Delta$, the $\angle v, u$ can take two values between 0 and 2π. For simplicity, we will always talk about the smaller angle, i.e. $\angle : \mathbb{R}^n \times \mathbb{R}^n \to [0, pi]$.

This would be a hypercone that simplifies to a halfspace when $\Delta = 0$.

Let C_a^Δ be a hypercone with axis $a \in \mathbb{R}^n$ and angle $\frac{\pi}{2} - \Delta$, i.e.

$$C_a^\Delta = \{\boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x}\| = 0 \vee \frac{\boldsymbol{a}^T \boldsymbol{x}}{\|\boldsymbol{a}\| \|\boldsymbol{x}\|} \geq \cos(\frac{\pi}{2} - \Delta)\} \tag{4.2}$$

which uses the dot product formula above to see if the cosine of the angle between $\boldsymbol{a}$ and $\boldsymbol{x}$ is greater than the cosine of $\frac{\pi}{2} - \Delta$. For $0 \leq \Delta \leq \frac{\pi}{2}$, this corresponds to the angle between $\boldsymbol{a}$ and $\boldsymbol{x}$ being in the interval $[0, \frac{\pi}{2} - \Delta]$.

Then, the projection of a vector $\boldsymbol{g} \in \mathbb{R}^n$ onto C is defined as

$$\boldsymbol{g}_C^p = \arg\min_{\hat{\boldsymbol{g}} \in C} \|\hat{\boldsymbol{g}} - \boldsymbol{g}\|_2 \tag{4.3}$$

Solving this equation is not as straightforward as for the halfspaces. We will first show that $\boldsymbol{g}^p$ is planar with $\boldsymbol{g}$ and $\boldsymbol{a}$, i.e. they can be written as linear combinations of each other. are all on the same plane. This is intuitive and well-known in lower dimensions, but below can be seen a formal proof for higher dimensions. Once this is proven, we can utilize some two-dimensional geometric intuition to simplify the algebra.

4.1.4 Proof of Planarity

The projection is a constrained optimization problem:

$$\begin{aligned} &\min \|\boldsymbol{x} - \boldsymbol{g}\|_2 \\ &\text{subject to} \quad \frac{\boldsymbol{a}^T \boldsymbol{x}}{\|\boldsymbol{a}\|\|\boldsymbol{x}\|} \geq \cos\left(\frac{\pi}{2} - \Delta\right) \end{aligned}$$

If we can show that the solution to this vector is planar with $\boldsymbol{g}$ and $\boldsymbol{a}$, we will be done. The solution to this constrained optimization problem should satisfy Karush-Kuhn-Tucker (KKT) conditions which generalize the Lagrange Multiplier method to problems with inequality constraints. However, applying KKT conditions in this format does not provide a clean result. Therefore, we will prove a stronger claim that gives cleaner KKT conditions:

Lemma 4.1.1. *For any fixed length $\boldsymbol{x}$, the projection is minimized when $\boldsymbol{x}$, $\boldsymbol{g}$, and $\boldsymbol{a}$ are planar.*

Proof. This gives us the following modified optimization problem with an additional constraint. Now, we will show that the planarity does not depend on $\|\boldsymbol{x}\|$, which will be denoted as c.

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) = \|\boldsymbol{x} - \boldsymbol{g}\|_2 \\ \text{subject to} \quad & r(\boldsymbol{x}) = \frac{\boldsymbol{a}^T \boldsymbol{x}}{\|\boldsymbol{a}\|\|\boldsymbol{x}\|} \geq \cos\left(\frac{\pi}{2} - \Delta\right) = \sin \Delta \\ & h(\boldsymbol{x}) = \|\boldsymbol{x}\| = c \end{aligned}$$

Swapping norms with their dot product equivalents (replacing the norm in the objective and equality constraint with a norm square for conciseness) and writing the remaining in the standard format gives us:

$$
\begin{aligned}
\min \quad & f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{g}^T\boldsymbol{x} + \boldsymbol{g}^T\boldsymbol{g} \\
\text{subject to} \quad & r(\boldsymbol{x}) = \sin\Delta - \frac{\boldsymbol{a}^T\boldsymbol{x}}{\sqrt{\boldsymbol{a}^T\boldsymbol{a}}\sqrt{\boldsymbol{x}^T\boldsymbol{x}}} \leq 0 \\
& h(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{x} - c^2 = 0
\end{aligned}
$$

KKT conditions for this problem require that any minimum point $\hat{\boldsymbol{x}}$ should satisfy the following condition [42]:

$$
\begin{aligned}
& \nabla f(\hat{\boldsymbol{x}}) + \lambda\nabla h(\hat{\boldsymbol{x}}) + \mu\nabla r(\hat{\boldsymbol{x}}) = \boldsymbol{0} \\
\implies & 2\hat{\boldsymbol{x}} - 2\boldsymbol{g} + \lambda 2\hat{\boldsymbol{x}} + \mu\frac{\boldsymbol{a}(\sqrt{\boldsymbol{a}^T\boldsymbol{a}}\sqrt{\hat{\boldsymbol{x}}^T\hat{\boldsymbol{x}}}) - \frac{1}{2}\frac{\sqrt{\boldsymbol{a}^T\boldsymbol{a}}}{\sqrt{\hat{\boldsymbol{x}}^T\hat{\boldsymbol{x}}}}2\hat{\boldsymbol{x}}(\boldsymbol{a}^T\hat{\boldsymbol{x}})}{(\hat{\boldsymbol{x}}^T\hat{\boldsymbol{x}})(\boldsymbol{a}^T\boldsymbol{a})} = \boldsymbol{0} \qquad \hat{\boldsymbol{x}}^T\hat{\boldsymbol{x}} = c^2 \text{ (feasibility)} \\
\implies & 2\hat{\boldsymbol{x}} - 2\boldsymbol{g} + \lambda 2\hat{\boldsymbol{x}} + \mu\frac{\boldsymbol{a}(c\sqrt{\boldsymbol{a}^T\boldsymbol{a}}) - \frac{\sqrt{\boldsymbol{a}^T\boldsymbol{a}}}{c}\hat{\boldsymbol{x}}(\boldsymbol{a}^T\hat{\boldsymbol{x}})}{(c^2)(\boldsymbol{a}^T\boldsymbol{a})} = \boldsymbol{0} \\
\implies & 2\hat{\boldsymbol{x}} - 2\boldsymbol{g} + \lambda 2\hat{\boldsymbol{x}} + \mu\frac{c\boldsymbol{a} - \frac{\boldsymbol{a}^T\hat{\boldsymbol{x}}}{c}\hat{\boldsymbol{x}}}{c^2\sqrt{\boldsymbol{a}^T\boldsymbol{a}}} = \boldsymbol{0} \qquad \frac{\boldsymbol{a}^T\hat{\boldsymbol{x}}}{\sqrt{\hat{\boldsymbol{x}}^T\hat{\boldsymbol{x}}}} = \sin\Delta\sqrt{\boldsymbol{a}^T\boldsymbol{a}} \text{ (comp. slack.)} \\
\implies & 2\hat{\boldsymbol{x}} - 2\boldsymbol{g} + \lambda 2\hat{\boldsymbol{x}} + \mu\frac{c\boldsymbol{a} - \sin\Delta\sqrt{\boldsymbol{a}^T\boldsymbol{a}}\hat{\boldsymbol{x}}}{c^2\sqrt{\boldsymbol{a}^T\boldsymbol{a}}} = \boldsymbol{0} \\
\implies & 2\hat{\boldsymbol{x}} - 2\boldsymbol{g} + \lambda 2\hat{\boldsymbol{x}} + \mu\frac{\boldsymbol{a}}{c\sqrt{\boldsymbol{a}^T\boldsymbol{a}}} - \mu\frac{\sin\Delta}{c^2}\hat{\boldsymbol{x}} = \boldsymbol{0} \qquad \text{Reorganize the terms} \\
\implies & \hat{\boldsymbol{x}}(2 + 2\lambda - \mu\frac{\sin\Delta}{c^2}) = 2\boldsymbol{g} - \mu\frac{\boldsymbol{a}}{c\sqrt{\boldsymbol{a}^T\boldsymbol{a}}}
\end{aligned}
$$

Since $\boldsymbol{g}^p$ is such a minimum point, the above analysis holds for it too. Hence, it can be written as a linear combination of $\boldsymbol{a}$ and $\boldsymbol{g}$. This means that the three vectors are planar.

