Journal of Machine Learning Research X (XXXX) X-XX Submitted 12/05; Published XX/06

Collaborative Multiagent Reinforcement Learning by Payoff Propagation

Jelle R. Kok JELLEKOK@SCIENCE.UVA.NL
Nikos Vlassis VLASSIS@SCIENCE.UVA.NL
Informatics Institute, University of Amsterdam
Kruislaan 403, 1098SJ, Amsterdam, The Netherlands

Editor: Michael Littman

Abstract

In this article we describe a set of scalable techniques for learning the behavior of a group of agents in a collaborative multiagent setting. As a basis we use the framework of coordination graphs of Guestrin, Koller, and Parr (2002a) which exploits the dependencies between agents to decompose the global payoff function into a sum of local terms. First, we deal with the single-state case and describe a payoff propagation algorithm that computes the individual actions that approximately maximize the global payoff function. The method can be viewed as the decision-making analogue of belief propagation in Bayesian networks. Second, we focus on learning the behavior of the agents in sequential decision-making tasks. We introduce different model-free reinforcement-learning techniques, unitedly called Sparse Cooperative Q-learning, which approximate the global action-value function based on the topology of a coordination graph, and perform updates using the contribution of the individual agents to the maximal global action value. The combined use of an edge-based decomposition of the action-value function and the payoff propagation algorithm for efficient action selection, result in an approach that scales only linearly in the problem size. We provide experimental evidence that our method outperforms related multiagent reinforcement-learning methods based on temporal differences.

Keywords: collaborative multiagent system, coordination graph, reinforcement learning, Q-learning, belief propagation

1. Introduction

A multiagent system (MAS) consists of a group of agents that reside in an environment and can potentially interact with each other (Sycara, 1998; Weiss, 1999; Durfee, 2001; Vlassis, 2003). The existence of multiple operating agents makes it possible to solve inherently distributed problems, but also allows one to decompose large problems, which are too complex or too expensive to be solved by a single agent, into smaller subproblems.

In this article we are interested in collaborative multiagent systems in which the agents have to work together in order to optimize a shared performance measure. In particular, we investigate sequential decision-making problems in which the agents repeatedly interact with their environment and try to optimize the long-term reward they receive from the system, which depends on a sequence of joint decisions. Specifically, we focus on *inherently cooper-*

ative tasks involving a large group of agents in which the success of the team is measured by the specific combination of actions of the agents (Parker, 2002). This is different from other approaches that assume implicit coordination through either the observed state variables (Tan, 1993; Dutta et al., 2005), or reward structure (Becker et al., 2003). We concentrate on model-free learning techniques in which the agents do not have access to the transition or reward model. Example application domains include network routing (Boyan and Littman, 1994; Dutta et al., 2005), sensor networks (Lesser et al., 2003; Modi et al., 2005), but also robotic teams, for example, exploration and mapping (Burgard et al., 2000), motion coordination (Arai et al., 2002) and RoboCup (Kitano et al., 1995; Kok et al., 2005).

Existing learning techniques have been proved successful in learning the behavior of a single agent in stochastic environments (Tesauro, 1995; Crites and Barto, 1996; Ng et al., 2004). However, the presence of multiple learning agents in the same environment complicates matters. First of all, the action space scales exponentially with the number of agents. This makes it infeasible to apply standard single-agent techniques in which an action value, representing expected future reward, is stored for every possible state-action combination. An alternative approach would be to decompose the action value among the different agents and update them independently. However, the fact that the behavior of one agent now influences the outcome of the individually selected actions of the other agents results in a dynamic environment and possibly compromises convergence. Other difficulties, which are outside the focus of this article, appear when the different agents receive incomplete and noisy observations of the state space (Goldman and Zilberstein, 2004), or have a restricted communication bandwidth (Pynadath and Tambe, 2002; Goldman and Zilberstein, 2003).

For our model representation we will use the collaborative multiagent Markov decision process (collaborative multiagent MDP) model (Guestrin, 2003). In this model each agent selects an individual action in a particular state. Based on the resulting joint action the system transitions to a new state and the agents receive an *individual* reward. The global reward is the sum of all individual rewards. This approach differs from other multiagent models, for example, multiagent MDPs (Boutilier, 1996) or decentralized MDPs (Bernstein et al., 2000), in which all agents observe the global reward. In a collaborative MDP, it is still the goal of the agents to optimize the global reward, but the individually received rewards allow for solution techniques that take advantage of the problem structure.

One such solution technique is based on the framework of coordination graphs (CGs) (Guestrin et al., 2002a). This framework exploits that in many problems only a few agents depend on each other and decomposes a coordination problem into a combination of simpler problems. In a CG each node represents an agent and connected agents indicate a local coordination dependency. Each dependency corresponds to a local payoff function which assigns a specific value to every possible action combination of the involved agents. The global payoff function equals the sum of all local payoff functions. To compute the joint action that maximizes the global payoff function, a variable elimination (VE) algorithm can be used (Guestrin et al., 2002a). This algorithm operates by eliminating the agents one by one after performing a local maximization step, and has exponential complexity in the induced tree width (the size of the largest clique generated during the node elimination).

In this article we investigate different distributed learning methods to coordinate the behavior between the agents. The algorithms are distributed in the sense that each agent only needs to communicate with the neighboring agents on which it depends. Our contribution

is two-fold. First, we describe a 'payoff propagation' algorithm (max-plus) (Vlassis et al., 2004; Kok and Vlassis, 2005) to find an approximately maximizing joint action for a CG in which all local functions are specified beforehand. Our algorithm exploits the fact that there is a direct duality between computing the maximum a posteriori configuration in a probabilistic graphical model and finding the optimal joint action in a CG; in both cases we are optimizing over a function that is decomposed in local terms. This allows message-passing algorithms that have been developed for inference in probabilistic graphical models to be directly applicable for action selection in CGs. Max-plus is a popular method of that family. In the context of CG, it can therefore be regarded as an approximate alternative to the exact VE algorithm for multiagent decision making. We experimentally demonstrate that this method, contrary to VE, scales to large groups of agents with many dependencies.

The problem of finding the maximizing joint action in a fixed CG is also related to the work on distributed constraint satisfaction problems (CSPs) in constraint networks (Pearl, 1988). These problems consist of a set of variables which each take a value from a finite, discrete domain. Predefined constraints, which have the values of a subset of all variables as input, specify a cost. The objective is to assign values to these variables such that the total cost is minimized (Yokoo and Durfee, 1991; Dechter, 2003).

As a second contribution, we study sequential decision-making problems in which we learn the behavior of the agents. For this, we apply model-free reinforcement-learning techniques (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). This problem is different than finding the joint action that maximizes predefined payoff relations, since in this case the payoff relations themselves have to be learned. In our approach, named *Sparse Cooperative Q-learning* (Kok and Vlassis, 2004), we analyze different decompositions of the global action-value function using CGs. The structure of the used CG is determined beforehand, and reflects the specific problem under study. For a given CG, we investigate both a decomposition in terms of the nodes (or agents), as well as a decomposition in terms of the edges. In the agent-based decomposition the local function of an agent is based on its own action and those of its neighboring agents. In the edge-based decomposition each local function is based on the actions of the two agents forming this edge. Each state is related to a CG with a similar decomposition, but with different values for the local functions. To update the local action-value function for a specific state, we use the contribution of the involved agents to the maximal global action value, which is computed using either the max-plus or VE algorithm. We perform different experiments on problems involving a large group of agents with many dependencies and show that all variants outperform existing temporal-difference based learning techniques in terms of the quality of the extracted policy. Note that in our work we only consider temporal-difference methods; other multiagent reinforcement-learning methods exist that are based, for example, on policy search (Peshkin et al., 2000; Moallemi and Van Roy, 2004) or Bayesian approaches (Chalkiadakis and Boutilier, 2003).

The remainder of this article is structured as follows. We first review the notion of a CG and the VE algorithm in Section 2. Next, in Section 3, we discuss our approximate alternative to VE based on the max-plus algorithm and perform experiments on randomly generated graphs. Then, we switch to sequential decision-making problems. First, we review several existing multiagent learning methods in Section 4. In Section 5, we introduce the different variants of our Sparse Cooperative Q-learning method, and give experimental results on several learning problems in Section 6. We end with the conclusions in Section 7.

2. Coordination Graphs and Variable Elimination

All agents in a collaborative multiagent system can potentially influence each other. It is therefore important to ensure that the actions selected by the individual agents result in optimal decisions for the group as a whole. This is often referred to as the *coordination problem*. In this section we review the problem of computing a coordinated action for a group of n agents as described by Guestrin et al. (2002a). Each agent i selects an individual action a_i from its action set $\mathcal{A}_i$ and the resulting *joint* action $\mathbf{a} = (a_1, \ldots, a_n)$, as all other vectors of two or more variables in this article emphasized using a bold notation, generates a payoff $u(\mathbf{a})$ for the team. The coordination problem is to find the optimal joint action $\mathbf{a}^*$ that maximizes $u(\mathbf{a})$, that is, $\mathbf{a}^* = \arg\max_{\mathbf{a}} u(\mathbf{a})$.

We can compute the optimal joint action by enumerating over all possible joint actions and select the one that maximizes $u(\mathbf{a})$. However, this approach quickly becomes impractical, as the size of the joint action space $|\mathcal{A}_1 \times \ldots \times \mathcal{A}_n|$ grows exponentially with the number of agents n. Fortunately, in many problems the action of one agent does not depend on the actions of all other agents, but only on a small subset. For example, in many real-world domains only agents which are spatially close have to coordinate their actions.

The framework of coordination graphs (CGs) (Guestrin et al., 2002a) is a recent approach to exploit these dependencies. This framework assumes the action of an agent i only depends on a subset of the other agents, $j \in \Gamma(i)$. The global payoff function $u(\mathbf{a})$ is then decomposed into a linear combination of local payoff functions, as follows,

$$u(\mathbf{a}) = \sum_{i=1}^{n} f_i(\mathbf{a}_i). \tag{1}$$

Each local payoff function f_i depends on a subset of all actions, $\mathbf{a}_i \subseteq \mathbf{a}$, where $\mathbf{a}_i = \mathcal{A}_i \times (\times_{j \in \Gamma(i)} \mathcal{A}_j)$, corresponding to the action of agent i and those of the agents on which it depends. This decomposition can be depicted using an undirected graph $G = (V, E)$ in which each node $i \in V$ represents an agent and an edge $(i, j) \in E$ indicates that the corresponding agents have to coordinate their actions, that is, $i \in \Gamma(j)$ and $j \in \Gamma(i)$. The global coordination problem is now replaced by a number of local coordination problems each involving fewer agents.

In the remainder of this article, we will focus on problems with payoff functions including at most two agents. Note that this still allows for complicated coordinated structures since every agent can have multiple pairwise dependency functions. Furthermore, it is possible to generalize the proposed techniques to payoff functions with more than two agents because any arbitrary graph can be converted to a graph with only pairwise inter-agent dependencies (Yedidia et al., 2003; Loeliger, 2004). To accomplish this, a new agent is added for each local function that involves more than two agents. This new agent contains an individual local payoff function that is defined over the combined actions of the involved agents, and returns the corresponding value of the original function. Note that the action space of this newly added agent is exponential in its neighborhood size (which can lead to intractability in the worst case). Furthermore, new pairwise payoff functions have to be defined between each involved agent and the new agent in order to ensure that the action selected by the involved agent corresponds to its part of the (combined) action selected by the new agent.

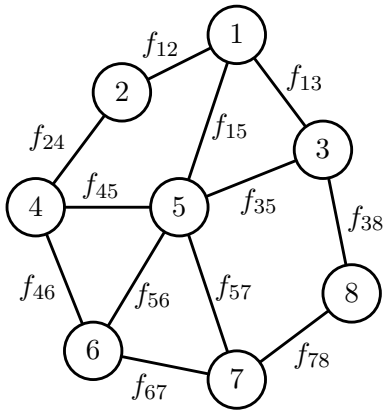


Figure 1: Example CG with eight agents; an edge represents a coordination dependency.

Allowing only payoff functions defined over at most two agents, the global payoff function $u(\mathbf{a})$ can be decomposed as

$$u(\mathbf{a}) = \sum_{i \in V} f_i(a_i) + \sum_{(i,j) \in E} f_{ij}(a_i, a_j). \tag{2}$$

A local payoff function $f_i(a_i)$ specifies the payoff contribution for the individual action a_i of agent i, and f_{ij} defines the payoff contribution for pairs of actions (a_i, a_j) of neighboring agents $(i, j) \in E$. Fig. 1 shows an example CG with 8 agents.

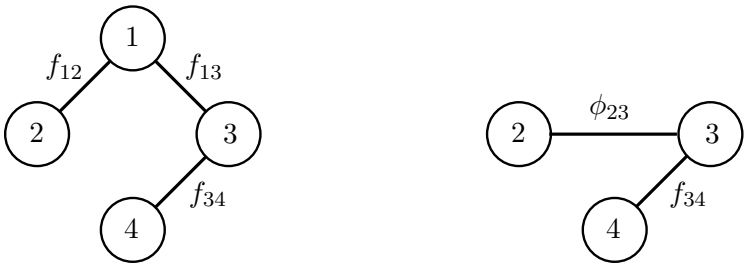
In order to solve the coordination problem and find $\mathbf{a}^* = \arg\max_{\mathbf{a}} u(\mathbf{a})$ we can apply the variable elimination (VE) algorithm (Guestrin et al., 2002a), which is in essence identical to variable elimination in a Bayesian network (Zhang and Poole, 1996). The algorithm eliminates the agents one by one. Before an agent (node) is eliminated, the agent first collects all payoff functions related to its edges. Next, it computes a conditional payoff function which returns the maximal value it is able to contribute to the system for every action combination of its neighbors, and a best-response function (or conditional strategy) which returns the action corresponding to the maximizing value. The conditional payoff function is communicated to one of its neighbors and the agent is eliminated from the graph. Note that when the neighboring agent receives a function including an action of an agent on which it did not depend before, a new coordination dependency is added between these agents. The agents are iteratively eliminated until one agent remains. This agent selects the action that maximizes the final conditional payoff function. This individual action is part of the optimal joint action and the corresponding value equals the desired value $\max_{\mathbf{a}} u(\mathbf{a})$. A second pass in the reverse order is then performed in which every agent computes its optimal action based on its conditional strategy and the fixed actions of its neighbors.

We illustrate VE on the decomposition graphically represented in Fig. 2(a), that is,

$$u(\mathbf{a}) = f_{12}(a_1, a_2) + f_{13}(a_1, a_3) + f_{34}(a_3, a_4), \tag{3}$$

We first eliminate agent 1. This agent does not depend on the local payoff function f_{34} and therefore the maximization of $u(\mathbf{a})$ in (3) can be written as

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_2, a_3, a_4} \Big\{ f_{34}(a_3, a_4) + \max_{a_1} [f_{12}(a_1, a_2) + f_{13}(a_1, a_3)] \Big\}. \tag{4}$$



(a) Initial graph. (b) After elimination of agent 1.

Figure 2: CG corresponding to the decomposition (3) before and after eliminating agent 1.

