Multiagent Reinforcement Learning and Self-Organization in a Network of Agents[*]

Sherief Abdallah
British University in Dubai
Dubai, United Arab Emirates
sherief.abdallah@buid.ac.ae

Victor Lesser
University of Massachusetts
Amherst, MA
lesser@cs.umass.edu

ABSTRACT

To cope with large scale, agents are usually organized in a network such that an agent interacts only with its immediate neighbors in the network. Reinforcement learning techniques have been commonly used to optimize agents local policies in such a network because they require little domain knowledge and can be fully distributed. However, all of the previous work assumed the underlying network was fixed throughout the learning process. This assumption was important because the underlying network defines the learning context of each agent. In particular, the set of actions and the state space for each agent is defined in terms of the agent's neighbors. If agents dynamically change the underlying network structure (also called self-organizing) during learning, then one needs a mechanism for transferring what agents have learned so far before (in the old network structure) to their new learning context (in the new network structure).

In this work we develop a novel self-organization mechanism that not only allows agents to self-organize the underlying network during the learning process, but also uses information from learning to guide the self-organization process. Consequently, our work is the first to study this interaction between learning and self-organization. Our self-organization mechanism uses heuristics to transfer the learned knowledge across the different steps of self-organization. We also present a more restricted version of our mechanism that is computationally less expensive and still achieves good performance. We use a simplified version of the distributed task allocation domain as our case study. Experimental results verify the stability of our approach and show a monotonic improvement in the performance of the learning process due to self-organization.

[*]This material is based upon work supported by the National Science Foundation Engineering Research Centers Program under NSF Award No. EEC-0313747. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence

General Terms

Algorithms, Experimentation

Keywords

Reinforcement Learning, Multiagent Systems, Reoganization, Network

1. INTRODUCTION

Many problems that an agent faces in a multiagent system can be formulated as decision making problems, where an agent needs to decide which action to execute in order to maximize the agent's objective function. Optimizing decision making in multiagent systems is challenging because each agent needs to take into account other agents in the system: what agents are available and what are their current state. As the number of agents grows, a common approach, in order to cope with scale, is to organize agents into an *overlay network*, where each agent interacts only with its immediate neighbors. Therefore, the *context* within which each agent optimizes its decision is defined in terms of each agent's neighbors. This context consists of an agent's state (which should reflect neighbors' states) and what actions are available to the agent (which should include at least an action for each neighbor).

Multiagent Reinforcement Learning (MARL) is a common approach for solving multiagent decision making problems. It allows agents to dynamically adapt to changes in the environment, while requiring minimum domain knowledge. Using MARL, each agent starts with an arbitrary policy[1] that gradually improves as agents interact with each other and with the environment. Several MARL algorithms have been applied to a network of agents [4, 7, 1]. However, all of the previous work assumed the underlying network was fixed throughout the learning process. This assumption was important because it keeps the decision context of each agent fixed as well. If agents can dynamically change the underlying network structure (also called self-organizing) during learning, then one needs a mechanism for transferring what agents have learned so far (in the old network structure) to

[1]As will be described shortly, a policy is a solution to the decision making problem that specifies which action to execute in every state.

their new learning context (in the new network structure). Otherwise, agents would need to start learning from scratch every time the network structure changes.

The main contribution of this work is developing a novel self-organization mechanism that not only allows agents to self-organize the underlying network during the learning process, but also uses information from learning to guide the self-organization process. In particular, using our mechanism, an agent can add and remove other agents to its neighborhood while still learning. The mechanism uses heuristics to transfer the learned knowledge across different steps of self-organization as we will describe shortly. While several algorithms were developed for self-organization [8, 5], they all assumed agents' local policies were fixed (i.e. no learning). Consequently, our work is the first to study and analyze the interaction between learning and self-organization. Furthermore, because the operations of adapting the network are computationally expensive, we also present a more restricted version of our mechanism that is computationally less expensive and still achieve good performance. We use a simplified version of the distributed task allocation domain as our case study. Experimental results verify the stability and effectiveness of our approach in a network of up to 100 learning agents.

The paper is organized as follows. Section 2 describes the distributed task allocation domain, which we will use throughout the document as our case study. Section 3 briefly reviews previous MARL algorithms and describes the learning algorithm we use in conjunction with our self-organizing mechanism. Section 4 describes our self-organizing mechanism. Section 5 presents and discusses our experimental results. Section 6 reviews the previous work. Finally, Section 7 concludes and proposes possible future extensions.

2. CASE STUDY: DISTRIBUTED TASK ALLOCATION PROBLEM (DTAP)

We use a simplified version of the distributed task allocation domain (DTAP) [1], where the goal of the multiagent system is to assign tasks to agents such that the service time of each task is minimized. Agents interact via communication messages (there are three types of messages that are described in Section 4). Communication delay between two agents is proportional to the Euclidean distance between them, one time unit per distance unit (each agent has a physical location). Each time unit, agents make decisions regarding all task requests received during this time unit. For each task, the agent can either execute the task locally or send the task to a neighboring agent. If the agent decides to execute the task locally, the agent adds the task to its local queue, where tasks are executed on a first come first serve basis, with unlimited queue length.

