Task Allocation Learning in a Multiagent Environment: Application to the RoboCupRescue Simulation

Sébastien Paquet, Brahim Chaib-draa, Patrick Dallaire and Danny Bergeron
Computer Science and Software Engineering Dept
Laval University, Québec, PQ, Canada

March 21, 2010

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 these 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 contribute to this problem, we present in this paper a selective perception reinforcement learning algorithm which enables agents to learn the required number of agents that should coordinate their efforts on a given task. Even though there are continuous variables in the task description, agents in our algorithm are able to learn their expected reward according to the task description and the number of agents. The results, obtained in the RoboCupRescue simulation environment, show an improvement in the agents overall performance.

1 Introduction

Our main motivation was to develop a method to determine how many agents to assign to a task in a complex multiagent environment. An example of such a complex multiagent environment with complex tasks is the RoboCupRescue simulation. In this environment, the *FireBrigade* agents have to coordinate themselves on the most important fires. To efficiently extinguish fires, the *FireBrigade* agents have to divide themselves on different fires. Solutions to coordination problems can be divided into three general classes [1]:

- Those based on communication in which agents can communicate and negotiate together to: (i) determine the allocation of the tasks; (ii) solve conflicts and (iii) share resources.
- Those based on conventions in which agents use "predefined rules" imposed by the system designer to assure a joint optimal action.
- Those based on learning, in which agents can learn coordination policies (or conventions) by repetitive interactions with other agents.

In environments where the communications are limited or uncertain, approaches highly based on communication are not really appropriate. In such environments, the quantity of

information that can be sent is limited and some messages may never reach their recipients. Consequently, approaches highly based on communication may become inefficient, since the agents' coordination relies on uncertain communications. The convention-based coordination consists of defining all the necessary conventions *a priori* and consequently it requires that all the characteristics of the task are known. The learning approach enables to go beyond this limitation particularly for complex environments where it is very hard to determine parameters of each task. Notice that learning is considered here as a process that modifies the different agent components in order to better align them to the information returned by the environment, thus improving their individual performance [21].

In our case, the required number of resources for each task (including the required number of agents) is completely unknown. Thus we focused our attention on developing an algorithm enabling agents to learn this important task's characteristic. Some researchers considered that the number of resources is unknown, but for very simplified tasks requiring only one or two resources [8]. Here, we present a more general approach that allows us to deal with many resources in a complex task description space. For instance, in the context of the RoboCupRescue simulation, *FireBrigade* agents need to learn how they can divide themselves *efficiently* over the different fires. This is a hard learning task since since the targeted learning algorithm has to manage the following constraints:

- It should deal with changes in the dynamics of the RoboCupRescue environment;
- It should be efficient with complex tasks described with many attributes;
- It should deal with tasks having discrete and continuous variables;
- It should be general enough so that it could be applied to previously unseen task descriptions;
- It should be as precise as possible, because there are few resources and many tasks.

To address this difficult problem, we have developed a selective perception reinforcement learning algorithm [17], 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 showed that the agents are able to learn a compact representation of the state space, facilitating their task of accumulating good expected rewards. Furthermore, agents 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 following sections, we describe our selective perception reinforcement learning algorithm used to learn the number of resources necessary to efficiently accomplish a task. Then, we present the results obtained by testing our algorithm in the RoboCupRescue simulation. But first, we present the application domain.

2 Application Domain

As we previously stated, our work was motivated by the resource allocation in the RobocupRescue simulation and particularly the problem to know how to spread the *FireBrigade* agnets over the different fires. Notice that the RoboCupRescue simulation [11] takes the form of an annual competition in which participants design 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.

In this article, only the agents responsible for extinguishing fires are considered, i.e. the *FireBrigade* agents which extinguish fires and the *FireStation* agent which constitutes their communication center. All those agents are in contact by radio, but they are limited on the number of messages they can send as well as the length of those messages. Furthermore, in the simulation, each individual brigade agent receives visual information of only the region surrounding it, characterizing the better local view of the mobile brigade agents. Therefore, agents rely on the communication to acquire some knowledge about the environment. One should note that, since the center (immobile) agent has more communication capabilities, corresponding to its useful perceptions, it normally has a better global view of the situation than the *FireBrigade* agents.

As mentioned before, the task of the *FireBrigade* agents consists in extinguishing 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 coordinate their choices about the burning buildings to extinguish, because more than one agent is often required to extinguish a building on fire. We suppose here that our *FireBrigade* agents have a utility function allowing them to order the different fires according to the fires' priorities (for more details about this utility function, see Equation 9 on page 14). Evidently, the agents' choices about the different burning buildings to extinguish depends: (i) on the number of available *FireBrigade* agents and (ii) on their distance from those fires.

As previously stated, the *FireStation* agent can have a better global view of the situation, because it has more communication capabilities than the *FireBrigade* agents. Therefore it can suggest good fire areas to *FireBrigade* agents. On the other hand, the *FireBrigade* agents have a more accurate local view because they can directly perceive near objects, consequently they can choose more efficiently which particular building on fire to extinguish in their surrounding area (see Figure 1). By doing so, 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.

The main decision for the *FireBrigade* agents is to choose the required number of agents to send to the most important fires. However, the problem is that they do not know a priori how to do that adequately. The required number of agents depends on the characteristics of the building. For example, a small building on fire may require only two agents to extinguish it, but a bigger one may require 10 agents. In these conditions, each different task might be described by:

- the fire's fierceness (3 possible values),
- the building's composition (3 possible values),
- the building's size (continuous value),

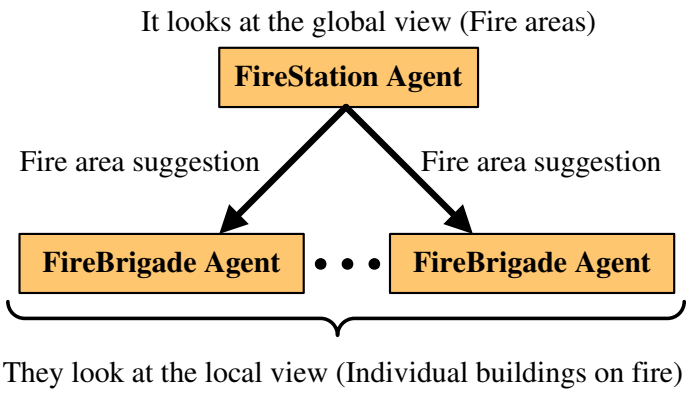

Figure 1: Collaboration between the *FireStation* and the *FireBrigade* agents.