Agent 1 computes a conditional payoff function $\phi_{23}(a_2, a_3) = \max_{a_1}[f_{12}(a_1, a_2) + f_{13}(a_1, a_3)]$ and the best-response function $B_1(a_2, a_3) = \arg\max_{a_1}[f_{12}(a_1, a_2) + f_{13}(a_1, a_3)]$ which respectively return the maximal value and the associated best action agent 1 is able to perform given the actions of agent 2 and 3. Since the function $\phi_{23}(a_2, a_3)$ is independent of agent 1, it is now eliminated from the graph, simplifying (4) to $\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_2,a_3,a_4}[f_{34}(a_3, a_4) + \phi_{23}(a_2, a_3)]$. The elimination of agent 1 induces a new dependency between agent 2 and 3 and thus a change in the graph's topology. This is depicted in Fig. 2(b). We then eliminate agent 2. Only ϕ_{23} depends on agent 2, so we define $B_2(a_3) = \arg\max_{a_2} \phi_{23}(a_2, a_3)$ and replace ϕ_{23} by $\phi_3(a_3) = \max_{a_2} \phi_{23}(a_2, a_3)$ producing

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_3,a_4}[f_{34}(a_3, a_4) + \phi_3(a_3)], \tag{5}$$

which is independent of a_2. Next, we eliminate agent 3 and replace the functions f_{34} and ϕ_3 resulting in $\max_{\mathbf{a}} u(\mathbf{a}) = \max_{a_4} \phi_4(a_4)$ with $\phi_4(a_4) = \max_{a_3}[f_{34}(a_3, a_4) + \phi_3(a_3)]$. Agent 4 is the last remaining agent and fixes its optimal action $a_4^* = \arg\max_{a_4} \phi_4(a_4)$. A second pass in the reverse elimination order is performed in which each agent computes its optimal (unconditional) action from its best-response function and the fixed actions from its neighbors. In our example, agent 3 first selects $a_3^* = B_3(a_4^*)$. Similarly, we get $a_2^* = B_2(a_3^*)$ and $a_1^* = B_1(a_2^*, a_3^*)$. When an agent has more than one maximizing best-response action, it selects one randomly, since it always communicates its choice to its neighbors. The described procedure holds for the case of a truly distributed implementation using communication. When communication is restricted, additional common knowledge assumptions are needed such that each agent is able to run a copy of the algorithm (Vlassis, 2003, ch. 4).

The VE algorithm always produces the optimal joint action and does not depend on the elimination order. The execution time of the algorithm, however, does. Computing the optimal order is known to be NP-complete, but good heuristics exist, for example, first eliminating the agent with the minimum number of neighbors (Bertelé and Brioschi, 1972). The execution time is exponential in the induced width of the graph (the size of the largest clique computed during node elimination). For densely connected graphs this can scale exponentially in n. Furthermore, VE will only produce its final result after the end of the second pass. This is not always appropriate for real-time multiagent systems where decision making must be done under time constraints. In these cases, an anytime algorithm that improves the quality of the solution over time is more appropriate (Vlassis et al., 2004).

3. Payoff Propagation and the Max-Plus Algorithm[1]

Although the variable elimination (VE) algorithm is exact, it does not scale well with densely connected graphs. In this section, we introduce the *max-plus algorithm* as an approximate alternative to VE and compare the two approaches on randomly generated graphs.

3.1 The Max-Plus Algorithm

The max-product algorithm (Pearl, 1988; Yedidia et al., 2003; Wainwright et al., 2004) is a popular method for computing the *maximum a posteriori* (MAP) configuration in an (unnormalized) undirected graphical model. This method is analogous to the belief propagation or sum-product algorithm (Kschischang et al., 2001). It operates by iteratively sending locally optimized messages $\mu_{ij}(a_j)$ between node i and j over the corresponding edge in the graph. For tree-structured graphs, the message updates converge to a fixed point after a finite number of iterations (Pearl, 1988). After convergence, each node then computes the MAP assignment based on its local incoming messages only.

There is a direct duality between computing the MAP configuration in a probabilistic graphical model and finding the optimal joint action in a CG; in both cases we are optimizing over a function that is decomposed in local terms. This allows message-passing algorithms that have been developed for inference in probabilistic graphical models, to be directly applicable for action selection in CGs. Max-plus is a popular method of that family. In the context of CG, it can therefore be regarded as a 'payoff propagation' technique for multiagent decision making.

Suppose that we have a coordination graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges. In order to compute the optimal joint action $\mathbf{a}^*$ that maximizes (2), each agent i (node in G) repeatedly sends a message μ_{ij} to its neighbors $j \in \Gamma(i)$. The message μ_{ij} can be regarded as a local payoff function of agent j and is defined as

$$\mu_{ij}(a_j) = \max_{a_i} \left\{ f_i(a_i) + f_{ij}(a_i, a_j) + \sum_{k \in \Gamma(i) \setminus j} \mu_{ki}(a_i) \right\} + c_{ij}, \tag{6}$$

where $\Gamma(i) \setminus j$ represents all neighbors of agent i except agent j, and c_{ij} is a normalization value (which can be assumed zero for now). This message is an approximation of the maximum payoff agent i is able to achieve for a given action of agent j, and is computed by maximizing (over the actions of agent i) the sum of the payoff functions f_i and f_{ij} and all incoming messages to agent i except that from agent j. Note that this message only depends on the payoff relations between agent i and agent j and the incoming message to agent i. Messages are exchanged until they converge to a fixed point, or until some external signal is received. Fig. 3 shows a CG with four agents and the corresponding messages.

A message μ_{ij} in the max-plus algorithm has three important differences with respect to the conditional payoff functions in VE. First, before convergence each message is an approximation of the exact value (conditional team payoff) since it depends on the incoming (still not converged) messages. Second, an agent i only has to sum over the received messages from its neighbors which are defined over individual actions, instead of enumerating over all

[1] Section 3 is largely based on (Kok and Vlassis, 2005).

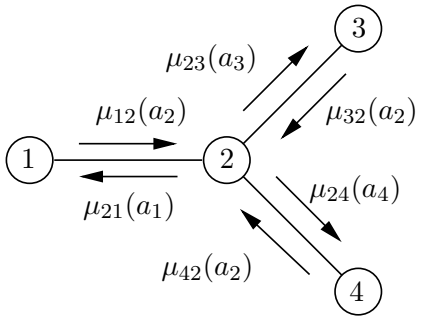


Figure 3: Graphical representation of different messages μ_{ij} in a graph with four agents.

possible action combinations of its neighbors. This is the main reason for the scalability of the algorithm. Finally, in the max-plus algorithm, messages are always sent over the edges of the original graph. In the VE algorithm, the elimination of an agent often results in new dependencies between agents that did not have to coordinate initially.

For trees the messages converge to a fixed point within a finite number of steps (Pearl, 1988; Wainwright et al., 2004). Since a message $\mu_{ji}(a_i)$ equals the payoff produced by the subtree with agent j as root when agent i performs action a_i, we can at any time step define

$$g_i(a_i) = f_i(a_i) + \sum_{j \in \Gamma(i)} \mu_{ji}(a_i), \tag{7}$$

which equals the contribution of the individual function of agent i and the different subtrees with the neighbors of agent i as root. Using (7), we can show that, at convergence, $g_i(a_i) = \max_{\{\mathbf{a}' | a'_i = a_i\}} u(\mathbf{a}')$ holds. Each agent i can then individually select its optimal action

$$a_i^* = \arg\max_{a_i} g_i(a_i). \tag{8}$$

If there is only one maximizing action for every agent i, the globally optimal joint action $\mathbf{a}^* = \arg\max_{\mathbf{a}} u(\mathbf{a})$ is unique and has elements $\mathbf{a}^* = (a_i^*)$. Note that this optimal joint action is computed by only local optimizations (each node maximizes $g_i(a_i)$ separately). In case the local maximizers are not unique, an optimal joint action can be computed by a dynamic programming technique (Wainwright et al., 2004, sec. 3.1). In this case, each agent informs its neighbors in a predefined order about its action choice such that the other agents are able to fix their actions accordingly.

Unfortunately there are no guarantees that max-plus converges in graphs with cycles and therefore no assurances can be given about the quality of the corresponding joint action $\mathbf{a}^* = (a_i^*)$ with a_i from (8) in such settings. Nevertheless, it has been shown that a fixed point of message passing exists (Wainwright et al., 2004), but there is no algorithm yet that provably converges to such a solution. However, bounds are available that characterize the quality of the solution if the algorithm converges (Wainwright et al., 2004). Regardless of these results, the algorithm has been successfully applied in practice in graphs with cycles (Murphy et al., 1999; Crick and Pfeffer, 2003; Yedidia et al., 2003). One of the main problems is that an outgoing message from agent i which is part of a cycle eventually becomes part of its incoming messages. As a result the values of the messages grow extremely large.

centralized max-plus algorithm for $CG = (V, E)$

initialize $\mu_{ij} = \mu_{ji} = 0$ for $(i, j) \in E$, $g_i = 0$ for $i \in V$ and $m = -\infty$
while *fixed_point* = *false* and deadline to send action has not yet arrived **do**
  // run one iteration
  fixed_point = *true*
  for every agent i **do**
    for all neighbors $j = \Gamma(i)$ **do**
      send j message $\mu_{ij}(a_j) = \max_{a_i} \left\{ f_i(a_i) + f_{ij}(a_i, a_j) + \sum_{k \in \Gamma(i) \setminus j} \mu_{ki}(a_i) \right\} + c_{ij}$
      if $\mu_{ij}(a_j)$ differs from previous message by a small threshold **then**
        fixed_point = *false*
    determine $g_i(a_i) = f_i(a_i) + \sum_{j \in \Gamma(i)} \mu_{ji}(a_i)$ and $a'_i = \arg\max_{a_i} g_i(a_i)$
  if use anytime extension **then**
    if $u((a'_i)) > m$ **then**
      $(a^*_i) = (a'_i)$ and $m = u((a'_i))$
  else
    $(a^*_i) = (a'_i)$
return (a^*_i)

Algorithm 1: Pseudo-code of the centralized max-plus algorithm.

Therefore, as in (Wainwright et al., 2004), we normalize each sent message by subtracting the average of all values in μ_{ik} using $c_{ij} = \frac{1}{|A_k|} \sum_k \mu_{ik}(a_k)$ in (6). Still, the joint action might change constantly when the messages keep fluctuating. This necessitates the development of an extension of the algorithm in which each (local) action is only updated when the corresponding global payoff improves. Therefore, we extend the max-plus algorithm by occasionally computing the global payoff and only update the joint action when it improves upon the best value found so far. The best joint action then equals the last updated joint action. We refer to this approach as the *anytime* max-plus algorithm. [2]

The max-plus algorithm can be implemented in either a centralized or a distributed version. The centralized version operates using iterations. In one iteration each agent i computes and sends a message μ_{ij} to all its neighbors $j \in \Gamma(i)$ in a predefined order. This process continues until all messages are converged, or a 'deadline' signal (either from an external source or from an internal timing signal) is received and the current joint action is reported. For the anytime extension, we insert the current computed joint action into (2) after every iteration and only update the joint action when it improves upon the best value found so far. A pseudo-code implementation of the centralized max-plus algorithm, including the anytime extension, is given in Alg. 1.

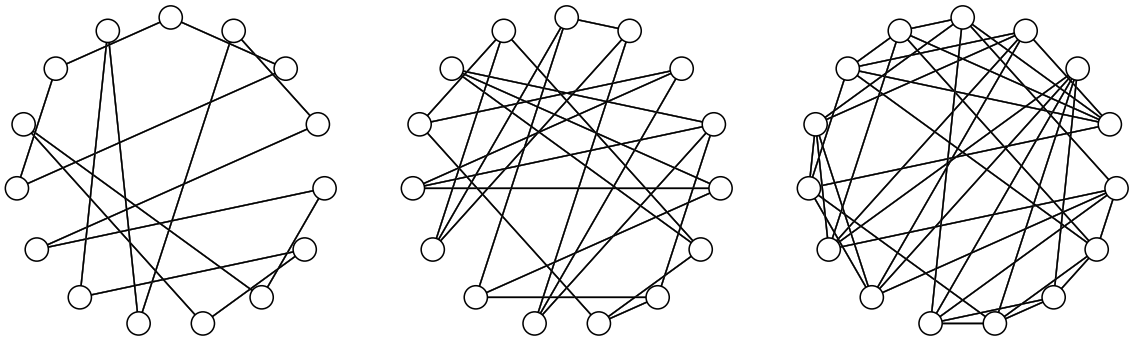
The same functionality can also be implemented using a distributed implementation. Now, each agent computes and communicates an updated message directly after it has received a new (and different) message from one of its neighbors. This results in a computational advantage over the sequential execution of the centralized algorithm since messages are now sent in parallel. We additionally assume that after a finite number of steps, the agents receive a 'deadline' signal after which they report their individual actions.

[2] An alternative, and perhaps more accurate, term is 'max-plus with memory'. However, we decided on the term 'anytime' for reasons of consistency with other publications (Kok and Vlassis, 2005; Kok, 2006)

`distributed max-plus for agent` i, $CG = (V, E)$, `spanning tree` $ST = (V, S)$
initialize $\mu_{ij} = \mu_{ji} = 0$ for $j \in \Gamma(i)$, $g_i = 0$, $p_i = 0$ and $m = -\infty$
while deadline to send action has not yet arrived **do**
  wait for message msg
  if $msg = \mu_{ji}(a_i)$ // max-plus message **then**
    for all neighbors $j \in \Gamma(i)$ **do**
      compute $\mu_{ij}(a_j) = \max_{a_i} \left\{ f_i(a_i) + f_{ij}(a_i, a_j) + \sum_{k \in \Gamma(i) \setminus j} \mu_{ki}(a_i) \right\} + c_{ij}$
      send message $\mu_{ij}(a_j)$ to agent j if it differs from last sent message
    if use anytime extension **then**
      if heuristic indicates global payoff should be evaluated **then**
        send `evaluate`(i) to agent i // initiate computation global payoff
    else
      $a_i^* = \arg\max_{a_i} [f_i(a_i) + \sum_{j \in \Gamma(i)} \mu_{ji}(a_i)]$
  if $msg =$ `evaluate`(j) // receive request for evaluation from agent j **then**
    if a_i' not locked, lock $a_i' = \arg\max_{a_i} [f_i(a_i) + \sum_{j \in \Gamma(i)} \mu_{ji}(a_i)]$ and set $p_i = 0$
    send `evaluate`(i) to all neighbors (parent and children) in $ST \neq j$
    if $i =$ leaf in ST **then**
      send `accumulate_payoff`(0) to agent i // initiate accumulation payoffs
  if $msg =$ `accumulate_payoff`(p_j) from agent j **then**
    $p_i = p_i + p_j$ // add payoff child j
    if received accumulated payoff from all children in ST **then**
      get actions a_j' from $j \in \Gamma(i)$ in CG and set $g_i = f_i(a_i') + \frac{1}{2} \sum_{j \in \Gamma(i)} f_{ij}(a_i', a_j')$
      if $i =$ root of ST **then**
        send `global_payoff`($g_i + p_i$) to agent i
      else
        send `accumulate_payoff`($g_i + p_i$) to parent in ST
  if $msg =$ `global_payoff`(g) **then**
    if $g > m$ **then**
      $a_i^* = a_i'$ and $m = g$
    send `global_payoff`(g) to all children in ST and unlock action a_i'
return a_i^*

Algorithm 2: Pseudo-code of a distributed max-plus implementation.

For the distributed case, the implementation of the anytime extension is much more complex since the agents do not have direct access to the actions of the other agents or the global payoff function (2). Therefore, the evaluation of the (distributed) joint action is only initiated by an agent when it believes it is worthwhile to do so, for example, after a big increase in the values of the received messages. This agent starts the propagation of an 'evaluation' message over a spanning tree ST. A spanning tree is a tree-structured subgraph of G that includes all nodes. This tree is fixed beforehand and is common knowledge among all agents. An agent receiving an evaluation message fixes its individual action until after the evaluation. When an agent is a leaf of ST it also computes its local contribution to the global payoff and sends this value to its parent in ST. Each parent accumulates all payoffs of its children and after adding its own contribution sends the result to its parent. Finally,



(a) Graph with 15 edges (average degree of 2).

