Cooperative Prioritized Sweeping

Eugenio Bargiacchi
Department of Computer Science
Vrije Universiteit Brussel
svalorzen@gmail.com

Timothy Verstraeten
Department of Computer Science
Vrije Universiteit Brussel
timothy.verstraeten@vub.ac.be

Diederik M. Roijers
Vrije Universiteit Brussel
HU University of Applied Sciences
Utrecht
diederik.yamamoto-roijers@hu.nl

ABSTRACT

We present a novel model-based algorithm, Cooperative Prioritized Sweeping, for sample-efficient learning in large multi-agent Markov decision processes. Our approach leverages domain knowledge about the structure of the problem in the form of a dynamic decision network. Using this information, our method learns a model of the environment to determine which state-action pairs are the most likely in need to be updated, significantly increasing learning speed. Batch updates can then be performed which efficiently back-propagate knowledge throughout the value function. Our method outperforms the state-of-the-art sparse cooperative Q-learning and QMIX algorithms, both on the well-known SysAdmin benchmark, randomized environments and a fully-observable variation of the well-known firefighter benchmark from Dec-POMDP literature.

KEYWORDS

model-based; reinforcement learning; multi-agent; MDP; cooperation; Bayesian network

ACM Reference Format:
Eugenio Bargiacchi, Timothy Verstraeten, and Diederik M. Roijers. 2021. Cooperative Prioritized Sweeping. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 9 pages.

1 INTRODUCTION

Consider a control problem where multiple agents must learn to cooperate to achieve a common goal in an environment with unknown and complex dynamics, such as robot soccer [14], warehouse commissioning [7], and traffic light control [39]. Such problems cannot be solved optimally in general even if the dynamics of the environment are known in advance, due to the size of both state and action spaces being exponential in the number of agents [27]. Large environments also require increasingly large amounts of data to learn effectively, which are often impractical to obtain in real-life scenarios.

An effective way to improve sample-efficiency is to perform batch updates on the value function in between interactions with the environment. The data required to perform these updates can be sampled from an existing pool, with experience replay being a well-known example of this technique [18]. Alternatively, the data can be generated on the fly from a learned model, as it is done in the Dyna-Q algorithm [33]. 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. This is one of the reasons of the recent success of prioritized experience replay [28]. 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 this idea. PS improves sample-efficiency by sorting the state-action pairs to update using a *priority queue*, where each pair's priority is proportional to the likelihood that their update will significantly affect the value function [22]. The key idea 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.

The main drawback is that PS does not readily extend to large environments, and in particular to the multi-agent domain [1]. As the priority queue increases in size, the computational cost of bookkeeping rapidly becomes unsustainable, making the approach impractical. In large environments PS can also update states that will not be experienced again, reducing efficiency.

In this paper we propose *cooperative prioritized sweeping* (CPS), a novel algorithm that generalizes the ideas of PS to large scale, discrete multi-agent environments. In order to keep the problem tractable, we exploit domain knowledge in the form of *coordination graphs*. Coordination graphs describe the conditional dependencies between agents and state features. Such graphs are often used in multi-agent settings, and can be assessed by domain experts or learned from data [1, 3, 9, 15, 17, 23, 30, 36, 37]. Exploiting the structure of a coordination graph is central to our approach, as it allows CPS to reason locally about the dynamics of the environment. Using this information, CPS is able to compute priorities for subsets of joint state-action pairs, avoiding the curse of dimensionality. We show 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 empirically show that the policies learned with CPS perform close to the theoretical optimum, even though convergence is not guaranteed.

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.

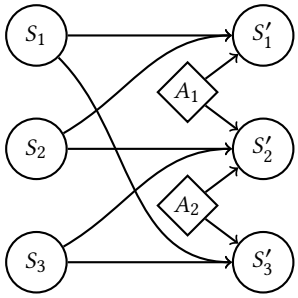


Figure 1: A simple DDN with 2 agents and 3 state features. The incoming arrows represent the conditional dependencies in the transition function, between the state features in the next timestep and the current state features and agent actions.

