Model-based Multi-Agent Reinforcement Learning with Cooperative Prioritized Sweeping

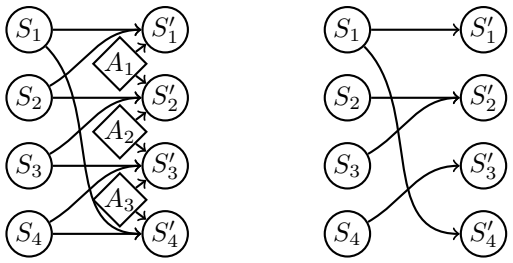
Eugenio Bargiacchi[1], **Timothy Verstraeten**[1], **Diederik M. Roijers**[2], **Ann Nowé**[1]
[1]Vrije Universiteit Brussel
svalorzen@gmail.com

Abstract

We present a new model-based reinforcement learning algorithm, *Cooperative Prioritized Sweeping*, for efficient learning in multi-agent Markov decision processes. The algorithm allows for sample-efficient learning on large problems by exploiting a factorization to approximate the value function. Our approach only requires 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 and performs temporal difference updates which affect multiple joint states and actions at once. Batch updates are additionally performed which efficiently back-propagate knowledge throughout the factored Q-function. Our method outperforms the state-of-the-art algorithm sparse cooperative Q-learning algorithm, both on the well-known SysAdmin benchmark and randomized environments.

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 (Kok et al. 2003), warehouse commissioning (Claes et al. 2017), and traffic light control (Wiering 2000). 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. Large environments also require increasingly large amounts of data to learn effectively, which are often impractical to obtain in real-life scenarios.

In this paper, we introduce an approximate method for fast, sample efficient reinforcement learning (RL) with a large set of cooperative agents, called *cooperative prioritized sweeping* (CPS). Specifically, we generalize *prioritized sweeping* (PS) (Moore and Atkeson 1993) to linearly factored Q-functions (Guestrin, Koller, and Parr 2001; 2002). We use model-based RL, which consists in learning a model of the environment, and using it to learn a *value function* and extract a policy for the agents. We represent this model using a *dynamic decision network* and exploit the state-action dependency graph to efficiently construct and manage the priority queue. CPS performs updates similar to Q-Learning, but is able to back-propagate these changes through the Q-function by using the factored model. All updates are stored in a priority queue, such that the largest changes are back-propagated first in an anytime fashion.



(a) A simple DDN in compact form.

(b) A DBN induced by a particular joint action.

Figure 1: How to represent a multi-agent factored MDP.

1 Problem Formulation

A multi-agent Markov decision process (MMDP) is a tuple $\langle \mathbf{S}, \mathbf{A}, T, R, \gamma \rangle$, where $\mathbf{S}$ and $\mathbf{A}$ are the state and action spaces, T is the transition function which describes the environment's dynamics, R is the reward function associating rewards with joint state-action pairs and $\gamma \in [0, 1)$ is the discount factor representing the importance of future rewards. The joint action space is the Cartesian product of the action spaces of each agent, i.e., $\mathbf{A} = A_1 \times \cdots \times A_K$. The state space is the Cartesian product of state factors, i.e., $\mathbf{S} = S_1 \times \cdots \times S_N$.

In an MMDP, the interactions between the state factors and agents are assumed to be *sparse*. Therefore, we use a *Dynamic Decision Network* (DDN) to describe them (Guestrin, Koller, and Parr 2002). 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. Note that, while compact, such a representation can lose information, as in a DDN actions induce conditional dependencies which may be subsets of the edges represented. More specifically, an edge in the graph marks that a relationship exists for at least one action. For example, in Figure 1, the state variable S'_1 may depend solely on S_1 when $A_1 = 0$, i.e., $P(S'_1 \mid S_1, S_2, A_1 = 0) = P(S'_1 \mid S_1, A_1 = 0)$, but may depend on both S_1 and S_2 when $A_1 = 1$. This information is part of the DDN itself— but usually not represented graphically.

Using the DDN formulation, we can formulate the transition function of an MMDP as

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

where $\overline{\mathbf{a}}$ is the part of $\mathbf{a}$ on which s'_i depends, and $\overline{\mathbf{s}}$ is the part of $\mathbf{s}$ on which s'_i depends given action $\overline{\mathbf{a}}$. Therefore each joint action induces a *Dynamic Bayesian Network* (DBN) out of the DDN (Figure 1b).

We assume the reward function has the same structure as T, such that $R(\mathbf{s}, \mathbf{a}) = \sum_i R_i(\overline{\mathbf{s}}, \overline{\mathbf{a}})$ where each R_i has the same domain $(\overline{\mathbf{S}}, \overline{\mathbf{A}})$ 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 is useful when updating the factored Q-function (see Section 2.2).