- the building's damage (4 possible values).

As we can see, there are many possible task descriptions. In fact, with the continuous attributes, there is an infinite number of task descriptions. With such a huge number of task descriptions, it is necessary to find an algorithm that can generalize the information learned on one task to similar tasks.

3 Problem Definition

In this article, we consider 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 coordinate their tasks' choices and they need to know the required number of agents to accomplish each task. To estimate this number, 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.

In our approach, an agent dynamically learns a tree representation of the task description space in order to reduce the number of task descriptions considered. Task descriptions yielding similar reward are grouped together into the leaves of the tree. This approach has been used before to find a compact representation of the state space to facilitate the definition of the agent's policy [3, 13, 14, 16, 20, 28, 29].

Starting from the original U-Tree algorithm [13], we have conceived our own algorithm for the allocation problem. Conversely to U-Tree, we do not use the tree to calculate Q-values for every possible basic action that an agent can take, e.g. to move in some direction, to communicate, to sprinkle. We use the tree to calculate the expected reward of a particular goal decision, i.e. extinguishing a particular fire by a set of agents. Thus, the tree is used at the abstract decision level, not at the action decision level.

To be more precise, 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 a positive immediate reward only when 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. Moreover, a task description is described in a factored way by a set of discrete or continuous attributes: $\{A_1, A_2, \ldots, A_n\}$. As previously stated, 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 presents our selective perception reinforcement learning algorithm and explains how the tree is built and how it is used to allocate the ressources.

4 A Selective Perception Reinforcement Learning Algorithm

Our task allocation 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 instances 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 instances to the tree and the tree is expanded. Those instances are tasks that the agents tried to accomplish in the simulation with their associated rewards. All instances 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 instances 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' instances 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. Furthermore, each leaf of the tree 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.

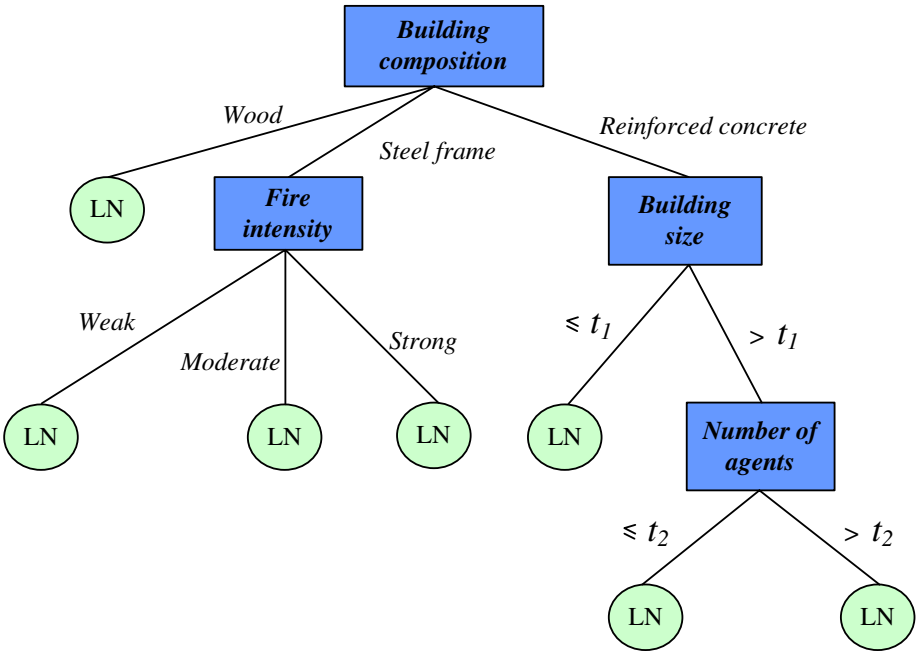

Figure 2: Structure of a tree. Rectangle nodes represents test on an attribute. LN (leaf nodes) are the leaves where instances and the corresponding Q-value are stored.

An example of a tree, relative to the RoboCupRescue, is shown in Figure 2. Each rectangular node represents a test on the specified attribute. The labels 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' instances and the Q-values are stored. Notice that, in a complete tree, there are always many nodes "Number of agents". These nodes are used to evaluate the number of required agents for a task, as we will see later.

4.2 Recording the Agents' instances

In the RoboCupRescue, each simulation takes 300 time steps. During a simulation, each *FireBrigade* agent records, at each time step t, its experiment about which fire it is trying to extinguish. More precisely, an experiment is recorded as an *instance* that contains the task in consideration ($d_t \in D$), the number of agents that tried the same task (n_t) and the obtained reward (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 that an agent chooses to extinguish. A chain contains all instances from the time an agent chooses to extinguish a fire until it changes to another fire. Therefore, during a simulation, each *FireBrigade* agent records many instances organized in many instance chains.

Algorithm 1 Algorithm used to update the tree.

1: **Procedure** UPDATE-TREE(*Instances*)

Input: *Instances*: all instances to add to the tree.
Static: *Tree*: the tree.

2: **for all** i in $Instances$ **do**
3: ADD-INSTANCE($Tree$, i)
4: **end for**
5: UPDATE-Q-VALUES($Tree$)
6: EXPAND($Tree$)
7: UPDATE-Q-VALUES($Tree$)

One should note that, in the original U-Tree algorithm [13], there is only one chain of instances which links all instances in the simulation. In our case however, it is better to use many instance chains because the tasks are independent. Moreover, all those instances regrouped in many instance chains are only recorded during a simulation. Agents do not have time to learn during a simulation, because they have to act while respecting the real-time constraint of the RoboCupRescue simulation. Therefore, the learning process only 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.

To sum up, all *FireBrigade* agents and the *FireStation* agent have the same tree learned from all the *FireBrigade* agents' instances. Afterwards, as is explained in Section 4.4, the *FireStation* agent uses the learned tree to assign *FireBrigade* agents to fire areas and each *FireBrigade* agent uses the learned tree to choose which fire to extinguish in the assigned area.

4.3 Update of the Tree

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

4.3.1 Add Instances

The first step of Algorithm 1 is simply to add all the new instances, recorded by the *FireBrigade* agents, to the leaves they belong to (Algorithm 1, lines 2-4). 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. When building the tree, the number of agents that tried to accomplish the task is considered as a normal task attribute. Thus, it makes possible to get the expected reward for different allocation schemes, given a task description, and to select the number of agents yielding a desirable outcome.