(b) Graph with 23 edges (average degree of 3.07).

(c) Graph with 37 edges (average degree of 4.93).

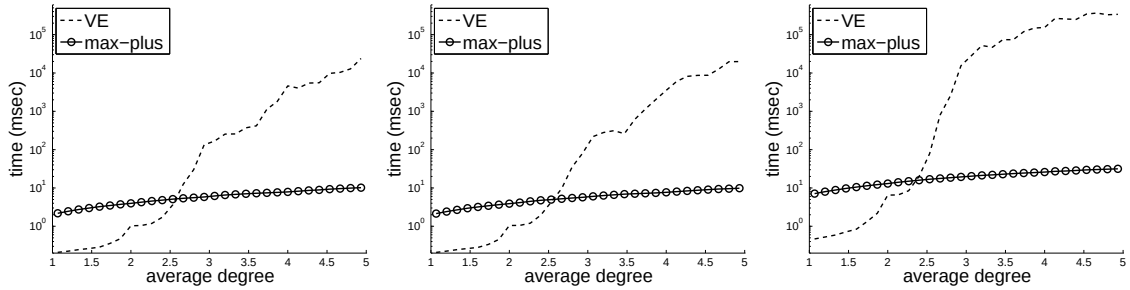
Figure 4: Example graphs with 15 agents and cycles.

when the root of ST has received all accumulated payoffs from its children, the sum of these payoffs (global payoff) is distributed to all nodes in ST. The agents only update their best individual action a_i^* when this payoff improves upon the best one found so far. When the 'deadline' signal arrives, each agent reports the action related to the highest found global payoff, which might not correspond to the current messages. Alg. 2 shows a distributed version in pseudo-code.

3.2 Experiments

In this section, we describe our experiments with the max-plus algorithm on differently shaped graphs. For cycle-free graphs max-plus is equivalent to VE when the messages in the first iteration are sent in the same sequence as the elimination order of VE and in the reverse order for the second iteration (comparable to the reversed pass in VE). Therefore, we only test max-plus on graphs with cycles.

We ran the algorithms on differently shaped graphs with 15 agents and a varying number of edges. In order to generate balanced graphs in which each agent approximately has the same degree, we start with a graph without edges and iteratively connect the two agents with the minimum number of neighbors. In case multiple agents satisfy this condition, an agent is picked at random from the possibilities. We apply this procedure to create 100 graphs for each $|E| \in \{8, 9, \ldots, 37\}$, resulting in a set of $3,000$ graphs. The set thus contains graphs in the range of on average 1.067 neighbors per agent (8 edges) to 4.93 neighbors per agent (37 edges). Fig. 10 depicts example graphs with respectively 15, 23 and 37 edges (on average 2, 3.07 and 4.93 neighbors per node). We create three copies of this set, each having a different payoff function related to the edges in the graph. In the first set, each edge $(i, j) \in E$ is associated with a payoff function f_{ij} defined over five actions per agent and each action combination is assigned a random payoff from a standard normal distribution, that is, $f_{ij}(a_i, a_j) \sim \mathcal{N}(0, 1)$. This results in a total of 5^{15}, around 3 billion, different possible joint actions. In the second set, we add one outlier to each of the local payoff functions: for a randomly picked joint action, the corresponding payoff value is set to $10 \cdot \mathcal{N}(0, 1)$. For the third test set, we specify a payoff function based on 10 actions per



(a) 5 actions per agent. (b) 5 actions and outliers. (c) 10 actions per agent.

Figure 5: Timing results for VE and max-plus for different graphs with 15 agents and cycles.

agent resulting in 10^{15} different joint actions. The values of the different payoff functions are again generated using a standard normal distribution.

For all graphs we compute the joint action using the VE algorithm, the standard max-plus algorithm, and the max-plus algorithm with the anytime extension. Irrespectively of convergence, all max-plus methods perform 100 iterations. As we will see later in Fig. 6 the policy has stabilized at this point. Furthermore, a random ordering is used in each iteration to determine which agents sends its messages.

The timing results for the three different test sets are plotted in Fig. 5.[3] The x-axis shows the average degree of the graph, and the y-axis shows, using a logarithmic scale, the average timing results, in milliseconds, to compute the joint action for the corresponding graphs. Remember from Section 2 that the computation time of the VE algorithm depends on the induced width of the graph. The induced width depends both on the average degree and the actual structure of the graph. The latter is generated at random, and therefore the complexity of graphs with the same average degree differ. Table 1 shows the induced width for the graphs used in the experiments based on the elimination order of the VE algorithm, that is, iteratively remove a node with the minimum number of neighbors. The results are averaged over graphs with a similar average degree. For a specific graph, the induced width equals the maximal number of neighbors that have to be considered in a local maximization.

In Fig. 5, we show the timing results for the standard max-plus algorithm; the results for the anytime extension are identical since they only involve an additional check of the global payoff value after every iteration. The plots indicate that the time for the max-plus algorithm grows linearly as the complexity of the graphs increases. This is a result of the relation between the number of messages and the (linearly increasing) number of edges in the graph. The graphs with 10 actions per agent require more time compared to the two other sets because the computation of every message involves a maximization over 100 instead of 25 joint actions. Note that all timing results are generated with a fixed number of 100 iterations. As we will see later, the max-plus algorithm can be stopped earlier without much loss in performance, resulting in even quicker timing results.

For the graphs with a small, less than 2.5, average degree, VE outperforms the max-plus algorithm. In this case, each local maximization only involves a few agents, and VE

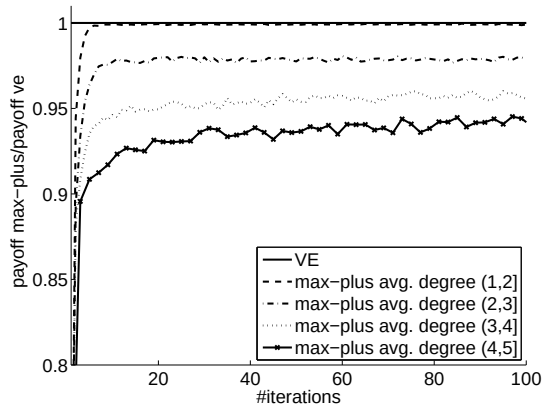
[3] All results are generated on an Intel Xeon 3.4GHz / 2GB machine using a C++ implementation.

average degree	$(1, 2]$	$(2, 3]$	$(3, 4]$	$(4, 5]$
induced width	1.23 ($\pm$0.44)	2.99 ($\pm$0.81)	4.94 ($\pm$0.77)	6.37 ($\pm$0.68)

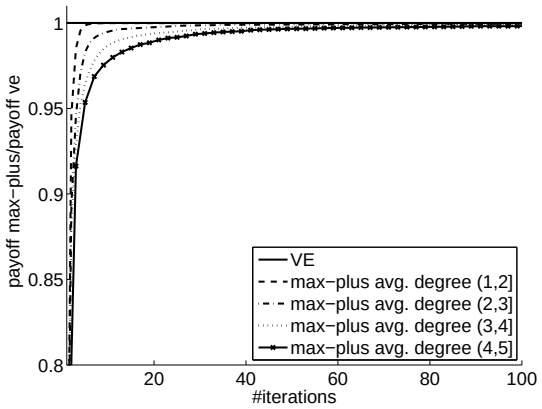
Table 1: Average induced width and corresponding standard deviation for graphs with an average degree in $(x - 1, x]$.

is able to finish its two passes through the graph quickly. However, the time for the VE algorithm grows exponentially for graphs with a higher average degree because for these graphs it has to enumerate over an increasing number of neighboring agents in each local maximization step. Furthermore, the elimination of an agent often causes a neighboring agent to receive a conditional strategy involving agents it did not have to coordinate with before, changing the graph topology to an even denser graph. This effect becomes more apparent as the graphs become more dense. More specifically, for graphs with 5 actions per agent and an average degree of 5, it takes VE on average 23.8 seconds to generate the joint action. The max-plus algorithm, on the other hand, only requires 10.18 milliseconds for such graphs. There are no clear differences between the two sets with 5 actions per agent since they both require the same number of local maximizations, and the actual values do not influence the algorithm. However, as is seen in Fig. 5(c), the increase of the number of actions per agent slows the VE algorithm down even more. This is a result of the larger number of joint actions which has to be processed during the local maximizations. For example, during a local maximization of an agent with five neighbors $5^5 = 3,125$ actions have to be enumerated in the case of 5 actions per agent. With 10 actions per agent, this number increases to $10^5 = 100,000$ actions. During elimination the topology of the graph can change to very dense graphs resulting in even larger maximizations. This is also evident from the experiments. For some graphs with ten actions per agent and an average degree higher than 3.2, the size of the intermediate tables grows too large for the available memory, and VE is not able to produce a result. These graphs are removed from the set. For the graphs with an average degree between 3 and 4, this results in the removal of 81 graphs. With an increase of the average degree, this effect becomes more apparent: VE is not able to produce a result for 466 out of the 700 graphs with an average degree higher than 4; all these graphs are removed from the set. This also explains why the increase in the curve of VE in Fig. 5(c) decreases: the more difficult graphs, which take longer to complete, are not taken into account. Even without these graphs, it takes VE on average 339.76 seconds, almost 6 minutes, to produce a joint action for the graphs with an average degree of 5. The max-plus algorithm, on the other hand, needs on average 31.61 milliseconds.

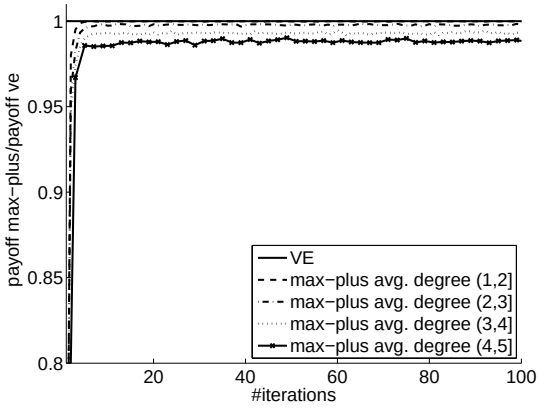
The max-plus algorithm thus outperforms VE with respect to the computation time for densely connected graphs. But how do the resulting joint actions of the max-plus algorithm compare to the optimal solutions of the VE algorithm? Fig. 6 shows the payoff found with the max-plus algorithm relative to the optimal payoff, after each iteration. A relative payoff of 1 indicates that the found joint action corresponds to the optimal joint action, while a relative payoff of 0 indicates that it corresponds to the joint action with the minimal possible payoff. Each of the four displayed curves corresponds to the average result of a



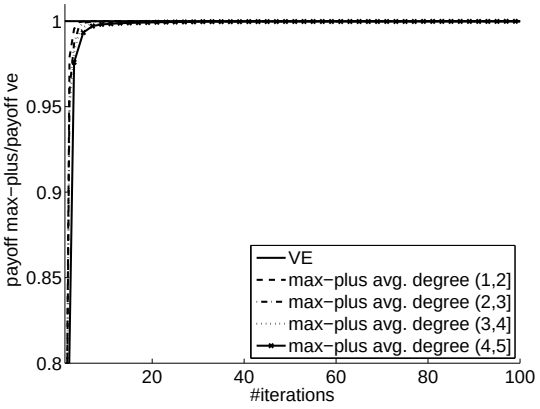
(a) Max-plus (5 actions per agent).



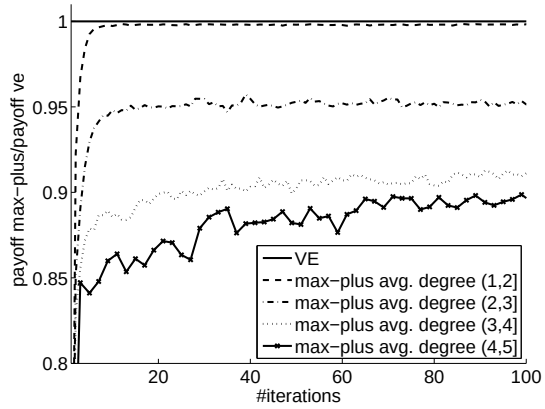
(b) Anytime max-plus (5 actions).



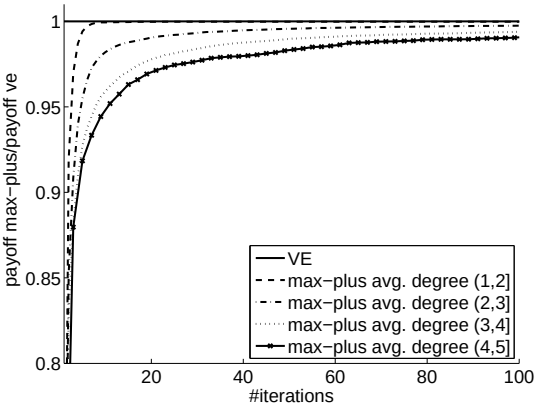
(c) Max-plus (5 actions per agent and outliers)



(d) Anytime max-plus (5 actions and outliers).



(e) Max-plus (10 actions per agent).



(f) Anytime max-plus (10 actions).

Figure 6: Relative payoff compared to VE for both standard max-plus (graphs on the left) and anytime max-plus (graphs on the right) for graphs with 15 agents and cycles.

subset with a similar average degree. Specifically, each subset contains all graphs with an average degree in $(x-1, x]$, with $x \in \{2, 3, 4, 5\}$.

We first discuss the result of the standard max-plus algorithm in the graphs on the left. For all three sets, the loosely connected graphs with an average degree less than two converge to a similar policy as the optimal joint action in a few iterations only. As the average degree increases, the resulting policy declines. As seen in Fig. 6(c), this effect is less evident in the graphs with outliers; the action combinations related to the positive outliers are clearly preferred, and lowers the number of oscillations. Increasing the number of actions per agent has a negative influence on the result, as is evident from Fig. 6(e), because the total number of action combinations increases. The displayed results are an average of a large set of problems, and an individual run typically contains large oscillations between good and bad solutions.

When using the anytime version, which returns the best joint action found so far, the obtained payoff improves for all graphs. This indicates that the failing convergence of the messages causes the standard max-plus algorithm to oscillate between different joint actions and 'forget' good joint actions. Fig. 6 shows that for all sets near-optimal policies are found, although more complex graphs need more iterations to find them.

4. Collaborative Multiagent Reinforcement Learning

Until now, we have been discussing the problem of selecting an optimal joint action in a group of agents for a given payoff structure and a single state only. Next, we consider *sequential decision-making problems*. In such problems, the agents select a joint action which provides them a reward and causes a transition to a new state. The goal of the agents is to select actions that optimize a performance measure based on the received rewards. This might involve a *sequence* of decisions. An important aspect of this problem is that the agents have no prior knowledge about the effect of their actions, but that this information has to be *learned* based on the, possibly delayed, rewards. Next, we review a model to represent such a problem and describe several solution techniques.

4.1 Collaborative Multiagent MDP and Q-Learning

Different models exist to describe a group of agents interacting with their environment. We will use the collaborative multiagent MDP framework (Guestrin, 2003) which is an extension of the single-agent Markov decision process (MDP) framework (Puterman, 1994). It consists of the following model parameters:

- A time step $t = 0, 1, 2, 3, \ldots$.
- A group of n agents $A = \{A_1, A_2, ..., A_n\}$.
- A set of discrete state variables S_i. The global state is the cross-product of all m variables: $S = S_1 \times ... \times S_m$. A state $\mathbf{s}^t \in S$ describes the state of the world at time t.
- A finite set of actions $\mathcal{A}_i$ for every agent i. The action selected by agent i at time step t is denoted by $a_i^t \in \mathcal{A}_i$. The joint action $\mathbf{a}^t \in \mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$ is the combination of all individual actions of the n agents.