This representation is somewhat different from what some other multi-agent RL algorithms use (e.g., (Kok and Vlassis 2004)), 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.

2 Cooperative Prioritized Sweeping

To be able to learn in a factored MMDP, it is key to exploit the factored structure to learn an (approximate) sparse Q-function (Section 2.2). For example, Sparse Cooperative Q-Learning (SCQL) (Kok and Vlassis 2004) learns a factorised Q-function on a per-agent or a per-edge (2-agent factor) basis. However, the model-free approach with one-step look-ahead bootstrapping that SCQL uses is not very sample-efficient. A straightforward idea to increase sample-efficiency would be to learn an approximate MMDP model (Section 2.1), and perform planning on this model to obtain an approximate sparse Q-function (Guestrin et al. 2003; Guestrin, Koller, and Parr 2002). The state-of-the-art planning algorithm employs an LP that computes the approximate value function that minimizes the max-norm distance to the optimal one (Guestrin, Koller, and Parr 2002). However, because this requires solving a large scale LP at every time step as we learn the model, the computational requirements make such a model-based approach infeasible.

In this section, we propose a new model-based approach for MMDPs that generalizes the idea of prioritized sweeping (Moore and Atkeson 1993) to the multi-agent setting, by both using direct experience and simulated experience from an MMDP model (Section 2.3–2.5) to learn the sparse Q-function. Our algorithm—cooperative prioritized sweeping (Algorithm 1)—thus benefits from the sample-efficiency of a model-based approach, while keeping the computation time in check.

2.1 Learning the Model

Cooperative prioritized sweeping (CPS) approximates a model of the environment to determine how changes must be back-propagated through the value function after an update. Furthermore, this model is also used to generate new simulated experience that is sampled in batches and used to update the value function multiple times every time step.

In order to learn the MMDP model, we assume that the DDN structure is fully described prior to learning. We use this structure for three purposes: to improve learning, to increase sample-efficiency, and to structure the CPS queue so that it contains parent sets rather than complete states (see Section 2.4). Given the structure of the MMDP, we learn the transition dynamics by maintaining estimates of the proportion that a state s'_i is reached after taking a local joint action $\overline{\mathbf{a}}$ in a local joint state $\overline{\mathbf{s}}$ of its parents in the DDN (Andre, Friedman, and Parr 1998):

$$T_i(s'_i \mid \overline{\mathbf{s}}, \overline{\mathbf{a}}) = \frac{N_{\overline{\mathbf{s}},\overline{\mathbf{a}},s'_i} + N^0_{\overline{\mathbf{s}},\overline{\mathbf{a}},s'_i}}{\sum_{s''_i \in S_i} N_{\overline{\mathbf{s}},\overline{\mathbf{a}},s''_i} + N^0_{\overline{\mathbf{s}},\overline{\mathbf{a}},s''_i}} \tag{2}$$

where $N^0_{\overline{\mathbf{s}},\overline{\mathbf{a}},s''_i}$ are the priors on the model before interaction begins. The reward function can be learned accordingly.

2.2 One-step Q-Function Updates

CPS learns the Q-function of the MMDP in a factorized manner. This means that it approximates the true Q-function as a linear sum of smaller components:

$$Q(\mathbf{s}, \mathbf{a}) \approx \hat{Q}(\mathbf{s}, \mathbf{a}) = \sum_x \hat{Q}_x(\overline{\mathbf{s}}, \overline{\mathbf{a}}) \tag{3}$$

We assume that the domains of each $\hat{Q}_x$ are obtained by back-propagating some subset of the components in $\mathbf{S}$ (we call these *basis domains*). This is the same process that is used to obtain $\hat{Q}$ from a suitably factored value function $\hat{V}$ made out of basis functions in approximate MMDP planning (Guestrin, Koller, and Parr 2002). This assumption is important in order to correctly back-propagate rewards. For example, in Figure 1a, selecting variable S'_1 as a basis results in a corresponding domain of its $\hat{Q}$ component of $S_1 \wedge S_2 \wedge A_1$. Selecting both $S'_1 \wedge S'_2$ results in a corresponding domain of $S_1 \wedge S_2 \wedge S_3 \wedge A_1 \wedge A_2$. The set of basis domains can be chosen arbitrarily—although some choices tend to perform better than others. Defining larger and more domains leads to better accuracy, but at the cost of higher computation times for joint action selection. For example, CPS uses variable elimination to select the optimal action, which is known to be exponential in the induced width of the coordination graph of the factored Q-function (after conditioning on the current state $\mathbf{s}$), which in turn increases with the size and number of the domains in the factorization. Constructing a $\hat{Q}$ made up of a single factor with the entire state space $\mathbf{S}$ as its basis degenerates the algorithm to single-agent PS, as it would make $\hat{Q} = Q$.