The rest of this paper is organized as follows. Section 2 gives a formal definition of the setting and our main assumptions. Section 3 describes our main contributions: the mechanism used to efficiently prioritize updates within our model-based setting, and the full CPS algorithm for sample-efficient RL. Section 4 describes our experimental setup and our benchmarks, the empirical results and associated discussion. Section 5 contains several references pointing to similar work. Finally, Section 6 summarizes the results and presents future work.

2 PROBLEM FORMULATION

A multi-agent Markov decision process (MMDP) is described as a tuple $\langle \mathbf{S}, \mathbf{A}, T, R, \gamma \rangle$, where:

- $\mathbf{S} = S_1 \times \cdots \times S_N$ is the joint state space, which is the Cartesian product of N state features.
- $\mathbf{A} = A_1 \times \cdots \times A_K$ is the joint action space, which is the Cartesian product of the action spaces of K agents.
- $T(\mathbf{s}' | \, \mathbf{s}, \mathbf{a})$ is the transition function which describes the environment's dynamics.
- R is the reward function associating rewards to joint state-action pairs, $\langle \mathbf{s}, \mathbf{a} \rangle$.
- $\gamma \in [0, 1)$ is the discount factor representing the importance of future rewards.

In an MMDP, the interactions between the state factors and agents are assumed to be *sparse*, such that they can be represented using coordination graphs. In particular, we describe these dynamics using *Dynamic Decision Networks* (DDN) [9, 23], i.e. Bayesian networks where the action nodes are not random variables. A DDN can be graphically represented in a compact form as a directed graph where each edge marks the direct influence of one variable over another. An example DDN is shown in Figure 1. We assume that the DDN structure is provided to the agents prior to any learning.

Using the DDN formulation, we can describe the transition function of an MMDP in a factored manner:

$$T(\mathbf{s}' | \, \mathbf{s}, \mathbf{a}) = \prod_i T_i(s'_i \mid \mathbf{s}^{\textcircled{1}}, \mathbf{a}^{\textcircled{1}}) \tag{1}$$

where we use the symbol $^{\textcircled{1}}$ to denote the *parent nodes* of S'_i in the DDN graph. For example, in Figure 1, $S^{\textcircled{2}} = \{S_2, S_3\}$, and $A^{\textcircled{2}} = \{A_1, A_2\}$.

We assume that the reward function has the same structure as T, such that $R(\mathbf{s}, \mathbf{a}) = \sum_i R_i(\mathbf{s}^{\textcircled{1}}, \mathbf{a}^{\textcircled{1}})$ where each R_i has the same domain $(\mathbf{S}^{\textcircled{1}}, \mathbf{A}^{\textcircled{1}})$ as T_i. Given this structure, we assume that rewards are sampled, both from the environment and from the model, as vectors with N elements.

This representation is somewhat different from what some other multi-agent RL algorithms use (e.g., [15]), 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 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 state factor per agent to convey the rewards.

3 COOPERATIVE PRIORITIZED SWEEPING

Here we describe *cooperative prioritized sweeping* (CPS), a novel model-based approach for MMDPs that generalizes the ideas of PS to the multi-agent setting.

3.1 Motivation

The main idea behind prioritized sweeping is that 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:

$$V^\star(s) = R(s, a^\star) + \gamma \sum_{s'} T(s' | \, s, a^\star) V^\star(s') \tag{2}$$

During learning, after each interaction with the environment we obtain new experience that can be used to update the value function. Equation 2 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 $T(s' | \, s, a)$.

Ideally, one would recursively update the value function until it satisfied Equation 2; 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.

3.2 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 $\psi(s, a)$:

$$\psi(s, a) = |\Delta_{s'}| \, T(s' | \, s, a) \tag{3}$$

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

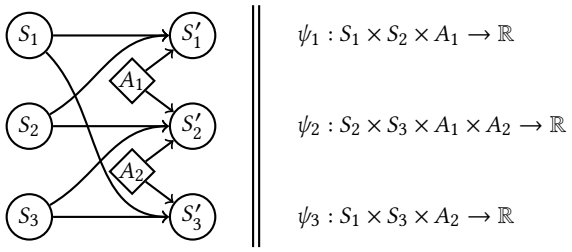


Figure 2: The structure of the DDN allows to approximate the priority function ψ with a set of ψ_i, one for each S'_i node. The domain of each ψ_i corresponds to the parents of the corresponding S'_i node.

