Ad hoc Team Formation: Applying Machine Learning Techniques to Cooperate without Pre-Coordination

Themistoklis Diamantopoulos

Master of Science
Computer Science
School of Informatics
University of Edinburgh
2012

Abstract

Nowadays, Multi-Agent Systems are widely applicable to various real-life situations, such as confronting cooperative problems. However, the agents are usually a priori coordinated towards their common goals. In situations where no prior coordination is possible, the problem is described as an ad hoc team setting. This dissertation considers successfully confronting the problem of creating an ad hoc agent able to cooperate with his teammates without being a priori aware of their strategies. Upon providing an overview of Machine Learning, the main lines of research concerning the ad hoc team setting are reviewed and the problem is decomposed to three subproblems. Firstly, the ad hoc agent has to construct models of his teammates' strategies, then he has to be able to determine which of the constructed models are followed by each of his teammates, and finally he has to devise an efficient strategy in order to successfully cooperate with them. Concerning the construction of teammates' models, the problem is reduced to classifying any teammate's action given the system state. A methodology for extracting data instances from observed state-action pairs is described, and upon training an appropriate classifier, the complementary task of mapping the classifier's output to valid actions given a current state is considered. The problem of determining which model is followed by a teammate is confronted using the Naïve Bayes classifier which estimates the probability of a model given the actions of the teammate. Finally, the strategy construction task is accomplished using Q-learning policies as answer models to any combination of teammates' models. A novel approach to the computational complexity of having multiple teammates is described; the ad hoc agent devises an answer model for each possible teammate model, and merges the models operating on their Q-values. Finally, the implementation is tested for its effectiveness and efficiency in a search-and-rescue testbed. Furthermore, an analysis of the agent's learning parameters is performed and agent teams with multiple ad hoc agents are considered. The results are quite promising and shall certainly call for future research.

Keywords: Multi-Agent Systems, Ad hoc Teams, Agent Cooperation, Machine Learning, Reinforcement Learning

Acknowledgements

I would like to take the opportunity to express my gratitude to the people that have supported me in accomplishing this dissertation. Each in their own way have had significant influence on the course of this project.

First and foremost, I would like to thank my supervisor, Dr. Michael Rovatsos, for his trust in me to accomplish this dissertation. His guidance, support and constructive criticism helped me to overcome the difficulties of this project.

I would also like to express my gratitude to my girlfriend, Ria, for her emotional support, understanding, and patience throughout all phases of this dissertation.

I am sincerely grateful to my family. Without their continuous encouragement and unconditional support, I would not be able to perform this work.

Last, but by no means least, I would like to thank my friends for their support in completing this project.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Themistoklis Diamantopoulos*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

During the past few decades, there has been an increasing interest on researching the capabilities and applications of *intelligent agents*. According to S. J. Russell et al. [2], an *agent* is defined as an autonomous entity which interacts with his environment and plans his following moves so as to achieve his goals. Much of the research has mainly focused on what are called *Multi-Agent Systems (MAS)*, i.e. systems consisting of multiple intelligent agents. Depending on the desired setting, agents in such systems may need to cooperate or compete against each other.

In cooperative MAS settings, successful coordination among the agents is usually the key to achieving satisfactory results. Most MAS techniques attempt to attain a desirable coordination level by optimally exploiting prior knowledge about the agents that compose the team. However, real-world situations may be unpredictable. For example, consider playing football with unknown teammates; one has to successfully cooperate with his teammates in order to achieve the common goal (win the game), without having any a priori information about their abilities, style of play, etc.

1.2 Scope of the Dissertation

Considering the football scenario given in the previous subsection, the players may have different abilities. For example, one of them may be strong and good at tackling, whereas another one may be slow but competent shooter. However, they do not know each other and thus they are unaware of each other's abilities. Furthermore, the foot-

ballers might not even speak the same language. Thus, each team has to perform well in order to win without any prior coordination.

The situation described above is actually an ad hoc team formation problem. The problem was posed by P. Stone et al. [3] as the design of an agent capable of efficiently collaborating with unknown teammates without any prior coordination. Apart from being applicable to various human problems, ad hoc problems usually arise in agent (or robot) coordination situations.

Lately, the research conducted on agents (or robots) has given rise to multiple applications, such as production lines, auctions, search-and-rescue situations etc. However, most agents are developed without any team-aware extensibility other than the one initially given to them. The cost and effort of maintaining existing agent systems or creating new agents from scratch could be drastically reduced, by initially implementing them with respect to cases where an agent has to effectively extend an existing team.

1.3 Aim of the Dissertation

This dissertation describes the implementation of an ad hoc agent able to address efficiently and effectively the challenge of successfully extending a team of agents that performs a specific task. In particular, the agent's effectiveness concerns whether his team performs near-optimally when compared to a fully-coordinated team. In addition, the ad hoc agent has to be efficient in terms of computational complexity, so that he is capable of addressing real-time situations with minimum delay. The design of the agent extends existing work on the area by providing with insightful ideas concerning modeling the policies of teammate agents, identifying the policies followed by them, and constructing an effective response strategy.

In terms of evaluation, a *search-and-rescue (SAR)* domain is considered, where multiple agents with diverse abilities (e.g. robots which may have been created from different manufacturers) have to coordinate in order to achieve their common goal, i.e. rescuing people. SAR simulators are widely acceptable as satisfactory testbeds, developed and improved constantly by the community (e.g. Robocup [4, 5]). However, a simple simulator was implemented to effectively match the project's specifics.

As opposed to the above specific testbed, effort has been made so that the methods designed are as generic as possible, in order to ensure broad applicability. Machine Learning algorithms are used to confront the various challenges that the agent faces.

The project's primary objective is the design of an agent that can effectively take part in a MAS without any prior coordination. Another secondary objective is to explore the effectiveness of such techniques when the system comprises not only by fixed but also by ad hoc learning agents. Finally, since the ad hoc agent does not follow fixed policies, efficient tweaking of the various learning parameters is studied.

1.4 Overview of the Chapters

In accordance with the general aim of this dissertation as well as the methods used to successfully confront the various issues that arose, the dissertation is split into 7 chapters[1].

Chapter 2 of this dissertation is a brief review of the general field of Machine Learning, while mainly focusing on areas and techniques that were deemed useful for the implementation of the project.

Chapter 3 reviews the research conducted on the problem of ad hoc team formation. The various approaches on the subject are discussed and their relation to the current work is pointed out.

Chapter 4 thoroughly describes the methodology used for constructing the proposed ad hoc agent, upon describing each discrete component of the strategy.

Chapter 5 briefly introduces the concepts of SAR domains and describes the SAR testbed used to evaluate the ad hoc agent that was implemented. The applicability of the main components of the agent is demonstrated and the evaluation framework is defined.

Chapter 6 provides an overview of the experiments conducted, analyzing the appropriate metrics and describing the objectives of the various experiments. In addition, the results are interpreted and their significance is discussed.

Chapter 7 discusses whether the project objectives are accomplished as well as the extent to which the methods implemented are effective and efficient. In addition, ideas for future work on the domain are provided.

Finally, Appendices A and B provide with information about the implementation specifics of the SAR simulator as well as the various agents that were created. Appendix C contains detailed results of the experiments that were conducted.

[1] Parts of the material of introductory and background sections have been presented before in terms of the Informatics Research Review (IRR) or the Informatics Research Proposal (IRP). However, their scope and proportion is insignificant when compared to the main contributions of this dissertation.

Chapter 2

Background on Machine Learning

2.1 Overview

According to T. M. Mitchell [6], *Machine Learning (ML)* is the field of Artificial Intelligence which concerns constructing computer programs that automatically improve with experience. A typical ML problem concerns generating a function that maps inputs to desired outputs. Although the field of ML is remarkably broad, the various ML approaches to the aforementioned problem could be roughly categorized as follows:

- *Supervised Learning*
  The algorithms of this area are given as input a so-called *supervised* or *labeled* set of data, which they use to construct the function. The function can then be tested on a non-labeled test set (see Section 2.3).

- *Unsupervised Learning*
  Unsupervised ML algorithms attempt to construct the model function without any labeled training set of data. This line of work typically concerns estimating certain properties of the data.

- *Semi-Supervised Learning*
  Combines the two aforementioned categories, usually as an extension to unsupervised learning, where a relatively small labeled dataset is given.

- *Reinforcement Learning*
  The algorithms of this category learn to adapt their behavior upon observing their environment (see section 2.4).

The above taxonomy is by no means definitive, since not only there are more areas in ML, but also several ML methods may overlap instead of concretely belonging to one specific category. ML has applications in several domains, solving diverse problems.

In terms of this dissertation, there are two main issues that are addressed by ML techniques. The first is the problem of creating a model that can classify data according to certain features. The second is the problem of acting intelligently in a specific environment that provides rewards. Both problems are analyzed and handled in the following sections, along with a minimal reference to the clustering problem, since certain of its techniques are applicable in this dissertation.

2.2 Unsupervised Learning

2.2.1 Overview – The clustering problem

Unsupervised Learning (USL) is the field of ML that concerns finding specific structure in data that is unlabeled, i.e. no information is provided for it. One of the most well-studied problems of USL is the *clustering* task (also known as *cluster analysis* task). Clustering is the task of distributing instances of data to *clusters*, such that each cluster has similar instances. The term "similar" refers to a particular attribute of the data.

There are several lines of research concerning optimally clustering data instances. Building a clustering model may be based on concentrating instances around *centroids* (*Centroid-based Clustering*), distributing them statistically (*Distribution-based Clustering*), or simply connecting "nearby" instances (*Connectivity-based Clustering*). The aforementioned provide a rather small part of all the clustering approaches. However, analyzing the various clustering techniques is redundant in terms of this dissertation. Hence, the k-means clustering algorithm is the only algorithm analyzed here, not only because it provides a typical example but also because it is an essential part of this dissertation.

2.2.2 k-means Clustering

One of the most well-known Centroid-based Clustering techniques is the *k-means clustering* algorithm, also known as *Lloyd's algorithm* due to its creator [7]. The algorithm distributes data points into k clusters, according to the centroid of each cluster.

Initially, the centroids are given some random values[1]. During the first step, the algorithm runs for each data point and computes its distance from all centroids. Thus, the point is assigned to the cluster which has the closest centroid. During the second step, the new centroid of each cluster is computed as the average of its data points. The aforementioned steps are repeated until the algorithm converges, i.e. until the centroid values do not change.

The *k*-means algorithm is applicable to problems with multi-dimensional data. An example for its application on a two-dimensions problem is shown in Figure 2.1. The

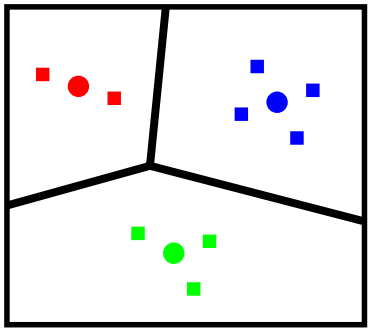


Figure 2.1: An example final state of the *k*-means algorithm, when applied to a two-dimensional problem. Squares are data instances and circles are centroids. Clusters are distinguished by their color.

color of the data points (represented as squares) determines the cluster at which they are assigned. The circles are the centroids of the clusters. The position of the centroids is found by computing the average positions of the data points.

2.3 Supervised Learning

2.3.1 Overview – The classification problem

Supervised Learning (SL) techniques are very interesting because of their remarkable applicability to real-life problems. One of the most well-known applications of these techniques is the *classification problem*.

The definition of the classification problem is quite simple: construct a model that can classify data according to a class attribute. At first, a set of labeled data instances (hereafter referred to as the *training set*) is given to the algorithm. Each data instance

[1] If there is some perception about the data, it is possible that the centroids are given specific values that accelerate the clustering process.

consists of a set of values that correspond to certain *attributes* of the data, one of which is called the *class attribute*. Since the values of all attributes (including the class attribute's) are known, the training set is used to construct the *model*. In simple terms, the model is a function that provides with the value of the class attribute for a data instance given the values of the other attributes of the instance.

Hence, a classifying technique creates a model that actually categorizes (classifies) any instance according to the value of the class attribute. Thus, such algorithms are also know as *classifiers*. As noted by S. B. Kotsiantis [8], no single classification technique outperforms all others. The techniques' accuracy depends on several properties, such as the size of the training set, or the model's tolerance to noise.

Using a taxonomy similar to that given by S. B. Kotsiantis [8], the classification problem is confronted using the following categories of techniques:

- *Desicion tree learning techniques*
- *Neural network techniques*
- *Statistical learning techniques*
- *Support vector machines techniques*

Although the above categories do not cover all the possible classification techniques, they are sufficient not only as representative examples of the literature. Thus, certain algorithms of the above categories are presented and their efficiency is discussed in the following subsections.

2.3.2 Decision Tree Learning Techniques

Decision trees are trees that classify the instances by recursively splitting them based on attribute values. The nodes represent the instance attributes, and the branches represent the attributes' values. The leaf node has the value of the class attribute. A simple decision tree is shown in Figure 2.2. As seen in Figure 2.2, the tree is actually a set of rules[2], deciding whether the weather is good enough for playing tennis or not. The nodes `Outlook`, `Humidity` and `Wind` are attributes of weather data. In this example all attributes are nominal (i.e. their values are drawn from a finite set). The possible values of each attribute are given as branches, e.g. the possible values of the nominal attribute

[2] In fact, according to S. B. Kotsiantis [8], decision trees can be represented with equivalent decision rules, using conditionals (e.g. IF Outlook IS Sunny AND Humidity IS High THEN No).

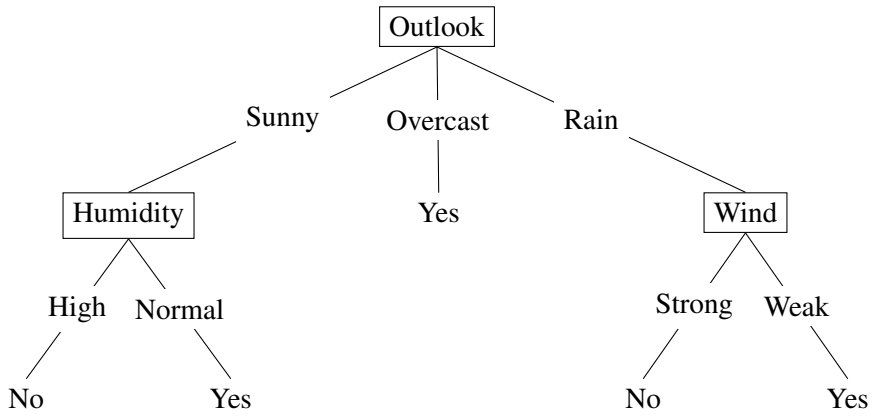


Figure 2.2: An example decision tree determining if the weather is suitable for playing tennis, as seen in [1]. Nodes are attributes and edges are their values.

`Outlook` are drawn from the set {`Sunny, Overcast, Rain`}. The class attribute (not shown in the decision tree of Figure 2.2) is `Weather_Good_For_Tennis` and has two possible values, `Yes` or `No`. Hence, a tree classification algorithm constructs the tree model and given the values of the attributes, it can determine the value of the class attribute. For example, if the values of `Outlook` and `Humidity` are `Sunny` and `Normal` respectively, then the data instance is classified as a `Yes`.

One of the most well-known decision tree algorithms is the *ID3* algorithm[3], created by J. R. Quinlan [1]. The algorithm uses a *TDIDT (Top-Down Induction of Decision Trees)* greedy logic. It starts by selecting a root node and keeps traversing the tree starting from its root and moving towards its leafs. During each traversal, the algorithm determines which node should be selected; the selected node is actually the attribute according to which the data instances are split. The node chosen at each step is the one having the minimum *entropy*. The entropy of the node is given by equation (2.1):

$$Entropy(Node) = -\sum_{n} n \log_2 n \tag{2.1}$$

where n is a possible value of the node's branches (i.e. a possible value of the attribute).

Intuitively, the entropy of an attribute is lower when the distribution of its possible values is skewed. For example, in the tree of Figure 2.3.2, if `Humidity` is `High` 90% of the time (and `Low` 10% of the time), then its entropy is approximately 0.47, whereas if it is `High` 50% of the time (and `Low` 50% of the time), then its entropy is 1. Consequently, the ID3 algorithm actually places the attributes that have unequally balanced distributions closer to the root of the tree.

[3]Means *Iterative Dichotomiser 3*, although the acronym ID3 is used more often.

Although ID3 is effective, it is limited to discrete-value attributes. In addition, it is not very efficient, since it calculates all possible permutations of attributes. A later extension from the same author is the *C4.5* algorithm [9] which resolved the aforementioned issues. In particular, C4.5 also allows continuous-value attributes (as opposed to only discrete-value ones). In addition, the algorithm prunes certain subtrees, replacing them with single nodes, provided that the effectiveness is not severely influenced.

2.3.3 Neural Network Techniques

The field of *Neural Networks (NN)* is based on the notion of the *perceptron* [10]. Addressing the classification problem with a single-layered perceptron is straightforward. The attribute values $x_1, x_2, \ldots, x_n$ are given as input to the model. The values are multiplied each with a weight value (out of $w_1, w_2, \ldots, w_n$) and summed together. Then, their sum is checked for overcoming a threshold t. The output of the perceptron is given by:

$$y = \begin{cases} 1, & \text{if } \sum_i x_i w_i \geq t \\ 0, & \text{if } \sum_i x_i w_i < t \end{cases} \tag{2.2}$$

The training set is used to adjust the weight values and the threshold. Upon training, the network should be able to provide a good estimation of the value of the class attribute y given the several input attribute values. Equation (2.2) is generally known as the activation function of a perceptron, formally defined as a function that receives the sum of the input values multiplied with the respective weights and returns an output value. The function itself may be discrete or continuous and its output typically is drawn from a range that corresponds to the desired output, i.e. the set of the possible values of the class attribute.

Generally, the set of data instances is called *linearly separable* if they can be separated into two subsets by a single hyperplane. In the two-dimensional case the hyperplane is actually reduced to a line. Although single-layer perceptrons are effective for classification of linearly separable instances, they are unable to train optimal models when the instances are not linearly separable. Due to the nature of the perceptron, classifying non-linearly separable sets requires *multiple-layer perceptrons* (also known as *multilayer perceptrons*). The latter fall into the category of *Artificial Neural Networks (ANN)* [11]. Such a network can be seen in Figure 2.3.

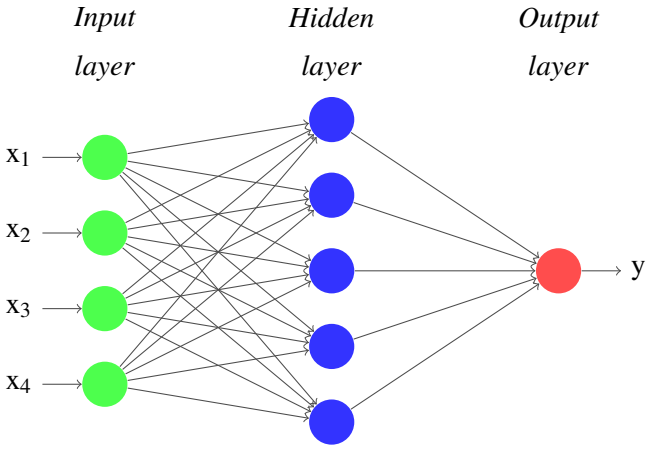


Figure 2.3: An Artificial Neural Network consisting of three layers. The network's functionality depends highly on the hidden layer, thus the number of hidden layers and the number of neurons in those layers are important decisions.

The nodes of the ANN of Figure 2.3 are perceptrons, while the edges include the weights of the network. Intuitively, the weights between the layers represent the influence of the particular perceptron on the output. The multiple layers ensure that the model created by the network is capable of addressing linearly inseparable data. Finally, the activation functions of the network are generally fixed, thus its behavior depends only upon its weights. Describing the various ways of updating the weights lies beyond the scope of this dissertation.

2.3.4 Statistical Learning Techniques

As noted by S. B. Kotsiantis [8], the algorithms of this category actually implement probabilistic models which in turn may be used to classify the instances. The most representative statistical learning technique is the *Naïve Bayes* classifier.

The Naïve Bayes classifier is a generalization of the Bayes theorem for modeling beliefs. Making "naïve" conditional independence assumptions among the attribute values $A_1,\ldots,A_n$, the probability of each value C_i of the class attribute C can be estimated as follows:

$$P(C_i|A_1,A_2,\ldots,A_n) = \frac{\prod_k \left\{P(A_k|C_i)\right\} \cdot P(C_i)}{\sum_j \left\{\prod_k \left\{P(A_k|C_j)\right\} \cdot P(C_j)\right\}} \tag{2.3}$$

Concerning equation (2.3), for any value of the class attribute C_i the *prior* $P(C_i)$ is actually an estimation of the probability of selecting the value[4]. Given the values of all attributes for a set of data instances, $P(A_k|C_i)$ is the probability of an attribute having a value A_k given that the class attribute's value is C_i. This is computable using the training set. Multiplying all such probabilities for all attributes gives the product seen in the nominator of equation (2.3). This product implies that any probability $P(A_k|C_i)$ is independent from any other probability $P(A_l|C_i)$ with $k \neq l$. Although this assumption seems rather naïve, the classifier seems to perform well in certain cases.

The denominator of equation (2.3) is actually used only for normalizing the probabilities of all values of the class attribute to 1. Thus, the Naïve Bayes classifier is effective even if the denominator of equation (2.3) is omitted. Finally, zero probabilities in attribute values may be easily handled by introducing very small values or adding constant numbers to all probabilities before calculating the product (*m-estimate*).

2.3.5 Support Vector Machine Techniques

Support Vector Machines (SVM)[5] are some of the most well-known classification techniques and arguably the state-of-the-art in SL. Original research, conducted by V. N. Vadnik [13], was based on a simple idea: construct a hyperplane that sets apart the classifications. A simple classification example using an SVM technique is shown in Figure 2.4.

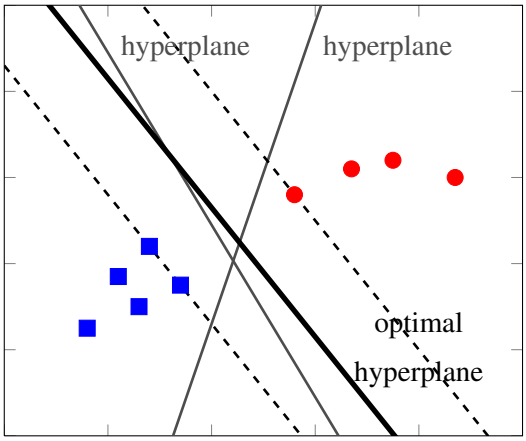


Figure 2.4: A separation example using a Support Vector Machine, for a two-dimensional (i.e. two-attribute) problem. Squares and circles represent data instances that are separated by various hyperplanes.

[4] If no information about the data is known, then the priors of all values may be set to be equal, thus eliminating their influence.

[5] Also referred to as Support Vector Networks, a name given in [12].

The example depicted in Figure 2.4 concerns a two-dimensional space, where data instances are classified according to two attributes providing the dimension values. Example instances are depicted as small squares or circles. Thus, as far as the two-dimensional example of Figure 2.4 is concerned, the hyperplanes are actually reduced to single lines.

As seen in Figure 2.4, there may be various hyperplanes that separate successfully a dataset. SVN techniques try to find the optimal hyperplane, i.e. the hyperplane whose distance from the instances on either side is maximum. This distance is actually known as the *margin*. According to J. C. Platt [14], maximizing the margin comes down to solving the *Quadratic Programming (QP)* problem. The problem specifics lie beyond the scope of this dissertation.

Several researchers have tried to solve the QP problem. An interesting approach is the one followed by J. C. Platt [14], which arrived at the *Sequential Minimal Optimization Algorithm (SMO)*. The latter is widely accepted as the state-of-the-art method to train SVMs.

2.3.6 Choosing the Appropriate Technique

As far as the classification problem is concerned, there is no optimal technique that outperforms all others. All techniques analyzed in this section may be more or less effective depending on certain criteria. For an extensive comparison along several of them, see [8].

For example, while SVMs are probably the most effective techniques in terms of accuracy, the learning procedure is relatively slow. This is also a crucial drawback of ANNs. Thus, when the number of instances or the number of attributes is rather large, it would be preferable to use a decision tree or a statistical learning algorithm. In particular, the C4.5 algorithm is also considered quite effective, while it's also faster.

Finally, there are also cases where the training data is provided incrementally, such as one instance at a time. When it comes down to incremental learning, Naïve Bayes supports it out-of-the-box, whereas the other algorithms need certain modifications that may have an effect on their effectiveness or efficiency.