4.3.2 Update Q-values

The second step of Algorithm 1 updates the Q-values of each leaf node to take into consideration the new instances which were just added (Algorithm 1, line 5). The objective here is to have precise Q-values when the time comes to expand the tree. The Q-update equation is as follows:

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

In this update equation, $Q(l)$ is the expected reward if the agent tries to accomplish a task belonging to the leaf l, γ is the discount factor ($0 \leq \gamma \leq 1$), $\hat{R}(l)$ 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. $\hat{R}(l)$ corresponds to the average immediate reward of all the instances stored in leaf l. More specifically, it is an average of null or positive rewards gained when a fire is extinguished. $\hat{T}(l, l')$ is the number of times that the following instance of an instance stored in leaf l is in leaf l', divided by the total number of instances in leaf l. More formally, the equations defining those values are as follows:

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

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

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 previous Equation 2 is applied iteratively until the average squared error is less than a small specified threshold. The average squared error E is in fact 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 next step checks all leaf nodes to see if it would be useful to expand a leaf and replace it with a new center node (Algorithm 1, line 6). The objective is to divide the instances more finely and to refine the agent's representation of the task description space, in order to help the agent to predict rewards.

To find the best test to divide the instances in each leaf, the agent tries all possible tests, i.e. it tries to divide the instances according to each attribute describing a task. Once all attributes tested, it chooses the attribute that maximizes the error reduction according to the Equation 6.

The error measure considered is the standard deviation ($sd(I_l)$) on the instances' expected rewards. Therefore, a test is chosen if, by splitting the instances, it ends up with a reduction

of 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 [19] where I_k denotes the subset of instances in I_l that have the k^{th} outcome for the potential test:

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

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

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

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 experimented 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 into 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 used techniques similar to those of Quinlan [18]. 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 $\{v_1, v_2, \ldots, v_m\}$, 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.

At the end, when the tree has been updated, the UPDATE-Q-VALUES function is called again to take the new tree structure into consideration (Algorithm 1, line 7). The updates are done exactly the same way as presented previously in this section.

4.4 Use of the Tree

During a simulation, all 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, different leaves and thus different rewards may be found when different values are tested, even with the same task description. Consequently, to find the required number of agents for a particular task, the algorithm can test different values for this attribute 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 at line 3 returns the

expected reward if n agents are trying to accomplish a task described as d. To this end, 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 numbers 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.

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

1: **Function** NUMBER-AGENTS-REQUIRED(d, N) **returns** an integer

Inputs: d: a task description.
N: the number of available agents.
Output: n: the number of agents to get an expected reward greater or equal than the Threshold.
Statics: *Tree*: the tree learned.
Threshold: the limit to surpass.

2: **for** $n = 1$ to N **do**
3: $expReward \leftarrow$ EXPECTED-REWARD($Tree$, d, n)
4: **if** $expReward \geq Threshold$ **then**
5: **return** n
6: **end if**
7: **end for**
8: **return** ∞

We should note that this "naive" algorithm is given here just as a first indication, particularly because we deal with small N. In the case where N is high, this algorithm can be improved by employing a variation of binary search which starts with $N = 2$ and double it until $expReward \geq Threshold$, then using binary search between $N/2$ and N to identify the smallest value of N for which $expReward \geq Threshold$.

4.5 Characteristics of the Update Tree Algorithm

Firstly, let's now consider the complexity of our update tree algorithm 1. 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 instances, 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 linked 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 only slightly modified when the new instances are added, it generally does not take a lot of iterations to converge. In our tests, it took less then 10 iterations most of the time.

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.

Notice that this analysis is based on the following assumption: all agents have the same vision of the situation. In the RoboCupRescue it is almost always the case, because the agents are close to one another when they are extinguishing fires and the fires can be seen from a far distance. Another Assumption is that all the agents have the same learned tree. In the RoboCupRescue, the tree is learned offline after each simulation, using all the instances gathered by the *FireBrigade* agents during the simulation. In fact, the communications between the agents are really limited during a simulation, and the tree learned offline constitutes a common ground on which they can base their decisions in order to reduce the amount of communication necessary to maintain the coordination between them.

In the next section, we present some experiments in which we described in more detail how the learned trees can be 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 show how our learning algorithm can be used in the RoboCupRescue simulation to help the *FireBrigade* agents to coordinate themselves, in order to select which fires to extinguish.

Since there could be a lot of fires, agents do not consider all fires at once. As previously stated in Section 2, the *FireStation* agent has a better global view of the situation and therefore it can suggest fire areas to *FireBrigade* agents. Fire areas are simply groups of close buildings on fire. Figure 3 shows an example of a situation with four fire areas. The

Figure 3: Example of fire areas. There are four active fire areas on this map, which are identified by the circles. This figure should be seen in color.

FireBrigade agents have however a more accurate local view, consequently they choose which particular building on fire to extinguish in the suggested area. By doing so, one can take advantage of the better global view of the *FireStation* agent and the better local view of the *FireBrigade* agents at the same time.

The *FireStation* agent uses the tree to evaluate the maximum number of agents needed to extinguish a fire in each area, while the *FireBrigade* agents use the tree to coordinate each other on priority fire in their indicated sector. Both types of agents use the tree created offline (as shown in Algorithm 2) to estimate the required number of agents for each evaluated building. All agents have the same tree and it does not change during a simulation.

In the next two sub-sections, we present in detail how the *FireStation* agent chooses the fire areas and how the *FireBrigade* agents choose the buildings on fire to extinguish.

5.1 Fire Areas Allocation

To allocate the fire areas, the *FireStation* agent has a list of all fire areas; see Algorithm 3. For each fire area, it has to estimate the number of agents that are required to extinguish this area. To achieve that, it makes a list of all the buildings that are at the edge of the

fire area (line 8 of Algorithm 3). 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 *FireStation* agent determines the number of agents required to extinguish the fire. This agent then estimates the required number of agents for the fire area as the maximum number of agents returned for one building in the area (to do that, line 12, Algorithm 3 calls Algorithm 2). The *FireStation* agent does the same with all fire areas, ending up with a number of agents (n_z) for each area z.