After each interaction with the environment, we obtain an experience tuple, $\langle \mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r} \rangle$, which CPS uses to update the factored Q-function. Since CPS takes a model-based approach, the ideal update would first use the data to update the model (both T and R), and then use the model to update the Q-function:

$$\hat{Q}(\mathbf{s}, \mathbf{a}) = R(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}'} T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) \hat{V}(\mathbf{s}') \tag{4}$$

Indeed, this operation, using a $\hat{V}$ factored with the relevant basis domains, is not only possible but also relatively cheap (Guestrin, Koller, and Parr 2001): each $\hat{Q}_x$ component can be updated separately, while ensuring that $\hat{Q}$ fully reflects the changes of the just updated model. However, back-propagating the changes made to $\hat{Q}$ to $\hat{V}$ is not straightforward; to do so, an algorithm would need to solve an expensive LP (Guestrin, Koller, and Parr 2001). This operation makes performing batches of updates to the Q-function computationally infeasible, and would prevent the algorithm from leveraging the learnt MMDP model.

Because full backups are not feasible for the factored Q-functions, CPS employs local TD updates:

$$\hat{Q}(\mathbf{s},\mathbf{a}) = \hat{Q}(\mathbf{s},\mathbf{a}) + \alpha \left(R(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} \hat{Q}(\mathbf{s}',\mathbf{a}') - \hat{Q}(\mathbf{s},\mathbf{a}) \right) \tag{5}$$

To perform this update, CPS requires the optimal joint action, $\mathbf{a}'^{\star}$, for the next joint state $\mathbf{s}'$, and thus needs to maximize $\hat{Q}$ for this joint state. This can be done efficiently using *Variable Elimination* (VE).[1] Once the optimal full joint action $\mathbf{a}'^{\star}$ for $\mathbf{s}'$ has been found, we can then split the update of the Q-function per component:

$$\hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) = \hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) + \alpha \left(R_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) + \gamma \hat{Q}_x(\overline{\mathbf{s}'},\overline{\mathbf{a}'^{\star}}) - \hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) \right) \tag{6}$$

The only thing left is to compute $R_x(\bar{\mathbf{s}},\bar{\mathbf{a}})$, as its domain will likely be different from any of the R_i functions in the learnt MMDP model, due to the back-propagation of the basis domains. Hence, for each S_i in the original basis domain of $\hat{Q}_x$, we compute R_x as a weighted sum of their respective R_i, where the weights are inversely proportional to how many $\hat{Q}_y$ have S_i in their original basis domain:

$$R_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) = \sum_i I(S_i \in Basis_x(\bar{\mathbf{S}})) \frac{R_i(\bar{\bar{\mathbf{s}}},\bar{\bar{\mathbf{a}}})}{N_{\hat{Q}}(Basis(S_i))} \tag{7}$$

where I is the indicator function and $N_{\hat{Q}}(Basis(S_i))$ is the number of $\hat{Q}_y$ factors ($\hat{Q}_x$ included) which have S_i in their basis domain. We also indicate with $\bar{\bar{\mathbf{s}}},\bar{\bar{\mathbf{a}}}$ the parts of $\bar{\mathbf{s}},\bar{\mathbf{a}}$ required by R_i. This works because we are assuming that R has the same DDN structure as T, and due to the back-propagation, if R_x had S_i in its basis domain, then $\bar{\mathbf{s}}$ and $\bar{\mathbf{a}}$ must contain the parents of s_i which are required by R_i.

These updates are similar to those performed in SCQL (Kok and Vlassis 2004; 2005). However, CPS determines the shape of the Q-function and how it partitions the rewards using the basis domains, while SCQL does this in terms of agents and their direct neighbours.

[1] We note that VE is an exact algorithm for coordination graphs that can be replaced by approximate algorithms (Kok and Vlassis 2005; Marinescu and Dechter 2005; Liu and Ihler 2012) if need be.

2.3 Temporal Differences for Batch Q-Function Updates

In order to perform batch updates on the Q-function to back-propagate useful values, CPS computes a series of temporal differences (TD) which represent how much the Q-function has changed due to the last update. This process is different from the single-agent PS algorithm (Moore and Atkeson 1993; Andre, Friedman, and Parr 1998) in two key aspects: how we define the TD, and how CPS computes it.

