Leveraging Organizational Guidance Policies with Learning to Self-Tune Multiagent Systems

Scott J. Harmon, Scott A. DeLoach, Robby, and Doina Caragea
Multiagent and Cooperative Robotics Laboratory
Kansas State University
234 Nichols Hall, Manhattan, Kansas, USA
{harmon, sdeloach, robby, dcaragea}@ksu.edu

Abstract

As organization-based multiagent systems are applied to more complex problems, configuring and tuning the systems can become nearly as complex as the original problem a system was designed to solve. A robust system should be able to adapt. It should be able to self-configure and self-tune. To this end, we propose a method for self-tuning using the concept of guidance policies, that is policies that are designed to guide the system without sacrificing its flexibility. Guidance policies allow us to apply traditional learning techniques online without many of the drawbacks associated with a system falling into a local optimum. They also help simplify the learning process. We examine the impact of this learning on various multiagent systems.

1 Introduction

Organization-based multiagent systems engineering has been proposed as a way to design complex systems that adapt to their environment [3, 11]. Agents interact and can be given tasks depending on their individual capabilities. The system designer, however, may not have planned for every possible environment within which the system may be deployed. The agents themselves may exhibit particular properties that were not initially anticipated. As multiagent systems grow, configuration and tuning of these systems can become as complex as the problems they claim to solve.

A robust system should adapt to environments, recover from failure, and improve over time. In human organizations, policies evolve over time and are adapted to overcome failures. New policies are introduced to avoid unfavorable situations. A robust organization-based multiagent system should also be able to evolve its policies and introduce new ones to avoid undesirable situations.

Learning from mistakes is one of the most common learning methods in human society. Learning from mistakes allows one to improve performance over time, this is commonly referred to as *experience*. Experience can allow proper tuning and configuration of a complex multiagent system. In this paper, we implement this tuning and configuration through the mechanism of organizational guidance policies. Guidance policies are policies that have been defined to constrain the system, without limiting the system's flexibility [12]. These policies may be suspended if the system cannot achieve its goal with the policies in place. These policies, however, still limit the system and can be used to *guide* the system while still allowing the system to adapt to changes in the environment.

Applying learning in multiagent systems is not new. Many authors have explored applying various learning techniques in a multiagent context [13, 18, 19]. Most of these learning applications, however, have been limited to improving an agent's performance at some task, but not the overall organization's performance. Some consider the overall team performance, but not in the structure of modern organization-based multiagent systems.

There is good reason past literature has not explored much polices over the entire organization. Reasoning over an entire multiagent system is a momentous task [4]. Care must be taken to ensure that all learned policies do not negatively impact the organization, for example, putting the organization in a situation where it is impossible for it to complete its goal. Interactions between policies can be subtle and many hidden interactions may exist. Our use of guidance policies [12] helps mitigate these risks substantially, thus allowing the system to experiment with different policies without much risk to the viability of the system.

The main contributions of this paper are: (1) an organizational policy-based approach to self-tuning, (2) a generalized algorithm based on Q-Learning to implement the policy-based tuning, and (3) validation of our approach through a set of simulated multiagent systems, a common

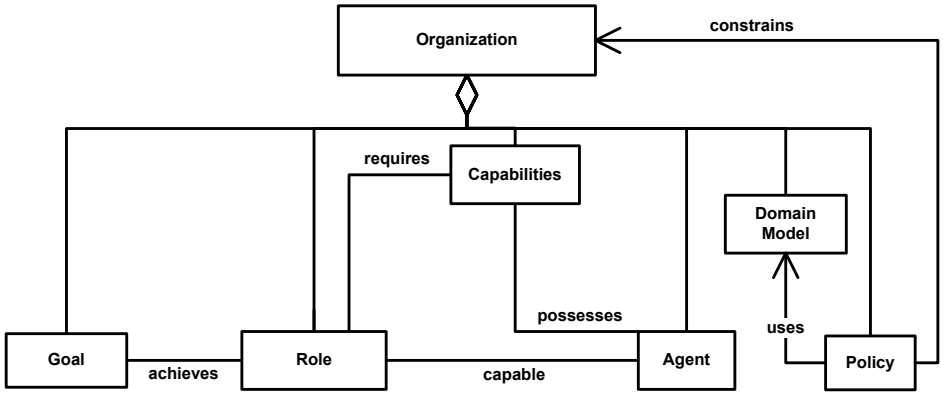

Figure 1. Organization Model for Adaptive Computational Systems.

approach for learning systems (e.g. [6, 14]).

The rest of the paper is organized as follows. In Section 2, we give some background on the models used in our multiagent systems. In Section 3, we present our policy learning algorithm. Section 4 presents and analyzes experimental results from applying our policy learning algorithm to three multiagent system examples. Related work is presented in Section 5. Section 6 concludes and Section 7 presents some ideas for future work.

2 Background

The multiagent systems model that we are using throughout the paper is called the Organization Model for Adaptive Computational Systems (OMACS) [11]. The basic OMACS metamodel is given in Figure 1. OMACS defines standard multiagent system entities and their relationships. Agents are *capable* of playing roles. Roles can *achieve* goals. Policies *constrain* the organization. The organization (which is comprised of agents), *assigns* agents to play specific roles in order to *achieve* specific goals. The Goal Model for Dynamic Systems (GMoDS) [17] is used to model our goal structures. GMoDS allows for such things as AND/OR decomposition of goals, as well as, *precedence* between goals and *triggers* (i.e. while trying to achieve a particular goal, an event may cause another goal to become active) between goals.