Agent i execute tasks with rate μ_i tasks per time unit, and receives tasks from the environment with arrival rate λ_i tasks per time unit. Both λ and μ satisfy the condition $\sum_i \lambda_i < \sum_i \mu_i$ in order to ensure system stability. The main goal of DTAP is to reduce the total service time, averaged over tasks, $ATST = \frac{\sum_{T \in \overline{T}_\tau} TST(T)}{|\overline{T}_\tau|}$, where $\overline{T}_\tau$ is the set of task requests received during a time period τ and TST is the total time a task spends in the system. TST consists of the time for routing a task request through the network, the time a task request spends in the local queue, and the time of actually executing the task. Because both learning and self-organization contribute to any improvement to $ATST$, we will also measure the average number of hops a task needs to go through before an agent executes the task locally, which is directly affected by self-organization.

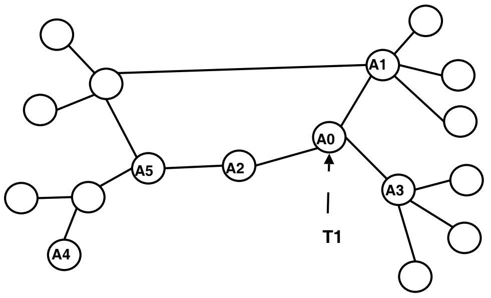

Figure 1: Task allocation using a network of agents.

For illustration, consider the example scenario depicted in Figure 1. Agent A0 receives task T1, which can be executed by any of the agents A0, A1, A2, A3, and A4. All agents other than agent A4 are overloaded, and therefore the best option for agent A0 is to forward task T1 to agent A2 which in turn forwards task T1 to its left neighbor (A5) until task T1 reaches agent A4. Although agent A0 does not know that A4 is under-loaded (because agent A0 interacts only with its immediate neighbors), agent A0 will eventually learn (through experience and interaction with its neighbors) that sending task T1 to agent A2 is the best action without even knowing that agent A4 exists.

Now suppose agent A0 sends most of its tasks to agent A2, while agent A2 executes half of the incoming tasks locally and sends the other half to its neighbor A5. One would expect performance to improve if A0 adds A5 as one of its own neighbors, because this reduces overhead. Similarly, if A0 rarely sends any request to its neighbor A3, then removing neighbor A3 from A0's neighbors will reduce the computational overhead associated with taking agent A3 into account whenever A0 is making a decision.

Our mechanism, allows agents to dynamically add and remove neighbors so that in the above example agent A0 becomes directly connected to A4, while removing unnecessary neighbors. The main difficulty of adding and removing neighbors, however, is that it changes the decision context of an agent. In the above example, agent A0 may know very well how to interact with its old neighbors due to a long history of interactions. On the other hand, A0 has no experience with the new neighbor A5. Even worse, not only does A0 need to learn about A5, but also all its previous experience may not be relevant after adding A5 (because A5 was not part of the state in the previous experience). Our mechanism, as we describe in Section 4 uses heuristics to retain most of the previous experience throughout the self-organization process. For simplicity, we assume each task type has a corresponding organization, i.e. each agent has multiple sets of neighbors, a set for each task type. The following section briefly reviews the previous work in MARL and describe the learning algorithm we use.

3. MULTIAGENT REINFORCEMENT LEARNING, MARL

When RL techniques are applied in a distributed multiagent system, the learning agents may fail to converge due to lack of synchronization [3]. Several MARL algorithms have been developed to address this issue [3, 2, 1], with theoretical convergence guarantees that do not hold for more than two agents. Despite using different heuristics to bias learning towards stable policies, most of these algorithms maintain and update the same two data structures for each agent i: action values, Q_i, and the policy π_i. Both data structures are represented using two dimensional tables $|S_i|$ rows $\times |A_i|$ columns, where S is the set of states encountered by agent i and A_i is the set of actions that agent i can execute.[2] The cell $Q_i(s, a)$ stores the reward agent i expects if it executes action a at state s. The cell $\pi_i(s, a)$ stores the probability that agent i will execute action a at state s. Together, Q and π encapsulates what an agent has learned so far. The main idea of most of these algorithms is to compute an approximate gradient of Q, then use that gradient to update π, with small step η.

The advantage of using this gradient ascent approach is that agents can learn stochastic policies, which is necessary for most of the convergence guarantees. Algorithm 1 describes the Weighted Policy Learner (WPL) algorithm [1], which we have chosen as the accompanying learning algorithm for our self-organizing mechanism. It should be noted, however, that our mechanism does not depend on the accompanying learning algorithm. In fact, the interaction between WPL and our self-organizing mechanism is encapsulated through the Q and π data structures, which are common to learning algorithms other than WPL.

WPL achieves convergence using an intuitive idea: slow down learning when moving away from a stable policy[3] and speedup learning when moving towards the stable policy. In that respect, the idea has similarity with the Win or Lose Fast heuristic (WoLF) [3], but the WPL algorithm is more intuitive and achieves higher performance than algorithms using WoLF.

Algorithm 1: WPL(state s, action a)

```
begin
    Let $r \leftarrow$ the reward of reaching state s
    Update $Q(s', a')$ using r
    $s' \leftarrow s$ and $a' \leftarrow a$
    $\widehat{r} \leftarrow$ total average reward $= \sum_{a \in A} \pi(s, a) Q(s, a)$.
    foreach action $a \in A$ do
        $\Delta(a) \leftarrow Q(s, a) - \widehat{r}$
        if $\Delta(a) > 0$ then $\Delta(a) \leftarrow \Delta(a)(1 - \pi(a))$
        else $\Delta(a) \leftarrow \Delta(a)(\pi(a))$
    end
    $\pi \leftarrow \pi + \eta\Delta$
end
```

4. SELF-ORGANIZATION

There are two basic operators for restructuring a network: adding a neighbor and removing a neighbor. A self-organizing mechanism would need to answer three questions: *which* neighbor to add or remove, *when* to stop adding or removing neighbors, and *how* to adjust π and Q (that encapsulate an agent's experience) to account for adding and removing a neighbor. The remainder of this section addresses the first two questions while the following section addresses the third question.

