VRIJE UNIVERSITEIT BRUSSEL

Faculty of Science and Bio-Engineering Sciences
↪ Department of Computer Science
↪ Artificial Intelligence Laboratory

Controlling Large Scale Multi-Agent Environments with Model-Based Reinforcement Learning

Dissertation submitted in fulfillment of the requirements for the degree of
Doctor of Science: Computer Science

Eugenio Bargiacchi

Promotor: Prof. Dr. Ann Nowé (Vrije Universiteit Brussel)
Co-promotor: Dr. Diederik M. Roijers (Vrije Universiteit Brussel)

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel : +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 9789464948028
NUR: 984
THEMA: UYQM

Abstract

Multi-agent reinforcement learning (RL) offers the opportunity to autonomously learn how to best operate multiple actors, where one's actions may affect the operations of the others. The subfield of cooperative multi-agent learning is especially important, as it focuses on those domains where the actors need to achieve a joint task, irrespectively of their individual preferences. Thus, cooperative multi-agent RL has the potential to significantly increase the efficiency of real world settings such as traffic control, warehouse management, wind farm control and many others. One of the main challenges when learning in these scenarios is how to handle large numbers of agents, especially without requiring vast amount of data, which may be hard and expensive to gather. This is because, as the number of agents to control increases, the number of possible joint policies increases exponentially, making naive approaches impractical if not outright infeasible.

In this dissertation, we specifically tackle the aspects of scalability and sample-efficiency in cooperative multi-agent RL. We approach these problems by leveraging model-based methods, which allow us to simultaneously achieve two distinct goals: to incorporate prior domain knowledge about a given problem into the learning process, and extract as much information as possible out of each interaction with the real environment. In particular, we focus on domain knowledge in the form of *coordination graphs*, which contain locality information about the environment, i.e. about which agents can directly or indirectly interact together. These graphs allow for much more efficient learning, by *factorizing* the problem into smaller, simpler components. These components can then be brought back together by using specialized optimization algorithms specifically designed to work on factored domains.

Our main contributions consist in several novel multi-agent bandit algorithms for both regret minimization and best-arm identification, as well as a novel multi-agent MDP algorithm that can efficiently scale to settings with hundreds of agents. We additionally provide theoretical proofs for the bandit algorithms proving the validity of our approach, as well as comprehensive empirical tests for all our presented methods against a variety of state-of-the-art benchmarks. Finally, we present a new, extensive software library that contains free and open source implementations of many reinforcement learning algorithms, including those that are presented in this dissertation.

Samenvatting

Multi-agent reinforcement learning (RL) biedt de mogelijkheid om autonoom te leren hoe meerdere actoren het beste kunnen opereren, waarbij de acties van een van hen de operaties van de anderen kunnen beïnvloeden. Het deelgebied van coöperatief multi-agent leren is bijzonder belangrijk, omdat het zich richt op die domeinen waar de actoren een gezamenlijke taak moeten uitvoeren, ongeacht hun individuele voorkeuren. Zo heeft coöperatieve multi-agent RL het potentieel om de efficiëntie van reële settings zoals verkeerscontrole, magazijnbeheer, controle van windmolenparken en vele andere aanzienlijk te verhogen. Een van de belangrijkste uitdagingen bij het leren in deze scenario's is hoe om te gaan met grote aantallen agenten, vooral zonder een enorme hoeveelheid gegevens nodig te hebben, die moeilijk en duur te verzamelen kunnen zijn. Dit komt omdat, naarmate het aantal te controleren agenten toeneemt, het aantal mogelijke gezamenlijke beleidslijnen exponentieel toeneemt, waardoor naïeve benaderingen onpraktisch of zelfs onuitvoerbaar worden.

In dit proefschrift pakken we specifiek de aspecten schaalbaarheid en steekproefefficiëntie in coöperatieve multi-agent RL aan. We benaderen deze problemen door gebruik te maken van modelgebaseerde methoden, die ons in staat stellen om gelijktijdig twee verschillende doelen te bereiken: het incorporeren van eerdere domeinkennis over een gegeven probleem in het leerproces, en zoveel mogelijk informatie halen uit elke interactie met de echte omgeving. In het bijzonder richten wij ons op domeinkennis in de vorm van coördinatiegrafieken, die localiteitsinformatie bevatten over de omgeving, d.w.z. over welke agenten direct of indirect met elkaar kunnen interageren. Deze grafieken maken veel efficiënter leren mogelijk, door het probleem in kleinere, eenvoudiger componenten te verdelen. Deze componenten kunnen dan weer worden samengebracht met behulp van gespecialiseerde optimalisatie-algoritmen die speciaal zijn ontworpen om te werken op gefactoriseerde domeinen.

Onze belangrijkste bijdragen bestaan uit verschillende nieuwe multi-agent bandit algoritmen voor zowel spijtminimalisatie als best-arm identificatie, alsook een nieuw multi-agent MDP algoritme dat efficiënt kan worden opgeschaald naar settings met honderden agenten. Daarnaast leveren wij theoretische bewijzen voor de bandit algoritmen die de geldigheid van onze aanpak aantonen, alsmede uitgebreide empirische tests voor al onze gepresenteerde methoden tegen een verscheidenheid van state-of-the-art benchmarks. Tenslotte presenteren we een nieuwe, uitgebreide softwarebibliotheek die gratis en open source implementaties bevat van vele algoritmen voor het leren van versterking, inclusief die welke in dit proefschrift worden gepresenteerd.

Acknowledgments

This dissertation is the product of six long years spent working on a topic I have always loved, AI, which I had the privilege of studying with an unparalleled amount of freedom. While this manuscript is a testament of the research work I put in, it would not have been possible without the help and support of the very many people around me.

First and foremost, my special thanks to my promotor and co-promotor, prof. dr. Ann Nowé and dr. Diederik M. Roijers. Ann, thank you for believing in me so quickly and offering me the opportunity to become a researcher, and for your guidance in these past years. A million thanks to Diederik, who has supported me along this path the entire way, from being my supervisor during my Master at the UvA, to his recommendation that helped me start my PhD at the AI Lab in Brussels, as well as his friendship throughout all the ups and downs of my journey.

I would like to thank the members of my doctoral jury, prof. dr. Bart Bogaerts, prof. dr. Ann Dooms, prof. dr. Bas Ketsman, prof. dr. Jilles Steeve Dibangoye and prof. dr. Matthijs Spaan, for the time and effort they have dedicated to help me improve my work and ensure that this dissertation sees the light at its best.

I sincerely thank the Flemish research foundation (FWO) for sponsoring my work these past four years. Their PhD Fellowship funding granted me the freedom to pursue my goals in exactly the way I wished to.

Thanks to all the members of the AI Lab who have made (and continue to make) it a wonderful place to be, but also for making me a better researcher and person. Timo, thanks for the fun and for truly making me the Bayesian man I had never dreamed I could be. Roxana, thanks for being the hand that is always there when you are falling. Elias, thanks for the unending discussions and chess matches. I will miss dearly the DnD sessions with Yuri and Hèléne. Thanks Raphaël for your insights and positive energy. Mathieu, you have been my laughter for my entire stay and I can never thank you enough for it.

Thanks to my family for supporting me to where I am today. You posed the first stone to all my accomplishments. Thanks to Carlo, Feo and Jack, for being the ones on which I can always truly count. And finally, thanks to my Marta and my Ettore, for being the lights in my life without which there would be no purpose.

Contents

1 Introduction

Nearly all interesting behaviors we observe during our lives are a result of interactions between countless moving parts. Down from atoms and up to our globalized and interconnected world, we express our power over the environment through the direct control we apply to its constituent elements. And while on a local scale our main constraints are the physical laws of nature, in the grand scheme of things the much harder challenge consists in coordinating our assets across a sea of emergent behaviors. International shipping [1], power production [2], traffic control [3], warehouse management [4], market predictions [5], wind farm control [6], all depend on the synchronized behaviors of a multitude of actors. As economies grow larger, scaling up this organizational orchestra between so many actors becomes harder and harder.

1.1 Reinforcement Learning

In the last decade, advances in the field of *Machine Learning*, and more specifically of *Reinforcement Learning* (RL), have shown great promise in producing autonomous computer programs that can tackle complex problems with superhuman performance [7]. Taking inspiration from neuroscience, RL stakes the learning process upon the concepts of action and feedback, relying on direct experimentation to gather new information [8].

In its simplest form, RL consists in an *agent* taking actions in a given *environment*, and directly observing the actions' results as a *reward* signal. This feedback is then used to refine its behavior, or *policy*, as to maximize its accrued reward over time. This focus on trial-and-error to obtain the optimal policy for a given task is what separates RL from other similar fields. For example, in the RL sister field of *planning*, the agents are provided with a full model of their environment dynamics, e.g. the distributions of reward conditioned on the selected actions. In this case, all the required information to compute an optimal policy is — and must be — available from the start, so there is no need for exploration, and most of the computation is used before the agents finally act. However, such knowledge is generally hard to obtain in realistic settings, especially in large scale problems.

Instead, RL considers agents that must learn from the direct interaction with their environment. In this dissertation, we model an RL problem as a discrete iterative process, where at each *timestep* one or more agents perform *actions* within their *environment*. They then receive a *reward*, which provides a direct mean of feedback as to whether the actions had an immediate positive, neutral or negative effect. In the most common setting, the reward is stochastic, and its distribution is encoded through a *reward function*. The goal in general is to learn the *policy* that maximizes the reward accrued by the agents over the duration of the task at hand. This duration can be expressed either as a fixed time limit (an *horizon*), or as a *discount* factor that reduces the magnitude of future rewards. Thus, discounting rewards incentivize the agents to perform their tasks quickly rather than slowly. Depending on the setting, the agents' actions might or might not affect the *state* of the environment via an unknown *transition function*. The state acts as a context that influences the immediate consequences of the actions of the agents. This allows to model sequential decision making, as the agents must take into account the consequences of their actions beyond their immediate reward.

We are specifically interested in the cooperation of multiple agents, as many real world coordination problems can often be modeled as environments where agents have to cooperate in order to optimize a shared team reward [9]. In any case, it is common to use the more specific term *multi-agent* RL when dealing with problems with more than one decision maker, as to highlight the many challenges inherent to these settings. These can be, for example:

Interactions between agents

Tasks might reward agents independently for their actions, and in some settings a positive reward for an agent will even result in a penalty for another, e.g. zero-sum games. Learning reliably in such settings can require expertise from the field of Game Theory, to identify Nash/Pareto equilibria and ways for the agents' policies to converge to a common ground, even if overall behaviors will be individualistic [10]. On the other hand, a task might require the agents to be fully cooperative. In this case the agents are only concerned with maximizing their collectively accrued reward, irrespective of their individual performance. Nevertheless, computing an optimal joint policy cannot be done in general by simply allowing each agent to act greedily: it might be necessary for an agent to select actions that are individually suboptimal, but that increase the overall reward of the team [11].

Open population
The number of agents might unexpectedly change over time. Thus, agents must be able to take into account that new actors might join in — or leave — and disrupt any existing equilibria, which may require learning more defensive behavior [12, 13].

Heterogeneity of agents
The agents might not be of the same type; they might have different action sets and goals, or they might receive different types of observations if they are equipped with different sensors. They might experience different environmental dynamics if they use different actuators [14, 15].

Communication limits
If the agents cannot communicate directly, they will need to infer the policies of the other agents uniquely from their observable behaviors. Inversely, if the agents can communicate, there might be limits to their bandwidth. In this case, agents require additional learning time to determine which information is actually worth being shared. Direct communication might be restricted to agents in close proximity, and who can actually receive information might change over time. The now common deep learning paradigm of *centralized training, decentralized execution* assumes that full communication is available during training, but that agents must be able to act independently at execution time [16].

Learning convergence
One of the main assumptions in single-agent RL is that the environment, even if stochastic, is fixed, i.e. its dynamics do not change over time. In multi-agent RL, from the point of view of a single agent, this assumption is broken. As the other agents learn, their observable behavior changes, which results in a non-stationary environment. This can have several negative effects; for example policy oscillation, where each agent in turn modifies its policy in response to the new behavior of the others, and the learning never converges [10].

Each of these issues uniquely affects multi-agent systems; and although they all might be relevant for any given multi-agent problem, in practice only a subset of them applies at any given time. However, there is one challenge that is inherently common to all multi-agent settings: *scalability*. As the number of agents increases, the number of ways that the agents can jointly interact with the environment grows exponentially. While this statement can seem obvious, it has a profound impact on the learning process that traditional single-agent algorithms cannot easily handle. Optimal policies cannot be represented exactly in general [17, 18], as they would be too large to store. The amount of data required for the agents to fully experience their environment grows larger than what can be obtained realistically. Convergence issues are exhacerbated by the number of possible interactions, and by the forced use of approximate techniques which might not provide theoretical guarantees.

1.2 Focus

No matter the setting, scalability is one of the most fundamental issues related to multi-agent systems: having more agents always results in harder learning. In the multi-agent literature, most empirical evaluations still restrict multi-agent experiments to single-digit numbers at best [19], and only recently has focus shifted to tackling problems with more than a dozen agents — with notable exceptions being specialized models such as mean-field RL [20].

Here, our goal is to study whether it is possible to use RL to learn effective policies when having to coordinate hundreds of agents concurrently. Specifically, we aim to answer the following question:

Can we design algorithms that allow many agents to learn to cooperate in large scale environments, without requiring exponential amounts of experience?

To this end, the work presented in this dissertation relies on a specific set of assumptions designed to maintain the focus on the scalability aspect. These are:

- The number of agents is fixed, and so are the environment's dynamics.
- The agents actions and environment's dynamics are finite and discrete.
- The agents act in a *fully observable* environment. In other words, agents always have access to the true state of the world at any point in time.

- The agents are able to communicate between them without any constraints or costs — or equivalently they can freely communicate with some centralized hub.
- The agents share a *fully cooperative* objective. This means that all agents will strive to maximize their joint reward, rather than their individual performance.

Note that a cooperative setting with unbounded communication can be theoretically seen as equivalent to a factored single-agent setting [21]. While we do not explicitly limit the communication channels between agents, we maintain our focus on the multi-agent aspect by relying on message-passing algorithms (which are standard in the cooperative multi-agent literature) to ensure that agents can coordinate, rather than on the unbounded transmission of each invididual agent's perceptions to the others.

In addition, we focus our attention to the *model-based* subfield of RL. In model-based RL, the agents use the experience they gather to learn a *model* of the dynamics of the environment to speed up the discovery of the optimal policy. This is a well-known technique that features several advantages over competing approaches; specifically it makes it easy to embed prior knowledge in the learning process, and provides excellent sample-efficiency, i.e. it can extract more information from each datapoint, resulting in less experience required to learn. Both these properties are important in enabling efficient learning in otherwise intractably large environments.

Finally, we heavily focus our methods on the exploitation of locality properties present in many real-world domains — e.g. wind farms [22] or transport networks [3]. In other words, we leverage the fact that in many settings the agents interact with each other only on a local level, so that their interdependencies can be represented through a *coordination graph*. For example, in a wind farm, the orientation of a given wind turbines affects the power production of other turbines downstream, but is otherwise independent of the power production of the rest of the farm. Leveraging locality can significantly boost learning by simplifying the effective complexity of the environment, which in turns well synergizes with our chosen model-based approach.

1.3 Contributions

In this dissertation we present five novel algorithmic contributions for learning in large-scale multi-agent environments, as well as one software package contribution. All algorithms can leverage approximate maximization techniques which allow them to efficiently compute policies for hundreds of concurrent agents.

The MAUCE algorithm [11] is designed to efficiently maximize cumulative reward over time in stateless environments (multi-agent multi-armed bandits), i.e. have agents pick the joint actions which results in the highest expected cumulative reward at the end of a task. It uses a novel maximization technique which correctly takes into account the uncertainty present within the gathered experience. We describe it in detail in Section 3.2.

The MATS algorithm [23] manages uncertainty through Bayesian inference, which allows it to finely tune its exploration depending on what actions are actually likely to be optimal. MATS usage of Bayesian priors integrates well with the usage of domain knowledge, and demonstrates excellent computational performance. We describe it in detail in Section 3.3.

The MARmax and MAVmax algorithms [24] are designed for best-arm identification in multi-agent multi-armed settings, where agents only focus on exploration in order to provide the user with the best possible estimate of the optimal joint action. To our knowledge, they are the first algorithms to provide formal performance guarantees (PAC-bound) to their performance. We describe them in detail in Section 3.4.

The CPS algorithm [25] is designed for cumulative reward maximization in stateful environments (multi-agent Markov decision processes). It efficiently updates the agents' knowledge about the environment by detecting where current estimates are not up-to-date with the currently available information. Combined with our model-based approach, this allows CPS to learn close-to-optimal policies extremely fast, even in very large environments. We describe it in detail in Section 4.2.

Finally, we present a new software library, `AI-Toolbox` [26], that contains implementations of more than 40 reinforcement learning algorithms. This library has been used as a foundation for the research work presented in this thesis, and to run all empirical evaluations of the algorithms and their respective baselines. This toolbox is presented in Appendix A.

1.4 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 formally introduces the concept of model-based multi-agent RL, as well as the challenges that the learning agents face in this domain. It additionally introduces the concept of a coordination graph, and details several algorithms that can be used by agents to coordinate their actions.

- Chapter 3 formally introduces the multi-agent bandit domain; its goals and its challenges. Section 3.2 details our first contribution, the MAUCE algorithm, a sample efficient learning algorithm for regret minimization. Section 3.3 describes the MATS algorithm, a different approach that tackles the same setting as MAUCE. Section 3.4 details the MARmax and MAVmax algorithmic contributions, for best-arm identification in the multi-agent setting.

- Chapter 4 formally introduces multi-agent Markov decision processes, as well as the implications of sequential decision making in multi-agent settings. Section 4.2 details our last and main contribution, the CPS algorithm, a highly sample-efficient algorithm that can scale up to hundreds of concurrent learning agents. Section 4.3 describes preliminary work into extending our contributions to continuous state spaces.

- Chapter 5 summarizes our contributions, as well as discussing its limitations and several avenues for future work.

- Appendix A describes AI-Toolbox, the software library that practically implements all algorithms and empirical experiments described in this dissertation (with the exception of the QMIX algorithm).

2 | Background

Reinforcement Learning (RL) is a field of AI that studies ways to induce optimal behavior from one or more agents, using a reward feedback signal to guide the learning process. In this chapter we provide a formal description and introduction to the main concepts and challenges of model-based RL in a multi-agent cooperative setting, which will be referenced throughout this dissertation. We leave to later chapters more in-depth background regarding the specific settings tackled in our work.

2.1 Multi-Agent Reinforcement Learning

To best understand how to model a multi-agent problem, it is useful to start from the simpler single-agent perspective. A single-agent RL problem consists of two main components: the *agent*, and the *environment*. The first *acts* upon the second, and receives *reward* as a direct feedback to determine whether its behavior was good or bad. It then uses the feedback to update its *policy*. More precisely, in our case:

- We consider the interactions between the agent and the environment to occur in discrete *timesteps*.
- H is the *horizon* of the problem, i.e. the number of available timesteps for the agent to act in. Often the horizon is infinite; in those cases a *discount* γ may be applied on rewards to avoid infinite returns (see Section 4.1.1).

At each timestep t:

- The agent performs an action a_t selected from a finite set of actions $\mathcal{A}$, given the environment's current state s_t.
- The environment transitions from its current state s_t to a new one s_{t+1}, both belonging to a finite set of states $\mathcal{S}$. The new state is known by the agent.
- The agent receives feedback in the form of a scalar reward $r_t \in \mathbb{R}$

We consider an *episode* to be a single execution of the task from the first to the last timestep. Depending on the task, the agent may learn across several consecutive episodes, i.e. the knowledge gathered from each is kept into the next. The cumulative reward obtained during the episode is called the episode's *return*.

We formally model the environment as two functions that describe the dynamics of the learning system: a *transition function*, which describes how actions affect the state transitions, and a *reward function*, which describes how rewards are tied to actions. Note that typically these functions are not known to the agents in RL, although some structure may be known (see Section 2.2.1).

The transition function is a conditional discrete distribution in the form:

$$\mathcal{T}(s'|s,a) \rightarrow [0,1] \tag{2.1}$$

where $\mathcal{T}(s'|s,a)$ denotes the probability of transitioning to a given state s' assuming we performed action a in state s. The reward function is described as a stochastic function:

$$\mathcal{R}(s,a) \rightarrow \mathbb{R} \tag{2.2}$$

so that $\mathcal{R}(s,a)$ denotes the reward obtained when taking action a in state s, which is drawn from a fixed random distribution associated with that state-action pair.

In a similar way, the agent is formally modeled through its policy, which is a conditional discrete distribution π such that:

$$P_\pi(a|s) \rightarrow [0,1] \tag{2.3}$$

where $P_\pi(a|s)$ denotes the probability that the agent selects action a when acting in state s. After each action, the agent collects *experience* in the form of $\langle s,a,s',r \rangle$ tuples, which are used in some way — depending on the algorithm — to update its policy and improve its behavior.

We can now start to consider the problem under a multi-agent point of view. In this case, the basic structure of the problem remains the same, but instead of a single agent acting upon the environment, there are more than one. Thus, at each timestep each agent will select its individual action, and the environment will respond to the full joint action accordingly. We can naturally refer to some or all of these individual actions in vector form, which we refer to as a *joint action*. Similarly, it is often also helpful — though not necessarily required — to work with a more granular description of the environment's state as a vector composed of individual *state features*. Thus:

- $\mathcal{K}$ is the finite set of agents that act in the environment.
- $\boldsymbol{\mathcal{A}} = \mathcal{A}_1 \times ... \times \mathcal{A}_{|\mathcal{K}|}$ is the set of *full joint actions* available to the agents. A full joint action $\mathbf{a}$ is thus the collection of all the individual actions performed by the agents in the same timestep.
- $\mathcal{F}$ is the finite set of features that compose each state of the environment. To each feature f there is an associated finite set $\mathcal{S}_f$ which contains the possible values of that feature.
- $\boldsymbol{\mathcal{S}} = \mathcal{S}_1 \times ... \times \mathcal{S}_{|\mathcal{F}|}$ is the set of states of the environment. Note that it may be possible that not all states in this set are realistically reachable in the modeled environment; this however does not result in any practical implications and the spurious states can be simply ignored.

Note that we use bold letters to refer to vector variables and functions. We can now restate our transition and reward functions, together with the policy, using the new vector definitions for states and actions. Note that, because we are considering the problem to be fully cooperative, our reward function can still return a single scalar representing the overall reward for all agents.

- The transition function $\mathcal{T}(\mathbf{s}'|\mathbf{s},\mathbf{a}) \to [0,1]$ specifies the probability of transitioning to a state $\mathbf{s}'$ given a state $\mathbf{s}$ and full joint action $\mathbf{a}$ pair.
- The reward function $\mathcal{R}(\mathbf{s},\mathbf{a}) \to \mathbb{R}$ specifies the stochastic joint reward received when the full joint action $\mathbf{a}$ is taken in state $\mathbf{s}$.
- The policy $P_\pi(\mathbf{a}|\mathbf{s}) \to [0,1]$ specifies the probability of taking the full joint action $\mathbf{a}$ when the agents are in state $\mathbf{s}$.

2.2 The Curse of Dimensionality

A unique feature of a fully cooperative problem with unrestricted communication and full observability is that it can be naively converted to a single-agent setting.

In this new perspective, we have a super-agent that has, for each action, one of the many possible full joint actions of the original agents, while its environment's states are each a unique combination of the features of the original problem. As this conversion does not modify the overall dynamics of the problem, the single-agent framing can theoretically be used in place of the multi-agent one without modifying the value of any joint policy. A reasonable question is then whether we can simply apply single-agent RL algorithms to the newly converted problem, without the requirement to develop novel methods for the multi-agent setting.

Unfortunately, this approach is generally infeasible. The sizes of the state and action spaces in the single-agent perspective are exponential in the original dimensionality of the problem. Thus, any single-agent RL algorithm equipped by the super-agent will need to deal with state and action spaces far larger than what it was designed for. Over a certain size, even simply storing and processing the optimal policy, let alone a model of the environment, becomes impossible. The amount of data required for training also increases dramatically with larger problems, as the agents must experience a much larger number of possible concurrent events to know how to act. We refer to the issues deriving from this exponential rate of growth as the *curse of dimensionality*.

2.2.1 Sparse Interactions

The key insight in dealing with the curse of dimensionality is exploiting the *sparse interactions* between the agents [27]. In other words, we take advantage of the fact that, in real world problems, agents only directly affect their close surroundings, and only directly interact with their close neighbors. Note the deliberate use of the word "directly": we are expressly not ignoring the fact that the actions of an agent can, indirectly, affect the behavior of the system as a whole. Instead, we leverage the property of *locality* to *factorize* the problem, so that we work with interconnected, but individually smaller, local components, rather than with their combined cross-product in its totality. This can enormously simplify learning in larger settings: the number of possible full joint actions of 50 agents with 10 actions each is 10^{50}, while the total number of local joint actions if we consider all pairwise agent groupings is $\frac{1}{2}(50 \cdot 49) \cdot 10^2 \approx 10^5$. For this reason it is crucial to take advantage of any structure present in a multi-agent environment as to keep the joint coordination problem tractable [28], and indeed this type of approach has seen a large focus in the multi-agent planning literature [17, 21, 29, 30].

In practice, the decomposition of a problem into local components is applied directly on the transition and reward functions of the environment. In the first case, we want to describe the full transition function as a combination of smaller factors, each describing the transition behavior of individual state features:

$$\mathcal{T}(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) = \prod_f \mathcal{T}_f(s'_f \mid \mathbf{s}^f, \mathbf{a}^f) \tag{2.4}$$

where with $\mathbf{s}^f$ and $\mathbf{a}^f$ we denote the subsets of $\mathbf{s}$ and $\mathbf{a}$ that are relevant as the input of $\mathcal{T}_f$. In other words, because we are leveraging locality, it is very likely that the behavior of a single state feature — which $\mathcal{T}^f$ describes — does not depend on the entire environment, but only on those features and agents that are logically adjacent to it. Note that because each local transition function describes a valid conditional probability, they must be combined as a product, and not as a sum.

In the case of the reward function, we can decompose it into individual *local payoff functions*; each specifies a reward depending on a subset of agents and state features.

$$\mathcal{R}(\mathbf{s}, \mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{s}^e, \mathbf{a}^e) \tag{2.5}$$

Note how in this case we do not necessarily associate each local payoff function with a state feature; instead each is associated with an agent *group* e — the agents whose coordination is required to achieve its corresponding reward. We can see each individual $\mathcal{R}_e$ as a way to reward a specific set of agents for doing something well locally. Then, maximizing their sum $\mathcal{R}$ automatically leads to the policy that maximizes cooperative behavior, by selecting for each agent the action that most benefits the group. Note that this maximization is expressely *not* equal to the maximization of each $\mathcal{R}_e$ individually, i.e. the full joint action that achieves the global maximum is not guaranteed (and in fact, is generally not expected) to be locally optimal for any given group. This is due to the fact that these functions share inputs (the agents' actions), and so cannot be maximized indepedently, thus making the global maximum a shared compromise.

The exact factorization of a problem — i.e. which features and agents can be considered adjacent — is an intrinsic property of the environment, since it depends on the task that is being modeled. In this dissertation we assume that the factorization is always known at the start of learning, as a form of prior knowledge. At the same time, if needed, it is possible to learn it by statistical analysis of the experience data [31–33], and, in some cases, the learned model can be even more efficient than the true one [34]. Note that the factorization does not necessarily extend to other functions other than what we discussed; most notably the policy. Because the actions of a single agent can indirectly influence the actions taken by the others – even at a later timestep – the optimal policy cannot in most cases be factored so that agents can act independently. We discuss this issue more thoroughly in Section 4.

2.3 Model-Based Reinforcement Learning

Due to the curse of dimensionality, learning in large environments requires large amounts of data. In RL, this data is in the form of experience, which means that the agents would need to interact with the environment for a really long time — possibly longer than feasible — before being able to compute a reasonable policy. One approach to learning which specifically tackles this issue is called *model-based* RL.

Model-based RL is a learning technique which focuses the use of the collected experience to build a *model* of the environment, i.e. an approximation of the environment estimated through the data and prior knowledge. Every time new experience is gathered, the model is updated to reflect the new data. The advantage of this approach is that the model can be used to perform additional computations (*planning*) without having to directly interact with the environment. In this way, model-based RL aims to be *sample-efficient*, i.e. learn as much as possible with as little data as possible. Thus, model-based RL trades off additional computation with a reduced need for data [35]. As practical examples, Dyna-Q [36] and Dyna2 [37] use learned models to perform random backups on the value function that are used to speed up learning. Prioritized sweeping [38, 39] introduced the idea of sorting updates by priority to decrease the computational costs associated with random sampling. Although farther removed, the use of UCT in Monte-Carlo based methods can be seen as prioritizing state-action pairs based on trajectories extracted from a generative model [40].

In general, the learned model aims to represent the transition and reward functions exactly. While normally this would be infeasible due to the size of the environment, our leverage of sparse interactions make the factored forms of these functions tractable. This is because, rather than having to learn an exponentially complex function, we can learn its individual components, which is significantly less difficult.

The parameters of the transition and reward functions are usually extracted from the experience data using maximum likelihood estimates. For example, a specific transition probability is estimated as:

$$\hat{\mathcal{T}}_f(s'_f|\mathbf{s}^f, \mathbf{a}^f) = \frac{\sum_t I(\mathbf{s}^f \in \mathbf{s}_t, \mathbf{a}^f \in \mathbf{a}_t, s'_f \in \mathbf{s}'_t)}{\sum_t I(\mathbf{s}^f \in \mathbf{s}_t, \mathbf{a}^f \in \mathbf{a}_t)} \tag{2.6}$$

where with I we refer to the indicator function — equal to one when its arguments are true. In simple words, this is simply the number of times we witnessed a given transition divided by the number of times we witnessed its starting partial state-action pair. For the reward function, usually it is enough to learn the expected reward of a specific transition, so that we have:

$$\hat{\mathcal{R}}_e(\mathbf{s}^e, \mathbf{a}^e) = \frac{\sum_t r_t I(\mathbf{s}^e \in \mathbf{s}_t, \mathbf{a}^e \in \mathbf{a}_t)}{\sum_t I(\mathbf{s}^e \in \mathbf{s}_t, \mathbf{a}^e \in \mathbf{a}_t)} \tag{2.7}$$

which is equal to the simple statement that we estimate the expected reward using the empirical average reward observed after witnessing a specific partial state-action pair. While maximum likelihood is the most common approach when estimating a model, there exist other techniques which can take into account the current uncertainty to inform the rest of the learning process. An example would be maintaining a Bayesian posterior distribution for each estimated quantity, and using Thompson-sampling to sample a model from them [41]. A policy can then be computed that optimizes that model, so that the agents gather more information to improve their posteriors, and then sample a model again at a later time.

2.4 Coordination Graphs

The prior knowledge regarding the factorization of the problem is useful not only to represent a model efficiently, but because it contains the dependency relationships between the various features and agents in the environment. These are useful because we can use them to take into account how each agent's actions affect the rest of the environment, and by extension the learning process.

One of the ways that we can exploit these dependencies is to efficiently perform a critical step of multi-agent reinforcement learning: to determine the joint action that maximizes the cumulative value across all payoff functions. This maximization is required by the agents to be able to exploit the knowledge gathered about the local payoff functions. At the same time, if performed naively, this maximization is infeasible, as it requires a time linear in the size of the joint action space — which as we know is exponential in the number of agents.

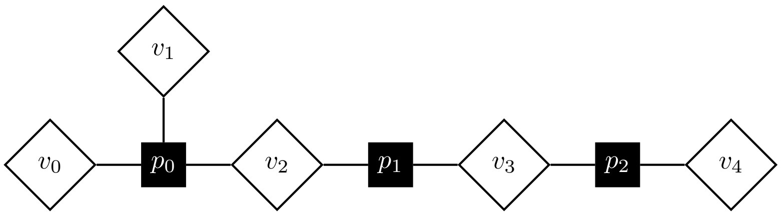


Figure 2.1: A simple coordination graph with five agent nodes and three payoff nodes. Given the shown edges, we can infer that the reward function is in the form $\mathcal{R}_0(a_0, a_1, a_2) + \mathcal{R}_1(a_2, a_3) + \mathcal{R}_2(a_3, a_4)$.

Fortunately, the knowledge of agent inter-dependencies allows us to significantly improve upon the naive approach. In particular, we can encode these dependencies into a mathematical tool called a *coordination graph*. A coordination graph is a factor graph, i.e. an undirected bipartite graph, which represents agents and their associated local payoff functions as two types of nodes, connected by pairwise edges. The coordination graph for a given problem be assessed by domain experts or learned from data [11, 17, 23, 31–33, 39, 42, 43]. Once obtained, a coordination graph can be used to efficiently execute optimal joint action selection strategies both in stateful and stateless environments, which makes it an invaluable tool when dealing with multi-agent problems.

Definition 1. *A coordination graph is a factor graph, i.e. a bipartite graph, where:*

- *$\mathcal{V}$ is a set of* variable nodes *(or* agent nodes*) each uniquely representing an agent.*
- *$\mathcal{P}$ is a set of* factor nodes *(or* payoff nodes*) each uniquely representing a local payoff function.*
- *An edge between a node $v \in \mathcal{V}$ and a node $p \in \mathcal{P}$ exists if and only if the agent represented by v participates in the domain of the payoff function represented by p.*

The problem of determining the joint action that maximizes the global payoff is also known as the *Distributed Constraint Optimization Problem* (DCOP) [44]. The naive way approach to DCOP problems is to simply enumerate all possible full joint actions, compute the corresponding total reward for each, and select the highest valued one. Due to the curse of dimensionality, this approach is infeasible, as it would have a computational cost exponential in the number of agents. Instead, we can exploit the information contained in the coordination graph, regarding which agents are directly dependent on each other, to improve significantly on this naive approach. There exist many algorithms that can perform this maximization efficiently, with different trade-off degrees between compute cost and accuracy of the solution [44]. Three well-known algorithms, which we use later in the thesis, are called Variable Elimination (also known as DPOP or non-serial dynamic programming) [45–47], Max-Plus (also known as Max-Sum or Max-Product in inference settings [45, 48, 49]) and Local Search [50]. Since computing the optimal joint-action of many agents is very important for learning coordination problems, we now give a through description of each.

2.4.1 Variable Elimination

Variable Elimination (VE) [45, 47] is an algorithm to efficiently determine the values of all variable nodes of a coordination graph to maximize the joint value of all factor nodes. The pseudocode for VE is shown in Algorithm 1.

The key insight exploited by VE is that each agent can only influence the rewards obtained from the payoff nodes it is connected to. Thus, its optimal action only depends on the actions selected by its direct neighbors, and not on the actions of all agents. In other words, given an agent k in the graph, if we knew in advance the actions of its neighboring agents it would be easy to determine k's optimal action — its *best response*. Unfortunately, we do not know this beforehand, but VE is not hindered by this detail. Instead, VE pre-computes *all possible best responses* of agent k for *all action combinations of its neighbors*. In this way, independently of the rest of the graph, VE can preemptively maximize k's action. Once this is done, the agent serves no purpose in the graph anymore, and can be removed, and the maximizing/removal process repeated again. Each time an agent is removed, VE replaces all its adjacent factor nodes with a single new factor node connecting all its neighboring agents. This new node will contain the appropriate cross-sum of the replaced factor nodes, with the removed agent maximized.

It is easier to understand this process if we visualize the single removal step of an arbitrary agent k from a graph $\mathcal{G}$ (see Figure 2.2a). Let's refer to agent's k factor node neighbors as $\mathbf{p}^k$, and to its agent neighbors as $\mathbf{v}^k$ (more precisely the direct agent neighbors of $\mathbf{p}^k$, excluding k). We know that each factor node $p \in \mathbf{p}^k$ represents a payoff function that takes as parameter the actions of its adjacent agent nodes and returns an expected reward, i.e.:

$$p(a_k, \mathbf{a}^{\mathbf{v}^k}) \to \mathbb{R} \tag{2.8}$$

Algorithm 1: Variable Elimination

```
Input: A factor graph $\mathcal{G}$, an agent selection heuristic $\mathcal{H}$
Output: The full joint action that maximizes the total return in $\mathcal{G}$
 1: For each factor node $p \in \mathcal{G}$, create an associated best response function $b_p(\cdot) \leftarrow \emptyset$
 2: while there are agents remaining in $\mathcal{G}$ do
 3:     Select agent $k$ from all remaining agents using heuristic $\mathcal{H}$
 4:     Let $\mathbf{v}^k$ be the neighboring agent nodes of $k$
 5:     Let $\mathbf{p}^k$ be the neighboring factor nodes of $k$
 6:     Create a new factor node $\pi$ connecting only the neighbor agents $\mathbf{v}^k$
 7:     for each local joint action $\mathbf{a}^{\mathbf{v}^k}$ of the neighbor agents do
 8:         Compute agent $k$'s best response: $\overset{\star}{a}_k \leftarrow \arg\max_{a_k} \sum_{\mathbf{p}_k} p(a_k, \mathbf{a}^{\mathbf{v}^k})$
 9:         Set $\pi(\mathbf{a}^{\mathbf{v}^k}) \leftarrow \sum_{\mathbf{p}_k} p(\overset{\star}{a}_k, \mathbf{a}^{\mathbf{v}^k})$
10:         Set $b_\pi(\mathbf{a}^{\mathbf{v}^k}) \leftarrow \left\{\overset{\star}{a}_k\right\} \cup \left(\bigcup_{\mathbf{p}_k} b_p(\overset{\star}{a}_k, \mathbf{a}^{\mathbf{v}^k})\right)$
11:     end for
12:     Remove all factors $\mathbf{p}^k$ from $\mathcal{G}$
13:     Remove agent $k$ from $\mathcal{G}$
14:     Add $\pi$ to $\mathcal{G}$ as $p_{\mathbf{v}^k}$
15: end while
16: return $b_\emptyset()$
```

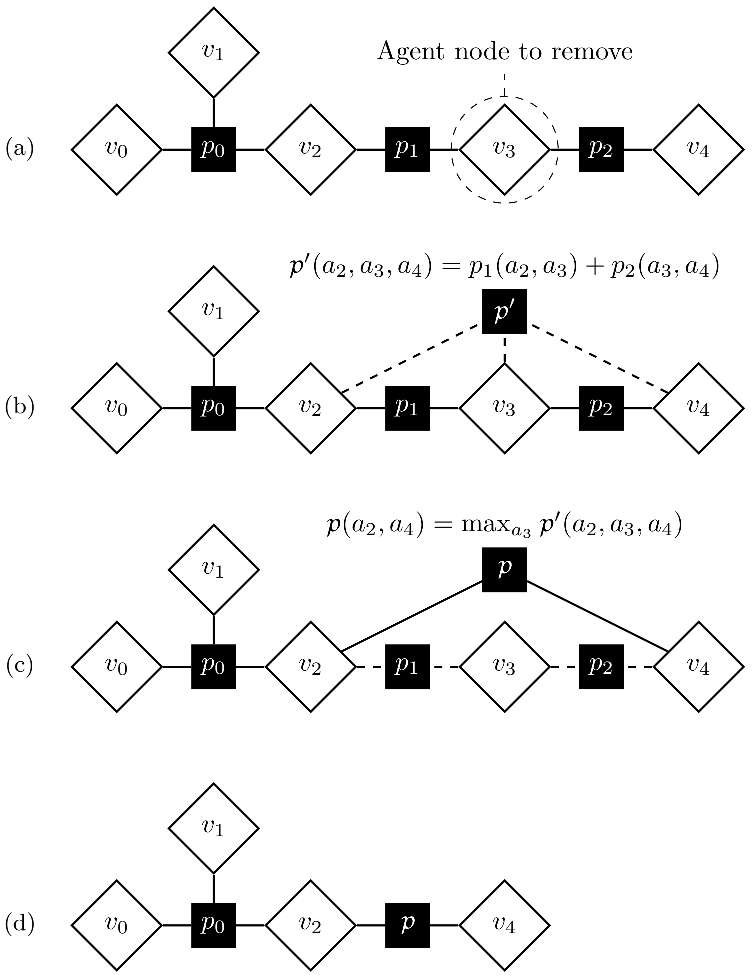
Note that Equation 2.8 contains a slight abuse of notation: while each p always depends on a_k (by definition), it likely does not depend on all the actions of the agents in $\mathbf{v}^k$, but only on a subset of them. Still, even if we refer to the actions of all neighbors, keep in mind that only the actions of those agents actually connected to p matter and contribute to its dimensionality.

