Learning the Required Number of Agents for Complex Tasks

Sébastien Paquet
DAMAS laboratory, department of computer science and software engineering
Laval University, Canada
spaquet@damas.ift.ulaval.ca

Brahim Chaib-draa
DAMAS laboratory, department of computer science and software engineering
Laval University, Canada
chaib@damas.ift.ulaval.ca

ABSTRACT

Coordinating agents in a complex environment is a hard problem, but it can become even harder when certain characteristics of the tasks, like the required number of agents, are unknown. In those settings, agents not only have to coordinate themselves on the different tasks, but they also have to learn how many agents are required for each task. To achieve that, we have elaborated a selective perception reinforcement learning algorithm to enable agents to learn the required number of agents. Even though there were continuous variables in the task description, the agents were able to learn their expected reward according to the task description and the number of agents. The results, obtained in the RoboCupRescue, show an improvement in the agents overall performance.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Coherence and Coordination, Intelligent Agents, Multiagent Systems*; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Performance, Experimentation.

Keywords

Multiagent, Reinforcement Learning, Coordination.

1. INTRODUCTION

In a cooperative multiagent environment, agents have to divide up the different tasks among themselves in order to accomplish them efficiently. However, if agents are faced with complex tasks, it might be hard for them to determine how many agents are required to accomplish each task, which is important information for the coordination process. Without this information, agents would not be able to divide up the different tasks efficiently.

To learn the required number of agents for each task, we have developed a selective perception reinforcement learning algorithm, which is useful to manage a large set of possible task descriptions with discrete or continuous variables. It enables us to regroup common task descriptions together, thus greatly diminishing the number of different task descriptions. Starting from this, the reinforcement learning algorithm can work with a relatively small state space.

Our tests in the RoboCupRescue simulation environment show that the agents are able to learn a compact representation of the state space, facilitating their task of learning good expected rewards. Furthermore, they were also able to use those expected rewards to choose the right number of agents to assign to each task. This information helped the agents to efficiently coordinate themselves on the different tasks, thus improving the group performance.

In the remaining sections of this paper, we introduce the RoboCupRescue simulation which motivated the development of our learning algorithm. Next, we describe our selective perception reinforcement learning algorithm. Then, we present the results obtained by testing our algorithm in the RoboCupRescue simulation.

2. DOMAIN

The resource allocation problem that motivated this work requires an efficient allocation of *FireBrigade* agents to the tasks of extinguishing fires. Therefore, throughout this paper, we consider *FireBrigade* agents to be resources that are allocated to fires. Those *FireBrigade* agents evolve in the RoboCupRescue simulation [4] which takes the form of an annual competition in which participants are designing rescue agents (fire fighters, policemen and paramedics) trying to minimize damages, caused by a big earthquake, such as civilians buried, buildings on fire and blocked roads.

For this article, only the agents responsible for extinguishing fires are considered, i.e. the *FireBrigade* agents which extinguish fires and the *FireStation* agent which is their communication center. All those agents are in contact by radio, but they are limited on the number of messages they can send and on the length of those messages. Furthermore, in the simulation, each individual agent receives visual information of only the region surrounding it. Therefore, agents rely on the communication to acquire some knowledge about the environment. Since the center agent has more communication capabilities, it normally has a better global view of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

the situation than the *FireBrigade* agents.

As mentioned before, the task of the *FireBrigade* agents is to extinguish fires. Therefore, at each step in time, each *FireBrigade* agent has to choose which burning building to extinguish. However, in order to be effective, *FireBrigade* agents have to be coordinated on the same burning buildings, because more than one agent is often required to extinguish a building on fire. For this article, we do not consider the importance of a task. We suppose that the agents have a utility function allowing them to place the different fires in order of priority. Their assignment to different burning buildings depends on the number of available agents and on their distance from those fires.

The main decision for the agents is to choose how many agents to send to the most important fires. However, the problem is that they do not know how many agents are required for each building on fire. The required number of agents depends on the characteristics of the task. For example, a small building on fire may require only two agents to extinguish it, but a bigger one may require 10 agents. In order to be effective, agents have to learn the right number of agents to assign to each task which is described as:

- the intensity of the fire (3 possible values),
- the building's composition (3 possible values),
- the building's size (continuous value),
- the building's damage (4 possible values).

Therefore, there are many possible task descriptions. In fact, with the continuous attribute "building's size", there is an infinite number of task descriptions. In the results section, we show an important reduction of this huge number of possible situations. But first, in the next sections, we present our selective perception reinforcement learning algorithm.