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:

$$\psi(\mathbf{s}, \mathbf{a}) = |\Delta_{\mathbf{s}'}|\, T(\mathbf{s}' | \mathbf{s}, \mathbf{a}) \tag{4}$$

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 parent set in the DDN, such that:

$$\begin{aligned}
\psi(\mathbf{s}, \mathbf{a}) &= |\Delta_{\mathbf{s}'}|\, T(\mathbf{s}' | \mathbf{s}, \mathbf{a}) && (5)\\
&= \sum_i |\Delta_i| \prod_i T_i(s'_i \mid \mathbf{s}^{①}, \mathbf{a}^{①}) && (6)\\
&\approx \sum_i |\Delta_i|\, T_i(s'_i \mid \mathbf{s}^{①}, \mathbf{a}^{①}) && (7)\\
&= \sum_i \psi_i(\mathbf{s}^{①}, \mathbf{a}^{①}) && (8)
\end{aligned}$$

where $\Delta_i = \Delta_{s'_i}$ represents the TD error attributed to the i-th component of the joint state. We leverage the underlying assumption of sparse interactions of an MMDP so that each ψ_i can be represented explicitly, as its domain is relatively small. Note that this factorization is similar to the one we perform on the Q-function in Section 3.3.

A simple example of the factorization of ψ can be seen in Figure 2. For each S'_i node, we construct a ψ_i function with domain equal to the parent nodes of S'_i, i.e. the nodes in $\langle \mathbf{S}, \mathbf{A} \rangle$ that directly point to S'_i. Note how each ψ_i 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.

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

Figure 3: An example of the maximization of the full priority function from Figure 2. At each update, depending on the current priorities stored in each ψ_i, 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.

3.3 Learning Priorities

In Equation 7, we can learn each T_i by maintaining estimates of the proportion that a state s'_i is reached after performing a local joint action $\mathbf{a}^{①}$ in a local joint state $\mathbf{s}^{①}$ [1]:

$$T_i(s'_i \mid \mathbf{s}^{①}, \mathbf{a}^{①}) = \frac{N_{\mathbf{s}^{①},\mathbf{a}^{①},s'_i} + N^0_{\mathbf{s}^{①},\mathbf{a}^{①},s'_i}}{\sum_{s''_i \in S_i} N_{\mathbf{s}^{①},\mathbf{a}^{①},s''_i} + N^0_{\mathbf{s}^{①},\mathbf{a}^{①},s''_i}} \tag{9}$$

where the $N_{\mathbf{s}^{①},\mathbf{a}^{①},s'_i}$ are the counts for the observed transitions, and the $N^0_{\mathbf{s}^{①},\mathbf{a}^{①},s''_i}$ 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.

The local TD errors in Equation 7 need to be computed after each update to the value function to provide up-to-date estimates of the priorities. Unfortunately, as each Δ_i refers to a specific state feature, it can be challenging to assign TD errors to each component exactly. However, we can obtain fairly good approximations by leveraging *factored value functions* [4, 9, 10, 15, 25]. Factored value functions have been empirically shown to perform well in multi-agent environments, as they significantly simplify the representation of large-dimensional value functions without significant losses. We approximate the Q-function as the sum of lower-dimensional components:

$$Q(\mathbf{s}, \mathbf{a}) = \sum_x Q_x(\mathbf{s}^{⊗}, \mathbf{a}^{⊗}) \tag{10}$$

where each $x \subset \{i \mid i \in \mathbf{S}\}$ is a subset of indices of the state variables, selected in advance by the user. We call these *basis domains*. Note that basis domains can overlap, and their selection affects the ability of the Q-function to represent correlations between state variables.

Given the factorization in Equation 10, 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_i by simple redistribution:

$$\Delta_i = \sum_x I(S_i \in \mathbf{S}^{⊗}) \frac{\Delta_x}{|\mathbf{S}^{⊗}|} \tag{11}$$

Using Equations 9 and 11 we can finally compute priorities for each parent set, i.e. $\psi_i(\mathbf{s}^{\oplus}, \mathbf{a}^{\oplus})$. 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 3 shows an example of this selection process across a set of already computed priorities. We can see that each ψ_i is not maximized independently, as we need to select a single joint state-action pair that maximizes their sum ψ. Unfortunately, performing this maximization exactly is equivalent to the maximum-weight independent set problem (MWIS), which is NP-hard in the general case [6, 13].