Policies (or norms) have been used in multiagent system engineering for some time. Various languages, frameworks, enforcement and checking mechanisms have been used [2, 5, 21, 22]. Taking a model checking perspective (e.g. [23]), we say that policies *restrict* the behavior of multiagent systems. We view the multiagent system as a set of states. Policies are rules designed to restrict this set of states. This restriction may happen at design time or at runtime. Guidance policies, while they do not sacrifice the flexibility of a multiagent system, still restrict their behaviors [12]. We use (guidance) policies as formal rules that are applied to our system. This is consistent with usage in formalizations such as KAoS [22] and PONDER [8].

Guidance policies are a trace-based formalization of policies 'that need not always be followed'. They must be followed when the system can still progress toward achieving its goal. If the system cannot continue with the guidance polices, they may be temporarily suspended. The guidance policies may be arranged in a *more-important-than* relation, creating a set of lattices. The policies are suspended from least-important to most-important. Thus it is possible to have conflicting guidance policies and yet still have a valid and viable system.

3 Self-Tuning Mechanism

An overview of how the learning takes place in our systems is given in Figure 2. The system first checks the goals that are available to be worked on, these goals come from GMoDS using the rules of precedence and trigger notations and support the overall goal of the system. Assignments of goals and the roles that can achieve them are made to the agents. The agents then indicate achievement failure or success. If an agent fails to achieve a goal, policies are learned over the current state knowledge. This cycles until either the system cannot achieve the main goal (system failure), or the system achieves the main goal (system success). Agents may fail to achieve a goal due to some aspect of the goal or due to some changes in the environment. The failure may be intermittent and random.

The aim of our learning algorithm is to discover new guidance policies in order to avoid 'bad states'. Thus we will be generating only negative authorization policies. While the precise definition of a *bad state* is domain specific, we assume that the set of bad states is a possibly

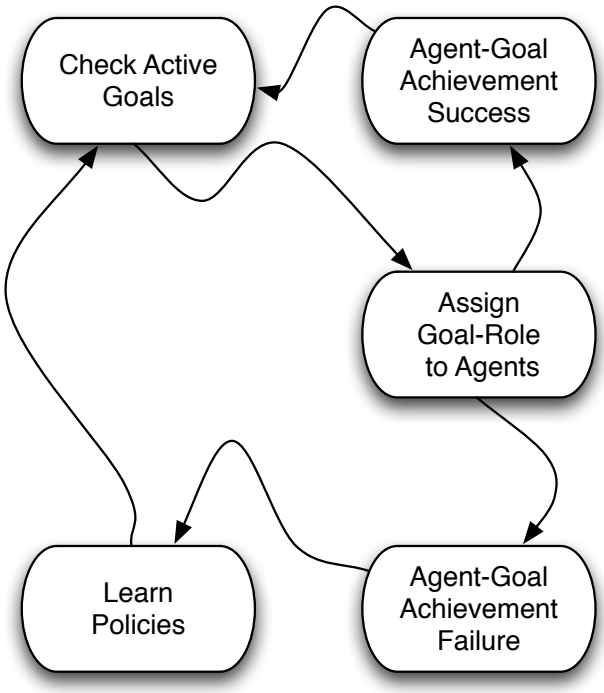

Figure 2. Learning Integration.

empty subset of all the states in the system. The remaining states are *good states*. Thus we have $S = S_B \cup S_G$ and $S_B \cap S_G = \emptyset$, where S is the set of all states in the system, S_G is the set of all good states in the system, and S_B is the set of all bad states in the system. Generally, bad states are states of the system that should be avoided. We use a scoring mechanism to determine which states are considered bad. The score of a state is domain specific, however, for our experiments we used $score(S_i) = \frac{1}{1+F_i}$, where F_i is the number of agent goal achievement failures thus far in state S_i.

To actually generalize and not simply memorize, our learning algorithm should derive policies that apply to more than one state through generalization. Thus, the learner attempts to discover the actual cause for the bad state so that it may avoid the cause and not simply avoid a single bad state.

Our learning algorithm takes a Q-Learning [24] approach. The learner keeps track of both *bad* and *good* actions (transitions) given a state.

Bad actions. *Bad actions given a state are defined as actions that result in a state that has a score lower than the state in which the action took place.*

Good actions. *Good actions given a state are defined as actions that result in a state that has a score equal to or greater than the score of the state in which the action took place.*

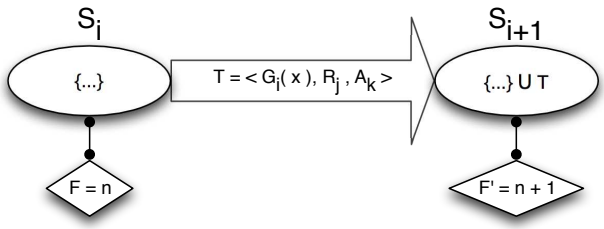

Figure 3. State and Action Transition.

The learner can then generate a set of policies that will avoid the bad states (by avoiding the bad action leading to this state). In the context of the experiments presented in this paper, the actions considered are *agent goal assignments*.