3. PROBLEM DEFINITION

In this article, we are considering agents accomplishing tasks in an uncertain and dynamic environment where most tasks require more than one agent to be accomplished. In this case, agents are forced to be coordinated and they need to know the required number of agents to accomplish each task. This can be hard to estimate if subtle changes in a task description can change the required number of agents.

To make this estimation, we propose a new approach that uses a selective perception reinforcement learning algorithm to learn the expected reward if a certain number of agents tries to accomplish a certain type of task. An advantage of learning expected rewards instead of directly learning the number of agents is that the rewards can encapsulate the time needed to accomplish a task. A task taking more time to be accomplished has a smaller expected reward, due to the discount factor.

The method we propose is inspired by the *Utree* [5] and the *Continuous Utree* [10] algorithms. Like those methods, the agent is dynamically learning a tree representation of the state space in order to reduce the number of states considered. In our case however, we do not consider states, but task descriptions and our objective is not to find a policy for the agent, but to evaluate the capability of a given group of agents to accomplish a task. Therefore, the only implicit action is to accomplish a task, but it is never explicitly considered in the model. Our model can be described as a tuple $< D, N, R, T >$ where:

- D is the set of all possible task descriptions.
- N is the number of available agents.
- R is a reward function that gives the reward if a task is accomplished.
- T is a transition function that gives the probability to go from one task description to another. In other words, it gives the probability that the task description changes while some agents are accomplishing it.

The transition function is useful to take the dynamic aspects of the environment into consideration. It takes time to accomplish a task and, during this time, some characteristics of the current task may change and this may have an impact on the required number of agents.

Moreover, a task description is described in a factored way by a set of discrete or continuous attributes: $\{A_1, A_2, \ldots, A_n\}$. The number of different task descriptions can be huge, especially if there are continuous attributes. The primary objective of building a tree representation of the task description space is to reduce the number of task descriptions considered. The next section explains how the tree is built and how it is used to estimate the required number of agents to accomplish each task.

4. TREE CONSTRUCTION

Our algorithm uses a tree structure similar to a decision tree in which each leaf of the tree represents an abstract task description that regroups many task descriptions. This compact representation is iteratively expanded when new experiences are gathered by the learning agents.

At the beginning, all tasks are considered to be the same, so there is only the root of the tree. After each simulation, agents add new experiences to the tree and the tree is expanded. Those experiences are tasks that the agents tried to accomplished in the simulation with the rewards they obtained. All experiences are stored in the leaves of the tree. To expand the tree, the algorithm tests for each leaf l whether it would be interesting to divide the experiences stored in l by adding a new test on a task's attribute. The addition of a new test refines the agents' view of the task description space.

An advantage of this algorithm is that it distinguishes only tasks that really need to be distinguished. Therefore, the task description space is reduced, thus facilitating the reinforcement learning process.

4.1 Tree Structure

The algorithm presented here is an instance-based algorithm in which a tree is used to store all agents' experiences which are kept in the leaves of the tree. The other nodes of the tree, called center nodes, are used to divide the instances with a test on a specific attribute. Each leaf of the tree also contains a Q-value indicating the expected reward if a task that belongs to this leaf is chosen. In our approach, a leaf l of the tree is considered to be a task description (a state) for the learning algorithm.

An example of a tree is shown in Figure 1. Each rectangular node represents a test on the specified attribute.

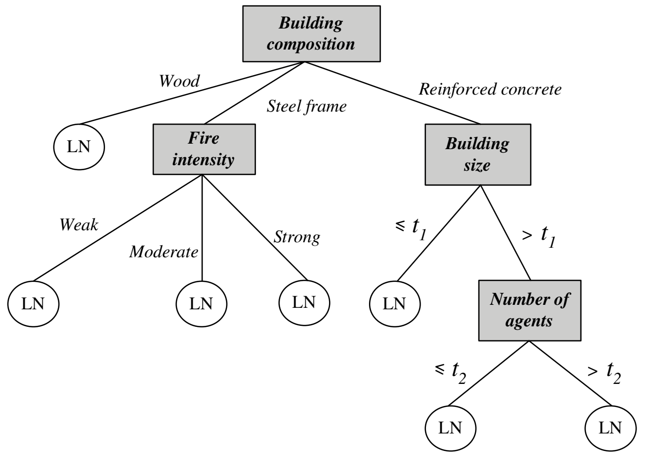

Figure 1: Structure of a tree.