Algorithm 2 illustrates the decision process that takes place in each agent every cycle. The algorithm uses three types of messages. A REQUEST message $\langle i, T \rangle$ indicates a request from neighbor i to execute task T. An UPDATE message $\langle i, \tilde{S}_i \rangle$ indicates an update $\tilde{S}_i$ to the state feature corresponding to neighbor i An ORGANIZE message $\langle i, j \rangle$ indicates a self-organization proposal from neighbor i to add neighbor j.

Algorithm 2: Decision Making Algorithm

```
begin
    $MSGS \leftarrow$ messages received in this cycle.
    for each UPDATE message $\langle i, \tilde{S}_i \rangle \in MSGS$, update the
    current state, s.
    for each ORGANIZE message $\langle i, j \rangle \in MSGS$, call
    $processOrganize(\langle i, j \rangle)$
    for each REQUEST message $\langle n, T \rangle \in MSGS$ do begin
        Choose a neighbor $a$ randomly according to $\pi(s, .)$
        If $a$ is self then add $T$ to the local queue, otherwise
        forward the request to $a$.
        $sendUpdate()$
        $proposeOrganize($ $n$ $)$
    end
    learn( s, a )
end
```

The algorithm uses four functions: *learn*, *sendUpdate*, *proposeOrganize*, and *processOrganize*. Function *learn* encapsulates the accompanying learning algorithm (e.g. WPL). Function *sendUpdate* is responsible for maintaining the state of each agent updated via communicating UPDATE messages as follows. An agent's state is defined by the tuple $\langle \beta, \tilde{S}_0, \tilde{S}_1, ..., \tilde{S}_n \rangle$, where β is the rate of incoming requests and $\tilde{S}_i$ is a feature corresponding to neighbor i. We assume there exists an abstraction function that summarizes a neighboring agent state to an abstract state, which is then used as a feature. In the DTAP domain the abstract state of an agent i is approximated by the ATST for that agent, i.e. $\tilde{S}_i \simeq ATST_i = -\sum_k \pi_i(s, k) Q_i(s, k)$ (note that $Q_i(s, k)$ holds the expected reward of sending a task to neighbor k if at state s, and in the DTAP problem the reward = -ATST). In order to avoid excessive sending of UPDATE messages, each agent i keeps track of $\tilde{S}_i^j$, the last communicated abstract state to a neighbor j. An agent sends an UPDATE message to a neighbor j when the difference between its current state and the last communicated state to neighbor j exceeds certain threshold (we assume there exists a function that computes the difference between two agent states). More formally, Agent i sends an UPDATE message to neighbor j if and only if $|\tilde{S}_i^j - \tilde{S}_i| > \varphi$, where φ is a threshold and S_i is the current abstract state of agent i. It is therefore possible for an agent have an outdated abstract state of a neighbor.[4]

The two functions *processOrganize* (Algorithm 3) and *proposeOrganize* (Algorithm 4) encapsulate the self-organizing

[2]More approximate representations of Q and π are possible, but will make the transfer of learned knowledge (Section 4.1) more complicated.

[3]The stable policy is in fact a Nash Equilibrium [1]. We omit details for space reasons.

[4]This possibility is true even if $\varphi = 0$, because of communication delays.

mechanism. Function *proposeOrganize* chooses a neighbor to add or remove based on the policy averaged over states $= \sum_s P(s)\pi(s,a)$, where $P(s)$ is the probability of visiting state s (approximated by counting the number of visits for each state). We have tried several alternatives for choosing a neighbor, including choosing the most likely neighbor at the most likely state, i.e. $n^* \leftarrow argmax_a\{\pi(argmax_s\{P(s)\},a)\}$ (where $P(s)$ is the probability of reaching state s), and choosing a neighbor stochastically according to $\pi(s',a)$, where state s' is chosen stochastically as well according to $P(s)$. The strategy that *proposeOrganize* uses has outperformed both.

Instead of adding or removing neighbors deterministically, *processOrganize* and *proposeOrganize* do that stochastically with probabilities P_{add} and P_{remove} respectively. Intuitively, P_{remove}, should slowly increase as the number of neighbors increase in order to discourage the set of neighbors from growing indefinitely. We have used the simple form below in our analysis.

$$P_{remove} = \begin{cases} 0 & \text{if } |N_i| <= n_0 \\ \frac{1}{|N_i|-n_0} & \text{otherwise} \end{cases}$$

, where N_i is the set of neighbors of agent i and n_0 is a constant. Similar to P_{remove}, when agent i receives a REORGANIZE message, it should add a new neighbor with probability P_{add} that is inversely proportional to the current number of neighbors of agent i. We use $P_{add} = 1 - P_{remove}$. The following lemma proves that although our self-organizing mechanism is stochastic, the number of neighbors per agent is bounded.