2.4 Reinforcement Learning

2.4.1 Overview – Markov Decision Process

According to L.P. Kaelbling et al. [15], *Reinforcement Learning (RL)* is the area of Machine Learning which studies the problem faced by an agent that tries to adapt his behavior through trial-and-error in order to successfully interact with a dynamic environment. The field of RL combines two research areas of Artificial Intelligence: *Supervised Learning* and *Dynamic Programming*. As R. Sutton and A. Barto point out [16], Supervised Learning methods are dependent upon a knowledgeable external supervisor that should provide the agent with labeled training sets. Thus, this requirement is eliminated in RL, since the agent learns solely from its own experience.

The general framework of RL is usually modeled as a *Markov Decision Process (MDP)*. MDPs are actually a notion used to effectively model game states. According to early work on the field by R. A. Howard [17][6], an MDP is defined as a tuple:

$$\{S, A(s), P_a(s,s'), R_a(s,s')\}, \quad \forall s \in S,\ a \in A \tag{2.4}$$

where S is a finite set of states, A is a finite set of actions and $A(s)$ is a finite set of possible actions when being at state s, $P_a(s,s')$ is the probability of transitioning from state s to state s' by performing action a, and $R_a(s,s')$ is the reward of the aforementioned transition. Any problem that is modeled with MDPs is considered to satisfy the *Markov property*. The latter states that the system's next state and reward depend only on the system's current state and the last action chosen by the agent [16]. The property can be described by the following equation:

$$Pr(s_{t+1} = s', r_{t+1} = r' \,|\, s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0) = Pr(s_{t+1} = s', r_{t+1} = r' \,|\, s_t, a_t) \tag{2.5}$$

where s_t is the state of the system at time t[7], a_t is the chosen action, and r' is the reward of the agent for transitioning to the next state s' by performing action a_t ($r' = R_{a_t}(s_t, s')$). Thus, recalling equation (2.4), the probability that each possible state s' is the next is formally given as follows:

[6] Arguably not the first piece of work that refers to MDPs, but certainly one of the most well-known.

[7] Obviously s_{t+1} denotes the state at time $t+1$. Note, however, that the algorithms may use the future state that occurs j steps ahead, defined as s_{t+j}. This generalization provides great flexibility, especially in episodic games [15], but is usually omitted for simplicity. In terms of this dissertation, the time factor may also be omitted for equations that are actually update rules. In such cases, the symbol of assignment ($\leftarrow$) shall be used in place of equality ($=$). Furthermore, the next possible state or action shall be primed (e.g. s' or a'), instead of subscripted in order to avoid confusion with the next definitive state.

$$P_a(s,s') = Pr(s_{t+1} = s' \,|\, s_t = s, a_t = a), \quad s \in S,\ a \in A(s) \tag{2.6}$$

and the expected reward of transitioning to state s' from the current state s_t is:

$$R_a(s,s') = E(r_{t+1} \,|\, s_t = s, a_t = a, s_{t+1} = s'), \quad s \in S,\ a \in A(s) \tag{2.7}$$

where s and a are the current state and the chosen action respectively.

Finally, concerning equation (2.6), the probability of transition is also known as the *transition function*, thus discarding the element of probability. In other words, the function could generally be definitive, defining which transition(s) are possible from state s by performing action a. This specialization is usually quite straightforward for simple MDPs, hence it is omitted.

2.4.2 The Problem

Obviously, the agent's goal is to maximize his rewards. Although simply maximizing the sum of the rewards is rational and correct, it is not optimum since identical weights are given to short-term and long-term rewards [16]. Establishing a recency property, the agent tries to maximize his *expected discounted sum*, which is computed as follows:

$$EDS_t = \sum_{j=0}^{\infty} \gamma^j \cdot r_{t+j+1}, \quad 0 \leq \gamma < 1 \tag{2.8}$$

where r_{t+j+1} is the reward of the agent for transitioning to the state s_{t+j+1} and γ is a *discount factor* which controls the effect of future rewards in current decisions. The larger the value of γ, the more important the long-term rewards.

The agent's policy is defined as a function that determines the probability of choosing an action when the system is in a particular state. Let h be the agent's policy. Then the value of a state s is given by the following *state-value function*:

$$V^h(s) = E_h\{EDS_t \,|\, s_t = s\}, \quad s \in S \tag{2.9}$$

where E_h denotes that the above value is expected based on the agent's policy h. Intuitively, the value of a state denotes how beneficial it is for the agent to be in that particular state. Another useful function is the *action-value function*, defined as the value of choosing an action a when being at a state s:

$$Q^h(s,a) = E_h\{EDS_t \,|\, s_t = s, a_t = a\}, \quad s \in S,\ a \in A(s) \tag{2.10}$$

Thus, the above function gives the value of an action concerning a particular state. Intuitively, the value of an action given a state denotes how beneficial it is for the agent to make that particular action when being in that particular state. Equations (2.9) and (2.10) are also called *value functions*.

Finally, a policy h^* is optimal if it has the maximum value functions among the ones of all other policies. Thus, the maximum value functions are defined as follows:

$$V^*(s) = \max_h \{V^h(s)\}, \quad s \in S \tag{2.11}$$

$$Q^*(s,a) = \max_h \{Q^h(s,a)\}, \quad s \in S,\ a \in A(s) \tag{2.12}$$

Combining equations (2.9), (2.10), (2.11), and (2.12) is proven [16] to give the following equation:

$$Q^*(s,a) = E_h\{r_{t+1} + \gamma V^*(s_{t+1}) \,|\, s_t = s,\ a_t = a\}, \quad s \in S,\ a \in A(s) \tag{2.13}$$

Thus, it is now obvious that finding either one of the values (2.11) and (2.12) means immediately finding the other too. Finally, the optimal policy may be defined as:

$$h^*(s) = \arg\max_{a \in A(s)} \{Q^*(s,a)\}, \quad s \in S \tag{2.14}$$

Hence, finding an optimal policy comes down to finding any of the two optimal value functions. Note, however, that finding an optimal policy without finding any value function is also adequate, since the agent is usually asked to play optimally regardless of how he accomplishes optimal play.

2.4.3 The Solution – Techniques

Upon reducing the problem to finding an optimal value function or, if possible, finding an optimal policy, various solutions are considered. This dissertation provides a taxonomy of all algorithms that effectively solve the problem, following the approach of L.P. Kaelbling et al. [15]. Typically, RL techniques can be classified according to various criteria. Some common criteria are the active or passive nature of the learner, or the degree to which the learner's perceptions are faithful to the actual environment. Although those taxonomies are interesting, they deviate from the purpose of this dissertation.

The taxonomy given in this dissertation is directly connected to the problem discussed in the previous subsection. At first, let the probability function $P_a(s,s')$ and the reward function $R_a(s,s')$ of this equation be called "model" for simplicity. Then the taxonomy is considered along the following axes:

- *Known-model techniques*
  All elements of (2.4) are considered to be known. The problem can be solved with simple Dynamic Programming techniques.

- *Unknown-model techniques*
  Only the state space and the respective actions are considered to be known. It is necessary to either approximate the model or just not use it. Thus, two variants are considered:

  – *Model-free techniques*
    Find an optimal policy without approximating the model.

  – *Model-based techniques*
    Approximate the model and use it to find an optimal policy.

Most RL techniques can be classified according to this taxonomy. An indicative classification is shown in Figure 2.5.

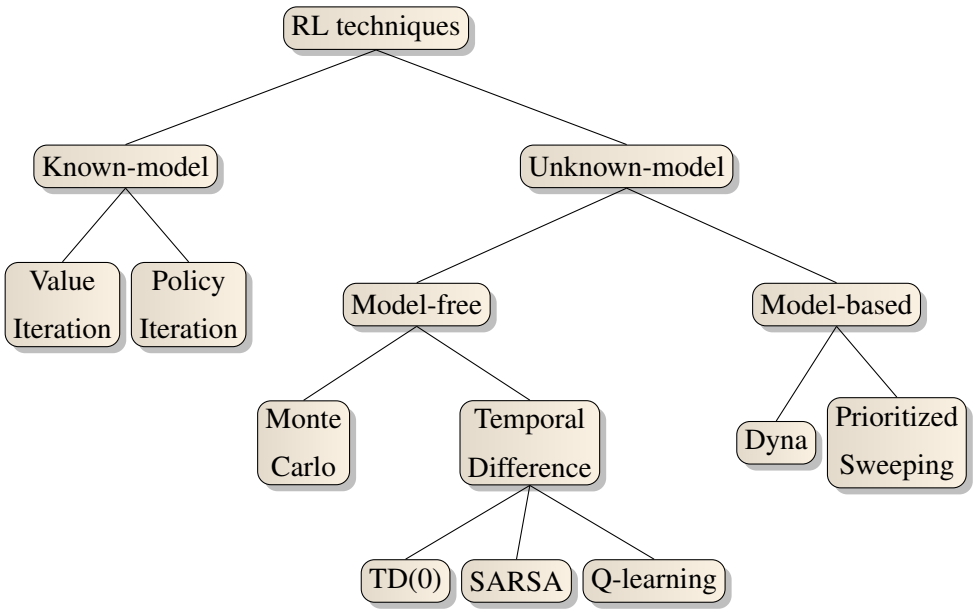


Figure 2.5: Taxonomy of RL techniques according to whether the model is known or unknown. If the model is unknown, the techniques are categorized according to whether they devise a policy by finding the model (model-based) or not (model-free).

The techniques shown in Figure 2.5 are briefly discussed in the following subsections.

2.4.3.1 Known-model Techniques

At first, one could observe that the problem gets relatively simpler if all elements of (2.4) are considered to be known. In this case, two Dynamic Programming algorithms are considered: *Value Iteration* [18] and *Policy Iteration* [17]. The former finds the optimal state-value function (see (2.11)) upon iterating through all possible action-value functions using known properties, whereas the latter finds the optimal policy using methods for solving linear equations on the action-value function (see (2.12)).

2.4.3.2 Unknown-model Techniques

Although known-model techniques are effective, they are usually inapplicable since in most RL problems the probability function P and the reward function R are unknown[8] (see (2.4)) [15]. Thus, the main goal of the RL techniques of this subsection is finding an optimal policy without knowledge of the aforementioned functions. Model-free techniques try to find an optimal policy without learning the model, whereas model-based techniques approximate the model in order to use it to find an optimal policy.

Model-free Techniques

The techniques of this category are categorized to those using *Monte Carlo* methods and those using *Temporal Difference* methods.

An algorithm using Monte Carlo was first introduced by A. Barto and M. Duff [19] as a model-free approach that attempts to improve the Policy Iteration technique in order for the latter to discard the need for a model. Typically, the Policy Iteration variant executes a loop: it updates the optimal policy using the action-value function and then it updates the action-value function using the optimal policy. This procedure is quite effective for games containing random variables since the rewards for transitioning from the current state to a new state are averaged using Monte Carlo sampling. Monte Carlo sampling ensures that the random samples effectively cover the simulation. Thus, the action-value function is constantly improving and so does the policy.

Although Monte Carlo methods are useful for solving several kinds of problems, they have certain disadvantages that render them inefficient in most cases. Specifically, they work only in episodic problems and if the number of episodes is large, computational resources may be wasted easily on suboptimal solutions. To confront these

[8] Consider also the special case where the functions are known but the state space is so large that processing all states is ineffective.

problems, R. S. Sutton [20] introduced Temporal Difference methods that combine Monte Carlo techniques with Dynamic Programming algorithms.

Temporal Difference methods can be said to extend the Monte Carlo ones since they actually use the same Policy Iteration variant, updating the action-value function and then using it to update the optimal policy. The main difference between the two classes of algorithms is that Temporal Difference techniques do not rely on updating the action-value function by using the optimal policy. Instead, the update is done recursively, thus a Dynamic Programming update function known as *update rule* is used to approximate the current values of the function based on the former ones.

A primal approach on the field by R. S. Sutton [20] is the *TD(0)* algorithm. In contrast with other algorithms, the update rule of TD(0) is applied on the state-value function. This is rational since, as noted in 2.4.2, the value functions are actually connected. Thus, the update rule for each state s is:

$$V(s) \leftarrow V(s) + \alpha \cdot \Big[r + \gamma \cdot V(s') - V(s) \Big], \quad s \in S \tag{2.15}$$

where s is the current state, r is the reward of the agent for transitioning to the next state s', γ is once again the discount factor and is used similarly to equation (2.8), and the *learning rate* α defines how quickly the algorithm converges to its target.

Another well-known algorithm of the field is *SARSA*, created by G. A. Rummery and M. Niranjan [21]. The update rule of SARSA is given by the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \Big[r + \gamma \cdot Q(s',a') - Q(s,a) \Big], \quad s \in S,\ a \in A(s) \tag{2.16}$$

where s is the current state, r is the reward of the agent for performing action a to transition to the next state s', a' are the new possible actions. γ and α are the discount factor and the learning rate and are used similarly to equation (2.15).

Finally, probably the most well-known algorithm of RL is *Q-learning*. Q-learning, an algorithm created by C. J. Watkins and P. Dayan [22], relies on finding the optimum action values by using the maximum following action value. Therefore, its update rule is said to be *off-policy*. The algorithm's update rule is shown below for each state s:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \Big[r + \gamma \cdot \max_{a' \in A(s')} \big\{ Q(s',a') \big\} - Q(s,a) \Big], \quad s \in S,\ a \in A(s) \tag{2.17}$$

where the notation symbols are identical to the ones of equation (2.16). The algorithm being off-policy is crucial since no knowledge of the action values is needed, as opposed to the former *on-policy* techniques. Thus, the optimal action-value function can be estimated even if a non-optimum policy is followed.

Finally, the optimum policy for the aforementioned Temporal Difference techniques is computed using the following equation:

$$h(s) \leftarrow \arg\max_{a \in A(s)} \{Q(s,a)\}, \quad s \in S \tag{2.18}$$

The careful reader may have noticed the similarity of the above equation to equation (2.14). This assures that the problem is correctly approximated, as long as the Q-values are computed optimally.

Model-based Techniques

One could say that the techniques of this category try to construct a model. A model is estimated and the model-based techniques use this estimation to find the optimal policy. The main class of algorithms in this area derives from *Dyna*, an architecture created by R. S. Sutton [23]. At first, Dyna observes the current state, chooses a policy and performs an action. Upon receiving a random state and action Dyna uses a Temporal Difference method (e.g. Q-learning) and updates the model (see (2.6) and (2.7)) using the statistics obtained for states and actions by the Temporal Difference Method.

Dyna is effective for a variety of problems and in most cases converges to optimal strategies. However, the algorithm keeps on randomly selecting random states and actions even upon finding an optimal strategy. Moreover, in cases where the policy reached is not optimal, the algorithm's randomness may result in an endless loop among ineffective policies. A technique that improves on these shortcomings is *Prioritized Sweeping*, created by A. W. Moore and C. G. Atkeson [24]. According to the authors, each state is assigned a priority (hence the algorithm's name) that determines whether it should be updated or be left out as ineffective. Thus, the algorithms modifies the priority of each state as well as the state's predecessors.

2.4.4 Choosing the Appropriate Method

When trying to find solutions in complex RL problems, there is no method that outperforms the others in all cases. In fact, it all comes down to choosing the appropriate method that solves optimally the problem posed. Thus, when the model is known and its computational burden is bearable, known-model techniques seem to be the optimal solution to the problem. If, however, the model is not known, then an unknown-model technique has to be selected.

Concerning the choice between model-free and model-based techniques, L.P. Kaelbling et al. [15] prove an interesting point. According to the authors, although model-based approaches seem to require fewer steps in order to converge to an optimal policy, they also require much more computational effort for each step. Thus, an a priori estimation of the state space should be the hint for selecting an approach. However, there are cases where even this estimation is impossible. In these cases, there is usually a tendency to use model-free approaches as it provides some "safety" knowing that the computational complexity is not an issue.

Finally, an algorithm's popularity may depend on several reasons. As L.P. Kaelbling et al. [15] denote, Q-learning is the most popular RL algorithm since it effectively converges to an optimal solution, even if the agent behaves randomly. That, along with the algorithm's simplicity and low computational complexity, are the main reasons why Q-learning is popular not only for solving single-agent problems but also for extending its functionality to the multi-agent case.

Chapter 3

Existing Work on Ad hoc Team Formation

3.1 Defining the problem

The problem in study was posed by Stone et al. [3] as the design of an agent capable of efficiently collaborating with unknown teammates without any prior coordination. The ad hoc notion of cooperating without prior coordination is a challenge posed recently, thus literature is rather limited on the subject. However, the problem is surprisingly broad, resulting in diverse research that handles different aspects of it. The basic most faithful line of research is the one solving the problem of creating an autonomous agent that is able to collaborate successfully with unknown teammates [3].

However, either due to the problem's difficulty or due to it having also other interesting aspects, several other lines of research have emerged.

3.2 Lines of Research

Since the problem may be interpreted in various ways, an indicative categorization of its aspects is attempted in this dissertation. In particular, based upon various criteria, one can construct different subproblems. These criteria are:

- *Partial knowledge of teammates' policies*
  The ad hoc agent has a perception of the strategies that his teammates play. The problem is reduced to determining the strategy that is selected.

- *Role of the ad hoc agent*
  The ad hoc agent may have to select among a set of fixed strategies to play or devise an efficient strategy.

- *Adaptiveness of teammates' strategies*
  The teammates' strategies may be either fixed or adaptive. In addition, a fixed strategy may also be deterministic or probabilistic.

- *Assistance given by teammates*
  The teammates may help the ad hoc agent adapt to the game by playing so that he can learn more easily.

The following subsections cover the work on the various lines of research that correspond to the above criteria.

3.2.1 Policy Selection

Probably the most reasonable compromise one could make for the teammate strategies is that they are derived from a set of known policies. In other words, the ad hoc agent has a probability distribution over the possible policies of the other agents. Although this line of research seems to simplify the problem, it is still fairly difficult, since it is possible that the policies are non-deterministic.

As S. Barrett et al. point out [25], the problem then is reduced to defining which policy the ad hoc agent should play. At first, this problem is successfully confronted using the Naïve Bayes classifier (see subsection 2.3.4). Furthermore, the authors produce interesting insight on the subject by further extending it to make the ad hoc agent construct (rather than merely select) his own policy using Value Iteration (see subsection 2.4.3.1).

A similar approach on the subject is the one followed by K. Genter et al. [26]. The authors define the separate tasks as "roles" and devise a role selection algorithm. They also study the case where role mapping is limited, i.e. a minimum number of agents must select to play the same role in order to acquire maximum rewards. In addition, upon selecting a role, role parameter fitting is a new problem posed when limited data is available to the ad hoc agent.

3.2.2 Unknown Teammate Model

Although the compromises mentioned in the previous subsection provide with interesting insights, they tend to reduce the realism of the problem. A much more difficult, yet also more realistic problem is the collaboration with totally unknown teammates. As far as current literature is concerned, there are two different ways of solving the aforementioned problem[1].

The problem is interpreted by S. Barrett et al. [25] as a task of modeling the unknown teammates. Hence, such a task may be accomplished by observing other teammates' actions and constructing models that describe them. The model construction process can be seen as a classification problem (see subsection 2.3.1) as long as the attributes are clearly defined. Although having to observe teammates' actions seems like a compromise, S. Barrett et al. [25] reduce its significance by minimizing the number of observed runs.

Another interesting approach is the one followed by F. Wu et al. [27]. The authors regard finding teammates' models redundant and construct an algorithm to plan the next actions online, by constructing and solving a series of stage games. They use Monte Carlo tree search (see subsection 2.4.3.2) to simulate action selection and essentially find a satisfactory solution[2].

3.2.3 Adaptive Teammates – The Multi-Agent Learning Aspect of the Problem

This line of work essentially models the ad hoc problem as a team task where all agents are learners. As S. V. Albrecht and S. Ramamoorthy point out [28], when multiple learning agents try to adapt their behaviour then the problem may be confronted using *Multi-Agent Learning (MAL)* algorithms.

MAL is the scientific field which concerns MAS that consist of learning agents. The MDP tuple can be extended to the multi-agent case introducing a set of agents I.

[1] Generally, teammate (or opponent) modeling is an interesting task that concerns multiple areas of MAS. However, in terms of this dissertation, the area of interest is reduced to the ad hoc team setting, and the problem is solved under the context of modeling other agents to either imitate them or devise efficient counter-strategies.

[2] One could not help but notice the similarities of the argument between this and the previous paragraph approach with the RL argument posed in subsection 2.4.3: constructing the model in order to devise the strategy or constructing the strategy at once. However, the term "model" here refers to the agent's teammates, whereas in subsection 2.4.3 it refers to the transition and reward functions of the MDP.

Thus, in respect to (2.4), a *stochastic* or *markov game* (*SG*) is defined as a tuple [17]:

$$\{I, S, A_i(s), P_{a_i}(s,s'), R_{a_i}(s,s')\}, \quad \forall s \in S,\ a \in A,\ i \in I \tag{3.1}$$

where S is a finite set of states, $A_i(s)$ is a finite set of agent i's possible actions when being at state s, $P_{a_i}(s,s')$ is the probability of transitioning to state s' when being at state s and performing action a_i, and $R_{a_i}(s,s')$ is agent i's reward for this transition. Furthermore, each agent's expected discounted sum is defined in respect to (2.8).

Much of the work on the field relies on extending the Q-learning algorithm to take into account the teammates' actions. Extending (2.17) to the multi-agent case is rather straightforward. Considering a vector of all possible players' actions:

$$\vec{a} = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}, \quad a_i \in A_I(s),\ A_I(s) = A_1(s) \cup A_2(s) \cup \cdots \cup A_n(s) \tag{3.2}$$

the new update function for each player $i \in I$ is:

$$Q_i(s,\vec{a}) \leftarrow Q_i(s,\vec{a}) + \alpha_i \cdot \left[r_i + \gamma_i \cdot \max_{\vec{a}'} \left\{ Q_i(s',\vec{a}') \right\} - Q_i(s,\vec{a}) \right], \quad s \in S,\ \vec{a} \in A_I(s) \tag{3.3}$$

where s is the current state, r_i is the reward of agent i for performing action a_i and transitioning to the next state s', $\vec{a}'$ are the new possible actions. In addition, γ is the discount factor which controls the effect of future rewards and α is the learning rate which defines how quickly the algorithm converges to its target.

Distinguishing the various algorithms initially comes down to defining the optimal policy function, whether it is defined as an extension of (2.18) or not. Furthermore, being aware or not of the other agents' actions and rewards results in two lines of research. When full knowledge of these parameters is assumed, the agents can use techniques such as Team Q-learning [29] or Distributed Q-learning [30]. However, if each agent is not aware of his teammates' actions and rewards, he also has to learn them. This line of research contains learning the teammates' actions while playing and using this beliefs in update rules similar to that of equation (3.3). Some of the most widely known techniques are Joint Action Learning [31] and Nash Q-learning [32, 33].

Considering the aforementioned algorithms are only indicative, MAL literature is remarkably broad. However, as noted by S. V. Albrecht and S. Ramamoorthy [28], these algorithms are designed for homogeneous groups of agents. Thus, the authors evaluate the performance of such algorithms when facing the ad hoc team formation problem. Despite deviating from the previous paradigms, the authors point out an interesting view on the topic.

3.2.4 Teacher – Learner

Another slightly different line of research is the one described in this subsection. Consider a situation where the agents try to help their ad hoc teammate select the optimal moves. Since the agents actually act as teachers while the ad hoc agent is a learner, this line of research is known as the *teacher–learner* model. Currently research on this field is limited.

An interesting piece of research is the one by S. Barrett and P. Stone [34]. The authors provide a theoretical approach for the cooperative multi-armed bandit problem. The problem contains two agents, a teacher and an ad hoc learner, who have to cooperate in order to receive the maximum reward. In particular, consider having three arms such that the teacher can pull all three of them, whereas the learner has to choose between arms 2 and 3. In addition, let the maximum reward be given for pulling arms 1 and 2 (each by one of the two agents).

The learner does not have any prior information about the distribution of rewards according to arm pulling. By contrast, the teacher has full information, yet he is incapable of teaching without deviating from his optimal move. In other word, the teacher may either select the optimal move (i.e. pull arm 1) or teach the learner by pulling another arm, so that the latter may observe.

3.2.5 Relation of this Work to Existing Work

Upon defining the various lines of research on the ad hoc problem, the contribution of this dissertation is assorted to the appropriate areas and compared with similar approaches on these areas. Generally, the problem confronted by this dissertation is the design of an agent that shall be able to perform efficiently and effectively in an ad hoc (previously unknown) situation.

Firstly, the role of the ad hoc agent is to construct models of his teammates. The approach on this topic is similar to the one followed by S. Barrett et al. [25]. Both implementations actually identify the need for a classification model to simulate the actual policy of the teammates. However, the authors construct a model for the pursuit domain, whereas the solution given here is more generic; a high-level framework is constructed. The domain specific components are identified and a discussion is made as to how they should be constructed.