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. Therefore, we can settle for a lower priority pair and use heuristics to avoid the NP-hard selection. We (randomly) traverse all ψ_i, 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.

3.4 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 main contribution: the CPS algorithm.

We maintain a discrete factored Q-function as shown in Equation 10, 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 [12, 35, 38], 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 [9].

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 work proposed by Sparse Cooperative Q-learning, a Q-learning variant for multi-agent environments [15].

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

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma Q(s', a'^{\star}) - Q(s,a) \right) \tag{12}$$

where α is the learning rate parameter.

Algorithm 1: Cooperative Prioritized Sweeping

while True **do**
  $\mathbf{a} \leftarrow$ Select action for state $\mathbf{s}$ following policy π
  $\langle \mathbf{s}', \mathbf{r} \rangle \leftarrow$ Execute action $\mathbf{a}$ in state $\mathbf{s}$
  Update T, R using $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r} \rangle$ (see Eq. 9)
  $\mathbf{a}'^{\star} \leftarrow \arg\max_{\mathbf{a}'^{\star}} Q(\mathbf{s}', \mathbf{a}'^{\star})$ using VE
  $\Delta_x \leftarrow$ Compute using $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{a}'^{\star} \rangle$ (see Eq. 14)
  $Q(\mathbf{s}, \mathbf{a}) \leftarrow$ Update using Δ_x (see Eq. 15)
  for each state feature i **do**
    $\Delta_i \leftarrow$ Compute from Δ_x (see Eq. 11)
    for each pair $\langle \dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus} \rangle$ **do**
      $\psi_i(\dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus}) \leftarrow \psi_i(\dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus}) + |\Delta_i| \, T(s'_i = s_i \mid \dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus})$ (see Eq. 7)
    end for
  end for
  for each batch update **do**
    $\langle \dot{\mathbf{s}}, \dot{\mathbf{a}} \rangle \leftarrow \arg\max_{\dot{\mathbf{s}}, \dot{\mathbf{a}}} \psi(\dot{\mathbf{s}}, \dot{\mathbf{a}})$ (approximate)
    for each pair $\langle \dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus} \rangle$ **do**
      $\psi_i(\dot{\mathbf{s}}^{\oplus}, \dot{\mathbf{a}}^{\oplus}) \leftarrow 0$
    end for
    $\langle \dot{\mathbf{s}}', \dot{\mathbf{r}} \rangle \leftarrow$ Sample from T, R using $\dot{\mathbf{s}}^{\oplus}$ and $\dot{\mathbf{a}}^{\oplus}$
    Perform steps 5 to 13 using $\langle \dot{\mathbf{s}}, \dot{\mathbf{a}}, \dot{\mathbf{s}}', \dot{\mathbf{r}} \rangle$
  end for
end while

In our setting, we need to adjust Equation 12 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 12 per state component, for each computing a target that takes into account only its associated Q factors:

$$\delta_i = r_i + \sum_x I(S_i \in \mathbf{S}^{\otimes}) \frac{\gamma Q_x(\mathbf{s}'^{\otimes}, \mathbf{a}'^{\star\otimes}) - Q_x(\mathbf{s}^{\otimes}, \mathbf{a}^{\otimes})}{|\mathbf{S}^{\otimes}|} \tag{13}$$

where I is the indicator function, which we use to select only the Q factors with domain over S_i. Computing $\mathbf{a}'^{\star}$ requires maximizing the whole factored Q-function, which can be done efficiently using variable elimination (VE) [9].

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

$$\Delta_x = \alpha \left(\sum_i I(S_i \in \mathbf{S}^{\otimes}) \frac{\delta_i}{|\{x : i \in x\}|} \right) \tag{14}$$

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

$$Q_x(\mathbf{s}^{\otimes}, \mathbf{a}^{\otimes}) = Q_x(\mathbf{s}^{\otimes}, \mathbf{a}^{\otimes}) + \Delta_x \tag{15}$$

CPS then recomputes the priorities using the new TD errors, as shown in Section 3.3.