- A state transition function $T : S \times \mathcal{A} \times S \rightarrow [0, 1]$ which gives the transition probability $p(\mathbf{s}^{t+1}|\mathbf{s}^t, \mathbf{a}^t)$ that the system will move to state $\mathbf{s}^{t+1}$ when the joint action $\mathbf{a}^t$ is performed in state $\mathbf{s}^t$.
- A reward function $R_i : S \times \mathcal{A} \rightarrow \mathbb{R}$ which provides agent i with an individual reward $r_i^t \in R_i(\mathbf{s}^t, \mathbf{a}^t)$ based on the joint action $\mathbf{a}^t$ taken in state $\mathbf{s}^t$. The global reward is the sum of all local rewards: $R(\mathbf{s}^t, \mathbf{a}^t) = \sum_{i=1}^{n} R_i(\mathbf{s}^t, \mathbf{a}^t)$.

This model assumes that the Markov property holds which denotes that the state description at time t provides a complete description of the history before time t. This is apparent in both the transition and reward function in which all information before time t is ignored. Furthermore, it also assumes that the environment is stationary, that is, the reward and transition probabilities are independent of the time step t. Since the transition function is stationary, we will in most cases omit the time step t superscript when referring to a state $\mathbf{s}^t$, and use the shorthand $\mathbf{s}'$ for the next state $\mathbf{s}^{t+1}$.

A policy $\pi : \mathbf{s} \rightarrow \mathbf{a}$ is a function which returns an action $\mathbf{a}$ for any given state $\mathbf{s}$. The objective is to find an optimal policy π^* that maximizes the expected discounted future return $V^*(\mathbf{s}) = \max_\pi E\left[\sum_{t=0}^{\infty} \gamma^t R(\mathbf{s}^t, \pi(\mathbf{s}^t))|\pi, \mathbf{s}^0 = \mathbf{s}\right]$ for each state $\mathbf{s}$. The expectation operator $E[\cdot]$ averages over stochastic transitions, and $\gamma \in [0, 1)$ is the discount factor. Rewards in the near future are thus preferred over rewards in the distant future. The return is defined in terms of the sum of individual rewards, and the agents thus have to cooperate in order to achieve their common goal. This differs from self-interested approaches (Shapley, 1953; Hansen et al., 2004) in which each agent tries to maximize its own payoff.

Q-functions, or action-value functions, represent the expected future discounted reward for a state $\mathbf{s}$ when selecting a specific action $\mathbf{a}$ and behaving optimally from then on. The optimal Q-function Q^* satisfies the Bellman equation:

$$Q^*(\mathbf{s}, \mathbf{a}) = R(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}'} p(\mathbf{s}'|\mathbf{s}, \mathbf{a}) \max_{\mathbf{a}'} Q^*(\mathbf{s}', \mathbf{a}'). \tag{9}$$

Given Q^*, the optimal policy for the agents in state $\mathbf{s}$ is to jointly select the action $\arg\max_{\mathbf{a}} Q^*(\mathbf{s}, \mathbf{a})$ that maximizes the expected future discounted return.

Reinforcement learning (RL) (Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1996) can be applied to estimate $Q^*(\mathbf{s}, \mathbf{a})$. Q-learning is a widely used learning method for single-agent systems when the agent does not have access to the transition and reward model. The agent interacts with the environment by selecting actions and receives (s, a, r, s') samples based on the experienced state transitions. Q-learning starts with an initial estimate $Q(s, a)$ for each state-action pair. At each time step the agent selects an action based on an exploration strategy. A commonly used strategy is ϵ-greedy which selects the greedy action, $\arg\max_a Q(s, a)$, with high probability, and, occasionally, with a small probability ϵ selects an action uniformly at random. This ensures that all actions, and their effects, are experienced. Each time an action a is taken in state s, reward $R(s, a)$ is received, and next state s' is observed, the corresponding Q-value is updated with a combination of its current value and the temporal-difference error, the difference between its current estimate $Q(s, a)$ and the experienced sample $R(s, a) + \gamma \max_{a'} Q(s', a')$, using

$$Q(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)] \tag{10}$$

where $\alpha \in (0, 1)$ is an appropriate learning rate which controls the contribution of the new experience to the current estimate. When every state-action pair is associated with a unique Q-value and every action is sampled infinitely often (as with the ϵ-greedy action selection method), iteratively applying (10) is known to converge to the optimal $Q^*(s, a)$ values (Watkins and Dayan, 1992).

Next, we describe four multiagent variants of tabular Q-learning to multiagent environments, and discuss their advantages and disadvantages. We do not consider any function-approximation algorithms. Although they have been been successfully applied in several domains with large state sets, they are less applicable for large action sets since it is more difficult to generalize over nearby (joint) actions. Furthermore, we only consider model-free methods in which the agents do not have access to the transition and reward function. The agents do observe the current state and also receive an individual reward depending on the performed joint action and the unknown reward function. Finally, we assume the agents are allowed to communicate in order to coordinate their actions.

4.2 MDP Learners

In principle, a collaborative multiagent MDP can be regarded as one large single agent in which each joint action is represented as a single action. It is then possible to learn the optimal Q-values for the joint actions using standard single-agent Q-learning, that is, by iteratively applying (10). In this *MDP learners* approach either a central controller models the complete MDP and communicates to each agent its individual action, or each agent models the complete MDP separately and selects the individual action that corresponds to its own identity. In the latter case, the agents do not need to communicate but they have to be able to observe the executed joint action and the received individual rewards. The problem of exploration is solved by using the same random number generator (and the same seed) for all agents (Vlassis, 2003).

Although this approach leads to the optimal solution, it is infeasible for problems with many agents. In the first place, it is intractable to model the complete joint action space, which is exponential in the number of agents. For example, a problem with 7 agents, each able to perform 6 actions, results in almost $280,000$ Q-values per state. Secondly, the agents might not have access to the needed information for the update because they are not able to observe the state, action, and reward of all other agents. Finally, it will take many time steps to explore all joint actions resulting in slow convergence.

4.3 Independent Learners

At the other extreme, we have the *independent learners* (IL) approach (Claus and Boutilier, 1998) in which the agents ignore the actions and rewards of the other agents, and learn their strategies independently. Each agent stores and updates an individual table Q_i and the global Q-function is defined as a linear combination of all individual contributions, $Q(\mathbf{s}, \mathbf{a}) = \sum_{i=1}^{n} Q_i(\mathbf{s}, a_i)$. Each local Q-function is updated using

$$Q_i(\mathbf{s}, a_i) := Q_i(\mathbf{s}, a_i) + \alpha[R_i(\mathbf{s}, \mathbf{a}) + \gamma \max_{a_i'} Q_i(\mathbf{s}', a_i') - Q_i(\mathbf{s}, a_i)]. \tag{11}$$

Note that each Q_i is based on the global state $\mathbf{s}$. This approach results in big storage and computational savings in the action-space, for example, with 7 agents and 6 actions per

agent only 42 Q-values have to be stored per state. However, the standard convergence proof for Q-learning does not hold anymore. Because the actions of the other agents are ignored in the representation of the Q-functions, and these agents also change their behavior while learning, the system becomes non-stationary from the perspective of an individual agent. This might lead to oscillations. Despite the lack of guaranteed convergence, this method has been applied successfully in multiple cases (Tan, 1993; Sen et al., 1994).

4.4 Coordinated Reinforcement Learning

In many situations an agent has to coordinate its actions with a few agents only, and acts independently with respect to the other agents. In Guestrin et al. (2002b) three different *Coordinated Reinforcement Learning* approaches are described which take advantage of the structure of the problem. The three approaches are respectively a variant of Q-learning, policy iteration, and direct policy search. We will concentrate on the Q-learning variant which decomposes the global Q-function into a linear combination of local agent-dependent Q-functions. Each local Q_i is based on a subset of all state and action variables,

$$Q(\mathbf{s}, \mathbf{a}) = \sum_{i=1}^{n} Q_i(\mathbf{s}_i, \mathbf{a}_i), \tag{12}$$

where $\mathbf{s}_i$ and $\mathbf{a}_i$ are respectively the subset of state and action variables related to agent i. These dependencies are established beforehand and differ per problem. Note that in this representation, each agent only needs to observe the state variables $\mathbf{s}_i$ which are part of its local Q_i-function. The corresponding CG is constructed by adding an edge between agent i and j when the action of agent j is included in the action variables of agent i, that is, $a_j \in \mathbf{a}_i$. As an example, imagine a computer network in which each machine is modeled as an agent and only depends on the state and action variables of itself and the machines it is connected to. The coordination graph would in this case equal the network topology.

A local Q_i is updated using the global temporal-difference error, the difference between the current global Q-value and the expected future discounted return for the experienced state transition, using

$$Q_i(\mathbf{s}_i, \mathbf{a}_i) := Q_i(\mathbf{s}_i, \mathbf{a}_i) + \alpha[R(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}') - Q(\mathbf{s}, \mathbf{a})]. \tag{13}$$

The global reward $R(\mathbf{s}, \mathbf{a})$ is given. The maximizing action in $\mathbf{s}'$ and the associated maximal expected future return, $\max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}')$, are computed in a distributed manner by applying the VE algorithm discussed in Section 2 on the CG. The estimate of the global Q-value in $\mathbf{s}$, $Q(\mathbf{s}, \mathbf{a})$ in (13), is computed by fixing the action of every agent to the one assigned in $\mathbf{a}$ and applying a message passing scheme similar to the one used in the VE algorithm. We use a table-based representation for the Q-functions in our notation. However, since each individual Q-function is entirely local, each agent is allowed to choose its own representation, for example, using a function approximator as in Guestrin et al. (2002b).

The advantage of this method is that it is completely distributed. Each agent keeps a local Q-function and only has to exchange messages with its neighbors in the graph in order to compute the global Q-values. In sparsely connected graphs, this results in large computational savings since it is not necessary to consider the complete joint action-space.

However, the algorithm is still slow for densely connected graphs because of two main reasons. First, the size of each local Q-function grows exponentially with the number of neighbors of the corresponding agent. Secondly, the computational complexity of the VE algorithm is exponential in the induced width of the graph, as shown in Section 3.2.

4.5 Distributed Value Functions

Another method to decompose a large action space is the distributed value functions (DVF) approach (Schneider et al., 1999). Each agent maintains an individual local Q-function, $Q_i(\mathbf{s}_i, a_i)$, based on its individual action and updates it by incorporating the Q-functions of its neighboring agents. A weight function $f(i, j)$ determines how much the Q-value of an agent j contributes to the update of the Q-value of agent i. This function defines a graph structure of agent dependencies, in which an edge is added between agents i and j if the corresponding function $f(i, j)$ is non-zero. The update looks as follows:

$$Q_i(\mathbf{s}_i, a_i) := (1 - \alpha)Q_i(\mathbf{s}_i, a_i) + \alpha[R_i(\mathbf{s}, \mathbf{a}) + \gamma \sum_{j \in \{i \cup \Gamma(i)\}} f(i, j) \max_{a'_j} Q_j(\mathbf{s}', a'_j)]. \qquad (14)$$

Note that $f(i, i)$ also has to be defined and specifies the agent's contribution to the current estimate. A common approach is to weigh each neighboring function equally, $f(i, j) = 1/|i \cup \Gamma(j)|$. Each Q-function of an agent i is thus divided proportionally over its neighbors and itself. This method scales linearly in the number of agents.

5. Sparse Cooperative Q-Learning

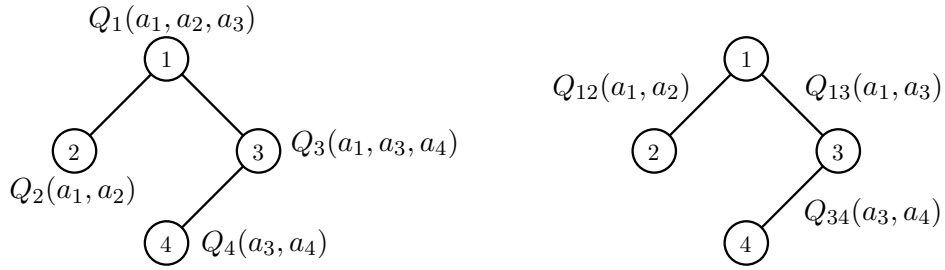
In this section, we describe our Sparse Cooperative Q-learning, or SparseQ, methods which also approximate the global Q-function into a linear combination of local Q-functions. The decomposition is based on the structure of a CG which is chosen beforehand. In principle we can select any arbitrary CG, but in general a CG based on the problem under study is used. For a given CG, we investigate both a decomposition in terms of the nodes (or agents), as well as the edges. In the agent-based decomposition the local function of an agent is based on its own action and those of its neighboring agents. In the edge-based decomposition each local function is based on the actions of the two agents it is connected to. In order to update a local function, the key idea is to base the update not on the difference between the current global Q-value and the experienced global discounted return, but rather on the current local Q-value and the *local contribution* of this agent to the global return.

Next, we describe an agent-based decomposition of the global Q-function and explain how the local contribution of an agent is used in the update step. Thereafter, we describe an edge-based decomposition, and a related edge-based and agent-based update method.

5.1 Agent-Based Decomposition[4]

As in Guestrin et al. (2002b) we decompose the global Q-function over the different agents. Every agent i is associated with a local Q-function $Q_i(\mathbf{s}_i, \mathbf{a}_i)$ which only depends on a subset of all possible state and action variables. These dependencies are specified beforehand and

[4]Subsection 5.1 is based on (Kok and Vlassis, 2004).



(a) Agent-based decomposition.

(b) Edge-based decomposition.

Figure 7: An agent-based and edge-based decomposition of the global Q-function for a 4-agent problem.

depend on the problem. The Q_i-functions correspond to a CG which is constructed by connecting each agent with all agents in which its action variable is involved. See Fig. 7(a) for an example of an agent-based decomposition for a 4-agent problem.

Since the global Q-function equals the sum of the local Q-functions of all n agents, $Q(\mathbf{s}, \mathbf{a}) = \sum_{i=1}^{n} Q_i(\mathbf{s}_i, \mathbf{a}_i)$, it is possible to rewrite the Q-learning update rule in (10) as

$$\sum_{i=1}^{n} Q_i(\mathbf{s}_i, \mathbf{a}_i) := \sum_{i=1}^{n} Q_i(\mathbf{s}_i, \mathbf{a}_i) + \alpha\big[\sum_{i=1}^{n} R_i(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}') - \sum_{i=1}^{n} Q_i(\mathbf{s}_i, \mathbf{a}_i)\big]. \quad (15)$$

Only the expected discounted return, $\max_{a'} Q(\mathbf{s}', \mathbf{a}')$, cannot be directly written as the sum of local terms since it depends on the *globally* maximizing joint action. However, we can use the VE algorithm to compute, in a distributed manner, the maximizing joint action $\mathbf{a}^* = \arg\max_{\mathbf{a}'} Q(\mathbf{s}', \mathbf{a}')$ in state $\mathbf{s}'$, and from this compute the local contribution $Q_i(\mathbf{s}'_i, \mathbf{a}^*_i)$ of each agent to the total action value $Q(\mathbf{s}', \mathbf{a}^*)$. Note that the local contribution of an agent to the global action value might be lower than the maximizing value of its local Q-function because it is unaware of the dependencies of its neighboring agents with the other agents in the CG. Since we can substitute $\max_{a'} Q(\mathbf{s}', \mathbf{a}')$ with $\sum_{i=1}^{n} Q_i(\mathbf{s}', \mathbf{a}^*_i)$, we are able to decompose all terms in (15) and rewrite the update for each agent i separately:

$$Q_i(\mathbf{s}_i, \mathbf{a}_i) := Q_i(\mathbf{s}_i, \mathbf{a}_i) + \alpha[R_i(\mathbf{s}, \mathbf{a}) + \gamma Q_i(\mathbf{s}'_i, \mathbf{a}^*_i) - Q_i(\mathbf{s}_i, \mathbf{a}_i)]. \quad (16)$$

This update is completely based on local terms and only requires the distributed VE algorithm to compute the maximizing joint action $\mathbf{a}^*$. In contrast to CoordRL, we directly take advantage of the local rewards received by the different agents. Especially for larger problems with many agents, this allows us to propagate back the reward to the local functions related to the agents responsible for the generated rewards. This is not possible in CoordRL which uses the global reward to update the different local functions. As a consequence, the agents are not able to distinguish which agents are responsible for the received reward, and all functions, including the ones which are not related to the received reward, are updated equally. It might even be the case that the high reward generated by one agent, or a group of agents, is counterbalanced by the negative reward of another agent. In this case, the combined global reward equals zero and no functions are updated.