Algorithm 3: *proposeOrganize*(requesting neighbor k)

```
begin
    with probability $P_O$ do begin
        $n^* \leftarrow argmax_a \sum_s P(s)\pi(s,a)$
        send an ORGANIZE message, with $n^*$ as its data
        field, to neighbor $k$.
        with probability $P_{remove}$ do begin
            $n^- \leftarrow argmin_a \sum_s P(s)\pi(s,a)$
            Remove $n^-$ from list of neighbors and adjust $Q$
            and $\pi$
            details in the following section
        end
    end
end
```

Algorithm 4: *processOrganize*($\langle$ existing neighbor n^e, new neighbor n^+ $\rangle$)

```
begin
    with probability $P_{add}$ do begin
        if $n^+ \notin$ list of neighbors then Add neighbor $n^+$ to
        list of neighbors and adjust $Q$ and $\pi$
    end
end
```

LEMMA 1. *For every agent i, the average number of neighbors $|N_i| \leq n_0 + 2$*

PROOF. Because the probability of adding a neighbor decreases as the number of neighbors increase, while the probability of removing a neighbor increases, there must be a point in between where the rate of adding a neighbor equals the rate of removing a neighbor. If $|N_i| \leq n_0$, the probability of adding a neighbor is 1 and the probability of removing a neighbor is 0, so $|N_i|$ must be greater than n_0 The rate of removing a neighbor equals $\eta_i P_0 P_{remove} = \frac{\eta_i P_0}{|N_i|-n_0}$, where η_i is the rate of receiving REQUEST messages by agent i per time unit (note, from Algorithm 2, that the *proposeOrganize* method is executed for each received request message).

The rate of adding a neighbor is less or equals the rate of receiving ORGANIZE messages multiplied by P_{add}. It can be less than the rate of received ORGANIZE messages because some proposals are redundant (have already been added before). The rate of receiving ORGANIZE messages from neighbor j= rate of REQUEST messages agent i sends to that neighbor multiplied by $P_0 = P_0\eta_i \sum_s P(s)\pi(s,a^j)$, where $P(s)$ is the probablity of visiting state s. The rate of adding a neighbor is therefore $\leq P_{add} \sum_j P_0\eta_i \sum_s P(s)\pi(s,a^j)$ $= P_0 P_{add}\eta_i \sum_s P(s) \sum_j \pi(s,a^j) = P_0 P_{add}\eta_i$. In steady state, the rate of adding a neighbor equals the rate of removing a neighbor, $\frac{1}{|N_i|-n_0} \leq 1 - \frac{1}{|N_i|-n_0}$, which is only true if $|N_i| \leq n_0 + 2$. □

It should be noted that P_O must be low enough in order to allow learning to converge. This is important because self-organization uses π in choosing which neighbor to add and remove, then after adding or removing a neighbor, both Q and π are adjusted approximately (as the following section illustrates) and need to be optimized further by learning. Lowering P_O is also necessary from a practical point of view because adding and removing a neighbor is an expensive operation. Lowering P_O too much, however, leads to very slow self-organization and therefore slower improvement in performance.

Executing the function *proposeOrganize* iff a REQUEST message is received achieves several desirable properties. First, the rate of receiving ORGANIZE messages from a neighbor n^e is proprtional to how important n^e is (so for example, if a neighbor is rarely used, the rate of accepting a proposal to add a neighbor from that neighbor should be low). Similarly, the rate of sending an ORGANIZE message to a neighbor is proportional to the rate of receiving REQUEST message from that neighbor. The following section discusses in further detail how Q and π are adjusted when an agent adds or removes a neighbor.

4.1 Adjusting Q and π

When a restructuring operator is executed (adding or removing a neighbor), the learned knowledge (the Q and the π tables) need to change in order to reflect the new state features and action set. Resetting both P and Q whenever an agent executes a restructuring operator, while valid, wastes valuable experience and learning the agent has already gone through. Instead, we take advantage of the natural factorization of the state and corresponding actions in order to retain as much as possible of π and Q. For convenience, we will use $Q(s,.)$ to refer to a row, $Q(s,a)$ to refer to a particular cell, and $Q(s,a_i...a_j)$ to refer to partial row in the Q table. A similar notation is used for π.

When the self-organizing mechanism removes a neighbor, the mechanism also removes the action and state feature associated with that neighbor. Removing an action results in removing the corresponding table column in both the Q and the π data structures. Unlike Q, however, π needs to be normalized after removing a colummn, because the policy for any state must always sum to 1. Removing a state feature means that states that were originally distinguishable in the old state space are now indistinguishable in the new state space and consequently states will be clustered into fewer number of states. The question is how to merge the rows corresponding to each state cluster in π and Q. A simple way is to choose one of the rows at random. However, this does not take into account that some states may be more important than others. Instead, we weigh each row by the frequency of visiting that state and then sum all the rows for both Q and π.

When a neighbor is added, an action is added to the set of actions and a new feature is added to the state features. Adding an action results in a new column in both Q and π data structures. Adding a feature results in expanding the state space, where each state is expanded to a set of states that only differ in the new neighbor's state. To transfer what an agent has previously learned in the old state space to the newly expanded state space, we need a mapping from the old Q and π tables (defined over the old state space) to new Q' and π' (defined over the new state space). Intuitively, the idea is to treat the newly added neighbor as an identical twin to the existing neighbor that has proposed this self-organizing operator (i.e. the sender of the ORGANIZE message) as identical twins. In other words, their action values and corresponding policy are equal to one another. More formally, let n^e be the existing neighbor that proposed adding a new neighbor n^+ and let $\langle \underline{s}^{-e}, s^e \rangle$ be a factored old state, where s^e is the feature corresponding to neighbor n^e and $\underline{s}^{-e}$ is all the other features of the old state. Similarly, let a^e be the action corresponding to neighbor e and $\underline{a}^{-e}$ be the list of all other actions (not including the action corresponding to the new neighbor a^+). For every two old states $\langle \underline{s}^{-e}, s_1^e \rangle$ and $\langle \underline{s}^{-e}, s_2^e \rangle$ (i.e. the two states only differ in the feature value corresponding to neighbor n^e) we generate four new states and initialize the next Q and π entries as follows.