Afterwards, for each area z, the *FireStation* agent calculates the average distance of the n_z closest *FireBrigade* agents from z (line 17). Then, it chooses the fire area z with the smallest average distance. Although some routes may be blocked or be inexistant, causing the Euclidean distance to be actually an inexact metric, the algorithm just uses it as a clue to prioritize the areas since the *FireStation* is limited in its perception of the real situation. Consequently, the n_z closest *FireBrigade* agents from the chosen area z are assigned to z. The *FireStation* agent then removes the assigned area and *FireBrigade* agents from its lists (lines 25-26) 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 its assigned fire area.

5.2 Choice of Buildings on Fire

To choose a building to extinguish, a *FireBrigade* agent builds a list of all the buildings on fire in the fire area specified by the *FireStation* agent (see Algorithm 4). This list is sorted according to a utility function that gives an idea about the usefulness of extinguishing a fire (line 2). The utility function $U(f_i)$ gives a value to a fire f_i based on the buildings and the civilians in danger if f_i propagates to buildings close by. The utility function considers all buildings in danger by the given fire f_i. Buildings in danger are near buildings which are not on fire, but that may catch fire if fire f_i is not extinguished. For each building in danger, the utility function returns the amount of points lost if the building in danger catches fire. The utility function uses the official score function of the RoboCupRescue simulation environment which is:

$$Score = \left(A + \frac{H}{H_{ini}}\right)\sqrt{\frac{B}{B_{ini}}} \tag{8}$$

where A is the number of living agents, H is the remaining number of health points (HP) of all agents, H_{ini} is the total number of HP of all agents at the beginning, B_{ini} is total buildings' area at the beginning and B is the undestroyed area which is calculated using the fierceness value of all buildings. The fierceness attribute indicates the intensity of the fire and how badly the building has been damaged by the fire. This attribute can take values from 0 to 8, as presented in Table 1. Using these fierceness values, Table 2 presents the rules used to evaluate the unburned area of each building.

Algorithm 3 Algorithm used by the *FireStation* agent to allocate a fire area to each *FireBrigade* agent.

```
1: Function ASSIGN-FIRE-AREAS(Agents, FireAreas) returns a fire area per agent
   Inputs: Agents: a list of all available agents.
           FireAreas: a list of all fire areas.
2: while FireAreas.size() > 0 ∧ Agents.size() > 0 do
3:   nbAgents ← Agents.size()
4:   smallestDistance ← ∞
5:   chosenArea ← null
6:   chosenAgents ← null
7:   for each fireArea in FireAreas do
8:     borderBuildingsList ← GET-BORDER-BUILDINGS(fireArea)
9:     n_z ← 0
10:    for each borderBuilding in borderBuildingsList do
11:      d ← GET-TASK-DESCRIPTION(borderBuilding)
12:      nbRequiredAgents ← NUMBER-AGENTS-REQUIRED(d, nbAgents)
13:      if n_z < nbRequiredAgents then
14:        n_z ← nbRequiredAgents
15:      end if
16:    end for
17:    ⟨averageDistance, listAgents⟩ ← AVERAGE-DISTANCE(Agents,n_z)
18:    if smallestDistance > averageDistance then
19:      smallestDistance ← averageDistance
20:      chosenArea ← fireArea
21:      chosenAgents ← listAgents
22:    end if
23:  end for
24:  agentsAssigned[chosenArea] ← chosenAgents
25:  FireAreas ← FireAreas − chosenArea
26:  Agents ← Agents − chosenAgents
27: end while
28: return agentsAssigned
```

More formally, the utility function $U(f_i)$ is calculated using the following equations:

$$U(f_i) = \sum_{b \in D(f_i)} (Score_{ini} - ScoreLost(b)) \tag{9a}$$

$$Score_{ini} = \left(A + \frac{H_{ini}}{H_{ini}}\right) \sqrt{\frac{B_{ini}}{B_{ini}}} = A + 1 \tag{9b}$$

$$ScoreLost(b) = \left(A - nCiv(b) + \frac{H_{ini} - sumHP(b)}{H_{ini}}\right) \sqrt{\frac{B_{ini} - area(b)}{B_{ini}}} \tag{9c}$$

where, $D(f_i)$ is the set of all buildings in danger from the fire f_i, $Score_{ini}$ is the initial score at the beginning of the simulation, $ScoreLost(b)$ is the score lost if the building in danger

Algorithm 4 Algorithm used to choose a fire in the fire area specified by the *FireStation* agent.

```
1: Function CHOOSE-FIRE(Fires, nbAgents, rank) returns a fire
   Inputs: Fires: a list of all fires.
           rank: the agent's rank in the group of FireBrigade agents.
           nbAgents: the number of agents assigned to the same fire area.
2: sortedFires ← SORT-FIRES(Fires)
3: currentIndex ← 0
4: for each fire in sortedFires do
5:    d ← GET-TASK-DESCRIPTION(fire)
6:    nbRequiredAgents ← NUMBER-AGENTS-REQUIRED(d, nbAgents)
7:    currentIndex ← currentIndex + nbRequiredAgents
8:    if currentIndex ≥ rank then
9:       return fire
10:   end if
11: end for
12: return first fire in sortedFires
```

b catches fire, $nCiv(b)$ returns the number of civilians trapped in b, $sumHP(b)$ returns the sum of the health points (HP) of all the civilians trapped in b and $area(b)$ returns the area of b.

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 (to do so, at line 6, Algorithm 4 calls Algorithm 2).

The *FireBrigade* agents choose 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 rank 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

Fierceness	**Meaning**
0	Intact building.
1	Small fire.
2	Medium fire.
3	Huge fire.
4	Not on fire, but damage by the water.
5	Extinguished, but slightly damage.
6	Extinguished, but moderately damage.
7	Extinguished, but severely damage.
8	Completely burned down.

Table 1: Meaning of the building's fierceness attribute values in the RoboCupRescue simulator.

Fierceness	Score rules
0	No penalty.
1 or 5	$\frac{1}{3}$ of the building's area is considered destroyed.
4	Water damage, also $\frac{1}{3}$ of the building's area is considered destroyed.
2 or 6	$\frac{2}{3}$ of the building's area is considered destroyed.
3, 7 or 8	The whole building is considered destroyed.

Table 2: Score rules used to evaluate the area burned based on the building's fierceness attribute.