□

Note that we can see another important result from the analysis above. The complementary slackness condition of KKT requires that $\mu r(\hat{\boldsymbol{x}}) = 0$. However, if $\mu = 0$, the last line equation in the proof simplifies to

$$\hat{\boldsymbol{x}}(2 + 2\lambda) = 2\boldsymbol{g}$$

If $(2 + 2\lambda) \geq 0$, it means $\boldsymbol{g}$ and $\hat{\boldsymbol{x}}$ in the same direction. This is only possible if $\boldsymbol{g}$ is already in the hypercone. If $(2 + 2\lambda) < 0$, this means $\boldsymbol{g}$ and $\hat{\boldsymbol{x}}$ are in the opposite directions which cannot be the projection, as choosing 0 vector would give a smaller projection error. Hence, unless $\boldsymbol{g}$ is already in the hypercone and does not require a projection, $r(\hat{\boldsymbol{x}})$ should be 0. That means the angle between $\boldsymbol{a}$ and $\hat{\boldsymbol{x}}$ is $\frac{\pi}{2} - \Delta$.

4.1.5 Derivation of the Projection Formula

Now that it is known that all three vectors are planar, we can just use two-dimensional geometry to reason about it and derive the formula. This can be done as these three vectors in $\mathbb{R}^n$ will span a two-dimensional subspace of $\mathbb{R}^n$ unless they are all collinear, i.e. scalar multiplicative of each other. This would mean that $\boldsymbol{a}$ and $\boldsymbol{g}$ are already in the same direction and no projection is needed, which is a special case we will consider separately. Also, any 2-dimensional subspace of $\mathbb{R}^n$ is isomorphic to $\mathbb{R}^2$, i.e. identical in structure [43]. Figure 4.1 shows the case when the angle between $\boldsymbol{a}$ and $\boldsymbol{g}$, ϕ, is larger than $\frac{\pi}{2}$. It can be confirmed that the other configurations like will result in the same equations too. Note that when writing the equations below, we considered when $\boldsymbol{g}$ is outside of the cone. When $\boldsymbol{g} \in C$, we will simply call $\boldsymbol{g}^p = \boldsymbol{g}$ similar to the piecewise function in Section 4.1.2.

Firstly, we will find the direction of the projection. Let $\boldsymbol{p'}$ be a vector with the same direction as $\boldsymbol{g}^p$ and it can be written as below.

$$\boldsymbol{p'} = \boldsymbol{g} + \alpha \boldsymbol{a}$$

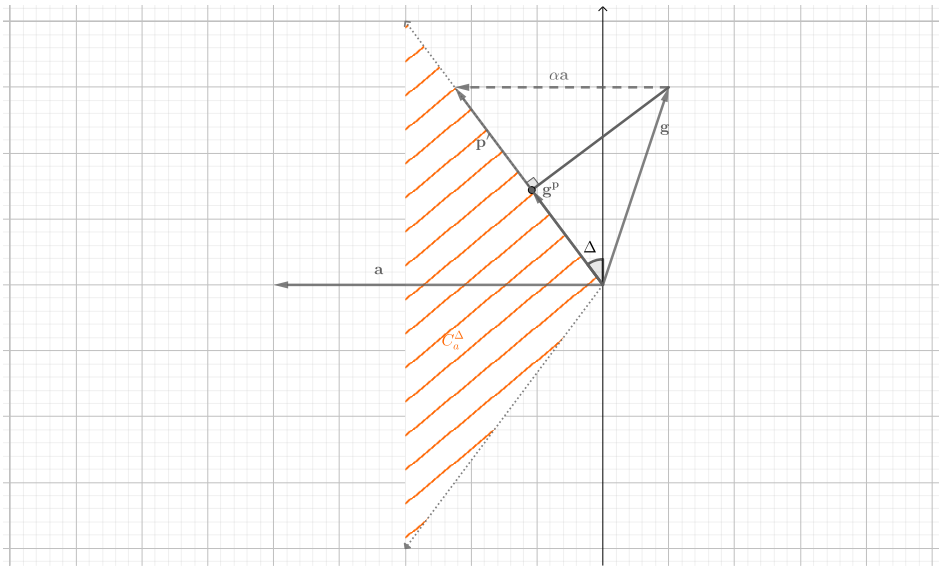

Figure 4.1: This figure shows how the vectors would be positioned on a plane. The orange region shows the cone. The angle between $\boldsymbol{a}$ and $\boldsymbol{g}$, ϕ, is omitted to not crowd the figure.

Then, we can find α as below by using the law of sines:

$$
\begin{aligned}
\alpha\|\boldsymbol{a}\| &= \|\boldsymbol{g}\| \sin(\phi - \frac{\pi}{2}) + \|\boldsymbol{g}\| \cos(\phi - \frac{\pi}{2}) \frac{1}{\sin(\frac{\pi}{2} - \Delta)} \sin\Delta \\
\implies \alpha\|\boldsymbol{a}\| &= -\|\boldsymbol{g}\| \sin(\frac{\pi}{2} - \phi) + \|\boldsymbol{g}\| \cos(\frac{\pi}{2} - \phi) \frac{1}{\sin(\frac{\pi}{2} - \Delta)} \sin\Delta \\
\implies \alpha\|\boldsymbol{a}\| &= -\|\boldsymbol{g}\| \cos\phi + \|\boldsymbol{g}\| \sin\phi \frac{1}{\cos\Delta} \sin\Delta \\
\implies \alpha\|\boldsymbol{a}\| &= \|\boldsymbol{g}\| (\sin\phi \frac{1}{\cos\Delta} \sin\Delta - \cos\phi) \\
\implies \alpha &= \frac{\|\boldsymbol{g}\|}{\|\boldsymbol{a}\|} (\sin\phi \frac{1}{\cos\Delta} \sin\Delta - \cos\phi) \\
\implies \alpha &= \frac{\|\boldsymbol{g}\|}{\|\boldsymbol{a}\|} (\sin\phi \tan\Delta - \cos\phi) \\
\implies \boldsymbol{p'} &= \boldsymbol{g} + \frac{\|\boldsymbol{g}\|}{\|\boldsymbol{a}\|} (\sin\phi \tan\Delta - \cos\phi)\boldsymbol{a}
\end{aligned}
$$

This $\boldsymbol{p'}$ has the correct direction but not necessarily the correct norm to minimize the projection error. The correct projection will be $\boldsymbol{g}^p = k\boldsymbol{p'}$ where $k \in \mathbb{R}$. We can find the k using the well-known rule that the projection error is perpendicular to the projection.

$$
\begin{aligned}
&\langle \boldsymbol{g} - k\boldsymbol{p'}, \boldsymbol{p'} \rangle = 0 \\
\implies &\langle \boldsymbol{g}, \boldsymbol{p'} \rangle - k\langle \boldsymbol{p'}, \boldsymbol{p'} \rangle = 0 \\
\implies &\|\boldsymbol{g}\|\|\boldsymbol{p'}\| \cos(\Delta + \phi - \frac{\pi}{2}) - k\|\boldsymbol{p'}\|^2 = 0 \\
\implies &\|\boldsymbol{g}\|\|\boldsymbol{p'}\| \cos(\frac{\pi}{2} - \Delta + \phi) - k\|\boldsymbol{p'}\|^2 = 0 \\
\implies &\|\boldsymbol{g}\|\|\boldsymbol{p'}\| \sin(\Delta + \phi) - k\|\boldsymbol{p'}\|^2 = 0 \\
\implies &\|\boldsymbol{p'}\|(\|\boldsymbol{g}\| \sin(\Delta + \phi) - k\|\boldsymbol{p'}\|) = 0 \\
\implies &\|\boldsymbol{g}\| \sin(\Delta + \phi) - k\|\boldsymbol{p'}\| = 0 \qquad \text{if } \|\boldsymbol{p'}\| \neq 0 \\
\implies &\|\boldsymbol{g}\| \sin(\Delta + \phi) = k\|\boldsymbol{p'}\| \\
\implies &k = \frac{\|\boldsymbol{g}\|}{\|\boldsymbol{p'}\|} \sin(\Delta + \phi)
\end{aligned}
$$

The same result also could be obtained by solving another optimization problem with k as the variable. Combining the formula for $\boldsymbol{p'}$ and k gives the formula for $\boldsymbol{g}^p$.