Just as the coordinated RL approach, both the representation of the local Q_i-functions and the VE algorithm grow exponentially with the number of neighbors. This becomes problematic for densely connected graphs, and therefore we also investigate an edge-based decomposition of the Q-function which does not suffer from this problem in the next section.

5.2 Edge-Based Decomposition

A different method to decompose the global Q-function is to define it in terms of the edges of the corresponding CG. Contrary to the agent-based decomposition, which scales exponentially with the number of neighbors in the graph, the edge-based decomposition scales linearly in the number of neighbors. For a coordination graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges, each edge $(i, j) \in E$ corresponds to a local Q-function Q_{ij}, and the sum of all local Q-functions defines the global Q-function:

$$Q(\mathbf{s}, \mathbf{a}) = \sum_{(i,j)\in E} Q_{ij}(\mathbf{s}_{ij}, a_i, a_j), \tag{17}$$

where $\mathbf{s}_{ij} \subseteq \mathbf{s}_i \cup \mathbf{s}_j$ is the subset of the state variables related to agent i and agent j which are relevant for their dependency. Note that each local Q-function Q_{ij} always depends on the actions of two agents, a_i and a_j, only. Fig. 7(b) shows an example of an edge-based decomposition for a 4-agent problem.

An important consequence of this decomposition is that it only depends on pairwise functions. This allows us to directly apply the max-plus algorithm from Section 3 to compute the maximizing joint action. Now, both the decomposition of the action-value function and the method for action selection scale linearly in the number of dependencies, resulting in an approach that can be applied to large agent networks with many dependencies.

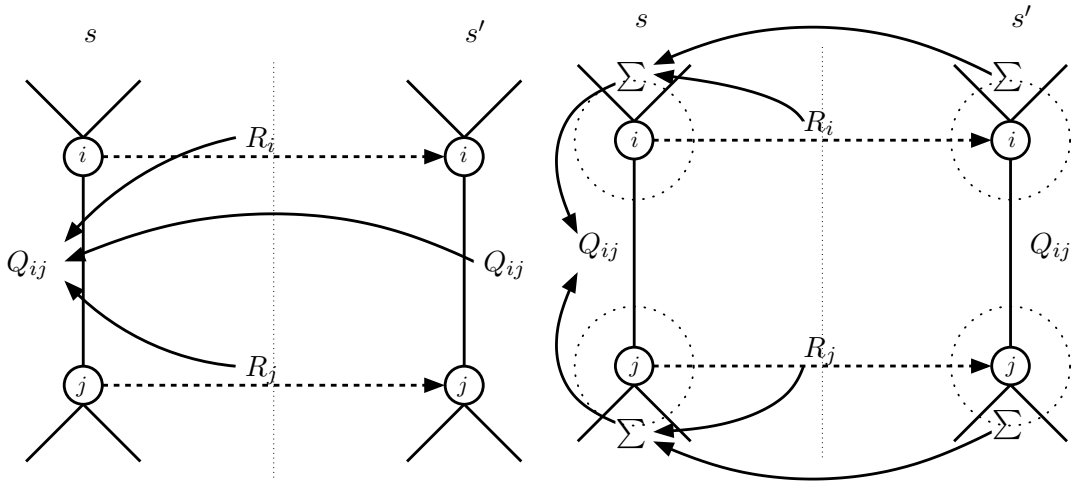
In order to update a local Q-function, we have to propagate back the reward received by the individual agents. This is complicated by the fact that the rewards are received per agent, while the local Q-functions are defined over the edges. For an agent with multiple neighbors it is therefore not possible to derive which dependency generated (parts of) the reward. Our approach is to associate each agent with a local Q-function Q_i that is directly computed from the edge-based Q-functions Q_{ij}. This allows us to relate the received reward of an agent directly to its agent-based Q-function Q_i. In order to compute Q_i, we assume that each edge-based Q-function contributes equally to the two agents that form the edge. Then, the local Q-function Q_i of agent i is defined as the summation of half the value of all local Q-functions Q_{ij} of agent i and its neighbors $j \in \Gamma(i)$, that is,

$$Q_i(\mathbf{s}_i, \mathbf{a}_i) = \frac{1}{2} \sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}_{ij}, a_i, a_j). \tag{18}$$

The sum of all local Q-functions Q_i equals Q in (17). Next, we describe two update methods for the edge-based decomposition defined in terms of these local agent-based Q-functions.

5.2.1 Edge-Based Update

The first update method we consider updates each local Q-function Q_{ij} based on its current estimate and its contribution to the maximal return in the next state. For this, we rewrite



(a) Edge-based update. (b) Agent-based update.

Figure 8: A graphical representation of the edge-based and agent-based update method after the transition from state $\mathbf{s}$ to $\mathbf{s}'$. See the text for a detailed description.

(16) by replacing every instance of Q_i with its definition in (18) to

$$\frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}_{ij},a_i,a_j) := \frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}_{ij},a_i,a_j) + \alpha\left[\sum_{j\in\Gamma(i)}\frac{R_i(\mathbf{s},\mathbf{a})}{|\Gamma(i)|} + \gamma\frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}'_{ij},a_i^*,a_j^*) - \frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}_{ij},a_i,a_j)\right]. \quad (19)$$

Note that in this decomposition for agent i we made the assumption that the reward R_i is divided proportionally over its neighbors $\Gamma(i)$. In order to get an update equation for an individual local Q-function Q_{ij}, we remove the sums. Because, one half of every local Q-function Q_{ij} is updated by agent i and the other half by agent j, agent j updates the local Q-function Q_{ij} using a similar decomposition as (19). Adding the two gives the following update equation for a single local Q-function Q_{ij}:

$$Q_{ij}(\mathbf{s}_{ij},a_i,a_j) := Q_{ij}(\mathbf{s}_{ij},a_i,a_j) + \alpha\left[\frac{R_i(\mathbf{s},\mathbf{a})}{|\Gamma(i)|} + \frac{R_j(\mathbf{s},\mathbf{a})}{|\Gamma(j)|} + \gamma Q_{ij}(\mathbf{s}'_{ij},a_i^*,a_j^*) - Q_{ij}(\mathbf{s}_{ij},a_i,a_j)\right]. \quad (20)$$

Each local Q-function Q_{ij} is updated with a proportional part of the received reward of the two agents it is related to and with the contribution of this edge to the maximizing joint action $\mathbf{a}^* = (a_i^*) = \arg\max_{\mathbf{a}'} Q(\mathbf{s}',\mathbf{a}')$ in the state $\mathbf{s}'$. The latter is computed by either applying the exact VE algorithm or the approximate max-plus algorithm. We can also derive (20) from (10) directly using (17). However, we want to emphasize that it is possible to derive this update rule from the agent-based decomposition discussed in Section 5.1.

Fig. 8(a) shows a graphical representation of the update. The left part of the figure shows a partial view of a CG in state $\mathbf{s}$. Only the agents i and j, their connecting edge,

which is related to a local edge-based Q-function Q_{ij}, and some outgoing edges are depicted. The right part of the figure shows the same structure for state s'. Following (20), a local Q-function Q_{ij} is directly updated based on the received reward of the involved agents and the maximizing local Q-function Q_{ij} in the next state.

5.2.2 Agent-Based Update

In the edge-based update method the reward is divided proportionally over the different edges of an agent. All other terms are completely local and only correspond to the local Q-function Q_{ij} of the edge that is updated. A different approach is to first compute the temporal-difference error *per agent* and divide this value over the edges. For this, we first rewrite (16) for agent i using (18) to

$$\frac{1}{2}\sum_{j\in\Gamma(i)} Q_{ij}(\mathbf{s}_{ij},a_i,a_j) := \frac{1}{2}\sum_{j\in\Gamma(i)}[Q_{ij}(\mathbf{s}_{ij},a_i,a_j)] + \alpha[R_i(\mathbf{s},\mathbf{a}) + \gamma Q_i(\mathbf{s}'_i,\mathbf{a}^*_i) - Q_i(\mathbf{s}_i,\mathbf{a}_i)]. \quad (21)$$

In order to transfer (21) into a local update function, we first rewrite the temporal-difference error as a summation of the neighbors of agent i, by

$$R_i(\mathbf{s},\mathbf{a}) + \gamma Q_i(\mathbf{s}'_i,\mathbf{a}^*_i) - Q_i(\mathbf{s}_i,\mathbf{a}_i) = \sum_{j\in\Gamma(i)} \frac{R_i(\mathbf{s},\mathbf{a}) + \gamma Q_i(\mathbf{s}'_i,\mathbf{a}^*_i) - Q_i(\mathbf{s}_i,\mathbf{a}_i)}{|\Gamma(i)|}. \quad (22)$$

Note that this summation only decomposes the temporal-difference error into j equal parts, and thus does not use j explicitly. Because now all summations are identical, we can decompose (21) by removing the sums. Just as in the edge-based update, there are two agents which update the same local Q-function Q_{ij}. When we add the contributions of the two involved agents i and j, we get the local update equation

$$Q_{ij}(\mathbf{s}_{ij},a_i,a_j) := Q_{ij}(\mathbf{s}_{ij},a_i,a_j) + \alpha \sum_{k\in\{i,j\}} \frac{R_k(\mathbf{s},\mathbf{a}) + \gamma Q_k(\mathbf{s}'_k,\mathbf{a}^*_k) - Q_k(\mathbf{s}_k,\mathbf{a}_k)}{|\Gamma(k)|}. \quad (23)$$

This agent-based update rule propagates back the temporal-difference error from the two agents which are related to the local Q-function of the edge that is updated, and incorporates the information of *all* edges of these agents. This is different from the edge-based update method which directly propagates back the temporal-difference error related to the edge that is updated. This is depicted in Fig. 8(b). Again, the left part of the figure represents the situation in state $\mathbf{s}$, and the right part the situation in the next state $\mathbf{s}'$. The edge-based Q-function Q_{ij} is updated based on the local agent-based Q-functions of the two agents that form the edge. These functions are computed by summing over the local edge-based Q-functions of all neighboring edges.

Next, we will describe several experiments and solve them using both the agent-based and the edge-based decomposition. For the latter, we apply both the agent-based and edge-based update method, and show the consequences, both in speed and solution quality, of using the max-plus algorithm as an alternative to the VE algorithm.

6. Experiments

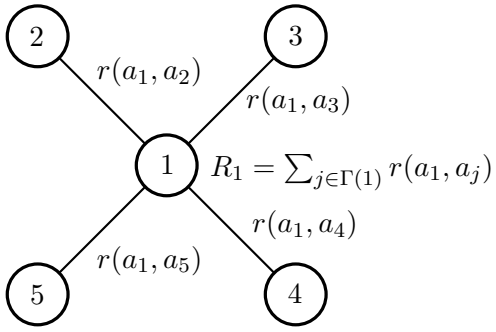
In this section, we describe experiments using the methods discussed in Section 4 and Section 5. We give results on a large single-state problem and on a distributed sensor network problem, which was part of the NIPS 2005 benchmarking workshop. We selected these problems because they are both fully specified and, more importantly, require the selection of a specific combination of actions at every time step. This is in contrast with other experiments in which coordination can be modeled through the state variables, that is, each agent is able to select its optimal action based on only the state variables (for example, its own and other agents' positions) and does not have to model the actions of the other agents (Tan, 1993; Guestrin et al., 2002b; Becker et al., 2003).

6.1 Experiments on Single-State Problems

Now, we describe several experiments in which a group of n agents have to learn to take the optimal joint action in a single-state problem. The agents repeatedly interact with their environment by selecting a joint action. After the execution of a joint action $\mathbf{a}$, the episode is immediately ended and the system provides each agent an individual reward $R_i(\mathbf{a})$. The goal of the agents is to select the joint action $\mathbf{a}$ which maximizes $R(\mathbf{a}) = \sum_{i=1}^{n} R_i(\mathbf{a})$. The local reward R_i received by an agent i only depends on a subset of the actions of the other agents. These dependencies are modeled using a graph in which each edge corresponds to a local reward function that assigns a value $r(a_i, a_j)$ to each possible action combination of the actions of agent i and agent j. Each local reward function is fixed beforehand and contains one specific pair of actions, $(\tilde{a}_i, \tilde{a}_j)$ that results in a high random reward, uniformly distributed in the range $[5, 15]$, that is, $5 + \mathcal{U}([0, 10])$. However, failure of coordination, that is, selecting an action $r(\tilde{a}_i, a_j)$ with $a_j \neq \tilde{a}_j$ or $r(a_i, \tilde{a}_j)$ with $a_i \neq \tilde{a}_i$, will always result in a reward of 0. All remaining joint actions, $r(a_i, a_j)$ with $a_i \neq \tilde{a}_i$ and $a_j \neq \tilde{a}_j$, give a default reward from the uniform distribution $\mathcal{U}([0, 10])$. The individual reward R_i for an agent i equals the sum of the local rewards resulting from the interactions with its neighbors, $R_i(\mathbf{a}) = \sum_{j \in \Gamma(i)} r(a_i, a_j)$. Fig. 9 shows an example of the construction of the individual reward received by an agent based on its interaction with its four neighbors, together with an example reward function $r(a_i, a_j)$ corresponding to an edge between agent i and agent j.

The goal of the agents is to learn, based on the received individual rewards, to select a joint action that maximizes the global reward. Although we assume that the agents know on which other agents it depends, this goal is complicated by two factors. First, the outcome of a selected action of an agent also depends on the actions of its neighbors. For example, the agents must coordinate in order to select the joint action $(\tilde{a}_i, \tilde{a}_j)$ which, in most cases, returns the highest reward. Failure of coordination, however, results in a low reward. Secondly, because each agent only receives an individual reward, they are not able to derive which neighbor interaction caused which part of the reward. An important difference with the problem specified in Section 3.2, in which the the agents have to select a joint action that maximizes predefined payoff functions, is that in this case the payoff relations themselves have to be learned based on the received rewards.

We perform experiments with 12 agents, each able to perform 4 actions. The group as a whole thus has $4^{12} \approx 1.7 \cdot 10^7$, or 17 million, different joint actions to choose from. We investigate reward functions with different complexities, and apply the method described



(a) Construction of reward $R_1(\mathbf{a})$.

action agent i \ action agent j	1	2	3	4
1	9.155	0	9.293	0.196
2	7.651	0	2.872	1.287
3	0	12.020	0	0
4	4.536	0	1.581	8.138

(b) Example $r(a_i, a_j)$ function.