- $\langle \underline{s}^{-e}, s_1^e, s_1^e \rangle$:
  $Q'(\langle \underline{s}^{-e}, s_1^e, s_1^e \rangle, .) = \langle Q(\langle \underline{s}^{-e}, s_1^e \rangle, .), Q(\langle \underline{s}^{-e}, s_1^e \rangle, a_e) \rangle$
  and $\pi'(\langle \underline{s}^{-e}, s_1^e, s_1^e \rangle, .) = \langle \pi(\langle \underline{s}^{-e}, s_1^e \rangle, \underline{a}^{-e}),$
  $\frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, a_e)}{2}, \frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, a_e)}{2} \rangle$

- $\langle \underline{s}^{-e}, s_1^e, s_2^e \rangle$:
  $Q'(\langle \underline{s}^{-e}, s_1^e, s_2^e \rangle, .) = \langle Q(\langle \underline{s}^{-e}, s_1^e \rangle, .), Q(\langle \underline{s}^{-e}, s_2^e \rangle, a_e) \rangle$
  and $\pi'(\langle \underline{s}^{-e}, s_1^e, s_2^e \rangle, .) = \langle \frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, \underline{a}^{-e}) + \pi(\langle \underline{s}^{-e}, s_2^e \rangle, \underline{a}^{-e})}{2},$
  $\frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, a_e)}{2}, \frac{\pi(\langle \underline{s}^{-e}, s_2^e \rangle, a_e)}{2} \rangle$

- $\langle \underline{s}^{-e}, s_2^e, s_1^e \rangle$:
  $Q'(\langle \underline{s}^{-e}, s_2^e, s_1^e \rangle, .) = \langle Q(\langle \underline{s}^{-e}, s_2^e \rangle, .), Q(\langle \underline{s}^{-e}, s_1^e \rangle, a_e) \rangle$
  and $\pi'(\langle \underline{s}^{-e}, s_1^e, s_2^e \rangle, .) = \langle \frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, \underline{a}^{-e}) + \pi(\langle \underline{s}^{-e}, s_2^e \rangle, \underline{a}^{-e})}{2},$
  $\frac{\pi(\langle \underline{s}^{-e}, s_2^e \rangle, a_e)}{2}, \frac{\pi(\langle \underline{s}^{-e}, s_1^e \rangle, a_e)}{2} \rangle$

- $\langle \underline{s}^{-e}, s_2^e, s_2^e \rangle$:
  $Q'(\langle \underline{s}^{-e}, s_2^e, s_2^e \rangle, .) = \langle Q(\langle \underline{s}^{-e}, s_2^e \rangle, .), Q(\langle \underline{s}^{-e}, s_2^e \rangle, a_e) \rangle$
  and $\pi'(\langle \underline{s}^{-e}, s_2^e, s_2^e \rangle, .) =$
  $\langle \pi(\langle \underline{s}^{-e}, s_2^e \rangle, \underline{a}^{-e}), \frac{\pi(\langle \underline{s}^{-e}, s_2^e \rangle, a_e)}{2}, \frac{\pi(\langle \underline{s}^{-e}, s_2^e \rangle, a_e)}{2} \rangle$

The procedure we have just described of maintaining π and Q when a neighbor is added or removed is expensive and not necessarily optimal. Despite its suboptimality, however, it serves as a good approximation that allows learning to pursue and tune Q and π further, without major disruptions. Our experimental results verify that indeed the performance remain monotonically increasing as self-organization and learning take place simultaneously. In the following section we describe a more restricted version of our mechanism that uses only one reogranization operator: replace one neighbor with another.

4.2 Restricted Self-Organizing Mechanism

The main intuition behind this restricted self-organization mechanism is to bypass the "middleman" by using one restructuring operator, replace a neighbor, instead of the previous two restructuring operators, add and remove a neighbor. If an agent a_{over} is overloaded, most of the time it will forward requests to its neighbors instead of executing tasks locally. It is therefore intuitive to try to bypass this overloaded agent. The mechanism uses identical *proposeOrganize* and *processOrganize* but without removing a neighor in *proposeOrganize* and instead of adding a neighbor in *processOrganize*, the new neighbor replaces the proposing neighbor.

We assume that action values Q (and therefore the policy π) will not significantly change if the network changes slowly using the replace-neighbor operator. Based on that assumption, when an agent replaces neighbor n_{old} with neighbor n_{new}, the agent modifies neither the corresponding action value $Q(., a)$ nor the policy $\pi(., a)$. Instead, the agent just changes the association of the corresponding action a from the old neighbor n_{old} to the new neighbor n_{new}. Experimental results verify that learning converges smoothly using this strategy in face of self-organization. The main limitation of this approach is that it preserves node out-degrees, therefore limiting the space of reachable policies. This limitation, however, makes our assumption of keeping both action values (Q) and policy (π) unchanged more intuitive.