Secondly, upon constructing models for his teammates, the agent should be able to determine which model is played by every agent. The approach followed by this

dissertation resembles once again the work of S. Barrett et al. [25]. Both implementations use the Bayes theorem (see subsection 2.3.4) in order to select among the various models. The Naïve Bayes classifier is sufficient as far as the policy selection task is concerned.

Thirdly, the problem of constructing an effective and efficient answer strategy for all combinations of teammate policies is put under consideration. As in [25], the problem is confronted using RL techniques. However, the task selected in this dissertation is more fine-grained since it encompasses multiple goals and multiple teammates. Having multiple goals means that pruning techniques such as Monte Carlo Tree Search (see subsection 2.4.3.2) that is used by S. Barrett et al. [25] are not applicable since local optima have to be avoided. Hence, the complexity has to be handled by reducing the number of policies encountered, i.e. having a single-agent Q-learner as a response for each policy.

Furthermore, since the agent was designed considering arbitrary number of teammates (in contrast to exactly four agents as in [25]), using an RL technique for any combination of policies introduces computational complexity issues. Thus, these issues are confronted using a single RL technique per possible policy and constructing a merger for any combination of policies. In addition, the amount of learning required concerning the main algorithms used in this project is put under consideration.

Finally, the evaluation testbed is an interesting example not only for the applicability of ad hoc techniques but also for the way these techniques can be evaluated. Thus, the problem of adaptive teammates is also partially confronted, since the experiments include testing with fixed, probabilistic, as well as other adaptive ad hoc learning agents. Concerning a scenario where multiple ad hoc learning agents form an effective team, the approach of this project bears interesting similarities with the work of S. V. Albrecht and S. Ramamoorthy [28]. Both lines of work focus on evaluating the performance of a team consisting of learning agents in an ad hoc team setting. However, the approach followed on this dissertation is directed mainly towards the creation of a single ad hoc agent able to address situations of heterogeneous teams. By contrast, the authors demonstrate the effectiveness of various algorithms that were designed with homogeneous teams in mind to heterogeneous situations.

Chapter 4

A Novel Approach to the Ad hoc Problem

4.1 Analyzing the problem

In accordance with Section 3, the main contributions of this dissertation refer to the main problems posed in current literature. Thus, concerning cooperative ad hoc scenarios, the three main challenges addressed are:

1. *Policy selection*
   The ad hoc agent has to play and at the same time observe his teammates and determine which policy each agent follows.

2. *Teammate modeling*
   The ad hoc agent constructs possible policies that his teammates may follow upon observing their actions.

3. *Strategy construction*
   The ad hoc agent has to devise an efficient strategy that conforms with the strategies played by his teammates.

It is important to note that the above challenges are not separable. Any ad hoc agent has to accomplish most or even all of these tasks in a satisfactory level. The following subsections define the problems outlined here more formally and suggest possible solutions.

The terminology and notation used in the following subsections are simple. Any agent is considered to play a *strategy*. When this strategy is not fully known (i.e. when

others refer to a model of it), then it is called a *policy* or a *model*. Striving to make our findings as generic as possible, the scenario contains I agents that make a move for each *timeslot* t. The agents observe the *state* s of the system and choose to play an *action* out of a predefined set of actions. Thus, the conditions defined by Stone et al. [3] are met; the ad hoc agent is only able to observe his teammates, while they make their moves. No communication with the other agents is possible.

4.2 Policy Selection

Concerning the policy selection task, the ad hoc agent is given a set of all possible policies that one of his teammates may play. Thus, the problem is actually reduced to estimating which policy is played by his teammate(s) at any time. Since no prior observation is possible, the agent has to constantly update his estimation of any teammate's model by observing his actions. Hence, the ad hoc agent wants to find the probability of a teammate following a model given his actions.

The approach followed here is similar to the Naïve Bayes classifier (see subsection 2.3.4). Firstly, the set of possible policies-models is defined as $M = \{M_1, M_2, \dots\}$. Each model is defined such that it receives the state of the system and returns a distribution over its possible actions. Consider a set of all possible actions:

$$A = \{A_1, A_2, \dots, A_n\} \tag{4.1}$$

The actual action A^t played by a teammate agent at timestep t is defined formally as

$$A^t = A_k, \quad k \in \{1, 2, \dots, n\},\ t \in \{1, 2, \dots, T\} \tag{4.2}$$

Then, using Bayes theorem, one could find the probability of selecting a model i using the following equation:

$$P(M_i|A^t) = \frac{P(A^t|M_i) \cdot P(M_i)}{\sum_j \Big\{ P(A^t|M_j) \cdot P(M_j) \Big\}} \tag{4.3}$$

where $P(M_i)$ is the prior distribution over the set of possible models, i.e. the initial given probability of selecting model i. $P(A^t|M_i)$ is the probability of an agent playing action A^t given that he follows model M_i, and can be formally introduced as:

$$\begin{aligned} P(A^t|M_i) &= Pr(action = A^t|model = M_i) \\ &= P(A_k|M_i), \quad \text{if } A_k = A^t \end{aligned} \tag{4.4}$$

where $P(A_k|M_i)$ is considered known since the model M_i is known. Considering an observation $Actions = \{A^1, A^2, \ldots, A^T\}$ for a series of timeslots, equation (4.3) becomes:

$$P(M_i|A^1, A^2, \ldots, A^T) = \frac{P(A^1, A^2, \ldots, A^T|M_i) \cdot P(M_i)}{\sum_j \left\{ P(A^1, A^2, \ldots, A^T|M_j) \cdot P(M_j) \right\}} \tag{4.5}$$

Making the "naïve" assumption that the probabilities of playing any actions are independent:

$$P(A^t, A^{t'}|M_i) = P(A^t|M_i) \cdot P(A^{t'}|M_i), \quad \forall t, t' \in 1, 2, \ldots, T,\ t \neq t' \tag{4.6}$$

then equation (4.5) gives:

$$P(M_i|A^1, A^2, \ldots, A^T) = \frac{\prod_t \left\{ P(A^t|M_i) \right\} \cdot P(M_i)}{\sum_j \left\{ \prod_t \left\{ P(A^t|M_j) \right\} \cdot P(M_j) \right\}} \tag{4.7}$$

Note that zero probabilities are handled by replacing them with very small numbers and re-normalizing. Equation (4.7) is actually a form of the Naïve Bayes classifier. Its similarity to (2.3) is quite obvious. A simpler representation may be derived by equation (4.5) as:

$$P(M_i|Actions) = \frac{P(Actions|M_i) \cdot P(M_i)}{\sum_j \left\{ P(Actions|M_j) \cdot P(M_j) \right\}} \tag{4.8}$$

considering that $P(Actions|M_i)$ is updated upon each new observation of the teammates' actions. An example that visualizes the procedure of selecting among models can be seen in Figure 4.1.

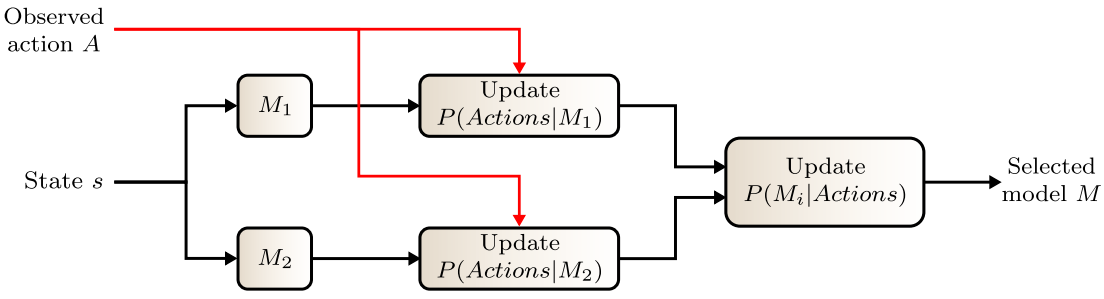


Figure 4.1: An example policy selection task with 2 models. For any new observed action, the probability of each model is updated using the probability that the action is selected given the model. Thus, a distribution is formed and the selected model is drawn from this distribution.

According to Figure 4.1, for any new observed action, the agent firstly updates the probability of the actions given the model ($P(Actions|M_i)$) for each model (M_i), and then he creates the probability of each model given the actions ($P(M_i|Actions)$).

The model described above was selected for various reasons. Firstly, it is an incremental classification function, meaning that the classifier's model can be easily updated for any new observation of agents' actions, while it is also available for use at any time. In addition, the classifier can give fast results, such that computing time is insignificant at any timeslot. Concerning its effectiveness, it is sufficient since the "naïve" conditional independence assumption is quite reasonable. As stated above, the agents act on an observe-and-then-act basis, meaning that they use elements of the environment (state) and not any significant pieces of history to determine their next action. Thus, any dependencies among the probabilities of actions are generally minimized.

Finally, there are two possible alternatives to choosing which model best fits a teammate. A viable option would be to select the model with the highest probability. However, this selection is rather over-simplified; if the probabilities among the models are similar, then it is possible that the agent gets biased towards a particular model only for it having slightly higher probability. A better approach would be to view the model probabilities as a discrete probability distribution over the models. Then, it is possible to draw the model from the distribution by drawing a floating-point number uniformly from the range $[0,1)$, and determine which model to use depending on the number's position in the distribution. For example, let M_1 and M_2 have probabilities 0.65 and 0.35 respectively, if the random number was in the range $[0,0.65)$, then M_1 would be selected, and if it was in the range $[0.65,1)$, then M_2 would be selected.

4.3 Teammate Modeling

In the previous subsection, it was shown how the ad hoc agent confronts the problem of selecting the model that best fits a teammate agent. The models themselves can be either hardcoded "black boxes" that represent agent policies, or they may be totally unknown. In this subsection, the teammates' models are considered unknown, thus the problem of constructing the model of a teammate agent is faced[1]. The problem has strict assumptions; the ad hoc agent knows nothing about the teammate's goals or style of play.

[1]Naturally, the technique of this section can be easily extended to cover constructing the models of all teammates. Thus, the subsection refers to modeling one teammate for simplicity.

Consequently, it is not possible for the agent to construct the teammate's model on-the-fly, thus it is necessary that the agent observes his teammate for a number of rounds[2]. However, the number of rounds may be quite small since the agent's desideratum is not to design a perfect strategy; instead the agent has to create policies that correctly recognize the main goals and can be clearly distinguished with a system similar to the one in subsection 4.2.

The overall problem is solved by decomposing it to various subproblems. As mentioned in subsection 4.1, any agent observes a state and performs an action. The term "model", thus, refers to constructing a mapping from any possible state to an action. Hence, the problem is decomposed to the following subproblems:

- Representing correctly the state and the action.
- Creating a model that successfully maps a state to an action.

A visualization of the model's creation is shown in Figure 4.2.

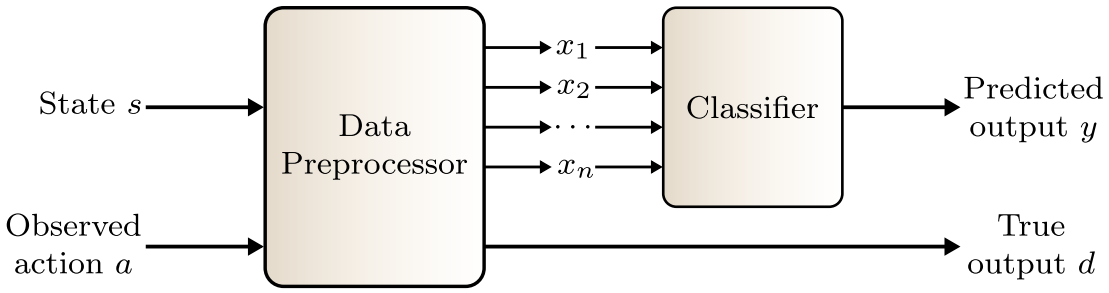


Figure 4.2: The training phase of a teammate modeling task. The data preprocessor receives an observed state-action pair and constructs a data instance $\{x_1, x_2, \ldots, x_n, d\}$, where d is the value of the class attribute.

The problem is modeled as a classification problem (see subsection 2.3.1). Each instance corresponds to a state of the environment as well as a possible action of the teammate agent. Note that it is necessary that the instance is constructed using full state and action information since the output of the classifier is given relatively to the state given. The attributes are constructed from the model using a data preprocessor as shown in Figure 4.2. Constructing the attributes is not trivial; one has to exhaustively describe the environment while avoiding any variables that are not generic enough to model different runs of the same game.

[2]Initially, model construction is indeed impossible. The ad hoc agent has to be given some data. However, note that it would be interesting to attempt an update of an existing model on-the-fly, i.e. while playing.

For example, consider a simple three-armed bandit game, where 3 randomly placed arms are labeled 1, 2, and 3. The agent has to pull the arm labeled 1 in order to win. The agent continuously plays the game with a fixed strategy: pull the arm labeled 1[3]. The definition of the problem should actually describe the attributes of the model. There are two ways to model the attributes: either model each arm pulling with respect to its relative position from the other arms (e.g. left, center, right) or model each arm pulling with respect to its label (e.g. 1, 2, 3). The key to selecting between the representations is to find which one of them exhaustively describes the game, yet is generic enough to be valid for different game situations. Hence, the first model is not appropriate to model the game, since it may be possible that an observed game instance had the three arms placed in ascending order from left to right (i.e. 1, 2, 3), whereas the next game instance had them placed in descending order (i.e. 3, 2, 1). Thus, had the model been determined by the first game instance, it would have modeled the optimal response "left", thus it would have failed to properly model the second game instance, where the optimal response is "right". On the contrary, if the arm pulling attribute is determined by its label, the model shall correctly identify the agent's strategy: pull arm labeled 1.

Considering the above example, it is obvious that the attributes contain also the class attribute. in other words, the preprocessor also has to model the teammate agent's action, in a way compliant with the classifier's exit, as seen in Figure 4.2. During the training phase, the classifier can be trained by comparing the true output d with the predicted output y.

Upon training the model, the usage of the classifier requires a postprocessor, as shown in Figure 4.3.

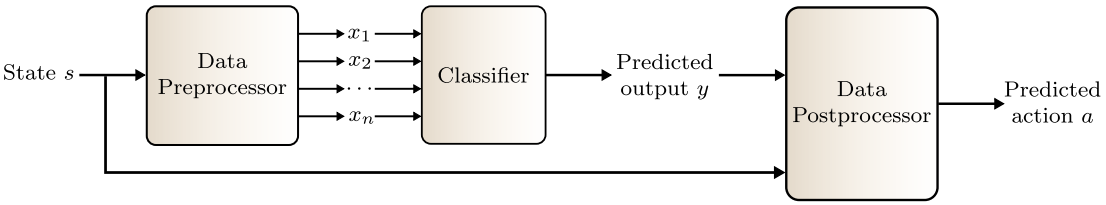


Figure 4.3: The usage phase of a teammate modeling task. The data preprocessor receives the state and constructs a data instance $\{x_1, x_2, \ldots, x_n\}$. The value of the class attribute is reconstructed to a valid action by the data postprocessor.

The output y of the classifier is actually a value relative to the current environment. Thus, following the three-armed bandit example given above, consider drawing the

[3]The game is indeed as simple as it gets, however the cause is demonstrated. For a more extensive example see Section 5.

output of the classifier from the set $\{1,2,3\}$. However, the agent's actions may be modeled as $\{left, center, right\}$, i.e. the system could only receive actions in the aforementioned form. Hence, y has to be interpreted with respect to the current state of the environment. Note that since a classifier typically outputs a distribution over the possible values of y, the mapping is actually performed between this distribution and the distribution over the possible actions a. For example, the distribution given by the model could be:

$$Distribution\{1,2,3\} = \{1 : 0.65, 2 : 0.20, 3 : 0.15\} \tag{4.9}$$

If the arms are labeled from left to right with labels 3,1,2, then a representation of the system's state could be:

$$s = \{arm_{position=left,label=3}, arm_{position=center,label=1}, arm_{position=right,label=2}\} \tag{4.10}$$

The mapping from the distribution of the set $\{1,2,3\}$ (see equation (4.9)) to that of the set $\{left, center, right\}$ is formally defined as:

$$P(a) = P(y), \quad \forall y \in \{1,2,3\},\ a \in \{left, center, right\} : a = posOfArmLabeled(y,s) \tag{4.11}$$

Function $posOfArmLabeled$ (short for "position of the arm of which the label is") is actually the function implementing the mapping between the two representations. It receives the system state s, and an output y. The former contains the ordering of the arms and the latter contains the distribution of labels of the arms. Thus, with respect to equations (4.9)–(4.11), the probability of selecting the arm labeled 1 is assigned to the probability of selecting the arm positioned in the center, i.e. $P(center) = P(1)$. Similarly $P(left) = P(3)$ and $P(right) = P(2)$. Thus, the distribution of equation (4.9) is transformed to that of equation (4.12):

$$Distribution\{left, center, right\} = \{left : 0.15, center : 0.65, right : 0.20\} \tag{4.12}$$

Finally, the task of selecting an appropriate classifier usually depends on the problem as well as the desiderata. Generally, as mentioned previously in this subsection, there are few observation timeslots, thus making the program appropriate for the use of "heavy" classifiers, such as C4.5 or SMO (see subsection 2.3.6). Since there are many attributes, techniques such as Naïve Bayes do not fit well the causes of the problem. In addition, an incremental algorithm would have no clear advantage since the scenario is split to processing and execution phases (see subsection 4.5).

4.4 Strategy Construction

According to the previous subsections, the ad hoc agent is able to create policies that his teammates may follow, and choose which policy is actually followed among those. What is not yet analyzed is how is the agent going to play in order to coordinate well with these models. It is obvious that no single optimal strategy can be created since every teammate's strategy may be classified as a different observed policy/model at different timeslots. The problem, thus, is defined as the construction of a strategy capable of interacting well with a set of observed policies.

The strategy is constructed as a set of policies, such that each of them is suitable for a combination of the other agents' observed policies. At first, concerning a single teammate, the ad hoc agent would have to create an *answer policy* for each of the teammate's policies. Thus, for any model M_i, the ad hoc agent creates an *answer model* $\mathcal{A}(M_i)$, where for the purposes of this subsection the operator $\mathcal{A}(\cdot)$ receives one or more models as arguments and returns an answer model. As opposed to subsection 4.3, the ad hoc agent has to create a specific strategy for a particular game instance as well as a particular teammate policy. The agent uses the teammate's policy to determine whether the goals of the game are assigned to his teammate, or himself, or even both.

Upon determining his goal, the ad hoc agent has to create a policy in order to accomplish it. Thus, before actually playing, the agent simulates the game and learns his policy using an RL method. The problem is modeled as an MDP; the environment is modeled as a reward space consisting of state-action pairs such that transitioning to the goal state of the agent has a very high reward whereas all other transitions have lower rewards. The rewards given to other agents' goal states are not significant[4]. Similarly to subsection 4.1, the state space is defined as the possible states of the environment with the ad hoc agent[5] in it and the action space contains the agent's possible actions.

The ad hoc agent applies the Q-learning algorithm, updating the Q-values according to equation (2.17). Although any RL algorithm could solve the problem, Q-learning seems to be a good fit. As mentioned in subsection 2.4.4, model-free techniques are generally considered safer choices when the state space is not known a priori. In addition, Q-learning is generally effective when most rewards are similar and only few of them deviate significantly. Finally, note that since the ad hoc agent has limited time to create his policy, it is possible that the algorithm does not run until con-

[4]In practice, the other agents' goal states could be given negative rewards, however if the reward of the ad hoc agent's goal is sufficiently large, then he shall not deviate. In any case, no high positive reward should be given since the ad hoc agent may confuse his goal.

[5]Including the teammate agents is redundant since the ad hoc agent has defined his goal independently. In addition, including other agents results in a very large increase of the problem's complexity.

vergence. A sub-optimal solution is in most cases acceptable as long as the strategy's goals are accomplished.

Thus, having m possible models for 1 teammate, the ad hoc agent constructs m strategies that "answer" the possible models. However, when having n teammates, the agent has to create as many strategies as the possible combinations of the teammates' models. Although generating all possible strategies is feasible, it is not scalable enough since the agent has to train m^n Q-learning algorithms. For example, consider having 4 teammates whose strategy is chosen out of 5 possible models. Then, the possible strategy combinations are $5^4 = 625$. Since computing a Q-value state action array is on its own a rather "heavy" learning procedure, computing 625 Q-value state action arrays provides with a rather inefficient way to construct the model.

The ad hoc agent can resort to certain compromises, such that the problem's complexity is drastically reduced. Consider having computed an answer model $\mathcal{A}(M_i)$ for each model M_i. Thus, instead of computing a different answer model for each model combination of the teammates, a *Merge* function can be used to combine the various answer models. An example of using the Merge function is shown in Figure 4.4.

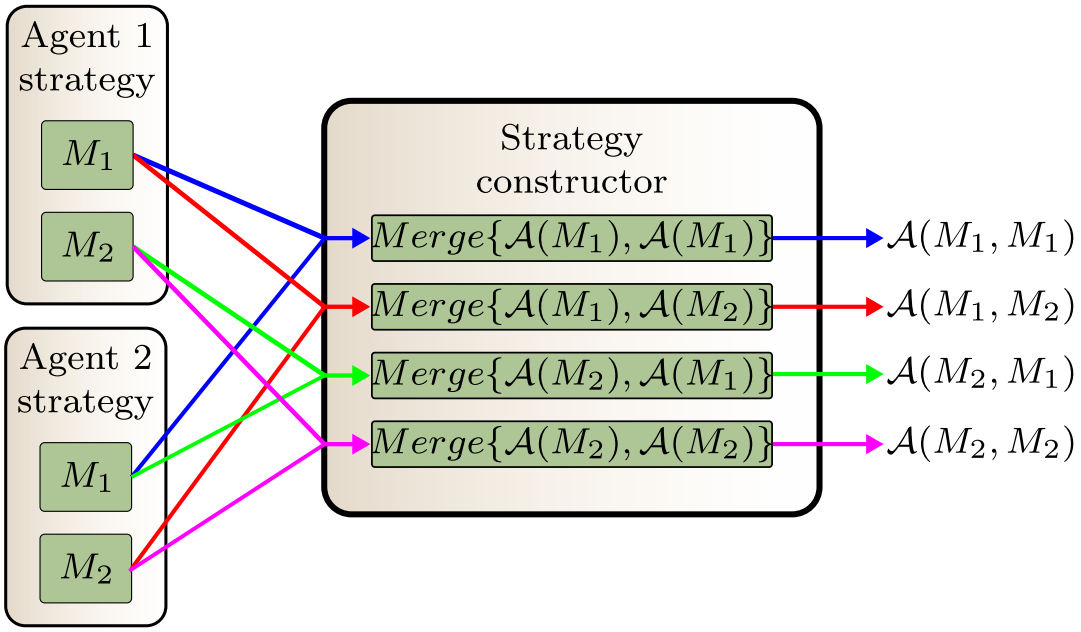


Figure 4.4: An example strategy construction task with 2 agents following 2 models (M_1,M_2). The answer strategy is constructed as a merger of the answer policies ($\mathcal{A}(M_1)$,$\mathcal{A}(M_2)$).

The *Merge* function is formally defined as follows:

$$Merge\{\mathcal{A}(M_{i_1}), \mathcal{A}(M_{i_2}), \ldots, \mathcal{A}(M_{i_n})\} = \mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n}), \quad \forall i_k \in [1, m] \tag{4.13}$$

where n is the number of agents and m is the number of policies that they may choose from. Upon formally defining the *Merge* function, the problem is now reduced to

creating such a function. Obviously, the *Merge* function operates on the Q-values of the answer models, since the Q-values actually constitute the model. A slightly more intuitive definition would be that the *Merge* function depends on the goals of each agent. However, at this phase each policy is actually a Q-value state action array that sufficiently describes not only the policy's goal but also the actions needed to accomplish it. In any case, the *Merge* function is problem specific, since the Q-values may differ depending on the goal states of the problem. Thus, different functions could be useful for different distributions of tasks among the agents.