Agent goal assignment. *An agent goal assignment is defined as a tuple, $\langle G_i(x), R_j, A_k \rangle$, containing a parametrized goal $G_i(x)$, a role R_j, and an agent A_k. The goal's parameter x may be any domain specific specialization of the goal G_i given at the time the goal is dispatched (triggered) to the system.*

Figure 3 shows an example state transition. In this example, state S_i is transitioning to state S_{i+1} via assignment T. In state S_{i+1} a failure occurs, thus $F' = F + 1$. The state is then given a score.

Generalization of the policies is done over the state. For each action leading to a bad state, the algorithm first checks to see if we already have a policy covering this action and the pre-state (state leading to the bad state), if so, nothing needs to be done for this bad action, otherwise we generate a policy for it. The algorithm generalizes the state by considering *matchings*.

Matching. *A matching is a binary relation between a substate and a state. $B_k \prec S_i$, means that B_k matches S_i.*

Intuitively, a matching occurs between a state and substate when the substate represents some part of the state. Figure 4 depicts the substate matching relationship. Each letter represents a *state quantum*.

State quantum. *A state quantum is the smallest divisible (in terms of the algorithm) unit within a state.*

In the unordered state, a matching substate may consist of any subset of state quanta. In the ordered state, the order of the quanta must be preserved in the substate. The empty substate is said to match *all* states. If a state is a set of unordered quanta, $S_i = \{s_1, s_2, ...\}$, then a substate, B_k, is said to be a matching for S_i iff $B_k \subseteq S_i$. In practice, since the number of ordered states for a given system is far greater than the number of unordered states for the same system, we tend to prefer using unordered states

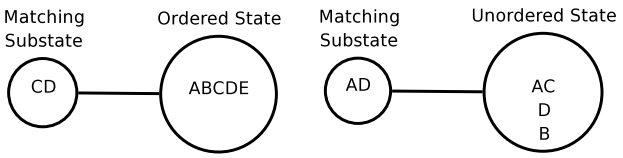

Figure 4. Ordered and Unordered states and their matching substates.

over ordered states. Using ordered states would give the learner more information, but in the experiments we conducted, that extra information was not worth the added state space. State space explosion with the ordered states was not offset by performance gains over the unordered version. In our experiments, we used agent role-goal assignment and achievement history as our states. The quanta are the actual agent assignment tuples. A substate is said to match a state when the agent assignment tuples in the achievement and assignment sets of the substate are subsets of the corresponding sets of the state. Intuitively, substates may be seen as generalizations of states.

The operation of the algorithm is independent of the state score calculation and the substate generation. For every bad action, T, given a state, the algorithm starting with the empty substate, which matches all states, computes a score using Formula (1). If this score is lower than a threshold, the algorithm asks the state for the set of smallest substates containing a specific substate (initially the empty substate). Each one of these substate-action pairs, (B_k, T), are given a score, $score(B_k, T) =$

$$1 - \frac{size(match_G)}{size(states_G) + size(states_B) + size(match_B)} \quad (1)$$

The variables in Formula (1) are as follows: $states_G$ is the entire set of good states given the transition T (gotten to by taking transition T); $states_B$ is the entire set of bad states given the transition T (gotten to by taking transition T); $match_G$ is, given the transition, the set of good states that the substate matches (a subset of $states_G$); and $match_B$ is, given the transition T, the set of bad states that the substate matches (a subset of $states_B$. It follows that the score is bounded as follows:

$$0 \leq score(B_k, T) \leq 1 \quad (2)$$

It can easily be seen that when the substate matches all good states, and we have not encountered any bad states $size(states_G) = size(match_G)$ and $size(states_B) = size(match_B) = 0$, thus $score(B_k, T) = 0$. Conversely, if the substate does not match any good states, $size(match_G) = 0$, thus $score(B_k, T) = 1$. Each substate is scored with Formula (1). The substate with the highest score is chosen. If this score is less than a threshold constant, the process is repeated but the substate that is given to the state to generate new substates is this maximum score substate. Thus we now build upon this substate and make it more specific.

```
for all ⟨action, preStateSet⟩ in badTGivenS do
  for all preState in preStateSet do
    if !isAlreadyMatched(action, preState) then
      maxSubstate = emptySubstate
      maxScore =score(maxSubstate, action)
      done =false
      while maxscore < THRHLD∧ !done do
        substateSet =getSubstates(preState,
        maxSubstate)
        if substateSet = ∅ then
          done = true
        end if
        maxScore = -1
        for all substate in substateSet do
          score = score(substate,action)
          if score > maxScore then
            maxScore = score
            maxSubstate = substate
          end if
        end for
      end while
      matches = matches ∪ ⟨action, maxSubstate⟩
      policies = policies∪ generatePolicy(action,
      maxSubstate)
    end if
  end for
end for
```

Figure 5. Pseudo-code for generating the policies from the state, action pair sets.

Intuitively we start with the most general policy and then make it as specific as necessary (taking a greedy approach) so that we exclude 'enough' good states. The closer the threshold constant is to 1, the lower the possibility of the learned policies matching good states. The closer the threshold constant is to 0, the higher the possibility that the learned policies will match good states. For our experiments, we used a threshold constant of 0.6. This proved to perform sufficiently well for us since we dealt with intermittent failures. Pseudo-code for the policy generation is given in Figure 5.