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 T and R functions. The priorities of the selected parents are set to zero, to mark that their values have been successfuly updated. Note that R can be learned using maximum-likelihood estimates similar to how T is learned

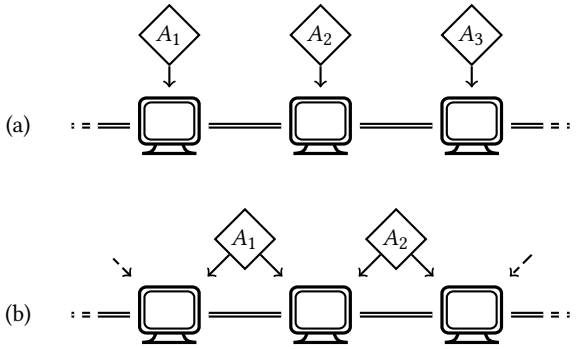


Figure 4: In the SysAdmin ring setting, each machine is directly connected to its two adjacent neighbors to form a ring, while each agent directly controls the actions of each machine (a). In the modified shared control setting, the connections between the machines are left unchanged. However, each agent 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).

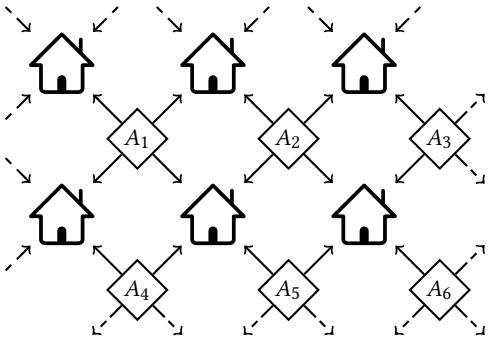


Figure 5: A representation of the firefighter setting. Houses are set in a toroidal grid. Every timestep, each agent goes to one of its 4 adjacent houses. The risk of fire for each house is determined by the number of firefighters at that house and by the average fire level in neighboring houses.

in Equation 9. A pseudocode description of the full algorithm is shown in Algorithm 1.

4 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 [9], Sparse Cooperative Q-learning (SCQL) with and without randomized experience replay [15], and QMIX [25] as competing algorithms. The algorithms were implemented using the AI-Toolbox [2] and PYMARL [26] frameworks.

The factored LP planning algorithm is trained in advance, and we show the performance of the final optimal policy. As its name implies, this algorithm solves a factored linear program that is designed to project the true optimal value function to the closest representable factored Q-function. 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 after hyperparameter optimization. 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 9 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.

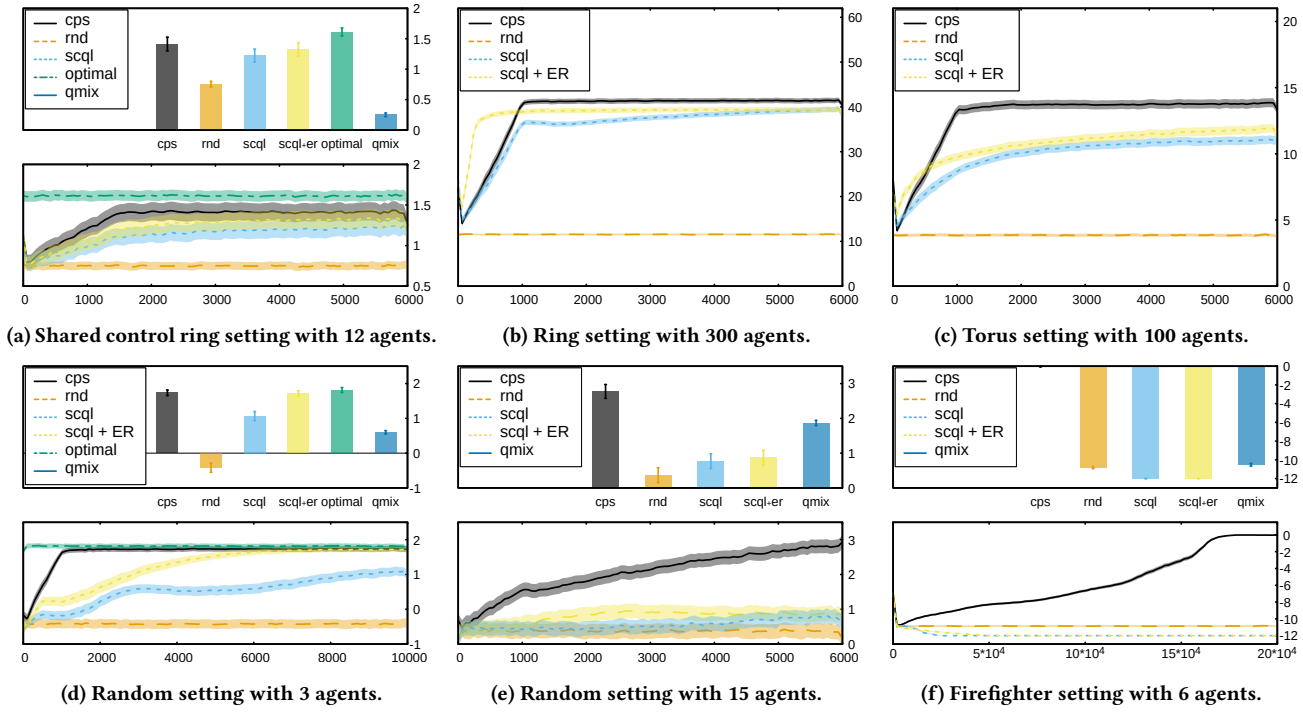
The training results for SCQL and CPS are shown in the line plots of Figure 6, and report the immediate rewards obtained at each timestep during a single, uninterrupted episode. The training results for QMIX are shown in Figure 7 and report average immediate reward for each training episode. The training 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 6 report the performance of the trained policies for all algorithms. All results are averaged over 100 runs.