Moving forward, we'll assume a function $projectCone(\boldsymbol{g}, \boldsymbol{a}, \Delta)$ which returns the projection of $\boldsymbol{g}$ onto $C_{\boldsymbol{a}}^{\Delta}$ possibly handling $\boldsymbol{g} \in C_{\boldsymbol{a}}^{\Delta}$ and $\boldsymbol{g} \notin C_{\boldsymbol{a}}^{\Delta}$ cases separately.

4.1.6 Gradient and Directional Derivatives

The gradient of a function gives the direction and rate of the fastest increase from point p. Moreover, directional derivative of F at p along direction $\boldsymbol{u}$, i.e. $\frac{\partial F}{\partial \boldsymbol{u}}(p)$, can be computed as $\langle \boldsymbol{u}, \nabla F(p) \rangle$.

Intuitively, the directional derivatives give the rate of change $\nabla F(p)$ in the given direction. As can the dot product implies, this rate is the largest when the angle between $\boldsymbol{u}$ and ∇F is zero. In other words, the gradient gives the direction of the fastest increase.

Using directional derivatives, we can reason about how changes to p affect the value of F. For example, since the gradient has the fastest instantaneous rate of change, $F(p + \epsilon \frac{\nabla F(p)}{\|\nabla F(p)\|}) \geq F(p + \epsilon \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|}), \forall \boldsymbol{u} \in \mathbb{R}^n$ for sufficiently small ϵ.

Similarly, if $\angle \boldsymbol{u}, \nabla F(p) \leq \frac{\pi}{2}$, $F(p + \epsilon \boldsymbol{u}) \geq F(p)$ for sufficiently small ϵ. This can be confirmed by computing the directional derivative using $\langle \boldsymbol{u}, \nabla F(p) \rangle = \|u\| \|\nabla F(p)\| \cos \angle \boldsymbol{u}, \nabla F(p)$. In other words, using the directional derivatives, we can obtain a direction of non-decrease for sufficiently small step size.

4.1.7 Gradient Ascent Algorithms

Guiding the search using the gradient, which is known to give the direction of the fastest ascent, has been the crux of many iterative optimization algorithms, collectively can be called gradient ascent algorithms ([44]). The simplest and the most common of these algorithms, known as vanilla gradient descent, uses $\theta_n = \theta_{n-1} + \alpha \nabla F(\theta_{n-1})$ as its update rule. There are also more sophisticated methods like Adam optimizer ([45]) which utilizes the history of θ and $\nabla F(\theta)$ for faster convergence.These algorithms are guaranteed to generate a sequence $(\theta)_n$ that converges to a local maximum for F if it is used with a sufficiently small step size. This local maximum is also a global maximum if F is a concave function.

4.1.8 Thresholded Lexicographic Multi-Objective Optimization

A generic multi-objective optimization problem with K objectives and n parameters can be formulated as:

Given a function $F : A \to \mathbb{R}^K$ where $A \in \mathbb{R}^n$ and a comparison relation $\geq^c$ for value vectors in $\mathbb{R}^K$, find an element $\theta^* \in A$ such that $f(\theta^*) \geq^c f(\theta)$ for all $\theta \in A$.

Notice that when we have multiple objectives, the gradients will form a K-tuple, $G = (\nabla F_1, \nabla F_2, \cdots, \nabla F_K)$, where ∇F_i is the gradient of i^{th} component of F.

Different instantiations of the comparison relation lead to various multi-objective problem families. In the case of Lexicographic Multi-Objective Optimization, the comparison relation $>^c$, is defined as

$$\boldsymbol{v_1} >^c \boldsymbol{v_2} \iff \exists i < K \text{ s. t. } \forall j < i \ \ \boldsymbol{v_1}(j) \geq \boldsymbol{v_2}(j)) \\ \wedge \boldsymbol{v_1}(i) > \boldsymbol{v_2}(i))$$

In *Thresholded* Lexicographic Multi-Objective Optimization, a threshold vector $\tau \in \mathbb{R}^{K-1}$ is introduced to express the values after which the user does not care about improvements in that objective. This new comparison relation can be denoted by $>^{(c,\tau)}$ which is defined as:

$\mathbf{u} >^{\boldsymbol{\tau}} \mathbf{v}$ iff there exists $i \leq K$ such that:

- $\forall j < i$ we have $\mathbf{u_j} \geq \min(\mathbf{v_j}, \tau_j)$; and
  - if $i < K$ then $\min(\mathbf{u_i}, \tau_i) > \min(\mathbf{v_i}, \tau_i)$,
  - otherwise if $i = K$ then $\mathbf{u_i} > \mathbf{v_i}$.

The relation $\geq^{\boldsymbol{\tau}}$ is defined as $>^{\boldsymbol{\tau}} \vee =$.

Notice that this completely parallels the definition of LMDPs from Section 2.

4.2 Lexicographic Constrained Ascent Direction

In this section, we will show how the concepts introduced in Section 4.1 can be utilized to solve Thresholded Lexicographic Multi-Objective Optimization problems. We'll start by introducing a simpler special case of our algorithm and then move on to the generalized algorithm. Finally, we will show how this function operates on simple multiobjective problems in the experiment section.

4.2.1 Algorithm

Since thresholded lexicographic multi-objective optimization problems impose a strict importance order on the objectives and it is not known how many objectives can be satisfied simultaneously beforehand, a natural approach is to optimize the objectives one by one until they reach the threshold values. However, once an objective is satisfied, optimizing the next objective could have

a detrimental effect on the satisfied objective. This could even lead the previous objective to fail. While we can always go back to optimizing this failing objective, this would be inefficient, even worse, potentially leading to endless loops of switching between objectives.

However, we could limit our search for a satisfying point for the new objective to the directions not detrimental to already satisfied objectives by using our results about directional derivatives. For simplicity, assume that we have a primary objective F_1 which is satisfied at the current point θ_n and a secondary objective F_2 which we are trying to optimize next. $\nabla_{\boldsymbol{u}} F_1$, the change in F_1 along a direction $\boldsymbol{u}$, is equal to $\langle \boldsymbol{u}, \nabla F_1(\theta_n) \rangle = \|\boldsymbol{u}\| \|\nabla F_1(\theta_n)\| \cos(\angle \boldsymbol{u}, \nabla F_1(\theta_n))$, choosing a direction which makes an angle $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ with $\nabla F_1(\theta_n)$ would make the directional derivative non-negative. Therefore, updating θ as $\theta_{n+1} = \theta_n + \epsilon \boldsymbol{u}$ with an infinitesimal ϵ would not reduce the value of F_1. If $\nabla_{\boldsymbol{u}} F_2$ is positive, we can optimize F_2 without jeopardizing F_1. Note that the same logic hold even if we have k already satisfied objectives $F_1, \ldots, F_k$ and now optimizing F_{k+1} as long as $\forall i \leq k \nabla_{\boldsymbol{u}} F_i \geq 0$.

While any such $\boldsymbol{u}$ allows us to carefully optimize our new objective F_2, we should pick an $\boldsymbol{u}$ with maximum $\frac{\partial F_2}{\partial \boldsymbol{u}}(\theta_n)$ to optimize F_2 most efficiently. While we know that $\nabla F_2(\theta_n)$ has the maximum directional derivative, it may not satisfy our previous requirements. Instead, we can use the vector projection to find the $\boldsymbol{u}$ which minimizes $\|\boldsymbol{u} - \nabla F_2(\theta_n)\|$ under the constraint $\nabla_{\boldsymbol{u}} F_1 \geq 0$. Notice that non-negative directional derivative means that $\boldsymbol{u}$ lies on the positive half-space of $\nabla F_1(\theta_n)$, i.e. $\boldsymbol{u} \in S^+_{\boldsymbol{u}}$. So, projecting $\nabla F_2(\theta_n)$ onto $S^+_{\nabla F_1(\theta_n)}$ will give us the $\boldsymbol{u}$ which satisfies the requirement and is closest to $\nabla F_2 \theta_n$, i.e. has the largest directional derivative. As a special case, when ∇F_1 and ∇F_2 point in opposite directions, this projection will give a zero vector which means that we cannot optimize F_2 without sacrificing F_1. This point would be locally Pareto optimal. In general, iteratively projecting $\nabla F_{k+1}(\theta_n)$ on the positive halfspaces of $\nabla F_1(\theta_n), \ldots, \nabla F_k(\theta_n)$ gives the desired vector as long as the final vector satisfies the requirements. If it does not satisfy the requirements, this point can be called a locally Pareto optimal point.