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 the learning phase and tests on a situation with a lot of fires, but with all roads cleared. The simulations started with 8 fires, but the agents began to extinguish fires only after 30 simulation steps (to allow fires to propagate). Figure 4 shows a view of the city at time 30, just before the *FireBrigade* agents begin to work. 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 different groups of tasks that permit to predict as precisely as possible the expected rewards associated with those tasks.

In Section 3 we introduced our model under the form $< D, N, R, T >$, with D is the set of all possible fire descriptions. In our experiments, a task was described by the fire's fierceness (3 possible values); the building's composition (3 possible values); the building's size (continuous value) and; the building's damage (4 possible values). N is the number of possible *FireBrigade* agents, fixed to 15. R is the reward function which was determined by the Equation 3 with r_t as being set to 100 for an extinguished fire and 0 otherwise. Finally, T is the transition function which was estimated using the instances stored in the leaves of the three as explained previously in Equation 4.

All these attributes lead to a number of possible instances which is quite important. In fact, with the continuous attribute "building's size", there is an infinite number of instances. However, since the learning simulations were done on the same map, the number of different buildings was only 730 in our tests. In the case where we consider 15 agents, the number of possible instances for our tests climbed up to 394 200 (i.e., $3 \times 3 \times 730 \times 4 \times 15$).

We have compared the results obtained by our agents with two other strategies, as de-

Figure 4: Initial situation. This figure should be seen in color.

scribed in Figure 5. The first one is the strategy of the team ZNU[1] which finished first at the 2008 RoboCupRescue simulation world competition. If we look at the performance of ZNU team approach (implemented on our PC just for the comparison), it only obtained an average percentage of 61% of intact buildings.

The second one is our strategy, but without the learning part, thus all agents chose the first building on the list. The comparison described in Figure 5 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 64% of intact buildings and after learning, they obtained 85%. This is a substantial improvement showing that the information learned is really useful. Notice that the substantial improvement 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 performance with trees having less than 2000 leaves. Therefore, they just distinguished 2000 task descriptions out of the 394 200 possible task descriptions. In other words, our agents were able to

[1]Zanjan University, IRAN.

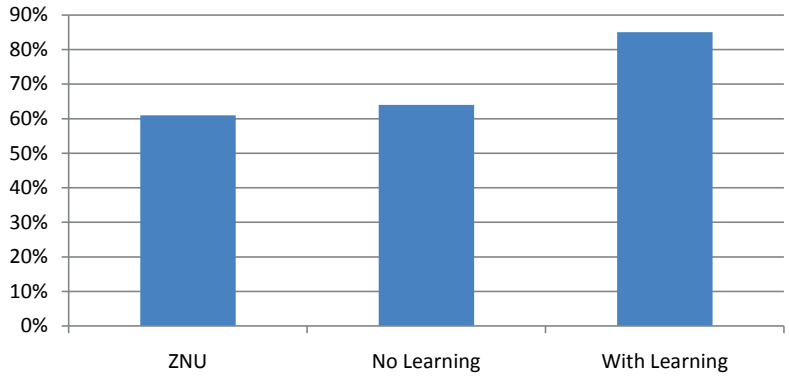

Figure 5: Comparison with other strategies.

perform efficiently with an internal task description space of only 0.5% of the complete task description space (the percentage is the number of leaves in the tree divided by the number of possible task descriptions). This shows a very good reduction of the task description space, enabling the learning algorithm to work on an easier problem with less possible states.

Moreover, Figure 6 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 all γ values from 0 to 1 with an increment of 0.1, but here, for an improved visibility, 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 learning 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. The best γ value we found was 0.5. Since the global performance decreases with γ values inferior to 0.5, it shows that it is important to consider future rewards. However, since we obtained better results with smaller γ values, it shows that it is not efficient to consider rewards too far away. This is the case in the RoboCupRescue simulation, because the simulation evolves fast and the fires have to be extinguished rapidly. With bigger γ values, the agents choose buildings that take too much time to extinguish, because they consider far rewards.

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 did not take too much space, this was not really a problem. Moreover, after the learning phase, the instances are not required 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 7).

During all the simulations, agents use the reward threshold to know what is the right allocation scheme. A too low threshold involves an ineffective dispersion of the agents on

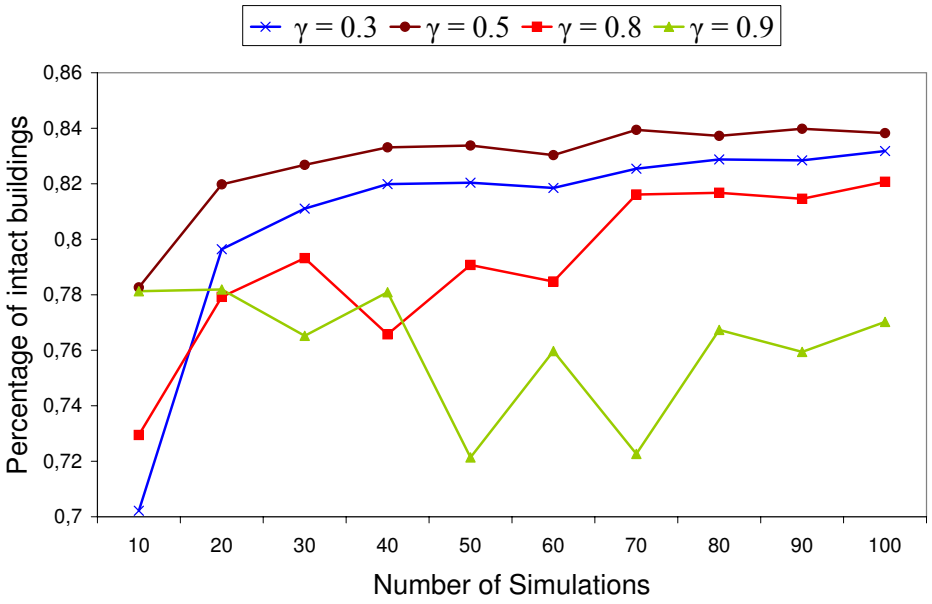

Figure 6: Percentage of intact buildings over 100 simulations for different γ values.