Single-agent PS computes a single TD for the state s that is being updated. In our case, we update each $\hat{Q}_x$ independently, resulting in a series of Δ_x, defined as

$$\Delta_x = R_x(\bar{\mathbf{s}},\bar{\mathbf{a}}) + \gamma \hat{Q}_x(\overline{\mathbf{s}'},\overline{\mathbf{a}'^{\star}}) - \hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}}). \tag{8}$$

Given the factored Q-function (Equation 3) and factor update rule (Equation 6), we could reconstruct a full TD by simply summing them together, i.e., $\Delta = \sum_x \Delta_x$. However, given the structure of our MMDP, computing a single TD for the whole state $\mathbf{s}$ is wasteful: given the structure of our MMDP, each Δ_x of $\hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}})$ not only reflects changes for the single state $\mathbf{s}$, but all states that contain $\bar{\mathbf{s}}$. In the same way, the change in value of $\hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}})$ not only affects the value of all possible parents of $\mathbf{s}$, but of all possible parents of $\bar{\mathbf{s}}$. An interesting consequence of this fact is that our definition of a TD changes from the single-agent PS algorithm, where the TD is computed with respect to the change in the value function for the updated state s. This is because there is no reason to perform additional updates if the actual value function for the state is not modified. In our case, this is different, as any single $\hat{Q}_x(\bar{\mathbf{s}},\bar{\mathbf{a}})$ value might contribute to $\hat{V}(\mathbf{s})$ in some state where $\bar{\mathbf{s}} \subseteq \mathbf{s}$—although actually obtaining this information is not practical. Instead, we assume that the change is indeed modifying $\hat{V}$, and compute the TDs directly on the change that CPS performs without the need to actually check $\hat{V}$ or perform any additional maximizations. As these TDs are computed without the knowledge of how much each Δ_x actually affects $\hat{V}$, this means that CPS tends to overestimate changes in order to not miss any.

Given these considerations, our goal is to obtain a series of TDs which:

1. Reflect the actual changes we have applied to the Q-function, without passing through $\hat{V}$.
2. Do not lose the information about how much each part of the state space was changed.
3. Are suitable for back-propagation in order to select new $(\mathbf{s},\mathbf{a})$ pairs to update.

The Δ_x satisfy the first two constraints, but not the third, as back-propagating their domain would mismatch the domain of any $\hat{Q}_x$. Therefore, we transform them into a set of N elements Δ_i, one for each factor of the state-space

$$\Delta_i = I(i \in x_s) \frac{\Delta_x}{|x_s|} \tag{9}$$

where x_s refers to the state domain part of $\hat{Q}_x$. In this format, these can be used in the back-propagation step (see Section 2.4). Note that the Δ_i are only used to sample from the queue, and not in the actual $\hat{Q}$ updates.

2.4 Updating the Queue

CPS maintains a priority queue containing the changes it has done to the Q-function, where the priority is equal to the magnitude of the change. The key idea is to back-propagate these changes during batch updates to the Q-values of the parents of each s_i. Since CPS must do this given the limited time between interactions with the environment, the updates should be done from the highest priority to the lowest until the time runs out.

CPS uses the Δ_i to update the queue. Note that we go a single step backwards in time, such that our $\mathbf{s}$ is now considered a *future state*. Then, for each s_i, CPS iterates over all the possible values of its parents $Parents(s_i) = (\overline{\mathbf{s}''}, \overline{\mathbf{a}''})$ as determined by the DDN of the problem.

The algorithm can now compute the priority p of these parents as $p = T_i(s_i|Parents(s_i)) \cdot \Delta_i$, i.e., the probability of observing this transition times the estimated amount of change of that particular component. Note that we do not include γ in the priority since it would not affect the relative ordering of the queue. If p is greater than a threshold value θ, it is added as a new entry to the queue, or increase its value by p if such an entry already exists. The entry will contain the parents of s'_i (and their values), together with the computed priority p.

2.5 Sampling from the Queue

CPS needs to sample from the queue to perform the batch updates required to increase the accuracy of the Q-function. In single-agent PS, this operation is relatively straightforward: a state-action pair is sampled, and a new Q update is done. However, in CPS, the queue only contains partial state-action pairs, while a complete state-action pair is necessary to sample a new state and reward from the model and select the next optimal joint action needed to perform a full Q-function update. Thus, CPS extracts multiple compatible entries from the priority queue, and combines them into a complete state-action pair.