5. EXPERIMENTAL RESULTS

The evaluation includes three measurements: the average total service time (ATST), the average number of REQUEST messages per request (AREQ), and the average number of UPDATE messages per request (AUPD). ATST represents the overall system performance and we use it to verify the stability (convergence) of our approach by showing monotonic improvement in ATST as agents gain more experience. AREQ reflects the average number of hops (intermediate agents) a task request goes through before being executed and is therefore the main measurement that reflects the benefit of self-organizing. AUPD indicates the overhead for maintaining the state and therefore should be kept at a minimum.

We have experimented with uniform two dimensional grid networks of different sizes: 2x2, 4x4, 6x6, and 10x10 agents. The results we have obtained are similar and for brevity we only report here the results for the 10x10 grid. For each simulation run, ATST, AREQ, and AUPD are computed every 5000 times steps to measure performance improvement as agents learn. Results are then averaged over 10 simulation runs and the variance is computed across the runs. The learning rate (η in the WPL algorithm) is set to 0.0001.

For simplicity, we assume that there is only one task type that is not decomposable and that all agents have same execution rate, $\forall i : \mu_i = \mu = 0.1$. However, task arrival rates, λ_i, differ from one agent to another (leading to unbalanced load). The goal of our system, therefore, is to learn how to route tasks effectively in order to balance the load (minimize ATST), using minimum number of hops (minimize AREQ) and low commuincation overhead (minimize AUPD). Tasks arrive according to two different patterns of load:

Boundary load : where the 36 nodes on the boundary receive tasks with $\lambda = 0.25$. Other nodes receive no tasks.

Center load : where the 16 nodes in the centric 4x4 grid have task arrival rate $\lambda = 0.5$. Other nodes receive no tasks.

We have tried different values of n_0. Larger n_0 leads to better performance but longer time to stabilize (agents need to learn about more neighbors). The results shown here use $n_0 = 5$ for center load and $n_0 = 3$ for boundary load. Table 1 and Table 2. summarize the results of the following figures (the table shows the different measures after learning and self-organization have stabilized). Figure 2 plots AREQ for different values of P_O: 0 (no re-organization), 0.00005, and 0.0002. As expected, AREQ drops significantly due to self-organization. Figure 3 shows the monotonic decrease in ATST. More importantly, self-organization leads to faster convergence. While surprising at first, it is actually intuitive because self-organization allows agents that are far from incoming requests to receive more requests quickly. Therefore, these initially distant agents gain more experience and learn faster.

P_0	ATST	AREQ	AUPD
0 (fixed organiz.)	45 ± 0.37	4.86 ± 0.05	0.23 ± 0.02
0.00005	40.75 ± 0.5	3.25 ± 0.05	0.32 ± 0.03
0.0002	38.69 ± 1.15	3.00 ± 0.07	0.36 ± 0.02

Table 1: Different measurements for different P_0 values under center-load.

P_0	ATST	AREQ	AUPD
0 (fixed organiz.)	50.4 ± 1.1	3.91 ± 0.11	0.24 ± 0.01
0.00005	47.5 ± 0.71	3.1 ± 0.05	0.26 ± 0.01
0.0002	45.34 ± 0.67	2.88 ± 0.05	0.28 ± 0.01

Table 2: Different measurements for different P_0 values under boundary-load.

The downside of self-organization is that it may lead to a higher AUPD as Figure 4 shows. The reason is that intermediate agents (which are bypassed by self-organization) hide some of the changes in other agents' states by aggregation. For example, if one neighbor of agent a gets overloaded while another neighbor is underloaded, then agent a's load (and therefore its aggregated state) may remain unchanged as the changes in the load of its neighbors even out. However, AUPD (the overhead) is still much lower than AREQ.

Figures 5 and 6 illustrate the self-organization process when $P_O = 0.0002$ for both types of load. For better visualization, the diameter of each node is proportional to the load on that node (i.e. its local queue length). Initially,

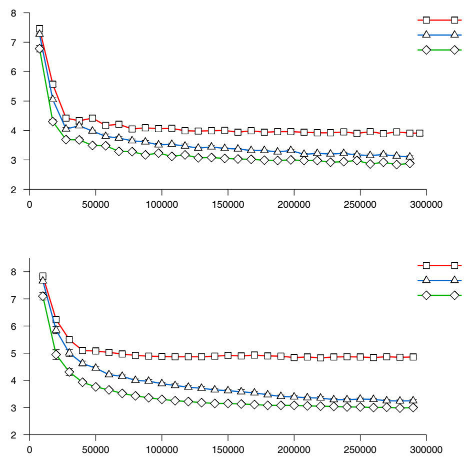

Figure 2: AREQ in 10x10 grid for different values of P_O, boundary load: top, center load: bottom.

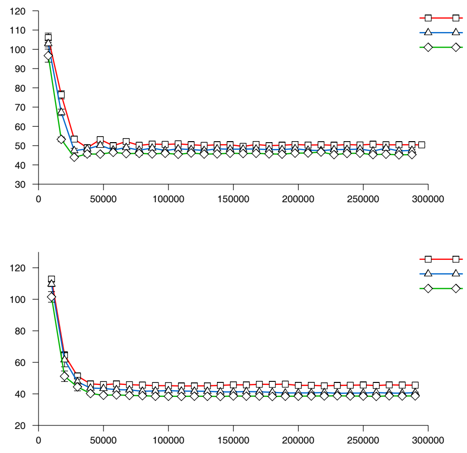

Figure 3: ATST in 10x10 grid for different values of P_O, boundary load: top, center load: bottom.