Now, it is straightforward to define a function p', connected to both agent k and all of $\mathbf{v}^k$, that can replace all $\mathbf{p}^k$ without altering the value of any joint action (see Figure 2.2b):

$$p'(a_k, \mathbf{a}^{\mathbf{v}^k}) = \sum_{\mathbf{p}^k} p(a_k, \mathbf{a}^{\mathbf{v}^k}) \tag{2.9}$$

Note that computing p' likely requires a larger space footprint than what the combined $\mathbf{p}^k$ need, due to its larger dimensionality. Nevertheless, the factor graph $\mathcal{G}_{p'}$ that contains p' in place of $\mathbf{p}^k$ is functionally equivalent to $\mathcal{G}$, because the values of all possible full joint actions are unchanged.

Given p', we can now define a new function p, which simply maximizes over



(a)
v_1
Agent node to remove
v_0
p_0
v_2
p_1
v_3
p_2
v_4
(b)
$p'(a_2, a_3, a_4) = p_1(a_2, a_3) + p_2(a_3, a_4)$
p'
(c)
$p(a_2, a_4) = \max_{a_3} p'(a_2, a_3, a_4)$
p
(d)

Figure 2.2: A single step in the VE agent elimination process. (a) The agent node v_3 is selected for elimination. (b) A new payoff node p' is created, equivalent to the cross-sum of the payoff nodes adjacent to v_3: p_1 and p_2. (c) The payoff node p is created, which maximizes p' over the actions of v_3 (and thus does not depend on it anymore). (d) p replaces the now obsolete payoff nodes p_1 and p_2, as well as the agent v_3.

agent k (see Figure 2.2c):

$$p(\mathbf{a}^{\mathbf{v}^k}) = \max_{a_k} p'(a_k, \mathbf{a}^{\mathbf{v}^k}) \tag{2.10}$$

This new function p is what is used by VE to reduce the number of agents in the graph: replacing p' with p and removing agent k creates a new graph $\mathcal{G}_p$ that is once again equivalent to the $\mathcal{G}$ — all joint actions have their value unchanged — with the only exception that agent k is missing (see Figure 2.2d). Once we finish to maximize $\mathcal{G}_p$, we can determine which action agent k should do by recording all its best responses used to compute p:

$$b_p(\mathbf{a}^{\mathbf{v}^k}) = \arg\max_{a_k} p'(a_k, \mathbf{a}^{\mathbf{v}^k}) \tag{2.11}$$

and selecting the one that matches the joint action that maximizes $\mathcal{G}_p$.

We can iteratively repeat this removal process, until a single agent remains, attached to a single factor node. At this point, its optimal action can be trivially determined, and from it the optimal actions of all other agents can be recovered from the various b functions. Algorithm 1 implements a slightly more efficient recovery method, which stores in each b entry all previously computed relevant best responses [51]. This is equivalent, but avoids the "backward" pass to reconstruct all optimal actions.

While a naive search would require a time that is exponential in the number of agents, VE requires a time that is exponential only in the largest clique formed during its runtime. This cost is due to the loop between line 7 and 11 in Algorithm 1, which must iterate across all local joint actions of each clique present in the graph to compute all best responses and construct the new factor node p.

At the same time, the size of the cliques encountered by VE depends on the exact ordering in which the agents are removed. This follows because at each agent's removal we are creating a clique that links all its neighbors $\mathbf{v}^k$ — which might not have been fully connected before. Removing an agent that is included in two separate large cliques will create a new clique exponentially larger than both. Therefore, the order in which VE removes agents can have a significant effect in its runtime cost, especially when dealing with large graphs. Unfortunately, determining the optimal agent removal order to minimize the size of the largest clique is an NP-hard problem [45]. Thus, VE usually also employs some sort of heuristic to determine which agent to remove at each iteration, e.g. the agent with the fewest neighbors [45].

Algorithm 2: Max-Plus

```
Input: A factor graph $\mathcal{G}$, a number of iterations $N$
Output: The full joint action that maximizes the total return in $\mathcal{G}$
 1: Set the current iteration $n \leftarrow 0$
 2: Initialize all messages $m_p$ and $m_k$ to 0
 3: while $n < N$ do
 4:     for each factor node $p \in \mathcal{G}$ do
 5:         Compute message $m_p \leftarrow p + \sum m_{k \to p}$
 6:         for each agent node $k$ neighbor of $p$ do
 7:             Compute message $m_{p \to k} \leftarrow m_p - m_{k \to p}$
 8:         end for
 9:     end for
10:     for each agent node $k \in \mathcal{G}$ do
11:         Compute message $m_k \leftarrow \sum_p m_{p \to k}$
12:         for each factor node $p$ neighbor of $k$ do
13:             Compute message $m_{k \to p} \leftarrow m_k - m_{p \to k}$
14:         end for
15:         Select action $a_k$ which maximizes $m_k$
16:     end for
17: end while
18: return Current action selection $\mathbf{a}$
```

2.4.2 Max-Plus

The Max-Plus algorithm [48] is a computationally cheaper alternative to Variable Elimination, which can still provide the guarantee of identifying the correct optimal full joint action in certain situations. The pseudocode for Max-Plus is shown in Algorithm 2.

Max-Plus belongs to a family of optimization routines called *message passing* algorithms. These algorithms operate by passing information between adjacent nodes of the graph as *messages*, with the idea that once a node has received messages originating from the entire graph, it has effectively gathered all the information it needs to compute its own maximization independently. In Max-Plus, this allows to pass a number of relatively simple messages equal to some multiple of the node count, and then have each agent separately determine its optimal action.

At each iteration, Max-Plus performs two interlocking steps. In the first step,

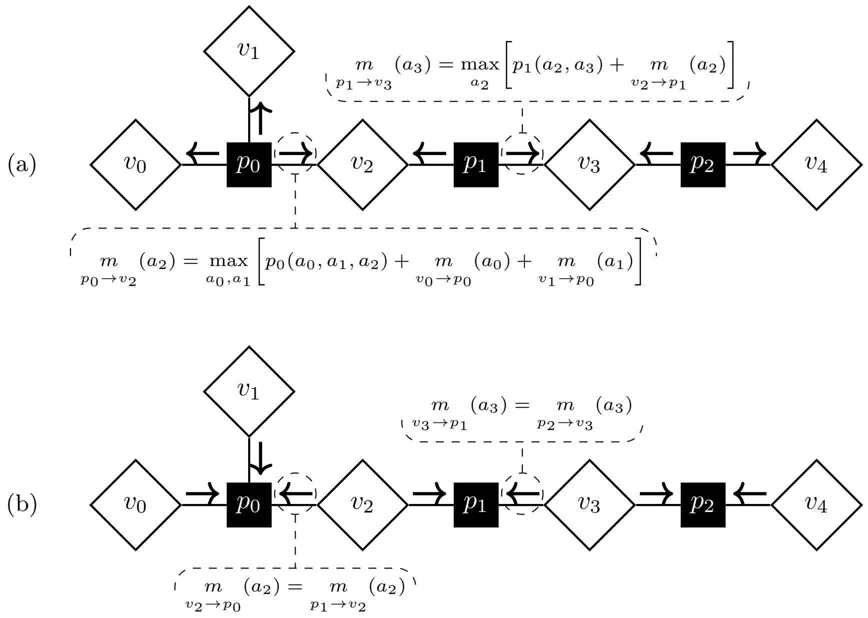

(a)
v_1
$\underset{p_1 \to v_3}{m}(a_3) = \max_{a_2} \left[p_1(a_2, a_3) + \underset{v_2 \to p_1}{m}(a_2) \right]$
v_0 p_0 v_2 p_1 v_3 p_2 v_4
$\underset{p_0 \to v_2}{m}(a_2) = \max_{a_0, a_1} \left[p_0(a_0, a_1, a_2) + \underset{v_0 \to p_0}{m}(a_0) + \underset{v_1 \to p_0}{m}(a_1) \right]$
(b)
v_1
$\underset{v_3 \to p_1}{m}(a_3) = \underset{p_2 \to v_3}{m}(a_3)$
v_0 p_0 v_2 p_1 v_3 p_2 v_4
$\underset{v_2 \to p_0}{m}(a_2) = \underset{p_1 \to v_2}{m}(a_2)$

Figure 2.3: A single step in the Max-Plus message passing process. (a) During the first phase, each factor node p sends a message to its adjacent agent nodes. Each message to an agent node v is equal to the sum of the payoff function of p plus all messages that p received from its neighbors, minus v. This message then maximizes over the actions of all agents other than v. (b) During the second phase, the agent nodes send messages to their adjacent factor nodes. Here the messages are simply the sums of all messages received by the agent node, aside from the one received to the receiver. Note how the dimensionality of the messages is always equal to the action space of the incoming/receiving agent node.

Max-Plus computes the messages that need to be sent from the factor nodes to the agent nodes in the coordination graph. During the second step, these messages are used to compute the outgoing messages from the agent nodes to the factor nodes. These are in turn used in the first step again as the process repeats, until convergence or for a fixed number of iterations. The computations for the two message types are very similar. For the messages generated by the factor nodes, each node first constructs a general message by cross-summing all incoming messages from adjacent agents, together with the function represented by the node itself (see Figure 2.3a). This general message is then tuned individually for each connected agent, by subtracting the original message received by the receiving agent. The messages generated by the agent nodes are constructed similarly, but during the generation of the general message there is no function to add, so the cross-sum is performed with only the incoming messages (see Figure 2.3b).

Max-Plus is guaranteed to be correct if the input graph $\mathcal{G}$ does not contain cycles. Without cycles, each node is guaranteed to receive information from the other nodes exactly once (provided enough iterations are performed), which makes each agent's estimates of the value of its actions exact. A minor complication arises if the optimal full joint action is not unique (i.e. there exists another full joint action with the same total value); in that case Max-Plus requires an additional coordination mechanism that traverses the graph carrying information about which maximum is being selected globally.

If $\mathcal{G}$ contains cycles, Max-Plus unfortunately offers no convergence guarantees. The reason is that, as messages are passed around the graph, they will reach the same node multiple times, which can often induce large value oscillations in the agent nodes. This in turn prevents them from having correct value estimates, which prevents an exact maximization. In practice, normalizing the message values can damp these oscillations, allowing Max-Plus to work quite well in practice. However, if the optimal full joint action is not unique in a graph containing cycles, then there exists no effective way to correctly coordinate the agents to converge to one of the optima; in these cases Max-Plus is best avoided.

2.4.3 Local Search

The Local Search (LS) algorithm [50] is a fairly simple heuristic to determine the optimal joint action, belonging to the class of coordinate descent algorithms [52]. The core idea of this class of algorithms is to maximize a multi-dimensional function by maximizing one dimension at a time while fixing the others in place. Coordinate descent algorithms are especially powerful when maximizing factored functions, due to the decreased correlations between dimensions. At the same time, while they are guaranteed to converge to some local optima, they cannot generally guarantee to find the optimal full joint action. The pseudocode for LS is shown in Algorithm 3.

LS begins from an input full joint action (usually generated from a uniform distribution), and repeatedly iterates over the agents in a random order. For each agent, LS selects its individual action that maximizes the value of the full joint

Algorithm 3: Local Search

```
Input: A factor graph $\mathcal{G}$, an initial full joint action $\mathbf{a}$
Output: A full joint action that approximately maximizes the total return in $\mathcal{G}$
1: repeat
2:     Set $\mathbf{b} \leftarrow \mathbf{a}$
3:     Generate a random ordering $\mathbf{o}$ of all agents
4:     for each agent $k$ in $\mathbf{o}$ do
5:         Set $a_k \leftarrow \arg\max_{a_k} \sum_{p \in \mathcal{G}} p(a_k, \mathbf{a}^{\neq k})$
6:     end for
7: until $\mathbf{b} = \mathbf{a}$
8: return $\mathbf{a}$
```

action while keeping the other agents fixed. LS does this until no agent can change its action and improve the global value. The reason why LS iterates over agents randomly is to prevent the systematical optimization of some agents before others, which might result in LS becoming stuck sooner inside some local (sub)optimum.

The main advantage of LS is its running time: the algorithm is quite cheap computationally, so it can be used to select a joint action even when the coordination graph is large and fairly dense. As with other coordinate descent algorithms, it is possible to mitigate the lack of guarantees by running LS multiple times from different initial random joint actions, or randomly perturbing the currently found best in the hope of escaping the current local optimum. These changes result in the Reusing Iterative Local Search (RILS) algorithm [50].

3 Multi-Agent Multi-Armed Bandits

One of the simplest types of problem in the reinforcement learning literature, the *multi-armed bandit* (MAB) [8] describes a setting where the environment has no state; or, equivalently, where it has a single never-changing state. Thus, a MAB problem considers the setting where the agent must determine the optimal action, with the only assumption that each action returns a payoff drawn stochastically from a fixed unknown distribution. The name multi-armed bandit derives from the term *one-armed bandit*, meaning slot machines, due to the similarity of the agent to that of a gambler pulling a lever and hoping for a payoff.

MABs embody some of the most quintessential RL challenges — e.g. exploration versus exploitation, sample efficiency — and their relative simplicity compared with other RL problems make them one of the most thoroughly studied frameworks in RL. In fact, many MAB algorithms can provide theoretical guarantees for their performance, which makes them an attractive option for real world settings: bandits are used to model choice problems like advertisement auctions, clinical trial design, A/B testing and fund allocation.

A multi-agent multi-armed bandit (MAMAB) extends the concept of a MAB to multiple agents. Here, each agent is associated with an individual set of arms. The agents then receive a joint reward once all have concurrently pulled an arm from their associated set. As a practical example, we might associate each agent with a particular feature of a trial design. In order to run the trial, we need to select an option for each feature; once this is done the trial is run, and a reward observed.

For a practical example of a cooperative multi-agent bandit task, consider an autonomously controlled wind farm in which the agents individually represent wind turbines that are able to adjust the alignment of their blades to the wind. Turbines can maximize their own power output by aligning the blades exactly perpendicular to the wind, but doing so may hinder turbines that are behind it due to turbulence [22]. If we consider the wind direction and intensity as static environmental variables, the problem is otherwise stateless and as such can be considered a bandit. In this case the goal of the agents would be to coordinate as to maximize the overall power production of the wind farm, regardless of the individual production of each turbine.

As we mentioned in Section 2.2, a multi-agent problem can be trivially converted to a single-agent problem by creating a super-agent with an action set equal to the cross-product of all the action sets of the original agents. When the agents' scalar reward happens to be produced by a singular global source whose output depends on the entire full-joint action at once, this conversion is forced; the MAMAB in this case truly is equivalent to its super-agent MAB counterpart, with no meaningful ways to make the problem more tractable. Instead, in this chapter we focus on those MAMABs where the reward can be factored in observable local components, each depending on a subset of the agent population. We show how leveraging locality allows us to exploit this factorization, both for computational performance and to make learning much more sample-efficient than in the super-agent case.

3.1 Background

We now formally define the multi-armed bandit problem, first from the perspective of a single agent, and later from the context of multiple cooperative agents. Later, we describe in detail two important types of optimization problems found in bandit settings: regret minimization, and best-arm identification. In the first case, the agents must maximize the reward they accrue over time. Thus, the agents must balance the exploitation of their knowledge — to select the currently known best action — with additional exploration to ensure that no better options are available [53]. This is known as the *exploration versus exploitation* trade-off. In the best-arm identification setting, the agents must identify the best joint action available, either within a certain accuracy [54], or subject to a constrained timestep budget [55]. In this case it does not matter what actions the agents select during the learning process, as long as the arm is correctly identified at the end.

3.1.1 Single-Agent Bandits

We can formally describe a single-agent multi-armed bandit following the same terminology we introduced in Section 2.1.

Definition 2. *A* **single-agent multi-armed bandit** *(MAB) is a tuple $\langle \mathcal{A}, \mathcal{R} \rangle$, where:*

- *$\mathcal{A}$ is the finite set of available actions to the agent.*
- *$\mathcal{R}(a) \to \mathbb{R}$ is the reward function, representing a stochastic reward drawn independently from an unknown reward distribution $R(a)$.*

At each timestep t, the agent selects an action $k_t \in \mathcal{A}$, which results in the agent obtaining a finite scalar reward $\mathcal{R}(a_t) \sim R(a_t)$. These rewards are the sole source of information for the agent to determine the moments of the underlying distributions of $\mathcal{R}$. The expected values $\mu(a) = \mathbb{E}[\mathcal{R}(a)]$ are of particular interest, as the optimal action in a bandit is $\overset{\star}{a} = \arg\max_a \mu(a)$.

The maximum number of timesteps that the agent can act in is either fixed — the number is then called the *horizon* — or we consider an infinite task. The set of all timesteps that the agent acts in is called an *episode*. It is worth noting that the term episode has a slightly different connotation in multi-armed bandits than the one it has in stateful settings. In a stateful setting, an episode is a set of timesteps delimited by some starting and ending states; in that case the learning process lasts several episodes, as an agent must complete a task several times in order to learn effectively (see Section 4). However, in a bandit, *the* episode is always singular, and is essentially a synonym for the task that needs to be solved, as without an ending state the agent will keep pulling arms until it reaches its horizon. In this case, multiple episodes are usually only discussed when evaluating an algorithm empirically, by averaging results over multiple independent runs.

3.1.2 Multi-Agent Bandits

In the multi-agent version of a MAB (a MAMAB), the core action-reward loop remains unchanged from the single-agent case. At the same time, we must now model the specific relationships between the multiple agents, and how they affect the obtained reward. We do this by defining *groups*: subsets of agents which jointly influence a part of the reward obtained. Generally a group will indicate agents that are either spatially or logically associated together during the task — going back to the concept of locality introduced in Section 2.2.1. Each group is then associated with its *local payoff function*, which returns a stochastic reward that depends on the local joint action of the group. The full reward for the task, which for a MAMAB is always considered cooperative, is then the sum of all local payoffs.

Definition 3. *A* **multi-agent multi-armed bandit** *(MAMAB) is a tuple* $\langle \mathcal{K}, \boldsymbol{\mathcal{A}}, \mathcal{E}, \mathcal{R} \rangle$*, where:*

- $\mathcal{K}$ *is the finite set of agents.*
- $\boldsymbol{\mathcal{A}} = \mathcal{A}_1 \times ... \times \mathcal{A}_{|\mathcal{K}|}$ *is the set of* full joint actions *available to the agents. A full joint action* $\mathbf{a}$ *is thus the collection of all the individual actions performed by the agents in the same timestep.*
- $\mathcal{E}$ *is the finite set of agent groups. Each group* $e \subset \mathcal{K}$ *is an unique subset of all agents (overlaps are possible).* $\mathcal{E}$ *contains a specific group only when a part of the problem specifically depends on the actions of all its agents simultaneously.*
- $\mathcal{R}(\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{a}^e)$ *is the reward function, which is decomposed as a sum of the local payoff functions* $\mathcal{R}_e(\mathbf{a}^e) \rightarrow \mathbb{R}$*.*

Our MAMAB representation is distantly related to combinatorial bandits (CB) [56–59], in which sets of arms can be pulled simultaneously, and, more specifically, to the *semi-bandit* variant of the problem [60], where the local components of the global reward are observable. In our setting, these sets correspond to full joint actions, and similarly to the CB framework, the action space grows exponentially with the dimensionality of the problem. At the same time, there are significant differences: in CB, the arms that are not pulled have no effect on the accrued reward, while in our case all agents must perform an action — which is not necessarily binary — and contribute to the final payoff. Additionally, we consider local rewards to always be drawn stochastically from fixed distributions, rather than dependent on a vector loss selected by an adversary. Finally, we assume that an underlying coordination graph is available, which is often a more reasonable assumption in multi-agent settings than what is required by other approaches (for example, [57] assumes access to an (α, β)-oracle that provides a joint action that outputs an α fraction of the optimal expected reward with a certain probability β).

A way to easily visualize the challenge of learning in a MAMAB is to consider that, because each local payoff function behaves identically to a single-agent MAB's reward function — with the one notable difference that its input is a set of actions rather than a single one — we can view a MAMAB as an interlocked set of single-agent bandits. Indeed, each group e and associated local payoff function $\mathcal{R}_e$ describes a MAB with an action set equal to the cross-product of the action sets of the relevant agents. The key detail is that in the MAMAB each individual agent selects its action only once, but in doing so affects the local joint actions of multiple groups, and thus multiple local payoff functions, at the same time. Thus, the main challenge in a MAMAB is that the actions of each agent can directly and indirectly affect the choices of all the others. When learning all agents must determine the full joint action that best gathers information and rewards for all local sub-bandits.

3.1.3 Regret Minimization

We can now begin to discuss the goals of the agents in a MAB. Because these goals are the same for both MABs and MAMABs, we will refer to the single-agent scenario only for simplicity.

The most common metric that is optimized for in the literature is called *regret*. The best arm of a MAB is, by definition, the one with the highest expected reward. Conversely, all other actions have lower (or equal) expected rewards. We call the regret of an action the difference between the expected reward of the best arm minus the expected reward of the arm associated with that action. Thus, selecting the best arms incurs in zero regret, while selecting any other arm — ignoring ties — will incur in a positive regret. We can define regret more formally as a function ρ such that:

$$\rho(a) = \mu(\overset{\star}{a}) - \mu(a) \tag{3.1}$$

In the task of *regret minimization* [61], the agent is tasked to minimize its expected regret during an episode:

Definition 4. *The* **expected cumulative regret** *of pulling a sequence of arms for timestep $t = 1$ to the horizon T (following the definition of [62]), is*

$$\mathbb{E}\left[\sum_{t=1}^{T} \mu(\overset{\star}{a}) - \mu(a_t)\right],$$

where a_t is the arm pulled at time t.

The main difficulty in doing so is that, at the start, it has no knowledge about the distributions of rewards associated with each action. Thus, the agent has to balance the two opposite needs of gathering new information versus minimizing its regret. In the first case, the agent wants to improve its estimates of the expected rewards of each arm, even though it might have to pull sub-optimal arms to do so. In the latter case, the agents wants to reduce its regret by pulling the arm that it considers best, even though other arms might actually be better. This duality is referred to as the *exploration versus exploitation* trade-off.

The metric of regret has the nice property that it is always positive, with the optimal policy, by definition, always obtaining zero regret. It is also deterministic, because it is computed from the expected means of the arms rather than from the actual stochastic rewards received from the environment. These properties allow the regret to be easily normalized, so to set a uniform goal across different bandits even when the environments are significantly different. Unfortunately, the true regret of an action can be measured only if its true expected reward is known. For this reason, regret is in practice used only in experimental settings where this information is known by construction and in theoretical proofs. When learning outside of this scope, we can achieve the same goal as regret minimization by instead measuring and maximizing the empirical cumulative reward of a policy $\sum_t \mathcal{R}(a_t)$.

3.1.4 Best-Arm Identification

Differently from regret minimization, the setting of *best-arm identification* (BAI) consists in identifying the bandit's highest mean arm without concern to the actual rewards obtained. In other words, it doesn't matter which arms are selected during the course of the episode, as long as by the end the agent can provide a good informed guess on what the optimal arm is.

The agent subject to a BAI task is generally bound by a set of constraints — it can't just keep pulling arms forever until absolute certainty is in sight. The two main sets of constraints that are considered in the literature are *fixed budget* and *fixed confidence* [63]. In the first case, the episode has a fixed horizon — the budget — that the agent is allowed to act in. At the end, the agent will need to recommend the arm that it estimates has the highest mean reward [55]. In the latter case, the horizon is unbounded. Instead, the agent is provided with a factor ε and a bound δ. It then must act in the MAB until it can recommend an arm that has a true expected reward of at least $1 - \varepsilon$ of the true optimal, with probability at least $1 - \delta$. It must also do so in as few timesteps as possible [54].

In BAI, the goal is generally achieved by focusing exploration on all arms that have a chance of being the optimal. Once an arm is considered unlikely to have a chance to be the best, it is disregarded. Thus, in a BAI setting the best arm will be identified much sooner than in the equivalent regret minimization problem; at the same time the accumulated reward will be much lower (although in BAI this does not matter).

Note that while the two sub-settings — fixed budget and fixed confidence — of BAI appear similar, they are not actually equivalent; generally theoretical proofs about the performance of an algorithm that hold for one of them do not hold for the other. Sometimes an algorithm can be adapted to work well in both instances by slightly tweaking its behavior depending on the situation [63], but sometimes this is not possible (e.g. fixed-budget algorithms generally rely on knowing their initial budget, which does not exist in the fixed-confidence setting).

3.2 Multi-Agent Upper-Confidence Exploration

We now describe our first main contribution, the *Multi-Agent Upper-Confidence Exploration* (MAUCE) algorithm [11]. MAUCE was one of the first algorithms to tackle the problem of regret minimization in factored MAMABs. This section is the result of joint work together with Timothy Verstraeten. Aside from our joint contributions to the algorithm itself, I focused on the efficient implementation of the algorithm and generation of the empirical results, while Timothy took the lead in the development of the theoretical regret bound proof.

MAUCE balances the agents' tradeoff between exploration and exploitation using local estimates and local upper bounds for each of the local joint actions of the MAMAB's groups. At each timestep, these estimates and bounds are combined in order to determine the full joint action that has the highest total upper bound on its expected reward; this action is the one that is executed by the agents. This maximization step is performed using a specifically designed VE variant called *upper confidence variable elimination* (UCVE), which was designed taking inspiration from the RL multi-objective literature. UCVE is able to correctly combine the local estimates without overexploring, which allows MAUCE to be much more frugal in its exploration than alternative approaches.

We additionally prove in Section 3.2.4 that MAUCE achieves a regret bound that depends on the *harmonic mean* of the local upper confidence bounds, rather than their sum, as we would get by applying the combinatorial bandit framework [56, 57]. This leads to a regret logarithmic in the number of arm-pulls and linear in the number of agents. In contrast, the naive approach of considering the full joint action is results in a regret exponential in the number of agents. In Section 3.2.5 we compare empirically the performance of MAUCE to other approaches from the literature, and show that it achieves much less regret in various settings, including wind farm control.

3.2.1 Upper-Confidence Bound

The exploration versus exploitation dilemma requires an autonomous agent to not only maintain estimates of the expected reward of its actions, but also to be able to manage the estimates' uncertainty. In general, an arm with a higher stochasticity in its returns will require more exploration for the agent to be able to trust its current mean estimate. One common strategy to approach this problem consists in associating each such estimate with an *upper-confidence bound* (UCB) [64], which represents the highest likely expected reward of an arm given the current data. As more data is gathered, the lower the UCB becomes, until it essentially matches the expected reward estimate. The agent then always selects the action with the highest current UCB, ensuring that if an arm has a chance of being the best, it is selected (and thus both explored and exploited), while ignoring arms that clearly offer sub-optimal returns.

The standard mechanism of UCB thus assigns to each action an upper bound as:

$$\text{UCB}_t(a) = \hat{\mu}(a) + c_t(a) \tag{3.2}$$

The additional term $c_t(a)$ is a function called the *exploration bonus*, which expresses the uncertainty still present in the current estimate. An often used specific implementation of the UCB mechanism, UCB1 [64], considers the exploration bonus for an action to be equal to:

$$c_t(a) = \sqrt{\frac{2 \ln t}{n_t(a)}} \tag{3.3}$$

where $n_t(a)$ is the number of times arm a has been pulled before timestep t. Here the exploration bonus is dependent on the percentage of times that a local arm has been selected, but does not depend on its estimated mean nor variance — nor other moments — of the data gathered. This allows the UCB1 strategy to be fairly robust in very disparate settings as it is essentially immune from odd edge-cases (e.g. an arm which nearly always returns some specific return, but sporadically returns a very high or low payoff); at the same time with the disadvantage of not being able to take advantage of arms with a noticeably low variance, which can slightly worsen its accrued regret.

3.2.2 The Algorithm

The core mechanism of the MAUCE algorithm relies on the UCB strategy to manage exploration. The key difference between MAUCE and algorithms that do not exploit the factorization inherent in a MAMAB is that MAUCE can keep pull counts for each local joint action separately, which allows it to significantly refine its exploration bonus with respect to Equation 3.3. MAUCE still executes at each timestep the full joint action $\mathring{\mathbf{a}}$ that maximizes the total estimated mean reward, $\hat{\mu}(\mathring{\mathbf{a}})$, plus the full exploration bonus, $c_t(\mathring{\mathbf{a}})$ (as in Equation 3.2), but these final quantities are computed using the intermediate estimates for the local joint actions of the MAMAB.

Algorithm 4: MAUCE

```
Input: A MAMAB with factored reward function $\mathcal{R}(\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{a}^e)$
       A time horizon $H$
 1: Initialize $\hat{\mu}_e(\mathbf{a}^e)$ and $n_e(\mathbf{a}^e)$ to zero.
 2: for $t = 1$ to $H$ do
 3:     Set $\mathbf{a} \leftarrow \arg\max_{\mathbf{a}} \hat{\mu}(\mathbf{a}) + c_t(\mathbf{a})$ where,
 4:         $\hat{\mu}(\mathbf{a}) = \sum_{e \in \mathcal{E}} \hat{\mu}_e(\mathbf{a}^e)$ and,
 5:         $c_t(\mathbf{a}) = \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)\sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}}$
 6:     Execute $\mathbf{a}$ and receive local rewards $r_e \leftarrow \mathcal{R}_e(\mathbf{a}^e)$
 7:     for $e \in \mathcal{E}$ do
 8:         Update $\hat{\mu}_e(\mathbf{a}^e)$ using $r_e$
 9:         $n_e(\mathbf{a}^e) \leftarrow n_e(\mathbf{a}^e) + 1$
10:     end for
11: end for
```

Specifically, MAUCE keeps mean estimates of local rewards $\hat{\mu}_e(\mathbf{a}^e)$, and local counts $n_e(\mathbf{a}^e)$ for each agent group. The estimated mean of a full joint action is computed, unremarkably, as the sum of the estimates of its component local actions. However, in order to take advantage of the factorization of the MAMAB, the exploration bonus in MAUCE is defined as:

$$c_t(\mathbf{a}) = \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)\sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}} \tag{3.4}$$

where each $\overline{r_e}$ represents a prior upper bound on the rewards obtainable by the local actions in group e (assuming non-negative rewards). Similarly to UCB1, MAUCE's exploration bonus also depends on the time index t and on the current local counters n_e, as well as on the total size of the joint action space $|\boldsymbol{\mathcal{A}}|$.

It is important to notice that because the exploration bonus in Equation 3.4 depends on local components, even if a full joint action has *never* been selected before by the agents it still might not be entitled to a high overall exploration bonus. Instead, the bonus of a full joint action will decrease proportionally to how many times its constituent local actions have been selected in the past. In this way, MAUCE leverages the graphical structure of the MAMAB and achieves a logarithmic regret, even without guaranteeing that all full joint actions will be explored. We discuss the proof for this regret bound more in detail in Section

3.2.4.

3.2.3 Upper Confidence Variable Elimination

Contrary to single-agent MABs, for MAUCE it is not trivial to maximize over $\hat{\mu}(\mathbf{a}) + c_t(\mathbf{a})$. Recall that, since we are in a MAMAB, we cannot simply enumerate over all joint actions to determine the best one, as their number is exponential in the number of agents (see Section 2.2). Nevertheless, one would think that, being in a MAMAB, we would be able to construct a coordination graph and use an algorithm designed to maximize over it, like variable elimination (see Section 2.4.1). However, in our case $c_t(\mathbf{a})$ is a non-linear function of the local counts $n_e(\mathbf{a}^e)$, which cannot be factored exactly into a sum of local components. In other words:

$$\sum_{e \in \mathcal{E}} \sqrt{\frac{1}{2} \log(t|\boldsymbol{\mathcal{A}}|) \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}} > \sqrt{\frac{1}{2} \log(t|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}} \tag{3.5}$$

This means that VE does not have the ability to maximize the full UCB expression exactly; and a factored approximation, being higher in value, would result in MAUCE performing unnecessary exploration, increasing its regret. Hence, MAUCE requires a special algorithm to maximize the joint exploration bonus: *Upper Confidence Variable Elimination* (UCVE), a VE variant we specifically introduce for MAUCE.

Our goal is to first determine what information UCVE needs from each local arm so that the maximum UCB value can be correctly determined. As we have already mentioned, the mean estimate $\hat{\mu}(\mathbf{a})$ can be straightforwardly constructed using the sum of its local estimated means $\hat{\mu}_e(\mathbf{a}^e)$, so passing those to UCVE is sufficient for the first term of the UCB value. For the second term, we can express $c_t(\mathbf{a})$ in a simpler form by noticing that, at any given timestep, the part that depends on t is a multiplicative constant $\kappa(t)$ that is equal for all full joint actions, and can thus be safely extracted:

$$c_t(\mathbf{a}) = \sqrt{\frac{1}{2} \log(t|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}} \tag{3.6}$$

$$= \kappa(t) \sqrt{\sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_e(\mathbf{a}^e)}} \tag{3.7}$$

The remaining non-linear term then only depends on the local $\overline{r_e}^2/n_e(\mathbf{a}^e)$ quantities, which are the ones we need to pass to UCVE.

Hence, MAUCE provides UCVE with information about the estimates and confidence bounds in the form of two-element vectors, each containing an estimated mean component and a weighted inverse counts component of a local arm:

$$\boldsymbol{\psi}_e(\mathbf{a}^e) = \langle \hat{\mu}_e(\mathbf{a}^e), \overline{r_e}^2 / n_e(\mathbf{a}^e) \rangle \tag{3.8}$$

Each vector is associated with a group e, so UCVE can construct a coordination graph containing them, and perform a VE-like elimination process to obtain the correct maximum. During each elimination step, the vectors from different factor nodes can be combined using a standard element-wise sum, so that UCVE maximizes over the expression:

$$\overset{\text{MAUCE}}{\text{UCB}}_t(\mathbf{a}) = \hat{\mu}(\mathbf{a}) + c_t(\mathbf{a}) = \boldsymbol{\psi}_{\mathbf{a}}[1] + \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)\boldsymbol{\psi}_{\mathbf{a}}[2]} \tag{3.9}$$

where

$$\boldsymbol{\psi}_{\mathbf{a}} = \sum_{e \in \mathcal{E}} \boldsymbol{\psi}_e(\mathbf{a}^e) \tag{3.10}$$

Vector formulations of reward, as those of Equations 3.8–3.10, are often used in the multi-objective decision making literature [65]. The *multi-objective variable elimination (MOVE)* algorithm [66–68] is a maximization procedure based on variable elimination that similarly processes multi-objective vector rewards. In MOVE's case, the output is not a single maximizing full joint action, but a set of all full joint actions that can possibly result in an optimal multi-objective vector reward. The reason for this is that the multi-objective setting explicitly does not provide a max operator between vectors, and so (nearly) all possible full joint action results must be included as possibly optimal. The only way to remove a vector from the pool is if it is *fully dominated*; meaning that there exist another vector that is element-wise greater or equal than it. This is quite similar to the situation of UCVE, as our two-element vectors cannot be directly maximized over as their second component belongs to the non-linear expression in Equation 3.9.

Like VE, at each agent elimination step MOVE computes the best response of the agent k that is being eliminated. However, for the reasons mentioned above, in the multi-objective setting the best response is not an individual vector, but the full *set* of all possible neighbors' joint actions. Because without the maximization step of VE the size of the best responses set would increase exponentially as more agents are eliminated, MOVE performs an additional *pruning* step, where vectors that are locally dominated are removed as they are provably suboptimal and do not need to be processed further to generate the final output set. This pruning is what allows MOVE to process multiple agents while still providing a computational advantage over the naive enumeration of full joint actions.

Taking inspiration from this, our goal for UCVE is to be able to also perform some sort of pruning operation during each step of the elimination phase. Fortunately, in our case UCVE does ultimately need to output only a single vector: the one that maximizes Equation 3.9. For this reason, our pruning can be more aggressive than MOVE's own, which results in an algorithm in which the number of vectors in the intermediate solution sets steeply decreases in the last agent eliminations — in contrast to MOVE, in which the intermediate sets typically continue to grow in size. In particular, our pruning strategy introduces additional lower and upper bound terms x_l and x_u on the exploration part of the vector, which are then used to determine the vectors that cannot contribute to the optimal full joint action. These bounds are computed from all yet unprocessed payoff nodes in the coordination graph, by selectively summing the highest and lowest exploration components of any local joint action in them.

We can now describe more in detail the complete UCVE algorithm, which is described in Algorithm 5. Given UCVE's similarities to the VE algorithm, we keep its same notation, with the difference that the factor nodes p and associated best response functions b are now functions that return sets of two-element vectors and best responses respectively. Thus, initially each factor node and its respective best response set are associated with the singleton sets:

$$p_e(\mathbf{a}^e) \rightarrow \Big\{\boldsymbol{\psi}_e(\mathbf{a}^e)\Big\} \tag{3.11}$$

$$b_{p_e}(\mathbf{a}^e) \rightarrow \Big\{\{\}\Big\} \tag{3.12}$$

where p_e is the factor node of group e, and $\boldsymbol{\psi}_e(\mathbf{a}^e)$ is defined as in Equation 3.8.