Another approach that we tested was to avoid the bad states, that is, ignoring transitions, simply construct policies that avoid generalizations (substates) of the bad states themselves. This alone can lead to problems. In our experiments, we found cases in which the system would live-

lock. Since the learner never kept track of the event that lead to the bad state, it was possible that the learner might discover policies that forbid every agent but one from trying to achieve some goal. However, if this one agent failed with 100% probability, the system might simply reinforce that the new bad state was caused by the older decisions (substate) and would keep trying to assign the goal to the failing agent. The action-substate learner gets around this because the policies are developed for the action, if an action keeps leading to a bad state, there will be a policy discovered that forbids it. In the case where there are policies forbidding every agent from trying to achieve a goal, since we are using guidance policies, the policies will be suspended and an agent will be chosen. Eventually an agent who can achieve the goal will be chosen, and the learner will learn that that agent was not the cause of the previous failures. Avoiding live-lock is also the reason we regenerate policies after every failure.

4 Evaluation

To test our algorithm, we simulated three different systems: a conference management system, an information system, and an improvised explosive device (IED) detection cooperative robotic system. These systems are a sampling across the usual multiagent system deployments: the conference management system, a human agent system; the information system, a software agent system; and the IED detection system, a robotic agent system. Each of these systems exhibit special tuning requirements given their different agent characteristics.

We simulated these systems by randomly choosing an active goal to achieve and then randomly choosing an agent capable of playing a role that can achieve the goal while following all current policies in the system. Active goals are simply goals in our goal model that GMoDS deems can be worked on. A goal may become active when triggered, or when precedence is resolved. In the case of learning, the learning code receives all events of the system. After every agent goal achievement failure, the system regenerates its learned guidance policies using all of the state, transition, and score information it has accumulated up to that point.

4.1 Conference Management System

A well known example in multiagent systems is the Conference Management [9, 12, 25] system. The Conference Management system models the workings of a scientific conference: authors submit papers, reviewers review the submitted papers, and certain papers are selected for the conference and printed in the proceedings.

We have modified the agents such that some of them fail to achieve their goal under certain conditions. In this system we have three types of Reviewer agents: one that never fails when given an assignment, another that fails 70% of the time after completing two reviews (*Limit Reviewer*), and the last type, that fails 70% of the time when trying to review certain types of papers (*Specialized Reviewer*).

We used the GMoDS goal model and role model for the Conference Management system as described in [10, 12]. In the Conference Management system we focused on failures of agents to achieve the *Review Paper* goal. We created the Reviewer types described above and observed the system performance. In this simulation, we varied the number of papers submitted for review to trigger the failure states. For our experiment, states contain the history of *assignments* and number of *failures* thus far. Actions are the *assignment* and *failure* events. Only leaf goals are used in assignments. Non-leaf goals are decomposed into the leaf goals.

In this experiment, we are concerned with the *Reviewer* role. Several different agent types are capable of playing this role, although they clearly behave differently. Each role may achieve specific goals as shown in the Figure 6.

Role Name	Goals Achieved
Assigner	Assign Reviewer
Reviewer	Review Paper
Partitioner	Partition Papers
Review Collector	Collect Reviews
Paper Database	Collect Papers
	Distribute Papers
	Collect Finals
Decision Maker	Make Decision
	Inform Accepted
	Inform Declined
Finals Collector	Send to Printer

Figure 6. CM Role Goal Relation.

We ran the system with no learning and recorded the number of failures. We then ran the system with the learning engaged and recorded the number of failures. Finally we ran the system with hand-coded policies that we thought should be optimal and recorded the number of failures.

The policies generated by the learner consist of forbidding an action given a state matching. For example, the learner discovered the policy: given the empty substate (meaning in any state), forbid the agent assignment ⟨*Specialized Reviewer, PC Reviewer, Review papers (Theory)*⟩. Thus this policy applies to all states of the system and tries to avoid assigning theory papers to the *SpecializedReviewer*. Another policy discovered by the learner is given the substate *Assigned:*⟨*Limit Reviewer, PC Reviewer, Review papers (Theory)*⟩, forbid the action ⟨*Limit Reviewer, PC Reviewer, Review papers (Theory)*⟩. The learner must learn all permutations on the abstracted goal