The words on the links represent possible values for discrete variables. The tree also contains a center node testing on a continuous attribute, the "Building size". A test on a continuous attribute always has two possible results, it is either less or equal to the threshold or greater than the threshold. The oval nodes (LN) are the leaf nodes of the tree where the agents' experiences and the Q-values are stored.

4.2 Recording of the Agents' Experiences

In the RoboCupRescue, each simulation has 300 time steps. During a simulation, each *FireBrigade* agent records, at each time step t, its experience about which fire it is trying to extinguish. More precisely, an experience is recorded as an "instance" that contains the task it tried ($d_t \in D$), the number of agents that tried the same task (n_t) and the reward it obtained (r_t). Each instance also has a link to the preceding instance and the next one, thus making a chain of instances. Consequently, an instance at time t is defined as:

$$i_t = \langle i_{t-1}, d_t, n_t, r_t, i_{t+1} \rangle \tag{1}$$

In our case, we have one chain for each fire f that an agent chooses to extinguish. A chain contains all instances from the time an agent chooses to extinguish the fire f until it changes to another fire. Therefore, during a simulation, each *FireBrigade* agent records many instances organized in many instance chains. They keep all those instances until the end of the simulation.

The learning process takes place after a simulation, when the agents have time to learn. At this time, the *FireBrigade* agents regroup all their instances together, then the tree is updated with all those new instances and the resulting tree is returned to each agent. By regrouping their instances, agents can accelerate the learning process.

4.3 Update of the Tree

This section presents the algorithm used to update the tree using all the new recorded instances. Algorithm 1 shows an abstract version of the algorithm and the following subsections present each function used in more detail.

4.3.1 Add Instances

The first step is simply to add all the new instances, recorded by the *FireBrigade* agents, to the leaves they belong to. To find those leaves, the algorithm starts at the root of the tree and heads down the tree choosing at each center node the branch indicated by the result of the test on the instance's attribute, which could be one of the attributes of the task description d or the number of agents n.

```
Procedure UPDATE-TREE(Instances)

Inputs: Instances: all instances to add to the tree.
Static: Tree: the tree.

for all $i$ in $Instances$ do
  ADD-INSTANCE($Tree$, $i$)
end for
UPDATE-Q-VALUES($Tree$)
EXPAND($Tree$)
UPDATE-Q-VALUES($Tree$)
```

Algorithm 1: Algorithm used to update the tree.

4.3.2 Update Q-values

The second step updates the Q-values of each leaf node to take into consideration the new instances which were just added. The updates are done with the following equation:

$$Q'(l) \leftarrow \hat{R}(l) + \gamma \sum_{l'} \hat{T}(l,l')Q(l') \tag{2}$$

where $Q(l)$ is the expected reward if the agent tries to accomplish a task belonging to the leaf l, γ is the discount factor, $\hat{R}(l)$ is the estimated immediate reward if a task that belongs to the leaf l is chosen, $\hat{T}(l,l')$ is the estimated probability that the next instance would be stored in leaf l' given that the current instance is stored in leaf l. Those values are calculated directly from the recorded instances:

$$\hat{R}(l) = \frac{\sum_{i_t \in I_l} r_t}{|I_l|} \tag{3}$$

$$\hat{T}(l,l') = \frac{|\{i_t \mid i_t \in I_l \wedge L(i_{t+1}) = l'\}|}{|I_l|} \tag{4}$$

where $L(i)$ is a function returning the leaf l of an instance i, I_l represents the set of all instances stored in leaf l, $|I_l|$ is the number of instances in leaf l and r_t is the reward obtained at time t when n_t agents were trying to accomplish the task d_t.

To update the Q-values, the equation 2 is applied iteratively until the average squared error is less than a small specified threshold. The error is calculated using the following equation, which is the average squared difference between the new and the old Q-values:

$$E = \frac{\sum_{l}(Q'(l) - Q(l))^2}{nl} \tag{5}$$

where nl is the number of leaf nodes in the tree.

4.3.3 Expand the Tree

After the Q-values have been updated, the third step checks all leaf nodes to see if it would be useful to expand a leaf and replace it with a new center node, thus dividing the instances more finely and refining the agent's representation of the task description space, in order to help the agent to predict rewards.

Before trying to split a leaf node, the algorithm first checks if there are enough instances stored in the leaf. If the number of instances is too small, the algorithm does not try to split the leaf node. This is done to make sure that the statistical measures used to split the nodes are meaningful and that the algorithm is not overfitting the data.

If there are enough instances, the algorithm tries different tests to divide the instances more finely. To find the best test, the agent tries all possible tests, i.e. it tries to divide the instances according to each attribute describing a task or the number of agents. After all attributes have been tested, it chooses the attribute that maximizes the error reduction as shown in equation 6 [8].