Just like VE, given an agent k and its neighbors $\mathbf{v}^k$, we can eliminate agent k by replacing all the neighboring factor nodes $\mathbf{p}^k$ with a new unique factor node π. However, given that computing the best responses of k using a maximization is not an option, this step is considerably more complex in UCVE. In particular, we first combine all sets associated with the $\mathbf{p}^k$ factor nodes using a cross-sum operator $\oplus$, defined as:

$$\Psi_1 \oplus \Psi_2 = \Big\{\boldsymbol{\psi}_1 + \boldsymbol{\psi}_2 : \boldsymbol{\psi}_1 \in \Psi_1 \wedge \boldsymbol{\psi}_2 \in \Psi_2\Big\} \tag{3.13}$$

for the sets of vector in the factor nodes and

$$B_1 \oplus B_2 = \Big\{b_1 \cup b_2 : b_1 \in B_1 \wedge b_2 \in B_2\Big\} \tag{3.14}$$

Algorithm 5: UCVE

```
Input: A factor graph $\mathcal{G}$ containing 2-element vector rewards
       An agent selection heuristic $\mathcal{H}$
Output: An optimal joint action, $\mathbf{a}^*$
 1: For each factor node $p \in \mathcal{G}$, create an associated best response function $b_p(\cdot) \leftarrow \emptyset$
 2: while there are agents remaining in $\mathcal{G}$ do
 3:     Select agent $k$ from all remaining agents using heuristic $\mathcal{H}$
 4:     Let $\mathbf{v}^k$ be the neighboring agent nodes of $k$
 5:     Let $\mathbf{p}^k$ be the neighboring factor nodes of $k$
 6:     Let $\mathbf{p}^{\neq k}$ be the non-neighboring factor nodes of $k$
 7:     $x_u,\ x_l \leftarrow$ compute the exploration upper and lower bounds from the factors in $\mathbf{p}^{\neq k}$
 8:     Create a new factor node $\pi$ connecting only the neighbor agents $\mathbf{v}^k$
 9:     for each local joint action $\mathbf{a}^{\mathbf{v}^k}$ of the neighbor agents do
10:         Set $\mathcal{V} \leftarrow \bigcup_{a_k \in \mathcal{A}_k} \bigoplus_{\mathbf{p}^k} p(a_k, \mathbf{a}^{\mathbf{v}^k})$
11:         Set $\beta \leftarrow \bigcup_{a_k \in \mathcal{A}_k} \{a_k\} \oplus \left( \bigoplus_{\mathbf{p}^k} b_p(a_k, \mathbf{a}^{\mathbf{v}^k}) \right)$
12:         Set $\pi(\mathbf{a}^{\mathbf{v}^k}) \leftarrow \mathtt{prune}(\mathcal{V}, x_u, x_l)$
13:         Set $b_\pi(\mathbf{a}^{\mathbf{v}^k}) \leftarrow$ the subset of $\beta$ associated with the non-pruned vectors.
14:     end for
15:     Remove all factors $\mathbf{p}^k$ from $\mathcal{G}$
16:     Remove agent $k$ from $\mathcal{G}$
17:     Add $\pi$ to $\mathcal{G}$ as $p_{\mathbf{v}^k}$
18: end while
19: return $b_\emptyset()$
```

for the best response sets. We denote the resulting sets as $\mathcal{V}$ and β, respectively (lines 10 and 11).

Unfortunately, the size of the sets that are created using the $\oplus$ operator increase exponentially in size with each newly removed agent. Thus, just as in MOVE, we want to define a `prune` operator that can remove those $\boldsymbol{\psi}_\pi$ vectors which are provably suboptimal. Unfortunately, a direct comparison is often not possible: because the weighted inverse counts in these vectors cannot be linearly added to the estimated mean components, we cannot a priori tell whether a vector $\boldsymbol{\psi}_1 \in \mathcal{V}$ is better than another vector $\boldsymbol{\psi}_2 \in \mathcal{V}$ when $\boldsymbol{\psi}_1[1] > \boldsymbol{\psi}_2[1]$ but $\boldsymbol{\psi}_1[2] < \boldsymbol{\psi}_2[2]$; only if one of the two vectors is dominated we can be sure it can be removed.

Instead, we compute an upper and a lower bound on the exploration bonus using the remaining non-neighboring factor nodes $\mathbf{p}^{\neq k}$, using the sums of the maximum, resp. minimum, exploration components:

$$x_u = \sum_{p_e \in \mathbf{p}^{\neq k}} \max_{\mathbf{a}^e} \max_{\boldsymbol{\psi} \in p_e(\mathbf{a}^e)} \boldsymbol{\psi}[2] \tag{3.15}$$

$$x_l = \sum_{p_e \in \mathbf{p}^{\neq k}} \min_{\mathbf{a}^e} \min_{\boldsymbol{\psi} \in p_e(\mathbf{a}^e)} \boldsymbol{\psi}[2] \tag{3.16}$$

These bounds are useful because a vector $\boldsymbol{\psi}_1 \in \mathcal{V}$ cannot contribute to the optimal value if there is another vector $\boldsymbol{\psi}_2 \in \mathcal{V}$ such that

$$\boldsymbol{\psi}_1[1] + \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)(\boldsymbol{\psi}_1[2] + x_u)} < \boldsymbol{\psi}_2[1] + \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)(\boldsymbol{\psi}_2[2] + x_l)} \tag{3.17}$$

Using Equation 3.17, the `prune` step of UCVE can remove many more vectors than a simple domination check, especially when most of the agents have been removed and the space between the upper and lower bounds becomes tighter. This allows UCVE to avoid the performance cost of handling an exponential number of vectors, and thus efficiently compute the highest UCB score of any full joint action.

The resulting vector pruning is then correspondingly applied onto the β set, only keeping the best responses of the unpruned vectors. Because, during the last agent elimination, the upper and lower exploration bounds are necessarily zero — there are no additional factor nodes to compute bounds from — and that $\mathcal{V}$ contains vectors that sum over all the factor nodes in the original graph, then the pruning in Equation 3.17 ends up maximizing over our original UCB condition in Equation 3.9. Thus, we are guaranteed that the last β set will contain a single, optimal joint action, ensuring that UCVE indeed maximizes the UCB constraint we had initially set to optimize.

3.2.4 Regret Analysis

The regret efficiency of MAUCE is achieved by exploiting the local structure of the global reward inherent in a MAMAB. When this structure is missing, a worst-case regret of $O(|A| \log T)$ can be achieved by employing an upper-confidence bound algorithm on the equivalent single-agent bandit [64, 69]. However, because in a MAMAB the size of the joint action space $|\boldsymbol{\mathcal{A}}|$ grows exponentially with the number of agents, this naive bound is of little practical use. Instead, we show that in our setting, MAUCE can provably obtain much less regret. In fact, when the local rewards all have the same range $\overline{r_e}$, the regret of MAUCE becomes *linear* in the number of agents.

Assume a standard MAMAB setting as stated in Definition 3. Further assume that each component of the reward function satisfies $\mathcal{R}_e(\mathbf{a}^e) \in [0, \overline{r_e}]$. Without loss of generality, assume that $\sum_{e\in\mathcal{E}} \overline{r_e} = 1$ as well. Let us define $\hat{\mu}_{e,t}(\mathbf{a}^e)$ as the sample mean reward for $\mathcal{R}_e(\mathbf{a}^e)$ at timestep t, and $n_{e,t}(\mathbf{a}^e)$ as the number of times local action $\mathbf{a}^e$ has been taken by timestep t. Finally, recall our definition of regret: the gap between the true expected rewards of the optimal action $\mathring{\mathbf{a}}$ and action $\mathbf{a}$ (see Equation 3.1).

Theorem 1. *If at each time t we choose $\mathbf{a}_t$ such that*

$$\begin{aligned} \mathbf{a}_t &= \arg\max_{\mathbf{a}} \overset{\text{MAUCE}}{\text{UCB}}{}_t(\mathbf{a}) \\ &= \arg\max_{\mathbf{a}} \hat{\mu}_t(\mathbf{a}) + c_t(\mathbf{a}) \end{aligned}$$

where

$$\begin{aligned} \hat{\mu}_t(\mathbf{a}) &= \sum_{e\in\mathcal{E}} \hat{\mu}_{e,t}(\mathbf{a}^e) \\ c_t(\mathbf{a}) &= \sqrt{\frac{1}{2}\log(t|\boldsymbol{\mathcal{A}}|)\sum_{e\in\mathcal{E}} \frac{\overline{r_e}^2}{n_{e,t}(\mathbf{a}^e)}} \end{aligned}$$

then the expected global regret is bounded by

$$\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t)\right] \leq \frac{2\widetilde{A}\log(T|\boldsymbol{\mathcal{A}}|)\sum_{e\in\mathcal{E}} \overline{r_e}^2}{\min_{\mathbf{a}\neq\mathring{\mathbf{a}}} \rho(\mathbf{a})^2} + \log T + 1\,,$$

where $\widetilde{A}$ is the total number of local joint actions, i.e.:

$$\widetilde{A} \equiv \sum_{e\in\mathcal{E}} \prod_{k\in e} |\mathcal{A}_k| \tag{3.18}$$

Proof. Our goal is to decompose the expected global regret into a sum of components, and then individually derive a regret bound for each component. In this way, their sum can then be straightforwardly bounded as well. We begin by letting

$C_t(\mathbf{a})$ be the event that $\rho(\mathbf{a}) > 2c_t(\mathbf{a})$ holds and $\overline{C}_t(\mathbf{a})$ its negation. By the law of the excluded middle, we can then write

$$\begin{aligned}\mathbb{E}\left[\rho(\mathbf{a}_t)\right] = \mathbb{E}\left[\rho(\mathbf{a}_t)\,|\,C_t(\mathbf{a}_t)\right] P(C_t(\mathbf{a}_t)) \\ + \mathbb{E}\left[\rho(\mathbf{a}_t)\,\middle|\,\overline{C}_t(\mathbf{a}_t)\right] P(\overline{C}_t(\mathbf{a}_t))\end{aligned} \tag{3.19}$$

which implies

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\right] \leq \sum_{t=1}^{T} P(C_t(\mathbf{a}_t)) \\ + \mathbb{E}\left[\rho(\mathbf{a}_t)\mid \overline{C}_t(\mathbf{a}_t)\right] \mathbb{E}\left[\mathcal{I}\{\overline{C}_t(\mathbf{a}_t)\}\right]\end{aligned} \tag{3.20}$$

where $\mathcal{I}\{\cdot\}$ is the indicator function. Note that this holds true because, at worst, $\mathbb{E}\left[\rho(\mathbf{a}_t)\,|\,C_t(\mathbf{a}_t)\right]$ can be 1, as we have assumed that $\sum_{e\in\mathcal{E}} \overline{r_e} = 1$.

We then first look at all time steps on which $C_t(\mathbf{a}_t)$ holds. Specifically, we bound the probability that this event occurs. Using the law of total probability and chain rule, we can derive

$$P(C_t(\mathbf{a}_t)) \leq \sum_{a\in\mathcal{A}} P\left(\mathbf{a} = \mathbf{a}_t\,|\,C_t(\mathbf{a})\right) \tag{3.21}$$

By definition, action $\mathbf{a}_t$ maximizes the upper bound $\overset{\text{MAUCE}}{\text{UCB}}_t(\cdot)$. Therefore,

$$P(\mathbf{a} = \mathbf{a}_t \mid C_t(\mathbf{a})) \tag{3.22}$$

$$= P\Big(\overset{\text{MAUCE}}{\text{UCB}}_t(\mathbf{a}) = \overset{\text{MAUCE}}{\text{UCB}}_t(\mathbf{a}_t)\Big|C_t(\mathbf{a})\Big) \tag{3.23}$$

$$\begin{aligned}&\leq P\Big(\overset{\text{MAUCE}}{\text{UCB}}_t(\mathbf{a}) \geq \overset{\text{MAUCE}}{\text{UCB}}_t(\mathring{\mathbf{a}})\Big|C_t(\mathbf{a})\Big) \\ &= P\left(\hat{\mu}_t(\mathbf{a}) + c_t(\mathbf{a}) \geq \hat{\mu}_t(\mathring{\mathbf{a}}) + c_t(\mathring{\mathbf{a}})\,\middle|\,C_t(\mathbf{a})\right)\end{aligned}$$

$$= P\left(\hat{\mu}_t(\mathbf{a}) - \hat{\mu}_t(\mathring{\mathbf{a}}) \geq c_t(\mathring{\mathbf{a}}) - c_t(\mathbf{a})\,\middle|\,C_t(\mathbf{a})\right) \tag{3.24}$$

Our goal is now to apply Hoeffding's inequality onto Equation 3.24 in order to bound the regret of this component of the global expected regret. To achieve this goal, we can take advantage of the fact that the left hand side of the inequality can be expressed as a sum of random variables, the expected sum of which is equal to the regret of action $\mathbf{a}$. In other words:

$$\mathbb{E}\left[\hat{\mu}_t(\mathbf{a}) - \hat{\mu}_t(\mathring{\mathbf{a}})\right] = \rho(\mathbf{a}) \tag{3.25}$$

and

$$\hat{\mu}_t(\mathbf{a}) - \hat{\mu}_t(\mathbf{\mathring{a}}) = \sum_{e \in \mathcal{E}} \Big[\hat{\mu}_t(\mathbf{a}^e) - \hat{\mu}_t(\mathbf{\mathring{a}}^e) \Big] \tag{3.26}$$

$$= \sum_{e \in \mathcal{E}} \left[\sum_{i=1}^{n_{e,t}(\mathbf{a}^e)} \frac{\mathcal{R}_{e,i}(\mathbf{a}^e)}{n_{e,t}(\mathbf{a}^e)} - \sum_{i=1}^{n_{e,t}(\mathbf{\mathring{a}}^e)} \frac{\mathcal{R}_{e,i}(\mathbf{\mathring{a}}^e)}{n_{e,t}(\mathbf{\mathring{a}}^e)} \right] \tag{3.27}$$

Thus, adding $\rho(\mathbf{a})$ to both sides of the inequality in Equation 3.24 allows us to apply Hoeffding's inequality to the sum of random variables described in Equation 3.27, each of which is bound in the interval $\left[0, \frac{\overline{r_e}}{n_{e,t}(\mathbf{a}^e)}\right]$:

$$P\left(\hat{\mu}_t(\mathbf{a}) - \hat{\mu}_t(\mathbf{\mathring{a}}) \geq c_t(\mathbf{\mathring{a}}) - c_t(\mathbf{a}) \,\middle|\, C_t(\mathbf{a})\right) \tag{3.28}$$

$$= P\left(\hat{\mu}_t(\mathbf{a}) - \hat{\mu}_t(\mathbf{\mathring{a}}) + \rho(\mathbf{a}) \geq c_t(\mathbf{\mathring{a}}) - c_t(\mathbf{a}) + \rho(\mathbf{a}) \,\middle|\, C_t(\mathbf{a})\right) \tag{3.29}$$

$$\leq \exp\left(- \frac{2(\rho(\mathbf{a}) + c_t(\mathbf{\mathring{a}}) - c_t(\mathbf{a}))^2}{\sum_{e \in \mathcal{E}} \overline{r_e}^2 \left(n_{e,t}(\mathbf{a}^e)^{-1} + n_{e,t}(\mathbf{\mathring{a}}^e)^{-1} \right)} \right) \tag{3.30}$$

We now apply condition $C_t(\mathbf{a})$ such that $\rho(\mathbf{a}) > 2c_t(\mathbf{a})$ and derive

$$\begin{aligned}
&P(\mathbf{a} = \mathbf{a}_t \mid C_t(\mathbf{a})) \\
&\leq \exp\left(- \frac{2(c_t(\mathbf{a}) + c_t(\mathbf{\mathring{a}}))^2}{\sum_{e \in \mathcal{E}} \overline{r_e}^2 \left(n_{e,t}(\mathbf{a}^e)^{-1} + n_{e,t}(\mathbf{\mathring{a}}^e)^{-1} \right)} \right) \\
&\leq \exp\left(- \frac{2c_t(\mathbf{a})^2 + 2c_t(\mathbf{\mathring{a}})^2}{\sum_{e \in \mathcal{E}} \overline{r_e}^2 \left(n_{e,t}(\mathbf{a}^e)^{-1} + n_{e,t}(\mathbf{\mathring{a}}^e)^{-1} \right)} \right) \\
&= \exp\left(- \frac{\log(t|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_{e,t}(\mathbf{a}^e)} + \log(t|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_{e,t}(\mathbf{\mathring{a}}^e)}}{\sum_{e \in \mathcal{E}} \overline{r_e}^2 \left(n_{e,t}(\mathbf{a}^e)^{-1} + n_{e,t}(\mathbf{\mathring{a}}^e)^{-1} \right)} \right) \\
&= \exp\left(- \log(t|\boldsymbol{\mathcal{A}}|) \right) \\
&\leq (t|\boldsymbol{\mathcal{A}}|)^{-1}
\end{aligned}$$

Using (3.21), we can conclude

$$\sum_{t=1}^{T} P(C_t(\mathbf{a}_t)) \leq \sum_{t=1}^{T} (t|\boldsymbol{\mathcal{A}}|)^{-1} |\boldsymbol{\mathcal{A}}| \leq \log T + 1 \tag{3.31}$$

where for the last step we used $\sum_{t=1}^{T} t^{-1} < \log T + \gamma + \frac{3}{6T+2} < \log T + 1$ [70], where γ is Euler's constant.

Now we can look at the timesteps where $\overline{C}_t(\mathbf{a}_t)$ holds. Here, either $\mathbf{a}_t = \overset{\star}{\mathbf{a}}$ and $\rho(\mathbf{a}_t) = 0$ both hold true, or

$$\begin{aligned}
\rho(\mathbf{a}_t) &\leq 2c_t(\mathbf{a}_t) \\
\rho(\mathbf{a}_t)^2 &\leq 2\log(t|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \frac{\overline{r_e}^2}{n_{e,t}(\mathbf{a}_t^e)} \\
1 &\leq \frac{2\log(t|\boldsymbol{\mathcal{A}}|)}{\min_e n_{e,t}(\mathbf{a}_t^e)} \frac{\sum_{e \in \mathcal{E}} \overline{r_e}^2}{\min_{\mathbf{a} \neq \overset{\star}{\mathbf{a}}} \rho(\mathbf{a})^2} \\
\min_e n_{e,t}(\mathbf{a}_t^e) &\leq 2\log(T|\boldsymbol{\mathcal{A}}|) \frac{\sum_{e \in \mathcal{E}} \overline{r_e}^2}{\min_{\mathbf{a} \neq \overset{\star}{\mathbf{a}}} \rho(\mathbf{a})^2}
\end{aligned} \tag{3.32}$$

Note that as there are at most $\widetilde{A}$ local joint actions, the left-hand side will increase every at most $\widetilde{A}$ time steps. Since the right-hand side is fixed and does not depend on t, Equation 3.32 can only be true on at most

$$2\widetilde{A}\log(T|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \overline{r_e}^2 / \min_{\mathbf{a} \neq \overset{\star}{\mathbf{a}}} \rho(\mathbf{a})^2$$

different time steps. This implies that

$$\begin{aligned}
&\sum_{t=1}^{T} \mathbb{E}\left[\rho(\mathbf{a}_t) \mid \overline{C}_t(\mathbf{a}_t)\right] \mathbb{E}\left[\mathcal{I}\{\overline{C}_t(\mathbf{a}_t)\}\right] \\
&\quad \leq \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{I}\{\overline{C}_t(\mathbf{a}_t) \wedge \mathbf{a}_t \neq \overset{\star}{\mathbf{a}}\}\right] \\
&\quad \leq \frac{2\widetilde{A}\log(T|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \overline{r_e}^2}{\min_{\mathbf{a} \neq \overset{\star}{\mathbf{a}}} \rho(\mathbf{a})^2}
\end{aligned}$$

Together with (3.20) and (3.31), this implies

$$\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t)\right] \leq \frac{2\widetilde{A}\log(T|\boldsymbol{\mathcal{A}}|) \sum_{e \in \mathcal{E}} \overline{r_e}^2}{\min_{\mathbf{a} \neq \overset{\star}{\mathbf{a}}} \rho(\mathbf{a})^2} + \log T + 1$$

□

Corollary 1. *If $|\mathcal{A}_k| \leq A$ for all agents k, and if $|e| \leq d$ for all groups $e \in \mathcal{E}$, then*

$$\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t)\right] \leq \frac{2|\mathcal{E}|A^d\left(\log T + |\mathcal{K}| \log A\right)}{\min_{\mathbf{a} \neq \mathbf{a}^\star} \rho(\mathbf{a})^2} + \log T + 1\,. \tag{3.33}$$

Proof. $\sum_{e\in\mathcal{E}} \overline{r_e} = 1$ implies $\sum_{e\in\mathcal{E}} \overline{r_e}^2 \leq 1$. Additionally, $\log|\boldsymbol{\mathcal{A}}| = \sum_{k\in\mathcal{K}} \log \mathcal{A}_k \leq |\mathcal{K}| \log A$. Finally, $\widetilde{A} = \sum_{e\in\mathcal{E}} \prod_{k\in e} |\mathcal{A}_k| \leq |\mathcal{E}|A^d$. □

The important thing to note is that the given regret bound is *linear* in the number of agents $|\mathcal{K}|$ and in the number of functions $|\mathcal{E}|$, which implies — since $|\mathcal{E}| \leq \binom{|\mathcal{K}|}{d} < |\mathcal{K}|^d$ — that it is *polynomial* in $|\mathcal{K}|$, with degree at most $d+1$. This is a huge improvement over the naive 'flattened' regret bound, which is *exponential* in the number of agents.

Corollary 2. *If, in addition to the assumptions in Corollary 1, each local function has the same range such that $\overline{r_e} = |\mathcal{E}|^{-1}$ for all e, then*

$$\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t)\right] \leq \frac{2A^d\left(\log T + |\mathcal{K}| \log A\right)}{\min_{\mathbf{a} \neq \mathbf{a}^\star} \rho(\mathbf{a})^2} + \log T + 1\,. \tag{3.34}$$

Proof. If $\overline{r_e} = |\mathcal{E}|^{-1}$ for each e, then $\displaystyle\sum_{e\in\mathcal{E}} \overline{r_e}^2 = |\mathcal{E}|^{-1}$. Thus, $\displaystyle\widetilde{A} \sum_{e\in\mathcal{E}} \overline{r_e}^2 \leq A^d$. □

Note that this implies that under the assumption that each local function has the same range, the regret no longer depends on $|\mathcal{E}|$ and the regret is *linear* in the number of agents.

3.2.5 Experiments

In order to test the performance of MAUCE, and compare it to competing approaches, we tested it on three different settings of increasing complexity, which are described below. We compared our results against several baselines: a uniformly random action selector, Sparse Cooperative Q-Learning (SCQL) [71], and Learning with Linear Rewards (LLR) [58].

SCQL is a multi-agent Q-Learning based algorithm that can leverage domain knowledge about agents' interdependencies to lower its sample requirements (see Section 4.1.6). SCQL was originally proposed in the context of multi-agent MDPs, but can be just as well applied to MAMAB problems. We manage SCQL's exploration by using both optimistic initialization of the Q-function (to 1, the maximum possible local reward) and an ε-greedy policy, with the ε parameter starting from 0.05 and linearly decreasing over time until reaching 0 (with the exact rate of decrease dependent on the problem). These hyperparameters were manually tuned to maximize the performance of SCQL, i.e. to minimize its final regret.

LLR is a UCB algorithm from the combinatorial bandit literature that applies to most MAMABs, as it assumes that the rewards are a linear combination of what we refer to as local reward functions. Contrary to MAUCE however, it computes upper confidence bounds on the local reward components separately, before summing them, rather than our vector-based formulation of Equations 3.8–3.10. LLR is parameterless, aside from the knowledge of which agents depend on each other.

All environments use normalized rewards so that either the maximum possible regret per timestep is one, or, when the true regret cannot be computed, the maximum possible joint reward is one. The maximum possible reward was computed by directly solving non-stochastic versions of the problems with variable elimination (or brute-force enumeration). Reward normalization allows to directly compare the regrets obtained in different environments, to see how each approach performs across different settings.

We now describe each of our problem settings: the 0101-Chain, which is simple but illustrates the fast learning properties of MAUCE; Gem Mining, which is real-world inspired and adapted from an established benchmark multi-objective coordination graph; and Wind Farm, a real-world coordination problem, in which we connect our learning problem to a state-of-the-art wind farm simulator. These environments are used throughout this chapter (see Sections 3.3.5 and 3.4.3), as they provide a good benchmark for both regret minimization and best arm identification.

0101-Chain

The 0101-Chain environment [11] is a simple MAMAB consisting in a number of local reward functions that solely depend on successive pairs of agents, i.e. agents 1 and 2, 2 and 3, 3 and 4, and so on. Thus, if the problem consists of n agents, then it will contain $n-1$ local reward functions. Each local reward function $p_k(a_k, a_{k+1})$ is connected to the two adjacent agents k and $k+1$.All agents have binary actions. An example environment with three agents is shown in Figure 3.1.

Local rewards in this environment are drawn from independent Bernoulli distributions, with a probability that depends on the local joint action. The distributions corresponding to each local action are shown in Table 3.1. These rewards

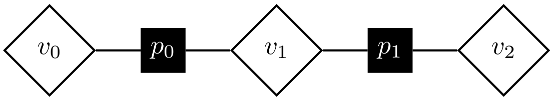


Figure 3.1: The coordination graph for a 0101-Chain instance of with three agents.

k *is even*	$a_{k+1} = 0$	$a_{k+1} = 1$
$a_k = 0$	$\frac{f(\texttt{suc};0.75)}{n-1}$	$\frac{1}{n-1}$
$a_k = 1$	$\frac{f(\texttt{suc};0.25)}{n-1}$	$\frac{f(\texttt{suc};0.9)}{n-1}$

Table 3.1: The reward table for a payoff node p_k in 0101-Chain. n is the number of agents in the problem. $f(\texttt{suc};p)$ is a Bernoulli distribution with success probability p, i.e., $f(1;p) = p$ and $f(0;p) = 1 - p$. The table when k is odd is the same but transposed.

result in a problem where the optimal action can be trivially determined even for large numbers of agents: even-indexed agents need to take action 0, while agents with odd indices must take action 1.

Gem Mining

The Gem Mining problem is adapted from the Mining Day problem from [67], which is a multi-objective coordination graph benchmark problem.

In Gem Mining, a mining company mines gems from a set of mines (local reward functions) located in the mountains (see Figure 3.2). The mine workers live in villages at the foot of the mountains. The company has one van in each village (agents) for transporting workers and must determine every morning to which mine each van should go (actions), but vans can only travel to nearby mines (graph connectivity). Workers are more efficient when there are more workers at a mine: the probability of finding a gem in a mine is $x \cdot 1.03^{w-1}$, where x is the base probability of finding a gem in a mine and w is the number of workers at the mine (as long as at least one worker is present). To generate an instance with v villages (agents), we randomly assign 1-5 workers to each village and connect it to 2–4 mines. Each village is only connected to mines with a greater or equal index, i.e., if village i is connected to m mines, it is connected to mines i to $i + m - 1$. The last village is connected to 4 mines and thus the number of mines is $v + 3$.

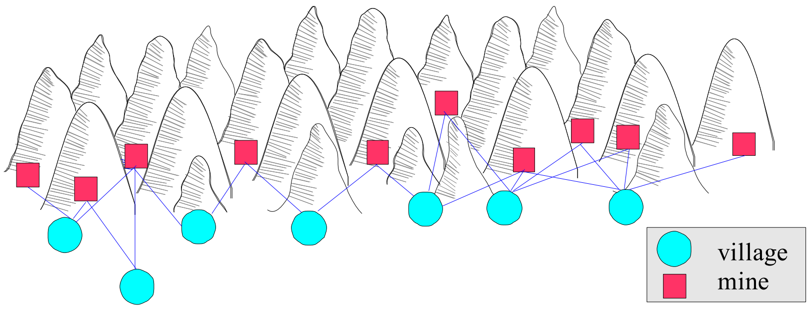


Figure 3.2: Gem Mining example. Each village represents an agent, while the mines represent the local reward functions.

Wind Farm

In our wind farm experiment, we use the state-of-the-art WISDEM FLORIS simulator [72] to mimic the energy production of a series of wind turbines when exposed to a global incoming wind vector. In the real world, turbines can often be oriented at certain angles to maximize production. This is a non-trivial control task, as the turbulence caused by a turbine will negatively affect turbines downwind. The direction of this *wake effect* depends on the angle that the turbine has w.r.t. the incoming wind vector, and its strength decreases over distance. Thus, it is possible to angle a wind turbine to deflect the turbulence away from downwind generators, lowering the its individual production but increasing the overall energy produced by the farm [22].

We setup our simulated wind farm using 11 turbines, and a fixed wind direction (see Figure 3.3). Each turbine has a choice between three different actions (angles) that it can turn to. The three turbines in the front are set in an asymmetrical position with respect to the rest of the farm to make the problem more challenging. The last four turbines downwind are always set directly against the wind and are not controlled by agents, as they cannot generate turbulence that can impact power production. However, the remaining seven turbines do influence the rest of the farm, and so must cooperate to maximize power production.

We setup the coordination graph of the farm given the known angles that all turbines can take, and the maximum possible wind speed, which follows a truncated normal distribution with mean 8.1 m/s. Thus, each local group contains those wind turbines that can directly affect each other's power production through

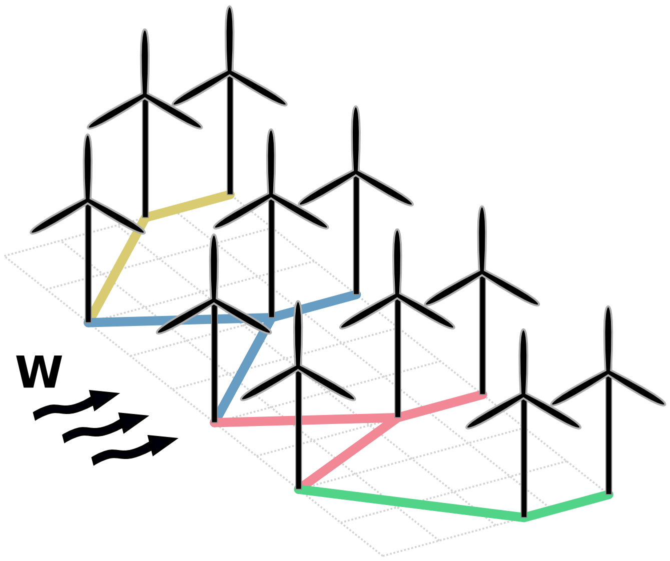


Figure 3.3: Wind farm setup. The incoming wind direction is denoted by the arrows on the left side. The local groups between the agents are denoted by lines of different colors. Note how, if a wind turbine is in a group, the group also contains all turbines that can affect its incoming turbulence. Each of the three most upwind agents is also attached to an individual single-agent group (not shown), to help manage the per-agent rewards returned by the physical simulator.

at least some local joint action combination. Note that our chosen coordination graph is valid only for the specified wind direction and speed. Nevertheless, atmospheric conditions are typically discretized when analyzing operational regimes [73], thus, a graph structure can be made independently for each possible incoming discretized wind vector.

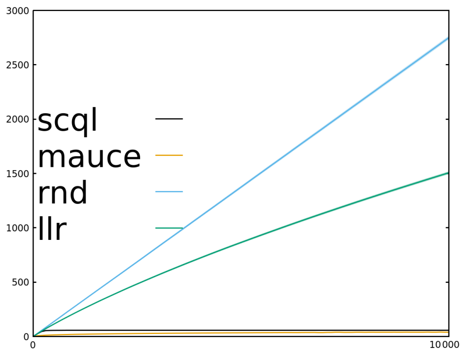
The overall reward is normalized to a $[0, 1]$ interval using the maximum possible overall reward (obtained by empirical trials) at the highest wind strength and the minimum possible reward per turbine at the minimum wind strength. Thus, a reward of 1 is the highest any arm can return, and will not actually be achievable in expectation by the agents. While this makes it impossible to compute the true regret, as choosing the optimal action does not result in a 0 regret in expectation, it avoids having to calculate the true expected reward for all actions in this scenario, which is non-trivial.

Differently from the previous experiments, the underlying physical simulator in this setting provides us with rewards for each individual agent (i.e. the power productions of each windmill) rather than for each group. In order to obtain per-group rewards, we simply sum, for each group, the individual rewards of all agents that solely belong to it. Because in our case the agents belonging to multiple groups are only upwind turbines (whose power production depends on nothing but their individual actions), we can create individual payoff nodes in the coordination graph of the problem which contain their respective rewards. This solution has the advantage of reducing the variance of the rewards of the other groups (because they will be a sum of fewer terms), slightly improving learning overall. Note that environments where this optimization is not possible will simply assign the reward of each shared agent to an arbitrary group it belongs to.

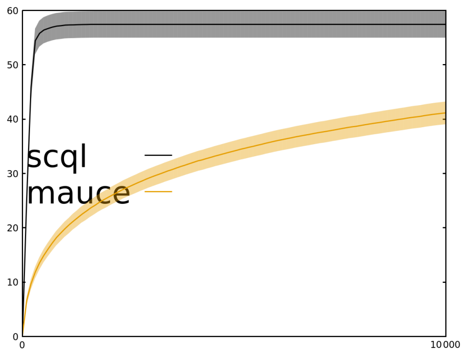
3.2.6 Discussion

We tested the performance of MAUCE on the higly structured 0101-Chain problem with 11 agents for 10 000 joint action executions, and compare its performance against random, SCQL and LLR. The results (Figure 3.4a) indicate that both SCQL and MAUCE can learn effectively, far outclassing random joint action selection and LLR.

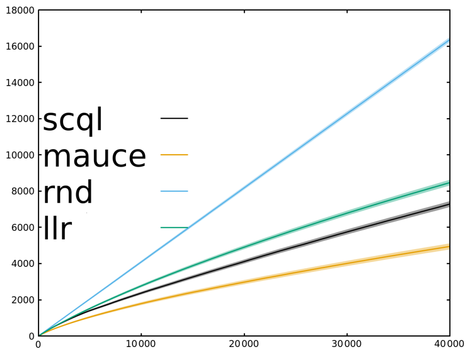
When comparing MAUCE and SCQL (Figure 3.4b), MAUCE achieves considerably less regret than SCQL. This is because MAUCE's exploration strategy is based on the aggregation of local exploration bounds, while SCQL uses an ε-greedy exploration strategy. On the other hand, SCQL does learn the optimal joint action quickly, thanks to optimistic initialization and this aggressive exploration strategy. Note that the annealing of ε needed to be empirically fine-tuned as to calibrate the amount of exploration performed: overexploring results in higher regret, while underexploring risks not identifying the optimal arm. Additionally, once ε reaches 0, i.e., only exploit, the regret graph for SCQL becomes a flat line from that point onward. On the other hand, MAUCE's regret continues to rise as the UCB exploration mechanism increases the exploration bonus for all competing actions at each timestep — albeit with a lower than logarithmic rate — as it is always decreasing the probability of having missed the optimal arm. In this case, if we were to stop this exploration, MAUCE would always select the optimal arm as well. We thus conclude that MAUCE is an effective algorithm that can exploit the graphical structure, leading to superior performance for this highly-structured problem.



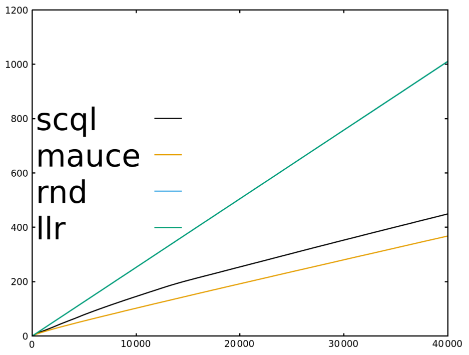
(a) 0101-Chain, All Algorithms



(b) 0101-Chain, MAUCE&SCQL



(c) Gem Mining



(d) Wind Farm

Figure 3.4: Cumulative regret for all experiments as a function of the number of actions executed: a) 0101-Chain with 11 agents averaged over 100 runs, b) Same as (a) but shows only SCQL and MAUCE, c) Gem Mining, averaged over 5 random environment generations with 100 runs for each, and d) Wind Farm, 10 runs.

We then tested MAUCE against the baseline algorithms on randomly generated Gem Mining instances with 5 villages and 8 mines, to compare performances on a more challenging problem. Figure 3.4c represents the average regret over multiple different scenarios. We observe that, while SCQL and LLR are all able to achieve sublinear regret curves, MAUCE handles the exploration-exploitation tradeoffs best, resulting in the lowest regret over time.

Finally, to test the performance of MAUCE on a real-world problem, we run the algorithms on a Wind Farm instance (Figure 3.4d). Due to the high computational costs of running the simulator, we performed only ten runs. As explained before, the measure shown is not an exact form of regret, as the optimal action will not result in a 0 regret in expectation, which explains why the both MAUCE and SCQL regret scores converge to a linear increase rather than a flat line.

The MAUCE algorithm once again performs best, with less cumulative regret than both LLR and SCQL. The LLR algorithm also does not seem to achieve any significant learning with respect to the random policy. Note that MAUCE keeps learning and fine tuning this policy over the whole duration of the experiment, which allows it to increasingly achieve lower regret than SCQL. At timestep 10 000, the difference between the two is $\sim$43, while at timestep 40 000 it is $\sim$81. It is important to note that SCQL could probably be made to perform better by finely tuning the initialization values and epsilon updates, but this would take significant human time and repetitive trials. MAUCE can instead directly manage the exploration-exploitation trade-off by using its local bounds for each local joint action.

We thus conclude that MAUCE is an effective algorithm for trading off exploration versus exploitation in MAMABs, and has superior performance with respect to the alternative algorithms.

3.2.7 Conclusion

In this section we described one of our main contributions, the *multi-agent upper confidence exploration (MAUCE)* algorithm for MAMABs. While learning, MAUCE leverages the graphical properties of the MAMAB by treating as separate objectives both exploration, expressed as a function of the sum over weighted inverse local counts, and exploitation, i.e., the sum over estimated mean local rewards. Via a subroutine, *upper confidence variable elimination (UCVE)*, that can handle these objectives, MAUCE selects the action that best balances exploration and exploitation according to the joint overall mean reward plus an upper confidence exploration bound. We have proven a regret bound for MAUCE that is only linear in the number of agents, rather than exponential, as it would be if we were to flatten the MAMAB to a single-agent MAB. Furthermore, the regret bound is logarithmic in the number of arm pulls. We compared MAUCE empirically to state-of-the-art algorithms in multi-agent reinforcement learning and combinatorial bandits, and have shown that MAUCE achieves much lower empirical regret than these approaches.