For instance, consider the case where the goals of n agents are aligned such that half of them have to follow model M_1 and the other half model M_2, and the agent believes $n/2$ agents shall follow model M_1 (n is even). As a result, the agent has $n/2$ $\mathcal{A}(M_1)$ models and $(n/2-1)$ $\mathcal{A}(M_2)$ models. Since the game has two models, $\mathcal{A}(M_1)$ actually has the same goals as model M_2 and $\mathcal{A}(M_2)$ has the same goals as model M_1. Since a "goal" in a Q array is denoted as a large Q-value having a *Merge* function that sums all Q-arrays would make the ad hoc agent play a mixed model that has as primary goal the same as $\mathcal{A}(M_1)$ (or M_2). Thus, the agent would play a policy similar to the optimal, which is M_2. The *SumOfQvalues* function is formally defined as follows:

$$\begin{aligned}
\mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n}) &= SumOfQValues\{\mathcal{A}(M_{i_1}), \mathcal{A}(M_{i_2}), \ldots, \mathcal{A}(M_{i_n})\} \Leftrightarrow \\
Q_{\mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n})}(s,a) &= Q_{\mathcal{A}(M_{i_1})}(s,a) + Q_{\mathcal{A}(M_{i_2})}(s,a) + \cdots + Q_{\mathcal{A}(M_{i_n})}(s,a) \Leftrightarrow \\
Q_{\mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n})}(s,a) &= \sum_{k=1}^{n} Q_{\mathcal{A}(M_{i_k})}(s,a), \quad \forall s \in S,\ a \in A(s)
\end{aligned} \tag{4.14}$$

The above model may also be easily formalized using weights, given there are models that are chosen by more than one agent, as in the example given. Other specific *Merge* functions could be defined for different cases. For example, the average of the Q-values could be computed instead of the sum of them. The average operator could be useful if the absolute Q-values (and not only their relative values to one another) needed to have the same order as a simple Q-array. Another interesting operator is given in subsection 5.4.2.

4.5 Agent Design

Upon analyzing the various components that correspond to the challenges posed in subsection 4.1, the design of an ad hoc agent is demonstrated. The various design decisions are analyzed and discussed. The core algorithm of the agent is shown in Figure 4.5.

DECLARATIONS

Data Instance $DI = \{x_1, x_2, \ldots, x_n, d\}$

Unknown Data Instance $UDI = \{x_1, x_2, \ldots, x_n\}$

$DI =$DATAPREPROCESSOR(s, a) *(see Figure 4.2)*

$UDI =$DATAPREPROCESSOR(s) *(see Figure 4.3)*

$a =$DATAPOSTPROCESSOR(s, y) *(see Figure 4.3)*

$y =$USECLASSIFIER$(\{x_1, x_2, \ldots, x_n\})$ *(see Figure 4.3)*

TRAINCLASSIFIER$(DI_0, DI_1, \ldots, DI_T)$ *(see Figure 4.2)*

UPDATENAÏVEBAYES(A, M_i) *(see Figure 4.1)*

OBSERVATION PHASE

foreach (Policy M_i)
  foreach (Observed timestep t)
    Observe state s_t and action a_t
    Compute instance $DI_{M_i t} =$DATAPREPROCESSOR(s_t, a_t)

PROCESSING PHASE

foreach (Policy M_i)
  TRAINCLASSIFIER$(DI_{M_i 0}, DI_{M_i 1}, \ldots, DI_{M_i T})$
foreach (Policy M_i)
  Create answer policy $\mathcal{A}(M_i)$ using Q-learning
foreach (Possible combination of n teammates and i policies)
  Compute $\mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n}) = Merge\{\mathcal{A}(M_{i_1}), \mathcal{A}(M_{i_2}), \ldots, \mathcal{A}(M_{i_n})\}$

EXECUTION PHASE

foreach (Timestep t)
  foreach (Teammate n)
    foreach (Policy M_i)
      Observe teammate's action A^t
      UPDATENAÏVEBAYES(A^t, M_i)
    Select a model M_i according to its probability $P(M_i|Actions)$
  Select the answer model $\mathcal{A}(M_{i_1}, M_{i_2}, \ldots, M_{i_n})$
  Select an action a from the answer model

Figure 4.5: The core algorithm of an ad hoc agent. The agent creates teammate models by observing a limited number of runs, constructs effective answer policies, and determines which one to use upon determining his teammates' selected models.

The algorithm is divided into 3 phases, the observation, the processing and the execution phase. During the observation phase, the ad hoc agent observes his teammates and creates instances using the data preprocessor. During the processing phase, the agent constructs his strategy. At first, he trains a classifier for each policy, thus constructing a model M_i that may be selected by any agent. In addition, the agent constructs an answer policy $\mathcal{A}(M_i)$ for any policy M_i using Q-learning. After that, he uses a *Merge* function in order to construct an answer policy for each possible combination of his teammates' selected policies. Finally, during the execution phase, the agent observes the actions of his teammates and updates a Naïve Bayes classifier concerning the policies for each one of them. Hence, he probabilistically selects a model for each one of his teammates, thus constructing a combination of policies, one for each of the teammate agents. The ad hoc agent selects the corresponding answer policy and selects an action according to it.

Consequently, implementing the ad hoc agent's strategy comes down to defining certain functions that are mainly problem-specific. Although, these functions were discussed in the respective subsections, the design decisions are also summarized here. Hence, one has to:

1. *Create a data preprocessor and a data postprocessor*
   The preprocessor and the postprocessor are actually the functions that provide the mappings needed to use the classifier in order to model a teammate's policy. Creating them is rather simple, as long as two main conditions are met:
   - *The attributes have to sufficiently cover the entire range of the problem.*
   - *The attributes have to depend only on the relative state of the problem, i.e. any attributes that may be valid only for a specific run of the problem have to be avoided.*

2. *Use an appropriate classifier*
   The choice of the classifier should not be crucial for the effectiveness of the strategy. However, it is generally preferable that the classifier is selected with respect to the attributes. For example, when having many attributes, a rather complex classifier like C4.5 may ideal. Since the number of data instances is generally considered to be small, "heavy" classifiers such as the SMO or the multi-layer perceptron are also considered good choices.

3. *Use an appropriate Merge function*

   The *Merge* function is crucial since the merger of the Q-values not only has to combine their information but it must also be a playable strategy. The goals of the game should give a hint for the choice of a sufficient *Merge* function. Generally, attention should be given to the Q-values themselves. Thus, if the Q-values are skewed towards a specific state (or an area of states), then a function that sums the Q-values should be sufficient. However, if the absolute value of the merger is important, then the average of the Q-values seems like a more viable option. Having multiple goals may require more complex *Merge* functions. An example of multiple goals can be seen in subsection 5.4.2.

Chapter 5

The Search-And-Rescue Domain

5.1 Overview

The term "search-and-rescue" (SAR) generally refers to the procedure of searching for people that are in danger and rescuing them. The people are usually in danger due to a natural disaster (hence the similar term "disaster-rescue" as in [4]). SAR is a large domain consisting of sub-domains that mainly concern the terrain of the disaster, e.g. urban search-and-rescue for disasters including collapsed city buildings. SAR testbeds have lately gained an increasing interest due not only to their social cause but also to their interesting domain features, e.g. heterogeneity of the agent teams, real-time requirements etc. [5].

An interesting approach on the field is the RoboCup Rescue [4, 5]. The competition fields involve creating realistic simulators and/or creating competent agents or robots to perform the rescue. In terms of this dissertation, search-and-rescue is considered along a rather simpler axis, in sense that no specific constraints about the terrain are imposed. Thus, the SAR system is a standard grid world. However, without loss of generality, the SAR testbed could resemble a simple situation of a collapsed building. Humans are trapped inside the building, and robots are sent to find them and bring them out of the building.

Thus, the SAR grid world consists of the following elements:

- *Open cells*
  An open cell, which may be occupied by the agents.

- *Obstacles*
  Walls and other objects that block the way of agents.

- *Humans in danger*
  The humans that should be rescued in as little time as possible.

- *Exit location*
  A location where humans are considered rescued and safe.

The above elements comprise the system. A scenario involves agents entering the SAR terrain, going to the humans and rescuing them by bringing them out of the terrain's dangerous locations to the exit location.

5.2 The Game

Although there are several simulator implementations for the SAR domain [35, 36], none of them could suit the purposes of this project. In particular, the project requirements contain a simple grid world with two humans in danger, which is rather unusual in terms of current literature. Thus, a SAR simulator was designed and implemented from scratch. The simulator is a properly designed Software Engineering project, of which the full specification lies beyond the scope of this dissertation. Appendix A provides with selected class diagrams for the interested; the diagrams, however, are not necessary to understand the functionality of the simulator.

A screenshot of the search phase of the simulator is shown in Figure 5.1.

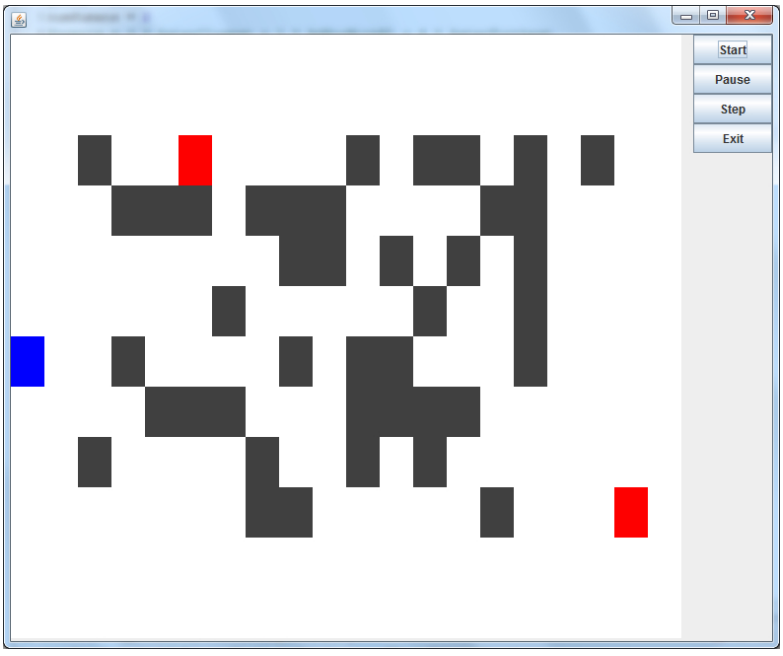


Figure 5.1: The search phase of a SAR terrain. The goal of the agents (■) is to save all the humans (■), while avoiding obstacles (■).

As seen in Figure 5.1, the agents all have the same starting point (blue cell). The goal of the agents is to save all humans (red cells). Considering n agents, $n/2$ should go to one of the humans and $n/2$ to the other. The two humans are placed such that their distance from the starting point is not equal, thus they shall be referred as *close* and *far* human. The rescue phase is quite simple. A screenshot is shown in Figure 5.2.

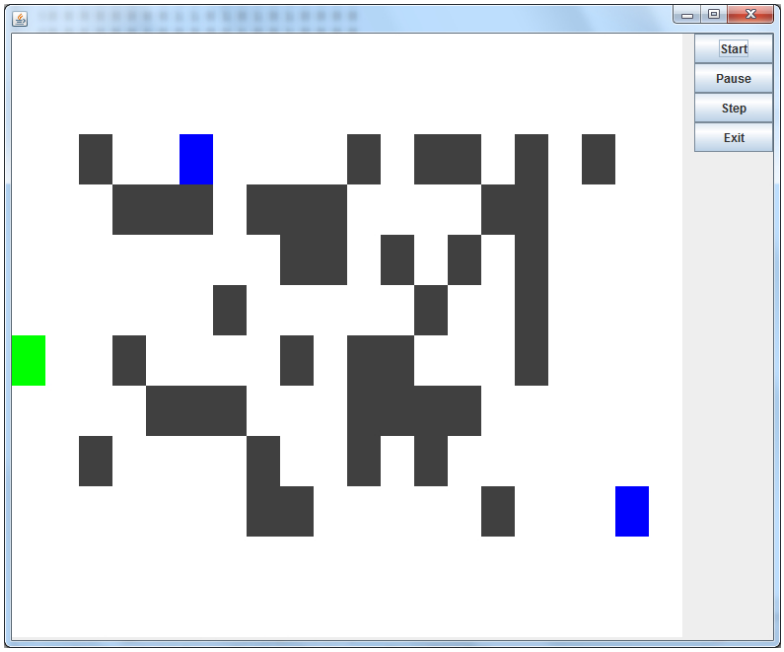


Figure 5.2: The rescue phase of a SAR terrain. The goal of the agents (■) is to return to the exit point (■), while avoiding obstacles (■).

The goal of all agents is to go to the exit point (green cell) which coincides with the entry point. The rescue phase is actually used as a complementary phase for the problem. The problem is solved similarly to that of the search phase, only simplier since this time all the agents have the same goal.

As shown in Figure 5.1, the terrain consists of distinct cells. Any cell can be unoccupied, such as the open area and the goal location cells, or unoccupied, such as the human and obstacle cells. When starting a new scenario, the agent is entering the terrain from what shall be called his *entry location*. This location is also the exit location. At any current position, the agent is able to move or try to move towards the 4 directions (`North`, `South`, `East`, `West`)[1] or decide to stay still (`Still`). Concerning a cell, it can be an *open cell*, or an *obstacle*, or it may contain a *human*.

A sample configuration file of the simulator is provided in Appendix A (see Figure A.3). The simulator provides two functionalities for each agent; the agent can

[1] Diagonal moves are not considered for simplicity.

either be an active player or an observer (needed for teammate modeling as in subsection 4.3). The simulator supports having as many agents as wanted, as long as the active players at any time are an even number due to the game specification; they have to split evenly so that exactly half of them go to each of the two agents.

Considering n agents and 2 humans, it is assumed that $n/2$ agents are required to carry each human back to the exit point. At the start of the search phase, the agents are given a full map of the environment. Thus, their aim is to find an optimal path towards the human of their choice and return back to the exit point once again by finding an optimal path.

Concerning the formal definition of the simulator, for each agent a state s is defined as the terrain map as well as his position in it. Thus, the state space is defined as a set:

$$S = \{s|s = (Map, pos)\} \tag{5.1}$$

where pos is the agent's position in the map[2], and is formally defined as:

$$pos = (x_A, y_A) \tag{5.2}$$

where x_A and y_A are the values of the positions concerning the vertical and horizontal axes. In terms of this dissertation, the notation (x, y) is used to represent a particular position on the grid. According to the above discussion, without loss of generality, the set of the agent's actions is defined as:

$$A = \{a|a \in \{North, South, East, West, Still\}\} \tag{5.3}$$

The following subsections analyze the agent strategies of this game. For each timeslot, all agent strategies receive the state of the system (equation (5.1)) and decide to make an action (equation (5.3)).

5.3 Agent Strategies

Thus, each agent has to follow his strategy in order to find one human. A simple robust strategy that finds the optimal path from the starting point to the human in danger is created by representing the terrain as a graph. The open cells of the terrain are nodes and any transition from an open cell to a neighboring open cell is an edge. The strategy then can easily use the A^* search algorithm [37] for graphs. The algorithm is actually

[2]Note that the Map is actually static during a whole search or rescue phase. However, including it in the system state is preferable since it is used to define relative variables, such as the Manhattan distance of the agent from the human.

an extension of *Dijkstra*'s algorithm [38] for path finding. Since the agent is given the location of the goal, human or exit, at the start of each search or rescue phase respectively, an algorithm such as A* is ideal.

The pseudocode of the A* algorithm for the SAR terrain is shown in Figure B.1 in Appendix B. In terms of this section, it is only important to refer to the algorithm's main functionality. The A* algorithm finds the optimal path between two graph nodes, i.e. in this case terrain positions. In addition, since obstacle cells are considered non-reachable nodes (since no edges reach them), the algorithm avoids successfully any obstacles. The heuristics of the algorithm ensure that the path found is optimal (see Appendix B).

Hence, two strategies were created, one that selects the close human as a goal and implements the A* algorithm to find it and one that selects the far human and finds it using A*. These two strategies are named `AstarClose` and `AstarFar` respectively. The rescue phase is handled similarly, only this time both strategies choose the starting position as the exit position and find a path using A*. Finally, a probabilistic strategy was created that makes either the actions of the close A* algorithm or the actions of the far A* algorithms with a predefined probability. This type of agent shall be named `MAstar(p,q)` where `p` and `q` are the probabilities of selecting the action that `AstarClose` and `AstarFar` would perform respectively.

5.4 The Ad hoc Agent Strategy

This subsection analyzes how the methodology of Chapter 4 can be applied in the specific SAR domain. The application of the three components described in Chapter 4 are analyzed in the following subsections.

5.4.1 Policy Selection

The strategies defined in the previous subsection implement two interfaces, the interface `AgentStrategy` and the interface `AgentPolicy`[3]. Since all policies have to conform to the `AgentPolicy` API, the A* agent policies can easily be used as models. The abstraction of the policies allows the agent to create an instance `TeammatePolicy` for each teammate and an instance `MyPolicy` for himself.

[3]These two APIs are actually the core of all agents of the simulator. However, analyzing them deviates from the purpose of this dissertation. The interested reader can read sections A.1 and A.2 of Appendix A for a detailed description concerning the functionality of `AgentStrategy` and `AgentPolicy` respectively.

Each instance `TeammatePolicy` actually keeps the perceived model of the teammate. It contains all models of for any teammate and implements the Naïve Bayes classifier as defined in subsection 4.2. The `MyPolicy` instance keeps the answer models of the agent, as well as the mapping from any combination of models to a particular answer model. Finally, note that the models of any `TeammatePolicy` as well as the models of `MyPolicy` all have to implement interface `AgentPolicy`[4].

The agent can therefore either use one of the known models or create one of his own, as long as it implements the `AgentPolicy` interface. Thus, if he has a set of predefined policies, then he can set any one of them in action.

5.4.2 Strategy Construction

The RL strategy of subsection 4.4 is created upon determining a goal position for the agent. Then the game can be modeled as an MDP. At first, the state space and the actions are modeled according to equations (5.1) and (5.3). The reward function has the following form:

$$R_a(s,s') = \begin{cases} -1, & \text{if } s' = OBSTACLE \\ -10, & \text{if } s' = OPEN\_AREA \\ 100, & \text{if } s' = goalPosition \end{cases} \tag{5.4}$$

where s is the agent's current state[5], and a his action of choice that leads to a new state s'. Note that if the new state s' is an obstacle, then the agent receives the negative reward without, however, transitioning to it. However, it is considered a valid transition. The probability of ending up in a state is actually reduced to a transition function where all valid transitions are defined in a deterministic way. Any action from state s to state s' is considered valid as long as the distance between the two states' positions is 1 cell. Thus, the SAR system is successfully described by (2.4). Furthermore, it is obvious that the system's next state as well as the rewards depend only on the system's current state and the last action chosen by the agent. Hence, the SAR system satisfies the Markov property[6] (see (2.5)).

[4]The models in the `TeammatePolicy` instances actually extend the `BayesPolicy` abstract class, while the latter implements the `AgentPolicy` interface. See Figures A.5 and A.6 for a fine-grained view of the system.

[5]The state actually refers to the agent's perception of state, i.e. the terrain and his position in it.

[6]Typically, the system could be said to be a stochastic game rather than a simple MDP. However, since other agents are not modeled at this stage, the system is actually an MDP.

Upon defining the problem, applying an RL technique, such as Q-learning, to solve it is straightforward. Equations (2.17) and (2.18) are indeed fully applicable to the defined SAR problem. The steps followed by the algorithm are shown in Figure 5.3:

$Q(s,a) \leftarrow 0, \quad \forall s \in S,\ a \in A$

Observe initial state s

while $(s \neq$ goal_found$)$

$h(s) \leftarrow \arg\max_{a \in A} \left\{ Q(s,a) \right\}$

$$a \leftarrow \begin{cases} h(s), & if\ \ \varepsilon > Random([0,1]) \\ Random(A), & otherwise \end{cases}$$

Execute action a

Observe reward r and next state s'

$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \left[r + \gamma \cdot \max_{a' \in A(s')} \left\{ Q(s',a') \right\} - Q(s,a) \right]$

$s \leftarrow s'$

Figure 5.3: The steps of the Q-learning algorithm for a SAR terrain. During each timestep, the action is selected using e-greedy exploration, and the Q-values are updated according to the observed reward. The algorithm iterates until the goal is found.

Thus, according to Figure 5.3, the agent executes actions until he finds his goal (either human in danger or exit). The agent explores the terrain using ε-greedy exploration. Thus, the actions that he performs are either optimal with probability $1-\varepsilon$ or random with probability ε. This ensures that the agent's Q-values are updated uniformly. Finally, the number of simulated runs is given as a parameter to the agent.

As noted in subsection 4.4, when having more than one teammates, the ad hoc agent constructs a merger strategy to ensure scalability. As in subsection 4.4, for any state s and any action a, $Q_m(s,a)$ is the Q-value of model m, and $Q_M(s,a)$ is the Q-value of the merger of the policies. In this case, summing the Q-values (see equation (4.14)) might be sufficient. However, since Q-learning in the SAR terrain is used as a path-finding algorithm, it is preferable that the final Q-values have similar properties to the original ones. Hence, the Q-values have to be merged with a voting-like algorithm so as to preserve the so-called propensity of the agent towards a specific direction.

A simple example is provided to illustrate the notion of propensity in Q-values when it comes down to path-finding. Consider having 2 policies that have the Q-value $Q_1(s_1,a_1) = Q_2(s_1,a_1) = 40$ and 1 policy that has the Q-value $Q_3(s_1,a_1) = -100$ for a particular state s_1 and action a_1. Furthermore, suppose that performing action a_1 when

being in state s_1 results in the goal state s'_1. The merged Q-value $Q_M(s_1, a_1)$ should then have a value close to 40 because if the agents were to be treated as one, their path should indeed contain the action a_1 from the state s_1 as a preferable move since it actually results in a goal state. However, using equation (4.14), the final merged value is $Q_M(s_1, a_1) = 2 \cdot 40 + 1 \cdot (-100) = -20$, which means that the merger strategy would not consider going to s'_1. Generalizing, if s'_1 was an intermediate (as opposed to a goal) state, then the merger would follow a suboptimal path. Hence, identifying the main path of other agents actually comes down to find the path of the largest group of agents. In terms of the above example, the value 40 should receive 2 "votes", whereas the value -100 should receive 1 "vote".

Thus, each possible value of a Q-value should receive a vote for each agent that uses it. However, generalizing the above example, it is possible that certain Q-values are "close" to 40, and others "close" to -100, but none of them exactly 40 or -100. This proximity feature is successfully confronted using the k-means clustering algorithm. The algorithm of the *Merge* function is shown in Figure 5.4.

```
for (s ∈ States)
    for (a ∈ Actions)
        Initialize 2 clusters
        Set clusterA centroid to max{Q_m(s,a)}, ∀m ∈ Models
        Set clusterB centroid to min{Q_m(s,a)}, ∀m ∈ Models
        do
            for (m ∈ Policies)
                Assign Q_m(s,a) to the closest cluster.
            Set clusterA centroid to E(x), ∀x ∈clusterA
            Set clusterB centroid to E(x), ∀x ∈clusterB
        while (clusters have not changed)
        Q_M(s,a) =centroid of the cluster with most elements
```

Figure 5.4: The k-means voting *Merge* function. 2 clusters are created for each state-action pair and the largest cluster's average value is assigned to the merger's Q-value.

The algorithm iterates for every state-action pair and creates 2 clusters for each Q-value. Then, the k-means clustering algorithm is executed until convergence (see 2.2.2), assigning the Q-value of each model to a cluster. Finally, the merger's Q-value for every state-action pair is defined as the centroid of the largest cluster (in terms of population).

5.4.3 Teammate Modeling

As mentioned in subsection 4.3, the ad hoc agent has to observe his teammates for a number of timeslots in order to construct models of them. The simulator has the appropriate functionality that allows substituting agents. At first, a number of agents are created as active players and the ad hoc agent observes them for a (configurable) number of rounds. Then, in accordance with subsection 5.2, the ad hoc agent substitutes one of the active agents.

Applying the methodology described in subsection 4.3 is straightforward. Both training and usage phases can be easily used as long as the appropriate attributes as well as the class attribute are constructed. In other words, the problem is reduced to constructing the data preprocessor and the data postprocessor of Figures 4.2 and 4.3 respectively. The way these two processors are constructed is crucial for the effectiveness of the model.

Since the process is the same for all agents, without loss of generality, the case where the ad hoc agent has to model one teammate is considered. Thus, the attributes have to cover the state as seen from the teammate agent, while the class attribute has to cover the possible actions of the teammate agent. In addition, it is necessary that all these values are constructed in a relative way because the terrain is different for every round. For example consider the terrains of Figure 5.5 and assume the agent wants to go to the close human.