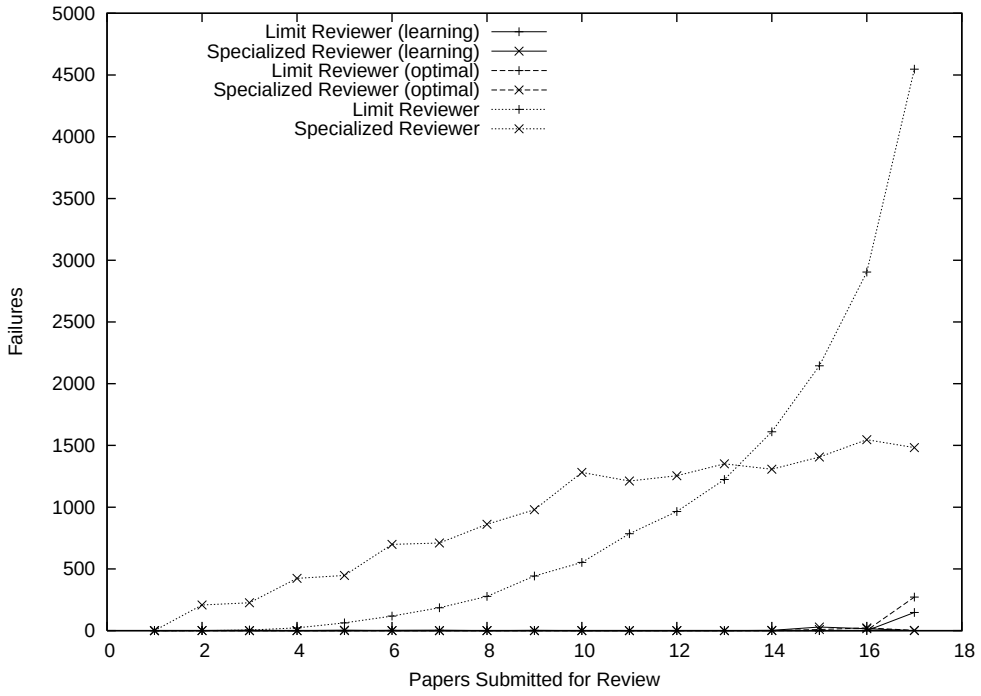

Figure 8. Agent goal achievement failures with self-tuning (learning), hand-tuning (optimal), and no tuning.

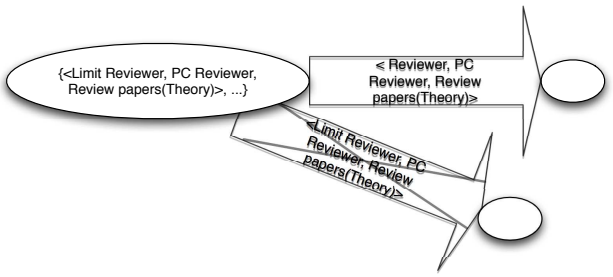

Figure 7. Limit Reviewer policy preventing theory papers from being assigned.

parameter. It is interesting here to note that the learner tries to forbid the *LimitReviewer* after just one successful assignment to it. This has the added benefit that the assignment fails as late as possible, thus the system or environment could have improved before a failure occurs. The number of policies generated is relatively small, staying less than 10 in all runs. Figure 7 gives a graphical depiction of how a learned policy relates and is applied to the system states. From a state containing the substate, *Assigned:*$\langle$*Limit Reviewer, PC Reviewer, Review papers (Theory)*$\rangle$, the assignment action $\langle$*Limit Reviewer, PC Reviewer, Review papers (Theory)*$\rangle$ is forbidden.

The agent type, role type, goal type, and a parameter abstraction is given. The parameter abstraction is currently domain specific, although a separate learner may categorize the parameters thus creating the abstraction function without the need for a domain expert.

Figure 8 compares the self-tuning, learning, system to one where this learning does not occur. We also compared the learning to a system, for which, a policy expert with knowledge of the agent goal achievement failure, hand coded policies he thought would tune the system. As can be seen, our learning algorithm does no worse than the hand-coded policies, and does vastly better than a non-tuned system. The learning tuning adapts the system quickly to its deployed environment—without requiring a policy expert to analyze the specific deployment environment and hand-craft policies to tune for said environment.

4.2 Information System

Another multiagent system we tested was an Information System. Peer-to-peer information gathering and retrieval systems have been constructed using multiagent systems, e.g. Remote Assistant for Information Sharing (RAIS) [16]. In our information system we had four types of information retrieval agents: *CacheyAgent*, this agent fails with a 30%

probability the first time it is asked for a certain piece of information, subsequent requests for info it has retrieved previously always succeeds; *LocalAgent*, this agent fails 100% of the time when asked for remote information, otherwise it succeeds with a 100% probability; *PickyLocalAgent*, this agent fails 100% of the time on any request except for on particular piece of local data; and *RemoteAgent*, this agent fails 100% of the time on all data except for remote data.

The leaf-goals of our system are as follows: *1.1 Monitor Information Search Requests, 1.2.1 Perform Local Search, 1.2.2 Perform Remote Search, 1.2.3 Present Results, 2.1 Monitor Information Requests, 2.2 Retrieve Information, 3.1 Monitor for New Information, and 3.2 Incorporate new Information into Index*.

The role-goal relation is shown in Figure 9. *CacheyAgent* is capable of playing roles *Local Information Retriever* and *Remote Information Retriever*. *LocalAgent* and *PickyLocalAgent* are capable of playing *Local Searcher* and *Local Information Retriever* roles. *RemoteAgent* is capable of playing the *Remote Searcher* and the *Remote Information Retriever* roles.

Role Name	Goals Achieved
GUI	1.1, 1.2.3, 2.1
Local Searcher	1.2.1
Remote Searcher	1.2.2
Information Monitor	3.1
Indexer	3.2
Local Information Retriever	2.2
Remote Information Retriever	2.2

Figure 9. IS Role Goal Relation.