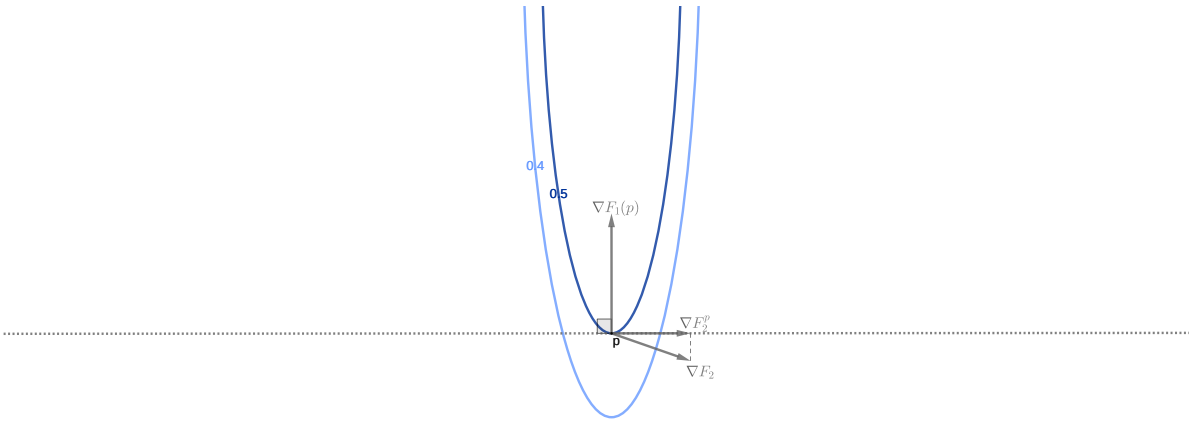

Figure 4.2: This figure shows why projecting onto the positive hyperspace is not always enough. The curves show the level curves of a function F_1. ∇F_2^p shows the projection of ∇F_2 on the positive halfspace of ∇F_1. Note that following ∇F_2^p reduces the function from 0.5 to 0.4.

While the approach above has the theoretical guarantees for the infinitely small step size, this does not translate to practice as the step sizes are not small enough. For example, Figure 4.2 shows how a direction that lies on the positive halfspace of the gradient can lead to a decrease. It can be also seen that unless the step size is infinitely small, this would always lead to a decrease. We can overcome this issue by generalizing halfspace to a hypercone for which the central angle is $\frac{\pi}{2} - \Delta$ where Δ is the hyperparameter of conservativeness. For $\Delta = 0$, this would be the halfspace case introduced above. Figure 4.3 shows how hypercone projection differs from halfspace projection and keeps the function above or at the current level for reasonably large step sizes.

Before giving the pseudocode of our algorithm, we will discuss a heuristic that prevents overly conservative solutions and lead to better performance in certain cases. When we make conservative updates that try to ensure that there is no decrease in the satisfied objectives, it leads to further increases on the already satisfied objectives instead of keeping them at the same level. This means most of the time, we have enough buffer between the current value of the satisfied objectives and their thresholds to sacrifice some of it for further gains in the currently optimized objective. Then, we can define a set of "active constraints" which is a subset of all satisfied objectives that we will not allow any sacrifice, and only consider these when projecting the gradient. The "ac-

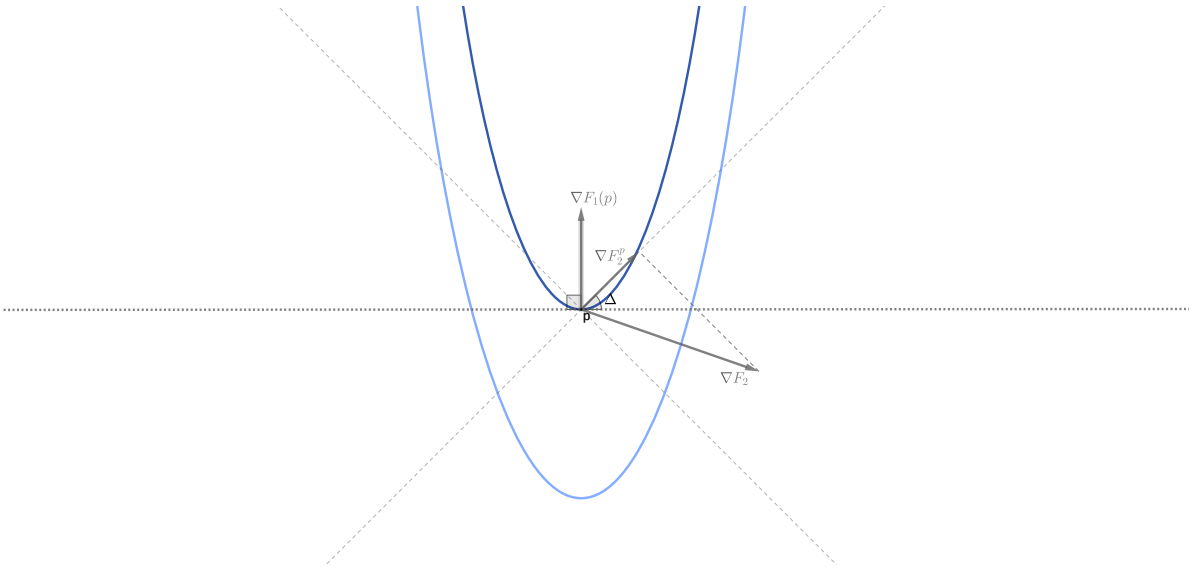

Figure 4.3: A visualization of cone projection. The dashed line shows the boundaries of the cone which are two lines in two dimensions. Notice that following ∇F_2^p keeps the function from at or above 0.5 unless a very large step size is chosen.

tive constraints" can be defined loosely, potentially allowing a hyperparameter that determines the minimum buffer zone needed to sacrifice from an objective.

How all of these ideas come together can be seen in the main component of our algorithm, `FindDirection` function (see Algorithm 4). This function takes the tuple of all gradients (M), the tuple of current function values ($F(\theta)$), threshold values (τ), the conservativeness hyperparameter (Δ), a boolean that determines whether "active constraints" heuristic will be used or not (AC), and a buffer value b to be used alongside active constraints heuristic as inputs. Then, it outputs the direction that should be followed at this step, which can replace the gradient in a gradient ascent algorithm. In this section, we will be using the vanilla gradient ascent algorithm described in Section 4.1.7.

Algorithm 4: Finding Lexicographic Constrained Ascent Direction

Function `FindDirection` $(M, F(\theta), \tau, \Delta, AC, b)$ **:**
  Initialize action-value function Q with random weights
  for $o = 1, K$ **do**
    if $o = K - 1$ **or** $F_o(\theta) < \tau_o$ **then**
      Initialize direction $\boldsymbol{u}$ with initial state M_o
      for $j = 1, o$ **do**
        if not *(*$(AC$ **and** $F_j(\theta) > \tau_j + b)$ **or** $\angle(\boldsymbol{u}, M_j) < \frac{\pi}{2} - \Delta)$ **then**
          $\boldsymbol{u} \leftarrow projectCone(\boldsymbol{u}, M_j, \Delta)$
      for $j = 1, o + 1$ **do**
        if not *(*$(j \neq K - 1$ **and** AC **and** $F_j(\theta) > \tau_j + b)$ **or** $\angle(\boldsymbol{u}, M_j) < \frac{\pi}{2} - \Delta)$ **then**
          return None
      return $\boldsymbol{u}$

Algorithm 4 finds the first objective that has not passed its threshold and iteratively projects its gradient onto hypercones of all previous objectives. If such a projection exists, it returns the projection as the "Lexicographic Constrained Ascent" direction. Otherwise, it returns null.