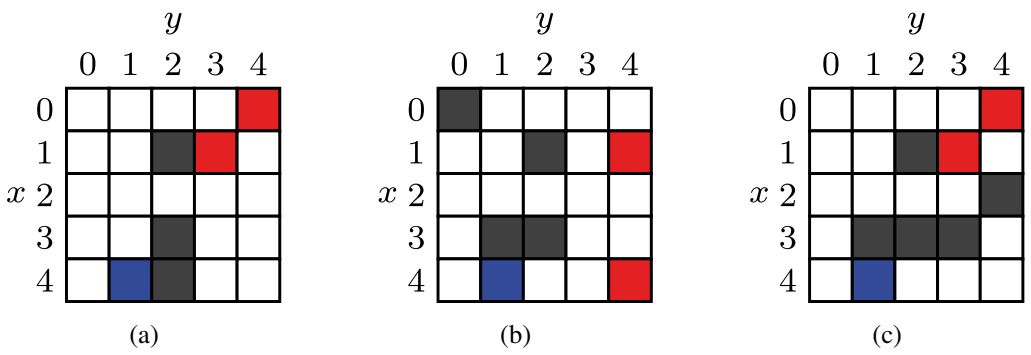
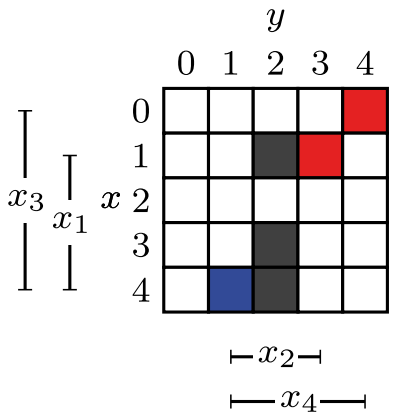


Figure 5.5: Examples of different terrains. The agent's optimal action depends on his environment (distance from human, obstacles) and not on his position (4,0).

In Figure 5.5a the agent may find that while being in position $(4,0)$, moving *North* is the optimal action. Although the agent's position in the terrains of Figures 5.5b and 5.5c is identical, the agent's optimal action in these terrains is *East* and *West* respectively. In other words, the agent's optimal action depends not only on his position

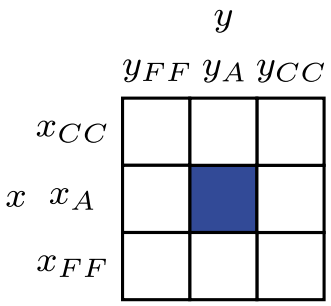
in the terrain but also on the full terrain state, consisting of humans, obstacles, and open area cells. This state actually provides an absolute representation of the attributes.

A relative representation is constructed as in Figure 5.6.

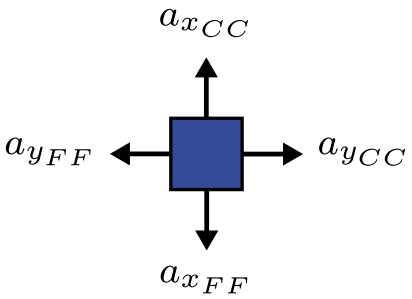


$$x_5 = 0 \ ((x_{CC}, y_A) = OPEN\_AREA)$$
$$x_6 = 0 \ ((x_{CF}, y_A) = UNDEFINED)$$
$$x_7 = 0 \ ((x_{FC}, y_A) = UNDEFINED)$$
$$x_8 = 1 \ ((x_{FF}, y_A) = OBSTACLE)$$
$$x_9 = 1 \ ((x_A, y_{CC}) = OBSTACLE)$$
$$x_{10} = 0 \ ((x_A, y_{CF}) = UNDEFINED)$$
$$x_{11} = 0 \ ((x_A, y_{FC}) = UNDEFINED)$$
$$x_{12} = 0 \ ((x_A, y_{FF}) = OPEN\_AREA)$$

(a) Attributes x_1–x_{12}



(b) Representation of positions



(c) Representation of actions – Attribute y

Figure 5.6: An example representation of the attributes. Attributes x_1–x_4 are distances of the agent from the humans, while x_5–x_{12} and y are defined using the agent's position and action respectively with relation to the positions of the close and the far human.

The first 4 attributes are defined as the absolute distance of the agent from the two humans, along the two axes. Formally, let A be the agent, C be the close and F be the far human, the first 4 attributes can be defined as:

$$x_1 = |x_A - x_C| \tag{5.5}$$

$$x_2 = |y_A - y_C| \tag{5.6}$$

$$x_3 = |x_A - x_F| \tag{5.7}$$

$$x_4 = |y_A - y_F| \tag{5.8}$$

Before defining the 8 remaining attributes, it is necessary that the action of the agent is also expressed in relative terms. The representation for the example of Figure 5.6a

can be seen in Figure 5.6b. The action of the agent is represented in relation with his position compared to the positions of the two humans. As before, the agent's position is (x_A, y_A). Any other position is defined as (x_{ij}, y_{kl}) where the first pointers, i and k, are defined with respect to the close human's position, and the second pointers, j and l, are defined with respect to the far human's position. Thus, if the agent moves from (x_A, y_A) to (x_{ij}, y_{kl}) and this action makes him move closer to the close human in the vertical axis, then $i = C$. Additionally, if the same move makes him move closer to the far human in the same axis, then $j = C$. Concerning the example of Figure 5.6a, the definition of the pointers for the neighbor positions of the agents is shown in 5.6b.

Finally, in accordance with the above discussion, the 8 remaining attributes can be defined as follows:

$$x_5 = \begin{cases} 1, & \text{if } (x_{CC}, y_A) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

$$x_6 = \begin{cases} 1, & \text{if } (x_{CF}, y_A) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.10}$$

$$x_7 = \begin{cases} 1, & \text{if } (x_{FC}, y_A) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.11}$$

$$x_8 = \begin{cases} 1, & \text{if } (x_{FF}, y_A) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.12}$$

$$x_9 = \begin{cases} 1, & \text{if } (x_A, y_{CC}) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.13}$$

$$x_{10} = \begin{cases} 1, & \text{if } (x_A, y_{CF}) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.14}$$

$$x_{11} = \begin{cases} 1, & \text{if } (x_A, y_{FC}) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.15}$$

$$x_{12} = \begin{cases} 1, & \text{if } (x_A, y_{FF}) = OBSTACLE \\ 0, & \text{otherwise} \end{cases} \tag{5.16}$$

The class attribute is actually defined similarly. As shown in Figure 5.6c, the action needed to move closer or further to the humans is now considered. Thus, the possible actions of the agent are given in the form:

$$y = \{a_{x_{CC}}, a_{x_{CF}}, a_{x_{FC}}, a_{x_{FF}}, a_{y_{CC}}, a_{y_{CF}}, a_{y_{FC}}, a_{y_{FF}}, a_S\} \tag{5.17}$$

where the notation $a_{x_{ij}}$ denotes the action needed in order to move from the current position (x_A, y_A) to position (x_{ij}, y_A) and the notation $a_{y_{kl}}$ denotes the action needed in order to move from the current position (x_A, y_A) to position (x_A, y_{kl}). Finally, the notation a_S refers to the agent staying still in his current position.

As one could observe, the relative representation of positions of Figure 5.6b and the relative representation of actions of Figure 5.6c are quite similar in terms of the input needed to acquire them. Thus, both mappings are handled using the decision tree given in Figure 5.7.

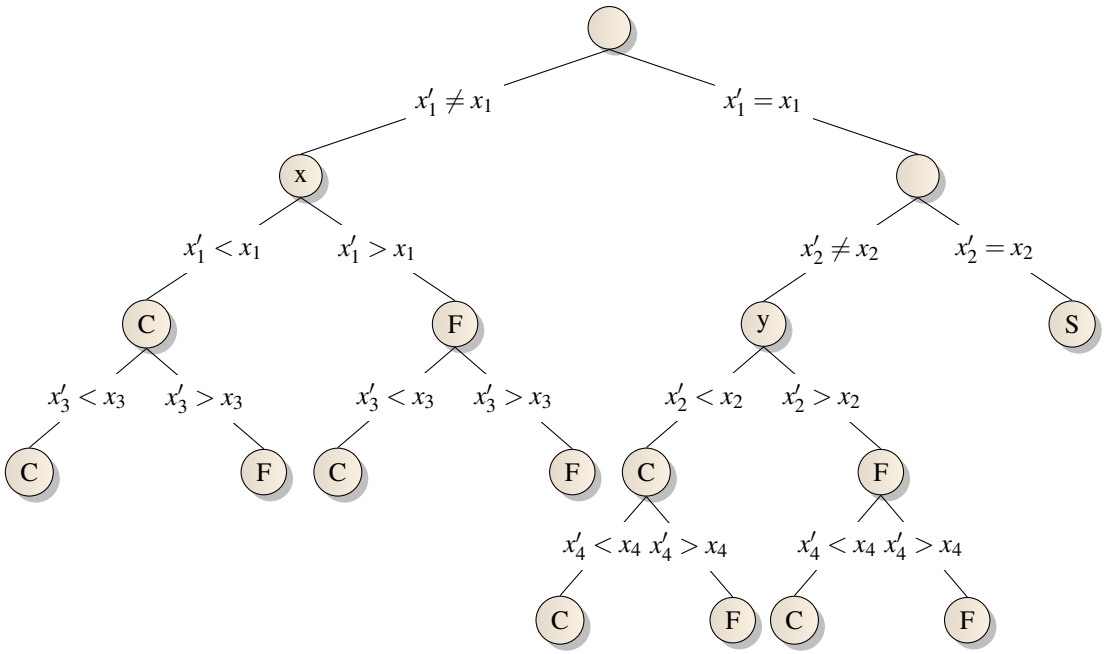


Figure 5.7: A decision tree that determines the output representation of an action given the current state and the next state that resulted from making the action.

The tree receives as input the attributes x_1–x_4 and x'_1–x'_4 which are created from states s and s' using equations (5.5)–(5.8). Its output is given in the form of $(x|y)_{((C|F)(C|F))|S}$ with | being the `OR` operator and parentheses simply declaring precedence, thus omitted in the final result. The notation is interpreted similarly to that of Figure 5.6b, e.g. if the output is x_{CF} then the position noted by the tree is closer to the close human and further from the far one.

The actions are interpreted similarly. Consider, for example, that the agent of Figure 5.6a makes the action *North*. Thus the agent's x distance from the close human has been $x_1 = 3$ and now is $x'_1 = 2$. In addition, his distance from the far human has been $x_3 = 4$ and now is $x'_3 = 3$. In addition, $x'_2 = x_2$ and $x'_4 = x_4$ since the agent did not move along the y dimension. Parsing the tree gives xCC, meaning that the observed

action ($North$), which is terrain-specific, is transformed to the output $d = a_{x_{CC}}$ which adequately describes the move regardless of the terrain.

Conclusively, the preprocessor and the postprocessor of Figures 4.2 and 4.3 respectively can be defined. The preprocessor actually has two different tasks:

- Create a data instance containing the class attribute given the state of the system, the teammate's position and his action.
- Create a data instance not containing the class attribute given the state of the system and the teammate's position.

The first task is handled using the algorithm shown in Figure 5.8

```
Input: s, a
Output: x1-x12, d
Find x1-x4 using equations (5.5)-(5.8)
for (at ∈ {North, South, East, West, Still})
    Find s' given state s and action a
    Find x'1-x'4 using equations (5.5)-(5.8)
    dt = representation of at using the decision tree of Figure 5.7
    if (a = at)
        d = dt
    Find one of x5-x12 using respective equation ((5.9)-(5.16))
Assign all attributes x5-x12 that were not found to 0
return x1-x12, d
```

Figure 5.8: The preprocessor algorithm that receives as input a state-action pair and returns a data instance which includes the class attribute. The algorithm iterates over all possible actions and uses the attribute representation of each action in order to construct the data instance.

As shown in Figure 5.8, the first four attributes are easily computable. After that, an iteration over all possible actions a_t is made and for each action its representation d_t is found using the decision tree. Then, the representation is used in order to find the 8 remaining attributes. Finally, the class attribute is actually one of the checked actions, thus its respective representation d is also found and returned.

The second task of the preprocessor is similarly handled using the algorithm shown in Figure 5.9.

```
Input: s
Output: x1-x12
Find x1-x4 using equations (5.5)-(5.8)
for (at ∈ {North, South, East, West, Still})
    Find s' given state s and action a
    Find x'1-x'4 using equations (5.5)-(5.8)
    dt = representation of at using the decision tree of Figure 5.7
    Find one of x5-x12 using respective equation ((5.9)-(5.16))
Assign all attributes x5-x12 that were not found to 0
return x1-x12
```

Figure 5.9: The preprocessor algorithm that receives as input a state and returns a data instance which does not include a value for the class attribute. The algorithm iterates over all possible actions and constructs the data instance using the attribute representation for each direction.

As shown in Figure 5.9, the first four attributes are once again computed easily. After that, an iteration over all possible actions a_t is made and for each action its representation d_t is found using the decision tree. Then, the representation is used in order to find the 8 remaining attributes.

Finally, the postprocessor handles the task of receiving the current state s and the classifier's output y and transforming the latter to an action a (see Figure 4.3). This is accomplished using the algorithm shown in Figure 5.10.

```
Input: x, y, yProbabilities
Output: ActionProbabilities
Find x1-x4 using equations (5.5)-(5.8)
for (at ∈ {North, South, East, West, Still})
    Find s' given state s and action a
    Find x'1-x'4 using equations (5.5)-(5.8)
    dt = representation of at using the decision tree of Figure 5.7
    if (y[a] = d)
        ActionProbabilities[a] = yProbabilities[a]
        normalizer = normalizer + ActionProbabilities[a]
for (a ∈ {North, South, East, West, Still})
    ActionProbabilities[a] = ActionProbabilities[a] / normalizer
```

Figure 5.10: The postprocessor algorithm that receives as input the current values of the attributes and the predicted class attribute and returns the predicted action. The algorithm maps the distribution given by the classifier to a respective distribution of valid actions.

As shown in Figure 5.10, an iteration over all possible actions is made and each action is checked for its representation in the current terrain using Figure 5.7. Thus, upon mapping from the set of actions to the set of the possible values of the class attribute, the probability of every action is assigned to the probability of the respecting class attribute value. Finally, note that the mapping of the attributes is safe, meaning that it is not possible for the sum of the probabilities to be more than 1. However, it is possible for it to be less than 1, since not all class attribute values are necessarily represented by valid actions. Hence, it is necessary to normalize the values as in Figure 5.10.

Finally, little discussion has to be made about the choice of a classifier in order to classify properly the class attribute, i.e. actually predict the next action. As long as the classifier is complex enough to handle the number of attributes (which is rather large), satisfactory results shall be produced. Since the number of observing timeslots is not very large, the classifier's speed is rather irrelevant. As far as its accuracy is concerned, there are no proper metrics due to the nature of the problem. In other words, the input and output of the classifier does not coincide with the input and output of the system. Only qualitative claims can be made for any relationship between them. In terms of this particular problem, the C4.5 classifier was chosen (see subsection 2.3.2). The implementation used is the *J48*, an open-source implementation of the algorithm included in the *Waikato Environment for Knowledge Analysis (Weka)* [39].

Chapter 6

Experiments

6.1 Overview

The experiments conducted test the ad hoc agent's performance along two basic axes: effectiveness and efficiency. Concerning effectiveness, the objective is to determine whether a team containing the ad hoc agent can perform near-optimally when compared to other teams. The metric used to measure effectiveness is the mean number of timesteps required by a team in order to successfully go to the humans in danger and return to the entry point. As far as efficiency is concerned, the mean time per timestep in milliseconds is used to determine whether a particular team is more efficient than another[1]. The experiments are conducted along the following axes:

- *Known teammate models*
  The ad hoc agent has a set of known models that his teammates follow. Two cases are considered regarding the ad hoc agent's strategy:
  - *The agent follows one of the known models.*
  - *The agent constructs his own strategy to follow.*
- *Unknown teammate models*
  The ad hoc agent does not know his teammates' models so he has to construct them. In addition, the agent constructs his own strategy to follow.

The results of the experiments are presented in the following subsections.

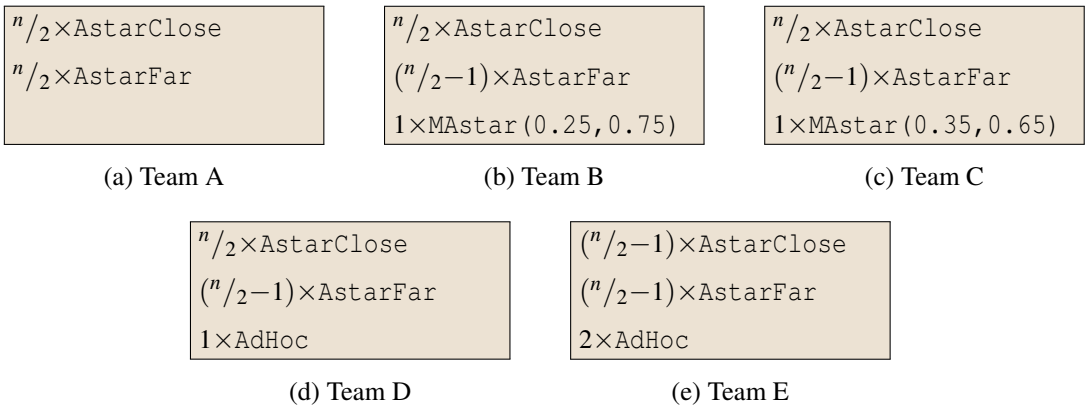
[1]Note that the agents of each team are fully synchronized. Thus, each team's efficiency is actually measured as the efficiency of the team's "slowest" agent.

6.2 Known Teammate Models

Concerning this set of experiments, the ad hoc agent is given a set of defined policies as black boxes. Thus, the agent has to understand which model is followed by each one of his teammate agents. Two cases are distinguished according to whether the agent follows one of the known policies or constructs his own policy.

6.2.1 Using Modeled Policy

The ad hoc agent, upon determining the policies followed by the teammate agents, selects one among them as his own strategy. The experiments of this subsection aim to demonstrate the effectiveness of the Naïve Bayes classifying method used for policy selection (see subsection 4.2). The evaluation method resembles the one proposed by S. Barrett et al. [25]. Five teams, each having n agents, are defined as in Figure 6.1.



(a) Team A (b) Team B (c) Team C

(d) Team D (e) Team E

Figure 6.1: Three simple A* teams, and two teams with 1 and 2 ad hoc agents. Team A is optimal, whereas teams B and C have one sub-optimal player each, and teams D and E have 1 and 2 ad hoc agents respectively.

As shown in Figure 6.1, the teams are quite similar; for example teams A, B, C, and D have $n-1$ identical agents and only 1 agent that is different for every team. However, note that the `AstarClose` and the `AstarFar` agents are actually optimal (see subsection 5.3). The algorithms always find the shortest path[2] to the close and far human respectively. Hence, team A is obviously optimal because, according to the game specifics, agents have to be equally shared between the two humans.

As a result of the above discussion, comparing two agents comes down to comparing two teams that have $n-1$ optimal agents and 1 different agent, since the latter

[2]See Appendix B for the implementation of A*, along with a discussion about his optimality.

is definitely less or equally effective to his optimal counterpart. Teams B and C, each have `MAstar` agent that selects the optimal action with probability 75% and 65% respectively. Finally, team D has an ad hoc agent that has to select among the policies he is given, while team E has 2 ad hoc agents.

Concerning the specifics of the following experiments, each team consists of $n = 4$ agents. In addition, the ad hoc agent is given a set of 4 possible policies: {`AstarClose`, `AstarFar`, `MAstar(0.25,0.75)`, `MAstar(0.75,0.25)`}[3], which he uses not only to determine which policy is selected by his teammates but also to select a policy for himself. The goal of the following experiments is to prove that team D (the ad hoc agent's team) is more effective than teams B and C and comes quite close to team A. Concerning team E (team having 2 ad hoc agents), its purpose is to study the effect that one adaptive strategy has to another. Thus, team E is also expected to have satisfactory results compared to teams B and C. However, its effectiveness shall be compared with team D as an attempt to demonstrate that the strategy is powerful enough when having non-stationary teammates.

All teams entered 10 terrains, and each terrain run was repeated 10 times. In particular, each run in a specific terrain provides with the number of timesteps required for a full game scenario. Thus, each run is repeated 10 times, providing with the mean numbers of timesteps and the standard deviation for the specific terrain. Full results are shown in subsection C.1.1 of Appendix C. Table 6.1 shows the total number of timesteps computed as the sum of the mean number of timesteps (and the sum of the respective standard deviations) for 10 terrains for 3 different terrain sizes: 6×10, 9×15, and 12×20.

Table 6.1: Total number of timesteps for 10 terrains, testing the ad hoc agent's policy selection component (teams D and E with 1 and 2 ad hoc agents respectively) against one optimal (team A) and two suboptimal teams (teams B and C).

	Total Number of Timesteps for Terrains of size		
	6×10	9×15	12×20
Team A	146.0 ± 0.0	232.0 ± 0.0	320.0 ± 0.0
Team B	179.8 ± 32.0	301.8 ± 65.7	401.5 ± 60.9
Team C	215.7 ± 74.6	375.4 ± 112.3	518.0 ± 120.2
Team D	152.5 ± 5.2	236.2 ± 5.1	327.7 ± 6.2
Team E	165.1 ± 23.5	249.5 ± 19.4	335.6 ± 22.3

[3] As seen in Figure 6.1, no mixed A^* strategies are actually used in team D. Hence, the mixed A^* policies are included in the set of possible policies in order to obscure the agent's policy selection model.

The graph of Figure 6.2 provides a straightforward illustration of the results.

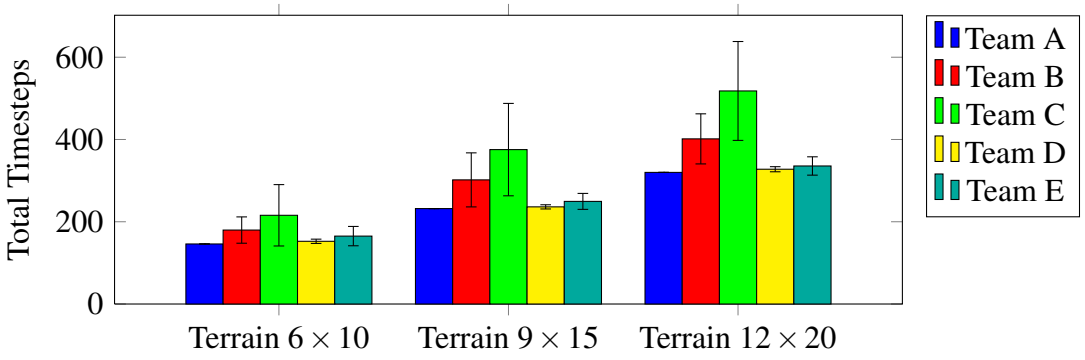


Figure 6.2: Graph showing the total number of timesteps, testing the policy selection component of the ad hoc agent (teams D and E with 1 and 2 ad hoc agents respectively) against one optimal (team A) and two suboptimal teams (teams B and C).

As seen in that Figure, the ad hoc agent's team (team D) indeed has very positive results. It outperforms all suboptimal strategies (teams B and C), while its effectiveness is quite similar to the optimal (team A). Thus, the agent's policy selection component is quite effective when the agent observes stationary teammates. Furthermore, the team that has 2 ad hoc agents (team E) has similar performance. As seen in Table 6.1, team E also outperforms all suboptimal strategies, being slightly less effective than team D in most cases. Although one might expect that the ad hoc agents would confuse one another by obscuring the classification task, the agents quickly manage to decide on an effective sharing of the tasks, with each one of them going to a different human.

6.2.2 Constructing Strategy

The experiments of this subsection aim to demonstrate the efficiency of the Reinforcement Learning model used for strategy construction (see subsection 4.4). Two new teams are defined as in Figure 6.3.

$^{n}/_{2}\times$`AstarClose`
$(^{n}/_{2}-1)\times$`AstarFar`
$1\times$`AdHocWithRL`

(a) Team F

$^{n}/_{2}\times$`AstarClose`
$(^{n}/_{2}-1)\times$`AstarFar`
$1\times$`AdHocWithRLMerger`

(b) Team G

Figure 6.3: The two Q-learning teams. The ad hoc agent of team F constructs an answer policy for all possible combinations of teammate policies, whereas the agent of team G constructs answer policies for the distinct teammate policies and merges them.

As in the previous subsection, both teams have $n-1$ identical optimal agents and 1 ad hoc agent each. The ad hoc agent of team F constructs an answer policy using Q-learning for all possible combinations of agent policies, whereas the agent of team G constructs an answer policy for all distinct policies and merges their Q-values for any combination (see subsection 4.4). Due to the optimality of the $n-1$ agents for each team, the two ad hoc agents' performance is isolated. In addition, the measured performance is now decomposed to both effectiveness and efficiency. Apart from the ad hoc agent being less or equally effective than his teammates, he is also less efficient. The efficiency of the ad hoc agent's is obviously expected to be lower than any A^* agent's, since the former computes (at least) a Q-learning answer policy for each possible policy. The learning procedure is indeed "heavy", since the Q-learner has to simulate a number of runs in order to compute the Q-values (see subsection 4.4).