The system quickly learned policies restricting the *PickyLocalAgent* from various information retrieval assignments. These policies were learned that they apply in all states. The *LocalAgent* also became restricted from being assigned any goal with any of the various remote information, regardless of the system state. The policies concerning assignments to the *RemoteAgent* were similar to the *LocalAgent*. The *Cachey Agent*, however, was restricted by various policies, usually of the form ⟨*CacheyAgent, Remote Information Retriever, Retrieve Information (remote info 2)*⟩ agent goal assignment is forbidden given the state assignments contain ⟨*CacheyAgent, Local Information Retriever, Retrieve Information (local info 1)*⟩. The learner did not have a mechanism for generating a negation policy. For example, do not allow the assignment if the state does *not* match the given substate. This could be a future enhancement to the policy generation of the learning. Although the learner did not have this capability, it still managed to keep the *Cachey Agent's* failures low, and the introduction of this type of agent did not confuse the learner with regards to the type of failure of the other agents.

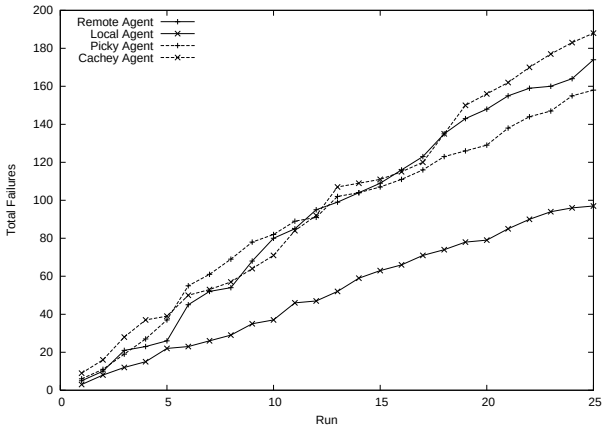

Figure 10. Agent failures with no self-tuning.

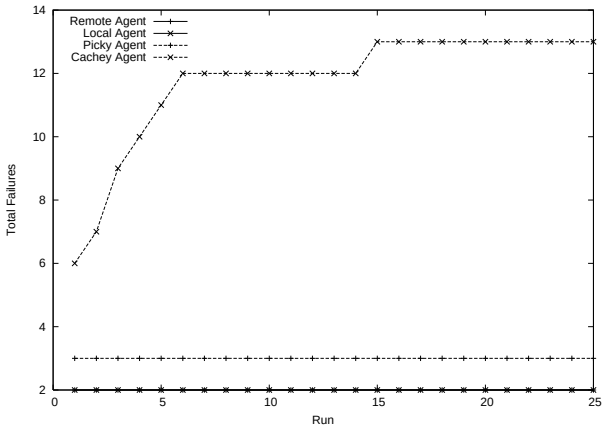

Figure 11. Agent failures using action and substate learning for self-tuning.

Figure 10 shows the agent achievement failures for a non-tuned system. An expert may analyze these failures and craft policies to guide the system to avoid the failures, but this would be an error-prone and tedious task. In fact, by the time a solution is proposed, the problem may well have changed.

Our self-tuning learning achieved the results given in Figure 11. The system was able to self-tune and adapt itself to an environment that contained multiple unique causes of failure. The adaptation happened quickly enough to greatly benefit the system.

4.3 IED Detection System

The use of multiagent systems in robotic teams is a natural application of multiagent systems. Unfortunately, robots and their physical environment contain more vari-

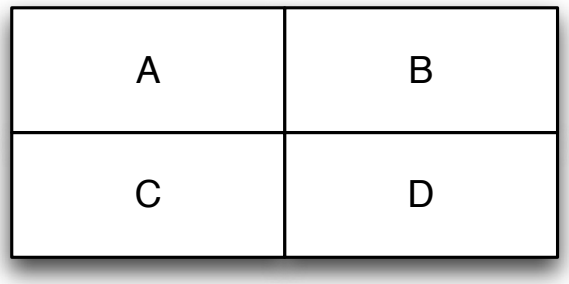

Figure 12. IED Search Area for various Patrollers.

ability than purely software-based multiagent systems. System designers may not be able to plan for all this potential variability when designing their system. Capabilities of robots may vary with their physical environment. For example, a robot needing line-of-sight for communication, may go out of communication range if they move behind a physical structure in their environment. Other robots may find that they are not able to diffuse certain types of IEDs, perhaps because the IEDs are too big for the agent's grippers. Certainly, if the system designer could think of and design for all possible environmental variations, their system would be able to perform efficiently in all environments. In practice, however, this is not practical. Thus, the system should be able to automatically learn and adapt to environmental variations on its own.

The IED detection system we used [15] consists of agents to *Patrol* an area, *Identify* IEDs, and *Defuse* IEDs. Various agents are capable of playing these roles. The *Patrol* role may be played by our *LargePatroller* or our *SmallPatroller* agents. The *Defuser* role may be played by our *LargeGripper* or *SmallGripper* agents. IEDs may be found while an agent is playing the *Patrol* role, this event will trigger an Identify goal, which in turn could trigger a Defuse goal. The Defuse goal is parametrized on the type of IED identified (large or small). The IED patrol area is first broken into four parts as shown in Figure 12.

For the first experiment, we made the *ShortPatroller* fail on area *D* with a 40% probability for the first 10 assignments made to it. After the first 10 assignments to it, the *ShortPatroller* fails with a 40% probability on area *A* and no longer fails on area *D*. The *LargePatroller* agent always fails on area *B* with a 20% probability. The *SmallGripper* fails with a 100% probability on diffusing large IEDs.