We note that the range parameters $\overline{r_e}$ for MAUCE, which represent the difference between the maximum and minimum possible reward for each local joint action, can be difficult to guess in advance when the problem is not exactly known, as in the Wind Farm experiments. One way to mitigate this, could be to estimate them from the coordination graph of expected mean rewards learnt while running the algorithm, rather than running preliminary experiments as we did for the Wind Farm. Still, although it might over or underexplore, MAUCE will still perform some exploration regardless of their value, which can make it a robust choice in environments unknown a priori.

3.3 Multi-Agent Thompson Sampling

In this section, we describe our second contribution in the field of regret minimization in MAMABs: the *multi-agent Thompson sampling* (MATS) algorithm [23], which was developed as joint work with Timothy Verstraeten [74], following the success of the MAUCE algorithm. As with MAUCE, while the algorithm was developed jointly, I then focused on the efficient implementation of the algorithm and generation of the empirical results, while Timothy took the lead in the development of the theoretical regret bound proof.

Similarly to MAUCE, MATS's main challenge is how to handle the exploration-exploitation tradeoff of the agents to minimize the regret they accrue over time. To do this, MATS manages the uncertainty over the MAMAB by maintaining a Bayesian posterior over the expected rewards of each local arm, and uses the mechanism of Thompson sampling (TS) [75] to follow a stochastic policy that selects a full joint action with the same probability that it is optimal. TS has been shown to be highly competitive to other popular action selection methods [76], and the recent theoretical guarantees on its regret [62] have made the method increasingly popular in the literature. Another advantage is that MATS's use of Bayesian priors are generally easier to specify than the hard reward upper-bounds required by MAUCE, and they mesh well with our theme of tackling hard problems by leveraging domain knowledge. Finally, MATS computational performance is significantly higher than MAUCE, as it only requires running standard VE for joint action selection rather than the expensive UCVE algorithm.

In order to explain the MATS algorithm, we give a brief background on Bayesian inference and single-agent Thompson sampling in Sections 3.3.1 and 3.3.2. Then, in Section 3.3.3, we describe MATS in full details. In Section 3.3.4 we provide a finite-time Bayesian regret analysis and prove that the upper regret bound of MATS is low-order polynomial in the number of actions of a single agent for sparse coordination graphs. This is a significant improvement over the exponential bound of classic TS, which is obtained when the coordination graph is ignored [62]. We finally empirically show in Section 3.3.5 that MATS improves upon the state of the art, including MAUCE, in several settings.

3.3.1 Bayesian Inference

MATS, just like Thompson sampling, manages its uncertainty about the bandit's arms using the concept of *Bayesian inference*. This statistics tool allows us to take pre-existing domain knowledge about the local arms of the bandit in the form of *prior distributions*, and update them using the experience data collected from the environment. The updated priors are not simple estimates of the most likely expected rewards for each local arms, but take the form of *posterior distributions* encoding all the uncertainty about our estimates. In a certain sense, in Bayesian methods, the posteriors can be in fact considered the actual arms of the bandit, as at any given timestep our action selection is solely determined by the information that they contain — they constitute our current *belief*.

While we are not going to delve in-depth into Bayesian posterior derivations, we can give a brief introduction on how the posterior distribution is computed. In essence, Bayesian inference allows to obtain an expression that fully describes the posterior in terms of the data, based on our initial knowledge. Given an arm k, this expression can be determined using *Bayes' rule*:

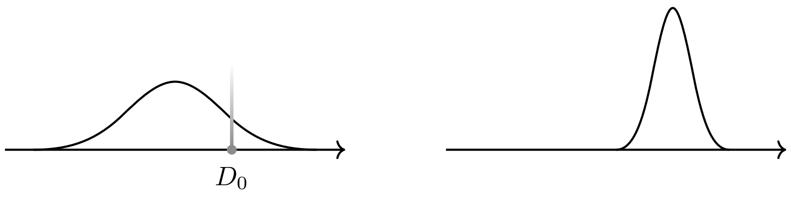
$$P(\theta_k|D_k) = \frac{P(D_k|\theta_k)P(\theta_k)}{P(D_k)} \tag{3.35}$$

where:

- $P(D_k|\theta_k)$ is called the *likelihood distribution*, which in our case is the distribution used to sample the rewards when the arm k is pulled. Thus, the distribution of the data — observed rewards of arm k — depends on the shape of the likelihood distribution and its hyperparameters.
- $P(\theta_k)$ is the prior distribution, which contains our domain information about the hyperparameters of the arm. Note how the prior is a distribution over hyperparameters, and thus may not look like the likelihood at all. For example, in the case of binary rewards sampled from a Binomial distribution (the likelihood), the prior might be represented by a Beta distribution.
- $P(\theta_k|D_k)$ is our posterior, describing the distribution of the hyperparameters θ_k given the data D_k. In particular, MATS and TS are interested into the distribution of the expected mean of the arm.

Note that the distribution $P(D_k)$ is generally ignored as its only purpose is to behave as a normalizing factor to ensure that the posterior is a proper distribution; thus its constant value can be determined once the posterior expression has been computed, and there is generally no need to characterize it further.

Bayes' rule constitutes the core of Bayesian inference: it provides an expression that describes the posterior solely in terms of experience data, where the shape of this expression depends entirely on our initial assumptions about the prior and likelihood distributions. For this reason, care must be taken when choosing which



(a) Initial posterior, together with a single datapoint D_0.

(b) The updated posterior distribution.

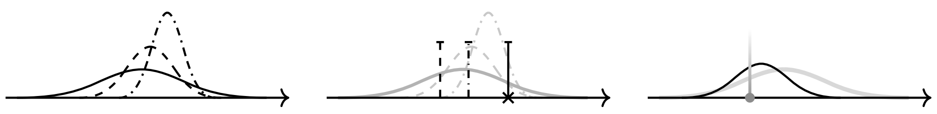
Figure 3.5: As we gather more data, the posterior distribution naturally shifts as to reflect our new belief over the hyperparameters that govern the likelihood distribution. In this case, we can see how our new belief over the expected mean of the data is shifted, together with a reduction of the overall posterior variance.

distributions will model both prior and likelihood, as the wrong choice might result in a non-closed-form posterior distribution, which are impractical to work with and computationally expensive. Thankfully, the field of statistics describes plenty of closed-form relationships between known distributions, so it is often possible to find one that matches a given setting. A simple example of updating a posterior given some data is shown in Figure 3.5.

3.3.2 Thompson Sampling

Thompson sampling is an action selection mechanism that leverages the posteriors obtained via Bayesian inference to efficiently manage uncertainty. Overall, TS's strategy can be summarized as selecting actions with the same probability that they are optimal. The intuition behind this scheme is that it allows TS to mainly select low-regret actions, while still gathering information about actions which have a chance to actually be optimal. The main challenge in implementing this policy is that computing these probabilities exactly for each arm would require expensive integrations over the posteriors, which are generally not feasible.

Instead, TS uses a clever trick to bypass this hurdle. Instead of actually computing the probability of each arm to be the optimal, it simply samples the hyperparameters associated with the expected means of each arm out of their respective posteriors, and then selects the action associated with the highest valued sample. This seems at first glance like a rough empirical approach, and yet under close inspection it can be shown that it perfectly satisfies TS's original goal, and, in the



(a) Initial mean posteriors for three arms.

(b) Select arm with the highest sampled mean.

(c) The selected arm's posterior is updated with new data.

Figure 3.6: A basic overview of Thompson sampling on a MAB. (a) From the initial posteriors over the expected means of each arm, we sample a value from each (b). The arm with the highest associated sampled value is pulled, and a new datapoint is collected, which is then used to update its posterior (c).

case of closed-form posteriors, is also computationally efficient.

A brief example of a single timestep of TS applied to a single-agent bandit with three arms can be seen in Figure 3.6. Figure 3.6a shows three posteriors modeling the distribution of the expected means of three individual arms. The arm that is most likely optimal, given the current knowledge, is associated with the rightmost posterior, which has the highest mean. At the same time, there is still significant uncertainty on what the actual expected means of the arms are. For this reason, in Figure 3.6b, TS samples a value from each of the posteriors, and the agent then pulls the arm that generated the highest sample. Note how in this case the arm that is pulled — the one associated with the flattest posterior curve — is not the one that is, with current knowledge, the most likely best. However, the fact that its sample was the highest signals to TS that the arm has a reasonable chance of being the best, and thus worth exploring. Finally, in Figure 3.6c, a new datapoint is collected from the environment by pulling the selected arm, which leads to the corresponding posterior to be updated — in this case lowering both its mean and variance.

3.3.3 The Algorithm

We have now the background necessary to fully describe the MATS algorithm. At the beginning of the learning process, the true local means $\mu_e(\mathbf{a}^e)$ are treated as unknown. However, we encode our prior knowledge about their possible values in the form of prior distributions, and our current belief over them, given our past history, as posterior distributions. We denote the priors for each local arm as

Algorithm 6: MATS

```
Input: A MAMAB with factored reward function $\mathcal{R}(\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{a}^e)$
       A prior $Q_{\mathbf{a}^e}$ for each local action of the MAMAB.
       A time horizon $H$
1: Initialize $\mathcal{H} \leftarrow \emptyset$
2: for $t = 1$ to $H$ do
3:     for all local joint actions $a_e$ do
4:         $\varrho(\mathbf{a}^e) \sim Q_{\mathbf{a}^e}(\,\cdot \mid \mathcal{H})$
5:     end for
6:     $\mathbf{a} \leftarrow \arg\max_{\mathbf{a}} \sum_{e \in \mathcal{E}} \varrho(\mathbf{a}^e)$
7:     Execute $\mathbf{a}$ and receive local rewards $r_e \leftarrow \mathcal{R}_e(\mathbf{a}^e)$
8:     $\mathcal{H} \leftarrow \mathcal{H} \cup \left\{ \langle \mathbf{a}^e, r_e \rangle_{e=1}^{|\mathcal{E}|} \right\}$
9: end for
```

$Q_{\mathbf{a}^e}(\cdot)$, and their respective posteriors as $Q_{\mathbf{a}^e}(\cdot \mid \mathcal{H}_t)$, where with $\mathcal{H}_t$ we denote our history up to, but not including, timestep t. The history itself is composed by the all the pairs of joint action and respective rewards obtained in the past:

$$\mathcal{H}_t \triangleq \{(\mathbf{a}_i^e, r_{e,i}))\}_{i=1}^{t-1} \tag{3.36}$$

MATS, at each time step t, draws a sample $\varrho_t(\mathbf{a}^e)$ from the expected mean posterior of all local arms given the history $\mathcal{H}_t$:

$$\varrho_t(\mathbf{a}^e) \sim Q_{\mathbf{a}^e}(\cdot \mid \mathcal{H}_t) \tag{3.37}$$

Note that during this step, MATS samples directly the posterior over the unknown local means, which implies that the samples $\varrho_t(\mathbf{a}^e)$ and the unknown means $\mu_e(\mathbf{a}^e)$ are independent and identically distributed at time step t.

Following the same strategy as Thompson sampling, MATS chooses the full joint arm with the highest overall sample:

$$\mathbf{a}_t = \arg\max_{\mathbf{a}} \sum_{e \in \mathcal{E}} \varrho_t(\mathbf{a}^e) \tag{3.38}$$

As this step requires a maximization over a factored function, it can be performed efficiently using coordination graphs and an algorithm like VE (see Section 2.4).

Finally, the joint arm that maximizes Equation 3.38, $\mathbf{a}_t$, is pulled and a reward r_e will be obtained for each local group, which is used to update the history. The full MATS algorithm is formally described in Algorithm 6.

3.3.4 Regret Analysis

In this section we provide a regret bound for MATS, showing that regret scales sublinearly with a factor $\widetilde{A}T$, where $\widetilde{A}$ is the number of local arms (see Equation 3.18).

Before we begin, we set the following assumptions in place:

Assumption 1. *The global rewards have a mean between 0 and 1, i.e.:*

$$\mu(\mathbf{a}) \in [0,1], \forall \mathbf{a} \in \boldsymbol{\mathcal{A}}$$

Assumption 2. *The local rewards shifted by their mean are σ-subgaussian distributed, i.e., $\forall e \in \mathcal{E}, \mathbf{a}^e \in \boldsymbol{\mathcal{A}}^e$, we have:*

$$\mathbb{E}\Big[\exp\Big(t\big(\mathcal{R}_e(\mathbf{a}^e) - \mu_e(\mathbf{a}^e)\big)\Big)\Big] \leq \exp(\frac{1}{2}\sigma^2 t^2)$$

Additionally, for each local arm $\mathbf{a}^e$, we define the variables $n_{e,t}(\mathbf{a}^e)$, representing the number of times it has been pulled before timestep t, and $\hat{\mu}_{e,t}(\mathbf{a}^e)$, representing its currently estimated mean. Note that these variables work identically to the ones defined in MAUCE's Section 3.2.

Consider the event C_T, which states that, until time step T, the differences between the local sample means and true means are bounded by a time-dependent threshold, i.e.,

$$C_T \triangleq \Big(\forall e, \mathbf{a}^e, t \ : \ \big|\hat{\mu}_{e,t}(\mathbf{a}^e) - \mu_e(\mathbf{a}^e)\big| \leq c_{e,t}(\mathbf{a}^e)\Big) \tag{3.39}$$

with

$$c_{e,t}(\mathbf{a}^e) \triangleq \sqrt{\frac{2\sigma^2 \log(\delta^{-1})}{n_{e,t}(\mathbf{a}^e)}} \tag{3.40}$$

where δ is a free parameter that will be chosen later. We denote the complement of the event by $\overline{C}_T$.

Lemma 1. (Concentration inequality) *The probability of exceeding the error bound on the local sample means is linearly bounded by $\widetilde{A}T\delta$. Specifically,*

$$P(\overline{C}_T) \leq 2\widetilde{A}T\delta \tag{3.41}$$

Proof. Using the union bound (U), we can bound the probability of observing event $\overline{C}_T$ as

$$\begin{aligned} P(\overline{C}_T) &\overset{(3.39)}{=} P\left(\exists e, \mathbf{a}^e, t \ : \ |\hat{\mu}_{e,t}(\mathbf{a}^e) - \mu_e(\mathbf{a}^e)| > c_{e,t}(\mathbf{a}^e)\right) \\ &\overset{(U)}{\leq} \sum_{t=1}^{T} \sum_{e \in \mathcal{E}} \sum_{\mathbf{a}^e \in \boldsymbol{\mathcal{A}}^e} P\left(|\hat{\mu}_{e,t}(\mathbf{a}^e) - \mu_e(\mathbf{a}^e)| > c_{e,t}(\mathbf{a}^e)\right) \end{aligned} \tag{3.42}$$

The estimated mean $\hat{\mu}_{e,t}(\mathbf{a}^e)$ is a weighted sum of $n_{e,t}(\mathbf{a}^e)$ random variables distributed according to a σ-subgaussian with mean $\mu_e(\mathbf{a}^e)$. Hence, Hoeffding's inequality (H) is applicable [77].

$$\begin{aligned} P\Big(\big|\hat{\mu}_{e,t}(\mathbf{a}^e) - \mu_e(\mathbf{a}^e)\big| > c_{e,t}(\mathbf{a}^e) \ \Big| \ \mu_e(\mathbf{a}^e)\Big) &\overset{(H)}{\leq} 2\exp\left(-\frac{n_{e,t}(\mathbf{a}^e)}{2\sigma^2}(c_{e,t}(\mathbf{a}^e))^2\right) \\ &\overset{(3.40)}{=} 2\exp\left(-\frac{n_{e,t}(\mathbf{a}^e)}{2\sigma^2}\frac{2\sigma^2\log(\delta^{-1})}{n_{e,t}(\mathbf{a}^e)}\right) \\ &= 2\exp\left(-\log(\delta^{-1})\right). \\ &= 2\delta \end{aligned} \tag{3.43}$$

Therefore, the following concentration inequality on $\overline{C}_T$ holds:

$$P(\overline{C}_T) \leq \sum_{t=1}^{T} \sum_{e \in \mathcal{E}} \sum_{\mathbf{a}^e \in \boldsymbol{\mathcal{A}}^e} 2\delta = 2\widetilde{A}T\delta \tag{3.44}$$

□

Lemma 2. (Bayesian regret bound under C_T) *Provided that the error bound on the local sample means is never exceeded until time T, the Bayesian regret bound, when using the MATS policy π, is of the order*

$$\mathbb{E}\Big[\sum_{t=1}^{T} \rho(\mathbf{a}_t) \ \Big| \ C_T\Big] \leq \sqrt{32\sigma^2\widetilde{A}T|\mathcal{E}|\log(\delta^{-1})} \tag{3.45}$$

Proof. Consider this upper bound on the sample means:

$$u_t(\mathbf{a}) \triangleq \sum_{e \in \mathcal{E}} \hat{\mu}_{e,t}(\mathbf{a}^e) + c_{e,t}(\mathbf{a}^e) \tag{3.46}$$

Given history $\mathcal{H}_t$, the statistics $\hat{\mu}_{e,t}(\mathbf{a}^e)$ and $n_{e,t}(\mathbf{a}^e)$ are known, thus making $u_t(\cdot)$ a deterministic function. Because MATS, like TS, belongs to the class of probability matching methods [78], i.e. the probability distribution of the pulled arm $\mathbf{a}_t$ is equal to the probability distribution of the optimal arm $\mathbf{\mathring{a}}$, we know that:

$$P(\mathbf{a}_t = \cdot \mid \mathcal{H}_t) = P(\mathbf{\mathring{a}} = \cdot \mid \mathcal{H}_t) \tag{3.47}$$

Therefore, the probability matching property of MATS (Equation 3.47) can be applied as follows:

$$\mathbb{E}\left[u_t(\mathbf{a}_t) \mid \mathcal{H}_t\right] = \mathbb{E}\left[u_t(\mathbf{\mathring{a}}) \mid \mathcal{H}_t\right]. \tag{3.48}$$

Hence, using the tower-rule (T), the regret can be bounded as

$$\begin{aligned}
\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t) \mid C_T\right] &\overset{(\mathrm{T})}{=} \mathbb{E}\left[\sum_{t=1}^{T} \mathbb{E}\left[\mu(\mathbf{\mathring{a}}) - \mu(\mathbf{a}_t) \mid \mathcal{H}_t, C_T\right]\right] \\
&= \mathbb{E}\Bigg[\sum_{t=1}^{T} \mathbb{E}\Big[\mu(\mathbf{\mathring{a}}) - u_t(\mathbf{a}_t) \Bigm| \mathcal{H}_t, C_T\Big] \\
&\qquad + \sum_{t=1}^{T} \mathbb{E}\Big[u_t(\mathbf{a}_t) - \mu(\mathbf{a}_t) \Bigm| \mathcal{H}_t, C_T\Big]\Bigg] \\
&\overset{(3.48)}{=} \mathbb{E}\Bigg[\sum_{t=1}^{T} \mathbb{E}\Big[\mu(\mathbf{\mathring{a}}) - u_t(\mathbf{\mathring{a}}) \Bigm| \mathcal{H}_t, C_T\Big] \\
&\qquad + \sum_{t=1}^{T} \mathbb{E}\Big[u_t(\mathbf{a}_t) - \mu(\mathbf{a}_t) \Bigm| \mathcal{H}_t, C_T\Big]\Bigg].
\end{aligned} \tag{3.49}$$

Note that the expression $\mu(\mathbf{\mathring{a}}) - u_t(\mathbf{\mathring{a}})$ is always negative under C_T, i.e.,

$$\begin{aligned}
\mu(\mathbf{\mathring{a}}) - u_t(\mathbf{\mathring{a}}) &\overset{(3.46)}{=} \sum_{e \in \mathcal{E}} \mu_e(\mathbf{\mathring{a}}^e) - \hat{\mu}_{e,t}(\mathbf{\mathring{a}}^e) - c_{e,t}(\mathbf{\mathring{a}}^e) \\
&\overset{(3.39)}{\leq} \sum_{e \in \mathcal{E}} c_{e,t}(\mathbf{\mathring{a}}^e) - c_{e,t}(\mathbf{\mathring{a}}^e) = 0,
\end{aligned} \tag{3.50}$$

while $u_t(\mathbf{a}_t) - \mu(\mathbf{a}_t)$ is bounded by twice the threshold $c_{e,t}(\mathbf{a}^e)$, i.e.,

$$\begin{aligned} u_t(\mathbf{a}_t) - \mu(\mathbf{a}_t) &\overset{(3.46)}{=} \sum_{e\in\mathcal{E}} \hat{\mu}_{e,t}(\mathbf{a}_t^e) + c_{e,t}(\mathbf{a}_t^e) - \mu_e(\mathbf{a}_t^e) \\ &\overset{(3.39)}{\leq} \sum_{e\in\mathcal{E}} c_{e,t}(\mathbf{a}_t^e) + c_{e,t}(\mathbf{a}_t^e) = 2\sum_{e\in\mathcal{E}} c_{e,t}(\mathbf{a}_t^e). \end{aligned} \tag{3.51}$$

Thus, Equation 3.49 can be bounded as

$$\begin{aligned} \mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t) \mid C_T\right] &\leq 2\sum_{t=1}^{T}\sum_{e\in\mathcal{E}} c_{e,t}(\mathbf{a}_t^e) \\ &\leq 2\sum_{t=1}^{T}\sum_{e\in\mathcal{E}} \sqrt{\frac{2\sigma^2\log(\delta^{-1})}{n_{e,t}(\mathbf{a}_t^e)}} \\ &= 2\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\sum_{t=1}^{T} \mathcal{I}\{\mathbf{a}_t^e = \mathbf{a}^e\}\sqrt{\frac{2\sigma^2\log(\delta^{-1})}{n_{e,t}(\mathbf{a}^e)}}, \end{aligned} \tag{3.52}$$

where $\mathcal{I}\{\cdot\}$ is the indicator function. The terms in the summation are only non-zero at the time steps when the local action $\mathbf{a}^e$ is pulled, i.e., when $\mathcal{I}\{\mathbf{a}_t^e = \mathbf{a}^e\} = 1$. Additionally, note that only at these time steps, the counter $n_{e,t}(\mathbf{a}^e)$ increases by exactly 1. Therefore, the following equality holds:

$$\sum_{t=1}^{T} \mathcal{I}\{\mathbf{a}_t^e = \mathbf{a}^e\}\sqrt{(n_{e,t}(\mathbf{a}^e))^{-1}} = \sum_{k=1}^{n_{e,T}(\mathbf{a}^e)} \sqrt{k^{-1}}. \tag{3.53}$$

The function $\sqrt{k^{-1}}$ is decreasing and integrable. Hence, using the right Riemann sum,

$$\sqrt{k^{-1}} \leq \int_{k-1}^{k} \sqrt{x^{-1}}dx. \tag{3.54}$$

Combining Equations 3.52-3.54 leads to a bound

$$\begin{aligned}
\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\ \middle|\ C_T\right] &\overset{(3.52)}{=} 2\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\sum_{t=1}^{T}\mathcal{I}\{\mathbf{a}_t^e=\mathbf{a}^e\}\sqrt{\frac{2\sigma^2\log(\delta^{-1})}{n_{e,t}(\mathbf{a}^e)}} \\
&\overset{(3.53)}{=} \sqrt{8\sigma^2\log(\delta^{-1})}\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\sum_{k=1}^{n_T^e(\mathbf{a}^e)}\sqrt{k^{-1}} \\
&\overset{(3.54)}{\leq} \sqrt{8\sigma^2\log(\delta^{-1})}\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\int_0^{n_T^e(\mathbf{a}^e)}\sqrt{x^{-1}}dx \\
&= \sqrt{8\sigma^2\log(\delta^{-1})}\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\sqrt{4n_{e,T}(\mathbf{a}^e)}.
\end{aligned} \tag{3.55}$$

We use the relationship $||\mathbf{x}||_1 \leq \sqrt{n}||\mathbf{x}||_2$ between the 1- and 2-norm of a vector $\mathbf{x}$, where n is the number of elements in the vector, as follows:

$$\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\left|\sqrt{n_{e,T}(\mathbf{a}^e)}\right| \leq \sqrt{\widetilde{A}}\sqrt{\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\left(\sqrt{n_{e,T}(\mathbf{a}^e)}\right)^2}. \tag{3.56}$$

Finally, note that the sum of all counts $n_{e,T}(\mathbf{a}^e)$ is equal to the total number of local pulls done by MATS until time T, i.e.,

$$\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e} n_{e,T}(\mathbf{a}^e) = |\mathcal{E}|T. \tag{3.57}$$

Using the Equations 3.55-3.57, the complete regret bound under C_T is given by

$$\begin{aligned}
\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\mid C_T\right] &\overset{(3.55)}{\leq} \sqrt{8\sigma^2\log(\delta^{-1})}\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\sqrt{4n_{e,T}(\mathbf{a}^e)} \\
&\overset{(3.56)}{\leq} \sqrt{32\sigma^2\log(\delta^{-1})}\sqrt{\widetilde{A}}\sqrt{\sum_{e\in\mathcal{E}}\sum_{\mathbf{a}^e\in\boldsymbol{\mathcal{A}}^e}\left(\sqrt{n_{e,T}(\mathbf{a}^e)}\right)^2} \\
&\overset{(3.57)}{=} \sqrt{32\sigma^2\log(\delta^{-1})}\sqrt{\widetilde{A}}\sqrt{|\mathcal{E}|T}.
\end{aligned} \tag{3.58}$$

□

Theorem 2. *If Assumptions 1 and 2 hold, then the MATS policy π in a MAMAB satisfies a Bayesian regret bound of*

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\right] &\leq \sqrt{64\sigma^2\widetilde{A}|\mathcal{E}|T\log(\widetilde{A}T)} + \frac{2}{\widetilde{A}} \\ &\in O\left(\sqrt{\sigma^2\widetilde{A}|\mathcal{E}|T\log(\widetilde{A}T)}\right).\end{aligned} \tag{3.59}$$

Proof. Using the law of excluded middle (M) and the fact that $\rho(\mathbf{a}_t)$ and $P(C_T \mid \mathcal{H}_t)$ are between 0 and 1 (B), the regret can be decomposed as

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\right] &\overset{(\mathrm{M})}{=} \mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t) \mid C_T\right]P(C_T) + \mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t) \mid \overline{C}_T\right]P(\overline{C}_T) \\ &\overset{(\mathrm{B})}{\leq} \mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t) \mid C_T\right] + TP(\overline{C}_T).\end{aligned} \tag{3.60}$$

Then, according to Lemmas 1 and 2 (L), we have

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\right] &\overset{(3.60)}{\leq} \mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t) \mid C_T\right] + TP(\overline{C}_T) \\ &\overset{(\mathrm{L})}{\leq} \sqrt{32\sigma^2\widetilde{A}|\mathcal{E}|T\log(\delta^{-1})} + 2\widetilde{A}T^2\delta.\end{aligned} \tag{3.61}$$

Finally, choosing $\delta = (\widetilde{A}T)^{-2}$, we conclude that

$$\begin{aligned}\mathbb{E}\left[\sum_{t=1}^{T}\rho(\mathbf{a}_t)\right] &\overset{(3.61)}{\leq} \sqrt{32\sigma^2\widetilde{A}|\mathcal{E}|T\log(\delta^{-1})} + 2\widetilde{A}T^2\delta \\ &\leq \sqrt{64\sigma^2\widetilde{A}|\mathcal{E}|T\log\left(\widetilde{A}T\right)} + \frac{2}{\widetilde{A}} \\ &\in O\left(\sqrt{\sigma^2\widetilde{A}|\mathcal{E}|T\log(\widetilde{A}T)}\right).\end{aligned} \tag{3.62}$$

□

Corollary 3. *If $|\mathcal{A}_k| \leq A$ for all agents k, and if $|e| \leq d$ for all groups $e \in \mathcal{E}$, then*

$$\mathbb{E}\left[\sum_{t=1}^{T} \rho(\mathbf{a}_t)\right] \in O\left(|\mathcal{E}|\sqrt{\sigma^2 A^d T \log(|\mathcal{E}|A^d T)}\right). \tag{3.63}$$

Proof. $\widetilde{A} = \sum_{e\in\mathcal{E}} |\boldsymbol{\mathcal{A}}^e| = \sum_{e\in\mathcal{E}} \prod_{k\in e} |\mathcal{A}_k| \leq \mathcal{E}A^d$. □

Corollary 3 states that the regret is sub-linear in terms of the number of timesteps T and low-order polynomial in terms of the largest action space of a single agent when the number of groups and agents per group are small. This reflects the main contribution of this work. When agents are loosely coupled, the *effective* joint arm space is significantly reduced, and MATS provides a mechanism that efficiently deals with such settings. This is a significant improvement over the established classic regret bounds of vanilla TS when the MAMAB is "flattened" and the factored structure is neglected [78, 79]. The classic bounds scale exponentially with the number of agents, which renders the use of vanilla TS unfeasible in many multi-agent environments.

3.3.5 Experiments

In this section we empirically evaluate MATS against a random policy (rnd), Sparse Cooperative Q-Learning (SCQL) [42] and MAUCE [11] on the set of environments introduced in Section 3.2.5. In particular, we test on the 0101-Chain, Gem Mining and Wind Farm environments, detailed in Section 3.2.5. For SCQL and MAUCE, we use the same exploration parameters as discussed in Section 3.2.5. For MATS, we always use non-informative Jeffreys priors, which are invariant toward reparametrization of the experimental settings [80]. Although including additional prior domain knowledge could be useful in practice, we use well-known non-informative priors in our experiments to compare fairly with other state-of-the-art techniques. Thus, in the 0101-Chain and Gem Mining environments, MATS employs Jeffreys priors on the unknown local means, which for the Bernoulli likelihood is a Beta prior, $\mathcal{B}(\alpha = 0.5, \beta = 0.5)$ [81]. In the Wind Farm environment, MATS assumes that the local power productions are sampled from Gaussians with unknown mean and variance, which leads to a Student's t-distribution on the mean when using a Jeffreys prior [82].

$f^i \sim \mathcal{P}$	$a_{i+1} = 0$	$a_{i+1} = 1$
$a_i = 0$	0.1	0.3
$a_i = 1$	0.2	0.1

Table 3.2: Poisson 0101 Chain – The unscaled local reward distributions of agents i and $i+1$. Each entry shows the mean for each local arm of agents i and $i+1$.

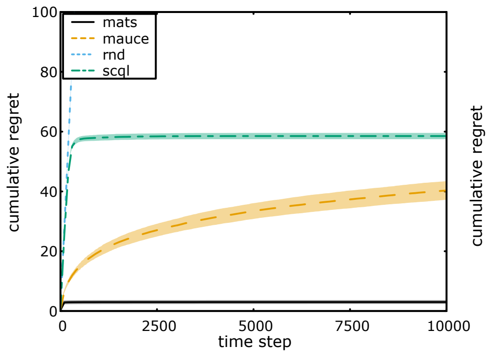
In addition to the environments proposed in Section 3.2.5, we introduce a novel variant of the 0101-Chain environment with Poisson-distributed local rewards, with means specified in Table 3.2. A Poisson distribution is supergaussian, meaning that its tails tend slower towards zero than the tails of any Gaussian, making this setting much more challenging than the original. In this case, both the assumptions made in Theorem 2 and in the established regret bound of MAUCE in Theorem 1 are violated. Additionally, as the rewards are highly skewed, we expect that the use of symmetric exploration bounds in MAUCE will often lead to either over- or underexploration of the local arms.

As a Poisson distribution has unbounded support, we set MAUCE's upper bound parameters $\overline{r_e}$ to the 95% percentiles of the reward distribution, which in this setting is 1 for all groups. For MATS we use non-informative Jeffreys priors on the unknown means, which for the Poisson likelihood is a Gamma prior, $\mathcal{G}(\alpha = 0.5, \beta = 0)$ [81].

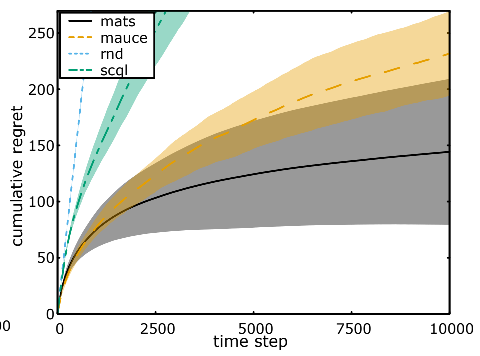
3.3.6 Discussion

The results for all experiments are shown in Figure 3.7. In all plots, both MAUCE and MATS achieve sub-linear regret in terms of time and low-order polynomial regret in terms of the number of local arms for sparse coordination graphs. However, empirically, MATS consistently outperforms MAUCE as well as SCQL. In Figure 3.7a, we see that MATS solves the Bernoulli 0101-Chain problem in only a few time steps, while MAUCE still pulls many sub-optimal actions after 10 000 time steps. In the more challenging Gem Mining problem, shown in Figure 3.7c, the cumulative regret of MAUCE is three times as high as the cumulative regret of MATS at around 40 000 time steps. In Figure 3.7d, MATS achieved a five-fold increase of the normalized power productions with respect to MAUCE, which is itself beating the other benchmarks.

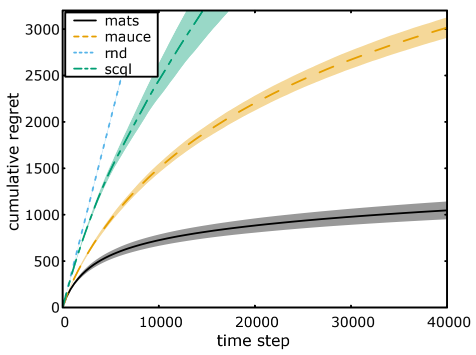
We argue that the high performance of MATS is due to the ability to seamlessly



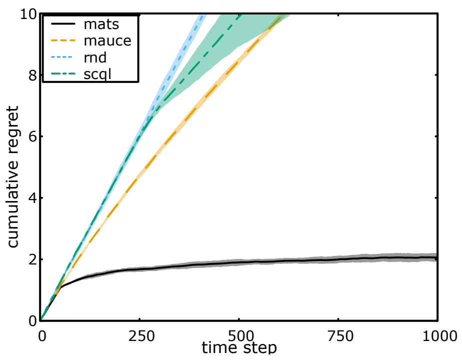
(a) 0101-Chain



(b) Poisson 0101-Chain



(c) Gem Mining



(d) Wind Farm

Figure 3.7: Cumulative regret as a function of the number of actions executed: a) 0101-Chain with 11 agents averaged over 100 runs, b) Same as (a) but for the Poisson 0101-Chain environment, c) Gem Mining, averaged over 5 random environment generations with 100 runs for each, and d) Wind Farm, 10 runs.

include domain knowledge about the shape of the reward distributions and treat the problem parameters as unknowns. To highlight the power of this property, we introduced the Poisson 0101-chain. In this setting, the reward distributions are highly skewed, for which the mean does not match the median. Therefore, in our case, since the mean falls well above 50% of all samples, it is expected that for the initially observed rewards, the true mean will be higher than the sample mean.

Naturally, this bias averages out in the limit, but may have a large impact during the early exploration stage. The high standard deviations in Figure 3.7b support this impact. Although the established regret bounds of MATS and MAUCE do not apply for supergaussian reward distributions, we demonstrate that MATS exploits density information of the rewards to achieve more targeted exploration. In Figure 3.7b, the cumulative regret of MATS stagnates around 7 500 time steps, while the cumulative regret of MAUCE continues to increase significantly. As MAUCE only supports symmetric exploration bounds, it is challenging to correctly assess the amount of exploration needed to solve the task.

MATS was also significantly simpler to setup than MAUCE, which required as input, for each experiment, the reward upper bounds for all agent groups. These values are generally difficult to choose and interpret in terms of the density of the data. In contrast, MATS uses either statistics about the data (if available) or, potentially non-informative, beliefs defined by the user. For example, in the Wind Farm setting, the spread of the data is unknown. MATS effectively maintains a posterior on the variance and uses it to balance exploration and exploitation, while still outperforming MAUCE with a manually calibrated exploration range.

One last point which is not directly visible from the regret figures is the much higher computational efficiency of MATS with respect to MAUCE. This is due to the fact that the UCVE algorithm is generally slower than VE, as it must maintain many more optimal candidates throughout the agent elimination process, which requires significant work. Instead, MATS only requires running vanilla VE on the sampled expected means of the local arms, which allows for a significant overall speedup compare to MAUCE.

3.3.7 Conclusions

We proposed *multi-agent Thompson sampling* (MATS), a novel Bayesian algorithm for multi-agent multi-armed bandits. The method exploits loose connections between agents to solve multi-agent coordination tasks efficiently, while simultaneously managing uncertainty through Bayesian inference. MATS's exploration strategy relies on determining the full joint action that maximizes samples obtained from the local arms' posteriors. This process is very computationally efficient, and requires minimal configuration and setup. We proved that, for σ-subgaussian rewards with bounded means, the expected cumulative regret decreases sub-linearly in time and low-order polynomially in the highest number of actions of a single agent when the coordination graph is sparse. This is in contrast with the regret bound of single-agent TS, which is exponential in the number of agents.

We additionally empirically compared MATS performance against MAUCE and SCQL on several benchmarks. In all tested environments, MATS obtained significantly lower regret than the other methods. The Poisson 0101-Chain setting, which we introduced specifically to test MATS due to its supergaussian rewards, shows that our method can seamlessly adapt to the available prior knowledge and achieve state-of-the-art results.

3.4 Multi-Agent R-max

We now describe our next contribution, the *multi-agent R-max* (MARmax) and *multi-agent V-max* (MAVmax) algorithms [24]. These algorithms tackle the problem of *best arm identification* (see Section 3.1.4) in the multi-agent setting, which, to our knowledge, has not otherwise been focused on in the bandit literature. More specifically, we tackle the problem the *fixed confidence* setting, which requires determining an ε-optimal action with a probability of $1 - \delta$ in the fewest number of timesteps, where both ε and δ are input parameters. This type of guarantee is also called a *probably approximately correct guarantee*, or PAC(ε, δ) for short.

In this section, we show our algorithms can provide a PAC(ε, δ) for the full joint action of the MAMAB's agents, while simultaneously bounding the number of timesteps required to achive the correctness guarantee. In addition, we show that when the local reward functions are all bounded within the same interval, this number of joint action executions does not depend on the number of local reward functions, nor the size of the full joint action space, but only on the maximum number of *local* joint actions for a local reward function. This renders the exploration process in MAMABs much more efficient than by flattening the problem to single-agent bandits. To the best of our knowledge these were the first algorithms that provide PAC-guarantees for best-arm identification in MAMABs.