In our experiments, we will set $b = 0$. In general, b can be set to any non-negative value and higher values of b would result in a more conservative algorithm that does not sacrifice from an objective unless it is *well* above the threshold.

4.2.2 Experiments

In this section, we will present the results of our experiments for Lexicographic Projection Algorithm. As a benchmark, we will use $F_1(x, y) = -4x^2 + -y^2 + xy$ and $F_2(x, y) = -(x - 1)^2 - (y - 0.5)^2$ which are taken from [46]. We modified F_2 slightly for better visualization and multiplied both functions with -1 to convert this to a maximization problem. We set the threshold for F_1 to -0.5. The behavior of our cone algorithm without using active constraints heuristic on

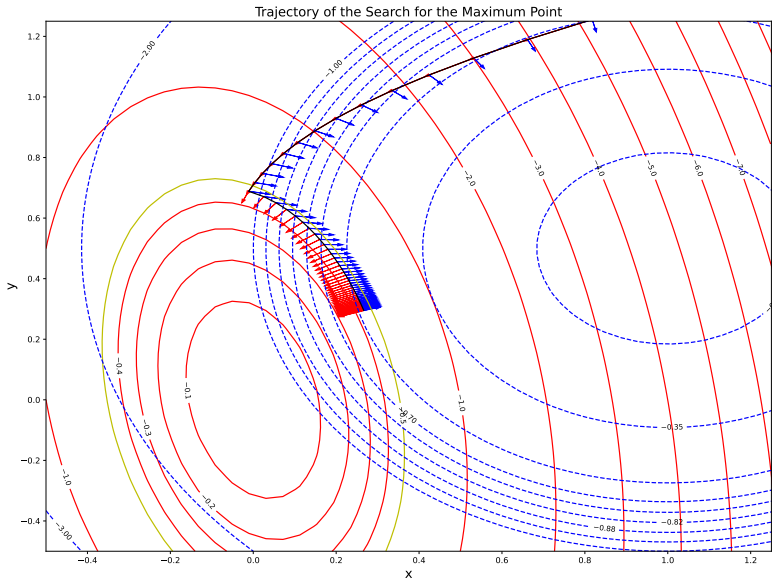

Figure 4.4: Behavior of the algorithm without the active constraints heuristic and hyperparameters $\alpha = 0.2$ and $\Delta = \frac{\pi}{90}$. The red and blue curves show the level curves of F_1 and F_2, respectively. The single yellow curve shows the threshold for F_1. The black line shows the trajectory of the solution, while the red and blue arrows show the gradients w.r.t. F_1 and F_2, respectively.

this problem with $\tau = (-0.5)$ and $\Delta = \frac{\pi}{90}$ can be seen in Figure 4.4. Notice that F_2, in blue, is completely ignored until the threshold for F_1 is reached. Then, the algorithm optimizes F_2 while respecting the passed threshold of F_1. This pattern also can be observed from the changes in the values of F_1 and F_2 in Figure 4.5.

We can squeeze in a little bit more gain in F_2 by using the active constraint heuristic. Figures 4.6 and 4.7 shows the behavior of the algorithm with $AC = True$ and $b = 0.01$. The small zig-zags are caused by sacrificing too much from F_1 so it goes below the threshold and the algorithm makes a correction.

4.3 Using Lexicographic Projection Algorithm in RL

In this section, we will finally close the circle by using the LPA introduced in Section 4.2 with the policy gradient algorithms introduced in Section 2.2.2 We will start by giving an overview

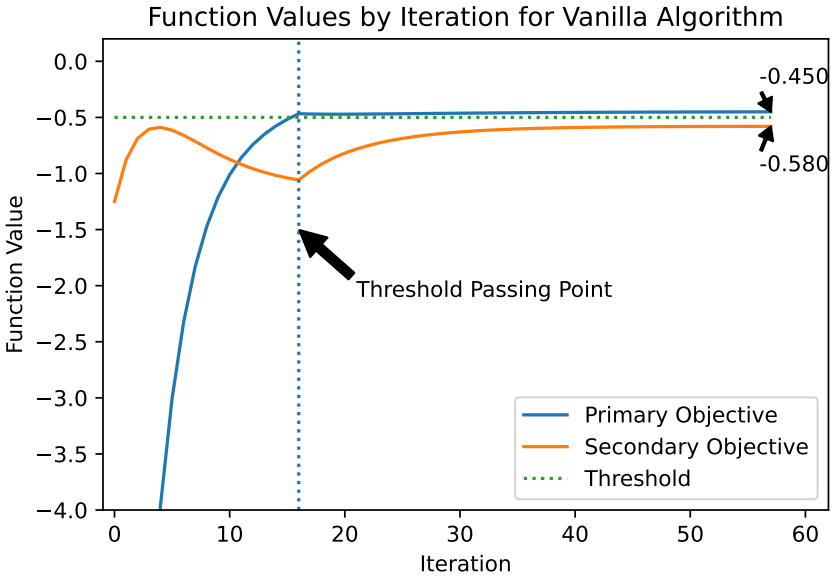

Figure 4.5: The changes in the function values for the experiment described in Figure 4.4.

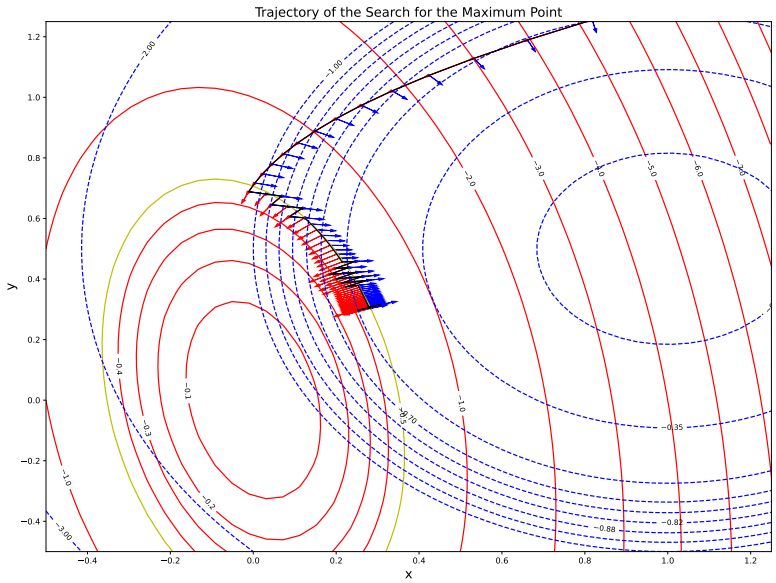

Figure 4.6: Behavior of the algorithm with the active constraints heuristic and $b = 0.01$. The rest of the hyperparameters are as described in Figure 4.4.

of our adaptation of REINFORCE algorithm ([35], [47]) that uses LPA to solve the objectıves lexicographically.

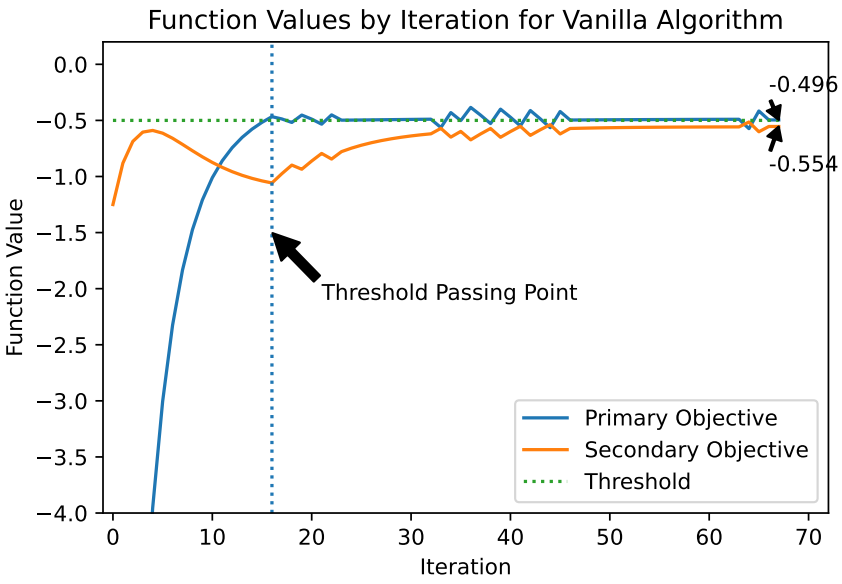

Figure 4.7: The changes in the function values for the experiment described in Figure 4.6.