While the optimal choice would be the set with the highest combined priority, finding this set is equivalent to the knapsack problem, which is known to be NP-hard. A good heuristic would be to traverse the priority queue in order, greedily extracting the first $(\overline{\mathbf{s}}, \overline{\mathbf{a}})$ entry from the queue, and then greedily selecting all entries which are compatible with both it and previously selected entries, where compatible means that the same parents (s_i, a_i) must have the same values in all selected entries. Unfortunately, this option is also not viable, as it would require examining all entries in the queue, which would be too computationally expensive to do repeatedly. However, by ignoring the order of priority, this strategy can be implemented very efficiently with the help of *radix trees* for bookkeeping, which significantly reduces the number of comparisons needed. Thus, CPS still pops the highest priority entry from the queue, but randomizes the subsequent search order. In this way, we ensure that every compatible entry in the queue has a chance of being selected, and that all updates are performed in the limit.

Once the set of entries has been finalized, CPS removes them from the priority queue, and proceeds to use them to update the factored Q-function. If some elements local state-action (s_i, a_i) required to construct a full joint action are not in the selected set, CPS supplements these by uniformly sampling them from their respective domain. Another option would be to iterate over every possible combination of the missing values and do an update for each combination, but this operation would too expensive, as the number of updates would increase exponentially with the number of unassigned values. Finally, using the newly obtained $(\mathbf{s}, \mathbf{a})$ pair, CPS uses the model to sample a new state $\mathbf{s}'$ and rewards $\mathbf{r}$, and updates $\hat{Q}$ as described in Section 2.2.

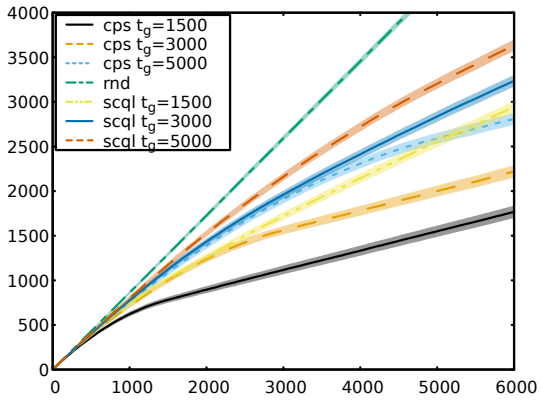
Algorithm 1: Cooperative Prioritized Sweeping

while True **do**
  Select action $\mathbf{a}$ given state $\mathbf{s}$ following policy π
  $(\mathbf{s}', \mathbf{r}) \leftarrow$ Execute action $\mathbf{a}$ in state $\mathbf{s}$
  Update T, R using $(\mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{r})$
  $\mathbf{a}'^{\star} \leftarrow$ Use variable elimination for $\mathbf{s}'$
  Update $\hat{Q}(\mathbf{s}, \mathbf{a})$ using $(\mathbf{s}, \mathbf{a}, \mathbf{s}', \mathbf{a}'^{\star})$
  for every state factor i **do**
    $\Delta_i \leftarrow$ Compute from TDs of $\hat{Q}$
    for every pair $\overline{\mathbf{s}''}, \overline{\mathbf{a}''}$ that is parent of s_i **do**
      $p \leftarrow \Delta_i \cdot T(s'_i = s_i \mid \overline{\mathbf{s}''}, \overline{\mathbf{a}''})$
      if $p > \theta$ **then**
        Add to Queue $\overline{\mathbf{s}''}, \overline{\mathbf{a}''}$ with priority p
      end if
    end for
  end for
  for number of batch updates **do**
    $(\overline{\mathbf{s}}, \overline{\mathbf{a}}) \leftarrow$ Pop Queue max
    for every pair $(\overline{\mathbf{s}''}, \overline{\mathbf{a}''})$ in Queue, randomly **do**
      if $(\overline{\mathbf{s}}, \overline{\mathbf{a}})$ and $(\overline{\mathbf{s}''}, \overline{\mathbf{a}''})$ are compatible **then**
        Append $(\overline{\mathbf{s}''}, \overline{\mathbf{a}''})$ to $(\overline{\mathbf{s}}, \overline{\mathbf{a}})$
        Remove $(\overline{\mathbf{s}''}, \overline{\mathbf{a}''})$ from Queue
      end if
    end for
    $(\mathbf{s}, \mathbf{a}) \leftarrow$ Fill missing values of $(\overline{\mathbf{s}}, \overline{\mathbf{a}})$ randomly
    $(\mathbf{s}', \mathbf{r}) \leftarrow$ Sample from T, R using $\mathbf{s}$ and $\mathbf{a}$
    Update $\hat{Q}$
    Update Queue with all Δ_i
  end for
end while