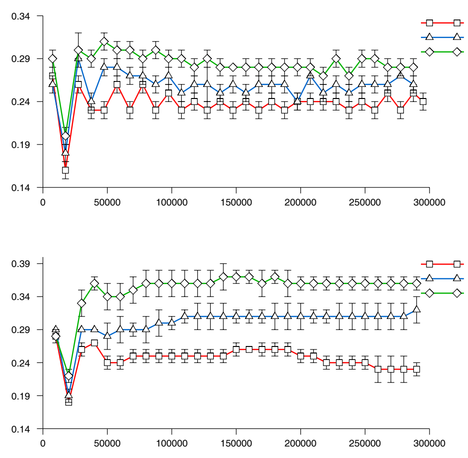

Figure 4: AUPD in 10x10 grid for different values of P_O, boundary load: top, center load: bottom.

agents that receive the initial task requests are overloaded, while other agents are underloaded. As learning and self-organization take place, the load becomes more balanced and the path between heavy loaded agents and underloaded agents gets shorter. In particular, more edges are pointing outward in the case of the center load, while more edges are pointing inward in the case of the boundary load.

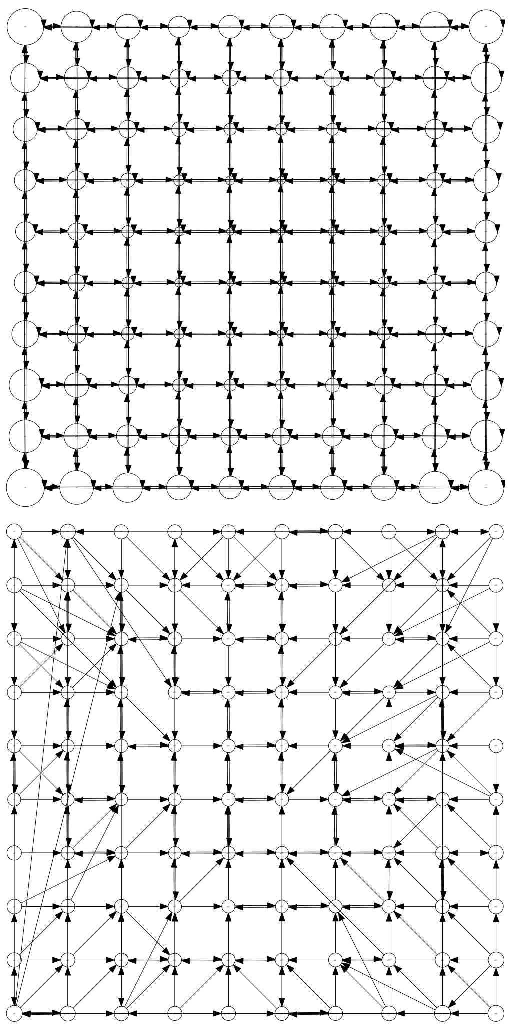

Figure 5: Self-Organization when load on boundary, at time 10,000 (left) and 290,000 (right)

As we have discussed previously, the main limitation of our mechanism is that it is computationally expensive to update the learning data structures Q and π when adding and removing a neighbor. Figure 7 compares our mechanism (CMPLX) against the more restricted version of it (SMPL), which only uses the replace-neighbor operator, for the center-load case. The restricted version initially performs better (converges faster), because it merges two self-organizing operations in one (similar to organizing at double the speed). However, eventually the more complex mechanism catch up and outperform. The difference is not significant, however, so the simpler mechanism provide a good compromise between optimiality and speed.

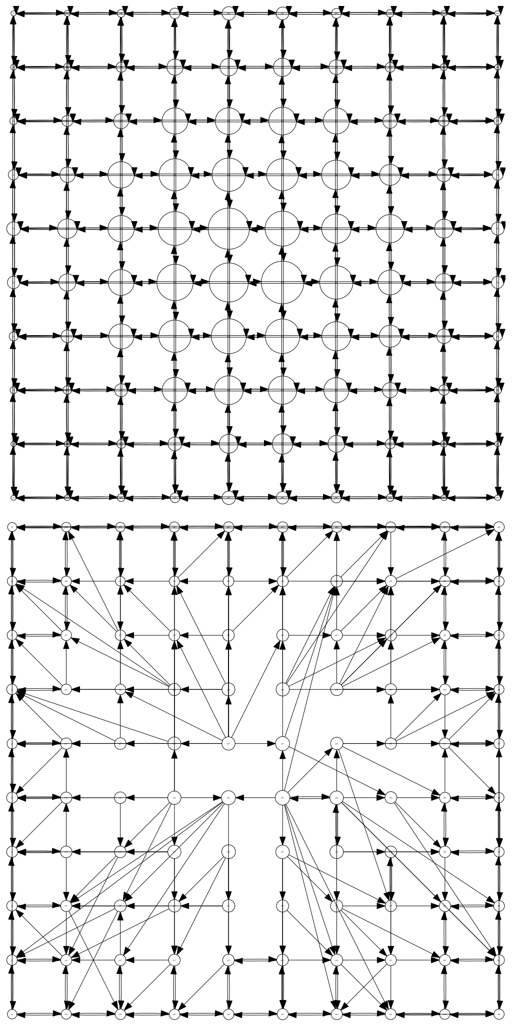

Figure 6: Self-Organization when load at center, at time 10,000 (left) and 290,000 (right)

6. RELATED WORK

Q-routing [4] pioneered the use of reinforcement learning to optimize packet routing in a network of agents. The work mapped the routing problem to a set of local decision problems (one per router), then used a simple reinforcement learning algorithm to learn a deterministic policy. Q-routing suffers from its inability to learn a non-deterministic policy.