4.3.1 REINFORCE Algorithm

REINFORCE is one of the simplest and most common policy gradient algorithms. While there are more up-to-date alternatives with better performance, the simplicity of REINFORCE makes it a perfect choice for conceptual work.

The update equation for REINFORCE can be derived from Equation 2.8 by replacing the weighted summations with expectations (see Section 13.3 of [31]). Then, we will use the collected trajectories to estimate these expectations. The idea is that we cannot compute these expectations analytically but we can sample from the same distribution and the mean of the sampled values is a good estimator for the expectation. Note that G_t is defined as $\sum_{t'=t}^{T-1} \gamma^t R_i(s_{t'}, a_{t'}, s_{t'+1})$, that is the discounted future reward for a timestep t of a trajectory sampled under π.

$$
\begin{aligned}
\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a Q_\pi(s,a) \nabla \pi(a|s,\theta) \\
&= \mathop{\mathbb{E}}_{\pi}\left[\sum_a Q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] && S_t \text{ is sampled under } \pi,\ S_t \sim \pi \\
&= \mathop{\mathbb{E}}_{\pi}\left[\sum_a \pi(a|S_t,\theta) Q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] && \text{multiplied and divided by } \pi(a|s,\theta) \\
&= \mathop{\mathbb{E}}_{\pi}\left[Q_\pi(S_t,A_t) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] && \text{replace } a \text{ by } A_t \sim \pi \\
&= \mathop{\mathbb{E}}_{\pi}\left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] && \mathop{\mathbb{E}}_{\pi}[G_t|S_t,A_t] = Q_\pi(S_t,A_t) \\
&= \mathop{\mathbb{E}}_{\pi}[G_t \nabla \ln \pi(A_t|S_t,\theta)] && \text{Log derivative trick: } \nabla \ln x = \frac{\nabla x}{x}
\end{aligned}
$$

Now, we can give the pseudocode for REINFORCE algorithm:

Algorithm 5: Vanilla REINFORCE

Process `REINFORCE`**:**

- Initialize policy function $\pi(a|s,\theta)$ with random parameter θ
- **for** *ep* $= 1, N_e$ **do**
  - Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ and save $\ln \pi(A_t|S_t)$ at every step.
  - $G_{T+1} \leftarrow 0$
  - **for** $t = T, 1$ **do**
    - $G_t \leftarrow R_t + \gamma G_{t+1}$
  - $L \leftarrow -\sum_{t=0,T-1} \ln \pi(A_t|S_t) G_{t+1}$
  - Update θ by taking an optimizer step for loss L
- **return** $\pi(a|s,\theta)$

Note that Algorithm 5 can be used with optimizers other than vanilla gradient descent. In our experiments, we found that Adam is easier to use with the tasks at hand.

4.3.2 Our Adaptation of REINFORCE

In this section, we will show how REINFORCE can be generalized to solve Lexicographic Markov Decision Processes. Our approach can be used to adapt other policy gradient algorithms too. Algorithm 6 shows the pseudocode for our algorithm. It can be seen that the only part that needs to change from the vanilla single-objective REINFORCE algorithm in Algorithm 5 is repeating the gradient computation for each objective and computing a new direction out of it.

Algorithm 6: Lexicographic REINFORCE

Process `REINFORCE` $(\tau, \Delta, AC, b, N_e)$ **:**
- Initialize policy function $\pi(a|s, \theta)$ with random parameter θ
- **for** $ep = 1, N_e$ **do**
  - Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ and save $\ln \pi(A_t|S_t)$ at every step.
  - $M \leftarrow \emptyset$
  - $F \leftarrow 0$
  - **for** $o = 1, K$ **do**
    - $G_{T+1} \leftarrow 0$
    - **for** $t = T, 1$ **do**
      - $G_t \leftarrow R_t + \gamma G_{t+1}$
      - $F_o \leftarrow F_o + R_t$
    - $L \leftarrow -\sum_{t=0,T-1} \ln \pi(A_t|S_t) G_{t+1}$
    - Compute the gradient of L with respect to θ and append it to M
  - $d =$ `FindDirection`$(M, F, \tau, \Delta, AC, b)$
  - Use d as the gradient for the optimizer step to update θ.
- **return** $\pi(a|s, \theta)$

Note that our algorithm is compatible with most policy gradient algorithms. [21] shows how a similar idea is applied to the actor-critic family of policy gradient algorithms which reduces the variance in the gradient estimation by using a *critic* network. We believe that more stable policy

gradient algorithms like actor-critic methods could further improve the performance of lexicographic projection approach as our algorithm might be sensitive to noise in gradient estimation.

4.3.3 Experiments

In this section, we will demonstrate the performance of Lexicographic REINFORCE algorithm on some Maze problems. We will give our results for two different experiments with different mazes and objectives. We will be using the same benchmark introduced in Section 3.4.2.

In both experiments, we use a two layer neural network ([48]) for policy function. We represent the state via one-hot encoding ([49]), hence the input dimension is the same as the size of the state space. For example, 20 for the maze in Figure 4.11. Then the hidden layer is a fully connected layer with 128 units and they use *ReLU* activation function [50]. We also used a dropout layer ([51]) with drop probability 0.6. Finally, the output layer has 4 units, representing the four valid actions in our benchmark. The outputs of these units are converted to action probabilities by applying a softmax function with temperature 10 [48]. The temperature hyperparameter allows making the policy less deterministic by making the action probabilities closer to each other. This was important to make sure that the policy keeps exploring so it does not get stuck in a local minimum.

Reachability Experiment

In the first experiment, we consider the case where the primary objective is a reachability objective, i.e. the agent gets a reward 1 when it reaches the terminal state and 0 everywhere else. The secondary, unconstrained, objective is avoiding the bad tiles which give -5 for H and -4 for h. We set $\tau_1 = 0.5$. We use the maze in Figure 4.8. As we only care about the agent eventually reaching the goal, the agent can completely avoid going on a bad tile. All the policies where it reaches the goal but goes through a bad tile in the process will be dominated by this policy. Hence, we will expect our agent to learn the policy where it eventually reaches the goal and never steps on a bad tile.

We run Algorithm 6 for $N_e = 4000$ episodes and we repeat our experiment with 10 different random seeds. As the policy we use is stochastic, different seeds give significantly different results.

```
MAZE
_____________
|___|G_|___|  4
|___|hh|hh|  3
|___|___|___|  2
|HH|HH|___|  1
|S_|___|___|  0
 0   1   2
```

Figure 4.8: The maze to be used in Section 4.3.3.

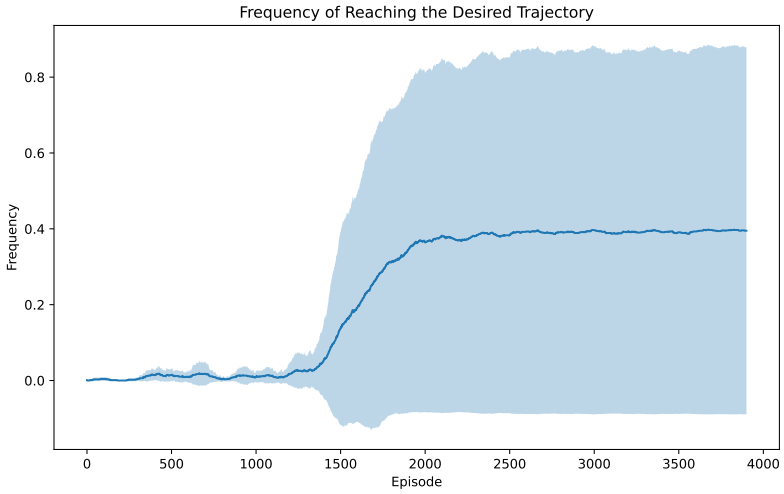

Figure 4.9: Average satisfaction frequency of 10 seeds for the experiment described in Section 4.3.3. The shaded region shows a confidence interval of two standard deviation width around the mean.