Figure 9: Construction of the reward for agent 1 in the single-state problem. (a) The individual reward R_1 is the sum of the rewards $r(a_1, a_j)$ generated by the interactions with its neighbors $j \in \Gamma(1) = \{2, 3, 4, 5\}$. (b) Example $r(a_i, a_j)$ function.

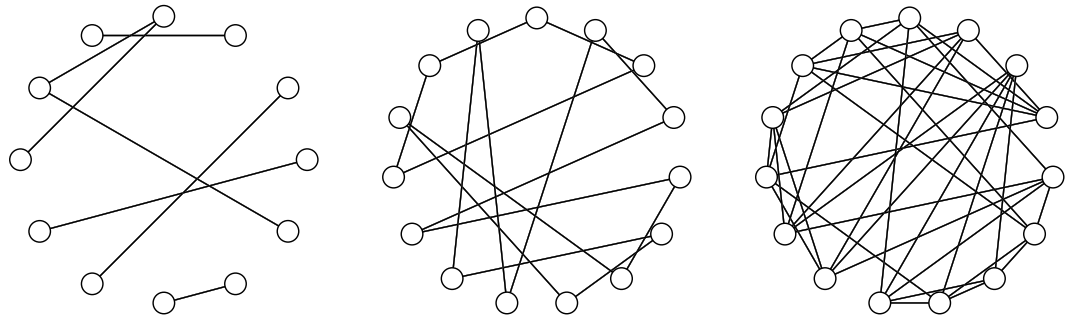
in Section 3.2 to randomly generate 20 graphs $G = (V, E)$ with $|V| = 12$ for each $|E| \in \{7, 8, \ldots, 30\}$. This results in 480 graphs, 20 graphs in each of the 24 groups. The agents of the simplest graphs (7 edges) have an average degree of 1.16, while the most complex graphs (30 edges) have an average degree of 5. Fig. 10 shows three different example graphs with different average degrees. Fig. 10(a) and (c) depict respectively the minimum and maximal considered average degree, while Fig. 10(b) shows a graph with an average degree of 2.

We apply the different variants of our sparse cooperative Q-learning method described in Section 5 and different existing multiagent Q-learning methods, discussed in Section 4, to this problem. Since the problem consists of only a single state, all Q-learning methods store Q-functions based on actions only. Furthermore, we assume that the agents have access to a CG which for each agent specifies on which other agents it depends. This CG is identical to the topology of the graph that is used to generate the reward function. Apart from the different Q-learning methods, we also apply an approach that selects a joint action uniformly at random and keeps track of the best joint action found so far, and a method that enumerates all possible joint actions and stores the one with the highest reward. To summarize, we now briefly review the main characteristics of all applied methods:

Independent learners (IL) Each agent i stores a local Q-function $Q_i(a_i)$ only depending on its own action. Each update is performed using the private reward R_i according to (11). An agent selects an action that maximizes its own local Q-function Q_i.

Distributed value functions (DVF) Each agent i stores a local Q-function based on its own action, and an update incorporates the Q-functions of its neighbors following (14). For stateless problems, as the ones in this section, the Q-value of the next state is not used and this method is identical to IL.

Coordinated reinforcement learning (CoordRL) Each agent i stores an individual Q-function based on its own action and the actions of its neighbors $j \in \Gamma(i)$. Each function is updated based on the *global* temporal-difference error using the update



(a) A graph with 7 edges (average degree of 1.16).

(b) A graph with 12 edges (average degree of 2).

(c) A graph with 30 edges (average degree of 5.00).

Figure 10: Example graphs with 12 agents and different average degrees.

equation in (13). This representation scales exponentially with the number of neighbors. VE is used to determine the optimal joint action which scales exponentially with the induced width of the graph.

Sparse cooperative Q-learning, agent-based (SparseQ agent) Each agent stores a Q-function based on its own action and the actions of its neighbors $j \in \Gamma(i)$. A function is updated based on the *local* temporal-difference error following (16). The representation and computational complexity are similar to the CoordRL approach.

Sparse cooperative Q-learning, edge-based (SparseQ edge) Each edge in the used CG is associated with a Q-function based on the actions of the two connected agents. We apply both the *edge-based* update method (SparseQ edge, edge) from (20) which updates a Q-function based on the value of the edge that is updated, and the *agent-based* update method (SparseQ edge, agent) from (23), which updates a Q-function based on the local Q-functions of the agents forming the edge.

The two update methods are both executed with the VE algorithm and the anytime max-plus algorithm in order to determine the optimal joint action, resulting in four different methods in total. The max-plus algorithm generates a result when either the messages converge, the best joint action has not improved for 5 iterations, or more than 20 iterations are performed. The latter number of iterations is obtained by comparing the problem under study with the coordination problem addressed in Section 3.2. Both problem sizes are similar, and as is visible in Fig. 6 the coordination problem reaches a good performance after 20 iterations.

Random method with memory Each iteration, each agent selects an action uniformly at random. The resulting joint action is evaluated and compared to the best joint action found so far. The best one is stored and selected.

Enumeration In order to compare the quality of the different methods, we compute the optimal value by trying every possible joint action and store the one which results in the highest reward. This requires an enumeration over all possible joint actions. Note that this approach does not perform any updates, and quickly becomes intractable for problems larger than the one addressed here.

method	$(1, 2]$	$(2, 3]$	$(3, 4]$	$(4, 5]$
IL/DVF	48	48	48	48
edge-based	152	248	344	440
agent-based	528	2,112	8,448	33,792

Table 2: Average number of Q-values needed for the different decompositions for graphs with an average degree in $(x - 1, x]$.

We do not apply the MDP learners approach since it would take too long to find a solution. First, it requires an enumeration over 4^{12} (≈ 17 million) actions at every time step. Secondly, assuming there is only one optimal joint action, the probability to actually find the optimal joint action is negligible. An exploration action should be made (probability ϵ), and this exploration action should equal the optimal joint action (probability of $\frac{1}{4^{12}}$).

Table 2 shows the average number of Q-values required by each of the three types of decompositions. The numbers are based on the generated graphs and averaged over similarly shaped graphs. Note the exponential growth in the agent-based decomposition that is used in both the CoordRL and agent-based SparseQ approach.

We run each method on this problem for $15,000$ learning cycles. Each learning cycle is directly followed by a test cycle in which the reward related to the current greedy joint action is computed. The values from the test cycles, thus without exploration, are used to compare the performance between the different methods. For all Q-learning variants, the Q-values are initialized to zero and the parameters are set to $\alpha = 0.2$, $\epsilon = 0.2$, and $\gamma = 0.9$.

Fig. 11 shows the timing results for all methods.[5] The x-axis depicts the average degree of the graph. The y-axis, shown in logarithmic scale, depicts the average number of seconds spent in the $15,000$ learning cycles on graphs with a similar average degree. For the enumeration method it represents the time for computing the reward of all joint actions.

The results show that the random and IL/DVF approach are the quickest and take less than a second to complete. In the IL/DVF method each agent only stores functions based on its individual action and is thus constant in the number of dependencies in the graph. Note that the time increase in the random approach for graphs with a higher average degree is caused by the fact that more local reward functions have to be enumerated in order to compute the reward. This occurs in all methods, but is especially visible in the curve of the random approach since for this method the small absolute increase is relatively large with respect to its computation time.

The CoordRL and the agent-based SparseQ method scale exponentially with the increase of the average degree, both in their representation of the local Q-functions and the computation of the optimal joint action using the VE algorithm. The curves of these methods overlap in Fig. 11. Because these methods need a very long time, more than a day, to process graphs with a higher average degree than 3, the results for graphs with more than 18 edges are not computed. The edge-based decompositions do not suffer from the exponential growth in the representation of the local Q-functions. However, this approach

[5] All results are generated on an Intel Xeon 3.4GHz / 2GB machine using a C++ implementation.

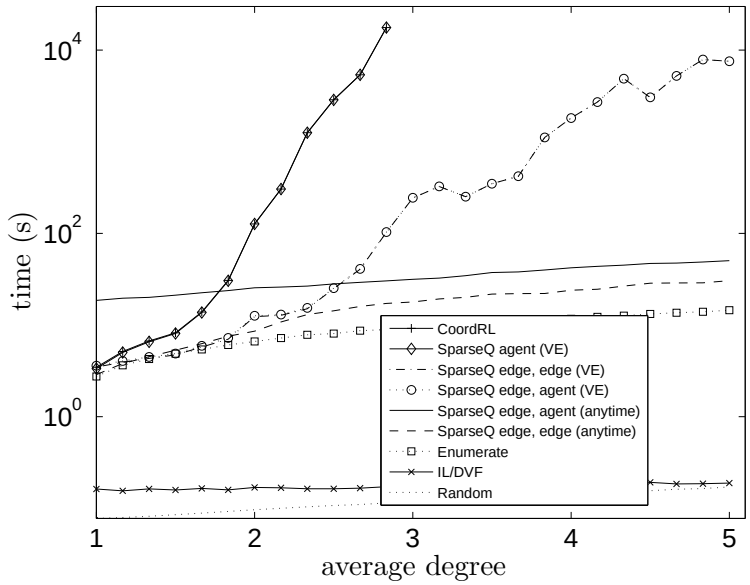


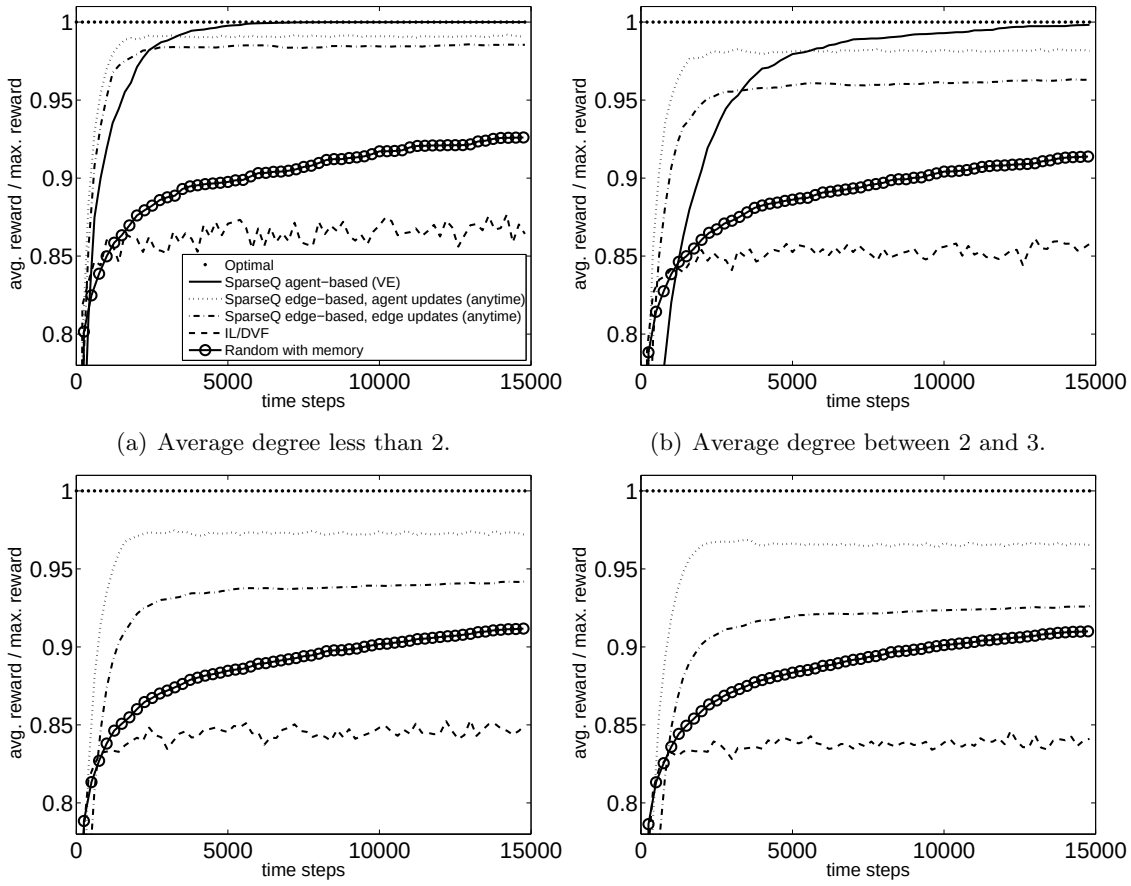
Figure 11: Timing results for the different methods applied to the single-state problems with 12 agents and an increasing number of edges. The results overlap for the CoordRL and the agent-based SparseQ decomposition, and the two edge-based decompositions using the VE algorithm.

still grows exponentially with an increase of the average degree when the VE algorithm is used to compute the maximizing joint action. This holds for both the agent-based and edge-based update method, which overlap in the graph. When the anytime max-plus algorithm is applied to compute the joint action, both the representation of the Q-function and the computation of the joint action scale linearly with an increasing average degree. The agent-based update method is slightly slower than the edge-based update method because the first incorporates the neighboring Q-functions in its update (23), and therefore the values in the Q-functions are less distinct. As a consequence, the max-plus algorithm needs more iterations in an update step to find the maximizing joint action.

Finally, the enumeration method shows a slight increase in the computation time with an increase of the average degree because it has to sum over more local functions for the denser graphs when computing the associated value. Note that the problem size was chosen such that the enumeration method was able to produce a result for all different graphs.

Fig. 12 shows the corresponding performance for the most relevant methods. Each figure depicts the running average, of the last 10 cycles, of the obtained reward relative to the optimal reward for the first $15,000$ cycles. The optimal reward is determined using the enumeration method. Results are grouped for graphs with a similar complexity, that is, having about the same number of edges per graph.

Fig. 12(a) depicts the results for the simplest graphs with an average degree less than or equal to 2. We do not show the results for the CoordRL approach since it is not able to learn a good policy and quickly stabilize around 41% of the optimal value. This corresponds



(a) Average degree less than 2.

(b) Average degree between 2 and 3.

(c) Average degree between 3 and 4.

(d) Average degree between 4 and 5.