Finally, since policy selection is not tested in this subsection, only 2 possible teammate policies are considered {`AstarClose, AstarFar`}. Thus, the influence of policy selection is minimized, whereas strategy construction is thoroughly tested.

6.2.2.1 Efficiency Tests

The first set of experiments aim to demonstrate the efficiency of the merger strategy (team G), as opposed to the naïve strategy that constructs an answer policy for each combination of policies (team F). Since the number of possible teammate policies is restricted to 2, the two teams' performance is tested with respect to the number of agents. The latter is highly relevant because of the computational complexity of the algorithms. As mentioned in subsection 4.4, the complexity of the naïve strategy is exponential with respect to the number of agents. By contrast, the merger strategy achieves reducing the complexity so that it is linearly proportional to the number of policies[4]. Finally, the different terrain sizes shall provide another interesting metric, since each simulated run of the Q-learning algorithm is completed by finding the target. Thus, larger terrains are expected to have higher overhead for both teams.

As a result of the above discussion, the experiments were conducted for teams of 2, 4, 6, and 8 agents in 3 terrain sizes: 6×10, 9×15, and 12×20. Teams F and G entered 3 terrains, and each terrain run was repeated 3 times, providing with the mean time per timestep and the standard deviation for this terrain. Full results are shown in subsection C.1.2.1 of Appendix C. Table 6.2 shows the average time per timestep

[4]Note, however, that an exponential overhead is expected, since the agent still has to create a strategy for all possible combinations of agents and policies.

computed as the sum of the mean time per timestep (and the sum of the respective standard deviations) needed for each agent, concerning different number of agents for each one of the terrain sizes.

Table 6.2: Average time per timestep for 3 terrains, testing the efficiency of the agent that constructs an answer policy for each combination of teammate policies (team F) versus that of the agent that merges the distinct answer policies (team G), for different number of agents.

		Average Time per Timesteps for Terrains of size		
		6×10	9×15	12×20
2 agents	Team F	44.1 ± 1.4	54.8 ± 1.8	70.6 ± 0.9
	Team G	57.6 ± 5.5	74.8 ± 3.8	93.2 ± 3.1
4 agents	Team F	174.1 ± 13.9	229.1 ± 5.7	341.4 ± 8.5
	Team G	61.0 ± 4.4	82.4 ± 1.8	106.7 ± 1.4
6 agents	Team F	848.0 ± 57.2	1233.7 ± 63.3	1752.7 ± 16.0
	Team G	81.2 ± 5.3	106.0 ± 2.7	126.9 ± 2.2
8 agents	Team F	3819.4 ± 254.8	5373.4 ± 88.3	8044.8 ± 443.4
	Team G	155.5 ± 12.6	202.3 ± 6.9	213.8 ± 4.1

As shown in Table 6.2, running the experiments for 3 terrains (as opposed to 10 in the previous subsection) is sufficient since the difference between the time per timestep for the two teams is significant.

The graph of Figure 6.4 provides an illustration of the results for the two teams.

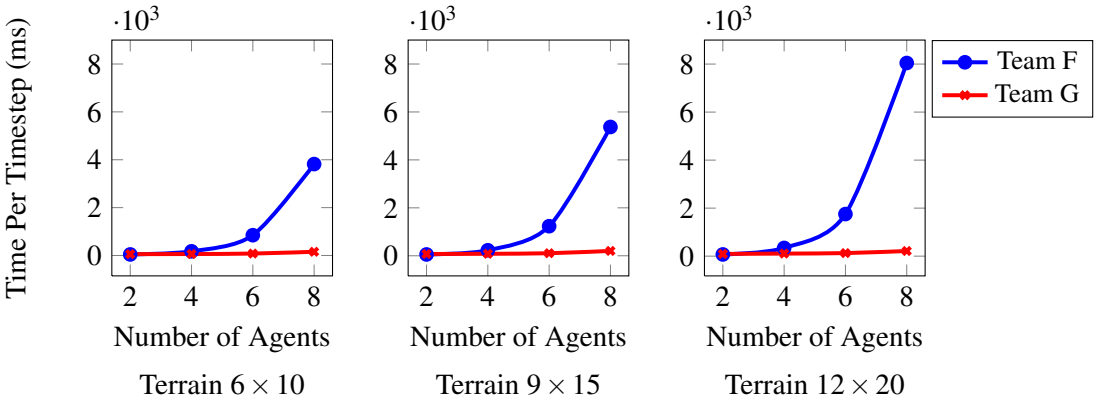


Figure 6.4: Graph showing the average time per timestep, testing two strategy construction components of the ad hoc agent for different number of agents, one that constructs answer policies for all combinations of teammate policies (team F), and one that merges constructed answer policies (team G).

As seen in Figure 6.4, the merger model is generally more efficient than its naïve counterpart. This is actually expected since the calculation of Q-values is quite a "heavy" procedure in terms of performance. For instance, having 3 teammate agents (i.e. 4 agents in total) and 2 possible policies means that the ad hoc agent of team F computes $2^3 = 8$ Q-value arrays, whereas the agent of team G computes only 2 Q-value arrays. Observing Figure 6.4, it is obvious that the merger model is much more scalable than the naïve strategy construction model.

Note, however, that both models have exponential complexity. This is observable in Table 6.2, where the average time per timestep for both teams grows exponentially with the number of agents. Nevertheless, the merger model's performance is much better since only the merging procedure is performed in exponential complexity. The overhead of the merge phase is also evident by the fact that the merger strategy is outperformed by its naïve counterpart when there is only 1 teammate (i.e. 2 agents in total). This is expected since in that case both ad hoc agents construct the same number of Q-learning policies (1 policy), thus the merger's overhead is observable. However, the aforementioned overhead is rather insignificant when compared to the overall gain in efficiency for larger teams.

Concerning the various terrain sizes, both teams have a significant performance overhead in larger terrains. However, by examining the Table 6.2, one can conclude that relative overhead is different for the two teams (this is also noticeable in Figure 6.4 since all three diagrams are in the same y-axis scale). For instance, concerning team F, its average time per timestep for terrains of size 12×20 is approximately double the one for terrains of size 6×10. This is expected since any simulated run of the Q-learning algorithm is completed when the goal is found and the number of timesteps needed is increased in large terrains. By contrast, observing the respective values for team G, the slope is rather smaller. Hence, although both agents have to compute more Q-values for any policy, since the naïve approach once again computes the Q-values for all combinations of policies, its performance is severely degraded.

6.2.2.2 Effectiveness Tests

Upon demonstrating the efficiency of the strategy construction models (teams F and G), their effectiveness is tested against teams A and D. Furthermore, a new team that has 2 ad hoc agents is created, as in Figure 6.5.

$(n/2-1)\times$AstarClose
$(n/2-1)\times$AstarFar
$2\times$AdHocWithRLMerger

Figure 6.5: Team H, a team that has 2 ad hoc agents that construct answer policies for the distinct teammate policies and merge them.

Concerning the following experiments, each team consists of $n = 4$ agents and, as mentioned above, the possible policies given to the teams are 2 ($\{$AstarClose, AstarFar$\}$) so as to isolate the performance of the strategy construction component.

The purposes of the following experiments are multiple. At first, the effectiveness of team G (containing the ad hoc agent that merges answer policies) is measured against teams A and D. Thus, the strategy construction model is compared to an optimal strategy (team A) as well as a strategy that detects teammate policies, yet counteracts using fixed optimal models (team D), instead of constructing its own policy. In addition, team F (naïve ad hoc with RL approach) must not have such significant gain over team G that could justify the need of using it despite its lack of scalability. Finally, concerning team H (team having 2 ad hoc agents that merge answer policies), the effect that one adaptive agent has to another is once again explored.

Once again, all teams entered 10 terrains for 10 runs per terrain, providing with the mean number of timesteps and the standard deviation for each terrain. Full results are shown in subsection C.1.2.2 of Appendix C. Table 6.3 contains the total number of timesteps computed as the sum of the mean number of timesteps (and the respective deviations) for 10 terrains for the 3 terrain sizes: 6×10, 9×15, and 12×20.

Table 6.3: Total number of timesteps for 10 terrains, testing the strategy construction components (teams G and H with 1 and 2 agents with merger components respectively, and team F with 1 agent that constructs all possible answer policies) against an optimal team (team A), and a team with an ad hoc agent that follows optimal strategies (team D).

	Total Number of Timesteps for Terrains of size		
	6×10	9×15	12×20
Team A	146.0 ± 0.0	232.0 ± 0.0	320.0 ± 0.0
Team D	151.8 ± 5.4	236.6 ± 5.2	326.9 ± 6.1
Team F	152.2 ± 5.8	245.3 ± 9.7	327.9 ± 8.1
Team G	153.3 ± 6.2	246.9 ± 12.2	328.1 ± 7.3
Team H	163.1 ± 21.4	258.5 ± 18.5	341.9 ± 26.3

At first glance, the results for all agent teams of Table 6.3 are actually quite similar. This is also illustrated by the graph of Figure 6.6.

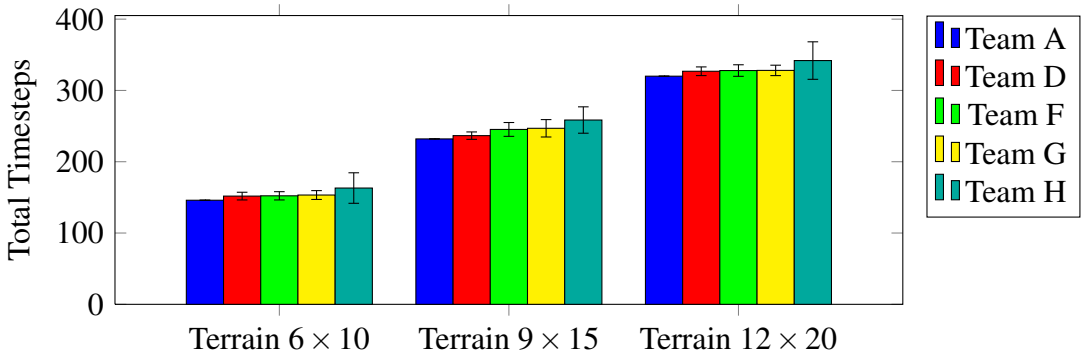


Figure 6.6: Graph showing the total number of timesteps, testing the strategy construction component of the ad hoc agent with answer policy merger (teams G and H with 1 and 2 ad hoc agents respectively) against its naïve counterpart that constructs answer policies for all possible combinations (team F), as well as against an optimal team (team A), and a team with an ad hoc agent that follows optimal strategies (team D).

As seen in Figure 6.6, the merger agent's team (team G) performs almost as effectively as its simple ad hoc counterpart that follows one of the predefined optimal policies (team D). Thus, the strategy construction component is highly effective, considering that it successfully creates an answer policy that is significantly close to the optimal. Furthermore, the total number of timesteps required by teams F and G for a full scenario is approximately equal in all cases. Consequently, the k-means *Merge* function of subsection 5.4.2 has been a good fit for the problem. Thus, team G should be used in place of team F, as long as the *Merge* function is appropriate for the game specifics.

Finally, the team that has 2 ad hoc agents (team H) is once again quite effective, since the results obtained are only slightly worse than those of team G. This is actually quite interesting since the agents of team H have to classify one another to one of the known policies. However, as opposed to the experiments of subsection 6.2.1, the agents do not follow one of the known policies. Thus, in other words, each ad hoc agent has to determine which policy his ad hoc teammate follows, while the latter follows a self-constructed version of the policies. Consequently, the results for team H indicate not only that the agents effectively classify one another but also that the policies they construct are such close to the optimal that the classification task is not obscured. Finally, note that the sum of standard deviations of team H is rather large compared to the one of team G. However, this is generally inevitable, yet acceptable, concerning the difficulty of the problem.

6.3 Unknown Teammate Models

Concerning this set of experiments, the policies followed by the ad hoc agent's teammates are unknown. Thus, the ad hoc agent has to construct these policies on his own, upon observing his teammates. The effectiveness of the teammate modeling component as well as the agent as a whole (see subsection 4.3) is tested in this subsection. Two new teams are defined as in Figure 6.7.

$^{n}/_{2}\times$AstarClose
$(^{n}/_{2}-1)\times$AstarFar
$1\times$AdHocModelerWithRL

(a) Team I

$(^{n}/_{2}-1)\times$AstarClose
$(^{n}/_{2}-1)\times$AstarFar
$2\times$AdHocModelerWithRL

(b) Team J

Figure 6.7: The ad hoc modeler teams. Teams I and J each have 1 and 2 ad hoc agents respectively, where the ad hoc agents construct models for their teammates, determine which of these models are followed by each one, and construct an efficient answer strategy.

As shown in Figure 6.7, the teams are defined similarly to those of the previous subsections. Team I has $n-1$ optimal agents and 1 ad hoc modeler agent, while team J has $n-2$ optimal agents and 2 ad hoc modeler agents.

The ad hoc modeler agent constructs his teammates' possible policies upon collecting data from 5 random terrain runs, with team A playing (see Figure 6.1). Since Team A is optimal, high quality instances are supplied to the agent's classifier (C4.5 algorithm as in subsection 5.4.3). Upon collecting data instances, the ad hoc agent's team (team I) replaces team A in the role of the active playing team. As in subsection 6.2.1, the ad hoc agent's efficiency is compared against the optimal team A as well as the suboptimal teams B and C.

Concerning the specifics of the following experiments, teams of $n = 2$ agents are considered and no information is provided to the agent[5]. Hence, the main goal of the following experiments is to explore whether the ad hoc agent can successfully operate in a previously unknown environment using as less data as possible. As far as team J (team with 2 ad hoc modeler agents) is concerned, the main goal is to explore whether a team of non-stationary ad hoc teammates can be more or at least equally effective to teams B and C, and achieve performance as close to team A as possible. Furthermore,

[5] Actually, the only information provided to the agent is the fact that he has to create 2 agent policies.

the performance of team J is compared to that of team I in order to analyze the effect that one ad hoc agent has to another.

All teams entered 10 terrains, and each terrain run was repeated 10 times. The mean number of timesteps as well as the respective standard deviation for each terrain is shown in subsection C.2 of Appendix C. The total number of timesteps for all terrains is again computed as the sum of the mean numbers of timesteps (and the sum of the respective standard deviations). Table 6.4 contains the results for 10 terrains for the 3 terrains sizes: 6×10, 9×15, and 12×20.

Table 6.4: Total number of timesteps for 10 terrains, testing the ad hoc agent's teammate modeling component (teams I and J with 1 and 2 ad hoc agents respectively) as well as his overall effectiveness against one optimal (team A) and two suboptimal teams (teams B and C).

	Total Number of Timesteps for Terrains of size		
	6×10	9×15	12×20
Team A	146.0 ± 0.0	232.0 ± 0.0	320.0 ± 0.0
Team B	169.1 ± 24.2	298.2 ± 51.0	412.1 ± 66.9
Team C	213.2 ± 64.5	370.4 ± 113.0	520.6 ± 128.6
Team I	155.4 ± 4.9	256.2 ± 6.7	333.9 ± 9.7
Team J	180.6 ± 40.0	270.0 ± 45.3	354.3 ± 37.8

The results shown in Table 6.4 are actually quite encouraging. The graph of Figure 6.8 provides an illustration of those results.

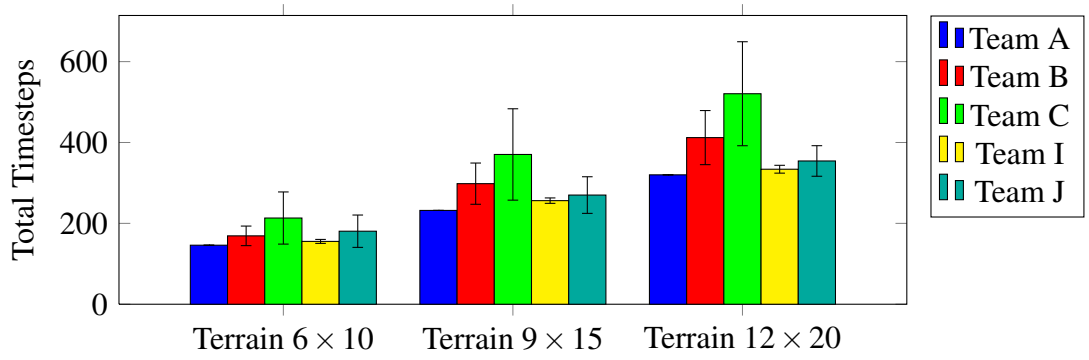


Figure 6.8: Graph showing the total number of timesteps, testing the teammate modeling component as well as the overall effectiveness of the ad hoc agent (teams I and J with 1 and 2 ad hoc agents respectively) against one optimal (team A) and two suboptimal teams (teams B and C).

As seen in Figure 6.8, the ad hoc modeler's effectiveness is indeed remarkable. Team I not only outperforms all suboptimal strategies (teams B and C), but also its effective-

ness is quite similar to the optimal (team A). The teammate modeling component of the ad hoc agent is quite effective as long as the observed runs provided to the agent are enough (see next subsection for an analysis of how many runs are enough). In addition, the results shown in this subsection are very interesting since they actually apply to the full ad hoc agent implementation with no prior information. Thus, as far as the SAR domain of Chapter 5 is concerned, the ad hoc agent has managed to effectively (and efficiently) participate in a game with unknown teammates.

Finally, the results for team J are also quite remarkable. Although the team consists only of ad hoc agents, their cooperation is satisfactory since they actually manage to outperform the agents of teams B and C, while their effectiveness is comparable to that of team I. Further interpreting the results, all the components of the ad hoc agents of Team J perform sufficiently, even if the problem becomes quite hard. In specific, each ad hoc agent constructs policy models to represent the optimal strategies that he expects his teammate to follow. However, both he and his teammate follow self-constructed policies that are based on suboptimal models. Nevertheless, the agents manage to effectively model unknown policies, create their own policies and classify the strategy of one another. In addition, although the sum of standard deviations for team J may seem slightly large, it is generally acceptable concerning the problem's difficulty. For example, a 12×20 terrain scenario can be run in approximately 35.43 timesteps with overall 3.78 more or fewer timesteps.

6.4 Learning Sensitivity Analysis

As a result of the analysis given in the above subsections, the ad hoc agent is effective enough even when the information provided to him is very limited. However, the agent's efficiency and effectiveness depend mainly on the parameters of the learning methods used. In particular, the performance of the C4.5 classifier used for teammate modeling (see subsection 5.4.3) depends on the number of runs that the ad hoc agent is allowed to observe. In addition, the quality of the constructed strategy (i.e. the Q-values, see subsection 5.4.2) produced by the Q-learning algorithm depends on the number of runs that the algorithm simulates. The effect of these parameters is analyzed in this subsection.

Thus, according to the above analysis, two parameters actually determine the ad hoc agent's effectiveness. A new set of experiments is defined similarly to those of the previous subsections. The teams tested are all of type team I (see Figure 6.7a).

The ad hoc agent's parameters are indicated using the notation team $I_{K,L}$, where K and L stand for the number of observed runs for teammate modeling and the number of simulated runs for strategy construction respectively. K belongs to the set $\{3,5,7\}$ and L belongs to the set $\{300,500,700\}$. Note that the lowest values of K and L (3 and 300 respectively) are the limit values, lower than which the agent fails to perform the required task. Consequently, the main purpose of these experiments is to explore the relation between the parameters analyzed above with the agent's effectiveness[6].

As in the previous subsection, teams of $n = 2$ agents are considered. Furthermore, each team of agents is tested on 10 terrains providing with the mean number of timesteps and the standard deviation for each terrain. Full results are shown in subsection C.3 of Appendix C. Table 6.5 contains the total number of timesteps computed as the sum of the mean number of timesteps (and the respective deviations) for the 3 terrain sizes: 6×10, 9×15, and 12×20.

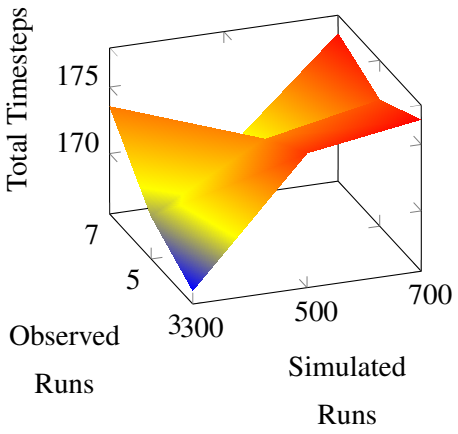
Table 6.5: Total number of timesteps for 10 terrains, testing the sensitivity of the ad hoc agent's learning parameters. Team $I_{K,L}$ contains an ad hoc agent who is given K observed runs to model his teammates and L simulated runs to train his strategy.

	Total Number of Timesteps for Terrains of size		
	6×10	9×15	12×20
Team $I_{3,300}$	166.5 ± 6.9	263.2 ± 10.8	372.9 ± 15.7
Team $I_{3,500}$	175.6 ± 6.4	273.6 ± 10.6	361.9 ± 11.1
Team $I_{3,700}$	176.9 ± 12.0	271.3 ± 13.6	368.0 ± 7.1
Team $I_{5,300}$	168.8 ± 7.7	260.2 ± 12.7	372.4 ± 10.1
Team $I_{5,500}$	173.3 ± 9.0	257.0 ± 8.3	366.3 ± 9.2
Team $I_{5,700}$	175.0 ± 7.2	276.3 ± 7.6	357.2 ± 8.2
Team $I_{7,300}$	173.6 ± 8.8	262.1 ± 9.5	373.7 ± 11.3
Team $I_{7,500}$	168.8 ± 6.0	261.9 ± 11.7	368.3 ± 7.7
Team $I_{7,700}$	176.6 ± 7.7	270.7 ± 13.6	354.6 ± 11.3

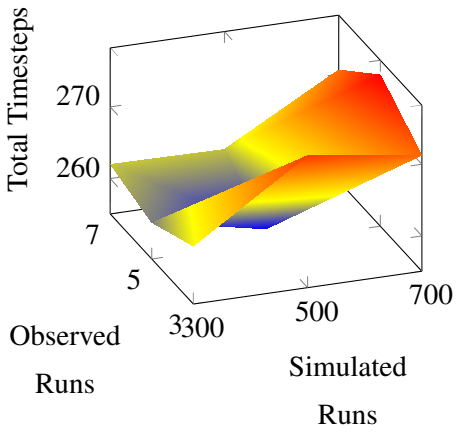
Observing Table 6.5, one could say that the differences among the timesteps required among the various number of agents are rather small.

A rather more substantial illustration of the results is shown in Figure 6.9, which illustrates how the sum of the average values is influenced by the values of the two fundamental parameters, number of observed runs and number of simulated Q-learning runs.

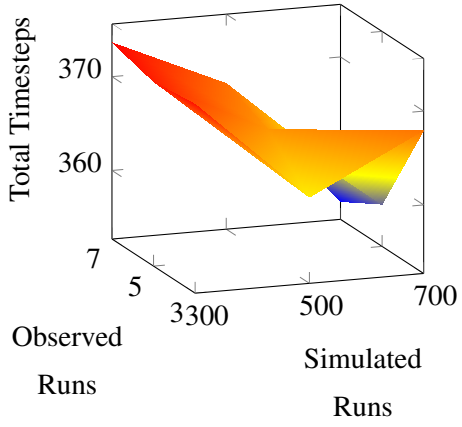
[6]Concerning efficiency is rather redundant since all agents considered in this subsection are efficient, regardless the values of the two parameters.



(a) Terrain 6×10



(b) Terrain 9×15



(c) Terrain 12×20

Figure 6.9: Graph showing the total number of timesteps needed versus the two learning parameters: the number of observed runs for teammate modeling and the number of simulated runs for strategy construction.

The diagrams of Figure 6.9 provide with interesting insight concerning the amount of learning required to ensure optimal behavior.

As seen in Figure 6.9a, the learning needed in a small (and thus easier) terrain is also relatively small, since having 3 runs to observe and 300 simulated runs is efficient. Interestingly enough, the 9×15 terrain (see Figure 6.9b) seems to be run optimally when medium values are selected, such as 5 observed and 500 simulated runs. By contrast, extreme values for both learning parameters seem to have worse performance. Finally, concerning large terrains, such as the one of Figure 6.9c, the ad hoc agent performs optimally when he has more timesteps to observe as well as more steps to constructs his strategy.

Intuitively, since the terrain's size actually represents the game's difficulty, the amount of learning required is proportional to the game's difficulty. Thus, if the amount of learning is less than the optimum, it is obvious that the agent performs worse than his optimum potential. Further analyzing the diagrams of Figure 6.9, one could also observe that when the amount of learning is more than the optimum, then the agent does not perform any better. This behavior is actually the result of over-training the learning techniques. In ML terms, the model created by the classifier of the teammate modeling component *overfits*.