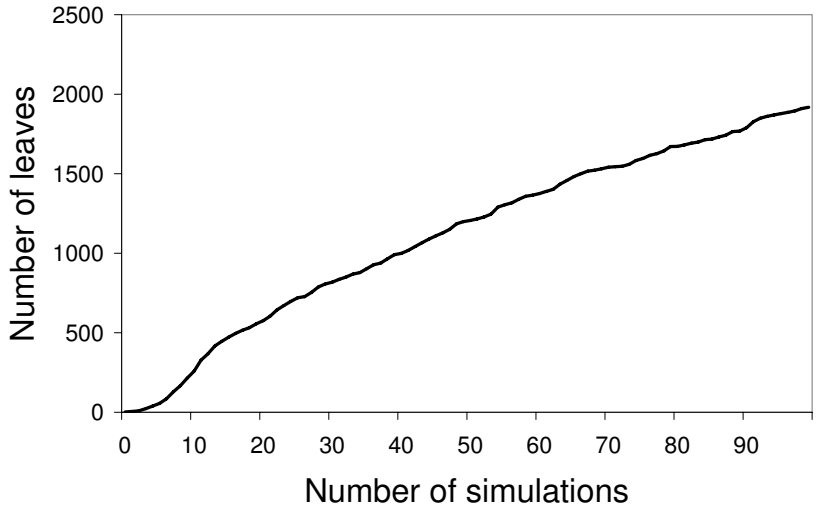

Figure 7: Number of leaves in the tree over 100 simulations.

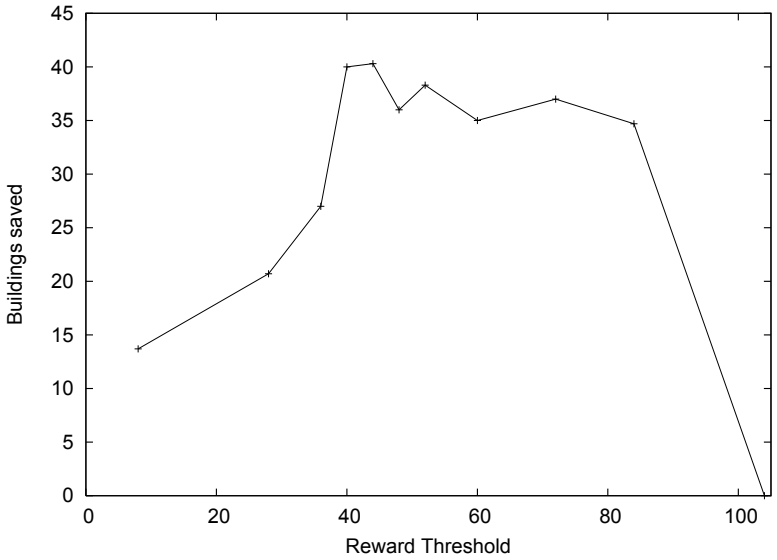

Figure 8: Number of saved buildings for different thresholds

fires. Too high, the tasks are considered impossible and the performance decline also. Since it is fixed arbitrarily, we show that the threshold in our experiments (60 as indicated in Figure 8) lies in the range of the best performances for a realistic scenario consisting of a large random map, with many blockades, 8 fires and 15 agents (see Figure 8). We have made a mean of the performance for 10 simulations on many different thresholds.

Another important result which sustains this approach is our performance in a past RoboCupRescue competition using a preliminary version of our approach. We finished in second place at this competition, really close to first place. One part of our success is due to the fact that we were quite good at extinguishing fires. Table 3 shows the percentage of saved buildings for all the maps used during the competition. In this table, the results of all the semifinalist teams are presented. We can see that our team (DAMAS) outperformed the other teams 5 times, the best number of wins. It also appears that some scenarios were easier for other teams but harder for us. However, during the competition, we got a mean performance comparable to other finalists.

We also conducted complementary experiments on random maps (see Table 4) to see if our approach is generalizable. To this end, we chose large maps because it makes a scenario as difficult as necessary for the variables to have a significative impact. For those maps, there was neither blockades nor civilians, our focus was specifically on the *FireBrigade* agents behavior. With these maps, we compared a scenario with 6 fires and 10 agents to another scenario with 8 fires and 15 agents; fires and agents were uniformly distributed on the map. For each scenario type, we used random maps with buildings in wood and selected building size specifications (Small, Medium and Large). We chose buildings in wood because maps with random building compositions was systematically easier. In our experiments, the buildings size and the number of fires proved to be the most influential factors. In addition to the standard percentage of building surface saved, we also counted the number of extinguished buildings. This is motivated by the fact that the reward function used is related to that task goal.

Map(↓) Competitor(→)	ResQ	Damas	Caspian	BAM	SOS	SBC	ARK	B.Sheep
Final-VC	47,21	54,13	**81,67**	43,19	N/A	N/A	N/A	N/A
Final-Random	24,04	**26,38**	15,03	12,35	N/A	N/A	N/A	N/A
Final-Kobe	38,24	**61,89**	38,38	13,51	N/A	N/A	N/A	N/A
Final-Foligno	**91,15**	62,77	60,92	34,56	N/A	N/A	N/A	N/A
Semi-VC	23,45	23,60	25,49	27,14	19,12	25,10	26,36	**27,22**
Semi-Random	23,18	**28,73**	18,09	19,55	22,82	21,45	17,09	18,91
Semi-Kobe	**96,49**	76,76	94,32	95,41	24,32	90,54	55,27	94,19
Semi-Foligno	36,22	**38,06**	32,72	37,79	31,89	28,48	26,82	23,23
Round2-Kobe	70,27	37,03	59,73	95,41	48,38	61,49	10,54	**95,54**
Round2-Random	99,04	60,91	54,68	99,16	63,55	97,60	80,70	**99,52**
Round2-VC	10,23	11,57	10,23	13,53	12,67	**71,99**	N/A	36,51
Round1-Kobe	99,46	98,92	**99,73**	**99,73**	99,05	98,78	67,16	91,89
Round1-VC	97,25	99,53	79,70	**99,76**	N/A	98,90	99,53	99,53
Round1-Foligno	**98,99**	**98,99**	36,13	45,99	32,53	54,29	43,59	29,86
Number of Wins	3	**5**	2	2	0	1	0	3
AVG (Average)%:	61,09	55,66	50,49	52,65	39,37	64,86	47,45	61,64
STD (Standard Deviation) %:	37,80	34,11	31,83	37,50	27,28	31,63	30,49	36,70

Table 3: Percentage of saved buildings during a recent RoboCupRescue international competition

Results reported by Kleiner et al. [12].