Figure 4.9 summarizes the performance of 10 seeds. The plot shows the ratio of the successful trajectories out of 100 trajectories where success is defined as satisfying the reachability constraint without stepping on a bad tile. The line shows the mean of 10 different seeds where the shaded region shows the variance in the experiment as two standard deviations around the mean. It can be clearly seen that as the training progresses, the satisfaction frequency increases. Out of the 10 seeds, 4 find policies that have 90% success over 100 episodes.

We can also take a closer look into how the training progresses for a successful seed. Figure 4.10 shows how the satisfaction frequency for each objective changes throughout the training. It can be seen that the primary objective, reaching the goal eventually starts with a high frequency but drops a little bit while the secondary is being learned. Then, the frequencies for both objectives

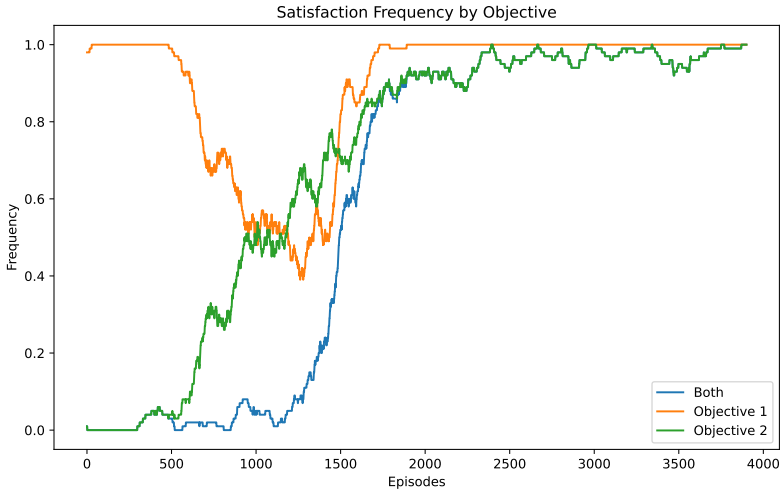

Figure 4.10: Satisfaction frequency for a single seed for the experiment described in Section 4.3.3.

start to increase together. Intuitively, the initial drop represents when the agent starts to consider "do nothing" policies which reduces the success of the primary objective. But the agent then learns that it can still maintain 0 penalties without just staying in place.

Non-reachability Experiment

In these experiments, we consider the primary objective to be a non-reachability objective, i.e. take non-zero values in some non-terminal states. For this, we will flip our objectives from Section 4.3.3. More specifically, our primary objective is to minimize the cost incurred from the bad tiles. HHs give -5 reward and hhs give -4 reward. A $+1$ reward is given in the terminal state to extend the primary objective to have rewards in both terminal and non-terminal states. The secondary objective is to minimize the time taken to the terminal state. We formalize this by defining our secondary reward function as 0 in the terminal state and -1 everywhere else.

This task has the Pareto front shown in Figure 4.12. Since the solutions marked as "Careful" and "Fast" can be obtained by ignoring the first and second objectives, respectively; we will attempt to learn the more interesting "Compromise" policy.

```
Extended Maze
_____________
|__|G_|__|__|  4
|__|hh|hh|hh|  3
|__|__|__|__|  2
|HH|HH|HH|__|  1
|S_|__|__|__|  0
 0  1  2  3
```

Figure 4.11: The Maze to be used for the experiments in Section 4.3.3

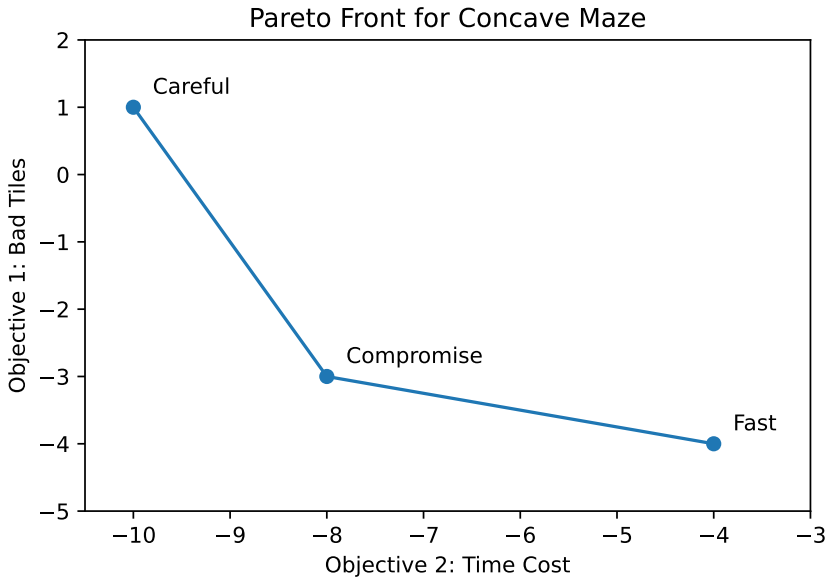

Figure 4.12: The pareto front for the concave matrix seen in Figure 4.11.

Following the analysis from Section 4.3.3, we first demonstrate the overall learning performance from 10 different random seeds in Figure 4.13. Out of the 10 seeds, 7 find policies that have 90% success over 100 episodes.

Then, the change in satisfaction frequencies of individual objectives for a successful seed can be seen in Figure 4.14. Notice that the primary objective initially starts very low and quickly increases while the secondary objective does the opposite. This part is while the algorithm mostly optimizes the primary objective. But once the primary objective is learned, the algorithm starts to learn the secondary objective.

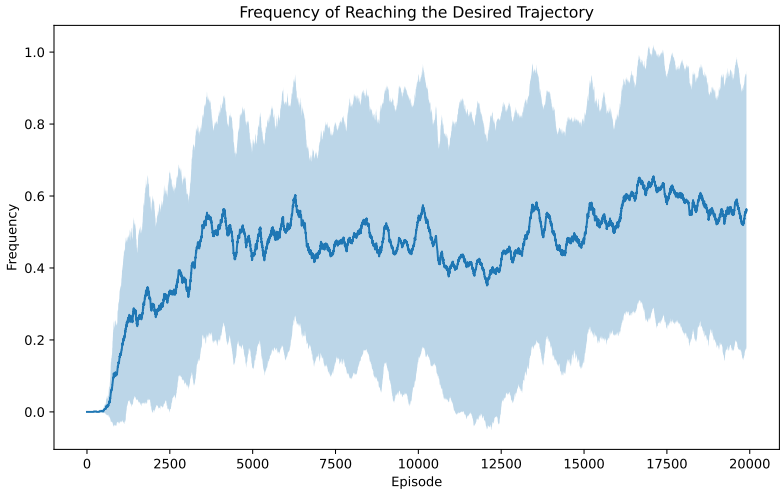

Figure 4.13: Average satisfaction frequency of 10 seeds for the experiment described in Section 4.3.3. The shaded region shows a confidence interval of two standard deviation width around the mean.

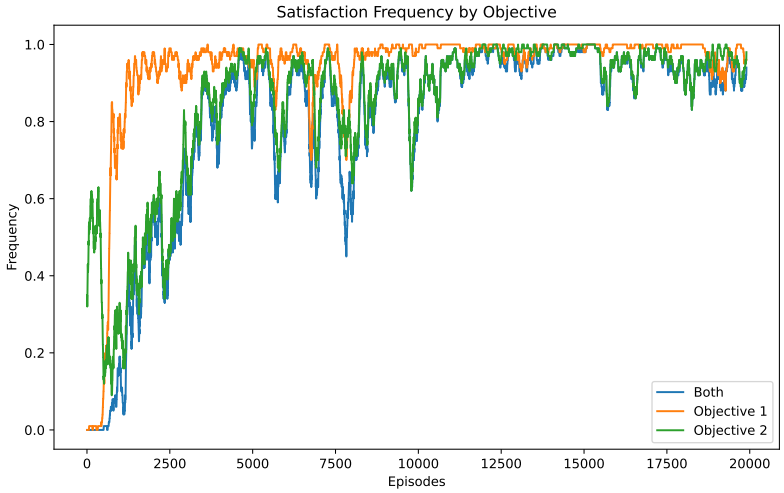

Figure 4.14: Satisfaction frequency for a single seed for the experiment described in Section 4.3.3.

These experiments illustrate the usefulness of projection-based policy gradient algorithm for different tasks. We believe that these results can be generalized to more complex tasks when our algorithm is combined with a more stable policy gradient algorithm.