The distributed gradient ascent policy search (DGAPS) [7] was evaluated in network routing. It outperformed other deterministic search policies (Q-routing, shortest path, best load, etc). Feedback occurred only after a packet is successfully delivered (or dropped due to a cycle), where an acknowledgment packet were passed to all nodes on the responsible path. While this may be more accurate, in a large network such feedback is slow. Our communication strategy reduces feedback considerably by exploiting the stability of the system. DGAPS also did not support self-organization.

Transferring learned knowledge has been recently proposed for the robocup domain [9] and has shown to reduce learning time. Aside from targeting a different domain, that work focused only on transferring the action value function

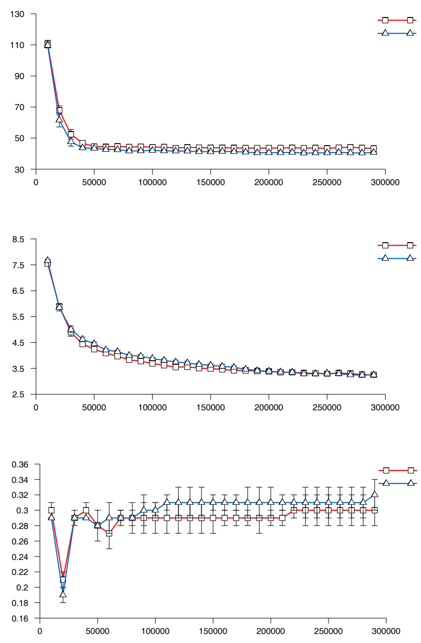

Figure 7: The self-organizing mechanism (CMPLX) against its restricted verion (SMPL): ATST (top), AREQ (middle), and AUPD (bottom).

Q, assuming policy π is deterministic. Our approach transfers both Q and π.

The work in [6] introduced a domain-independent organizational design representation that is able to model and predict the performance of agent organizations. However, optimizing agent policies was ignored and abstracted to simplify the analysis of organizations. The algorithm for finding an optimal organization was also centralized. Our self-organization mechanism is fully distributed (although it does not necessarily find the optimal organization). The work in [5] presented a self-organization algorithm that improved team formation in a network of agents. That algorithm implicitly relied on global synchronization (to ensure deterministic order among agent decisions), assumed tasks were globally broadcasted, and did not involve learning. Our mechanism does not make these assumptions and relies on information provided by learning. The work in [8] presented a self-organization algorithm based on negotiation. The algorithm was designed for a specific problem (distributed sensor networks) and did not involve reinforcement learning.

7. CONCLUSION AND FUTURE WORK

This paper proposes an integrated, scalable, and fully distributed framework for mlutiagent reinforcement learning in a network of agents. The framework consists of a learning algorithm and a self-organizing mechanism. The learning algorithm allows each agent to optimize its local policy and the self-organizing mechanism optimizes the underlying network. We have developed a novel self-organizing mechanism that uses information from learning to guide the restructure process (we have used an existing multiagent reinforcement learning algorithm, WPL). The mechanism executes while learning is taking place, which involves retaining the knowledge that each agent learns through the self-organization process. The framework has been applied to the distributed task allocation problem wher experimental results show significant improvement due to self-organization. Results also show the stability of our framework as agents performance monotonically increase as they gain more experience.

We are currecntly conducting experiments on wider variety of networks and load patterns. We are also considering different real world applications where our approach can be useful, such as adhoc routing. One of our future directions is to use learning inside our self-organizing mechanism (although our mechanism uses information learned by a learning algorithm, the mechanism itself is still a heuristic approach). Unlike the original learner that optimizes an agent's local decision, the self-organization learner will operate on a much slower pace while monitoring improvements of the long term system performance. Another future direction is analyzing the stability of our framework theoretically. This is more challenging than previous analysis in MARL because self-organization continuously changes the context of each agent. Guaranteeing convergence despite such dynamacity is challenging.

8. REFERENCES

[1] S. Abdallah and V. Lesser. Learning the task allocation game. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2006.

[2] M. Bowling. Convergence and no-regret in multiagent learning. In *Advances in Neural Information Processing Systems 17*, pages 209–216. MIT Press, Cambridge, MA, 2005.

[3] M. Bowling and M. Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.

[4] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems*, volume 6, pages 671–678. Morgan Kaufmann Publishers, Inc., 1994.

[5] M. E. Gaston and M. desJardins. Agent-organized networks for dynamic team formation. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 230–237, New York, NY, USA, 2005. ACM Press.

[6] B. Horling. *Quantitative Organizational Modeling and Design for Multi-Agent Systems*. PhD thesis, University of Massachusetts at Amherst, February 2006.

[7] L. Peshkin and V. Savova. Reinforcement learning for adaptive routing. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1825–1830, 2002.

[8] M. Sims, C. Goldman, and V. Lesser. Self-organization through bottom-up coalition formation. In *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS 2003)*, pages 867–874, Melbourne, AUS, July 2003. ACM Press.

[9] M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59, New York, NY, July 2005.