As mentioned in subsection 5.4.3, each data instance that is provided to the C4.5 classifier is extracted from the state and the action of the agent, in terms of a specific terrain run. These state-action pairs are actually remapped along two main axes: the agent's distance from the humans, and the agent's neighboring cells, either obstacles or open areas. Thus, consider a terrain situation where the agent has an obstacle on his left (west) and the human that he needs to go is 3 steps eastern than him[7]. If this terrain situation occurs in many observed runs, then the model may classify this situation such that the agent moves towards the human every time. However, in the actual executing terrain, the agent may have to face a similar state as above with only one difference; he may also have an obstacle on his right (east). In that case, the agent would try to move towards the human, thus staying in place because of the obstacle.

Thus, in accordance with the above example, the agent's model overfits when the training set has quite similar data instances, while the values of the test set are slightly different. As far as terrain size is concerned, similar situations are more likely to happen in relatively small terrains when the number of observed runs is relatively large. Ideally, the model should be trained with large terrains in order to assure that the data instances obtained are similar, yet not identical. Alternatively, when the terrain size is smaller, the agent may observe fewer runs. Conclusively, note that a straightforward way to avoid overfitting the model would be to construct a well-balanced training set and provide it to the agent.

[7] In ML terms, since the obstacle is irrelevant, it is denoted as *noise*.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In accordance with Chapter 1, the aim of this dissertation was to design an effective as well as efficient agent that can participate in a cooperative game without any prior coordination. The methodology described in Chapter 4 is spread along three axes: creating models of teammate policies, determining the teammates' chosen policies based on their actions, and constructing an effective, yet efficient response strategy concerning the team's desiderata.

Thus, the three aforementioned axes were combined to form an ad hoc strategy. The basic problems that each component of the strategy faces were identified as ML problems and were confronted using ML (and RL) techniques. The techniques used to construct the ad hoc agent's strategy are well-known ML techniques, widely used by the research community. In addition, the problems were solved in a novel way regarding their efficiency.

Although the design of the agent has certain domain-dependent properties, the basis of the implementation provides with clear methodology on recognizing and sufficiently modeling a domain to conform with it. A full working example of recognizing the specifics of a SAR domain was shown in Chapter 5. The SAR domain was also used as a testbed in order to support the claims given in the introduction. Chapter 6 provided specific experiments for all components of the ad hoc agent, testing his effectiveness and, when deemed necessary, his efficiency.

The results of these experiments were actually quite interesting. All components were found to be effective with relation to agents that play optimally. Concerning policy selection, the Naïve Bayes classifier provided a quite effective way of determining

the teammate agents' selected policies. Its performance was almost optimal despite being given more policies to get obscured. The strategy construction task was also especially effective. The merger of different answer policies considerably improved the agent's efficiency, without compromising his effectiveness.

Furthermore, since the ad hoc agent's teammate modeling component was also remarkably effective, the agent indeed had a complete approach on the ad hoc problem. Finally, tweaking the learning parameters provide with insightful ideas about the amount of learning required. In particular, both under-training and over-training the agent results in achieving worse results; there is a particular optimal value of the parameters that can be found experimentally.

7.2 Future Work

As shown in Chapter 3, the research directions regarding the ad hoc team setting are numerous. Since the methodology described covers several of these approaches, any extensions in these research lines may actually be applicable to it. As far as the methodology itself is concerned, future research is encouraged along all components of the strategy.

Since the design is generic enough, various ML algorithms may be tested. For example, the policy selection could use any classifier as long as it is incremental. In addition, the teammate modeling classifying task could also be done incrementally, thus constantly improving the models and minimizing the number of runs that are observed by the ad hoc agent. As far as the strategy construction task is concerned, it would be interesting to test whether model-based approaches perform satisfactorily as well. In addition, the merger of policies could be extended by using different merge functions or even perform incremental merger.

Apart from the above extensions, it would certainly be interesting to evaluate the ad hoc agent in a different testbed. Similar testbeds could even achieve different results, depending on the difficulty of the task involved. Furthermore, the ad hoc agent's performance in slightly more complex testbeds could be evaluated. In particular, the task could require further coordination among the agents. In such cases, a team Q-learning algorithm such as the ones of subsection 3.2.3 could replace the simple Q-learning of the ad hoc agent. The complexity issues that would arise could also be higher and thus they should be handled using other techniques (e.g. pruning parts of the state space or using an approximation).

Furthermore, concerning the core algorithm of the project, it would be interesting to extend the models presented so that they operate in real-time, without the need for observation and processing phases. In general, performing the teammate modeling task without any observed runs is very difficult. However, the number of runs required could be drastically reduced if the agent was able to update the models created by the classifier on-the-fly. In that case, the classifier should be able to work incrementally, or the data instances should be limited so that the classifier can create a new model efficiently. This could be accomplished by using a sliding window that keeps only few instances according to their recency.

Moreover, determining whether an observed policy is sufficiently described by an existing model or it should be individually modeled would be an interesting extension. Concerning the processing phase, the training of the Q-learning algorithm could also be done whilst playing, thus equally sharing any overhead. Conclusively, minimizing the observation and processing phases comes down to performing the aforementioned classifying and learning tasks on-the-fly. Concerning the time per timestep for the agent could be limited, it would be interesting to explore the problem of optimally allocating this time between the two tasks.

Finally, the extent to which the implementation presented is sufficient when all agents are ad hoc learners could also be put under consideration. It is proven in Chapter 6 that the agent is actually capable of achieving satisfactory results even when the other agents are themselves ad hoc learners. However, the task once again could be more difficult, possibly by enforcing certain characteristics (e.g. communication) to the fixed agents. Thus, the ad hoc agent's observing task would get more difficult since he would also have to receive certain messages using different metrics, such as confidence on his teammates.

Bibliography

[1] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, March 1986.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall series in artificial intelligence. Prentice Hall, 2 edition, December 2002.

[3] P. Stone, G. Kaminka, S. Kraus, and J. Rosenschein. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Proceedings of the 24th Conference on Artificial Intelligence*, July 2010.

[4] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada. RoboCup Rescue: search and rescue in large-scale disasters as a domain for autonomous agents research. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*, volume 6, pages 739–743. IEEE, 1999.

[5] H. Kitano. Robocup rescue: a grand challenge for multi-agent systems. In *Multi-Agent Systems, 2000. Proceedings. Fourth International Conference on*, pages 5 –12, 2000.

[6] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1 edition, March 1997.

[7] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[8] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica 31*, 31:249–268, 2007.

[9] Ross J. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[10] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, 1962.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(Oct):533–536+, 1986.

[12] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[13] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[14] John C. Platt. *Fast training of support vector machines using sequential minimal optimization*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[15] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: a survey. *J. Artif. Int. Res.*, 4(1):237–285, 1996.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[17] Ronald A. Howard. *Dynamic Programming and Markov Process (Technology Press Research Monographs)*. The MIT Press, first edition edition, June 1960.

[18] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(2):679–684, 1957.

[19] Andrew Barto and Michael Duff. Monte carlo matrix inversion and reinforcement learning. In *In Advances in Neural Information Processing Systems 6*, pages 687–694. Morgan Kaufmann, 1994.

[20] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3:9–44, 1988.

[21] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[22] Christopher J. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[23] Richard S. Sutton. Integrated modeling and control based on reinforcement learning and dynamic programming. In *Proceedings of the 1990 conference on Advances in neural information processing systems 3*, NIPS-3, pages 471–478, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[24] A. W. Moore and C. G. Atkeson. An investigation of memory-based function approximators for learning control. Technical report, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1992.

[25] Samuel Barrett, Peter Stone, and Sarit Kraus. Empirical evaluation of ad hoc teamwork in the pursuit domain. In *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, May 2011.

[26] Katie Genter, Noa Agmon, and Peter Stone. Role selection in ad hoc teamwork. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, June 2012.

[27] Feng Wu, Shlomo Zilberstein, and Xiaoping Chen. Online planning for ad hoc autonomous agent teams. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pages 439–445. AAAI Press, 2011.

[28] Stefano Albrecht and Subramanian Ramamoorthy. Comparative evaluation of mal algorithms in a diverse set of ad hoc team problems. In *Proceedings of The 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '12. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[29] Michael L. Littman. Value-function reinforcement learning in markov games. *Cognitive Systems Research*, 2(1):55–66, 2001.

[30] Martin Lauer and Martin A. Riedmiller. An Algorithm for Distributed Reinforcement Learning in Cooperative Multi-Agent Systems. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 535–542, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[31] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of the fifteenth national/tenth*

conference on Artificial intelligence/Innovative applications of artificial intelligence, AAAI '98/IAAI '98, pages 746–752, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[32] Junling Hu and Michael P. Wellman. Multiagent reinforcement learning: Theoretical framework and an algorithm. In *In Proceedings of the Fifteenth International Conference on Machine Learning*, pages 242–250, 1998.

[33] Junling Hu and Michael P. Wellman. Nash q-learning for general-sum stochastic games. *J. Mach. Learn. Res.*, 4:1039–1069, 2003.

[34] Samuel Barrett and Peter Stone. Ad hoc teamwork modeled with multi-armed bandits: An extension to discounted infinite rewards. In *Tenth International Conference on Autonomous Agents and Multiagent Systems - Adaptive Learning Agents Workshop (AAMAS - ALA)*, May 2011.

[35] Robin R. Murphy and J. Jake Sprouse. Strategies for searching an area with semi-autonomous mobile robots. In *In Proceedings of Robotics for Challenging Environments*, pages 15–21, 1996.

[36] Ryan Wegner and John Anderson. Balancing robotic teleoperation and autonomy for urban search and rescue environments. In Ahmed Tawfik and Scott Goodwin, editors, *Advances in Artificial Intelligence*, volume 3060 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24840-8_2.

[37] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[38] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[39] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

Appendix A

Simulator Specifics

A.1 Agent Strategy API

The simulator is implemented in a client-server architecture. Class `Simulator` implements the server of the system and actually is the first object created by `main`. Upon initialization, the `Simulator` spawns `Threads` of type `Agent`. Every `Agent` instance is actually a client that communicates with the server to receive information for the game and send his moves (see Figure A.4). Thus, the `Agent` instantiates the actual agent that must implement the `AgentStrategy` interface, shown in Figure A.1.

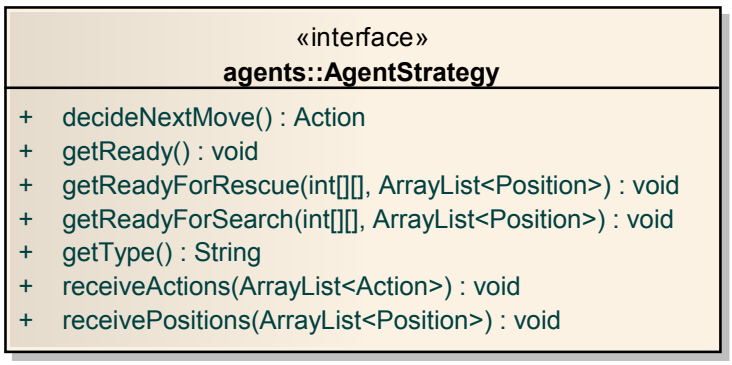


Figure A.1: The `AgentStrategy` interface, which is implemented by all active agents.

Any strategy implementing the `AgentStrategy` interface receives the field of play as well as the agents' positions at the start of each phase (`getReadyForSearch` and `getReadyForRescue`). After that, for each timeslot, the agent receives all agents' positions (`receivePositions`), decides for and returns his next action (`decideNextMove`) and receives all agents' actions (`receiveActions`) in case he wants to modify his strategy.

A.2 Agent Policy API

According to the problem specifics, the ad hoc agent should be able to handle strategies as black box policies, either they are his strategies or his teammates'. Thus, any policy should implement the `AgentPolicy` interface, shown in Figure A.2.

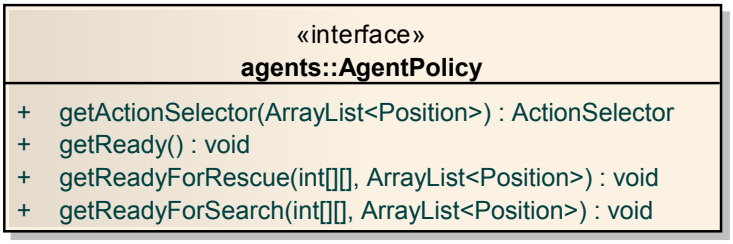


Figure A.2: The `AgentPolicy` interface, which is implemented by all agent policies.

The `AgentPolicy` interface is actually quite similar to the `AgentStrategy` interface. `AgentPolicy` receives similar data and returns an `ActionSelector` instance, containing a probability distribution over the possible actions.

A.3 Simulator Configuration

All variables have default values in case they are not provided. In addition, their values can be given as arguments from the command line. For any parameter, the respective argument value of the command line has higher precedence than its value in the configuration file, which in turn has higher precedence than its default value.

```
repetitions = 10
xsize = 12
ysize = 20
agents = 1 * AstarClose + 1 * AstarFar + 1 * S3 AdHocModelerWithRL
experiment = true
timestepDelay = 0.1325
enableGUI = true
loadTerrain = false
filename = field.txt
```

Figure A.3: Sample configuration file of the simulator.

`agents` are given in the form `num * 1 [Srun] name` where the optional argument `run` denotes the run at which the agent enters the terrain. `xsize` and `ysize` denote the size of the terrain in case that no terrain from file is given. If a terrain

is given (`loadTerrain = true`), then the aforementioned parameters are overriden. `enableGUI` determines whether the GUI is enabled. Finally, in experiment mode (`experiment = true`), the simulator runs without GUI and no timestep delay and outputs only the average and the standard deviation for the agents' performance.

A.4 Class Diagrams

The client-server architecture of the simulator is shown in Figure A.4. The `Simulator` (server) provides terrain information to the `Agent` (client), and the latter sends back an action which is determined by the `AgentStrategy` interface.

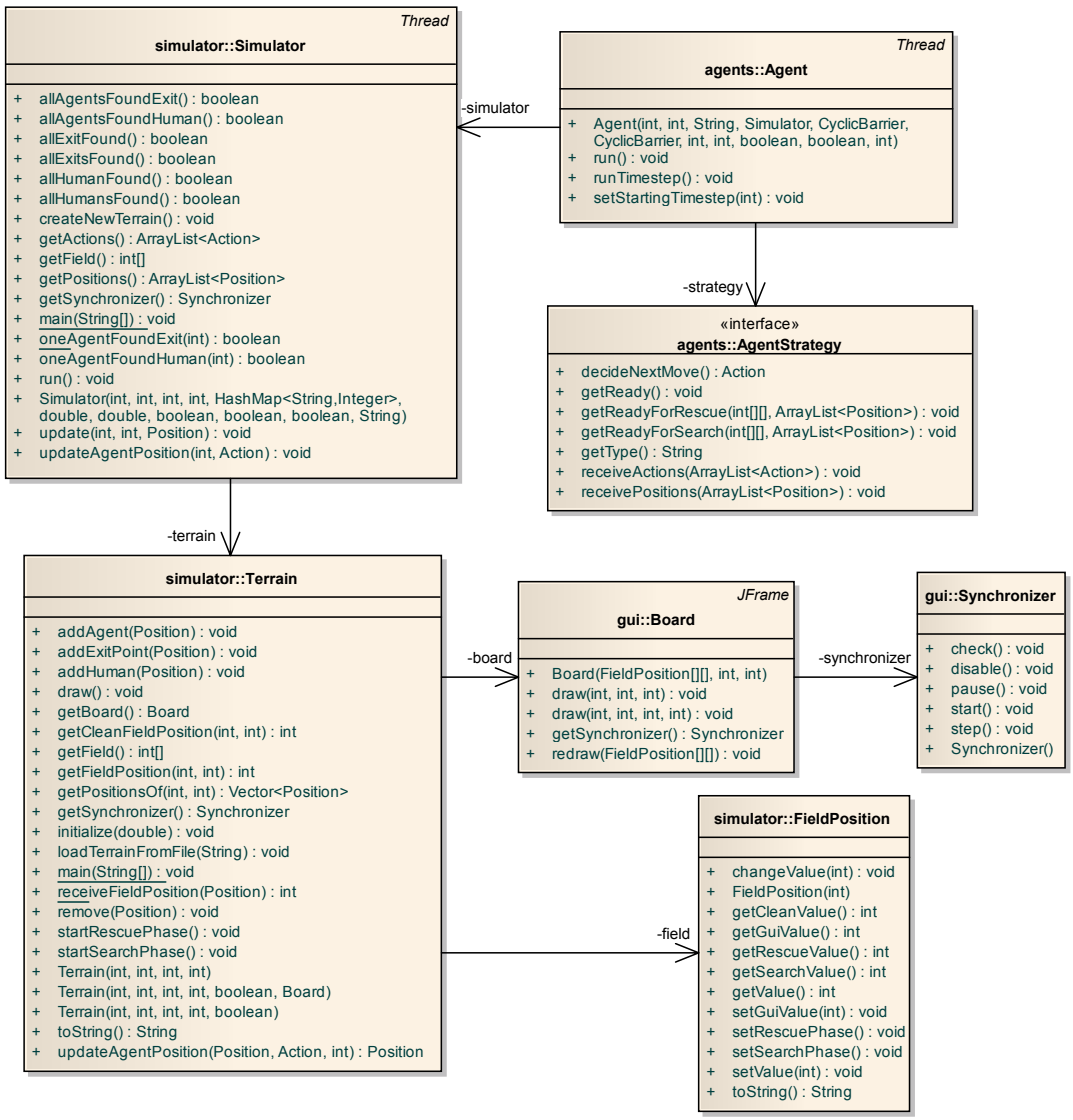


Figure A.4: Class diagram of the simulator. Class `Simulator` (server) sends messages back and forth with instances of type `Agent` (client).

A high-level overview of the various interfaces is shiown in Figure A.5. As already mentioned, the `AgentStrategy` interface has to be implemented by any agent that participates in a run. In addition, the `AgentPolicy` interface is implemented by any policy, i.e. by any model that can be used as a black box receiving system state and outputting an action. Finally, the `BayesPolicy` interface is implemented by policies that are provided to the ad hoc agent so that he can use the Naïve Bayes classifier in order to select among them.

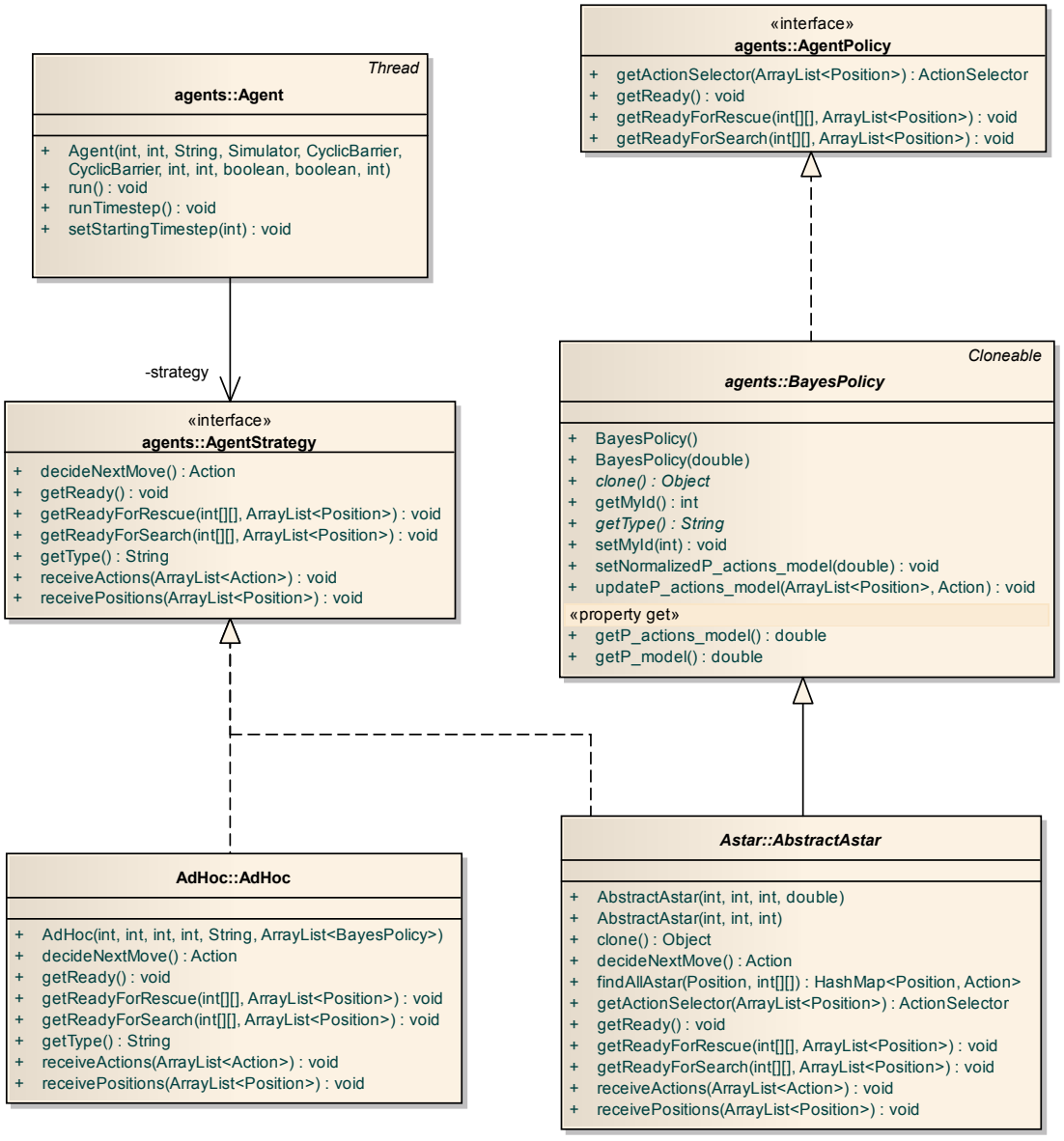


Figure A.5: Class diagram of the agent interfaces. Any agent strategy (such as `AdHoc` or `AbstractAstar`) implements the `AgentStrategy` interface, while any agent policy (such as `AbstractAstar`) implements the `AgentPolicy` interface.

Finally, Figure A.6 illustrates the basic architecture of an ad hoc team agent. The agent keeps one `TeammatePolicy` instance for each of his teammates and one `MyPolicy` instance for himself. Thus, each `TeammatePolicy` instance applies the Naïve Bayes classifier to determine which model is a better fit for the respective teammate. Upon determining the teammates' models, the ad hoc agent selects one of his own models, which are kept in the `MyPolicy` instance.

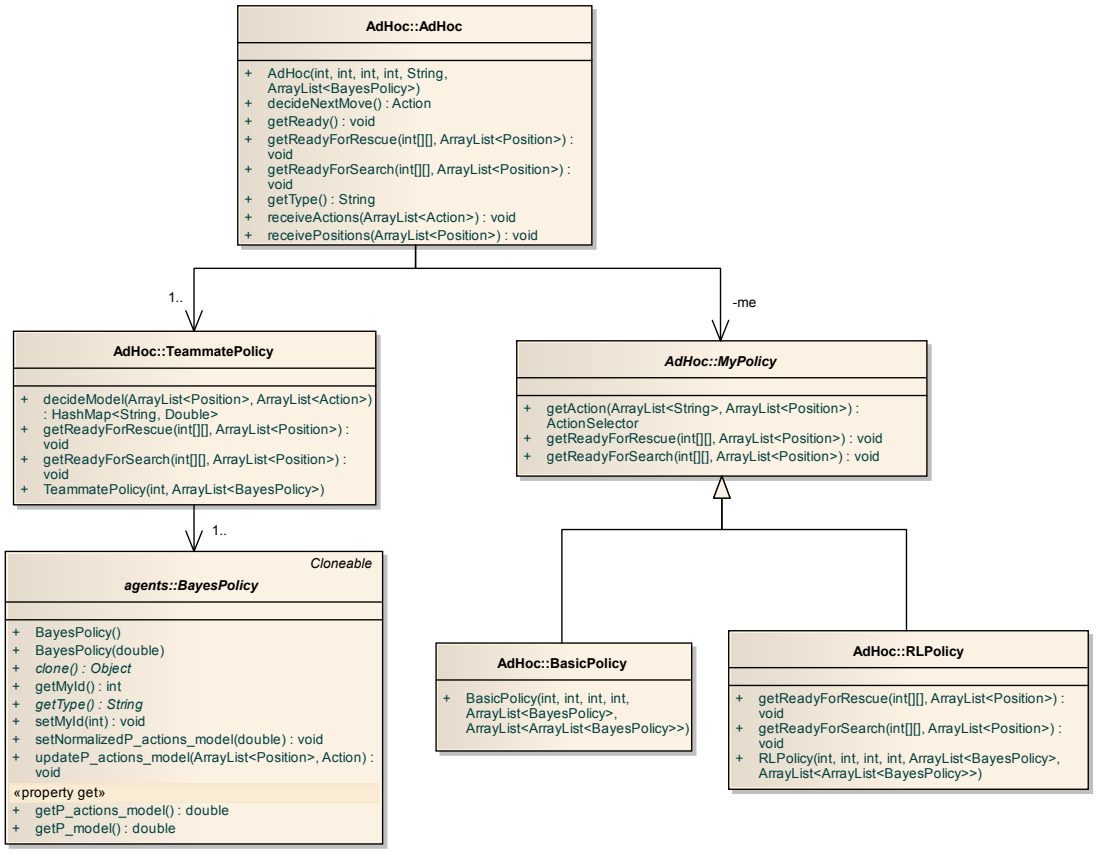


Figure A.6: Class diagram of the ad hoc agent. The agent creates a `TeammatePolicy` instance for any of his teammates and a `MyPolicy` instance for himself.