4.1 SysAdmin

The SysAdmin setting simulates a set of interconnected machines that need to complete abstract jobs [9]. 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. We select each machine's state



(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 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.

and load pair as the basis domains for the Q-function factorization, which is equivalent to the single basis proposed in [9].

4.2 Random

We randomly generate MMDPs by connecting each S'_i with S_i, 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. $|S_i| = |A_k| = 3$, 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'_i, selected with probability 0.3. We select $x = \{i, i+1\}$ as basis domains for the Q-function.

4.3 Firefighters

We modify the firefighter setting from Dec-POMDP literature [23, 24] 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 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.

4.4 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.

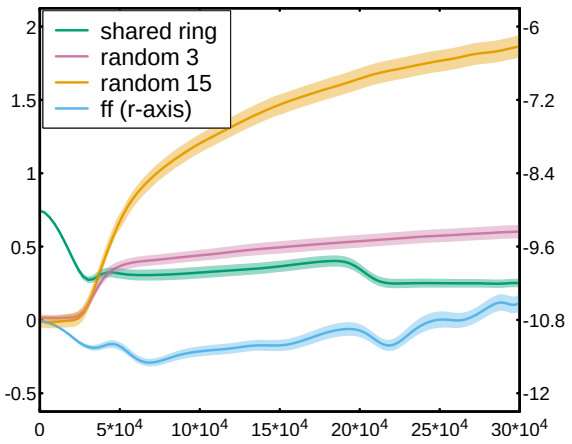


Figure 7: Best QMIX performance during training in all its environments, across hyperparameters. Both y-axes report mean and standard error for average per-timestep reward. A training episode corresponds to 1000 timesteps on the x-axis. Note the scale of the x-axis when comparing against Figure 6. Data is averaged over 100 runs. Higher is better.

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 6a), SCQL, SCQL-ER and CPS achieve 76%, 82% and 88% of the upper bound, respectively, while in the smaller random setting (Figure 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 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 threshold 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 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 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.

5 RELATED WORK

It is well-known that prioritizing updates using TD errors can have a significant impact on performance. Eligibility traces [32, 34, 38] 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 [18]. While experience replay is mostly known for randomly replaying experiences [21], 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 [28] showed this to be true, improving sample-efficiency by scoring experiences using already computed TD errors.

Model-based algorithms that leverage TD errors have been explored less, due to their increased complexity. Dyna-Q [33] and Dyna2 [29] use learned models to perform random backups on the value function that are used to speed up learning. Prioritized sweeping [1, 22] 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 [8].