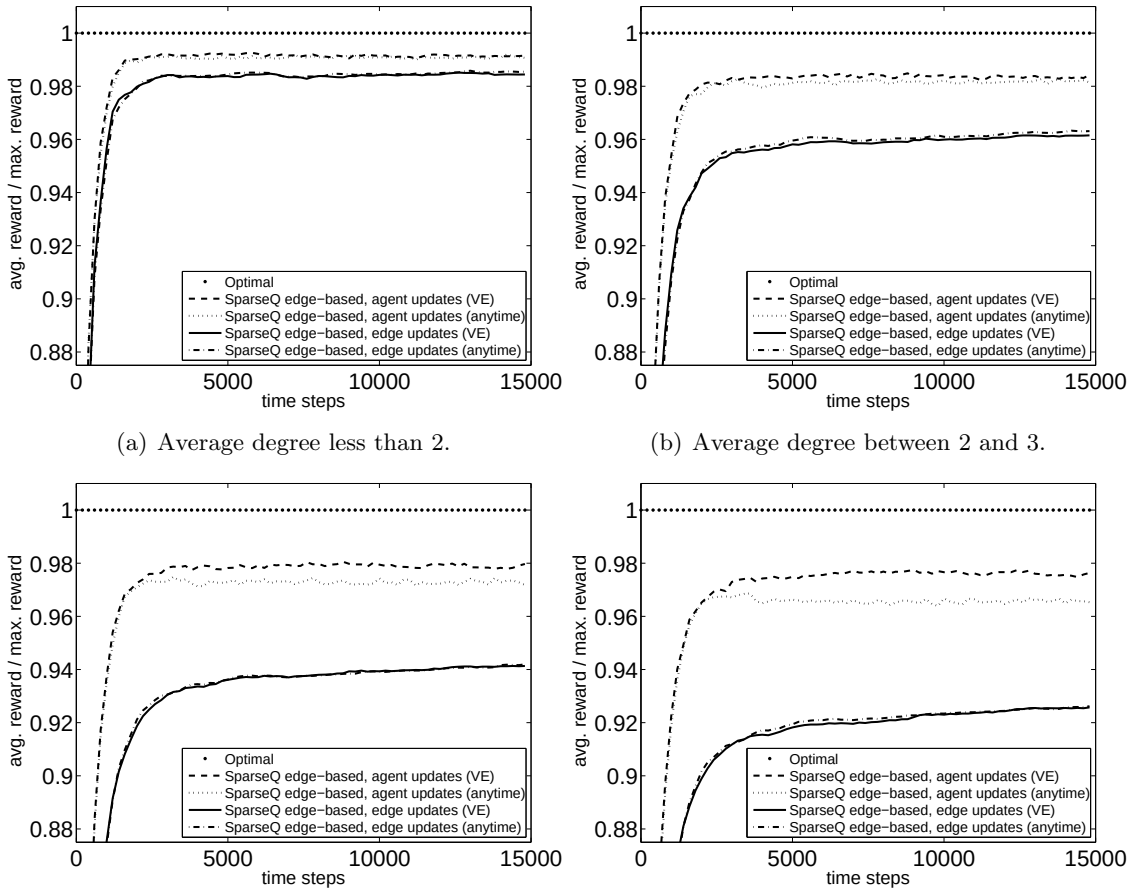
Figure 12: Running average, of the last 10 cycles, of the received reward relative to the optimal reward for different methods on the single-state, 12-agent problems. The legend of Fig. 12(a) holds for all figures. See text for the problem description.

to a method in which each agent selects a value uniformly at random each iteration. The CoordRL approach updates each local Q-function with the global temporal-difference error. Therefore, the same global reward is propagated to each of the individual Q-functions and the expected future discounted return, that is, the sum of the local Q-functions, is overestimated. As a result the Q-values blow up, resulting in random behavior.

The IL/DVF approach learns a reasonable solution, but it suffers from the fact that each agent individually updates its Q-value irrespective of the actions performed by its neighbors. Therefore, the agents do not learn to coordinate and the policy keeps oscillating.

The random method keeps track of the best joint action found so far and slowly learns a better policy. However, it learns slower than the different SparseQ methods. Note that this method does not scale well to larger problems with more joint actions.

The agent-based SparseQ decomposition converges to an optimal policy since it stores a Q-value for every action combination of its neighbors, and is able to detect the best



(a) Average degree less than 2.

(b) Average degree between 2 and 3.

(c) Average degree between 3 and 4.

(d) Average degree between 4 and 5.

Figure 13: Running average of the received reward relative to the optimal reward for the different edge-based methods, using either the VE or anytime max-plus algorithm, on the single-state, 12-agent problem.

performing action combination. However, this approach learns slower than the different edge-based decompositions since it requires, as listed in Table 2, more samples to update the large number of Q-values. The two edge-based decompositions using the anytime extension both learn a near-optimal solution. The agent-based update method performs slightly better since it, indirectly, includes the neighboring Q-values in its update.

As is seen in Fig. 12(b), the results are similar for the more complicated graphs with an average degree between 2 and 3. Although not shown, the CoordRL learners are not able to learn a good policy and quickly stabilizes around 44% of the optimal value. On the other hand, the agent-based decomposition converges to the optimal policy. Although the final result is slightly worse compared to the simpler graphs, the edge-based decompositions still learn near-optimal policies. The result of the agent-based update method is better than the edge-based update method since the first includes the neighboring Q-values in its update.

method	$(1, 2]$	$(2, 3]$	$(3, 4]$	$(4, 5]$
Random with memory	0.9271	0.9144	0.9122	0.9104
IL	0.8696	0.8571	0.8474	0.8372
CoordRL	0.4113	0.4423	-	-
SparseQ agent (VE)	1.0000	0.9983	-	-
SparseQ edge, agent (VE)	0.9917	0.9841	0.9797	0.9765
SparseQ edge, edge (VE)	0.9843	0.9614	0.9416	0.9264
SparseQ edge, agent (anytime)	0.9906	0.9815	0.9722	0.9648
SparseQ edge, edge (anytime)	0.9856	0.9631	0.9419	0.9263

Table 3: Relative reward with respect to the optimal reward after $15,000$ cycles for the different methods and differently shaped graphs. Results are averaged over graphs with an average degree in $(x - 1, x]$, as indicated by the column headers.

Similar results are also visible in Fig. 12(c) and Fig. 12(d). The agent-based decompositions are not applied to these graphs. As was already visible in Fig. 11, the algorithm needs too much time to process graphs of this complexity.

Fig. 13 compares the difference between using either the VE or the anytime max-plus algorithm to compute the joint action for the SparseQ methods using an edge-based decomposition. Fig. 13(a) and Fig. 13(b) show that the difference between the two approaches is negligible for the graphs with an average degree less than 3. However, for the more complex graphs (Fig. 13(c) and Fig. 13(d)) there is a small performance gain when the VE algorithm is used for the agent-based update method. The agent-based update method incorporates the neighboring Q-functions, and therefore the values of the Q-functions are less distinct. As a result, the max-plus algorithm has more difficulty in finding the optimal joint action. But note that, as was shown in Fig. 11, the VE algorithm requires substantially more computation time for graphs of this complexity than the anytime max-plus algorithm.

Although all results seem to converge, it is difficult to specify in which cases the proposed algorithms converge, and if so, whether they converge to an optimal solution. The difficulties arise from the fact that the reinforcement-learning algorithms deal with a double optimization: the computation of the optimal joint action with the maximal Q-value, and the global (long-term) optimization of the average discounted rewards. In this article we focus on the empirical results.

Table 3 gives an overview of all results and compares the value of the joint action corresponding to the learned strategy in cycle $15,000$ for the different methods. Although the results slowly decrease for the more complex reward functions, all SparseQ methods learn near-optimal policies. Furthermore, there is only a minimal difference between the methods that use the VE and the anytime max-plus algorithm to compute the joint action. For the densely connected graphs, the edge-based decompositions in combination with the max-plus algorithm are the only methods that are able to compute a good solution. The algorithms using VE fail to produce a result because of their inability to cope with the complexity of the underlying graph structure (see Section 3.2).

6.2 Experiments on a Distributed Sensor Network

We also perform experiments on a distributed sensor network (DSN) problem. This problem is a sequential decision-making variant of the distributed constraint optimization problem described by Ali et al. (2005), and was part of the NIPS 2005 benchmarking workshop.[6]

The DSN problem consists of two parallel chains of an arbitrary, but equal, number of sensors. The area between the sensors is divided into cells. Each cell is surrounded by exactly four sensors and can be occupied by a target. See Fig. 14(a) for a configuration with eight sensors and two targets. With equal probability a target moves to the cell on its left, to the cell on its right, or remains on its current position. Actions that move a target to an illegal position, that is, an occupied cell or a cell outside the grid, are not executed.

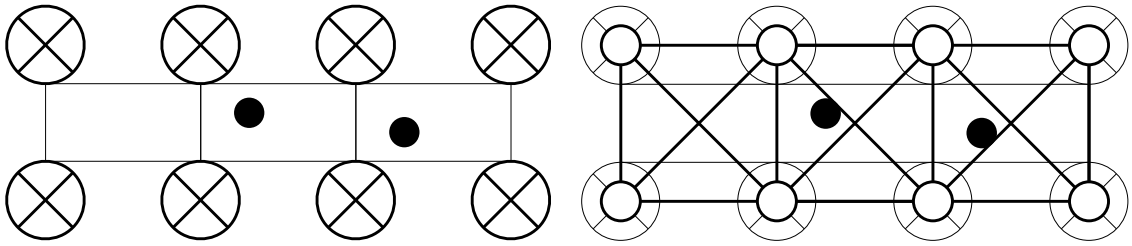
Each sensor is able to perform three actions: focus on a target in the cell to its immediate left, to its immediate right, or don't focus at all. Every focus action has a small cost modeled as a reward of -1. When in one time step at least three of the four surrounding sensors focus on a target, it is 'hit'. Each target starts with a default energy level of three. Each time a target is hit its energy level is decreased by one. When it reaches zero the target is captured and removed, and the three sensors involved in the capture each receive a reward of $+10$. In case four sensors are involved in a capture, only the three sensors with the highest index receive the reward. An episode finishes when all targets are captured.

As in the NIPS-05 benchmarking event, we will concentrate on a problem with eight sensors and two targets. This configuration results in $3^8 = 6,561$ joint actions and 37 distinct states, that is, 9 states for each of the 3 configurations with 2 targets, 9 for those with one target, and 1 for those without any targets. This problem thus has a large action space compared to its state space. When acting optimally, the sensors are able to capture both targets in three steps, resulting in a cumulative reward of 42. However, in order to learn this policy based on the received rewards, the agents have to cope with the delayed reward and learn how to coordinate their actions such that multiple targets are hit simultaneously.

In our experiments we generate all statistics using the benchmark implementation, with the following two differences. First, because the NIPS-05 implementation of the DSN problem only returns the global reward, we change the environment to return the individual rewards in order to comply to our model specification. Second, we set the fixed seed of the random number generator to a variable seed base on the current time in order to be able to perform varying runs.

We apply the different techniques described in Section 4 and Section 5 to the DSN problem. We do not apply the CoordRL approach, since, just as in the experiments in Section 6.1, it propagates back too much reward causing the individual Q-functions to blow up. However, we do apply the MDP learners approach which updates a Q-function based on the full joint action space. All applied methods learn for $10,000$ episodes which are divided into 200 episode blocks, each consisting of 50 episodes. The following statistics are computed at the end of each episode block: the average reward, that is, the undiscounted sum of rewards divided by the number of episodes in an episode block, the cumulative average reward of all previous episode blocks, and the wall-clock time. There is no distinction between learning and testing cycles, and the received reward thus includes exploration

[6] See `http://www.cs.rutgers.edu/~mlittman/topics/nips05-mdp/` for a detailed description of the benchmarking event and `http://rlai.cs.ualberta.ca/RLBB/` for the used RL-framework.



(a) Network configuration. (b) Network with corresponding CG.

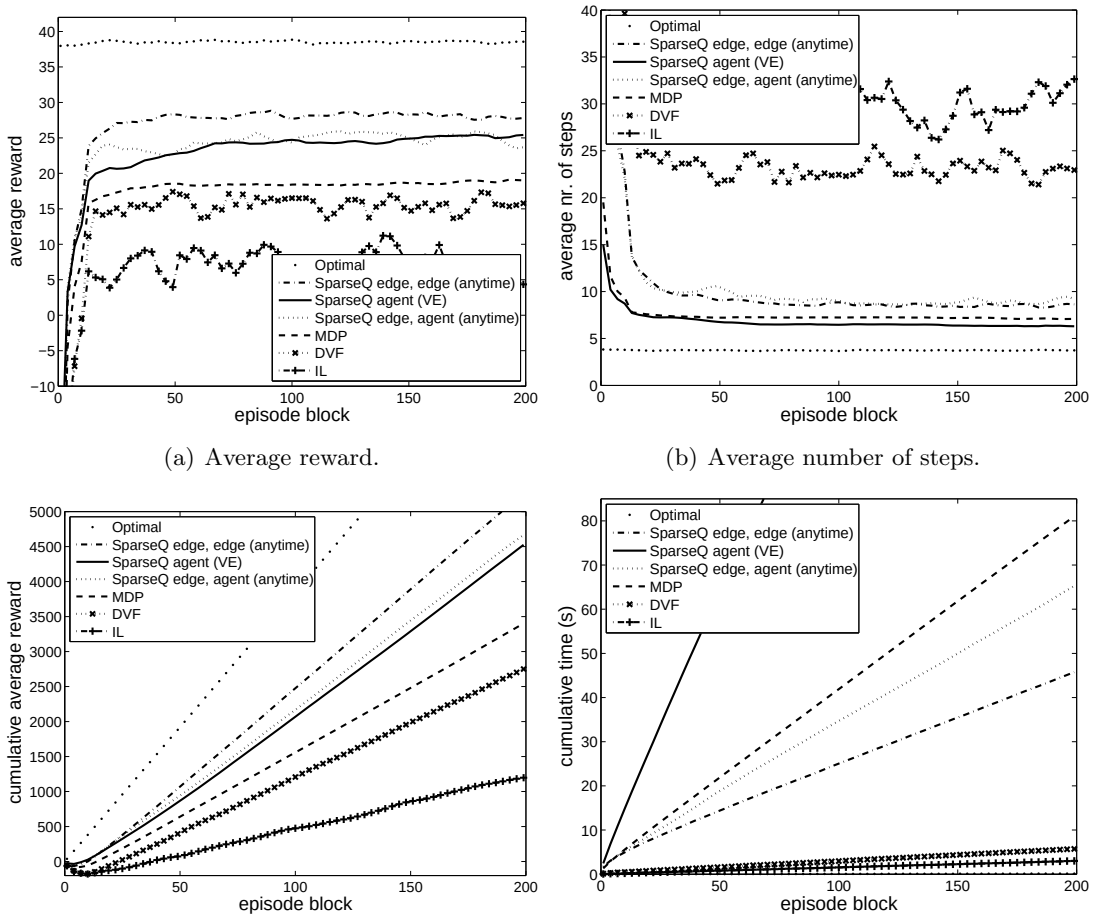
Figure 14: Fig. 14(a) shows an example sensor network with eight sensors ($\bigotimes$) and two targets ($\bullet$). Fig. 14(b) shows the corresponding CG representing the agent dependencies. The graph has an average degree of 4, and an induced width of 3.

actions. The Q-learning methods all use the following parameters: $\alpha = 0.2$, $\epsilon = 0.2$, and $\gamma = 0.9$, and start with zero-valued Q-values. We assume that both the DVF and the different SparseQ variants have access to a CG which specifies for each agent on which other agents it depends. This CG is shown in Fig. 14(b), and has an average degree of 4.

The results, averaged over 10 runs with different random seeds, for the different techniques are shown in Fig. 15. The results contain exploration actions and are therefore not completely stable. For this reason, we show the running average over the last 10 episode blocks. Fig. 15(a) shows the average reward for the different approaches. The optimal policy is manually implemented and, in order to have a fair comparison with the other approaches, also includes random exploration actions with probability ϵ. It results in an average reward just below 40. The MDP approach settles to an average reward around 17 after a few episodes. Although this value is low compared to the result of the optimal policy, the MDP approach, as seen in Fig. 15(b), does learn to capture the targets in a small number of steps. From this we conclude that the low reward is mainly a result of unnecessary focus actions performed by the agents that are not involved in the actual capture. The MDP approach thus discovers one of the many possible joint actions that results in a capture of the target and the generation of a positive reward, and then exploits this strategy. However, the found joint action is non-optimal since one or more agents do not have to focus in order to capture the target. Because of the large action space and the delayed reward, it takes the MDP approach much more than $10,000$ episodes to learn that other joint actions result in a higher reward.

Although the DVF approach performs better than IL, both methods do not converge to a stable policy and keep oscillating. This is caused by the fact that both approaches store action values based on individual actions and therefore fail to select coordinated joint actions which are needed to capture the targets.

In the different SparseQ variants each agent stores and updates local Q-values. Since these are also based on the agent's neighbors in the graph, the agents are able to learn coordinated actions. Furthermore, the explicit coordination results in much more stable policies than the IL and DVF approach. The agent-based decomposition produces a slightly lower average reward than the edge-based decompositions, but, as shown in Fig. 15(b), it needs less steps to capture the targets. Identical to the MDP approach, the lower reward obtained



(a) Average reward.