		% of Surface Saved		Nbre of Exting. Bldgs	
Fires-Agents(↓)	Buildings	AVG	STD	AVG	STD
6-10	Small	19.0	7.4	53.0	6.4
	Medium	34.7	18.0	42.6	7.6
	Large	64.1	7.14	40.0	7.3
8-15	Small	9.5	8.1	57.4	11.4
	Medium	19.5	10.1	45.9	5.4
	Large	41.6	11.6	42.1	8.0

Table 4: Saved buildings on random maps. AVG = Average and STD = Standard Deviation

Results shown on Table 4 indicate that higher percentages of surface saved are obtainable on maps with large buildings, even if the number of extinguished fires is higher with small ones. In this latter case, the fire spreads more rapidly to many other small buildings, thus, letting the agents loose control. The performance is also affected by the number of fires. Even if the team is composed of more agents, we have about the same number of extinguished fires. Thus, large buildings implies a significantly reduced saved surface. Finally, we may note that the standard deviation indicate a relatively stable performance under some fixed parameters for a complex multi-agent system.

To sum up, the comparison of our performance with other non learning strategies, in terms of building surface saved, showed us that the learned tree can help the agents to

allocate themselves effectively on the most important fires. But in some circumstances, the agents loose control where they could not. The first issue arises with the threshold used as a sufficient expected reward, but not an "optimal" one. The use of a threshold was motivated by the constrained environment and although we gave a preliminary empirical method to determine it, it is undoubtedly subject to improvements in future works. By example, we could seek the minimum number of agents necessary to get the maximum possible expected reward for a given priority task. The second major issue is about the priority function. The fire propagation risk is taken into account by this function, but not considered as a cost for the reinforcement signal. We could investigate the possibility of transferring some observations about surrounding buildings in the task description and consequently modifying the reward function.

7 Related Work

In multiagent domains, different methods for task allocation have emerged during the last decade. Among these methods, those aiming at solving the following problem: given a set of agents and a set of tasks which they have to execute, how to form a group of agents that can perform each task, knowing that those tasks cannot be performed by a single agent? One method referring to the formation of such groups of agents is the formation of coalitions [2, 23, 24, 30].

In the coalition approach, a task to be completed is proposed to the agent population. An agent can take the initiative and attempts to form a coalition. In the RoboCupRescue, this role can be devoted to the *FireStation* for instance. Before proposing a coalition to *FireBrigade*, this agent must determine the optimal set of agents which can form the "optimal" coalition for each specific task (i.e., fire to extinguish). We refer to this activity as a *coalition calculation*. To determine the optimal coalition, the agent in charge of that, must have a means of calculating the value of any given coalition. To achieve that, multiple metrics can be used: cost, time, quality, etc. and all these metrics should refer to abilities of agents and/or the characteristics of the different tasks to achieve. Using metrics in the coalition calculation can cause problems [22] as: conflicting metrics, differing metric importance and the variation of metric importance over time. Furthermore, knowing a priori the characteristics of tasks is not easy in dynamic environments as RoboCupRescue where the characteristics of fire can change, where the interactions between all the components of this environment, can be continuously modified, and where external conditions can be unexpectedly changed. In this case, it would be impossible to use coalitions and the only solution appears to be a solution based on learning and adaptation as we did in this paper.

In the same context, auctions have been suggested for allocation of computational resources since the 60s. They are traditionally used with self-interested agents, but there is a growing body of work on using auctions to allocate tasks among cooperative agents [5, 26]. Auction-based methods for allocation of tasks are becoming popular in robotics as an alternative to other allocation methods [6, 9, 10, 15].

Using auctions in our context needs to know all the characteristics, including the required number of *BrigadeFire* agents, attached to each fire to extinguished. In the case where the number of *BrigadeFire* agents is known *a priori*, each fire can then acts as auctioneer and calls for bids, from the *BrigadeFire* agents, according to a first-price reverse auction. However, in dynamic environments as RoboCupRescue where the characteristics of fire can change,

where the interactions between all the components of this environment, can be continuously modified, and where external conditions can be unexpectedly changed, a priori definitions can be very difficult, and sometimes impossible. In this case, the robots require learning and adaptation in response to dynamic events. The work presented in this paper is a first approach in this direction.

Another related work turns around the determination of a number of agents in order to obtain a better coordination. In this context, Dutech et al. [7] have introduced the incremental reinforcement learning (IRL) which consists of progressively increasing the problem's complexity. By doing so, IRL can use the solutions of the simpler tasks to find better solutions for the more complex tasks [7]. This incremental learning is done along two axes: gradually increasing the number of agents and gradually increasing the tasks' complexity. The authors have shown that they can obtain better results with the incremental algorithm than with a standard reinforcement learning algorithm. Another similar approach called *behavior transfer* (BT), developed by Taylor and Stone [27], uses the information learned in simpler tasks to accelerate the learning of more complex tasks.

One drawback of these two approaches, devoted to IRL and BT, is that they are dependant on the problem, because the system designer has to define all the progressive steps. In some problems as RoboCupRescue in particular, simpler tasks are not so apparent and we should do a deep analysis to rise them. Once such tasks well surrounded, it would be beneficial to use our approach for these simpler tasks and complete it by IRL and BL approaches for much more complex tasks.

Finally, approaches to task allocation have also extensively used different versions of the contract net protocol [25]. This protocol has been one of the most influential cooperation approaches proposed for multiagent systems [31]. By using it, agents coordinate their activities through contracts to accomplish specific tasks. More specifically, an agent acting as a *manager*, decomposes its contract (the task or problem it was assigned with) into sub-contracts to be accomplished by other *potential contractor* agents. This protocol has however a lot of communication overhead due to the broadcast of the task announcements and its performance degrades drastically when the number of communicating agents and the number of tasks announced increases. Adding to this problem of scalability, it is also important to notice that it is dependent on the reliability and bandwidth limitations of communication. Consequently, this type of protocol is not applicable to dynamic environments as RoboCupRescue where the communication are very limited.

8 Conclusion

In this article, we have presented a learning algorithm which allows to determine the necessary number of agents for each different task in a complex cooperative multiagent environment. In the past, most coordination learning approaches considered that the number of required number of agents to accomplish a task was known. In our approach, we consider that the tasks are complex and that the agents have to learn this information particularly when it is not available.