The error measure considered is the standard deviation on the instances' expected rewards ($sd(I_l)$). Therefore, a test is chosen if, by splitting the instances, it ends up reducing the standard deviation on the expected rewards. If the standard deviation is reduced, it means that the rewards are closer to one another. Thus, the tree moves toward its objective of dividing the instances in groups with similar expected rewards, in order to help the agent to predict rewards. In fact, the test is chosen only if the expected error reduction is greater than a certain threshold, if not, it means that the test does not add enough distinction, so the leaf is not expanded.

The expected error reduction obtained when dividing the instances I_l of leaf l is calculated using the following equation where I_k denotes the subset of instances in I_l that have the k^{th} outcome for the potential test:

$$\Delta error = sd(I_l) - \sum_k \frac{|I_k|}{|I_l|} \times sd(I_k) \qquad (6)$$

The standard deviation is calculated on the expected reward of each instance which is defined as:

$$Q_I(i_t) = r_t + \gamma \hat{T}(L(i_t), L(i_{t+1})) \times Q(L(i_{t+1})) \qquad (7)$$

where $\hat{T}(L(i_t), L(i_{t+1}))$ is calculated using equation 4 and $Q(L(i_{t+1}))$ using equation 2.

As mentioned earlier, one test is tried for each possible instance's attribute. For a discrete attribute, we divide the instances according to their value for this attribute. For instance, if an attribute has three possible values, it generates three subsets, thus adding three children nodes to the tree. We then use Equation 6 and record the error reduction for this test. For a continuous attribute, we have to test different thresholds to find the best one. A continuous attribute always divides the instances in two subsets, the first one is for the instances with a value less or equal to the threshold for the specified attribute and the second subset is for the instances with a value greater than the threshold.

To find the best threshold, we have used the technique described by Quinlan [7]. The instances are first sorted according to their value for the attribute being considered. Afterwards, we examine all $m-1$ possible splits, where m is the number of different values. For example, with an ordered list of values $\{v_1, v_2, ..., v_m\}$, we try all possible thresholds. So, we try the value v_1 as a threshold, thus dividing the instances in two subsets, those less or equal and those greater than v_1. We calculate and record the error reduction for this division. Then, we do the same thing for the other possible values, v_2 to v_{m-1}. At the end, we keep only the threshold with the best error reduction value.

```
Function NUMBER-AGENTS-REQUIRED($d$)

Inputs: $d$: a task description.
Static: $Tree$: the tree learned.
        $N$: the number of available agents.
        $Threshold$: the limit to surpass.

for $n = 1$ to $N$ do
  $expReward \leftarrow$ EXPECTED-REWARD($Tree$, $d$, $n$)
  if $expReward \geq Threshold$ then
    return $n$
  end if
end for
return $\infty$
```

Algorithm 2: Algorithm used to find the required number of agents for a given task description d.

When the tree has been updated, the UPDATE-Q-VALUES function is called again to take the new tree structure into consideration.

4.4 Use of the Tree

During a simulation, all learning agents use the same learned tree to estimate the number of agents that are required to accomplish a task. Since the number of agents is considered as an attribute when the tree is learned, if different numbers of agents are tested, different leaves and thus different rewards may be found, even with the same task description. Consequently, to find the required number of agents for a particular task, the algorithm can test different number of agents and look at the expected rewards returned by the tree.

Algorithm 2 presents the function used to estimate the required number of agents for a given task. In this algorithm, the function EXPECTED-REWARD returns the expected reward if n agents are trying to accomplish a task described as d. To do so, it finds the corresponding leaf in the tree, considering the task description d and the number of agents n, and records the expected reward for this abstract task.

The EXPECTED-REWARD function is called for all possible number of agents until the expected reward returned by the tree is greater than a specified *Threshold*. If the expected reward is greater then the *Threshold*, it means that the current number of agents should be enough to accomplish the task. If the expected reward is always under the *Threshold*, even with the maximum number of agents, the function returns ∞, meaning that the task is considered impossible with the available number of agents N. The *Threshold* value is set empirically and it corresponds to the minimum expected reward needed to accomplish a task. The agent stops when the *Threshold* is exceeded because the objective here is to find the minimum number of agents for each task.

4.5 Algorithm Characteristics

First of all, let's look at the algorithm complexity. The first step of the algorithm used to update the tree is to add all the new instances recorded during the last simulation. In the worst case, this step takes $O(|I|D_{\max})$, where $|I|$ is the number of instances to add and $D_{\max}$ is the maximal depth of the tree. Therefore, as the tree grows, this step takes more time. However, in our RoboCupRescue experiments, the maximum depth of the tree was always relatively small (≤ 30).