(b) Average number of steps.

(c) Cumulative average reward.

(d) Cumulative time.

Figure 15: Different results on the DSN problem, averaged over 10 runs. One run consists of 200 episode blocks, each corresponding to 50 learning episodes.

by the agent-based decomposition is a consequence of the large action space involved in each local term. As a result the agents are able to quickly learn a good policy that captures the targets in a few steps, but it takes a long time to converge to a joint action that does not involve the unnecessary focus actions of some of the agents. For example, each of the four agents in the middle of the DSN coordinates with 5 other agents, and each of them thus stores a Q-function defined over $3^6 = 729$ actions per state. Because in the agent-based decomposition the full action space is decomposed into different independent local action values, it does result in a better performance than the MDP learners, both in the obtained average reward and the number of steps needed to capture the targets. With respect to the two edge-based decompositions, the edge-based update method generates a slightly higher reward, and a more stable behavior than the agent-based update method. Although in both cases the difference between the two methods is minimal, this result is different compared to the stateless problems described in Section 6.1 in which the agent-based update method

method	reward	steps	method	reward	steps
Optimal	38.454	3.752	SparseQ edge, edge (anytime)	27.692	8.795
MDP	19.061	7.071	SparseQ edge, edge (VE)	28.880	8.113
DVF	16.962	22.437	SparseQ agent (VE)	24.844	6.378
IL	6.025	31.131	SparseQ edge, agent (VE)	25.767	8.413
			SparseQ edge, agent (anytime)	23.738	8.930

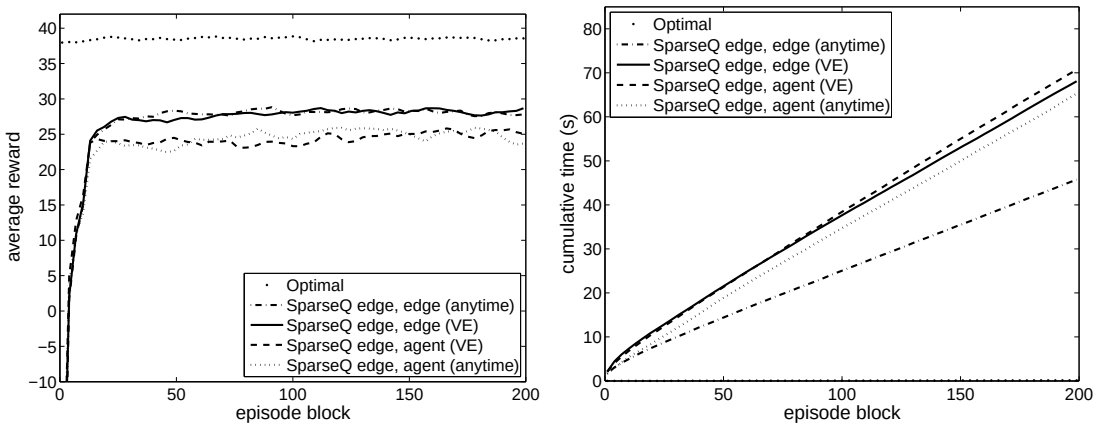
Table 4: Average reward and average number of steps per episode over the last 2 episode blocks (100 episodes) for the DSN problem. Results are averaged over 10 runs.

performed better. The effectiveness of each approach thus depends on the type of problem. We believe that the agent-based update method has its advantages for problems with fine-grained agent interactions since it combines all neighbors in the update of the Q-value.

Fig. 15(c) shows the cumulative average reward of the different methods. Ignoring the manual policy, the edge-based update methods result in the highest cumulative average reward. This is also seen in Table 4 which shows the reward and the number of steps per episode averaged over the last 2 episode blocks, that is, 100 episodes, for the different methods. Since the goal of the agents is to optimize the received average reward, the SparseQ methods outperform the other learning methods. However, none of the variants converge to the optimal policy. One of the main reasons is the large number of dependencies between the agents. This requires a choice between an approach that models many of the dependencies but learns slowly because of the exploration of a large action space, for example, the agent-based SparseQ or the MDP learners, or an approach that ignores some of the dependencies but is able to learn an approximate solution quickly. The latter is the approach taken by the edge-based SparseQ variants: it models pairwise dependencies even though it requires three agents to capture a target.

Fig. 15(d) gives the timing results for the different methods. The IL and DVF methods are the fastest methods since they only store and update individual Q-values. The agent-based SparseQ method is by far the slowest. This method stores a Q-function based on all action combinations of an agent and its neighbors in the CG. This slows down the VE algorithm considerably since it has to maximize over a large number of possible joint action combinations in every local maximization step.

Finally, Fig. 16 compares the difference between using the VE or the anytime max-plus algorithm to compute the joint action for the SparseQ methods using an edge-based decomposition. Fig. 16(a) shows that there is no significant difference in the obtained reward for these two methods. Fig. 16(b) shows that the edge-based SparseQ variants that use the anytime max-plus algorithm need less computation time than those using the VE algorithm. However, the differences are not that evident as in the experiments from Section 6.1 because the used CG has a relative simple structure (it has an induced width of 3), and VE is able to quickly find a solution when iteratively eliminating the nodes with the smallest degree.



(a) Average reward. (b) Cumulative time.

Figure 16: Results of the edge-based decomposition methods on the DSN problem, averaged over 10 runs. One run consists of 200 episode blocks, each corresponding to 50 learning episodes.

7. Conclusion and Future Directions

In this article we addressed the problem of learning how to coordinate the behavior of a large group of agents. First, we described a payoff propagation algorithm (max-plus) that can be used as an alternative to variable elimination (VE) for finding the optimal joint action in a coordination graph (CG) with predefined payoff functions. VE is an exact method that will always report the joint action that maximizes the global payoff, but it is slow for densely connected graphs with cycles because its worst-case complexity is exponential in the number of agents. Furthermore, this method is only able to report a solution after the complete algorithm has ended. The max-plus algorithm, analogous to the belief propagation algorithm in Bayesian networks, operates by repeatedly sending local payoff messages over the edges in the CG. By performing a local computation based on its incoming messages, each agent is able to select its individual action. For tree-structured graphs, this approach results in the optimal joint action. For large, highly connected graphs with cycles, we provided empirical evidence that this method can find near-optimal solutions exponentially faster than VE. Another advantage of the max-plus algorithm is that it can be implemented fully distributed using asynchronous and parallel message passing.

Second, we concentrated on model-free reinforcement-learning approaches to learn the coordinated behavior of the agents in a collaborative multiagent system. In our Sparse Cooperative Q-learning (SparseQ) methods, we approximate the global Q-function using a CG representing the coordination requirements of the system. We analyzed two possible decompositions, one in terms of the nodes and one in terms of the edges of the graph. During learning, each local Q-function is updated based on its contribution to the maximal global payoff found with either the VE or max-plus algorithm. Effectively, each agent learns its part of the global solution by only coordinating with the agents on which it depends. Results on both a single-state problem with 12 agents and more than 17 million actions,

and a distributed sensor network problem show that our SparseQ variants outperform other existing multiagent Q-learning methods. The combination of the edge-based decomposition and the max-plus algorithm results in a method which scales only linearly in the number of dependencies of the problem. Furthermore, it can be implemented fully distributed and only requires that each agent is able to communicate with its neighbors in the graph. When communication is restricted, it is still possible to run the algorithm when additional common knowledge assumptions are made.

There are several directions for future work. First of all, we are interested in comparing different approximation alternatives from the Bayesian networks or constraint processing literature to our max-plus algorithm. A natural extension is to consider factor graph representations of the problem structure (Kschischang et al., 2001), allowing more prior knowledge about the problem to be introduced beforehand. Another possible direction involves the 'mini-bucket' approach, an approximation in which the VE algorithm is simplified by changing the full maximization for each elimination of an agent to the summation of simpler local maximizations (Dechter and Rish, 1997). A different alternative for the VE algorithm is the usage of constraint propagation algorithms for finding the optimal joint action (Modi et al., 2005). Another interesting issue is related to the Q-updates of the edge-based decomposition of the SparseQ reinforcement-learning method. Now we assume that the received reward of an agent is divided proportionally over its edges (see (20) and (23)), but other schemes may also be possible. Furthermore, we like to apply our method to problems in which the topology of the CG differs per state, for example, when agents are dynamically added or removed from the system, or dependencies between the agents change based on the current situation (Guestrin et al., 2002c). Since all Q-functions and updates are defined locally, it is possible to compensate the addition or removal of an agent by redefining only the Q-functions in which this agent is involved. The max-plus algorithm and the local updates do not have to be changed as long as the neighboring agents are aware of the new topology of the CG.

Acknowledgments

We would like to thank Carlos Guestrin and Ron Parr for providing ample feedback to this work. Furthermore, we like to thank all three reviewers for their detailed and constructive comments. This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, project AES 5414.

References

S. Muhammad Ali, S. Koenig, and M. Tambe. Preprocessing techniques for accelerating the DCOP algorithm ADOPT. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1041–1048, Utrecht, The Netherlands, 2005.

T. Arai, E. Pagello, and L. E. Parker. Editorial: Advances in multi-robot systems. *IEEE Transactions on Robotics and Automation*, 18(5):665–661, 2002.

R. Becker, S. Zilberstein, V. Lesser, and C. V. Goldman. Transition-independent decentralized Markov decision processes. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Melbourne, Australia, 2003.

D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of Markov decision processes. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, Stanford, CA, 2000.

U. Bertelé and F. Brioschi. *Nonserial dynamic programming*. Academic Press, 1972.

D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

C. Boutilier. Planning, learning and coordination in multiagent decision processes. In *Proceedings of the Conference on Theoretical Aspects of Rationality and Knowledge*, 1996.

J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, editors, *Advances in Neural Information Processing Systems (NIPS) 6*, pages 671–678. Morgan Kaufmann Publishers, Inc., 1994.

W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative multi-robot exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2000.

G. Chalkiadakis and C. Boutilier. Coordination in multiagent reinforcement learning: A Bayesian approach. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 709–716, Melbourne, Australia, 2003. ACM Press.

C. Claus and C. Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Madison, WI, 1998.

C. Crick and A. Pfeffer. Loopy belief propagation as a basis for communication in sensor networks. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, 2003.

R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS) 8*, pages 1017–1023. MIT Press, 1996.

R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

R. Dechter and I. Rish. A scheme for approximating probabilistic inference. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, pages 132–141, 1997.

E. H. Durfee. Scaling up agent coordination strategies. *IEEE Computer*, 34(7):39–46, July 2001.

P. S. Dutta, N. R. Jennings, and L. Moreau. Cooperative information sharing to improve distributed learning in multi-agent systems. *Journal of Artificial Intelligence Research*, 24:407–463, 2005.

C. Goldman and S. Zilberstein. Decentralized control of cooperative systems: Categorization and complexity analysis. *Journal of Artificial Intelligence Research*, 22:143–174, November 2004.

C. V. Goldman and S. Zilberstein. Optimizing information exchange in cooperative multiagent systems. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 137–144, New York, NY, USA, 2003. ACM Press.

C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored MDPs. In *Advances in Neural Information Processing Systems (NIPS) 14*. The MIT Press, 2002a.

C. Guestrin. *Planning under uncertainty in complex structured environments*. PhD thesis, Computer Science Department, Stanford University, August 2003.

C. Guestrin, M. Lagoudakis, and R. Parr. Coordinated reinforcement learning. In *International Conference on Machine Learning (ICML)*, Sydney, Australia, July 2002b.

C. Guestrin, S. Venkataraman, and D. Koller. Context-specific multiagent coordination and planning with factored MDPs. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Edmonton, Canada, July 2002c.

E. A. Hansen, D. S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, San Jose, CA, 2004.

H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The robot world cup initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/Alife*, 1995.

J. R. Kok and N. Vlassis. Sparse cooperative Q-learning. In Russ Greiner and Dale Schuurmans, editors, *Proceedings of the International Conference on Machine Learning*, pages 481–488, Banff, Canada, July 2004. ACM.

J. R. Kok and N. Vlassis. Using the max-plus algorithm for multiagent decision making in coordination graphs. In *RoboCup-2005: Robot Soccer World Cup IX*, Osaka, Japan, July 2005.

J. R. Kok, M. T. J. Spaan, and N. Vlassis. Non-communicative multi-robot coordination in dynamic environments. *Robotics and Autonomous Systems*, 50(2-3):99–114, February 2005.

J. R. Kok. *Coordination and Learning in Cooperative Multiagent Systems*. PhD thesis, Faculty of Science, University of Amsterdam, 2006.

F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47:498–519, 2001.

V. Lesser, C. Ortiz, and M. Tambe. *Distributed sensor nets: A multiagent perspective*. Kluwer academic publishers, 2003.

H.-A. Loeliger. An introduction to factor graphs. *IEEE Signal Processing Magazine*, pages 28–41, January 2004.

C. C. Moallemi and B. Van Roy. Distributed optimization in adaptive networks. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems (NIPS) 16*. MIT Press, Cambridge, MA, 2004.

P. Jay Modi, W-M. Shen, M. Tambe, and M. Yokoo. ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.

K. Murphy, Y. Weiss, and M. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, Stockholm, Sweden, 1999.

A. Y. Ng, H. Jin Kim, M. Jordan, and S. Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS) 16*, 2004.

L. E. Parker. Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous Robots*, 12(3):231–255, 2002.

J. Pearl. *Probabilistic reasoning in intelligent systems*. Morgan Kaufman, San Mateo, 1988.

L. Peshkin, K.-E. Kim, N. Meuleau, and L. P. Kaelbling. Learning to cooperate via policy search. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, pages 489–496. Morgan Kaufmann Publishers, 2000.

M. L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. Wiley, New York, 1994.

D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16: 389–423, 2002.

J. Schneider, W.-K. Wong, A. Moore, and M. Riedmiller. Distributed value functions. In *International Conference on Machine Learning (ICML)*, Bled, Slovenia, 1999.

S. Sen, M. Sekaran, and J. Hale. Learning to coordinate without sharing information. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Seattle, WA, 1994.

L. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39:1095–1100, 1953.

P. Stone, R. S. Sutton, and G. Kuhlmann. Reinforcement learning for RoboCup-soccer keepaway. *Adaptive Behavior*, 13(3):165–188, 2005.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 1998.

K. Sycara. Multiagent systems. *AI Magazine*, 19(2):79–92, 1998.

M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *International Conference on Machine Learning (ICML)*, Amherst, MA, 1993.

G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), March 1995.

N. Vlassis. A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam, September 2003.

N, Vlassis, R. Elhorst, and J. R. Kok. Anytime algorithms for multiagent decision making using coordination graphs. In *Proceedings of the International Conference on Systems, Man, and Cybernetics (SMC)*, The Hague, The Netherlands, October 2004.

M. J. Wainwright, T. S. Jaakkola, and A. S. Willsky. Tree consistency and bounds on the performance of the max-product algorithm and its generalizations. *Statistics and Computing*, 14:143–166, April 2004.

C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

G. Weiss, editor. *Multiagent systems: A modern approach to distributed artificial intelligence*. MIT Press, 1999.

J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. In *Exploring Artificial Intelligence in the New Millennium*, chapter 8, pages 239–269. Morgan Kaufmann Publishers Inc., January 2003.

M. Yokoo and E. H. Durfee. Distributed constraint optimization as a formal model of partially adversarial cooperation. Technical Report CSE-TR-101-91, University of Michigan, Ann Arbor, MI 48109, 1991.

N. Lianwen Zhang and D. Poole. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328, 1996.