We empirically evaluate MARmax and MAVmax on a set of benchmarks from the MAMAB literature, and show that while our guarantees already provide a probability bound on the number of samples until the desired confidence level is reached, in practice our algorithms require significantly fewer samples. Especially in MAVmax, which uses more tempered optimism in the face of uncertainty compared to MARmax, we are able to significantly decrease the number of samples, i.e., the number of joint action executions, that is necessary to reach the desired confidence level compared to the theoretical upper bound on this number of joint action executions.

3.4.1 The Algorithm

In a multi-agent multi-armed bandit (MAMAB), during every timestep t we sample $|\mathcal{E}|$ local reward functions. To determine a PAC-bound, it is key to exploit that these reward samples are independently distributed.

Inspired by R-max [83, 84], we first propose the MARmax algorithm, described in Algorithm 7. Similarly to R-max, our strategy maintains an estimator for every possible local reward, $\hat{\mu}_e(\mathbf{a}^e)$, which is optimistically initialised with $\overline{r_e}$, the

Algorithm 7: MARMAX

```
Input: A MAMAB with factored reward function $\mathcal{R}(\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{a}^e)$
       The minimum number of samples per local reward estimator, $m$
Output: The estimated best joint action, $\hat{\mathbf{a}}^\star$
 1: Initialize $\hat{\mu}_e(\mathbf{a}^e)$ to $\overline{r_e}$
 2: Initialize $q_e(\mathbf{a}^e)$ and $n_e(\mathbf{a}^e)$ to 0
 3: Initialize $\upsilon_e(\mathbf{a}^e)$ to 1
 4: Set $\mathbf{a} \leftarrow$ random full joint action
    ▷ While the optimal action contains at least one unknown component
 5: while $\sum_{e\in\mathcal{E}} \upsilon^e(\mathbf{a}^e) > 0$ do
 6:     Pull joint action $\mathbf{a}$ and receive local rewards $r_e$
 7:     for all $e \in \mathcal{E}$ do
 8:         Set $q_e(\mathbf{a}^e) \leftarrow q_e(\mathbf{a}^e) + r_e$
 9:         Set $n_e(\mathbf{a}^e) \leftarrow n_e(\mathbf{a}^e) + 1$
10:         if $n_e(\mathbf{a}^e) \geq m$ then
11:             Set $\hat{\mu}_e(\mathbf{a}^e) \leftarrow \frac{q_e(\mathbf{a}^e)}{n_e(\mathbf{a}^e)}$
12:             Set $\upsilon_e(\mathbf{a}^e) \leftarrow 0$
13:         end if
14:     end for
15:     Set $\mathbf{a} \leftarrow \arg\max_{\mathbf{a}} \sum_{e\in\mathcal{E}} \hat{\mu}_e(\mathbf{a}^e)$
16: end while
17: return a
```

upper bound of the associated random variable. Until the local joint action is not sampled at least m times, we consider its current estimate as *unknown*, and therefore we use in its place the maximum value $\overline{r_e}$ of the local action. After MARMAX has performed a local joint action at least m times, its becomes *known*, and its estimator is replaced by the maximum likelihood estimator with respect to the $\geq m$ samples. In Algorithm 7 the known status is maintained by $\upsilon_e(\mathbf{a}^e)$ which is equal to 1 if it is unknown and 0 otherwise. We will show later how the number of samples m can be computed to achieve a desired PAC(ε, δ).

MARMAX calculates the best joint action $\mathbf{a}_t$ to execute on the basis of the estimators $\hat{\mu}_e$ — both known and unknown — using VE (Section 2.4.1). When all the local components of this joint action are known, i.e., $\sum_{e\in\mathcal{E}} \upsilon_e(\mathbf{a}^e_t) = 0$, MARMAX terminates and recommends $\mathbf{a}_t$ as its output. We note that when this happens, the reward of any other joint action $\mathbf{a}_{alt}$ consists of either known components — which use the current reward estimate — or components that are represented by

Algorithm 8: MAVMAX

```
Input: A MAMAB with factored reward function $\mathcal{R}(\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{a}^e)$
       The minimum number of samples per local reward estimator, $m$
Output: The estimated best joint action, $\hat{\mathbf{a}}^\star$
 1: Initialize $\hat{\mu}_e(\mathbf{a}^e)$ to $\overline{r_e}$
 2: Initialize $q_e(\mathbf{a}^e)$ and $n_e(\mathbf{a}^e)$ to 0
 3: Initialize $\upsilon_e(\mathbf{a}^e)$ to 1
 4: Set $\mathbf{a} \leftarrow$ random full joint action
 5: while $\sum_{e \in \mathcal{E}} \upsilon^e(\mathbf{a}^e) > 0$ do
 6:     Pull $\mathbf{a}$ and receive local rewards $r^e$
 7:     for all $e \in \mathcal{E}$ do
 8:         Set $q_e(\mathbf{a}^e) \leftarrow q_e(\mathbf{a}^e) + r_e$
 9:         Set $n_e(\mathbf{a}^e) \leftarrow n_e(\mathbf{a}^e) + 1$
10:         if $n_e(\mathbf{a}^e) < m$ then
                ▷ Here is the difference with MARMAX
11:             Set $\hat{\mu}_e(\mathbf{a}^e) \leftarrow \frac{q_e(\mathbf{a}^e) + (m - n_e(\mathbf{a}^e))\overline{r_e}}{m}$
12:         else
13:             Set $\hat{\mu}_e(\mathbf{a}^e) \leftarrow \frac{q_e(\mathbf{a}^e)}{n_e(\mathbf{a}^e)}$
14:             Set $\upsilon_e(\mathbf{a}^e) \leftarrow 0$
15:         end if
16:     end for
17:     Set $\mathbf{a} \leftarrow \arg\max_{\mathbf{a}} \sum_{e \in \mathcal{E}} \hat{\mu}_e(\mathbf{a}^e)$
18: end while
19: return $\mathbf{a}$
```

their associated upper bound. Thus, their respective joint value as computed by VE will also be an upper estimate of their value — if additional local components were known, their value would only be lower than the $\overline{r_e}$ upper bound. This allows us to base our PAC bound off the confidence interval for the recommended joint action, $\mathbf{a}_t$.

In Algorithm 7, the reward of local actions that have not been sampled at least m times is estimated as $\overline{r_e}$. This ensures that our estimator is an upper bound of the excepted reward, leading to selecting that local action if it could be part of the optimal joint action. This upper-bound is however naive as it does not take into account the $j < m$ local rewards already sampled. Note that for all $j \in [1, m-1]$

and with the random variables X_e bounded by $\overline{r_e}$ we have that:

$$\sum_{i=1}^{m} X_{e,i} \leq \sum_{i=1}^{k} X_{e,i} + (m-j)\overline{r_e} \tag{3.64}$$

Based on this observation we extend Algorithm 7 by updating our estimators progressively as in V-max [85].

Such tempered optimism does not affect the proof in any way, as long as the estimators, $\hat{\mu}_e(\mathbf{a}^e)$, for the m-sample-based averages remain an upper bound. This leads to the MAVmax algorithm, described in Algorithm 8. Updating the estimators progressively leads to a tighter upper bound which can be expected to speed-up convergence as sub-optimal actions could be discarded earlier.

3.4.2 PAC Analysis

We aim for a PAC-bound where we reach at least a factor $1-\varepsilon$ of the global upper bound on the team reward, with a probability of at least $1-\delta$. Recall that each local joint arm is a random variable $\mathcal{R}_e(\mathbf{a}^e)$ bounded in the interval $[0, \overline{r_e}]$. Once MARmax has acted for sufficient timesteps to provide the recommendation $\mathbf{a}_t$, we know that we have collected, for each local action $\mathbf{a}_t^e$, a set of m_e samples, $X_e[1, m_e]$, to estimate $\hat{\mu}_e(\mathbf{a}_t^e)$. Then, considering the Hoeffding bound for the scaled random variables $Z_e = X_e/m_e$ we obtain the following:

$$P\left(\left|\sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e} Z_{e,i} - \sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e}\frac{\mu_e}{m_e}\right| > t\right) \leq 2\exp\left(\frac{-2t^2}{\sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e}(\overline{r_e}/m_e)^2}\right) \tag{3.65}$$

Given that, with $m = \min_e m_e$, we have that

$$\sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e} Z_{e,i} - \sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e}\frac{\mu_e}{m_e} = \hat{\mu}(\mathbf{a}_t) - \mu(\mathbf{a}_t) \tag{3.66}$$

$$\sum_{e\in\mathcal{E}}\sum_{i=1}^{m_e}\left(\frac{\overline{r_e}}{m_e}\right)^2 = \sum_{e\in\mathcal{E}}\frac{\overline{r_e}^2}{m_e} \leq \frac{\sum_{e\in\mathcal{E}}\overline{r_e}^2}{m}, \tag{3.67}$$

and that we would like to express our bound in terms of the maximum possible reward $\sum_{e\in\mathcal{E}}\overline{r_e}$, we can state the bound in Equation 3.65 as:

$$P\left(|\hat{\mu}(\mathbf{a}_t) - \mu(\mathbf{a}_t)| > \varepsilon\sum_{e\in\mathcal{E}}\overline{r_e}\right) \leq 2\exp\left(-2\frac{\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2 m}{\sum_{e\in\mathcal{E}}\overline{r_e}^2}\right). \tag{3.68}$$

This leads, for each component, to a required number of samples per arm of:

$$\delta \leq 2\exp\left(-2\frac{\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2 m}{\sum_{e\in\mathcal{E}}(\overline{r_e})^2}\right) \tag{3.69}$$

$$\ln(2/\delta) \leq \frac{2\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2 m}{\sum_{e\in\mathcal{E}}\overline{r_e}^2} \tag{3.70}$$

$$m \geq \frac{\sum_{e\in\mathcal{E}}\overline{r_e}^2}{2\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2}\ln(2/\delta), \tag{3.71}$$

In order to reach m samples for each local payoff estimate $\hat{\mu}_e(\mathbf{a}^e)$, we need to consider the worst case number of full joint action arm pulls, n, to obtain these samples. Assuming we can only pull 1 component that is still unknown at every time-step, we would need $m|\mathcal{E}|A^c$ pulls in total, where A is the maximal size of the action space of an agent, $A = \max_k |\mathcal{A}_k|$, and c is the maximal number of agents in scope for a single local reward function, $c = \max_{e\in\mathcal{E}} |e|$. Together with Equation 3.71 we then have:

$$n = m|\mathcal{E}|A^c \geq \frac{\sum_{e\in\mathcal{E}}\overline{r_e}^2}{2\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2}|\mathcal{E}|A^c\ln(2/\delta), \tag{3.72}$$

This leads us to the following theorem:

Theorem 3. *MARmax is a PAC(ε,δ)-learning algorithm with for a MAMAB with a maximum number of A^c fields per local reward function and $|\mathcal{E}|$ local reward functions in total, when parameterised with*

$$m = \left\lceil \frac{\sum_{e\in\mathcal{E}}\overline{r_e}^2}{2\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2}\ln(2/\delta)\right\rceil.$$

The number of required full joint action executions to reach the bound is:

$$n = O\left(\frac{\sum_{e\in\mathcal{E}}\overline{r_e}^2}{2\varepsilon^2(\sum_{e\in\mathcal{E}}\overline{r_e})^2}|\mathcal{E}|A^c\ln(2/\delta)\right).$$

When all the $\overline{r_e}$ have the same value, $\overline{r}$, Equation 3.68 simplifies to:

$$P(|\hat{\mu}(\mathbf{a}_t) - \mu(\mathbf{a}_t)| > \varepsilon|\mathcal{E}|\overline{r}) \leq 2\exp\left(-2\varepsilon^2|\mathcal{E}|m\right), \tag{3.73}$$

which in turn leads to:

$$m \geq \frac{\ln 2/\delta}{2|\mathcal{E}|\varepsilon^2}, \tag{3.74}$$

and,

$$n \geq \frac{A^c \ln 2/\delta}{2\varepsilon^2}, \tag{3.75}$$

i.e., the minimum number of required arm pulls only depends on the highest number of local actions in a local reward function, but not on the number of local reward functions.

Lemma 3. *When all local reward functions in a MAMAB, M, are bounded random variables in the same interval $[0, \overline{r}]$, with a maximum number of A^c fields per local reward function and $|\mathcal{E}|$ local reward functions in total, MARmax is a PAC(ε, δ)-learning algorithm with for M when parameterised with*

$$m = \left\lceil \frac{\ln 2/\delta}{2|\mathcal{E}|\varepsilon^2} \right\rceil.$$

The number of required full joint action executions to reach the bound is:

$$n = O\left(\frac{A^c \ln 2/\delta}{2\varepsilon^2}\right).$$

3.4.3 Experiments

In this section we empirically evaluate our proposed methods, MARmax and MAVmax, on a set of MAMAB environments. In particular, we test on the 0101-Chain, Gem Mining and Wind Farm environments, detailed in Section 3.2.5.

For each environment we test performance over a range of different choices of δ and ε parameters, as well as varying the number of agents $|\mathcal{K}|$ and local reward functions. For each choice of parameters we show the corresponding value of m, the upper bound value of n, the empirical average value of n (the average number of timesteps needed to recommend an arm), and the percentage of runs which achieved ε-optimality, i.e. where regret was less than $\varepsilon \sum_{e \in \mathcal{E}} \overline{r_e}$. All environments use $\overline{r_e} = 1$ for all local reward functions. For the 0101-Chain and Gem Mining environments this is because rewards are always sampled from Bernoulli distributions. For the Wind Farm environment, we empirically evaluated the individual minimum and maximum production of each turbine across all possible joint actions, so that we could normalize each to a reward between 0 and 1.

In addition, the 0101-Chain environment we tested MARmax on is slightly different from the one described in Section 3.2.5. Because our implementation of MARmax selects the initial exploratory joint action deterministically — all

N	ρ	A^c	δ	ε	m	n	Avg n	ε-Opt
MARmax								
10	9	4	0.05	0.05	82	2952	328.00	1.00
			0.10	0.10	17	612	68.80	1.00
20	19	4	0.05	0.05	39	2964	156.15	1.00
			0.10	0.10	8	608	33.50	1.00
50	49	4	0.05	0.05	16	3136	65.09	1.00
			0.10	0.10	4	784	21.11	1.00
MAVmax								
10	9	4	0.05	0.05	82	2952	97.71	1.00
			0.10	0.10	17	612	29.90	1.00
20	19	4	0.05	0.05	39	2964	55.00	1.00
			0.10	0.10	8	608	19.81	1.00
50	49	4	0.05	0.05	16	3136	32.35	1.00
			0.10	0.10	4	784	16.91	1.00

Table 3.3: Results for the 0101-Chain environment, with randomized optimal actions, for different combinations of number of agents, δ and ε.

agents first try to take their action 0 — the reward matrix described in Figure 3.1 resulted in MARmax always discovering the optimal joint action of the chain in $2{\cdot}m$ timesteps. This is because in the default setup of 0101-Chain the optimal action is predetermined, with even-indexed agents needing to take action 0, and odd-indexed agents needing to take action 1. To avoid this issue we instead performed our tests on a randomized version of 0101-Chain. In particular, when generating the environment, we randomly sample a full joint action $\mathring{\mathbf{a}}$ as the target optimal action. Then, for each pair of agents, the rewards table shown in Table 3.1 is either used as-is, or transposed, or its columns swapped, or its rows swapped, so that the optimal reward entry with value of $\frac{1}{n-1}$ is matched with the correct local joint action from $\mathring{\mathbf{a}}$. For example, if agent 1 and agent 2 optimal actions are $(0, 0)$, then the reward table for their local reward is the same as Table 3.1 but with the columns swapped. Note that here the information on whether an agent's index is even or odd is not used.

N	ρ	A^c	δ	ε	m	n	Avg n	ε-Opt
MARmax								
7	10	144	0.05	0.05	74	106560	10656.00	1.00
			0.10	0.10	15	21600	2160.00	1.00
9	12	36	0.05	0.05	62	26784	2232.00	1.00
			0.10	0.10	13	5616	468.00	1.00
13	16	128	0.05	0.05	47	96256	6016.00	1.00
			0.10	0.10	10	20480	1280.00	1.00
MAVmax								
7	10	144	0.05	0.05	74	106560	8598.36	1.00
			0.10	0.10	15	21600	1520.17	1.00
9	12	36	0.05	0.05	62	26784	1855.89	1.00
			0.10	0.10	13	5616	377.12	1.00
13	16	128	0.05	0.05	47	96256	5039.90	1.00
			0.10	0.10	10	20480	898.44	1.00

Table 3.4: Results for the Mines environment, for different combinations of villages, mines, δ and ε.

3.4.4 Discussion

Results are an average over 1000 independent runs for the 0101-Chain and Gem Mining environments, and 100 independent runs for the Wind Farm environment.

In all environments, MARmax and MAVmax always fully respect the theoretical bounds, recommending ε-optimal arms nearly all the time. Even when a ε-optimal arm is not recommended, in expectation the δ bound is never broken. However, the arms recommended by both MARmax and MAVmax were most of the time not the optimal joint actions. This can be explained as in a factored setting the exponential number of full joint arms have expected total rewards that are more tightly distributed than in a more traditional flat bandit environment. This is because for each full joint action there are many similar ones that differ from it by only a single agent action — and the more agents, the less relative impact that agent is going to have. For this reason, there will be many joint arms with expected rewards close to the optimal one. Thus, uniquely identifying the optimal full joint action can require a significant number of pulls in our setting. As MARmax and MAVmax are optimized to find an ε-optimal action, they do not need to expend additional timesteps trying to determine the true optimal joint

N	ρ	A^c	δ	ε	m	n	Avg n	ε-Opt
MARmax								
7	7	27	0.05	0.05	106	20034	1161.42	0.99
			0.10	0.10	22	4158	237.43	1.00
MAVmax								
7	7	27	0.05	0.05	106	20034	424.21	1.00
			0.10	0.10	22	4158	95.39	1.00

Table 3.5: Results for the Wind environment, for different combinations of δ and ε.

action.

We additionally note that empirically the value of n, i.e. the number of arm pulls before an arm is recommended, is generally in the order of mA^c rather than $m|\mathcal{E}|A^c$ as in Equation 3.72. While in Equation 3.72 we considered the worst case scenario of being able to pull only a single unknown local arm at a time, in practice each pull of a joint action samples all $|\mathcal{E}|$ local actions concurrently (with hopefully most of them unknown). Given that A^c is the size of the largest local reward function, once it is fully explored (after mA^c timesteps), the others will generally be explored as well.

Additionally, we see that MAVmax is able to recommend an arm much faster than MARmax, with an average n between 2/3 and 3/4 of MARmax. In additional experiments we performed with higher δ and ε values (and thus lower m and looser bounds), MAVmax can sometimes fail to recommend a ε-optimal arm, but still never breaks the theoretical bounds.

3.4.5 Conclusion

In this section we described our two contributions MARmax and MAVmax, two novel algorithms for best-arm identification in multi-armed multi-agent bandits. The algorithms exploit the structured representation of the joint reward function, which allows them to efficiently learn and identify a ε-optimal joint action. We provided a PAC-bound for MARmax, proving that the sample complexity of the algorithm is linear in the size of the largest local reward function, rather than exponential in the number of agents. We tested both algorithms empirically in a variety of settings taken from the MAMAB literature, and show that the bounds hold in all cases.

4 Multi-Agent Markov Decision Processes

Not dissimilar from the definition of the multi-armed bandit, the *Markov decision process* (MDP) framework extends it to introduce the new fundamental concept of *state*. Whereas in an immutable bandit the agents face, at each timestep, the same identical task — a rhythm broken only by the refinement of their knowledge — in an MDP performing an action drives the environment to a new, different state. In each state, the agents are faced with immediate rewards and state transitions drawn from unknown and potentially unique distributions, which they must take into account when evaluating their options in the long term. The simple addition of states has profound effects on the performance of a policy: blindly selecting actions with excellent immediate rewards has a chance to condemn the agents to a bleak future — circumstances recently familiar to our species as well. Thus, the long-term consequences of each action give rise to the new challenge of *sequential* decision making, which only compounds on the original difficulties of exploration-versus-exploitation present in the simpler bandit.

Learning in sequential environments differs from bandits in several important respects. First, the agents are expected to learn across multiple episodes, rather than a single one. Indeed, they cannot do otherwise: during an individual episode the agents can only experience a single "path" across the states of the MDP, which in most cases only provides (a little) information about some subset of the state space. Thus, to learn a good policy, the agents must restart the task multiple times to gather data throughout the entire state space. In fact, in an MDP the challenge of exploration is not only about determining whether the agents require additional information about the environment, but also on how this information can be practically acquired: trying to reach insufficiently explored states can be a considerable problem in and of itself.

Another important problem that is introduced by the addition of the space dimension is *credit assignment* [86]. The term credit assignment refers to the problem of deciding which factor was responsible for a given accrued reward or cost. This problem is, for example, naturally present in any multi-agent environment: if an agent moves to give way so that another can reach its goal, which of the two is really responsible for the overall success? Credit assignment in this case consists in weighting how the actions of the individual agents actually affected the outcome, and updating the policy accordingly. Similarly, once we start considering sequential tasks, the credit assignment problem appears again, but in a different, temporal form. Specifically, it appears when we try to determine the value of an action taken in the past, knowing that a reward has been obtained in the present. This question can be quite hard to answer, because it must take into account all decisions taken during each timestep between the past and the present.

Finally, we must mention that learning in multi-agent MDPs (MMDPs) generally requires the use of approximations to keep the problem tractable. Differently from a multi-armed bandit, even if an MMDP can be described in a factored manner, it can be shown that the optimal policy cannot be represented in a factored way as well [27]. We know that the optimal policy must fully map the entire space of joint-states to the appropriate optimal joint-actions, but since the curse of dimensionality states that storing such a map exactly is impossible in general, we must instead settle on some form of approximation. These can range from representing the policy in a factored manner anyway, to the use of neural networks as function approximators. Unfortunately, the use of approximations generally means that theoretical guarantees of convergence or optimality are lost in this setting. Nevertheless, such methods have proved to work very well empirically, allowing us to tackle problems much larger than what is possible when looking for exact solutions only.

In this chapter we introduce the formalisms to represent MMDPs, and the mathematical tools used to efficiently learn and update multi-agent policies. Most importantly, we introduce the notion of *values*, which are used to represent the expected rewards that will be obtained from an action in both the present and future. We then discuss some foundational algorithms that are used to plan and learn in this new setting, and finally showcase, in Section 4.2, our main contribution to this topic: the *cooperative prioritized sweeping* (CPS) algorithm. CPS combines a model-based learning approach that takes advantage of the factorization of the model, together with *prioritized sweeping*, which allows it to efficiently update a factored value function multiple time for each interaction with the environment. The result is a highly sample-efficient algorithm that allows learning in environments with hundreds of agents in a computationally efficient manner.

4.1 Background

The *Markov decision process* (MDP) is a mathematical framework that allows to model sequential tasks. An MDP is functionally equivalent to a stochastic state machine, where the transitions probabilities between states also depend on the actions performed by the agent(s). What makes an MDP especially fit for reinforcement learning is what gives it its name: the *Markov property.* The Markov property states that the current state of an MDP must contain all the information necessary to encode the future behavior of the environment. In other words, an MDP is *memoryless*: it does not matter how a certain state was reached or what the past history is; as long as the current state is known then the agents have all the information they need to act. This is extremely important for decision making, because as the agents interact with the environment, the number of possible past histories increases exponentially. The Markov property ensures that it is not necessary to look at these histories in order to act optimally.

4.1.1 Single-Agent MDPs

We now first introduce formally the concept of a single-agent MDP, so that we can discuss more in depth how the Markov property can be leveraged to compute optimal policies.

Definition 5. *A* **single-agent Markov decision process** *(MDP) is a tuple $\langle\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma\rangle$, where:*

- *$\mathcal{S}$ is the finite set of all states of the MDP.*
- *$\mathcal{A}$ is the finite set of all actions of the MDP.*
- *$\mathcal{T}(s'|s,a) \rightarrow [0,1]$ is the transition function, which denotes the probability of transitioning to a given state s' assuming we performed action a in state s.*
- *$\mathcal{R}(s,a) \rightarrow \mathbb{R}$ is the reward function, which denotes the stochastic reward obtained when taking action a in state s (also called the* immediate *reward).*
- *$\gamma \in (0,1)$ is the scalar* discount *of the MDP.*

The *discount* is a scalar term that is applied to reduce, i.e. discount, the value of rewards obtained in the future, so that the return of an episode is computed as a weighted sum $\sum_t r_t \gamma^t$. The reason for its existence lies in the fact that in an MDP it is generally possible for an agent to go back to a previously visited state, so that completing a task can require an arbitrary number of timesteps; indeed an MDP can even be designed so that agents have to keep acting indefinitely. For this reason, the discount term is designed to encourage agents to obtain rewards sooner rather than later, and avoid taking actions that essentially waste time — which would not be inherently bad were the rewards not so weighted. A secondary advantage of the discount is that it ensures that the return of an episode is finite. This allows to determine the best policy when acting in a never ending environment: select the policy which ensures the highest discounted return. An alternative approach to the discount is to provide the agents with a fixed *horizon* in which to act; the horizon however acts as a hard limit, while the discount naturally incorporates the different magnitudes of the rewards obtainable within the environment into a more nuanced policy.

Because the Markov property ensures that each state holds enough information to plan for the future, a policy π can be represented as a simple map from states to action distributions:

$$P_\pi(a|s) \rightarrow [0,1] \tag{4.1}$$

We can then compute the expected return for the policy at a given state s by combining the transition and reward functions of the environment, together with account both the policy's own distribution of actions. We call this expected return the *value* of the policy at s:

$$V_\pi(s) = \sum_a P_\pi(a|s) \left[\mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{T}(s'|s,a) V_\pi(s') \right] \tag{4.2}$$

where we call V the *value function* of the MDP w.r.t. the policy π. This recursive definition is called the *Bellman's equation*, which forms the foundation of all reinforcement learning algorithms in stateful environments. The concept of value is central in RL, because it cleverly encodes all possible branching futures of the stochastic MDP into a single scalar, which can be straightforwardly evaluated for each individual state. An alternative but equally useful definition of the value function can be used by the agent to select the best action at each timestep:

$$Q_\pi(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{T}(s'|s,a) V_\pi(s') \tag{4.3}$$

This *Q-function* represents the expected return of a given action, assuming that policy π is followed in the succeeding timesteps. The agent can use Q by evaluating its value for each available action in its current state, and select the one with the highest Q-value.

4.1.2 Value Iteration

A simple and well-known algorithm that showcases the recursive relationship between V and Q and how it can directly lead to the optimal policy is a planning routine called *Value Iteration* (VI). Given a full model of the environment, VI provably converges to the optimal policy by iteratively updating and maximizing both value functions. The pseudocode for VI is shown in Algorithm 9.

VI works by computing, during its n-th iteration, the optimal value functions for the agents for an horizon of n, i.e. the optimal value functions if the agent only had n timesteps left to act. The key step is in the update on line 6:

$$\hat{V}'(s) \leftarrow \max_a \hat{Q}'(s,a) \tag{4.4}$$

Here $\hat{V}$ is updated to greedily maximize $\hat{Q}$. But because $\hat{Q}$ encodes future rewards (as per Equation 4.3), $\hat{V}$ does not become itself greedy, instead converging one additional (time)step towards the optimal value function. We can get a better appreciation of this fact by diving a bit deeper into the execution of VI: note how at the end of the first iteration $\hat{Q}(\cdot) = \mathcal{R}(\cdot)$ (since $\hat{V}$ is initialized to 0), and $\hat{V}$ then reflects the value function of a completely greedy policy, which only tries to maximize the immediate reward for the current timestep. However, on the second iteration, $\hat{Q}$ now includes the expected discounted reward of the greedy $\hat{V}$ across all possible future states s' for a given action a. This in turn will make Equation 4.4 select the action that maximizes both the immediate reward, and the expected reward for the following timestep, thus obtaining the optimal policy for an horizon of two timesteps. Unless a specific horizon is desired, this process is then repeated until convergence.

4.1.3 Q-Learning

While VI is an excellent tool in medium-sized MDPs, its use is limited to planning, which requires knowing the full environment's dynamics in advance. Fortunately, it is possible to use value functions even when learning in an unknown environment. The *Q-learning* algorithm is a model-free approach that leverages the Bellman

Algorithm 9: Value Iteration

```
Input: A fully known MDP model $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma \rangle$
       A precision threshold $\delta$
Output: The optimal value functions $\overset{\star}{V}$ and $\overset{\star}{Q}$ of $\mathcal{M}$
 1: Initialize $\hat{V}(s) \leftarrow 0 \quad \forall s$
 2: Initialize $\hat{Q}(s,a) \leftarrow 0 \quad \forall s, a$
 3: repeat
 4:     for each state-action pair $\langle s, a \rangle$ do
 5:         Set $\hat{Q}'(s,a) \leftarrow \mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{T}(s'|s,a)\hat{V}(s')$
 6:         Set $\hat{V}'(s) \leftarrow \max_a \hat{Q}'(s,a)$
 7:     end for
 8:     Set $\Delta \leftarrow \max_s |\hat{V}(s) - \hat{V}'(s)|$
 9:     Set $\hat{Q} \leftarrow \hat{Q}'$ and $\hat{V} \leftarrow \hat{V}'$
10: until $\Delta < \delta$
11: return $\langle \hat{V}, \hat{Q} \rangle$
```

equation to iteratively update an estimate of the true Q-function. Over time, this allows the agent to provably converge to the optimal value function. The pseudocode for Q-learning is shown in Algorithm 10.

Similar to VI, the mode of operation of Q-learning is based around the Bellman equation. Here the agent uses each interaction with the environment, in the form of a tuple $\langle s, a, s', r \rangle$, to update an estimate $\hat{Q}$ of the Q-function using an update rule modeled to fit the Bellman equation (line 6). In particular, Q-learning combines the experience point and the current estimate $\hat{Q}$ to generate a target value, called the *temporal difference* (TD) target:

$$r + \gamma \max_{a'} \hat{Q}(s', a') \tag{4.5}$$

The idea is that the expected TD target, over time, will converge to the true value of the state s, which makes it the ideal candidate to update the current estimate. At the same time, since each individual TD target will not equal the expectation — because the stochastic reward r we receive is not $\mathbb{E}[\mathcal{R}(s,a)]$, and the next state s' does not provide a direct expectation over $\mathcal{T}$ — Q-learning updates $\hat{Q}$ incrementally, following a *learning rate* parameter α. By carefully decreasing the learning rate parameter over time, Q-learning progressively lowers the magnitude of its updates, until $\hat{Q}$ converges to the optimal Q-function. Q-learning is an extremely powerful algorithm, as it requires virtually no prior knowledge about

Algorithm 10: Q-learning

```
Input: The state and action spaces $\mathcal{S}, \mathcal{A}$ and discount $\gamma$ of an MDP model $\mathcal{M}$
       A learning rate $\alpha \in (0, 1)$
Output: The optimal value function $\overset{\star}{Q}$ of $\mathcal{M}$
 1: Initialize $\hat{Q}(s, a) \leftarrow 0 \quad \forall\, s, a$
 2: Set $s \leftarrow$ initial state
 3: repeat
 4:     $a \leftarrow$ Select action for state $s$ following some exploratory policy $\pi_{\hat{Q}}$
 5:     Perform action $a$ and obtain experience point $\langle s, a, s', r\rangle$
 6:     Update $\hat{Q}(s, a) \leftarrow (1 - \alpha)\,\hat{Q}(s, a) + \alpha\left[r + \gamma \max_{a'} \hat{Q}(s', a')\right]$
 7:     Decrease $\alpha$ slightly
 8:     $s \leftarrow s'$
 9: until convergence
10: return $\hat{Q}$
```

the environment, it is simple to implement and computationally efficient, and is thus extensively used as a benchmark for most learning MDP algorithms. At the same time, because the rate of updates for a particular value of the Q-function depends on how often it is reached, Q-learning can often require a lot of exploration time — and thus data — before convergence. In other words, Q-learning is fairly *sample inefficient*, as each experience point is only used once and then forgotten. When episode trajectories tend to be dissimilar, for example in highly stochastic or very large environments, this can result in a significant challenge for the algorithm. In these cases, model-based approaches can support the core update loop of Q-learning by performing additional updates based on the model knowledge, essentially creating a bridge between planning and model-free RL approaches. This can significantly speed up learning by extracting more information out of the experience gathered in the environment. We will discuss model-based methods more thoroughly later in the context of multi-agent MDPs.

4.1.4 Multi-Agent MDPs

We can now describe how the single-agent MDP framework can be extended to multi-agent environments. As already mentioned in Section 2.1 and 2.2.1, we first want to redefine the state and action spaces to contain factored representation of, respectively, states and actions. We can then remap the inputs of the transition and reward functions on these new spaces, as well as factorizing their definition. This is especially important because, without a factored representation for these functions, there would not be any appreciable difference between a multi-agent and single-agent model — unless other assumptions are taken.

Definition 6. *A* **multi-agent Markov decision process** *(MMDP) is a tuple* $\langle \mathcal{F}, \mathcal{K}, \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \mathcal{T}, \mathcal{R}, \gamma \rangle$*, where:*

- $\mathcal{F}$ *is the finite set of features that compose each state of the environment. To each feature* f *there is an associated finite set* $\mathcal{S}_f$ *which enumerates the possible values of that feature.*
- $\mathcal{K}$ *is the finite set of agents that act in the environment.*
- $\boldsymbol{\mathcal{S}} = \mathcal{S}_1 \times ... \times \mathcal{S}_{|\mathcal{F}|}$ *is the set of joint states of the environment. Note that it may be possible that not all joint states in this set are realistically reachable in the modeled environment; this however does not result in any practical implications and the spurious states can be simply ignored.*
- $\boldsymbol{\mathcal{A}} = \mathcal{A}_1 \times ... \times \mathcal{A}_{|\mathcal{K}|}$ *is the set of* full joint actions *available to the agents. A full joint action* $\mathbf{a}$ *is thus the collection of all the individual actions performed by the agents in the same timestep.*
- $\mathcal{T}(\mathbf{s}'|\mathbf{s},\mathbf{a}) = \prod_f \mathcal{T}_f(s'_f|\mathbf{s}^f,\mathbf{a}^f) \rightarrow [0,1]$ *is the transition function, which denotes the probability of transitioning to a given state* $\mathbf{s}'$ *assuming we performed the full joint action* $\mathbf{a}$ *in state* $\mathbf{s}$*.*
- $\mathcal{R}(\mathbf{s},\mathbf{a}) = \sum_e \mathcal{R}_e(\mathbf{s}^e,\mathbf{a}^e) \rightarrow \mathbb{R}$ *is the reward function, which denotes the stochastic joint reward obtained when taking full joint action* $\mathbf{a}$ *in state* $\mathbf{s}$*.*
- $\gamma \in (0,1)$ *is the scalar* discount *of the MDP.*

We consider the underlying structure of the factorization — of the state action spaces and the transition and reward functions — as domain knowledge, which is intrinsic of the problem. In particular, we assume that this information is always known in advance. Note that, similarly to coordination graphs (see Section 2.4), this domain information can be learned from data if it not readily accessible, but we do not focus on this problem in this thesis.

We now take a moment to tie the MMDP framework we have introduced to related multi-agent literature and frameworks. As can be expected, the most common element between different multi-agent frameworks is the factorization of the action space: early examples can be found described in [87] as well as in [88]; note however that there is no state factorization here. Where state space factorization is accounted for, it can take place via arbitrary state components, for example in the single agent case in factored MDPs [21, 29] and in factored MMDPs [17, 30] (as well as in our setting), or by associating a state factor to each individual agent, for example in transition independent Dec-MDPs [89] and networked distributed pomdps [90]. A general description of the Dec-POMDP framework, which encompasses all these frameworks, can be found in [91], as well as some discussion of the consequences of different assumptions regarding available communication between agents.

A useful tool that helps visualize the inherent structure and complexity of a given MMDP instance is called a *dynamic decision network* (DDN) — a specific form of *dynamic Bayesian networks* (DBNs), themselves a type of *Bayesian networks* (BNs). In general, BNs are acyclic graphs that represent the dependencies between random variables, so that each directed (hyper-)edge between random variables encodes the conditional dependency between them. DBNs are particular forms of BNs that explicitly include the concept of time, and DDNs consider the inclusion of action nodes, whose value is set rather than dependent on the rest of the graph. In our setting, this allows us to model the conditional relationships between state features and agent actions across a single timestep. In this view, the DDN will contain a set of nodes for state features and agent actions associated with the current timestep, and a second set of nodes for the state features associated with the following timestep. The nodes in the first set are then appropriately connected with the nodes in the second set depending on the dynamics of the environment. The resulting graph allows to easily visualize the dependencies between components of the MMDP, and its inherent complexity which generally depends on the density of the DDN graph itself. Another use is to simplify the formulation of the factored form of an MMDP for a specific problem: given the DDN graph containing the dependencies between an environment's components, we know that each component of the transition function is in the form $\mathcal{T}_f(s'_f|\mathbf{s}^f, \mathbf{a}^f)$ where s'_f is a node in the "next-timestep" set of the DDN, and $\mathbf{s}^f$ and $\mathbf{a}^f$ are simply its parents. This way of constructing the MMDP can be helpful, as prior knowledge about the agent dependencies w.r.t. their environment is generally easier to express in graph form than as a set of factored components. We show a small example DDN in Figure 4.1.

Throughout Chapters 2 and 3 we denoted the arbitrary groups of inderdependent agents as $e \in \mathcal{E}$. Because in an MMDP the most important groups are the ones denoted by the edges of its equivalent DDN representation, we introduce a specific notation to identify them. With this notation, we can describe the factored

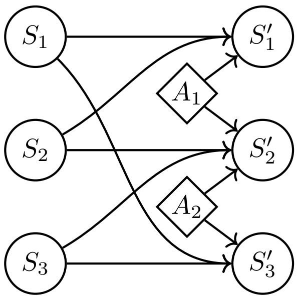


Figure 4.1: A simple DDN with 2 agents and 3 state features. The incoming arrows represent the conditional dependencies in the transition function between the state features in the next timestep and the current state features and agent actions.

nature of the transition function as:

$$T(\mathbf{s}'|\ \mathbf{s},\mathbf{a}) = \prod_f T_f(s'_f \mid \mathbf{s}^{\textcircled{f}}, \mathbf{a}^{\textcircled{f}}) \tag{4.6}$$

where we use the symbol $^{\textcircled{f}}$ to denote the *parent nodes* of S'_f in the DDN graph. For example, in Figure 4.1, $S^{\textcircled{2}} = \{S_2, S_3\}$, and $A^{\textcircled{2}} = \{A_1, A_2\}$.