Without learning, in the first 1000 runs, the *SmallGripper* failed a total of 14879 times, the *ShortPatroller* failed 409 times, and the *LargePatroller* failed 107 times. With learning, for the first 1000 runs, the *SmallGripper* failed 1 time, the *ShortPatroller* failed 4 times, and the *LargePatroller* failed 1 time. Subsequent runs using the accumulated knowledge displayed no failures by any agents.

In the second experiment, we left the agents the same except for the *LargePatroller*. The *LargePatroller* agent now fails on area *D* with a 20% probability. This overlaps with the failure area of the *SmallGripper* agent.

In this scenario, without learning, in the first 1000 runs the *SmallGripper* failed 15172 times, the *ShortPatroller* failed 434 times, and the *LargePatroller* failed 133 times. With learning, for the first 1000 runs, the *SmallGripper* failed 1 time, the *ShortPatroller* failed 3 times, and the *LargePatroller* failed 12 times. In a subsequent 1000 run, using the accumulated knowledge, the learning fell into a sub-optimum, the *SmallGripper* failed 0 times, but the *ShortPatroller* failed 2 times, and the *LargePatroller* failed 102 times. We hypothesize that this is due to the fact that we have the overlapping failing area as well as no partial ordering on learned guidance policies. Thus when the policies must be suspended due to conflicting, or because the system cannot progress with the policies, all the learned policies are suspended at once, creating the situation similar to having no learning.

4.4 Common Results

In all of our experiments, our self-tuning mechanism was able to quickly avoid multiple failure states that had multiple independent sources. The performance of the systems increased as the system tuned to its environment. The number of polices discovered was kept small, which can be important when considering policies at run-time, since more policies can mean more processing time and effort.

In all of the scenarios, we had multiple independent failure vectors. The learning was able to overcome this. The IED simulation explored an evolving failure situation where the cause of the failure changed over time. The information system also had a unique type of failure with the *CacheyAgent*. This agent cached results of previous queries and thus would always succeed once it had successfully retrieved a particular piece of information, otherwise it had a certain probability of failure. This failure was handled by the algorithm, but due to the nature of the policies generated, it was not handled optimally and in a general sense.

5 Related Work

There has been much work in independent agent learning outside of the organizational framework. Much of this learning is on the individual agent level, with the hope that the over-all system performance will improve.

Bulka et al. [6] devised a method allowing agents to learn team formation policies for individual agents using a Q-Learning and classifier approach. They showed notable

improvement in the performance of their system. While their approach works well for open multiagent systems, it does not leverage the properties of an organization-based multiagent approach. Abdallah and Lesser [1] developed a similar method to Bulka's except they were concerned with migrating the learned information to a changed network topology as well as using the learned information to optimize the network topology of the current agent system.

Kok and Vlassis [14] used coordination graphs along with a Q-Learning approach to learn to maximize overall agent coordination performance. Again, this work does not leverage the organizational approach to multiagent systems. Every agent is equal in society and only varies with respect to capabilities possessed. In human organizations, structures are built in which actors have roles that they can fill.

Other work has been done at the agent behavior level (e.g. [20]). As an agent tries to achieve a goal, it may affect the performance of its teammates. Policies are sometimes used to restrict the agents behavior while working on a goal.

Chiang et al. [7] have done some work on automatic learning of policies for mobile ad hoc networks. Their learning, however, was an offline approach using simulation to generate specific policies from general 'objectives' and possible configurations. Our research leverages the organizational framework to generate policies online that affect the system through the processes of the organization (i.e role-goal assignments).

6 Conclusions

Multiagent systems can become quite complex and may be deployed in environments not anticipated by the system designer. Furthermore, the system designer may not have the resources to spend on hand-tuning the system for a particular deployment. With these issues in mind, we have developed a method to create self-tuning multiagent system using guidance policies.

Using a variation of Q-Learning, we have developed an algorithm that allows the system to discover policies that will help maximize its performance over time and in varying environments. The use of guidance policies, helps remove some of the traditional problems with using machine learning at the organization level in multiagent systems. If the learner creates bad policies, they should not prevent the system from achieving its goal (although they may degrade the quality or timeliness of the achievement). In this way, our approach is 'safer' and thus we can use a simpler learner.

In the experiments we conducted, the system was able to adapt to multiple agent goal achievement failures. It was able to cope with randomized and history-sensitive failures. The learner was able to discover guidance policies which, in every case, caused our systems to perform better on the order of a magnitude when faced with these failures.

Since we are taking the approach of always trying to avoid bad states, there is the question of whether our approach will possibly drive the system away from the optimal state in certain diabolical cases. The argument is that in order to get to the best state, we must pass through some bad states. To address this, we need to look at what is being considered as a bad state and if it is possible for there to be a high-scored state that can only be reached through a bad state. In the experiments we have performed, bad states corresponded to agent goal achievement failures. Using agent goal achievement failures as the scoring method of our states, it is possible that you must pass through a bad state in order to get to an end state with the highest score. But, since the score is inversely proportional to agent goal failures, we will have to go through a bad state in any case. We argue that since our score is monotonic, our algorithm should be able to drive toward the best scored state even in the diabolical case that you must go through a bad state to reach it. This, however, requires that we order our learned guidance policies using the more-important-than relation by score.