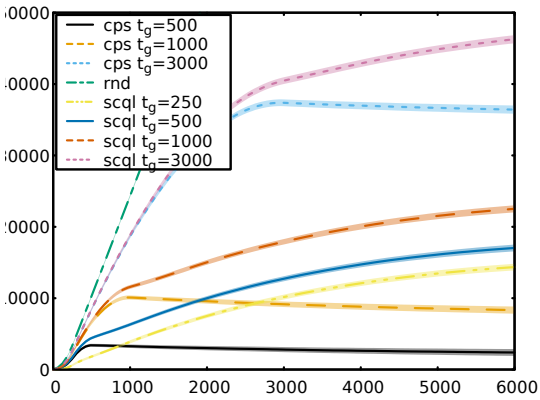
3 Experiments

To test the performance of CPS, we evaluate it empirically on the SysAdmin problem (Guestrin, Koller, and Parr 2002) and randomly generated MMDPs.

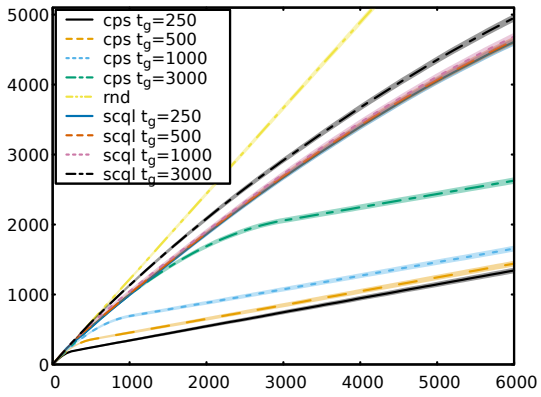
In the SysAdmin setting, a series of interconnected machines randomly obtain jobs that they must complete. Over time, the machines may fail or die, and this in turn affects the probability of failures for adjacent machines. Each machine is associated with an agent, which can either do nothing or reboot its machine. Rebooting a machine resets its state, but loses all work. Each completed job receives a reward of 1 for its associated agent. We use three different topologies: a ring with 300 agents, a torus with 100 agents in a 10x10 grid and



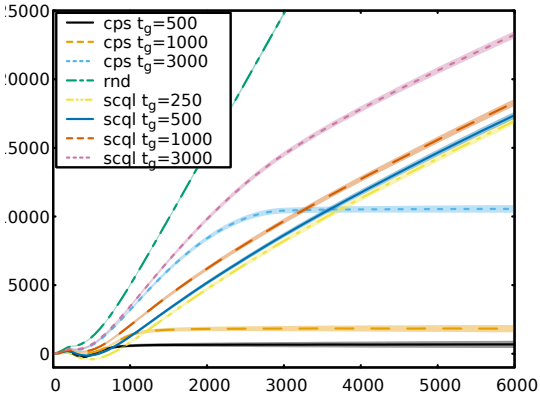
(a) Regret over time w.r.t. the best possible approximation (LP) in the shared ring control setting.



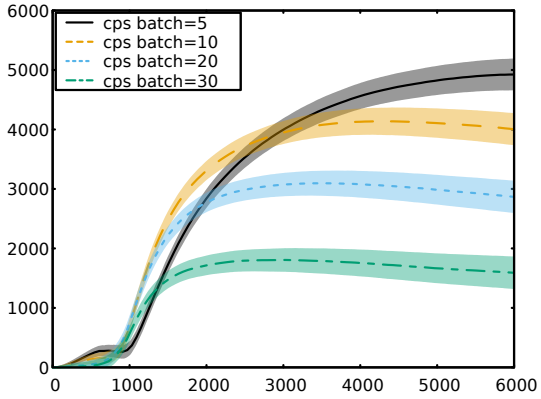
(b) Regret over time w.r.t. CPS $t_g = 250$ in the ring setting, with 300 agents.



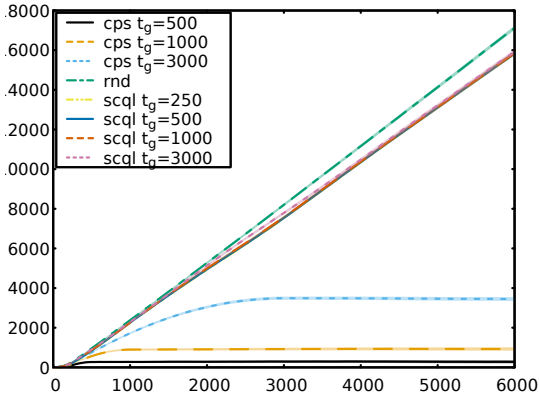
(c) Regret over time w.r.t. the best possible approximation (LP) in the random setting with 3 agents.