The factored definition of an MMDP has the explicit goal of skirting around the limitations that are imposed by the curse of dimensionality: the factored forms of the transition and reward functions still represent the full MMDP exactly, but are much simpler to manage. Thus, we can leverage the factorization to harness the power of model-based approaches when learning. However, this fact alone is not sufficient to guarantee that exact multi-agent sequential RL is actually feasible. Recall that the definition of value in Equation 4.2 specifies that the V function has as domain the entire state space (and similarly Q is defined on both the state and action spaces). Because this requirement is at odds with the curse of dimensionality, the question arises on whether the optimal value function — and by extension the optimal policy — for an MMDP could also be defined in a factored form, and thus finally allow for efficient learning in this setting.

Unfortunately it turns out that in general, the optimal value function of an MMDP cannot be factored, and is instead required to fully map the entire state space $\boldsymbol{\mathcal{S}}$ to scalar values explicitly [27]. The reason for this can be demonstrated fairly compactly by using the DDN in Figure 4.1 as the transition function of a simple example MMDP. Let's further assume that the MMDP has a simple reward function with a single component $\mathcal{R} = \mathcal{R}_0(s_0)$. Because an MMDP is a special case of an MDP (see Section 2.2), we can use the Value Iteration algorithm to compute its optimal value function. Recall that at the first iteration of VI we have that $Q(\cdot) = \mathcal{R}(\cdot)$, and so at this point V only depends on the single state feature s_0 — it is factored, and we are happy. However, on the second iteration, VI will not be able to compute the value for the parents of s_0 (s_0, s_1 and a_0) independently. This is because they all individually depend on the value of $V(s_0)$, but computing the conditional transition probabilities for s_0 requires knowing the value of all parents simultaneously. Thus, the Q-function will need to contain a factor that depends on these parents, $Q(s_0, s_1, a_0)$, rather than just a single node as in the previous iteration. The new factor's dependencies further expand at each new iteration of VI, and as long as the DDN graph is connected — which is generally always, as otherwise the problem would be divisible in independent components — the value function will end up depending on all state features and agents simultaneously. Thus, we see that the optimal value function cannot be represented exactly in a factored form, even when the reward function only depends on a single state feature.

MMDPs are thus intractable to solve optimally, and any attempt to learn in them must employ some form of approximation from the onset. In this thesis, we opt to approximate value functions as factored functions, i.e. we assume that the optimal value function can be reasonably approximated by an appropriately structured factored function, such that:

$$Q(\mathbf{s}, \mathbf{a}) \approx \sum_e Q_e(\mathbf{s}^e, \mathbf{a}^e) \tag{4.7}$$

While there exist other options to approximate the joint value function (for example artificial neural networks), one significant advantage of using factored value functions in our setting is that they mesh very well with the rest of the MMDP framework. In particular, using the DDN of the MMDP we can easily identify groups of related variables to use as domains for the components of the value function, so to improve the accuracy of our approximation. Such components then lend themselves to easily construct coordination graphs to determine the optimal full joint actions in any given state. In addition, an approximate version of the Bellman equation can be used to efficiently update a factored value function using the factored transition and reward functions of an MMDP.

To motivate further the use of factored value function, the use of low rank and factored value functions approximations has seen particular success in the past in discrete settings [17, 42, 92], and has more recently received increased attention due to the reduced computational costs and good empirical performance in deep multi-agent reinforcement learning [93, 94]. VDN [95] can learn a simple factorization of the Q-function, with a single component per agent. QMIX [16] improves upon this by adding a mixing network that allows to combine each factor in non-linear ways. More recently, DCG [96] has shown promising results by learning payoff functions for each pair of agents, leveraging parameter sharing, which allows to learn a factored value function using a single network.

If an algorithm does not require a specific factorization, the exact factorization of the value function that is to be used in any given MMDP is generally left to the user. It is common to create a component for each state feature and agent, plus additional components encompassing multiple variables at once, depending on which subsets of variables are presumed to likely impact the optimal policy. We call these subsets *basis domains*, which we denote with $x \subset \mathcal{F}$. Note that basis domains only contain state features and not agents, as V only takes states as parameters. We additionally denote the union of the parent nodes of a basis domain with $^{\textcircled{x}}$ (which can contain agent nodes). Note that basis domains can overlap, and their selection affects the ability of the value function to represent correlations between state variables.

4.1.5 Factored Linear Programming

Similarly to how we used Value Iteration to showcase how the value functions are used in an MDP, we can use a planning algorithm to demonstrate the complexity of dealing with factored value functions, and how the optimal value function in approximated practice. Thus, we now describe the *factored linear programming* (FLP) [17] algorithm.

Linear programming can be used to solve general single agent MDPs [97]. It works by setting up a set of inequality constraints in the form:

$$\mathcal{R}(s,a) + \gamma \sum_{s'} \mathcal{T}(s'|s,a) \overset{\star}{V}(s') \leq V(s) \tag{4.8}$$

for each s and a pair, where each $V(s)$ is a variable, and then minimizing:

$$\text{minimize} \sum_{s} \frac{1}{|S|} V(s) \tag{4.9}$$

Unfortunately, because this method requires $|S| \cdot |A|$ constraints, it does not scale computationally for large state and action spaces, which makes it unusable in an

MMDP context — as expected, given that an MMDP is not generally solvable exactly for this exact reason.

The FLP algorithm works by restructuring these constraints through a coordination factor, so that they become much smaller in number, and thus make the process of solving the linear program tractable again. In addition, it does not directly discover the *values* contained by the factored components, but rather computes the *weights* for each component of an arbitrary input value function that makes it best approximate the optimal value function. In other words, the algorithm gets as input a pre-set factored value function, and it will weight each component so to minimize the error with respect to the optimal value function.

In particular, we first note that, because an MMDP is described in factored form, the constraints described in Equation 4.8 can be restated as:

$$\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma \sum_{\mathbf{s}'} \mathcal{T}(\mathbf{s}'|\mathbf{s},\mathbf{a}) \left[\sum_{k} w_x h_x(\mathbf{s}'^x) \right] \leq \sum_{k} w_x h_x(\mathbf{s}^x) \tag{4.10}$$

where each h_x stands for one of the basis functions that compose our input factored V function, and the w_x are the component's weights that FLP will optimize.

Before continuing, for simplicity of exposition, we introduce a new function g, equal to:

$$g_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) \equiv \sum_{\mathbf{s}'^x} \mathcal{T}(\mathbf{s}'^x|\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) h_x(\mathbf{s}'^x) \tag{4.11}$$

Note that, due to the nature of the transition function of an MMDP — described by its DDN — the domain of g_x is a *backprojection* of the domain of h_x: its domain is composed by the parents nodes of the basis domain of h_x. Moreover, note that the weights w_x are intentionally missing here.

We can now restate Equation 4.10 more compactly using g:

$$\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma \sum_{x} w_x g_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) \leq \sum_{x} w_x h_x(\mathbf{s}^x) \tag{4.12}$$

$$\mathcal{R}(\mathbf{s},\mathbf{a}) + \sum_{x} w_x \left[\gamma g_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) - h_x(\mathbf{s}^x)\right] \leq 0 \tag{4.13}$$

Because our derivation started directly from the single-agent MDP ones, the constraint specified in Equation 4.13 must still be applied for every possible joint state-action pair combination — and they are unfortunately too many for this to be feasible. However, because all these constraints are in the form of *value* ≤ 0, we can use a max operator to construct a single constraint that equates all the others:

$$\max_{\mathbf{s},\mathbf{a}} \mathcal{R}(\mathbf{s},\mathbf{a}) + \sum_x w_x \left[\gamma g_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) - h_x(\mathbf{s}^x)\right] \leq 0 \tag{4.14}$$

Importantly, because the maximization of the left hand side of Equation 4.14 is performed on a set of factored functions — $\mathcal{R}$, g_x and h_x — it is possible to lay out the components of the maximized constraint as if it was a coordination graph, and use the VE algorithm (see Section 2.4.1) to generate only the minimum amount of constraints needed to optimize the weights. This results in a number of constraints that is exponential in the number of induced cliques of the coordination graph (just like VE's complexity), rather than exponential in the dimensionality of the state and action spaces. Once the constraints are generated, the weights can be minimized, and their optimal value recovered.

Given that FLP can only weight some predefined components rather than naturally discover the best values to use to populate them, it is generally used by passing as input a specifically tailored factored value function composed by *indicator functions* — i.e. where the local components contain values that follow a one-hot encoding of their local state space. For example, if the MMDP contains a state feature f_0 so that $|S_0| = 3$, we can create three value function components with the same domain $V_{0,0}(S_0) = \{1, 0, 0\}$, $V_{0,1}(S_0) = \{0, 1, 0\}$ and $V_{0,2}(S_0) = \{0, 0, 1\}$. In this way, the weights discovered by FLP will essentially allow to construct a factored value function with a single component for f_0 but containing the discovered weights as values, i.e. $V_0(S_0) = \{w_0, w_1, w_2\}$.

It is important to remember that, while the constraints used by FLP are equivalent to the Bellman equation of the optimal value function, the output factored value function can only approximate the optimum. Thus, FLP is essentially performing a *projection* of the optimal value function onto the space mapped by the shape of the input factored value function, so that it minimizes its overall error as expressed by its constraints. Yet, this approximation is still important because it represents the upper bound of performance that is achievable within a given value function factorization, and can be used to gain insight into the performance of other algorithms that operate on MMDPs.

4.1.6 Sparse Cooperative Q-learning

We are now ready to look into how factored value functions can be used in the context of reinforcement learning. The Sparse Cooperative Q-learning (SCQL) algorithm is an extension of the Q-learning algorithm we showed in Section 4.1.3

to the MMDP setting. In particular, SCQL employs a modified Bellman equation in order to learn a factored Q-function, which is represented sparsely in a special rule-based form. In this section we describe the algorithm in depth, as SCQL is not only well-suited to showcase the techniques necessary to learn using factored value functions, but is also a fundamental stepping stone to our own contribution to the field.

SCQL was designed to tackle learning in especially sparse settings, where each agent's actions affect the other agents only in a select number of states. For this reason, SCQL opts to represent the Q-function as a set of *rules* ρ: each rule is composed by a *context* and its associated value, with the context being a subset of specific state features and agent actions. Then, the value for any given joint state-action pair is represented by the sum of all rules with a context that matches the pair:

$$Q(\mathbf{s},\mathbf{a}) = \sum_j \rho_j(\mathbf{s},\mathbf{a}) \tag{4.15}$$

While each rule can apply to multiple agents simultaneously, the derivation of the update procedure for these rules begins by assuming a Q-function factorization with one component per agent, i.e. $Q(\mathbf{s},\mathbf{a}) = \sum_k Q_k(\mathbf{s}^k,\mathbf{a}^k)$. Note that while we assume one component per agent, the domains of each depend on the actual rules we want to use for storage (see Equation 4.18). With this in mind, we can rewrite the TD error update equation described on line 6 in Algorithm 10 as:

$$\sum_k Q_k(\mathbf{s}^k,\mathbf{a}^k) \leftarrow (1-\alpha)\sum_k Q_k(\mathbf{s}^k,\mathbf{a}^k) + \alpha\left[\sum_k r_k + \gamma \max_{\mathbf{a}'}\sum_k Q_k({\mathbf{s}'}^k,{\mathbf{a}'}^k)\right] \tag{4.16}$$

where we assume that the reward that is obtained from the environment is a vector with a component r_k for each agent. Because the maximization over $\mathbf{a}'$ can be performed efficiently (see Section 2.4), we can further modify the update rule so that it operates on individual components:

$$Q_k(\mathbf{s}^k,\mathbf{a}^k) \leftarrow (1-\alpha)\,Q_k(\mathbf{s}^k,\mathbf{a}^k) + \alpha\left[r_k + \gamma Q_k({\mathbf{s}'}^k,\mathring{\mathbf{a}}^k)\right] \tag{4.17}$$

SCQL also assumes that rewards are obtained from the environment in a factored manner, one per agent. Because its underlying value storage are rules, SCQL defines each Q_k as:

$$Q_k(\mathbf{s}^k,\mathbf{a}^k) = \sum_j \frac{\rho_j(\mathbf{s}^j,\mathbf{a}^j)}{n_j} \tag{4.18}$$

Algorithm 11: Sparse Cooperative Q-learning

```
Input: The state and action spaces $\boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}$ and discount $\gamma$ of an MMDP model $\mathcal{M}$
       An arbitrary factorization of the value function into contextual rules $\rho$
       A learning rate $\alpha \in (0, 1)$
Output: The learned value function $\hat{Q}$ of $\mathcal{M}$
 1: Initialize $\rho_j(\mathbf{s}^j, \mathbf{a}^j) \leftarrow 0 \quad \forall \mathbf{s}, \mathbf{a}$
 2: Set $\mathbf{s} \leftarrow$ initial state
 3: repeat
 4:     $\mathbf{a} \leftarrow$ Select action for state $\mathbf{s}$ following some exploratory policy $\pi_{\hat{Q}}$
 5:     Perform action $\mathbf{a}$ and obtain experience point $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r} \rangle$
 6:     $\rho_j(\mathbf{s}^j, \mathbf{a}^j) \leftarrow \rho_j(\mathbf{s}^j, \mathbf{a}^j) + \alpha \sum_{k=1}^{n_j} \left[ r_k + \gamma Q_k(\mathbf{s}'^k, \mathring{\mathbf{a}}^k) - Q_k(\mathbf{s}^k, \mathbf{a}^k) \right]$, where
 7:         $Q_k(\mathbf{s}^k, \mathbf{a}^k) = \sum_j \frac{\rho_j(\mathbf{s}^j, \mathbf{a}^j)}{n_j}$
 8:     Decrease $\alpha$ slightly
 9:     $\mathbf{s} \leftarrow \mathbf{s}'$
10: until convergence or action budget is over
11: return $\hat{Q}$
```

where the sum over j only includes rules that contain agent k in their context, and each n_j refers to the total number of agents included in the context of rule ρ_j, i.e. $|\mathbf{a}^j|$. This definition in turn allows to combine Equation 4.17 and 4.18 to create an update procedure for individual Q-rules:

$$\rho_j(\mathbf{s}^j, \mathbf{a}^j) \leftarrow \rho_j(\mathbf{s}^j, \mathbf{a}^j) + \alpha \sum_{k=1}^{n_j} \left[r_k + \gamma Q_k(\mathbf{s}'^k, \mathring{\mathbf{a}}^k) - Q_k(\mathbf{s}^k, \mathbf{a}^k) \right] \tag{4.19}$$

We can now summarize the entire SCQL main learning loop. SCQL first uses Equation 4.18 to generate the values for the Q-function at the current state $\mathbf{s}$. These values are inserted into an appropriately constructed coordination graph, where the optimal action $\mathring{\mathbf{a}}$ is efficiently computed and executed. A factored return $\mathbf{r}$ and a new state $\mathbf{s}'$ are then observed. A new coordination graph is then constructed to compute the optimal action $\mathring{\mathbf{a}}'$ for the new state. Combining all these values allows SCQL update the values of all relevant rules following Equation 4.19. The cycle is then repeated until learning stabilizes, or as long as there is time available. Note that SCQL does not guarantee convergence; this is because while Equation 4.19 does perform the Bellman update exactly for $\mathbf{s}$ and $\mathbf{a}$, it also indirectly updates the value for all other states and action pairs that depend — even partially — on the updated rules. These updates end up affecting the value function in ways which ultimately cannot be controlled or bounded, resulting in a loss of convergence guarantees.

While fundamentally the SCQL algorithm works very similarly to its single-agent Q-learning counterpart, SCQL is able to elegantly combine the advantages of factored functions in order to learn efficiently in an otherwise intractable setting. The update steps are technically simple, and their cost only scales linearly with the number of rules. At the same time, the number of rules can become unmanageable when the number of agents is high, if their interactions are not significantly sparse. Additionally, the computational cost of managing the rule-based definition of the Q-function can be substantial. Fortunately, these issues can be mitigated by substituting the rules with more dense representations. A more serious issue of SCQL is that, just like Q-learning, convergence time can be significant. When the environments are large, SCQL can take extremely long to reach a stable solution, as each environment interaction leads to only a single cumulative update to the value function. This fact, combined with the approximation of the Q-function, can also sometimes lead SCQL to fail to learn a good policy at all no matter the horizon used (see Section 4.2.3).

4.2 Cooperative Prioritized Sweeping

In this section we introduce the *Cooperative Prioritized Sweeping* (CPS) algorithm [25], our novel contribution that significantly improves both learning time and policy quality in the MMDP setting on alternative approaches. CPS achieves this by leveraging model-based learning, which allows it to extract more information out of fewer environment interactions than, for example, what SCQL can do. More specifically, CPS exploits the domain knowledge of graph structure of the MMDP to reason locally about the dynamics of the environment. CPS uses this information to determine the joint state-action pairs where the value function does not accurately reflect the experience data it has gathered, so that the computation spent on learning can be focused where it is most needed.

In Section 4.2.3 we show empirically that using CPS can significantly increase learning speed, even in states which were previously unseen by the agents, in environments with hundreds of agents. We demonstrate that CPS beats state-of-the-art algorithms SCQL and QMIX in several benchmark settings, both in terms of sample-efficiency and policy performance. Where possible, we show also that the policies learned with CPS perform close to the theoretical optimum, even though convergence is not guaranteed.

4.2.1 Prioritized Sweeping

A common way to improve the sample-efficiency of learning is to perform batch updates on the value function in between interactions with the environment. In other words, the value function is updated multiple times after each interaction with the environment to increase the speed of learning. This approach has the advantage of being conceptually simple: the *quality* of the updates does not change; learning is sped up by simply doing *more* of them. The data required to perform batch updates can be sampled from an existing pool, with experience replay being a well-known example of this technique [98]. Alternatively, the data can be generated on the fly from a learned model, as it is done in the Dyna-Q algorithm [36]. Whatever the approach, it is important that updates focus the value function where it does not follow the data, as performing updates randomly in high-dimensional spaces can result in significant waste of resources.

One way to detect what updates need to be prioritized is to use the TD errors from previous updates as guides. It is well-known that prioritizing updates using TD errors can have a significant impact on performance. Eligibility traces [8, 99, 100] collect a set of recent experiences, which can be used to propagate TD errors more quickly through the value function. Alternatively, all experiences can be stored and later presented again to the agent, in what is now famously referred to as experience replay [98]. While experience replay is mostly known for randomly replaying experiences [101], it was originally posited that replaying them in a specific order would significantly improve the propagation of information through the value function. Prioritized experience replay [102] showed this to be true, improving sample-efficiency by scoring experiences using already computed TD errors. Unfortunately, this algorithm is limited by its model-free approach in its identification of good state-action pairs to update. Instead, learning a model of the environment can make it possible to reason much more efficiently about the importance of any given update, before actually paying the cost of performing it.

Prioritized sweeping (PS) is a discrete, single-agent, model-based algorithm that exemplifies the idea of prioritizing updates using TD errors together with a model. PS's design is centered around the concept that, in theory, the most efficient way to use experience data would be to fully run Value Iteration after each interaction of the environment, to ensure that the value function is as up-to-date with the model as possible. Unfortunately, performing a full VI sweep across the state space after each interaction with the environment is too computationally expensive, and generally not feasible. Instead, PS determines which state-action pairs it needs update using a *priority queue*, where each pair's priority is proportional to the likelihood that their update will significantly affect the value function [38]. Thus, the priority is computed using the last TD error of the state-action pair together with the learned model's transition function. The intuition behind this is that if the value of a given state has changed significantly, then the Bellman equation suggests that it is likely that the value of its parents will have to be updated as well. This insight allows PS to efficiently update the value function, as it selectively samples and updates the pairs that are expected to lead to large changes in the value function, and by extension the agent's policy. This can often significantly reduce the amount of data required to learn a policy, as information is propagated through the value function much more quickly than would otherwise be possible. We describe PS in full in Algorithm 12.

The main drawback of the PS algorithm is that it does not readily extend to large environments, and in particular to the multi-agent domain [39]. As the priority queue increases in size, the computational cost of bookkeeping rapidly becomes unsustainable, making the approach impractical. In addition, in an MMDP a model-based update of the value function can be very expensive (see Section 4.1.5). Finally, in large environments PS might also update states that will not be experienced again, reducing efficiency.

Algorithm 12: Prioritized Sweeping

```
Input: The state and action spaces $\mathcal{S}, \mathcal{A}$ and discount $\gamma$ of an MDP model $\mathcal{M}$
Output: The optimal value function $\hat{Q}$ of $\mathcal{M}$
 1: Initialize $\hat{Q}(\cdot,\cdot) \leftarrow 0$
 2: Initialize $\psi(\cdot,\cdot) \leftarrow 0$
 3: while True do
 4:     $a \leftarrow$ Select action for state $s$ following some exploratory policy $\pi_{\hat{Q}}$
 5:     $\langle s', r\rangle \leftarrow$ Execute action $a$ in state $s$
 6:     Update $\mathcal{T}, \mathcal{R}$ using $\langle s, a, s', r\rangle$                    (see Eq. 2.6, 2.7)
 7:     $\delta \leftarrow |r + \gamma \sum_{s'} \mathcal{T}(s'|s,a) \left[\max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)\right]|$
 8:     if $\delta > \theta$ then
 9:         $\psi(s,a) \leftarrow \max(\psi(s,a), \delta)$
10:     end if
11:     for each batch update do
12:         $\langle \dot{s}, \dot{a}\rangle \leftarrow \arg\max_{s,a} \psi(s,a)$
13:         $\psi(\dot{s},\dot{a}) \leftarrow 0$
14:         $\hat{Q}(\dot{s},\dot{a}) = \mathcal{R}(\dot{s},\dot{a}) + \gamma \sum_{s'} \mathcal{T}(s'|\dot{s},\dot{a}) \max_{a'} \hat{Q}(s',a')$          (see Eq. 4.3)
15:         for all possible parents $s'', a''$ that can lead to $\dot{s}$ do
16:             $\delta \leftarrow |\mathcal{R}(\dot{s},\dot{a}) + \gamma \sum_{s'} \mathcal{T}(s'|\dot{s},\dot{a}) \left[\max_{a'} \hat{Q}(\dot{s},a') - \hat{Q}(s'',a'')\right]|$
17:             if $\delta > \theta$ then
18:                 $\psi(s,a) \leftarrow \max(\psi(s,a), \delta)$
19:             end if
20:         end for
21:     end for
22: end while
23: return $\hat{Q}$
```

4.2.2 The Algorithm

CPS is based on the same core idea behind prioritized sweeping: sample-efficiency can be improved by increasing the speed at which the value function incorporates agent experience. Recall the Bellman equation for the optimal value function:

$$\overset{\star}{V}(s) = \mathcal{R}(s, \overset{\star}{a}) + \gamma \sum_{s'} \mathcal{T}(s' |\ s, \overset{\star}{a}) \overset{\star}{V}(s') \tag{4.20}$$

During learning, after each interaction with the environment we obtain new experience that can be used to update the value function. Equation 4.20 suggests that after each update for a particular state s', it is likely that the values for all its predecessor states s should be changed as well. The magnitude of the change for s will be proportional to (i) the temporal difference (TD) error of the initial update for s', and (ii) the probability of the transition $\mathcal{T}(s' \mid s, a)$.

Ideally, one would recursively update the value function until it satisfied Equation 4.20; this is equivalent to planning and would guarantee maximum sample efficiency, as all the experienced information would always be fully reflected in the value function. However, this approach is unfortunately computationally infeasible.

Instead, it makes sense to perform the largest updates to the value function first, as they are the most likely to influence our learned policy. We can then limit the number of recursive updates depending on available resources, and still maximize the information extracted from the data.

Multi-agent priorities

Single-agent PS identifies those state-action pairs where the current estimates of the value-function are more likely to be incorrect, such that they can be updated. This is done by associating each pair with a priority $\psi(s, a)$:

$$\psi(s, a) = |\Delta_{s'}| \, \mathcal{T}(s' \mid s, a) \tag{4.21}$$

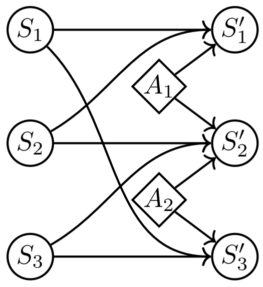
where $\Delta_{s'}$ represents the last obtained TD error for state s'. Each priority represents a backward step in the chain of causality: if the value for s' has changed, then all state-action pairs that can lead to s' likely need to be updated as well. PS stores the priorities in a priority queue, from which it extracts the state-action pairs that maximize ψ as the most likely to lead to improvements in the value function.

We could naively apply this approach in MMDP settings, by working directly with joint state-action pairs:

$$\psi(\mathbf{s}, \mathbf{a}) = |\Delta_{\mathbf{s}'}| \, \mathcal{T}(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) \tag{4.22}$$

However, since the number of joint state and actions in an MMDP increases exponentially with the number of agents, maintaining such a list is generally infeasible.

Our insight is that we can exploit the DDN structure of the MMDP to factorize the priorities into an approximate but much more compact representation. Instead of computing a priority for each joint state-action pair, we can do it for each local



$$\psi_1 : S_1 \times S_2 \times A_1 \to \mathbb{R}$$

$$\psi_2 : S_2 \times S_3 \times A_1 \times A_2 \to \mathbb{R}$$

$$\psi_3 : S_1 \times S_3 \times A_2 \to \mathbb{R}$$

Figure 4.2: The structure of the DDN allows to approximate the priority function ψ with a set of ψ_f, one for each S'_f node. The domain of each ψ_f corresponds to the parents of the corresponding S'_f node.

parent set in the DDN, such that:

$$\begin{aligned}
\psi(\mathbf{s}, \mathbf{a}) &= |\Delta_{\mathbf{s}'}| \, \mathcal{T}(\mathbf{s}' | \, \mathbf{s}, \mathbf{a}) && (4.23)\\
&= \sum_f |\Delta_f| \left[\prod_f \mathcal{T}_f(s'_f \mid \mathbf{s}^{(f)}, \mathbf{a}^{(f)}) \right] && (4.24)\\
&\approx \sum_f |\Delta_f| \, \mathcal{T}_f(s'_f \mid \mathbf{s}^{(f)}, \mathbf{a}^{(f)}) && (4.25)\\
&= \sum_f \psi_f(\mathbf{s}^{(f)}, \mathbf{a}^{(f)}) && (4.26)
\end{aligned}$$

where $\Delta_f = \Delta_{s'_f}$ represents the TD error attributed to the f-th component of the joint state. We leverage the underlying assumption of sparse interactions of an MMDP so that each ψ_f can be represented explicitly, as its domain is relatively small. Note that this factorization is conceptually similar to the one used to factor the value function in Equation 4.7.

A simple example of the factorization of ψ can be seen in Figure 4.2. For each S'_f node, we construct a ψ_f function with domain equal to the parent nodes of S'_f, i.e. the nodes in $\langle \mathbf{S}, \mathbf{A} \rangle$ that directly point to S'_f. Note how each ψ_f has a domain smaller than the full joint state-action spaces. In this way, the space complexity required to represent the priorities is limited by the density of the underlying DDN, allowing us to easily scale to large environments, with possibly hundreds of agents.

Learning Priorities

In Equation 4.25, we can learn each $\mathcal{T}_f$ by maintaining estimates of the proportion that a state s'_f is reached after performing a local joint action $\mathbf{a}^{\textcircled{f}}$ in a local joint state $\mathbf{s}^{\textcircled{f}}$ [39]:

$$\mathcal{T}_f(s'_f \mid \mathbf{s}^{\textcircled{f}}, \mathbf{a}^{\textcircled{f}}) = \frac{N_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s'_f} + N^0_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s'_f}}{\sum_{s''_f \in S_f} N_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s''_f} + N^0_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s''_f}} \tag{4.27}$$

where the $N_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s'_f}$ are the counts for the observed transitions, and the $N^0_{\mathbf{s}^{\textcircled{f}},\mathbf{a}^{\textcircled{f}},s''_f}$ are the priors on the model before interaction begins. This is the maximum likelihood estimate of the transition model with Laplace smoothing, or equivalently the maximum-a-posteriori estimate under a Dirichlet prior (see also Section 2.3).

The local TD errors in Equation 4.25 need to be computed after each update to the value function to provide up-to-date estimates of the priorities. Unfortunately, as each Δ_f refers to a specific state feature, it can be challenging to assign TD errors to each component exactly. However, as we have seen in Section 6, we can obtain fairly good approximations by leveraging factored value functions [16, 17, 42, 92, 96]. Thus, we approximate the Q-function as the sum of lower-dimensional components following Equation 4.7:

$$Q(\mathbf{s}, \mathbf{a}) = \sum_x Q_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) \tag{4.28}$$

where each x is an input basis domain.

Given the factorization in Equation 4.28, each update of the value function naturally induces a set of TD errors Δ_x, i.e. changes in values for each Q-function component Q_x. We can transform these into a set of component-wise TD errors Δ_f by simple redistribution:

$$\Delta_f = \sum_x I(S_f \in \mathbf{S}^{\textcircled{x}}) \frac{\Delta_x}{|\mathbf{S}^{\textcircled{x}}|} \tag{4.29}$$

Using Equations 4.27 and 4.29 we can finally compute priorities for each parent set, i.e. $\psi_f(\mathbf{s}^{\textcircled{f}}, \mathbf{a}^{\textcircled{f}})$. Since our goal is to determine where to update the value function as to maximize our sample-efficiency, we must construct a structure that, much like a priority queue, can determine the joint state-action pair that maximizes ψ. Note that as we have assigned priorities to partial state-action pairs, we need to select a set of parents that are compatible, i.e. where their values match. Figure 4.3 shows an example of this selection process across a set of already computed

$$
\begin{array}{rccccccl}
 & S_1 & S_2 & S_3 & A_1 & A_2 & & \\
\psi_1(& 3 & 0 & & 1 & & &) = 2.0 \\
\boldsymbol{\psi_1}(& \underline{\mathbf{0}} & \underline{\mathbf{2}} & & \underline{\mathbf{2}} & & &) = \mathbf{3.0} \\
\boldsymbol{\psi_2}(& & \underline{\mathbf{2}} & \underline{\mathbf{3}} & \underline{\mathbf{2}} & \underline{\mathbf{1}} & &) = \mathbf{5.0} \\
\psi_2(& & 4 & 1 & 1 & 2 & &) = 8.0 \\
\boldsymbol{\psi_3}(& \underline{\mathbf{0}} & & \underline{\mathbf{3}} & & \underline{\mathbf{1}} & &) = \mathbf{4.0} \\
\psi_3(& 2 & & 0 & & 2 & &) = 1.0 \\
\max_{\mathbf{s},\mathbf{a}} \psi(& 0 & 2 & 3 & 2 & 1 & &) = 12.0
\end{array}
$$

Figure 4.3: An example of the maximization of the full priority function from Figure 4.2. At each update, depending on the current priorities stored in each ψ_f, we want to select the joint state-action pair that maximizes their sum. Note how the selected entries must be *compatible*, meaning their arguments for the same variables must match.

priorities. We can see that each ψ_f is not maximized independently, as we need to select a single joint state-action pair that maximizes their sum ψ.

As ψ is a factored function, we could in theory use an exact algorithm to perform its maximization efficiently, like VE or Max-Plus (see Section 2.4). However, the maximization of ψ is significantly more costly than the usual cost of selecting an optimal joint action for the agents to perform. This is because the selection of a joint action only requires maximizing the value function over the actions of all agents, as the value function is already conditioned on the current known state. For ψ, this conditioning does not occur, and so we require a maximization on both the state and action variables. In addition, because conditioning a variable also removes its connections to the rest of the graph, when maximizing ψ we need to work with denser coordination graphs, which further contributes to the larger computational cost of the maximization.

For these reasons, we settle for an heuristic maximization of ψ rather than an exact one, as we wish batch updates to be inexpensive so the algorithm can perform as many as possible between interactions with the environment. Note that while the joint state-action pair that maximizes the overall priority is the best place to update the value function, any pair with a priority greater than zero still improves on random selection — given our current information. Combined with the fact that multiple batch updates are performed at each timestep, which guarantees a general coverage of the highest priority parents, this makes the use of heuristics in this particular case a non-issue. In our heuristic, we (randomly) traverse all ψ_f, greedily extracting from each the highest valued state-action pair that is compatible with all previously selected entries, using data structures similar to *radix trees* to efficiently guide the search. If, at the end of this procedure, some elements of the joint state-action pair have not been assigned, we uniformly sample their values from their respective domains.

These techniques allow to efficiently prioritize updates even in extremely large state-action spaces, and can remarkably select joint state-action pairs which have never been seen by the agents before, but that are likely to lead to improvements across multiple joint states. This allows us to significantly improve sample-efficiency at a moderate computational cost.

Model-based Reinforcement Learning

In order to demonstrate the advantages of prioritized updates in multi-agent settings, we integrate them into a fully functional model-based algorithm, completing our contribution: the CPS algorithm.

In order to simplify the update equations of the Q-function, we take the minor assumption that the reward function has the same structure as $\mathcal{T}$, such that $\mathcal{R}(\mathbf{s}, \mathbf{a}) = \sum_f \mathcal{R}_f(\mathbf{s}^{\mathcal{D}_f}, \mathbf{a}^{\mathcal{D}_f})$ where each $\mathcal{R}_f$ has the same domain $(\mathbf{S}^{\mathcal{D}_f}, \mathbf{A}^{\mathcal{D}_f})$ as $\mathcal{T}_f$. Given this structure, we assume that rewards are sampled, both from the environment and from the model, as vectors with $|\mathcal{F}|$ elements. While this assumption is not a strict requirement, is also generally not difficult to change the definition of a reward function so that it satisfies this constraint. Note that this representation is somewhat different from what some other multi-agent RL algorithms use (e.g., [42]), which is to consider rewards on a per-agent basis, rather than on a per-state factor basis. In other words, they consider reward samples to be vectors with $|\mathcal{K}|$ entries, i.e., one per agent. However, it is always possible to represent an agent-based reward function as a state-based one by simply adding one additional placeholder state factor per agent to convey the rewards. Thus, we maintain a discrete factored Q-function as shown in Equation 4.28, which is updated after every interaction with the environment using an experience tuple $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r} \rangle$.

In previous work, single-agent PS updates the Q-function using either full or small backups of the value function [99, 103, 104], i.e. one-step planning using the learned model. Unfortunately, this approach is extremely computationally expensive when using factored value functions, as each backup requires solving a full linear program [17].

Algorithm 13: CPS

```
Input: The state and action spaces $\boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}$ and discount $\gamma$ of an MMDP model $\mathcal{M}$
       The DDN structure of $\mathcal{T}$ and $\mathcal{R}$ of $\mathcal{M}$
       A set of basis domains $x \in \mathcal{X}$ for the factored value function
Output: The optimal factored value function $\hat{Q}$ of $\mathcal{M}$
 1: Initialize each factored value function component $\hat{Q}_x(\cdot,\cdot) \leftarrow 0$
 2: Initialize each priority function component $\psi_f(\cdot,\cdot) \leftarrow 0$
 3: while True do
 4:     $\mathbf{a} \leftarrow$ Select action for state $\mathbf{s}$ following some exploratory policy $\pi_{\hat{Q}}$
 5:     $\langle \mathbf{s}', \mathbf{r} \rangle \leftarrow$ Execute joint action $\mathbf{a}$ in state $\mathbf{s}$
 6:     Update $\mathcal{T}, \mathcal{R}$ using $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r} \rangle$                    (see Eq. 4.27)
 7:     $\overset{\star}{\mathbf{a}}{}' \leftarrow \arg\max_{\mathbf{a}'} \hat{Q}(\mathbf{s}', \mathbf{a}')$ using VE
 8:     $\Delta_x \leftarrow$ Compute using $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \overset{\star}{\mathbf{a}}{}' \rangle$                    (see Eq. 4.32)
 9:     $\hat{Q}(\mathbf{s}, \mathbf{a}) \leftarrow$ Update using $\Delta_x$                    (see Eq. 4.33)
10:     for each state feature $f$ do
11:         $\Delta_f \leftarrow$ Compute from $\Delta_x$                    (see Eq. 4.29)
12:         for each pair $\langle \dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)} \rangle$ do
13:             $\psi_f(\dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)}) \leftarrow \psi_f(\dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)}) + |\Delta_f| \, \mathcal{T}(s'_f = s_f \mid \dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)})$    (see Eq. 4.25)
14:         end for
15:     end for
16:     for each batch update do
17:         $\langle \dot{\mathbf{s}}, \dot{\mathbf{a}} \rangle \leftarrow \arg\max_{\mathbf{s},\mathbf{a}} \psi(\mathbf{s}, \mathbf{a})$                    (approximate)
18:         for each pair $\langle \dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)} \rangle$ do
19:             $\psi_f(\dot{\mathbf{s}}^{(f)}, \dot{\mathbf{a}}^{(f)}) \leftarrow 0$
20:         end for
21:         $\langle \dot{\mathbf{s}}', \dot{\mathbf{r}} \rangle \leftarrow$ Sample from $\mathcal{T}, \mathcal{R}$ using $\dot{\mathbf{s}}^{(f)}$ and $\dot{\mathbf{a}}^{(f)}$
22:         Perform steps 7 to 15 using $\langle \dot{\mathbf{s}}, \dot{\mathbf{a}}, \dot{\mathbf{s}}', \dot{\mathbf{r}} \rangle$
23:     end for
24: end while
25: return $\hat{Q}$
```

Instead, we update the Q-function directly using real and synthetic experience, with the latter being sampled from the learned model. Our update step takes inspiration from the update step of Sparse Cooperative Q-learning [42] that we discussed in Section 4.1.6.

We can show how to derive our update rule from single-agent Q-learning:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma Q(s', \overset{\star}{a}{}') - Q(s,a) \right) \tag{4.30}$$

where α is the learning rate parameter.

In our setting, we need to adjust Equation 4.30 to update each Q_x component individually, while preserving the credit assignment information carried by the reward vector $\mathbf{r}$ regarding which state component produced which reward. Thus, we first decompose Equation 4.30 per state component, for each computing a target that takes into account only its associated Q factors:

$$\delta_f = r_f + \sum_x I(S_f \in \mathbf{S}^{\textcircled{x}}) \frac{\gamma Q_x(\mathbf{s}'^{\textcircled{x}}, \overset{\star}{\mathbf{a}}'^{\textcircled{x}}) - Q_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}})}{|\mathbf{S}^{\textcircled{x}}|} \tag{4.31}$$

where I is the indicator function, which we use to select only the Q factors with domain over S_f. Computing $\overset{\star}{\mathbf{a}}'$ requires maximizing the whole factored Q-function, which can be done efficiently using variable elimination (VE) [17].