The second step is to update the Q-values using Equation 2. The reward and the transition function (Equations 3 and 4) do not take time, because they are simply updated each time a new instance is added to a leaf. The complexity of Equation 2 depends on the number of subsequent leaf nodes link to the current leaf node. To update the Q-values, the algorithm has to visit all the leaves and in the worst case, all the leaves are connected together, thus the complexity is $O(|L|^2)$, where $|L|$ is the number of leaves in the tree. This complexity is multiplied by the number of iterations needed until convergence of the Q-values. Since the Q-values are normally only slightly modified when the new instances are added, it generally does not take a lot of iterations to converge. Most of the time, it took less then 10 iterations in our tests.

The third step of the algorithm is to expand the tree. To achieve that, the algorithm has to visit all the leaves of the tree one time. For each leaf, it has to evaluate all possible splits using all the attributes describing a task. For a discrete attribute, there is only one split to try. For a continuous attribute, there are as many possible splits as there are different values for this continuous attribute in the current leaf. In the worst case, there are as many attribute values as there are instances in the leaf. Therefore, the complexity in the worst case is: $O(|L|n_d n_c |I|_{\max})$, where $|L|$ is the number of leaves in the tree, n_d and n_c are the number of discrete and continuous variables used to describe a task and $|I|_{\max}$ is the maximum number of instances in a leaf.

The last step of the algorithm does another update of the Q-values. Consequently, the total complexity of the algorithm is: $O(|I|D_{\max}) + 2O(|L|^2) + O(|L|n_d n_c|I|_{\max})$. The most expensive step is the expansion step, because it manipulates all the instances stored in all the leaves of the tree. In our tests, the time to update the tree was not a big factor since it was executed offline. It always took less time to learn the tree than to run a simulation.

Moreover, for this algorithm, we suppose that all the *FireBrigade* agents have the same vision of the situation, i.e. they see the same fires. In the RoboCupRescue it is almost always the case, because the agents are close from one another when they are extinguishing fires and the fires can be seen from a far distance.

Another hypothesis is that all the agents have the same learned tree. The tree is learned offline after each simulation, using all the instances gather by the *FireBrigade* agents during the simulation. The communications between the agents are really limited during a simulation, thus we wanted the agents to have some common ground on which to base their decisions in order to reduce the amount of communication necessary to maintain the coordination between the agents.

In the next section, we present some experiments in which we described in more detail how the learned trees are used. We also present results showing the quality of the solutions found and the speed at which the tree grows when we add new instances.

5. EXPERIMENTS

In this section, we present how our learning algorithm can be used in the RoboCupRescue simulation to help the *FireBrigade* agents to coordinate themselves on the buildings on fire to extinguish. Since there could be a lot of fires, agents do not consider all fires at once. They choose separately which fire area to extinguish and which specific building in the chosen fire area to extinguish. Fire areas are simply groups of close buildings on fire. To make their decision, agents use the tree created offline to estimate the required number of agents for each building on fire. All agents have the same tree and it does not change during a simulation.

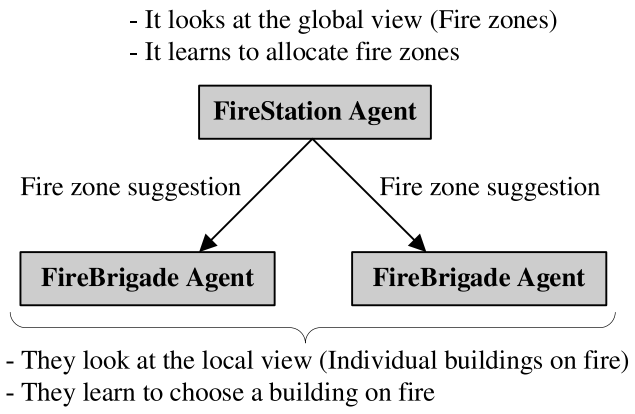
- It looks at the global view (Fire zones)
- It learns to allocate fire zones
FireStation Agent
Fire zone suggestion
Fire zone suggestion
FireBrigade Agent
FireBrigade Agent
- They look at the local view (Individual buildings on fire)
- They learn to choose a building on fire

Figure 2: Illustrates how the agents collaborate to choose which fires to extinguish.