Chapter 5

Conclusion

In this work, we considered the problem of solving multiobjective MDPs with thresholded lexicographic ordering. This subclass of MOMDPs, named Lexicographic MDPs (LMDP), is interesting for many real-life scenarios where a strict importance order exists between objectives. However, techniques to solve LMDPs have been scarce and their performance is poorly understood. All of them are based on value-function based algorithms family of RL, they are noted to lack any theoretical guarantees but the limits of their practical applicability are mostly still not acknowledged except for a few special cases.

In Chapter 3, we provided an example of a very common setting where TLQ fails, although it is not one of the previously acknowledged cases. Also, we gave a general shortcoming of a single threshold that does not depend on state space. Then, we proposed two solutions that still operate in the framework of value-function based algorithms that could address some of the shortcomings.

In Chapter 4, we took a different approach than the literature and proposed a policy-gradient based solution to solving LMDPs. We started with proposing a general lexicographic optimization framework using iterative projection of gradients and we showed how this can be combined with policy gradient algorithms to solve LMDPs.

We supported our analysis with experiments that show the usefulness of our algorithm. Firstly, we showed how our projection algorithm solves a simple optimization problem with two objectives represented by two polynomials. This empirically confirmed that during optimization of the secondary objective, which starts after the primary objective reaches the threshold, the algorithm is able to keep the primary objective above the threshold. We also showed that using the active constrained heuristic allows finding a better compromise by preventing updates from becoming unnecessarily conservative w.r.t. the primary objective.

Then, we demonstrated the effectiveness of combining our algorithm with REINFORCE, a simple policy gradient algorithm, on our Maze benchmark. We repeated this experiment with

different problem classes that are obtained by using different orderings of objectives and showed that our algorithm does not suffer from the shortcomings of existing methods.

We believe that further research on the empirical performance of our proposed algorithms could give more insight into their usefulness. Most importantly, we believe that combining our projection algorithm with more stable policy gradient algorithms could provide significant performance gain and extend the applicability of our approach to more complex tasks.

Bibliography

[1] Peter Vamplew, Benjamin J Smith, Johan Källström, Gabriel Ramos, Roxana Rădulescu, Diederik M Roijers, Conor F Hayes, Fredrik Heintz, Patrick Mannion, Pieter JK Libin, et al. Scalar reward is not enough: A response to silver, singh, precup and sutton (2021). *Autonomous Agents and Multi-Agent Systems*, 36(2):1–19, 2022.

[2] Eugene A Feinberg and Adam Shwartz. Markov decision models with weighted discounted criteria. *Mathematics of Operations Research*, 19(1):152–168, 1994.

[3] Patrice Perny and Paul Weng. On finding compromise solutions in multiobjective markov decision processes. In *ECAI 2010*, pages 969–970. IOS Press, 2010.

[4] Changjian Li and Krzysztof Czarnecki. Urban driving with multi-objective deep reinforcement learning. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 359–367, 2019.

[5] Johannes Fürnkranz, Eyke Hüllermeier, Weiwei Cheng, and Sang-Hyeun Park. Preference-based reinforcement learning: a formal framework and a policy iteration algorithm. *Machine learning*, 89(1):123–156, 2012.

[6] Krishnendu Chatterjee, Rupak Majumdar, and Thomas A Henzinger. Markov decision processes with multiple objectives. In *Annual symposium on theoretical aspects of computer science*, pages 325–336. Springer, 2006.

[7] Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707*, 2016.

[8] Runzhe Yang, Xingyuan Sun, and Karthik Narasimhan. A generalized algorithm for multi-objective reinforcement learning and policy adaptation. *Advances in Neural Information Processing Systems*, 32, 2019.

[9] Zoltán Gábor, Zsolt Kalmár, and Csaba Szepesvári. Multi-criteria reinforcement learning. In *ICML*, volume 98, pages 197–205. Citeseer, 1998.

[10] Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine learning*, 84(1):51–80, 2011.

[11] Diederik M Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48:67–113, 2013.

[12] Kyle Hollins Wray, Shlomo Zilberstein, and Abdel-Illah Mouaddib. Multi-objective mdps with conditional lexicographic reward preferences. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[13] Luis Enrique Pineda, Kyle Hollins Wray, and Shlomo Zilberstein. Revisiting multi-objective mdps with relaxed lexicographic preferences. In *2015 AAAI Fall Symposium Series*, 2015.

[14] Conor F Hayes, Enda Howley, and Patrick Mannion. Dynamic thresholded lexicograpic ordering. In *Adaptive and Learning Agents Workshop (AAMAS 2020)*, 2020.

[15] Jean-Antoine Désidéri. Multiple-Gradient Descent Algorithm (MGDA). Research Report RR-6953, INRIA, June 2009. In this report, the problem of minimizing simultaneously n smooth and unconstrained criteria is considered. A descent direction common to all the criteria is identified, knowing all the gradients. An algorithm is defined in which the optimization process is carried out in two phases : one that is cooperative yielding to the Pareto front, and the other optional and competitive.

[16] Ozan Sener and Vladlen Koltun. Multi-task learning as multi-objective optimization. *Advances in neural information processing systems*, 31, 2018.

[17] Xi Lin, Hui-Ling Zhen, Zhenhua Li, Qing-Fu Zhang, and Sam Kwong. Pareto multi-task learning. *Advances in neural information processing systems*, 32, 2019.

[18] Debabrata Mahapatra and Vaibhav Rajan. Multi-task learning with user preferences: Gradient descent with controlled ascent in pareto optimization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 6597–6607. PMLR, 13–18 Jul 2020.

[19] Simone Parisi, Matteo Pirotta, Nicola Smacchia, Luca Bascetta, and Marcello Restelli. Policy gradient approaches for multi-objective sequential decision making. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2323–2330. IEEE, 2014.

[20] Bo Liu, Xingchao Liu, Xiaojie Jin, Peter Stone, and Qiang Liu. Conflict-averse gradient descent for multi-task learning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 18878–18890. Curran Associates, Inc., 2021.

[21] Eiji Uchibe and Kenji Doya. Finding intrinsic rewards by embodied evolution and constrained reinforcement learning. *Neural Networks*, 21(10):1447–1455, 2008. ICONIP 2007.

[22] Akifumi Wachi and Yanan Sui. Safe reinforcement learning in constrained markov decision processes. In *International Conference on Machine Learning*, pages 9797–9806. PMLR, 2020.

[23] Javier García, Roberto Iglesias, Miguel A Rodríguez, and Carlos V Regueiro. Incremental reinforcement learning for multi-objective robotic tasks. *Knowledge and Information Systems*, 51(3):911–940, 2017.

[24] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. Safety-constrained reinforcement learning for mdps. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 130–146. Springer, 2016.

[25] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Model-free reinforcement learning for lexicographic omega-regular objec-

tives. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan, editors, *Formal Methods*, pages 142–159, Cham, 2021. Springer International Publishing.

[26] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Omega-regular objectives in model-free reinforcement learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 395–412. Springer, 2019.

[27] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[28] Anders Tolver. An introduction to markov chains. *Department of Mathematical Sciences, University of Copenhagen*, 2016.

[29] Eitan Altman. *Constrained Markov decision processes: stochastic modeling*. Routledge, 1999.

[30] Gilbert G Walter and Martha Contreras. Classification of markov chains. In *Compartmental Modeling with Networks*, pages 71–80. Springer, 1999.

[31] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[32] Dimitri P. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, Inc., USA, 1987.

[33] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[34] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[35] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[36] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[39] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[40] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

[41] Peter Geibel. Reinforcement learning for mdps with constraints. In *European Conference on Machine Learning*, pages 646–653. Springer, 2006.

[42] Edwin KP Chong and Stanislaw H Zak. *An introduction to optimization*. John Wiley & Sons, 2004.

[43] *Abstract Vectors*, chapter 5, pages 245–334. John Wiley & Sons, Ltd, 2021.

[44] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[45] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[46] Adrien Zerbinati, Jean-Antoine Desideri, and Régis Duvigneau. Comparison between MGDA and PAES for Multi-Objective Optimization. Research Report RR-7667, INRIA, June 2011.

[47] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[48] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[49] Sarah L Harris and David Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[50] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[51] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.