From these δ_f we can compute the individual TD errors for each Q-function factor by simple redistribution:

$$\Delta_x = \alpha \left(\sum_f I(S_f \in \mathbf{S}^{\textcircled{x}}) \frac{\delta_f}{|\{x : i \in x\}|} \right) \tag{4.32}$$

We can finally update the Q-function, one factor at a time:

$$Q_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) = Q_x(\mathbf{s}^{\textcircled{x}}, \mathbf{a}^{\textcircled{x}}) + \Delta_x \tag{4.33}$$

CPS then recomputes the priorities using the new TD errors, following Equations 4.23-4.29.

Using Q-learning-like updates allows us to efficiently perform batch updates by sampling new experience data from the learned model, rather than performing full backups of the value function. We use the queue to select a joint state-action pair, and sample a next state and reward with the learned $\mathcal{T}$ and $\mathcal{R}$ functions. The priorities of the selected parents are set to zero, to mark that their values have been successfuly updated. Note that $\mathcal{R}$ can be learned using maximum-likelihood estimates similar to how $\mathcal{T}$ is learned in Equation 4.27. A pseudocode description of the full algorithm is shown in Algorithm 13.

4.2.3 Empirical Results

We evaluate the empirical performance of CPS against 4 benchmarks: a random policy as a naive approach, the factored LP planning algorithm on the ground truth

MMDP model as the upper bound [17], Sparse Cooperative Q-learning (SCQL) with and without randomized experience replay [42], and QMIX [16] as competing algorithms. QMIX is a neural network-based algorithm, designed to learn a factorization of the value function with one component per agent, so that during execution each agent can act independently by maximizing its own component.

The factored LP planning algorithm is trained in advance, and we show the performance of the final optimal policy. Note that the factored LP algorithm is provided with the true model of the environment, and performs planning, which guarantees optimal performance within the bounds of the approximated Q-function. Therefore, the LP results should be interpreted as the upper bound of what is reachable with the given factorization only. Unfortunately, planning is computationally intensive, and can only be performed on simple environments with few agents.

QMIX was trained over 300 episodes, with 1000 timesteps per episode, and we report the best results across many hyperparameter settings. We did not test QMIX on the larger environments, due to the excessive memory requirements (400 GB and 3600 GB for the 100 and 300 agents settings respectively).

SCQL and CPS use a constant learning rate of 0.3, and an ε-greedy policy with ε linearly decreasing from 0.9 to 0. Additionally, they use optimistic initialization to improve exploration. CPS, SCQL and the LP approach all use the same Q-function factorization. CPS has no prior knowledge about the transition and reward functions at the beginning of each run, i.e. all N^0 in Equation 4.27 are equal to zero, and performs 50 batch updates per timestep. When using experience replay, SCQL performs 50 batch updates from randomly selected previous experiences.

The training results for SCQL and CPS are shown in the line plots of Figure 4.6, and report the immediate rewards obtained at each timestep during a single, uninterrupted episode. The training results for QMIX are shown in Figure 4.7 and report average immediate reward for each training episode. The training results are shown in separate figures due to the significant time frame differences when learning, and because for QMIX the environment is reset after each episode, in contrast to the uninterrupted episode of CPS and SCQL. The histograms of Figure 4.6 report the performance of the trained policies for all algorithms. All results are averaged over 100 runs.

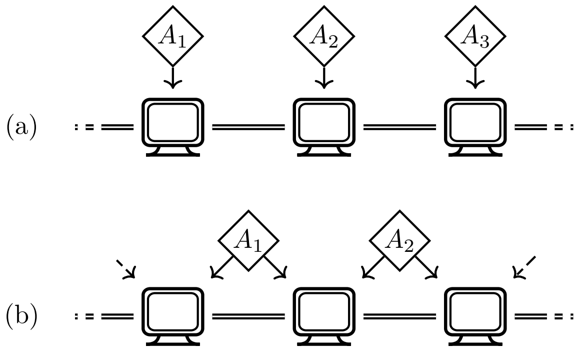


Figure 4.4: In the SysAdmin ring setting, each machine is directly connected to its two adjacent neighbors to form a ring, while each agent directly controls the actions of each machine (a). In the modified shared control setting, the connections between the machines are left unchanged. However, each agent is associated not with a machine but with a connection. The operations each machine performs depend on the joint action of the two adjacent agents (b).

SysAdmin

The SysAdmin setting simulates a set of interconnected machines that need to complete abstract jobs [17]. These jobs are obtained randomly when the machines are idle, and each job has a chance to complete at each timestep, which awards the associated agent a single reward point. Additionally, each machine has a chance to fail, which lowers the chance of completing jobs, or shutdown, which locks the machine completely. The failure of a machine also increases the chance of failing of its direct neighbors. Each machine can perform a binary action: keep working, or reboot, which recovers from failure and shutdown, but drops the current job if one is present.

We additionally introduce a modified ring setting with shared control, where each machine is controlled jointly by agents that are associated with the edges to its neighbors. The actions for the agents remain the same, i.e. work and reboot, but if the two adjacent agents send conflicting commands then a reboot is performed randomly. This setting requires a higher level of coordination than the original task, as agents must cooperate to control machines effectively.

We test on three different topologies: a ring with 300 agents, a torus with 100 agents in a 10×10 grid, and the modified shared control ring setting with 12 machines. A visualization of the two grid settings is shown in Figure 4.4. We select each machine's state and load pair as the basis domains for the Q-function factorization, which is equivalent to the single basis proposed in [17].

Random

We randomly generate MMDPs by connecting each S'_f with S_f, plus a random number of state and action nodes (maximum 3) locally close to i. For example, S'_1 might depend on S_3, but it will not depend on S_9. All state and action variables have 3 possible values, i.e. $|S_f| = |A_k| = 3$, and the number of agents K is $3/4$ of the number of state features N. The transition function is constructed using uniformly sampled transition vectors. We generate sparse reward functions containing random values in $\{-1, 0, 1\}$ for the parents of random nodes S'_f, selected with probability 0.3. We select $x = \{i, i+1\}$ as basis domains for the Q-function.

Firefighters

We modify the firefighter setting from Dec-POMDP literature [43, 105] by increasing the difficulty of coordination and removing partial observability. In our setting, houses are set in a toroidal grid, and we associate a single firefighter to each intersection. At each timestep, each firefighter agent must decide where it should go, selecting between one of the 4 buildings adjacent to it. Each house has a fire level from 0 to 2, which can increase or lower stochastically at every timestep. This probability depends on how many firefighters have selected that house and the average fire level of adjacent houses. At each timestep, each house returns a penalty equal to its negative fire level, so that higher fires correspond to negative reward.

We have designed and tuned the dynamics of the environment so that it is impossible to behave optimally without coordination between agents, as to make learning especially challenging. This was done by enforcing diminishing returns with respect to the number of firefighters in a single house, i.e. all firefighters going to the house with the highest fire level cannot result in optimal behavior. A visualization of the firefighting problem is shown in Figure 4.5. The basis domains in this setting are composed of all pairs of adjacent houses, to ensure that the Q-function can represent coordinating policies.

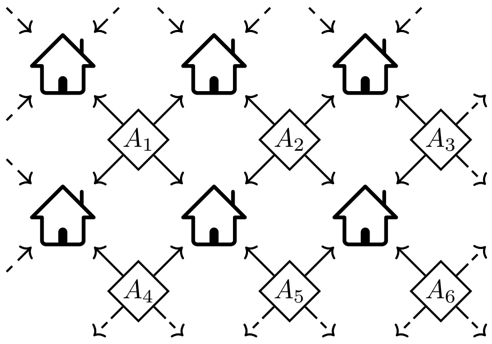
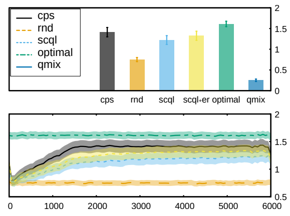


Figure 4.5: A representation of the firefighter setting. Houses are set in a toroidal grid. Every timestep, each agent goes to one of its 4 adjacent houses. The risk of fire for each house is determined by the number of firefighters at that house and by the average fire level in neighboring houses.

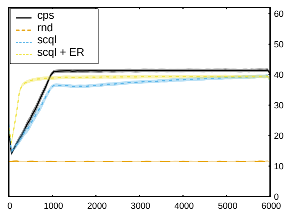
4.2.4 Discussion

In all performed experiments the sample-efficiency when learning using prioritized updates was consistently higher than for the other methods. The complexity of the underlying MMDP, i.e. the density of the DDN graph, and the amount of coordination required between the agents strongly affected the number of timesteps needed for the learning process to stabilize. This is expected in general as agents require more information to learn more challenging problems. As an example, learning with a fully-connected DNN, where all agents directly coordinate together, must always consider the entire exponential joint action space, which requires more data. Conversely, a fully-disconnected DDN is equivalent to independent agents, which greatly simplifies coordination.

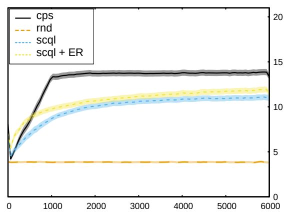
Because convergence to the optimal policy is not guaranteed when learning with approximate factored value functions, it is not surprising that CPS and SCQL end up on different greedy policies. In particular, both SCQL and CPS converge to lower quality policies than the best possible policy planned by the factored LP upper bound. In the shared control ring setting (Figure 4.6a), SCQL, SCQL-ER and CPS achieve 76%, 82% and 88% of the upper bound, respectively, while in the smaller random setting (Figure 4.6d), they achieve 59%, 94% and 96% of the upper bound. These two problems were the only settings small enough to determine the upper bound with the LP planning method. However, the policies trained by CPS



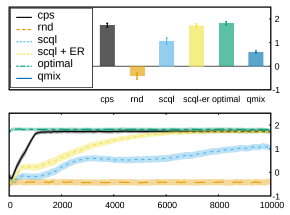
(a) Shared control ring setting with 12 agents.



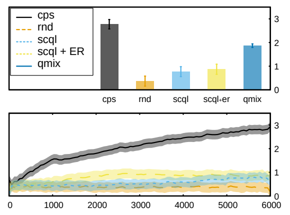
(b) Ring setting with 300 agents.



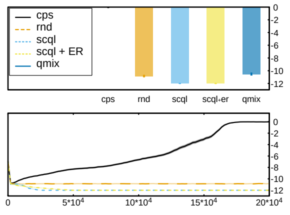
(c) Torus setting with 100 agents.



(d) Random setting with 3 agents.



(e) Random setting with 15 agents.



(f) Firefighter setting with 6 agents.

Figure 4.6: Histograms show average per-timestep reward over 1000 timesteps for all policies, after training. Line plots show the mean and standard error of per-timestep reward of CPS and SCQL during training, compared against a random policy and the LP planning upper bound. All data is averaged over 100 runs. Higher is better.

consistently outperform the ones trained by SCQL, with larger margins in the more complex environments. Experience replay does somewhat improve SCQL's performance, but the random sampling is unable to improve learning significantly in the more complex environments. This suggests that prioritizing updates is not only speeding up learning, but that the faster learning itself can have a positive influence on the final policies.

We experimented with different thresholds for the linear decay of the exploration probability ε. Longer exploration phases resulted in better policies, but not significantly so. The number of batch updates for CPS significantly affected learning speed, with more updates resulting in faster convergence times. However, the quality of the policy would often slightly degrade with more updates per timestep. This is due to the fact that we update our value function with data sampled from

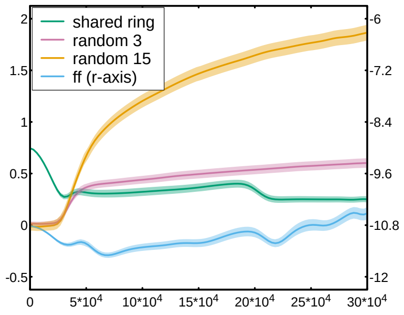


Figure 4.7: Best QMIX performance during training in all its environments, across hyperparameters. Both y-axes report mean and standard error for average per-timestep reward. A training episode corresponds to 1000 timesteps on the x-axis. Note the scale of the x-axis when comparing against Figure 4.6. Data is averaged over 100 runs. Higher is better.

the training model. Since during the initial phase of training this model is highly approximate, the resulting updates can negatively influence the value function in the long run. Further tests confirmed that this issue can be resolved by generating new data using a ground truth environment, such as a simulator, rather than by a learned model.

The firefighter setting, shown in Figure 4.6f, is interesting as CPS is the only method that was able to converge to the optimal policy, where no house is ever on fire. This setting is particularly challenging as the probability of fires increases as the average fire levels increase, requiring strong cooperation to extinguish fires when all houses are burning. QMIX does marginally better than the random policy, but its failure to improve is likely due to its inherent inability to represent non monotonic value functions and its per-agent Q-function factorization, which hinders cooperation. On the other hand, SCQL learns a policy which is worse than random selection, even when left exploring for a significant time. As the firefighter setting requires consistently good and coordinated actions to avoid negative reward, it is possible that SCQL is unable to learn fast enough to discover and retain the optimal policy. If it is left exploring for more, the exploratory actions result in negative rewards that get diffused in the factored Q-function, preventing the method from improving.

While less important, we remark that the use of factored value functions and priorities make CPS computationally and memory efficient. This turns out to be convenient when scaling to method to extremely large coordination tasks.

4.2.5 Conclusions

We have presented a new model-based algorithm, cooperative prioritized sweeping. CPS exploits domain knowledge in the form of a DDN, improving sample-efficiency in large-scale multi-agent RL tasks by detecting the best joint state-action pairs where to update the value function. CPS exploits the structure of a coordination graph in an MMDP to efficiently compute and store priorities for partial state-action pairs, and uses heuristics to quickly select the best update candidates.

We have demonstrated the effectiveness of these prioritized updates in an RL setting by using a learned model to generate new data to update the value function, and comparing CPS's performance to current state-of-the-art algorithms in several settings. CPS is computationally efficient and can be scaled to environments with hundreds of agents.

While in our experiments CPS samples new data from a learned model, we note that the same mechanism can be used to score existing experience data for experience replay, performing a similar job as prioritized experience replay [102] but in a more general way.

4.3 Gaussian Process Coop. Prioritized Sweeping

All algorithms presented in this thesis operate on the assumption of discrete agent actions and state features. However, such an assumption is unreasonable in many settings; a common example is the operation of analogue motors in the field of robotics, or when selecting an orientation for a wind turbine. Thus, it would be beneficial if there was a way for the learning process to work effectively even when the agents must deal with continuous environment.

The main challenge in extending CPS to these environments relates to the joint action maximization mechanism, which would need to be able to optimize across a continuous multi-dimensional joint action (we discuss this issue more in depth in Section 5.2.1). For this reason, we believe that it is not currently possible to directly deal with continuous actions in our MMDP setting. However, because during action selection in stateful environments we condition on the known state, continuous states do not impinge on the joint action maximization mechanism, and so can actually be supported. Still, doing so is not necessarily straightforward. In this section, we present some preliminary work in how CPS can be extended to continuous state environments.

4.3.1 Continuous States

Before we delve into the challenges of multi-agent learning in continuous state spaces, it is important to give some background on the assumptions required in this setting. In particular, the main difference that we need to be concerned about with respect to the discrete case is the definition of the transition function for the environment. While in a tabular setting $\mathcal{T}$ can be any arbitrary discrete distribution from a state-action pair to any set of future states, in the continuous case the transition function is generally taken to be continuous and deterministic — although some Gaussian noise can be tolerated. In other words, given a specific state and action pair, we will always reach (approximately) the same next state. The reason for this is that we cannot allow $\mathcal{T}(s, a)$ to map to an arbitrary continuous distribution; if that was the case the integrations — not summations anymore — over future states in the Bellman equation would simply not be tractable, and nothing could be learned. While these assumptions heavily restrict the shape of the transition function, they turn out to map really well onto our physical reality, and thus do not generally pose any practical problem.

The transition function being continuous and deterministic has some interesting consequences. The value functions and policy functions, which directly depend on $\mathcal{T}$, also end up more often than not being continuous and deterministic — if possibly with steeper gradients than $\mathcal{T}$. Thus, any learning strategy in this type of setting must rely on methods to learn arbitrary continuous functions that depend on continuous parameters. Two common alternatives in the field are respectively neural networks and Gaussian processes (GPs). We now give very brief introductions to both tools.

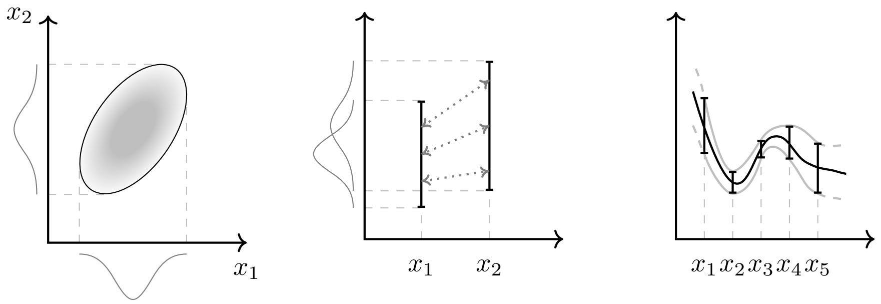
4.3.2 Neural Networks

In their simplest form, neural networks are function approximators that rely on layers of artificial neurons; each neuron's output is some non-linear transformation of the linear weighted combination of its inputs. The outputs of each layer are then fed as input to the next, until the process reaches the output neurons. Thus, a neural network encodes the function to learn within the weights between its neurons, and thus has an easier time "forgetting" the older shape of the value function as it does not store any specific state-value pair directly. Methods that rely on neural networks for value storage and manipulation are well established in the RL literature [101], and work has been done even in the case of handling factored value functions [16, 95, 96, 106, 107]. Neural networks are very flexible, commonly used in research, and demonstrated good computational performance in RL, especially given the recent GPU hardware support for them.

4.3.3 Gaussian Processes

Gaussian processes (GP) [108] are Bayesian function approximators with excellent sample-efficiency. In simple terms, a GP provides a posterior distribution over function shapes, provided a prior in the form a covariance function — or kernel —, and some data. In practice, a GP represents an infinite-dimensional multivariate normal distribution, where each dimension corresponds to a specific point in the domain of the function to learn. The kernel then correlates the dimensions, mapping the complete shape of the function, as well as our current uncertainty over its exact mean. A simple example is shown in Figure 4.8.

A neat property of a GP is that, even though it is defined over infinite dimensions, its underlying structure allows to easily query information about any finite set of points within its domain. For example, it is fairly straightforward to obtain the current expected mean and gradient of the function being learned at any given point. In some settings, it can be even computationally more efficient to use a learned GP in place of a function expensive to evaluate. Unfortunately, while a



(a) A 2-dimensional normal distribution.

(b) An alternative visualization of (a).

(c) An infinite-dimensional normal distribution.

Figure 4.8: (a) A simple 2-dimensional normal distribution, with each dimension mapped by one of the two axes. (b) The same distribution as before, with the two dimensions shown along the same vertical axis. In this representation the correlations between the two dimensions are not clearly visible anymore, although they are still present (see dotted arrows). (c) An infinite-dimensional normal distribution, or GP, is used to approximate an arbitrary function. This results in a distribution over functions, where the infinite dimensions of the GP correspond to the input domain of the function to model. The correlations will then influence the shape of the possible modeled functions, as well as the current uncertainty.

GP can be very efficient to evaluate — depending on what is required — optimizing its hyperparameters given the current data can be fairly expensive; cubic in the number of datapoints to be exact, due to a required matrix inversion to compute the derivative of its hyperparameters' likelihood. This restriction limits the use of GPs in an RL setting to learning functions with a fairly low dimensionality, lest the datapoint requirements grind the optimization process to a halt. However, as long as the functions to learn remain of limited scope, GPs are an excellent modeling choice.

4.3.4 The Algorithm

We here present the *Gaussian process Cooperative Prioritized Sweeping* (GCPS), our preliminary attempt to extend CPS to continuous state spaces. Because in our MMDP setting $\mathcal{T}$ is factored, the GPs we need to support are small in size and can be optimized and sampled in parallel, which should avoid unnecessary computational roadblocks.

The idea of GCPS is similar to that of CPS; to build a prioritized queue of state-action pairs that represent where the value function is likely in need of updates. Because of the continuous nature of the state space, the queue cannot be represented discretely; instead it is stored as sets of continuous distributions, each of which we call a *slice*. Recall that in CPS, after a single update to the value function, we also update the priority queue. These updates are themselves made of several step: during each step, CPS iterates over all possible parent sets for a single future state feature, assigning a score to those that are likely in need to be updated. In GCPS, each such update step generates a slice. We begin by going through the local $\mathcal{T}$ component associated with a future state feature and sampling points semi-uniformly throughout its domain — using for example Latin hypercube sampling. Once this is done, for each sampled point we perform a gradient descent step using the current maximum likelihood estimation of the GP, looking for those points that are closest to returning the input future state — thus finding probable state-action parent pairs. Note that we can do this specifically because we assume that the transition function is deterministic, and so either a state-action pair's expected future state is our input, or it is not.

Once this gradient descent step is done, we end up with points that are likely to be parents of the input future state. In CPS, a final joint state-action pair is then sampled by merging such points from all local priority queues (see Section 4.2.2). However, in GCPS's case, because we are in a continuous environment, our points obtained through gradient descent are extremely unlikely to match up. Thus, we convert our points to a continuous distribution by clustering them as a Gaussian mixture model. This can be done without knowing in advance the optimal number of clusters, using variational expectation maximization [109]. The points are additionally weighted by their likelihood of actually being parent of the target future states given the current GP uncertainty. The resulting GMM distribution is our slice. A simple visualization of this process is presented in Figure 4.9. Once obtained, a slice works as — and actually is — a distribution, and thus can be sampled with higher priority state-action pairs being more likely to be picked.

The priority queue for GCPS is thus composed of sets of slices, each set associated with a state feature. Note that after each update of the value function, we generate one additional slice for each state feature and local joint action; because each slice is a GMM, they can be easily combined by simply renormalizing the individual weights of their Gaussian components. Finally, because all sets of slices contained by the priority queue can be viewed as joint distribution over full

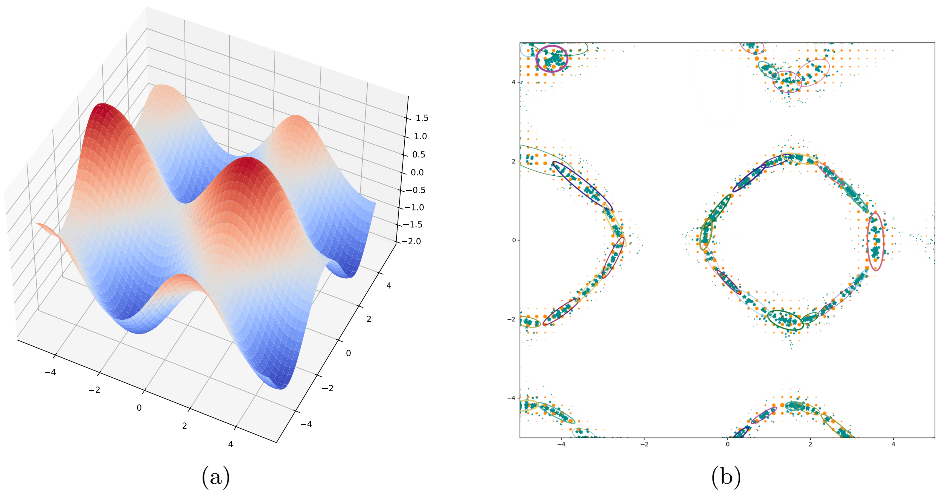


Figure 4.9: (a) Learned mean estimate of a GP using a squared-exponential kernel over a simple noisy continuous function with two inputs, and output on the z axis. (b) A *slice* of the same GP for a next state value of 0.5. The small orange dots represent a uniform sampling of the true pdf of the GP. The cyan stars are the points found using gradient descent. The ellipses represent the Gaussian mixture model approximating the true slice.

state-action pairs, we can sample from them jointly. In practice, this means sampling one local set of slices at a time, with each new sample conditioning on the previous samples. Once sampled, we discard the components of the priority queue that were used to generate the samples, thus completing the prioritized sweeping process.

We have now discussed how GCPS approaches the creation of the priority queue, but we have not yet talked about actually learning values. While GPs can be used to learn a continuous transition function effectively, they are unfortunately unsuited to learn a value function. The reason for this is that a GP learn by collecting datapoints about the function to learn, but in RL the value function changes — even dramatically — as we interact with the environment and update our current policy. Thus, older data about the value function becomes stale and poisons the GP's output. Instead, for this task neural networks are generally a better option. GCPS thus combines a neural network for value learning together with a set of GPs for model learning, into a single algorithm for model-based learning in continuous multi-agent environments.

4.3.5 Discussion

Unfortunately, while GCPS might introduce some novel ideas, it is still very much work-in-progress. A few challenges remain that we have not yet been able to overcome: how to ensure that the model generates correct priorities, and how to efficiently update the value function from our priority queue information.

The issue with the model is that, like CPS, GCPS's plan to achieve sample-efficiency relies on nesting priority queue updates: after each update to the value function, we recursively look at the parents of the latest update, and so on until the TD error of the last change is small enough. This is so we can perform multiple useful updates to the value function within a single environmental timestep. However, in the continuous setting, this deep reasoning about the $\mathcal{T}$ is extremely dependent on the accuracy on the model, as the dynamics of the environment are deterministic. As in the butterfly effect, a tiny error in the learned model can lead us to update state-action pairs that are extremely unlikely to actually lead to the initial state that started the update chain. This issue in general also rules out the generation of additional data until the model has a very high accuracy. This is a known issue in model-based approaches in continuous environments, one that has only recently seen some breakthroughs [110, 111].

The problems stemming from the value function relate to the interaction between the prioritized sweeping mechanism and the underlying learning process of neural networks. During the training phase neural networks can be prone to instability issues — also named *catastrophic forgetting* — unless certain precautions are taken. For example, it is important that the data samples that are used during each learning step are as independent of each other as possible, as constantly updating the same region of the value function may bias the network. This is generally achieved by separating the data collection phase, where we get data from our interactions with the environment, from the data feeding phase. Thus, we would first collect experience data into a buffer, and then randomly sample from it batches of data to feed to the neural network. Unfortunately, this randomization strongly contrasts with our goal of forcing specific updates to the value function by sampling important state-action pairs with our priority queue. While there are ways to dampen this effect via importance sampling [102], they rely on the ability of correctly calculating the probability of choosing each sample, which may not be obvious when using our priority queue.

A second issue is that the learning target of the neural network — in our case the actual values we want to learn — should not be allowed to wildly change after each update. In other words, we wish for the overall target value specified by the Bellman equation to be resistant to change, even as we update our network. As we cannot exert any influence over the immediate rewards we collect, this goal is achieved by using a separate fixed network to estimate the value of future states. This separate network is then only periodically updated to match our current neural network. However, these stability requirements heavily conflict with the entire concept of GCPS: determine where the value function is in need of updates, and quickly feed new data to alleviate the issue. Because our neural network is effectively not allowed to change too fast, the sample-efficiency advantages of GCPS are then mostly lost.

5 | Conclusions

Throughout this dissertation we have presented several novel methods to scale up reinforcement learning to settings with a large number of agents. In particular, our work focuses on tackling the *curse of dimensionality*, the intrinsic challenge that exists in all multi-agent environments which consists in the exponential increase in size of the state and action spaces as the number of agents grows larger. We showed how any successful approach must ensure that all functions that depend on these spaces — transition and reward functions, value functions, policy — can be represented and learned by the agents.

The common core to our contributions is the exploitation of *domain knowledge* in the form of coordination graphs and factorization of environments' dynamics: by leveraging known relationships between the agents and features of the environment, we can significantly simplify the space that the agents need to explore. When this is not enough, we additionally use factored representations to approximate the joint value function of the agents, so that it always remains manageable. While these approximations often come at the cost of convergence guarantees, they have been shown to obtain good results in practice. This is especially true when the actual shape of the factored function is determined by the prior domain knowledge, thus ensuring that it can model any important correlation between agents and state features.

Finally, all our approaches leverage some form of model-based learning. In particular, all our bandit algorithms model, in different ways, both the mean and uncertainty of each local arm to ensure that exploration takes both values into account. In our MMDP contribution, the CPS algorithm fully models the environment's dynamics to both efficiently determine the parts of the value function that are most in need to be updated, as well as generate additional data without requiring excessive interactions with the environment. The design choice of learning models meshes well with our exploitation of prior domain knowledge in the multi-agent domain, and further increases sample-efficiency when learning in very large environments. While model-based learning can be prone to systematic errors [110, 111], we showed how it can be employed successfully when learning with hundreds of agents concurrently, even in stateful environments.

5.1 Contributions

In Section 3.2, we introduced the MAUCE algorithm [11] for multi-agent regret minimization in multi-agent multi-armed bandit settings. MAUCE achieves a higher sample-efficiency than alternative approaches by carefully regulating the exploration performed by the agents. We have shown how the agents in MAUCE select full joint actions following a novel formulation of the upper-confidence bounds exploration strategy. While the standard UCB1 formulation would require estimating the mean value for all full joint actions, MAUCE's UCB definition is based on local estimates, taking advantage of the factorization of the problem. Key to efficiently computing the highest joint UCB score and executing a full joint action is the UCVE algorithm, which takes inspiration from the multi-objective literature to maximize exactly the local two-element UCB mean and exploration components. We have shown how MAUCE achieves a theoretical regret bound linear in the number of agents, rather than the exponential bound of prior work. This was further confirmed by our empirical experiments. Thus, we have shown MAUCE to be a provably efficient algorithm, both in terms of its sample efficiency and its computational requirements.

In Section 3.3, we introduced the MATS algorithm [23] for multi-agent regret minimization in MAMABs. MATS manages uncertainty through the use of Bayesian inference, maintaining a posterior distribution over the hyperparameters of the likelihoods of all arms. Through these posteriors, MATS is able to obtain samples for the expected means of each arm, which are then used to select the next full joint action. Thus, MATS always selects full joint actions with the same probability that they are optimal, simultaneously balancing the exploitation of its current knowledge as well as the exploration of possible optimal arms. We showed that MATS guarantees a regret bound that is sub-linear in time and low-order polynomial in the highest number of actions of a single agents, when rewards are σ-subgaussian with bounded means. MATS empirically outperforms in regret MAUCE as well as other state-of-the-art baselines, and demonstrates much higher computational performance than its competition.

In Section 3.4, we introduced the MARmax and MAVmax algorithms [24] for multi-agent best-arm identification in multi-agent bandit settings. Crucial for this setting, these algorithms provide theoretical PAC-guarantees of their performance, bounding the probability of obtaining an arm within some epsilon factor of the optimum. The core strategy of both algorithms is based on ensuring that, once recommended, a full joint action has had all its local components explored sufficiently often that any additional exploration is only likely to discover lower-valued joint actions. We have shown that the PAC bounds hold empirically for both algorithms in a series of experiments under different sets of PAC parameters, i.e. desired bound threshold ε and confidence δ.

In Section 4.2, we introduced the CPS algorithm [25] for cooperative learning in multi-agent sequential environments. Because large scale sequential environments cannot be solved exactly, CPS approximates the optimal value function using a factored decomposition. In particular, each component of the value function is specified by the user as dependent on the parent set of a particular set of state features, or *basis domain*. This allows the approximation to capture important value correlations between associated state features and agents, while simplifying the representation of the value function enough to keep it tractable. The main contribution of CPS consists in its *priority queue*, which can produce full state-action pairs where the current value function estimate is likely to lag behind the information collected through experience. This queue can be used to perform prioritized experience replay, or it can be used together with a generative model to generate additional training data. In our empirical evaluations, CPS manages to significantly speed up learning with respect to state-of-the-art baselines by focusing updates on the parts of the value function that actually need them. We have additionally shown that, in certain cases, faster learning tends to correlate to higher quality final policies. We speculate that, in the absence of convergence guarantees, faster learning might prevent the settling of incorrect values inside the approximate value function, thus allowing the agents' behaviors to evolve further than what they would have done otherwise.

5.2 Future Work

Our current contributions are subject to two main limitations that restrict their applicability to more general problems than those showcased in this thesis. Namely, the algorithms are all designed to work on discrete state features and action sets, and all rely on a *static* relationship model between the agents — i.e. the coordination graphs and/or DDNs are set during the initialization phase and never change afterwards.

We have discussed some preliminary work concerning continuous state spaces in Section 4.3. In this section we detail the current challenges and some possible avenues of approach regarding the other remaining issues.

5.2.1 Continuous Action Spaces

As we mentioned in Section 4.3, it would be beneficial if there was a way for the learning process to work effectively even when the agents must take continuous actions.

Unfortunately, it turns out that handling this additional complexity efficiently in a multi-agent setting is not trivial, and is still an open problem in general. While using continuous actions when learning would not be an issue per se — value functions and policies, even factored, can be learned with continuous actions — all our algorithms rely on coordination graphs and their associated optimizing algorithms (for example VE) in order to efficiently select the full joint actions to execute at each timestep. Thus, we would need some equivalent algorithm in order to maximize a coordination graph containing continuous payoff nodes.

This is an active field in optimization research, where many proposed algorithms try to tackle this problem. A simple approach relies on discretizing the action space and applying standard algorithms; the discretization can be tuned as the optimization progresses to improve the quality of the final solution [112]. Another common strategy is to restrict the family of payoff functions that must be maximized: for example, we might assume that each local payoff function is a piecewise-linear function, so that the local reward is a linear function of the actions of its connected agents [113]. A less restrictive alternative is to assume that payoff functions are Lipschitz continuous [114]. Another option is simply to sample the space enough so that the optimal joint action can be found [115].

Unfortunately, these approaches ultimately tend to be too expensive to be used in a reinforcement learning context, where one (or more) optimization problems need to be solved at each timestep. Furthermore, they are generally based on Max-Plus, and thus they inherit its convergence shortcomings when handling cyclic dependency graphs, which tend to be common in multi-agent problems (see Section 2.4.2). Thus, how to tackle efficient and exact large-scale multi-agent action selection with continuous actions remains an open question.

5.2.2 Approximating the Models

In our contributions we have assumed that the locality information about agent interactions is always available as prior information; even more importantly we assume that nodes and edges of the coordination graphs and DDNs do not change over time. Unfortunately, any algorithm that relies on these assumptions is restricted to settings where agents' relationships are static in nature — which is generally synonymous for immobile or abstract agents.

While many settings satisfy these constraints, it is hard to argue that they constitute the majority of multi-agent problems. The question then arises on whether these assumptions could be relaxed as to allow locality constraints to still exist but be dependent on the environment's current state. In this way, we could still exploit the intrinsic structure of a problem to improve our sample-efficiency, while broadening the range of problems that our methods can be applied to.

Unfortunately, requiring an amount of prior knowledge proportional to the size of the state space in settings where environments grow exponentially with the number of agents is not a reasonable approach. Not only obtaining this information exactly is unrealistic, but even if we could determine in advance all the possible local groupings of the agents, the resulting transition function would be so complex that there would likely be no savings in learning it exactly at all. The SCQL algorithm tries to tackle this exact issue with its sparse representation of the value function, which in theory decreases the size of the stored function but in practice ends up subjecting the algorithm to excessive bookkeeping costs.

Instead, we believe there might be promise in trying to learn an approximate model of the environment. Such a model would not — and could not — be used to generate additional data, but instead would be used to learn about correlations between subsets of state features and actions. For example, we could define a set of approximate transition functions:

$$\Gamma_i(s'_f|\mathbf{s}^i,\mathbf{a}^i) = \frac{\sum_t I(\mathbf{s}^i \in \mathbf{s}_t, \mathbf{a}^i \in \mathbf{a}_t, s'_f \in \mathbf{s}'_t)}{\sum_t I(\mathbf{s}^i \in \mathbf{s}_t, \mathbf{a}^i \in \mathbf{a}_t)} \tag{5.1}$$

where Γ_i is associated with the arbitrary parent subsets $\mathbf{s}^i$ and $\mathbf{a}^i$, and one — also arbitrary — future state feature s'_f. Note that the elements of parents sets might be chosen based on some assumption of relationship with s'_f, but are not guaranteed to correspond, in whole or in part, to some true $\mathcal{T}_f(s'_f|\mathbf{s}^f, \mathbf{a}^f)$. The value of each Γ_i is here learned via maximum likelihood estimation from experience data. Note how, if the chosen future variable does not depend on the chosen parent sets, the distribution represented by Γ_i will tend towards higher entropy, while the opposite will occur if a dependency does exist.

Such Γ functions cannot be used in a generative model as they do not actually model the dynamics of the environment correctly; yet they still contain useful information about which subsets of state features and actions are related during a timestep transition. Even more importantly, these learned functions are state-dependent, and so can be used to capture locality information that is dependent on the state of the environment. Thus, it might be possible to use such Γ functions to support an efficient prioritized sweeping algorithm in the absence of a DDN for a multi-agent environment.

The reason we believe this approach might be promising is that PS priorities do not really need to be exact values; they just need to be able to provide some sort of better-than-random ordering of state-action pairs that are likely in need to be updated. Recall from Equation 4.21 that priorities in PS are computed as:

$$\psi(s,a) = |\Delta_{s'}|\,\mathcal{T}(s'|\ s,a)$$

and that similarly CPS computes local priorities as:

$$\psi_f(\mathbf{s}^{(f)}, \mathbf{a}^{(f)}) = |\Delta_f|\,\mathcal{T}_f(s'_f \mid \mathbf{s}^{(f)}, \mathbf{a}^{(f)})$$

In our case we could use Γ to obtain something similar, so that:

$$\psi_i(\mathbf{s}^i, \mathbf{a}^i) = |\Delta_f|\,\Gamma_i(s'_f \mid \mathbf{s}^i, \mathbf{a}^i)$$

Computing the best joint state-action pairs to update using these priorities would work similarly to what CPS does, with the additional added care of normalizing priorities in case multiple Γ_i exist that refer to a given s'_f.

The main challenge in this approach is that, while there might be a possibility to factor the priority function so that it is not only manageable but also actually informative, there is no such simple solution for the value function. Without a full DDN it is much harder to define a useful and reasonable decomposition of the value function that can be learned and used to compute a policy. This in turn also affects the prioritized sweeping approach, as without access to local TD errors — the Δ — of the value function updates we cannot compute any priorities. Thus, this dissertation leaves this setting yet unresolved.

A AI-Toolbox

Decision theory is the branch of artificial intelligence that deals with action selection. Its applications span from robotics, to information gathering, to video games. The field tackles the problem of finding the optimal action-selection policy to perform in a given environment, and is roughly divided into planning and reinforcement learning. In the former case a model of the environment is known in advance [116], while in the latter case the agent must learn from its interactions with the world [8].