The use of low rank and factored value functions approximations has seen particular success in the past in discrete settings [9, 10, 15], and has more recently received increased attention due to the reduced computational costs and good empirical performance in deep multi-agent reinforcement learning [5, 11]. VDN [31] can learn a simple factorization of the Q-function, with a single component per agent. QMIX [25] improves upon this by adding a mixing network that allows to combine each factor in non-linear ways. More recently, DCG [4] has shown promising results by learning payoff functions for each pair of agents, leveraging parameter sharing. While CPS and DCG share similarities in their representation of the value function, our focus with CPS is to maximize sample-efficiency by improving the exploitation of experience data using domain knowledge. At the same time, we believe that the ideas in

CPS could enhance learning in deep methods, which is a topic to explore in future research.

In this paper, we make use of variable elimination [9] to select the optimal joint action given w.r.t. an approximate value function. Such joint action selection can also be done approximately, e.g., via max-plus [16], AND/OR tree search [20], or variational methods [19]. We note that this is orthogonal to our contributions and can be added easily if the complexity of the chosen graphical structure of the value function requires it.

6 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 [28] but in a more general way.

In future work we aim to extend the techniques presented here to multi-agent continuous state-spaces, to allow for easier comparisons between CPS and deep learning methods. Additionally, we believe that the input coordination graph does not need to correspond exactly to the true structure of a problem for CPS to be effective, as long as the data used for the updates is generated from a ground truth source (for example, from a simulator). This would allow the application of the model-based CPS even in real-world complex environments that cannot be modeled exactly. We plan to perform experiments in this direction to further validate this approach.

ACKNOWLEDGMENTS

The authors would like to acknowledge FWO (Fonds Wetenschappelijk Onderzoek) for their support through the SB grants of Eugenio Bargiacchi (#1SA2820N) and Timothy Verstraeten (#1S47617N). This research was additionally supported by funding from the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme.

We also would like to welcome Shion G. Roijers to the world.

REFERENCES

[1] David Andre, Nir Friedman, and Ronald Parr. 1998. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*. 1001–1007.

[2] Eugenio Bargiacchi, Diederik M. Roijers, and Ann Nowé. 2020. AI-Toolbox: A C++ library for Reinforcement Learning and Planning (with Python Bindings). *Journal of Machine Learning Research* 21, 102 (2020), 1–12. http://jmlr.org/papers/v21/18-402.html

[3] Eugenio Bargiacchi, Timothy Verstraeten, Diederik M. Roijers, Ann Nowé, and Hado van Hasselt. 2018. Learning to Coordinate with Coordination Graphs in Repeated Single-Stage Multi-Agent Decision Problems. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. 491–499.

[4] Wendelin Böhmer, Vitaly Kurin, and Shimon Whiteson. 2020. Deep Coordination Graphs. In *International Conference on Machine Learning*. https://arxiv.org/abs/1910.00091

[5] Jacopo Castellini, Frans A. Oliehoek, Rahul Savani, and Shimon Whiteson. 2019. The Representational Capacity of Action-Value Networks for Multi-Agent Reinforcement Learning. In *AAMAS'19*. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 1862–1864.

[6] Maria Chudnovsky, Marcin Pilipczuk, Michał Pilipczuk, and Stéphan Thomassé. 2020. On the Maximum Weight Independent Set problem in graphs without induced cycles of length at least five. *SIAM Journal on Discrete Mathematics* 34, 2 (2020), 1472–1483.

[7] Daniel Claes, Frans A. Oliehoek, Hendrik Baier, and Karl Tuyls. 2017. 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, 492–500.

[8] Rémi Coulom. 2007. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*. Vol. 4630. Springer Berlin Heidelberg, Berlin, Heidelberg, 72–83.

[9] Carlos E. Guestrin, Daphne Koller, and Ronald Parr. 2002. Multiagent Planning with Factored MDPs. In *NIPS 2002: Advances in Neural Information Processing Systems 15*. 1523–1530.

[10] Carlos E. Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. 2003. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research* 19 (2003), 399–468.

[11] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. 2018. Is multiagent deep reinforcement learning the answer or the question? A brief survey. *CoRR* abs/1810.05587 (2018).

[12] Jing Peng and Ronald J. Williams. [n.d.]. Efficient Learning and Planning Within the Dyna Framework. 1, 4 ([n. d.]), 437–454.