As previously stated, the *FireStation* agent has a better global view of the situation and therefore it can suggest fire areas to *FireBrigade* agents. The *FireBrigade* agents have however a more accurate local view, consequently they choose which particular building on fire to extinguish in the given area. By doing so, as illustrated on Figure 2, we can take advantage of the better global view of the *FireStation* agent and the better local view of the *FireBrigade* agent at the same time.

5.1 Fire Areas Allocation

To allocate the fire areas, the *FireStation* agent has a list of all fire areas. For each fire area, it has to estimate the number of agents that are required to extinguish this area. To do so, it makes a list of all the buildings that are at the edge of the fire area. Agents only consider buildings at the edge because those are the buildings that have to be extinguished to stop the propagation of the fire. For each burning building at the edge, the agent finds the number of agents required to extinguish the fire (Algorithm 2). It then estimates the required number of agents for the fire area as the maximum number of agents returned for one building in the area. The *FireStation* agent does the same thing with all fire areas, ending up with a number of agents (n_z) for each area.

Afterwards, for each area z, the *FireStation* agent calculates the average distance of the n_z closest *FireBrigade* agents from z. Then, it chooses the fire area z with the smallest average distance. Consequently, the n_z closest *FireBrigade* agents from the chosen z are assigned to z. The *FireStation* agent then removes the assigned area and *FireBrigade* agents from its lists and continues the process with the remaining agents and the remaining fire areas. It continues until there is no agent or fire area left. At the end, the *FireStation* agent sends to each *FireBrigade* agent their assigned fire area.

5.2 Choice of Buildings on Fire

When choosing a building to extinguish, a *FireBrigade* agent builds a list of buildings on fire it knows about in the fire area specified by the *FireStation*. This list is sorted according to a utility function that gives an idea about the usefulness of extinguishing a fire. The utility function gives a value to a fire based on the buildings and the civilians in danger if this fire propagates to buildings close by.

All *FireBrigade* agents have approximately the same list of buildings on fire. To choose their building on fire they go through the list, one building at a time. For each building, they use the tree to find the expected required number of agents to extinguish the fire, by using Algorithm 2.

The *FireBrigade* agents are assigned to the burning buildings following a prefixed order given to them at the beginning of the simulation. Knowing the sorted list of buildings on fire, the order of all *FireBrigade* agents and the number of agents required for each fire, each *FireBrigade* agent can choose the fire it should extinguish. For example, suppose that there are two fires requiring 5 and 3 agents respectively and that the *FireBrigade* agent A has to choose one of them. If the agent A's rank is 3, it would choose the first fire, because the five first agents have to go to the first fire. However, if the agent A's rank is 7, it would choose the second fire, because the agents ranked 6, 7 and 8 have to go to the second building on fire. With this coordination process, if all *FireBrigade* agents actually have the same information, they should be well coordinated on which buildings to extinguish.

6. RESULTS AND DISCUSSION

As mentioned before, experiments have been done in the RoboCupRescue simulation environment. We have made our tests on a situation with a lot of fires, but with all roads cleared. The simulations started with 8 fires, but the 15 *FireBrigade* agents were beginning to extinguish fires only after 30 simulation steps to leave some time for the fires to propagate. This gave us a hard situation to handle for the *FireBrigade* agents. Those agents started with an empty tree and they learned from one simulation to another to distinguish the tasks that had to be distinguished and the expected rewards associated with those tasks.

If we look back at our model in Section **??** ($< D, N, R, T >$), we can see that D is the set of all possible fire descriptions. In our experiments, a task was described by: the intensity of the fire (3 possible values), the building's composition (3 possible values), the building's size (730 different buildings), the building's damage (4 possible values) and the number of *FireBrigade* agents (15 possible values). Therefore, there were 394 200 possible task descriptions. N is the number of possible *FireBrigade* agents, which was 15. R is the reward function, which was simply 100 to extinguish a fire and 0 otherwise. Finally, T is the transition function which was estimated using the instances stored in the leaves of the tree as explained in Equation 4.

To compare the results obtained by our agents, we have compared them with two other strategies (see Figure 3). The first one is the team ResQFreiburg [1] which finished first at the 2004 RoboCupRescue simulation world competition. This team uses data-mining techniques to evaluate the propagation of the fires and a priority function to choose which fire to extinguish. If we look at the performance of ResQFreiburg, they only obtained an average percentage of intact buildings of 59%.