When new decision-theoretic algorithms are proposed, they need to be empirically compared to existing methods to prove their worth. However, this comparative work often requires re-implementing existing methods, which takes considerable effort. Furthermore, each new line of code increases the risk of bugs, and thus makes comparisons across implementations harder, if not impossible. It also diverts resources from research towards software maintenance. There is thus a great need in the research community for reusable, proven software that implements existing planning and reinforcement learning algorithms.

In this appendix, we present `AI-Toolbox` [26], a C++ framework containing implementations of reinforcement learning and planning algorithms for Markov decision process (MDPs) and its partially observable (POMDP) and multi-agent (MMDP) extensions. This framework has been used to implement virtually all the experimental sections in this work; with the sole exception of the QMIX algorithm.

A.1 Alternative Frameworks

Several libraries with partially overlapping functionality to `AI-Toolbox` exist.

`MADP` [117] is one of the best known toolboxes. It is written in C++, and geared towards multi-agent partially observable models, and offers a wide variety of algorithms. `MADP` is object oriented, which leads to a large hierarchy of classes, while `AI-Toolbox` has a more compact design. In addition, `MADP` does not have Python bindings.

`BURLAP` Is an extensive JAVA library for reinforcement learning and planning. It includes code to visualize environments and can be used with the ROS framework [118]. It is mostly focused on fully-observable environments, while `AI-Toolbox` contains multiple state-of-the-art POMDP algorithms.

`pomdp-solve` is a C library by Anthony Cassandra, which contains relatively old POMDP algorithms (the most recent was published in 2004). It additionally requires the commercially licensed CPLEX linear programming solver.

`MDPToolbox` [119] is a MATLAB toolbox for single agent MDP algorithms. In contrast, `AI-Toolbox` also supports bandits, POMDPs and MMDPs algorithms.

Other toolboxes exist, such as `PyMDPToolbox`, `JuliaPOMDP` [120], `ZMDP` and `APPL`, but they are much smaller in scope than `AI-Toolbox`.

A.2 Description

`AI-Toolbox` is written in C++20, taking advantage of all features of the language, and is built following modern standard practices, i.e. unit tests, continuous integration, separate concerns, etc. The goals of this framework are, in descending order of importance: usability and documentation, ease of modification, clarity and performance.

A.2.1 Features

Documentation. All public functions, classes and interfaces in the library are fully documented, and the implementation code is extensively commented (nearly 45% of the lines of code in the library are comments). The library's repository contains tutorials on how to use the library and examples, both in C++ and Python. The included unit tests can also be used as example implementations to understand how to use the library.

Architecture. In `AI-Toolbox` classes are organized in a template-based, mostly-flat hierarchy, which limits the amount of code that a user has to read in order to understand how a method works. Furthermore, extending the library only requires respecting few template interfaces. The only exception is for policy classes, which do not use templates but inheritance, as policies are designed to be composable. Thus, it may be necessary for custom policies to inherit from provided interfaces to leverage virtual dispatching.

The library is designed to simplify customization. We believe it is more important that users find easy to tune the code for their specific settings, rather than to provide a large number of options natively. To reach this goal, the API only offers parameters that do not increase the branching factor of the code. For example, an ε parameter to tune an epsilon-greedy policy is allowed, while a parameter that selectively enables an optimization is not. This choice allows users to customize the library for their particular usage more easily.

Classes are organized in separate folders and files, depending on their functions: algorithms reside in an `Algorithms` folder, policies in a `Policies` folder, and so on. Utility functions reside in `Utils` files and folders. Internal classes are hidden where possible in `Impl` namespaces and folders, so that users can safely ignore them.

`AI-Toolbox` is one of the largest libraries available in the field, currently containing over 40 learning and planning algorithms, 5 different policy types, and a large number of independent helper functions and classes. All algorithms are listed in the supplement.

Python Bindings. The framework offers Python bindings, to allow researchers who are not familiar with the C++ language to still benefit from its performance. The Python interface of the library is extremely close to its C++ counterpart, which results in a near line by line match between user code in Python and C++. The project's examples help to show the similarities, as they are implemented in both languages.

Performance. The C++ language allows for high computational performance compared to algorithms implemented purely in Python or MATLAB. `AI-Toolbox` has support for sparse matrices, yielding performance improvements on models with sparse transition, reward and observation tables. As an example of computational performance, our implementation of `GapMin` [121] takes only `0.8s` to reach results for a POMDP that in the original MATLAB implementation took `15s`—an improvement of nearly `19x`.

Community. The library has already been used in multiple projects, demos and papers, and has thus shown to be useful in practical applications and research.

The library is hosted on GitHub, where it has over 600 stars and has been forked 90 times by individuals and organizations. GitHub supports both an issue tracker, as well as pull request handling for contributing to the main codebase.

A.2.2 Considerations

Language Concerns. Nowadays, C++ is not so common in the research community anymore, due to its complexity. To mitigate this, we kept non-utility code as simple and accessible as possible in C++, and provide extensive documentation which does not require reading the code directly. In addition, Python bindings are provided, so that C++ does not need to be used at all, while still providing fast execution.
Non-Linear Function Approximation. Recently, there has been an incredible development in the field of AI regarding non-linear function approximation, e.g. deep learning. As of present, `AI-Toolbox` only supports linear function approximation in factored domains. However, more developments in this direction are currently planned.
Backwards Compatibility. As a software project evolves, there are often concerns of portability with newer versions. As additional methods are added to the library, sometimes we must make a choice on whether to keep old code as-is, or to refactor the library to improve the overall interface. Since one of the main goals of the library is clarity, we generally opt for refactoring, which may require users to update their code if they want to keep using the newest version. However, the documentation is always kept up to date, and all older versions are always accessible via version control.

A.3 Technical Details

The framework is easily built with CMake, a cross-platform tool that supports many operating systems. It additionally requires the `Boost` library, the `Eigen` matrix library and, for POMDPs and MMDPs, the `lp-solve` linear programming library. All dependencies are free and open source, to avoid having to buy expensive software licenses.

`AI-Toolbox` is released through the GPL 3.0 license, making it Free and Libre Open Source Software. It is versioned using the Git version control system, and it is hosted online on GitHub. It has been developed on an Ubuntu machine, but has also been successfully built on both Mac and Windows.

The library contains nearly 100 unit tests, each of which is composed of one or more tests. Adding and extending tests is easy, which allows to guarantee correctness even as the library is further developed. All tests are run using continuous integration, which alerts if any given commit fails to compile or breaks existing functionality.

A.4 Example

In this section we go over a brief example on how to use the incremental pruning POMDP algorithm on the tiger problem [122] using `AI-Toolbox`. This example is in C++, but the (nearly identical) equivalent example in Python is provided in the repository.

```cpp
// The model can be any custom class that respects a 10-method interface.
auto model = makeTigerProblem();
unsigned horizon = 10; // The horizon of the solution.

// The 0.0 is the convergence parameter. It gives a way to stop the
// computation if the policy has converged before the horizon.
AIToolbox::POMDP::IncrementalPruning solver(horizon, 0.0);

// Solve the model and obtain the optimal value function.
auto [bound, valueFunction] = solver(model);

// We create a policy from the solution to compute the agent's actions.
// The parameters are the size of the model (SxAxO), and the value function.
AIToolbox::POMDP::Policy policy(2, 3, 2, valueFunction);

// We begin a simulation with a uniform belief. We sample from the belief
// in order to get a "real" state for the world, since this code has to
// both emulate the environment and control the agent.
AIToolbox::POMDP::Belief b(2); b << 0.5, 0.5;
auto s = AIToolbox::sampleProbability(b.size(), b, rand);

// We sample the first action. The id is to follow the policy tree later.
auto [a, id] = policy.sampleAction(b, horizon);

double totalReward = 0.0;// As an example, we store the overall reward.
for (int t = horizon - 1; t >= 0; --t) {
    // We advance the world one step.
    auto [s1, o, r] = model.sampleSOR(s, a);
    totalReward += r;

    // We select our next action from the observation we got.
    std::tie(a, id) = policy.sampleAction(id, o, t);

    s = s1; // Finally we update the world for the next timestep.
}
```

A.5 Implemented Algorithms

In this section we list all the RL and planning algorithms currently implemented by `AI-Toolbox` at the time of writing. The algorithms are divided by the type of model when applicable. Note that this list does not include the large number of utility function that `AI-Toolbox` provides.

In the following lists two types of structures are listed: value-based algorithms and policies (exploration strategies and actor-critic). The library considers as value-based algorithms all classes and functions that relate to planning and learning through value functions. The library considers policies all classes and functions that use the learned values in order to produce a policy, or that learn a policy directly. This division is approximate as each algorithm is unique, but it results in a rough split where algorithms in the same group tend to have similar API. Note that different classes of models may list the same type of policy, as policies that apply to different models must have different API from each other.

The library contains detailed documentation for each class, along with a brief descriptive summary of each utility free function in the online documentation.

A.5.1 Bandit and Normal Games

Policies:

- Exploring selfish reinforcement learning (ESRL) [123]
- Q-Greedy policy [53]
- Softmax policy
- Linear reward penalty [124]
- Thompson sampling [75, 125]
- Top-Two Thompson Sampling [126]
- Successive Rejects [55]
- T3C [54]

A.5.2 Single Agent MDP and Stochastic Games

Value-based algorithms:

- Dyna-Q [36]

- Dyna2 [37]
- Expected SARSA [127]
- Hysteretic Q-Learning [128]
- Importance Sampling [129]
- Linear programming [97]
- Monte carlo tree search (MCTS) [40]
- Policy evaluation [130]
- Policy iteration [130]
- Prioritized sweeping [38]
- Q-learning [99]
- Double Q-learning [131]
- Q(λ) [132]
- R-Learning [133]
- SARSA [134]
- SARSA(λ) [134]
- Retrace(λ) [135]
- Tree backup(λ) [129]
- Value iteration [136]

Policies:

- Q-greedy policy
- Epsilon-greedy policy
- Softmax policy
- PGA-APP [137]
- Win or learn fast policy iteration (WoLF) [10]

A.5.3 Single Agent POMDP

Value-based algorithms:

- Augmented MDP (AMDP) [138, 139]
- Blind strategies [140]
- Fast informed bound [141]
- GapMin [121]
- Incremental pruning [142]
- Linear support [143]
- PERSEUS [144]
- POMCP with UCB1 [145]
- Point based value iteration (PBVI) [146]
- QMDP [139, 147]
- Real-time belief state search (RTBSS) [148]
- SARSOP [149]
- Witness [150]
- rPOMCP [151]

Policies:

- Policy from value function

A.5.4 Coordination Graph Optimization

- Variable elimination [17, 152, 153]
- Multi-objective variable elimination [67, 154]
- Upper confidence variable elimination (UCVE) [11]
- Max-Plus [48]
- Local Search [50]
- Reusing Iterative Local Search [50]

A.5.5 Factored/Joint Multi-Agent Bandits

Policies:

- Q-greedy policy
- Multi-agent Thompson sampling (MATS) [23]
- Multi-agent upper confidence exploration (MAUCE) [11]
- Learning with linear rewards (LLR) [155]
- MARmax [24]
- MAVmax [24]

A.5.6 Factored/Joint Multi-Agent MDP

Value-based algorithms:

- Cooperative Prioritized Sweeping [25]
- Factored LP [17]
- Multi-agent linear programming [17]
- Joint action learners [156]
- Sparse cooperative Q-learning [42]

Policies:

- Naive single action policy
- Epsilon-greedy policy
- Q-greedy policy

Bibliography

[1] X. Li, J. Zhang, J. Bian, Y. Tong, and T.-Y. Liu, “A cooperative multi-agent reinforcement learning framework for resource balancing in complex logistics network,” *arXiv preprint arXiv:1903.00714*, 2019.

[2] M. Roesch, C. Linder, R. Zimmermann, A. Rudolf, A. Hohmann, and G. Reinhart, “Smart grid for industry using multi-agent reinforcement learning,” *Applied Sciences*, vol. 10, no. 19, p. 6900, 2020.

[3] M. Wiering, “Multi-agent reinforcement learning for traffic light control,” in *Machine Learning: Proceedings of the Seventeenth International Conference (ICML’2000)*, 2000, pp. 1151–1158.

[4] D. Claes, F. Oliehoek, H. Baier, and K. Tuyls, “Decentralised online planning for multi-robot warehouse commissioning,” in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2017, pp. 492–500.

[5] J. Lussange, I. Lazarevich, S. Bourgeois-Gironde, S. Palminteri, and B. Gutkin, “Modelling stock markets by multi-agent reinforcement learning,” *Computational Economics*, vol. 57, pp. 113–147, 2021.

[6] P. Gebraad and J.-W. van Wingerden, “Maximum power-point tracking control for wind farms,” *Wind Energy*, vol. 18, no. 3, pp. 429–447, 2015.

[7] D. Silver *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] R. Sutton and A. Barto, *Reinforcement learning: An introduction.* MIT press Cambridge, 1998, vol. 1.

[9] C. Boutilier, “Planning, learning and coordination in multiagent decision processes,” in *TARK 1996: Proceedings of the 6th conference on Theoretical aspects of rationality and knowledge*, 1996, pp. 195–210.

[10] M. Bowling and M. Veloso, “Rational and convergent learning in stochastic games,” in *International Joint Conference on Artificial Intelligence*, Lawrence Erlbaum Associates Ltd, vol. 17, 2001, pp. 1021–1026.

[11] E. Bargiacchi, T. Verstraeten, D. Roijers, A. Nowé, and H. van Hasselt, “Learning to coordinate with coordination graphs in repeated single-stage multi-agent decision problems,” in *International Conference on Machine Learning*, vol. 80, PMLR, Oct. 2018, pp. 482–490.

[12] R. Rădulescu, M. Legrand, K. Efthymiadis, D. Roijers, and A. Nowé, “Deep multi-agent reinforcement learning in a homogeneous open population,” in *Artificial Intelligence: 30th Benelux Conference, BNAIC 2018,‘s-Hertogenbosch, The Netherlands, November 8–9, 2018, Revised Selected Papers 30*, Springer, 2019, pp. 90–105.

[13] I. Chao, O. Ardaiz, and R. Sanguesa, “Tag mechanisms evaluated for coordination in open multi-agent systems,” in *Engineering Societies in the Agents World VIII: 8th International Workshop, ESAW 2007, Athens, Greece, October 22-24, 2007, Revised Selected Papers 8*, Springer, 2008, pp. 254–269.

[14] H. Zheng, P. Wei, J. Jiang, G. Long, Q. Lu, and C. Zhang, “Cooperative heterogeneous deep reinforcement learning,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 455–17 465, 2020.

[15] Y. Ishiwaka, T. Sato, and Y. Kakazu, “An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning,” *Robotics and Autonomous Systems*, vol. 43, no. 4, pp. 245–256, 2003.

[16] T. Rashid, M. Samvelyan, C. De Witt, G. Farquhar, J. Foerster, and S. Whiteson, “QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning,” in *International Conference on Machine Learning*, 2018, pp. 4292–4301.

[17] C. Guestrin, D. Koller, and R. Parr, “Multiagent planning with factored MDPs,” in *Advances in neural information processing systems*, 2002, pp. 1523–1530.

[18] D. Koller and R. Parr, “Policy iteration for factored mdps,” in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, 2000, pp. 326–334.

[19] K. Zhang, Z. Yang, and T. Başar, “Multi-agent reinforcement learning: A selective overview of theories and algorithms,” *Handbook of reinforcement learning and control*, pp. 321–384, 2021.

[20] J. Subramanian, R. Seraj, and A. Mahajan, “Reinforcement learning for mean-field teams,” in *Workshop on Adaptive and Learning Agents at International Conference on Autonomous Agents and Multi-Agent Systems*, 2018.

[21] C. Guestrin, D. Koller, and R. Parr, “Max-norm projections for factored MDPs,” in *Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 2001, pp. 673–682.

[22] M. Van Dijk, J. Wingerden, T. Ashuri, Y. Li, and M. Rotea, “Yaw-misalignment and its impact on wind turbine loads and wind farm power output,” *Journal of Physics: Conference Series*, vol. 753, no. 6, 2016.

[23] T. Verstraeten, E. Bargiacchi, P. Libin, J. Helsen, D. Roijers, and A. Nowé, “Multi-agent Thompson sampling for bandit applications with sparse neighbourhood structures,” *Nature Scientific Reports*, vol. 10, no. 1, p. 6728, 2020.

[24] E. Bargiacchi, R. Avalos, T. Verstraeten, P. Libin, A. Nowé, and D. Roijers, “Multi-agent rmax for multi-agent multi-armed bandits,” *ALA 2022: Adaptive Learning Agents Workshop at AAMAS*, 2022.

[25] E. Bargiacchi, T. Verstraeten, and D. Roijers, “Cooperative prioritized sweeping,” in *AAMAS*, 2021, pp. 160–168.

[26] E. Bargiacchi, D. Roijers, and A. Nowé, “`AI-Toolbox`: A C++ library for reinforcement learning and planning (with Python bindings),” *Journal of Machine Learning Research*, vol. 21, no. 102, pp. 1–12, 2020. [Online]. Available: `http://jmlr.org/papers/v21/18-402.html`.

[27] D. Koller and R. Parr, “Computing factored value functions for policies in structured MDPs,” p. 8, 1999.

[28] A. Chapman, D. Leslie, A. Rogers, and N. Jennings, “Convergent learning algorithms for unknown reward games,” *SIAM Journal on Control and Optimization*, vol. 51, no. 4, pp. 3154–3180, 2013.

[29] D. Koller and R. Parr, “Policy iteration for factored MDPs,” in *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 326–334.

[30] C. Guestrin, S. Venkataraman, and D. Koller, "Context-specific multiagent coordination and planning with factored mdps," in *AAAI/IAAI*, 2002, pp. 253–259.

[31] M. Kroon and S. Whiteson, "Automatic feature selection for model-based reinforcement learning in factored mdps," in *2009 International Conference on Machine Learning and Applications*, IEEE, 2009, pp. 324–330.

[32] A. Strehl, C. Diuk, and M. Littman, "Efficient structure learning in factored-state mdps," in *AAAI*, vol. 7, 2007, pp. 645–650.

[33] C. Vigorito and A. Barto, "Incremental structure learning in factored mdps with continuous states and actions," *University of Massachusetts Amherst-Department of Computer Science, Tech. Rep*, 2009.

[34] T. Brys, Y.-M. De Hauwere, A. Nowé, and P. Vrancx, "Local coordination in online distributed constraint optimization problems," in *Multi-Agent Systems: 9th European Workshop, EUMAS 2011, Maastricht, The Netherlands, November 14-15, 2011. Revised Selected Papers 9*, Springer, 2012, pp. 31–47.

[35] A. Polydoros and L. Nalpantidis, "Survey of model-based reinforcement learning: Applications on robotics," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 153–173, 2017.

[36] R. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Machine Learning Proceedings 1990*, Elsevier, 1990, pp. 216–224.

[37] D. Silver, R. Sutton, and M. Müller, "Sample-based learning and search with permanent and transient memories," in *International Conference on Machine Learning*, ACM, 2008, pp. 968–975.

[38] A. Moore and C. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine learning*, vol. 13, no. 1, pp. 103–130, 1993.

[39] D. Andre, N. Friedman, and R. Parr, "Generalized prioritized sweeping," in *Advances in Neural Information Processing Systems*, 1998, pp. 1001–1007.

[40] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *International Conference on Computers and Games*, Springer, 2006, pp. 72–83.

[41] Y. Ouyang, M. Gagrani, A. Nayyar, and R. Jain, "Learning unknown markov decision processes: A thompson sampling approach," *Advances in neural information processing systems*, vol. 30, 2017.

[42] J. Kok and N. Vlassis, "Sparse cooperative Q-learning," in *International Conference on Machine Learning*, ACM, 2004, pp. 61–68.

[43] F. Oliehoek and C. Amato, *A concise introduction to decentralized POMDPs*. Springer, 2016, vol. 1.

[44] F. Fioretto, E. Pontelli, and W. Yeoh, "Distributed constraint optimization problems and applications: A survey," *Journal of Artificial Intelligence Research*, vol. 61, pp. 623–698, 2018.

[45] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[46] A. Petcu and B. Faltings, "A scalable method formultiagent constraint optimization," in *9thInternational Joint Conference on Artificial Intelligence, pages266*, vol. 271, 2005.

[47] F. Brioschi and S. Even, "Minimizing the number of operations in certain discrete-variable optimization problems," *Operations Research*, vol. 18, no. 1, pp. 66–81, 1970.

[48] J. Kok and N. Vlassis, "Using the max-plus algorithm for multiagent decision making in coordination graphs," in *RoboCup 2005: Robot Soccer World Cup IX 9*, Springer, 2006, pp. 1–12.

[49] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[50] R. Petri, E. Bargiacchi, H. Aldewereld, and D. Roijers, "Heuristic coordination in cooperative multi-agent reinforcement learning," in *Proceedings van de 33rd Benelux Conference on Artificial Intelligence en 30th Belgian Dutch Conference on Machine Learning (BNAIC/BENELEARN 2021)*, Hogeschool Utrecht, 2021.

[51] D. Roijers, "Multi-objective decision-theoretic planning," *AI Matters*, vol. 2, no. 4, pp. 11–12, 2016.

[52] S. Wright, "Coordinate descent algorithms," *Mathematical programming*, vol. 151, no. 1, pp. 3–34, 2015.

[53] D. Russo, B. Van Roy, A. Kazerouni, and I. Osband, *A tutorial on Thompson sampling*, arXiv:1707.02038, 2017.

[54] X. Shang, R. Heide, P. Menard, E. Kaufmann, and M. Valko, "Fixed-confidence guarantees for bayesian best-arm identification," in *International Conference on Artificial Intelligence and Statistics*, PMLR, 2020, pp. 1823–1832.

[55] J.-Y. Audibert, S. Bubeck, and R. Munos, "Best arm identification in multi-armed bandits.," in *COLT*, 2010, pp. 41–53.

[56] N. Cesa-Bianchi and G. Lugosi, "Combinatorial bandits," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1404–1422, 2012.

[57] W. Chen, Y. Wang, and Y. Yuan, "Combinatorial multi-armed bandit: General framework, results and applications," in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 151–159.

[58] Y. Gai, B. Krishnamachari, and R. Jain, "Combinatorial network optimization with unknown variables: Multi-armed bandits with linear rewards and individual observations," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 5, pp. 1466–1478, 2012.

[59] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *arXiv preprint:1204.5721*, 2012.

[60] J.-Y. Audibert, S. Bubeck, and G. Lugosi, "Minimax policies for combinatorial prediction games.," in *COLT*, vol. 19, 2011, pp. 107–132.

[61] S. Agrawal and N. Goyal, "Further optimal regret bounds for Thompson sampling," in *Artificial intelligence and statistics*, 2013, pp. 99–107.

[62] S. Agrawal and N. Goyal, "Analysis of Thompson sampling for the multi-armed bandit problem.," in *COLT*, 2012, pp. 39–1.

[63] V. Gabillon, M. Ghavamzadeh, and A. Lazaric, "Best arm identification: A unified approach to fixed budget and fixed confidence," *Advances in Neural Information Processing Systems*, vol. 25, 2012.

[64] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[65] D. M. Roijers and S. Whiteson, "Multi-objective decision making," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 11, no. 1, pp. 1–129, 2017.

[66] D. Roijers, S. Whiteson, and F. Oliehoek, “Multi-objective variable elimination for collaborative graphical games,” in *International Conference on Autonomous Agents and Multi-Agent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1209–1210.

[67] D. Roijers, S. Whiteson, and F. Oliehoek, “Computing convex coverage sets for faster multi-objective coordination,” *Journal of Artificial Intelligence Research*, vol. 52, pp. 399–443, 2015.

[68] E. Rollón and J. Larrosa, “Bucket elimination for multiobjective optimization problems,” *Journal of Heuristics*, vol. 12, pp. 307–328, 2006.

[69] P. Auer and R. Ortner, “UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem,” *Periodica Mathematica Hungarica*, vol. 61, no. 1-2, pp. 55–65, 2010.

[70] C.-P. Chen and F. Qi, “The best lower and upper bounds of harmonic sequence,” *RGMIA research report collection*, vol. 6, no. 2, 2003.

[71] J. Kok and N. Vlassis, “Collaborative multiagent reinforcement learning by payoff propagation,” *Journal of Machine Learning Research*, vol. 7, pp. 1789–1828, Dec. 2006.

[72] NREL, *FLORIS. Version 1.0.0*, 2019. [Online]. Available: `https://github.com/NREL/floris`.

[73] International Electrotechnical Commission, *Wind turbines – Part 4: Design requirements for wind turbine gearboxes (No. IEC 61400-4)*, accessed 6 March 2019, 2012. [Online]. Available: `https://www.iso.org/standard/44298.html`.

[74] T. Verstraeten, “A multi-agent reinforcement learning approach to wind farm control,” Ph.D. dissertation, Vrije Universiteit Brussel, 2021.

[75] W. Thompson, “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples,” *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[76] O. Chapelle and L. Li, “An empirical evaluation of Thompson sampling,” in *Advances in neural information processing systems*, 2011, pp. 2249–2257.

[77] R. Vershynin, *High-dimensional probability: An introduction with applications in data science*. Cambridge University Press, 2018, vol. 47.

[78] T. Lattimore and C. Szepesvári, “Bandit algorithms,” *preprint*, 2018.

[79] D. Russo and B. Van Roy, "Learning to optimize via posterior sampling," *Mathematics of Operations Research*, vol. 39, no. 4, pp. 1221–1243, 2014.

[80] C. Robert, *The Bayesian choice: from decision-theoretic foundations to computational implementation*. Springer Science & Business Media, 2007.

[81] D. Lunn, C. Jackson, N. Best, D. Spiegelhalter, and A. Thomas, *The BUGS book: A practical introduction to Bayesian analysis*. Chapman and Hall/CRC, 2012.

[82] J. Honda and A. Takemura, "Optimality of thompson sampling for gaussian bandits depends on priors," in *Artificial Intelligence and Statistics*, 2014, pp. 375–383.

[83] R. Brafman and M. Tennenholtz, "R-max - a general polynomial time algorithm for near-optimal reinforcement learning," *Journal of Machine Learning Research*, vol. 3, no. Oct, pp. 213–231, 2002.

[84] A. Strehl, L. Li, and M. Littman, "Reinforcement learning in finite MDPs: PAC analysis," *Journal of Machine Learning Research*, vol. 10, no. 11, 2009.

[85] K. Rao, S. Whiteson, *et al.*, "V-MAX: Tempered optimism for better PAC reinforcement learning.," in *AAMAS*, 2012, pp. 375–382.

[86] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[87] C. Boutilier, "Sequential optimality and coordination in multiagent systems," in *IJCAI*, vol. 99, 1999, pp. 478–485.

[88] D. Bernstein, R. Givan, N. Immerman, and S. Zilberstein, "The complexity of decentralized control of markov decision processes," *Mathematics of operations research*, vol. 27, no. 4, pp. 819–840, 2002.

[89] R. Becker, S. Zilberstein, V. Lesser, and C. Goldman, "Solving transition independent decentralized markov decision processes," *Journal of Artificial Intelligence Research*, vol. 22, pp. 423–455, 2004.

[90] R. Nair, P. Varakantham, M. Tambe, and M. Yokoo, "Networked distributed pomdps: A synthesis of distributed constraint optimization and pomdps," in *AAAI*, vol. 5, 2005, pp. 133–139.

[91] F. Oliehoek, C. Amato, *et al.*, *A concise introduction to decentralized POMDPs*. Springer, 2016, vol. 1.

[92] C. Guestrin, D. Koller, R. Parr, and S. Venkataraman, "Efficient solution algorithms for factored MDPs," *Journal of Artificial Intelligence Research*, vol. 19, pp. 399–468, 2003.

[93] J. Castellini, F. Oliehoek, R. Savani, and S. Whiteson, "The representational capacity of action-value networks for multi-agent reinforcement learning," English, in *AAMAS'19*, International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2019, pp. 1862–1864, ISBN: 978-1-4503-6309-9.

[94] P. Hernandez-Leal, B. Kartal, and M. Taylor, "Is multiagent deep reinforcement learning the answer or the question? A brief survey," *CoRR*, vol. abs/1810.05587, 2018.

[95] P. Sunehag *et al.*, "Value-decomposition networks for cooperative multi-agent learning based on team reward.," in *AAMAS*, 2018, pp. 2085–2087.

[96] W. Böhmer, V. Kurin, and S. Whiteson, "Deep coordination graphs," in *International Conference on Machine Learning*, PMLR, 2020, pp. 980–991.

[97] P. Schweitzer and A. Seidmann, "Generalized polynomial approximations in Markovian decision processes," *Journal of mathematical analysis and applications*, vol. 110, no. 2, pp. 568–582, 1985.

[98] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[99] C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.

[100] R. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[101] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.

[102] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[103] H. van Seijen and R. Sutton, "Efficient Planning in MDPs by Small Backups," in *Proc. 30th Int. Conf. Mach. Learn.*, vol. 28, 2013, pp. III–361.

[104] J. Peng and R. Williams, "Efficient Learning and Planning Within the Dyna Framework," *Adaptive Behavior*, vol. 1, no. 4, pp. 437–454, Mar. 1993.

[105] F. Oliehoek, M. Spaan, and N. Vlassis, "Optimal and approximate q-value functions for decentralized pomdps," *Journal of Artificial Intelligence Research*, vol. 32, pp. 289–353, 2008.

[106] R. Avalos, M. Reymond, A. Nowé, and D. Roijers, "Local advantage networks for cooperative multi-agent reinforcement learning," in *21st International Conference on Autonomous Agents and Multi-agent System*, IFAAMAS, 2022.

[107] J. Wang, Z. Ren, T. Liu, Y. Yu, and C. Zhang, "Qplex: Duplex dueling multi-agent q-learning," *arXiv preprint arXiv:2008.01062*, 2020.

[108] C. Rasmussen and C. Williams, *Gaussian processes for machine learning.* (Adaptive computation and machine learning). MIT Press, 2006, pp. I–XVIII, 1–248, ISBN: 026218253X.

[109] C. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag, 2006, ISBN: 0387310738.

[110] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, "Dream to control: Learning behaviors by latent imagination," *arXiv preprint arXiv:1912.01603*, 2019.

[111] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, "Mastering diverse domains through world models," *arXiv preprint arXiv:2301.04104*, 2023.

[112] T. Voice, R. Stranders, A. Rogers, and N. Jennings, "A hybrid continuous max-sum algorithm for decentralised coordination.," in *ECAI*, 2010, pp. 61–66.

[113] R. Stranders, A. Farinelli, A. Rogers, and N. Jennings, "Decentralised control of continuously valued control parameters using the max-sum algorithm," 2009.

[114] J. Fransman, J. Sijs, H. Dol, E. Theunissen, and B. De Schutter, "Distributed bayesian: A continuous distributed constraint optimization problem solver," *Journal of Artificial Intelligence Research*, vol. 76, pp. 393–433, 2023.

[115] D. Nguyen, W. Yeoh, and H. Lau, "Distributed gibbs: A memory-bounded sampling-based dcop algorithm," 2013.

[116] C. Boutilier, T. Dean, and S. Hanks, "Planning under uncertainty: Structural assumptions and computational leverage," in *Proceedings of the Second European Workshop on Planning*, 1995, pp. 157–171.

[117] F. Oliehoek, M. T. J. Spaan, B. Terwijn, P. Robbel, and J. V. Messias, “The MADP toolbox: An open-source library for planning and learning in (multi-)agent systems,” *Journal of Machine Learning Research*, vol. 18, no. 1, pp. 3112–3116, 2017.

[118] M. Quigley *et al.*, “Ros: An open-source robot operating system,” in *International Conference on Robotics and Automation Workshop on Open Source Software*, 2009.

[119] I. Chades, G. Chapron, M. Cros, F. Garcia, and R. Sabbadin, “MDPtoolbox: A multi-platform toolbox to solve stochastic dynamic programming problems.,” in *Ecography*, vol. 37, 2014, pp. 916–920.

[120] M. Egorov, Z. Sunberg, E. Balaban, T. Wheeler, J. Gupta, and M. Kochenderfer, “POMDPs.jl: A framework for sequential decision making under uncertainty,” *Journal of Machine Learning Research*, vol. 18, no. 26, pp. 1–5, 2017. [Online]. Available: `http://jmlr.org/papers/v18/16-300.html`.

[121] P. Poupart, K. Kim, and D. Kim, “Closing the gap: Improved bounds on optimal POMDP solutions.,” in *International Conference on Automated Planning and Scheduling*, 2011.

[122] A. Cassandra, L. Kaelbling, and M. Littman, “Acting optimally in partially observable stochastic domains,” in *AAAI Conference on Artificial Intelligence*, Seattle, WA, 1994.

[123] K. Verbeeck, A. Nowé, J. Parent, and K. Tuyls, “Exploring selfish reinforcement learning in repeated games with stochastic rewards,” *Autonomous Agents and Multi-Agent Systems*, vol. 14, no. 3, pp. 239–269, 2007.

[124] C. Unsal, “Self-organization in large populations of mobile robots,” Ph.D. dissertation, Virginia Tech, 1993.

[125] N. Korda, E. Kaufmann, and R. Munos, “Thompson sampling for 1-dimensional exponential family bandits,” in *Advances in Neural Information Processing Systems*, 2013, pp. 1448–1456.

[126] D. Russo, “Simple bayesian algorithms for best arm identification,” in *Conference on Learning Theory*, PMLR, 2016, pp. 1417–1418.

[127] H. Van Seijen, H. Van Hasselt, S. Whiteson, and M. Wiering, “A theoretical and empirical analysis of expected sarsa,” in *Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, IEEE, 2009, pp. 177–184.

[128] L. Matignon, G. Laurent, and N. Le Fort-Piat, "Hysteretic q-learning: An algorithm for decentralized reinforcement learning in cooperative multi-agent teams," in *International Conference on Intelligent Robots and Systems*, IEEE, 2007, pp. 64–69.

[129] D. Precup, "Eligibility traces for off-policy policy evaluation," *Computer Science Department Faculty Publication Series*, p. 80, 2000.

[130] R. Howard, *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[131] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010.

[132] A. Harutyunyan, M. Bellemare, T. Stepleton, and R. Munos, "Q(λ) with off-policy corrections," in *International Conference on Algorithmic Learning Theory*, Springer, 2016, pp. 305–320.

[133] A. Schwartz, "A reinforcement learning method for maximizing undiscounted rewards," in *International Conference on Machine Learning*, vol. 298, 1993, pp. 298–305.

[134] G. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994, vol. 37.

[135] R. Munos, T. Stepleton, A. Harutyunyan, and M. Bellemare, "Safe and efficient off-policy reinforcement learning," in *Advances in Neural Information Processing Systems*, 2016, pp. 1054–1062.

[136] R. Bellman, *Dynamic programming*. Princeton University Press, 1957.

[137] C. Zhang and V. Lesser, "Multi-agent learning with policy prediction.," in *AAAI Conference on Artificial Intelligence*, 2010.

[138] N. Roy and S. Thrun, "Coastal navigation with mobile robots," in *Advances in Neural Information Processing Systems*, 2000, pp. 1043–1049.

[139] S. Thrun, "Probabilistic robotics," *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

[140] M. Hauskrecht, "Incremental methods for computing bounds in partially observable Markov decision processes," in *AAAI Conference on Artificial Intelligence*, Citeseer, 1997, pp. 734–739.

[141] M. Hauskrecht, "Value-function approximations for partially observable Markov decision processes," *Journal of artificial intelligence research*, vol. 13, pp. 33–94, 2000.

[142] A. Cassandra, M. Littman, and N. Zhang, “Incremental pruning: A simple, fast, exact method for partially observable Markov decision processes,” in *Conference on Uncertainty in Artificial Intelligence*, Morgan Kaufmann Publishers Inc., 1997, pp. 54–61.

[143] H. Cheng, “Algorithms for partially observable Markov decision processes,” Ph.D. dissertation, University of British Columbia, 1988.

[144] M. Spaan and N. Vlassis, “Perseus: Randomized point-based value iteration for POMDPs,” *Journal of artificial intelligence research*, vol. 24, pp. 195–220, 2005.

[145] D. Silver and J. Veness, “Monte-Carlo planning in large POMDPs,” in *Advances in neural information processing systems*, 2010, pp. 2164–2172.

[146] J. Pineau, G. Gordon, S. Thrun, *et al.*, “Point-based value iteration: An anytime algorithm for POMDPs,” in *International Joint Conference on Artificial Intelligence*, vol. 3, 2003, pp. 1025–1032.

[147] M. Littman, A. Cassandra, and L. Kaelbling, “Learning policies for partially observable environments: Scaling up,” in *Machine Learning Proceedings 1995*, Elsevier, 1995, pp. 362–370.

[148] S. Paquet, L. Tobin, and B. Chaib-draa, “Real-time decision making for large POMDPs,” in *Conference of the Canadian Society for Computational Studies of Intelligence*, Springer, 2005, pp. 450–455.

[149] H. Kurniawati, D. Hsu, and W. Lee, “SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces,” in *In Proc. Robotics: Science and Systems*, 2008.

[150] L. Kaelbling, M. Littman, and A. Cassandra, “Planning and acting in partially observable stochastic domains,” *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[151] E. Bargiacchi, “Dynamic resource allocation for multi-camera systems,” M.S. thesis, University of Amsterdam, 2016.

[152] A. Rosenthal, “Nonserial dynamic programming is optimal,” in *ACM Symposium on Theory of Computing*, ACM, 1977, pp. 98–105.

[153] R. Dechter, “Bucket elimination: A unifying framework for probabilistic inference,” in *Learning in graphical models*, Springer, 1998, pp. 75–104.

[154] D. Roijers, S. Whiteson, and F. Oliehoek, “Computing convex coverage sets for multi-objective coordination graphs,” in *International Conference on Algorithmic Decision Theory*, Nov. 2013, pp. 309–323.

[155] Y. Gai, B. Krishnamachari, and R. Jain, "Combinatorial network optimization with unknown variables: Multi-armed bandits with linear rewards and individual observations," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 5, pp. 1466–1478, 2012.

[156] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in cooperative multiagent systems," *AAAI/IAAI*, vol. 1998, pp. 746–752, 1998.

[157] E. Bargiacchi, C. Verschoor, G. Li, and D. Roijers, "Decentralized solutions and tactics for RTS," *BNAIC*, vol. 25, pp. 372–373, 2013.

[158] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Elsevier, 2014.

[159] C. Guestrin, D. Koller, and R. Parr, "Max-norm projections for factored MDPs," in *International Joint Conference on Artificial Intelligence*, vol. 1, 2001, pp. 673–682.