(d) Regret over time w.r.t. CPS $t_g = 250$ in the torus setting, with 100 agents.



(e) Regret over time w.r.t. a batch size of 50 in the torus setting. t_g here is 1000 for all runs.



(f) Regret over time w.r.t. CPS $t_g = 250$ in the random setting, with 15 agents.

Figure 2: Mean and standard deviation of cumulative regret over time for different methods in each setting (lower is better). Note that in all plots but 2a and 2c regret may go down since it is computed w.r.t. a specific algorithm, as the LP upper bound is too expensive to compute.

a shared control setting with a ring of 12 machines where each is controlled by its two adjacent agents. In this last setting, a machine reboots with a probability of 0.15 if only one agent issues the rebooting command, and with a probability of 1 if both do. We only use 12 agents so that it is feasible to use a factored LP (Guestrin, Koller, and Parr 2002) to obtain the best possible approximation and compute the true regret. We select each machine's state and load pair as the basis do-

mains, which is equivalent to the "single" basis reported in (Guestrin, Koller, and Parr 2002).

Additionally, we evaluate the performance of CPS on randomly generated MMDPs. The DDNs for these problems were generated by connecting each S'_i with S_i, plus a random number of state and action factors (max 3) locally close to i. The transition tables were filled by uniformly sampling transition vectors. We generated sparse reward functions containing random values in $[-1, 0, 1]$ for the parents of random factors S_i, which are selected independently with probability 0.3. We select as basis domains all pairs of adjacent state factors (S_i, S_{i+1}). We show results for two randomly generated MMDPS, one with 4 state factors and 3 agents, and another with 20 state factors and 15 agents. Both environments had to be kept relatively small, as on one hand the LP approach was very slow in these less structured environments, while on the other SCQL had trouble handling the large number of rules required to represent the Q-function (in the millions) with more than 15 agents.

We compare our results against 3 benchmarks: a random policy as a naive approach, the factored LP planning method on the ground truth MMDP model as the upper bound (Guestrin, Koller, and Parr 2002), and Sparse Cooperative Q-Learning (SCQL) (Kok and Vlassis 2004). SCQL runs used optimistic initialization to improve its exploration, with all its Q-function values set to 5.0, as without it SCQL was not able to learn correctly. Both 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. We tested different time steps at which the exploration probability ε must reach 0, reported as parameter t_g in Figure 2. 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. All results were averaged over 100 runs. Each experiment ran on a single core of an AMD FX-8350 CPU.

CPS consistently outperforms SCQL in all settings, and is able to obtain good policies with very few samples: 250 time steps are enough to learn a close-to-optimal policy in an environment with 2^{300} actions and 9^{300} states. At the same time, exploring for a longer time leads to slightly better policies.

Figure 2e shows the relationship between regret and batch size in CPS in the SysAdmin torus setting. As the batch size decreases, regret increases. However, smaller batch sizes still converge to the same policy over time. Because of the limited exploration time, the learnt model is imperfect. This results in slightly sub-optimal decisions inversely correlated with the batch size, as batch updates are sampled from the learnt model. Therefore, lower batch sizes show decreasing regret in the later time steps.

4 Related Work

In general, exact planning in factored multi-agent settings is unfeasible for larger numbers of agents, for all but problems with very few agents (Scharpff et al. 2016). Some sub-classes of decentralized versions of such problems exist that do permit an exact factorization of the value function (Becker et al. 2004; Nair et al. 2005; Witwicki and Durfee 2010). However, the cost of decentralization is still arbitrarily large, and planning in these specialized models is still expensive. Therefore, for model-based reinforcement learning it is better to directly approximate a factored sparse Q-function, as this obviates the need for the additional oversimplifying model restrictions that using such models would entail.

In this paper, we make use of variable elimination (Guestrin, Koller, and Parr 2002) 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 (Kok and Vlassis 2005), AND/OR tree search (Marinescu and Dechter 2005), or variational methods (Liu and Ihler 2012). 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.

Finally, recent advances have been made in deep multi-agent reinforcement learning (Gupta, Egorov, and Kochenderfer 2017; Foerster et al. 2018; Hernandez-Leal, Kartal, and Taylor 2018). The most related method to ours from this line of research is QMIX (Rashid et al. 2018) which—to our knowledge—is the only deep MARL method to exploit some type of factorization. However, this method is limited to Q-functions that are monotonic in its constitution components, and thus not yet able to handle arbitrary factorizations of the Q-function. It is our hope that the contributions presented in this paper will also contribute to future deep MARL methods that can handle any factorization, especially in light of the fact that it has been observed that the sparse factored representation for the value functions we employ is equivalent to a single layer neural network with features corresponding to its basis functions (Guestrin, Koller, and Parr 2002).