The usage of guidance policies allow for retention of useful preferences and automatic reaction to changes in the environment. For example, there could be an agent that is very unreliable, thus the system may learn a policy to not make assignments to that agent, however, the system may have no alternatives and thus must use this agent. "We don't like it, but we deal with the reality."–that is until another agent that can do the job joins the system.

7 Future Work

Guidance policies may be ordered using a *more-important-than* relation. In this work, we did not utilize that ordering. However, we hypothesize that if we ordered the learned policies by confidence, the system would be able to recover from learning errors more quickly. This is because the learning error policy would have a lower confidence than the other policies and thus would be suspended first in the case of a policy conflict or when the system cannot progress with the current policy set. Confidence may be computed using the score function described in Section 3.

Currently the algorithm assumes that only one action happens at a time. The actions of a multiagent system are not always easily serializable. In order to handle concurrent actions, we propose to consider the concurrent actions as a new compound action. In this way you may use this algorithm with minimum modification.

Another idea to automatically abstract the goal parameters and the state space is to use model checking to create a state space abstraction. We could then use this state space abstraction during learning instead of the exact state space. This would help with state space explosion and automate

the goal parameter abstraction.

8 Acknowledgments

This research was performed as part of grants provided by the Air Force Office of Scientific Research grant number and FA9550-06-1-0058 and the National Science Foundation grant number IIS-0347545. Doina Caragea's work is supported by the National Science Foundation under grant number 0711396.

References

[1] S. Abdallah and V. Lesser. Multiagent Reinforcement Learning and Self-Organization in a Network of Agents. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 172–179, Honolulu, May 2007. IFAAMAS.

[2] A. Artikis, M. Sergot, and J. Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 2007.

[3] C. Bernon, M. Gleizes, and G. Picard. Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. *Agent-oriented Methodologies*, 2005.

[4] D. Bernstein, R. Givan, N. Immerman, and S. Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[5] J. Bradshaw, A. Uszok, R. Jeffers, N. Suri, P. Hayes, M. Burstein, A. Acquisti, B. Benyo, M. Breedy, M. Carvalho, D. Diller, M. Johnson, S. Kulkarni, J. Lott, M. Sierhuis, and R. V. Hoof. Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 835–842, New York, NY, USA, 2003. ACM Press.

[6] B. Bulka, M. Gaston, and M. desJardins. Local strategy learning in networked multi-agent team formation. *Autonomous Agents and Multi-Agent Systems*, 15(1):29–45, 2007.

[7] C. Chiang, G. Levin, Y. Gottlieb, R. Chadha, S. Li, A. Poylisher, S. Newman, R. Lo, and T. Technologies. On Automated Policy Generation for Mobile Ad Hoc Networks. *Policies for Distributed Systems and Networks, 2007. POLICY'07. Eighth IEEE International Workshop on*, pages 256–260, 2007.

[8] N. Damianou, N. Dulay, E. C. Lupu, and M. Sloman. Ponder: a language for specifying security and management policies for distributed systems. *Imperial College Research Report DoC 2000/1*, 2000.

[9] S. A. DeLoach. Modeling organizational rules in the multi-agent systems engineering methodology. In *Advances in Artificial Intelligence: 15th Conference of the Canadian Society for Computational Studies of Intelligence (AI 2002)*, volume 2338 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Berlin/Heidelberg, May 2002.

[10] S. A. DeLoach. Developing a multiagent conference management system using the o-mase process framework. In M. Luck, editor, *Agent-Oriented Software Engineering VIII: The 8th International Workshop on Agent Oriented Software Engineering*, volume LNCS 4951, pages 171–185. Springer-Verlag:Berlin, 2007.

[11] S. A. DeLoach, W. Oyenan, and E. T. Matson. A capabilities based theory of artificial organizations. *Journal of Autonomous Agents and Multiagent Systems*, 2007.

[12] S. J. Harmon, S. A. DeLoach, and Robby. Trace-based specification of law and guidance policies for multiagent systems. In *8th Annual International Workshop on Engineering Societies in the Agents World (ESAW)*, October 2007.

[13] N. K. Jong and P. Stone. Model-based function approximation for reinforcement learning. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, May 2007.

[14] J. Kok and N. Vlassis. Collaborative Multiagent Reinforcement Learning by Payoff Propagation. *The Journal of Machine Learning Research*, 7:1789–1828, 2006.

[15] MACR Lab. Human Robot Teams. `http://macr.cis.ksu.edu/hurt/`.

[16] M. Mari, A. Poggi, M. Tomaiuolo, and P. Turci. A Content-Based Information Sharing Multi-Agent System. *Modelling Approaches*, May 2006.

[17] M. Miller. A goal model for dynamic systems. Master's thesis, Kansas State University, April 2007.

[18] R. Nair, M. Tambe, M. Yokoo, D. Pynadath, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 705–711, 2003.

[19] L. Panait and S. Luke. Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[20] L. Peshkin, K. Kim, N. Meuleau, and L. Kaelbling. Learning to Cooperate via Policy Search. *Arxiv preprint cs.LG/0105032*, 2001.

[21] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252, 1995.

[22] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. Kaos policy and domain services: toward a description-logic approach to policy representation, deconfliction, and enforcement. In *POLICY 2003: IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pages 93–96. IEEE, 2003.

[23] F. Viganò and M. Colombetti. Symbolic Model Checking of Institutions. *Proceedings of the 9th International Conference on Electronic Commerce*, 2007.

[24] C. J. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[25] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.