The second one is our strategy, but without the learning part, thus all agents were going to the first building on the list. This comparison shows the advantage of learning the required number of agents to accomplish a task. If all agents go to the same building, they obtained an average percentage of intact buildings of 63% and after learning, they obtained 84%. This is a huge improvement showing that the information learned is really useful. They learned how many agents are required for all possible situations so they were able to divide the tasks efficiently between the different agents. The improvement in the performances is mainly due to the fact that agents are able to split themselves on the first two or three tasks and accomplish them all at once. Thus, the more efficient they become at estimating the required number of agents, the faster they become at accomplishing all their tasks.

Another interesting result is that our agents were able to attain such good performances with trees having less than 2000 leaves. Therefore, they were just distinguishing 2000 task descriptions out of the 394 200 possible task descriptions. In other word, our agents were able to perform efficiently with an internal task description space of only 0.5% of the complete task description space. This shows a very good reduction of the task description space, enabling the learning algorithm to work on an easier problem.

Moreover, Figure 4 presents results with different γ values for the equation 2. It shows the evolution of the percentage of intact buildings at the end of a simulation. Every point represents an average over 10 simulations. We have tested

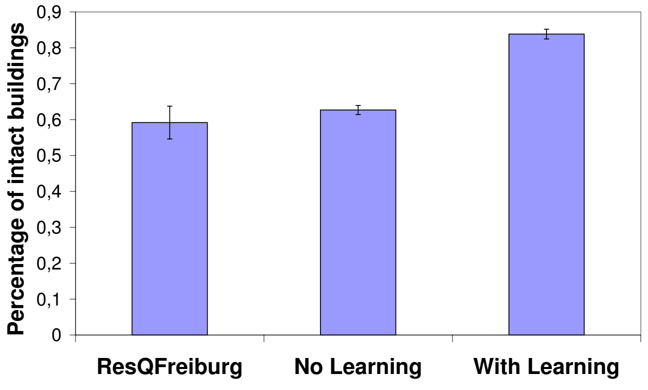

Figure 3: Comparison with other strategies.

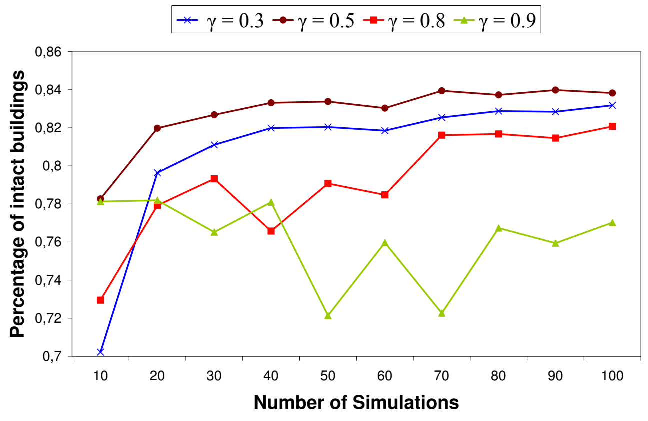

Figure 4: Percentage of intact buildings over 100 simulations for different γ values.

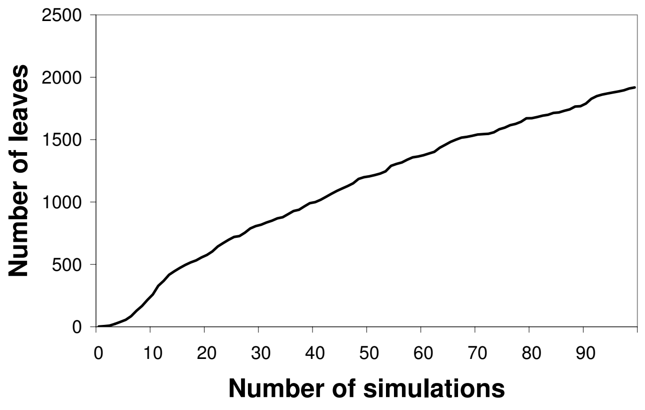

Figure 5: Number of leaves in the tree over 100 simulations.

all γ values from 0 to 1 with an increment of 0.1, but here we present only four representative learning curves. As we can see, with high γ values, the learning process is less effective. With 0.9, the agents do not get better at all, and with 1, it was even worse. With a value of 0.8, the agents have some trouble to learn at the beginning, but eventually they start to catch up after approximately 70 simulations. We have observed that with a higher gamma value, agents normally need more time to learn. With the small γ values of 0.3 and 0.5, the learning curves are quite smooth.

In each simulation, agents were able to gather approximately 2000 instances. Of course, with our algorithm, the necessary memory always grows to store all those instances, but since they were not taking too much space, this was not really a problem. Moreover, after the learning phase, the instances are not necessary anymore, therefore when the tree is used at the execution time it is really small.