5 Discussion and Conclusions

We have presented a new model-based reinforcement learning algorithm, Cooperative Prioritized Sweeping, which exploits the structure of a coordination graph in an MMDP to both learn a model of the environment and update the Q-function in a sample-efficient manner. The algorithm maintains a priority queue containing partial state-action pairs, representing the parts of the Q-function which are most beneficial to update. The queue is sampled in batches between interactions with the environment, and the model is used to sample possible new state and reward pairs, which are then used to update the Q-function. CPS can learn good policies using significantly less data than the previous state-of-the-art, resulting in significantly less regret over time.

Currently, CPS uses a naive ε-greedy exploration strategy. In future work, we aim to improve exploration; instead of randomly sampling a completely random joint-action when exploring, we would like to select the joint-action that maximizes the collection of information from each T and R factor, similar to previous work in multi-agent bandits (Bargiacchi et al. 2018). Such an approach could further reduce the sample complexity of CPS. We will also implement support for context-specific independence using rules (Guestrin, Venkataraman, and Koller 2002). Finally, we aim to use the

techniques presented in this paper as a basis for constructing a deep MARL that can fully support arbitrary factored Q-functions.

References

Andre, D.; Friedman, N.; and Parr, R. 1998. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*, 1001–1007.

Bargiacchi, E.; Verstraeten, T.; Roijers, D. M.; Nowé, A.; and van Hasselt, H. 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*, volume 80, 491–499.

Becker, R.; Zilberstein, S.; Lesser, V.; and Goldman, C. V. 2004. Solving transition independent decentralized markov decision processes. *Journal of Artificial Intelligence Research* 22:423–455.

Claes, D.; Oliehoek, F. A.; Baier, H.; and Tuyls, K. 2017. Decentralised online planning for multi-robot warehouse commissioning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, 492–500. International Foundation for Autonomous Agents and Multiagent Systems.

Foerster, J. N.; Farquhar, G.; Afouras, T.; Nardelli, N.; and Whiteson, S. 2018. Counterfactual multi-agent policy gradients. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research* 19:399–468.

Guestrin, C.; Koller, D.; and Parr, R. 2001. Max-norm projections for factored MDPs. In *IJCAI*, volume 1, 673–682.

Guestrin, C.; Koller, D.; and Parr, R. 2002. Multiagent planning with factored MDPs. In *NIPS 2002: Advances in Neural Information Processing Systems 15*, 1523–1530.

Guestrin, C.; Venkataraman, S.; and Koller, D. 2002. Context-specific multiagent coordination and planning with factored MDPs. In *AAAI/IAAI*, 253–259.

Gupta, J. K.; Egorov, M.; and Kochenderfer, M. 2017. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, 66–83. Springer.

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

Kok, J., and Vlassis, N. 2004. Sparse cooperative Q-learning. In *ICML 2004: Proceedings of the twenty-first international conference on Machine learning*, 61–68.

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

Kok, J. R.; Spaan, M. T. J.; Vlassis, N.; et al. 2003. Multi-robot decision making using coordination graphs. In *Proceedings of the 11th International Conference on Advanced Robotics, ICAR*, volume 3, 1124–1129.

Liu, Q., and Ihler, A. T. 2012. Belief propagation for structured decision making. *arXiv preprint arXiv:1210.4897*.

Marinescu, R., and Dechter, R. 2005. And/or branch-and-bound for graphical models. In *IJCAI*, 224–229.

Moore, A., and Atkeson, C. 1993. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning* 13(1):103–130.

Nair, R.; Varakantham, P.; Tambe, M.; and Yokoo, M. 2005. Networked distributed pomdps: A synthesis of distributed constraint optimization and pomdps. In *AAAI*, volume 5, 133–139.

Rashid, T.; Samvelyan, M.; Witt, C. S.; Farquhar, G.; Foerster, J.; and Whiteson, S. 2018. QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*, 4292–4301.

Scharpff, J.; Roijers, D. M.; Oliehoek, F. A.; Spaan, M. T. J.; and De Weerdt, M. M. 2016. Solving transition-independent multi-agent MDPs with sparse interactions. In *AAAI*, 3174–3180.

Wiering, M. A. 2000. Multi-agent reinforcement learning for traffic light control. In *Machine Learning: Proceedings of the Seventeenth International Conference (ICML'2000)*, 1151–1158.

Witwicki, S. J., and Durfee, E. H. 2010. Influence-based policy abstraction for weakly-coupled dec-pomdps. In *Twentieth International Conference on Automated Planning and Scheduling*.