The tasks considered in our simulations are described with discrete and continuous attributes. Therefore, there are a lot of possible task descriptions. To manage this complexity, we have adapted a selective perception reinforcement learning algorithm to the problem of learning the good allocation scheme. With this algorithm, it is easy to find a generalization

of the task description space, which helps the reinforcement learning algorithm to work on smaller task description spaces.

We also proposed a coordination algorithm using the information learned about the number of resources needed for a task. This algorithm uses really few messages between the agents, which is interesting in environments with limited and/or unreliable communications like the RoboCupRescue simulation.

Finally, we have presented some tests in the RoboCupRescue environment showing that the agents can efficiently learn and that the learned information is really helpful to improve the agents' performances. The agents obtained good results with an internal task description space of only 0.5% of the complete task description space. We have also shown results taken during an international competition showing that we had the most number of wins and a good average performance in terms of saved buildings. It figures out that our approach performed well in the competition, and that promising improvements are expected from future works. Among these improvements one could seek the minimum number of agents necessary to get the maximum possible expected reward for a given priority task. It would be also interesting to define a measure of performance that could automatically find the best depth of the tree. This measure of performance would have to balance the quality of the state's distinction with the time needed by the reinforcement learning algorithm if there are more states. Finally, in case where we have experts knowledge it would be very useful to elicit such a knowledge [4] and to see how it can complete our selective perception algorithm.

We should note that our approach might be used with success in (i) task allocation in the context of multiagent systems; (ii) task allocation in computational grids and similar systems; (iii) logistics applications where it is important to manage the flow of goods, information and other resources including energy and people; (iv) rescue and military applications.

Acknowledgement

This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). We greatly thank the anonymous reviewers for their valuable suggestions.

References

[1] C. Boutilier. Planning, Learning and Coordination in Multiagent Decision Processes. In *Proceedings of TARK-96: Theoretical Aspects of Rationality and Knowledge*, De Zeeuwse Stromen, Hollande, 1996.

[2] G. Chalkiadakis and C. Boutilier. Bayesian Reinforcement Learning for Coalitional Formation under Uncertainty. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, Ney York, USA, 2004.

[3] D. Chapman and L. P. Kaelbling. Learning from delayed reinforcement in a complex domain. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[4] T.Q. Chu, A. Drogoul, A. Boucher, and J.D. Zucker. Interactive Learning of Independent ExpertsŠ Criteria for Rescue Simulations. *Journal of Universal Computer Science*, 15(13):2701–2725, 2009.

[5] P. Cramton, Y. Shoham, and R. Steinberg. An overview of combinatorial auctions. *ACM SIGecom Exchanges*, 7(1):3–14, 2007.

[6] M.B. Dias and A. Stentz. A Free Market Architecture for Distributed Control of a Multirobot System. In *Proceedings of the Int'l Conference on Intelligent Autonomous Systems*, pages 115–122, Venice, Italy, 2000.

[7] A. Dutech, O. Buffet, and F. Charpillet. Multi-Agent Systems by Incremental Gradient Reinforcement Learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence IJCAI-01*, pages 833–838, Seattle, 2001.

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

[9] B. Gerkey and M. J. Mataric. Multi-robot Task Allocation: Analyzing the Complexity and Optimality of Key Architectures. In *Proceedings of the Int'l Conference on Robotics and Automation (ICRA'03)*, 2003.

[10] N. Kalra, R. Zlot, and B. Dias. Market-based multirobots coordination: A comprehensive survey and analysis. Technical Report CMU-RI-TR-05-16, Robotics Institute, Carnegie Mellon University, PA, 2005.

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

[12] A. Kleiner, M. Brenner, T. Brauer, C. Dornhege, M. Gobelbecker, M. Luber, J. Prediger, J. Stuckler, and B. Nebel. Successful Search and Rescue in Simulated Disaster Areas. In Itsuki Noda, Adam Jacoff, Ansgar Bredenfeld, and Yasutake Takahashi, editors, *RoboCup-2005: Robot Soccer World Cup IX*. Springer Verlag, Berlin, 2006.

[13] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, New-York, 1996.

[14] A. W. Moore. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state spaces. In *Proceedings of Advances of Neural Information Processing Systems (NIPS 6)*, pages 711–718. Morgan Kaufmann, 1993.

[15] M. Nanjanath and M. Gini. Auctions for Task allocation to robots. In *Proceedings of the Int'l Conference on Intelligent Autonomous Systems*, 2006.

[16] S. Paquet, N. Bernier, and B. Chaib-draa. Multi-attribute Decision Making in a Complex Multiagent Environment Using Reinforcement Learning with Selective Perception. *Proceedings of The Seventeenth Canadian Conference on Artificial Intelligence, London, Canada*, 2004.

[17] S. Paquet, N. Bernier, and B. Chaib-draa. Selective Perception Learning for Tasks Allocation. In *AAMAS-04 Workshop on Learning and Evolution in Agent Based Systems*, New York, 2004.

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

[19] 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.

[20] D. Ron, Y. Singer, and N. Tishby. Learning probabilitic automata with variable memory length. In *Proceedings of Computational Learning Theory*. ACM press, 1994.

[21] S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education, Upper Saddle River, New Jersey, second edition, 2003.

[22] T. Scully, M. G. Madden, and G. Lyons. Coalition Calculation in a Dynamic Agent Environment. In *Proceedings of the 21th Int'l Conference on Machine Learning (ICML'04)*, 2004.

[23] O. Sheory and S. Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101:165–200, 1998.

[24] M. Sims, C.V. Goldman, and V. Lesser. Self-organization through bottom-up coalition formation. *Proceedings of the Autonomous Agents and Multiagent Systems (AAMAS'03), Melbourne, Australia*, 2003.

[25] R.G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[26] M. Ströbel. On Auctions as the Negotiation Paradigm of Electronic Markets Success Factors, Limitations and Research Directions. *EM Journal of Electronic Markets*, 10(1):39–44, 2000.

[27] 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 (AAMAS-2005)*, pages 53–59, July 2005.

[28] W. T. B. Uther and 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.

[29] W.T.B. Uther and M.M. Veloso. TTree: Tree-Based State Generalization with Temporally Abstract Actions. *Proceedings of SARA-2002*, 2002.

[30] L. Vig and J.A. Adams. Coalition formation: From software agents to robots. *Journal of Intelligent and Robotic Systems*, 50(1):85–118, 2007.

[31] M. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002.