Since we are expanding a tree, the growth could be exponential. However, it is not the case, because we are only expanding leaves that help to predict rewards. Therefore, the growth of the tree is controlled and in our tests it was even sub-linear (see Figure 5).

7. RELATED WORKS

In cooperative multiagent learning, most researchers have focused on coordinating agents' actions, but most of them consider that the characteristics of the tasks are known [9, 2]. In our case, an important characteristic of the tasks was unknown, i.e. the required number of agents for each task. Thus we focused our attention on developing an algorithm enabling agents to learn this important task's characteristic. Some researchers consider that the number of agents is unknown, but for very simplified tasks needing only one or two agents [3].

Other researchers have used tree structures to represent and learn the state space of the agent. One of them is McCallum with its U-tree algorithm [5]. He has used a tree structure to store Q-values for all possible basic actions that an agent can take. In our work, we use the tree to calculate the expected reward of a particular task description, considering a certain number of agents accomplishing it. It still has to find the actions to accomplish this task. The learned tree is used to estimate the number of agents needed to accomplish a task and not to find the action of the agent. Also, instead of generating all possible subtrees like McCallum, we use an error reduction estimation measure, borrowed from the decision tree theory [8], that enables us to find good tests. In our case, the generation of subtrees would really be expensive since our state space is very large.

There are different ways that can be used to expand the tree ([5, 6, 10]), but we have used an approach that has been shown to be effective in decision tree algorithms ([8]) and that enabled us to have a fast algorithm and to consider continuous attributes.

The way we manage our instance chains is also quite different. In our work, there is not only one chain, but many chains, one for each attempt to extinguish a fire or a fire zone. Our concept of instance chain is closer to the concept of *episode* described by Xuan, Lesser and Zilberstein [11].

8. CONCLUSION

In this article, we have presented a learning algorithm which is useful to learn the required number of agents for each different tasks in a complex cooperative multiagent environment. To achieve that, we have adapted a selective perception learning algorithm that learns a tree representation of the task description space. Our algorithm enables us to reduce the number of different task descriptions considered by regrouping tasks that have similar rewards. In this context, agents can learn to distinguish only between tasks that really need to be distinguished.

Afterwards, the learned expected rewards are used by our coordination algorithm to find the required number of agents to accomplish a task, both at the global and the local decision level. We added an attribute "number of agents" in the task description, which enabled us to use a greedy algorithm to find the minimum required number of agents to accomplish a given task.

With our learning approach, our agents were able to obtain good results in the RoboCupRescue simulation with an internal task description space of only 0.5% of the complete task description space. All the results presented in this article show that our approach is efficient for learning the required number of agents to accomplish a task and to coordinate agents on those tasks in a partially observable, dynamic and uncertain real-time environment.

9. REFERENCES

[1] M. Brenner, A. Kleiner, M. Exner, M. Degen, M. Metzger, T. Nussle, and I. Thon. ResQ Freiburg: Deliberative Limitation of Damage. In D. Nardi, M. Riedmiller, and C. Sammut, editors, *RoboCup-2004: Robot Soccer World Cup VIII*, Berlin, 2005. Springer Verlag.

[2] C. B. Excelente-Toledo and N. R. Jennings. The Dynamic Selection of Coordination Mechanisms. *Journal of Autonomous Agents and Multi-Agent Systems*, 9(1-2):55–85, 2004.

[3] A. Garland and R. Alterman. Autonomous Agents that Learn to Better Coordinate. *Autonomous Agents and Multi-Agent Systems*, 8(3):267–301, May 2004.

[4] H. Kitano. Robocup rescue: A grand challenge for multi-agent systems. In *Proceedings of ICMAS 2000*, Boston, MA, 2000.

[5] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis,

University of Rochester, Rochester, New-York, 1996.

[6] L. D. Pyeatt and A. E. Howe. Decision tree function approximation in reinforcement learning. Technical Report TR CS-98-112, Colorado State University, Fort Collins, Colorado, 1995.

[7] J. R. Quinlan. *C4.5 Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[8] J. R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 236–243, Amherst, Massachusetts, 1993. Morgan Kaufmann.

[9] O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200, 1998.

[10] W. T. B. Uther and M. M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 769–774, Menlo Park, CA, 1998. AAAI-Press/MIT-Press.

[11] P. Xuan, V. Lesser, and S. Zilberstein. Modeling Cooperative Multiagent Problem Solving as Decentralized Decision Processes. *Autonomous Agents and Multi-Agent Systems*, 2004.