[13] Richard M. Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

[14] Jelle R. Kok, Matthijs T. J. Spaan, and Nikos Vlassis. 2003. Multi-robot decision making using coordination graphs. In *Proceedings of the 11th International Conference on Advanced Robotics, ICAR*, Vol. 3. 1124–1129.

[15] Jelle R. Kok and Nikos Vlassis. 2004. Sparse cooperative Q-learning. In *ICML 2004: Proceedings of the twenty-first international conference on Machine learning*. 61–68.

[16] Jelle R. Kok and Nikos Vlassis. 2005. Using the max-plus algorithm for multiagent decision making in coordination graphs. In *Robot Soccer World Cup*. Springer, 1–12.

[17] Mark Kroon and Shimon Whiteson. 2009. Automatic feature selection for model-based reinforcement learning in factored MDPs. In *2009 International Conference on Machine Learning and Applications*. IEEE, 324–330.

[18] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8, 3-4 (1992), 293–321.

[19] Qiang Liu and Alexander T. Ihler. 2012. Belief propagation for structured decision making. *arXiv preprint arXiv:1210.4897* (2012).

[20] Radu Marinescu and Rina Dechter. 2005. AND/OR branch-and-bound for graphical models. In *IJCAI*. 224–229.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari With Deep Reinforcement Learning. In *NIPS Deep Learning Workshop*.

[22] Andrew W. Moore and Christopher G. Atkeson. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning* 13, 1 (1993), 103–130.

[23] Frans A. Oliehoek and Christopher Amato. 2016. *A concise introduction to decentralized POMDPs*. Vol. 1. Springer.

[24] Frans A. Oliehoek, Matthijs T. J. Spaan, and Nikos Vlassis. 2008. Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research* 32 (2008), 289–353.

[25] Tabish Rashid, Mikayel Samvelyan, Christian S. De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. In *International Conference on Machine Learning*. 4292–4301.

[26] Mikayel Samvelyan, Tabish Rashid, Christian S. De Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. 2019. The StarCraft Multi-Agent Challenge. *CoRR* abs/1902.04043 (2019).

[27] Joris Scharpff, Diederik M. Roijers, Frans A. Oliehoek, Matthijs T. J. Spaan, and Mathijs M. De Weerdt. 2016. Solving Transition-Independent Multi-Agent MDPs with Sparse Interactions.. In *AAAI*. 3174–3180.

[28] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[29] David Silver, Richard S Sutton, and Martin Müller. 2008. Sample-based learning and search with permanent and transient memories. In *Proceedings of the 25th international conference on Machine learning*. 968–975.
[30] Alexander L. Strehl, Carlos Diuk, and Michael L. Littman. 2007. Efficient structure learning in factored-state MDPs. In *AAAI*, Vol. 7. 645–650.
[31] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z Leibo, Karl Tuyls, et al. 2018. Value-Decomposition Networks For Cooperative Multi-Agent Learning Based On Team Reward.. In *AAMAS*. 2085–2087.
[32] Richard S. Sutton. 1988. Learning to predict by the methods of temporal differences. *Machine learning* 3, 1 (1988), 9–44.
[33] Richard S. Sutton. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*. Elsevier, 216–224.
[34] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[35] Harm van Seijen and Richard S. Sutton. 2013. Efficient Planning in MDPs by Small Backups. In *Proc. 30th Int. Conf. Mach. Learn.*, Vol. 28. III–361.
[36] Timothy Verstraeten, Eugenio Bargiacchi, Pieter J. K. Libin, Jan Helsen, Diederik M. Roijers, and Ann Nowé. 2020. Multi-Agent Thompson Sampling for Bandit Applications with Sparse Neighbourhood Structures. *Nature Scientific Reports* 10, 1 (2020), 6728.
[37] Christopher M. Vigorito and Andrew G. Barto. 2009. Incremental structure learning in factored MDPs with continuous states and actions. *University of Massachusetts Amherst-Department of Computer Science, Tech. Rep* (2009).
[38] Christopher J. C. H. Watkins. 1989. *Learning from delayed rewards*. Ph.D. Dissertation. King's College, Cambridge.
[39] Marco A. Wiering. 2000. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*. 1151–1158.