Conclusively, note that the above class diagrams do not exhaustively cover neither the simulator nor the agents' functionality. However, they provide with an interesting high level view of the system, which may be useful as a model for future extensions.

Appendix B

A* Agent Specifics

The pseudocode of the A* algorithm for the SAR terrain is shown in Figure B.1.

```
Function Astar(Cell startingCell, Cell goalCell)
    Variables:
        LIST OPEN_LIST, CLOSED_LIST, FINAL_LIST
    OPEN_LIST.add(startingCell)
    while ((goalCell not in CLOSED_LIST) and (OPEN_LIST not empty))
        currentCell = cell with minimum F from the OPEN_LIST
        OPEN_LIST.remove(currentCell)
        CLOSED_LIST.add(currentCell)
        for (neighborCell : neighbors(currentCell))
            if (neighborCell not in CLOSED_LIST)
                if (neighborCell not in OPEN_LIST)
                    OPEN_LIST.add(neighborCell)
                    neighborCell.setParent(currentCell)
                    neighborCell.calculateFGH(goalCell)
                else if (neighborCell.G > currentCell.G + 1)
                    neighborCell.setParent(currentCell)
                    neighborCell.calculateFGH(goalCell)
    cell = goalCell
    while (cell != startingPosition)
        FINAL_LIST.add(cell)
        cell = cell.getParent()
    return FINAL_LIST.reverse()
```

Figure B.1: Pseudocode of the A* algorithm for the SAR terrain.

The class `Cell` represents a position on the terrain and is defined in Figure B.2.

```
Class Cell
    Variables:
        numeric: F, G, H
        Cell: parent
    Function setParent(Cell parentCell)
        parent = parentCell
    Function calculateFGH(Cell goalCell)
        G = parent.G + 1
        H = distance of this cell from goalCell
        F = G + H
```

Figure B.2: Pseudocode of the object `Cell` that represents a position of the terrain.

For each cell, the heuristic value F is computed as the sum of the (Manhattan) distance of the cell from the starting cell G and the (Manhattan) distance of the cell from the goal cell H. The A* function receives two cells, the agent's position and the goal cell. The `OPEN_LIST` contains the cells that are not yet explored, whereas the `CLOSED_LIST` contains the cells that have already been explored. For each iteration, the algorithm adds the cell with the minimum F value to the `CLOSED_LIST`, named the *current* cell. Then it iterates through its neigboring cells, setting their parent attribute to the current cell, calculating their heuristics values, and adding them to the `OPEN_LIST`. In addition, if a cell is found to be already checked (hence its heuristic values are already computed), it is still checked to determine if selecting the current cell as its new parent would result in better heuristic values. Thus, it is certain that the algorithm finds an optimal path. The algorithm iterates until the goal cell is added to the `CLOSED_LIST`. Finally, the algorithm starts from the goal cell and iterates until it finds the starting cell, by selecting each time the parent of the cell that is checked. The cell iteration is saved in a list, thus the resulting path is the reverse of the list.

Appendix C

Experiment Results

C.1 Known Teammate Models

C.1.1 Using Modeled Policy

Tables C.1, C.2, and C.3 contain the mean number of timesteps for 10 repetitions of 10 different terrains, concerning terrain sizes 6×10, 9×15, and 12×20 respectively. There are 4 agents per team and 4 possible policies. Team A is optimal, whereas teams B and C each have an agent that plays optimally 75% and 65% of the moves respectively. Team D has 1 ad hoc agent that determines which known policy to use in accordance with his teammates' policies, while team E has 2 ad hoc agents.

Table C.1: Mean number of timesteps testing the policy selection component of the ad hoc agent (teams D and E with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 6×10.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team D	Team E
Terrains	1	8.0±0.0	10.8±2.2	14.2±9.9	9.6±0.8	9.0±1.3
	2	12.0±0.0	14.0±1.8	16.4±4.3	12.8±1.0	14.8±3.0
	3	8.0±0.0	9.9±1.6	12.4±5.1	8.0±0.0	9.6±1.5
	4	18.0±0.0	20.2±2.3	20.6±4.7	18.0±0.0	19.2±2.6
	5	18.0±0.0	22.9±4.1	37.3±20.3	18.7±0.5	21.6±5.0
	6	10.0±0.0	20.3±9.4	20.2±7.3	11.2±1.0	12.2±2.1
	7	24.0±0.0	28.4±4.7	30.9±6.3	24.0±0.0	25.4±1.6
	8	12.0±0.0	16.0±3.8	26.9±14.8	13.0±1.0	15.4±3.5
	9	22.0±0.0	23.0±1.3	22.6±1.3	22.0±0.0	22.7±1.4
	10	14.0±0.0	14.3±0.9	14.2±0.6	15.2±1.0	15.2±1.6
Sum		**146.0±0.0**	**179.8±32.0**	**215.7±74.6**	**152.5±5.2**	**165.1±23.5**

Table C.2: Mean number of timesteps testing the policy selection component of the ad hoc agent (teams D and E with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 9×15.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team D	Team E
Terrains	1	24.0±0.0	35.0±6.1	50.0±16.3	24.4±0.8	26.2±2.1
	2	28.0±0.0	36.2±8.0	51.2±16.8	28.6±0.9	29.8±1.4
	3	22.0±0.0	28.4±3.9	38.2±7.8	22.0±0.0	22.6±1.3
	4	34.0±0.0	42.4±9.5	48.2±12.0	34.6±0.5	35.3±1.5
	5	16.0±0.0	19.4±7.3	19.5±2.5	16.0±0.0	17.4±1.6
	6	16.0±0.0	23.4±10.7	26.6±9.6	17.0±1.0	17.4±1.8
	7	32.0±0.0	40.0±5.9	52.0±10.3	32.5±0.5	34.2±2.5
	8	18.0±0.0	20.9±3.4	23.8±6.8	18.5±0.5	20.8±3.1
	9	20.0±0.0	32.4±8.2	39.5±23.6	20.6±0.9	22.8±2.9
	10	22.0±0.0	23.7±2.7	26.4±6.7	22.0±0.0	23.0±1.3
Sum		**232.0±0.0**	**301.8±65.7**	**375.4±112.3**	**236.2±5.1**	**249.5±19.4**

Table C.3: Mean number of timesteps testing the policy selection component of the ad hoc agent (teams D and E with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 12×20.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team D	Team E
Terrains	1	32.0±0.0	40.3±9.2	41.4±8.7	33.2±1.0	34.9±4.0
	2	16.0±0.0	20.6±1.9	31.7±17.7	16.5±0.5	18.4±5.0
	3	44.0±0.0	56.3±7.8	64.5±11.1	45.6±0.8	46.0±2.0
	4	48.0±0.0	61.6±8.3	103.2±23.5	48.6±0.5	48.9±1.2
	5	16.0±0.0	18.3±2.0	19.3±4.0	16.0±0.0	16.1±0.3
	6	40.0±0.0	46.8±8.1	55.8±13.4	40.0±0.0	41.2±2.0
	7	34.0±0.0	49.6±8.1	69.9±11.3	36.0±2.0	35.4±2.2
	8	36.0±0.0	41.8±5.7	51.4±8.9	36.0±0.0	36.4±0.8
	9	24.0±0.0	31.2±4.5	41.9±9.7	25.4±0.9	26.6±2.8
	10	30.0±0.0	35.0±5.3	38.9±11.9	30.4±0.5	31.7±2.0
Sum		**320.0±0.0**	**401.5±60.9**	**518.0±120.2**	**327.7±6.2**	**335.6±22.3**

C.1.2 Constructing Strategy

C.1.2.1 Efficiency Tests

The mean time per timestep in milliseconds for 3 repetitions on 3 terrains is shown in Tables C.4, C.5, and C.6 for the two strategy construction policies, concerning terrain sizes 6×10, 9×15, and 12×20 respectively. There are 2 possible policies. The ad hoc agent of team F constructs a policy for each possible combination, whereas the agent of team G constructs merger policy based on the 2 known models.

Table C.4: Mean time per timestep testing the efficiency of the strategy construction component that constructs an answer policy for each combination of teammate policies (team F) versus the one that merges the distinct answer policies, in terrain size 6×10.

		Mean Time per Timestep for Teams of							
		2 agents		4 agents		6 agents		8 agents	
		Team F	Team G	Team F	Team G	Team F	Team G	Team F	Team G
Terrains	1	50.0±0.0	60.3±7.6	172.3±16.1	64.4±6.3	736.5±85.6	89.9±9.4	4047.0±429.7	175.6±21.1
	2	43.3±2.5	54.3±4.2	165.6±14.9	54.6±1.8	903.5±81.0	72.6±1.0	3680.5±317.4	159.5±15.4
	3	39.0±1.7	58.3±4.8	184.5±10.8	63.8±5.1	904.0±5.0	81.2±5.4	3730.7±17.1	131.5±1.1
Avg		**44.1±1.4**	**57.6±5.5**	**174.1±13.9**	**61.0±4.4**	**848.0±57.2**	**81.2±5.3**	**3819.4±254.8**	**155.5±12.6**

Table C.5: Mean time per timestep testing the efficiency of the strategy construction component that constructs an answer policy for each combination of teammate policies (team F) versus the one that merges the distinct answer policies, in terrain size 9×15.

		Mean Time per Timestep for Teams of							
		2 agents		4 agents		6 agents		8 agents	
		Team F	Team G	Team F	Team G	Team F	Team G	Team F	Team G
Terrains	1	51.7±0.8	69.6±3.1	200.6±3.0	71.8±1.5	977.5±123.5	93.3±1.8	4800.5±9.1	178.6±2.0
	2	59.8±1.1	82.7±2.5	251.7±0.8	90.9±0.7	1437.9±7.9	118.3±1.4	5899.9±9.9	235.3±2.9
	3	53.0±3.4	72.1±5.9	235.0±13.2	84.6±3.2	1285.7±58.5	106.5±4.9	5419.7±245.8	192.9±16.0
Avg		**54.8±1.8**	**74.8±3.8**	**229.1±5.7**	**82.4±1.8**	**1233.7±63.3**	**106.0±2.7**	**5373.4±88.3**	**202.3±6.9**

Table C.6: Mean time per timestep testing the efficiency of the strategy construction component that constructs an answer policy for each combination of teammate policies (team F) versus the one that merges the distinct answer policies, in terrain size 12×20.

		Mean Time per Timestep for Teams of							
		2 agents		4 agents		6 agents		8 agents	
		Team F	Team G	Team F	Team G	Team F	Team G	Team F	Team G
Terrains	1	68.8±1.0	91.5±5.5	300.7±4.7	105.4±2.3	1759.8±7.5	126.8±2.4	7867.9±515.4	210.7±1.4
	2	63.4±0.9	83.1±0.5	273.3±6.7	93.8±1.4	1509.1±32.5	108.8±1.0	6840.8±283.1	180.2±3.3
	3	79.6±0.7	105.1±3.3	450.2±13.9	120.9±0.6	1989.4±8.0	145.1±3.2	9425.7±531.5	250.5±7.6
Avg		**70.6±0.9**	**93.2±3.1**	**341.4±8.5**	**106.7±1.4**	**1752.7±16.0**	**126.9±2.2**	**8044.8±443.4**	**213.8±4.1**

C.1.2.2 Effectiveness Tests

Tables C.7, C.8, and C.9 contain the mean number of timesteps for 10 repetitions of 10 terrains for terrain sizes 6×10, 9×15, and 12×20 respectively. There are 4 agents per team and 2 possible policies. Team A is optimal, whereas team D has an ad hoc agent that plays a known policy. Teams F and G each have an ad hoc agent that constructs a policy for each possible combination of teammates' policies or constructs a merger policy respectively. Team H has 2 agents that use merger strategy construction.

Table C.7: Mean number of timesteps testing the merger strategy construction component (teams G and H with 1 and 2 ad hoc agents respectively) versus its naïve counterpart (team F), an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 6×10.

		Mean Number of Timesteps for Teams				
		Team A	Team D	Team F	Team G	Team H
Terrains	1	8.0±0.0	9.2±1.0	8.8±1.0	9.0±1.0	10.0±3.2
	2	12.0±0.0	13.0±1.0	13.6±0.8	13.6±0.8	14.7±2.1
	3	8.0±0.0	8.0±0.0	8.0±0.0	8.0±0.0	10.0±3.6
	4	18.0±0.0	18.0±0.0	18.0±0.0	18.0±0.0	18.8±0.9
	5	18.0±0.0	18.4±0.5	19.0±1.3	18.7±0.8	20.7±4.0
	6	10.0±0.0	11.2±1.0	11.2±1.0	11.0±1.0	11.1±1.3
	7	24.0±0.0	24.0±0.0	24.0±0.0	24.0±0.0	24.8±1.3
	8	12.0±0.0	13.0±1.0	13.2±1.0	13.2±1.0	13.8±1.4
	9	22.0±0.0	22.0±0.0	22.0±0.0	22.0±0.0	22.8±1.0
	10	14.0±0.0	15.0±1.0	14.4±0.8	15.8±1.7	16.4±2.7
Sum		**146.0±0.0**	**151.8±5.4**	**152.2±5.8**	**153.3±6.2**	**163.1±21.4**

Table C.8: Mean number of timesteps testing the merger strategy construction component (teams G and H with 1 and 2 ad hoc agents respectively) versus its naïve counterpart (team F), an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 9×15.

		Mean Number of Timesteps for Teams				
		Team A	Team D	Team F	Team G	Team H
Terrains	1	24.0±0.0	24.6±0.9	24.2±0.6	24.2±0.6	25.0±1.0
	2	28.0±0.0	28.6±0.9	31.2±2.2	31.2±2.4	32.8±2.9
	3	22.0±0.0	22.0±0.0	22.4±0.8	22.6±0.9	23.2±1.3
	4	34.0±0.0	34.3±0.5	37.2±2.0	38.0±4.0	38.2±2.7
	5	16.0±0.0	16.0±0.0	16.0±0.0	16.0±0.0	17.1±0.9
	6	16.0±0.0	17.2±1.0	17.8±0.6	17.6±0.8	18.0±2.2
	7	32.0±0.0	32.5±0.5	34.3±1.3	33.4±1.2	35.9±2.0
	8	18.0±0.0	18.6±0.5	19.8±1.3	20.5±1.0	21.6±2.3
	9	20.0±0.0	20.8±1.0	20.4±0.8	21.4±1.3	23.2±1.6
	10	22.0±0.0	22.0±0.0	22.0±0.0	22.0±0.0	23.5±1.5
Sum		**232.0±0.0**	**236.6±5.2**	**245.3±9.7**	**246.9±12.2**	**258.5±18.5**

Table C.9: Mean number of timesteps testing the merger strategy construction component (teams G and H with 1 and 2 ad hoc agents respectively) versus its naïve counterpart (team F), an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 12×20.

		Mean Number of Timesteps for Teams				
		Team A	Team D	Team F	Team G	Team H
Terrains	1	32.0±0.0	32.6±0.9	32.2±0.6	32.2±0.6	34.0±1.5
	2	16.0±0.0	16.6±0.5	17.2±0.9	16.9±0.8	18.8±4.2
	3	44.0±0.0	44.8±1.0	45.0±1.8	45.2±1.3	46.2±2.1
	4	48.0±0.0	48.7±0.5	48.8±0.9	49.1±0.8	49.8±1.4
	5	16.0±0.0	16.0±0.0	16.0±0.0	16.0±0.0	17.6±3.5
	6	40.0±0.0	40.0±0.0	40.2±0.6	40.2±0.6	40.2±0.6
	7	34.0±0.0	36.0±2.0	34.2±0.6	34.0±0.0	36.0±2.0
	8	36.0±0.0	36.0±0.0	36.2±0.6	36.2±0.6	37.4±3.6
	9	24.0±0.0	25.6±0.8	24.4±0.8	24.6±0.9	27.6±5.2
	10	30.0±0.0	30.6±0.5	33.7±1.3	33.7±1.6	34.3±2.2
Sum		**320.0±0.0**	**326.9±6.1**	**327.9±8.1**	**328.1±7.3**	**341.9±26.3**

C.2 Unknown Teammate Models

Tables C.10, C.11, and C.12 present the number of timesteps for 10 repetitions of 10 terrains of sizes 6×10, 9×15, and 12×20 respectively. There are 4 agents per team and 2 possible policies. As before, team A is optimal, whereas teams B and C each have an agent that is 75% and 65% optimal respectively. Team D has one ad hoc agent that fully models his teammates, while team E has two ad hoc agents.

Table C.10: Mean number of timesteps testing the teammate modeling component of the ad hoc agent (teams I and J with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 6×10.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team I	Team J
Terrains	1	8.0±0.0	11.8±3.7	13.7±6.8	9.2±1.0	12.0±5.8
	2	12.0±0.0	13.6±1.5	19.2±7.0	12.6±0.9	14.2±3.1
	3	8.0±0.0	9.7±2.2	12.4±5.1	8.3±0.6	9.0±3.0
	4	18.0±0.0	18.7±1.1	22.1±5.8	18.0±0.0	21.0±3.7
	5	18.0±0.0	22.7±2.9	29.3±11.1	18.0±0.0	19.7±1.7
	6	10.0±0.0	13.4±3.6	18.0±7.6	10.8±1.0	11.4±1.8
	7	24.0±0.0	26.6±3.7	34.2±6.6	25.5±0.5	30.4±7.4
	8	12.0±0.0	14.0±1.8	22.4±6.2	14.0±0.9	14.5±3.2
	9	22.0±0.0	23.8±2.9	25.5±3.1	22.0±0.0	26.3±3.3
	10	14.0±0.0	14.8±1.0	16.4±5.1	17.0±0.0	22.1±6.9
Sum		**146.0±0.0**	**169.1±24.2**	**213.2±64.5**	**155.4±4.9**	**180.6±40.0**

Table C.11: Mean number of timesteps testing the teammate modeling component of the ad hoc agent (teams I and J with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 9×15.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team I	Team J
Terrains	1	24.0±0.0	38.6±9.8	44.7±24.6	22.0±0.0	25.1±2.3
	2	28.0±0.0	37.0±8.0	42.0±10.4	34.9±0.3	34.3±4.3
	3	22.0±0.0	32.9±9.2	38.9±12.8	21.4±0.5	34.7±20.9
	4	34.0±0.0	40.3±4.0	52.8±12.8	34.0±0.0	34.4±0.8
	5	16.0±0.0	17.0±1.3	20.6±2.7	16.0±0.0	18.2±3.3
	6	16.0±0.0	20.0±5.1	29.2±8.2	19.6±2.2	18.4±2.3
	7	32.0±0.0	37.9±2.6	51.6±12.6	35.0±1.0	37.2±2.1
	8	18.0±0.0	19.6±2.1	21.2±5.0	18.2±0.6	20.1±4.5
	9	20.0±0.0	30.9±7.5	38.6±11.4	24.2±0.6	21.8±2.3
	10	22.0±0.0	24.0±1.5	30.8±12.4	30.9±1.6	25.8±2.4
Sum		**232.0±0.0**	**298.2±51.0**	**370.4±113.0**	**256.2±6.7**	**270.0±45.3**

Table C.12: Mean number of timesteps testing the teammate modeling component of the ad hoc agent (teams I and J with 1 and 2 ad hoc agents respectively) versus an optimal (team A) and two suboptimal teams (teams B and C), in terrain size 12×20.

		Mean Number of Timesteps for Teams				
		Team A	Team B	Team C	Team I	Team J
Terrains	1	32.0±0.0	39.7±4.2	42.6±9.0	32.0±0.0	37.4±3.2
	2	16.0±0.0	20.0±4.9	33.0±11.3	18.7±1.6	16.6±0.9
	3	44.0±0.0	58.1±6.2	73.3±13.9	46.6±1.8	48.9±2.6
	4	48.0±0.0	64.6±7.7	84.5±22.5	48.0±0.0	50.3±3.7
	5	16.0±0.0	17.0±1.0	20.4±4.0	17.8±1.7	23.0±8.3
	6	40.0±0.0	48.8±10.2	61.8±12.0	43.6±2.9	42.2±4.7
	7	34.0±0.0	48.8±14.0	72.1±19.4	34.4±0.8	35.6±2.3
	8	36.0±0.0	45.0±8.3	52.4±18.7	36.8±1.0	38.8±4.7
	9	24.0±0.0	32.7±6.0	37.3±7.0	24.0±0.0	26.2±2.9
	10	30.0±0.0	37.4±4.4	43.2±10.8	32.0±0.0	35.3±4.5
Sum		**320.0±0.0**	**412.1±66.9**	**520.6±128.6**	**333.9±9.7**	**354.3±37.8**

C.3 Learning Sensitivity Analysis

Team $I_{K,L}$ has an ad hoc agent that models his teammates upon observing K runs and constructs answer policies by simulating L runs. Concerning 2 agents with 2 teammate models, the results for 10 repetitions over 10 terrains are shown in Tables C.13, C.14, and C.15 for terrains sizes 6×10, 9×15, and 12×20 respectively.

Table C.13: Mean number of timesteps testing the sensitivity of the ad hoc agent's learning parameters. $I_{K,L}$ contains an ad hoc agent that is given K observed runs to model his teammates and L simulated runs to construct his strategy, in terrain size 6×10.

	Mean Number of Timesteps for Teams							
Terrains	Team $I_{3,300}$	Team $I_{3,500}$	Team $I_{3,700}$	Team $I_{5,300}$	Team $I_{5,500}$	Team $I_{5,700}$	Team $I_{7,300}$	Team $I_{7,500}$	Team $I_{7,700}$
1	9.8±1.0	10.4±1.0	10.2±1.3	10.6±0.8	11.4±0.7	9.8±0.9	12.5±0.8	10.6±1.5	9.4±1.0
2	15.4±1.5	15.2±1.0	19.3±1.2	14.2±1.9	14.6±1.4	17.1±0.7	15.8±0.8	15.4±0.9	19.0±0.4
3	9.3±0.8	10.6±0.7	9.7±0.3	8.6±0.7	9.8±0.3	9.4±0.7	8.6±0.7	9.0±0.0	9.9±1.0
4	21.0±0.0	19.7±0.8	19.8±0.6	20.3±0.8	19.5±0.9	18.8±0.4	20.7±0.7	19.8±0.3	19.4±0.8
5	21.3±0.8	20.9±0.1	19.4±1.1	20.8±0.6	19.8±0.6	21.0±0.9	19.6±0.7	18.8±0.6	20.2±0.6
6	11.0±0.0	12.9±0.8	12.4±0.9	12.0±0.9	12.4±1.6	12.4±0.8	11.6±0.8	14.6±1.0	13.2±0.8
7	25.3±1.1	26.8±0.7	25.5±0.7	27.0±0.0	25.5±0.9	25.0±0.0	25.8±0.3	25.0±0.0	26.3±0.3
8	14.6±1.4	14.0±0.8	15.1±2.2	14.4±1.0	15.0±1.7	14.8±1.7	17.2±1.2	14.8±0.9	17.4±1.0
9	23.0±0.0	24.4±0.3	24.0±0.0	23.3±0.5	26.3±0.5	29.3±0.6	22.6±0.8	23.8±0.7	23.2±0.7
10	15.8±0.4	20.7±0.3	21.5±3.7	17.6±0.5	19.0±0.5	17.4±0.5	19.2±2.0	17.0±0.0	18.6±1.2
Sum	**166.5±6.9**	**175.6±6.4**	**176.9±12.0**	**168.8±7.7**	**173.3±9.0**	**175.0±7.2**	**173.6±8.8**	**168.8±6.0**	**176.6±7.7**

Table C.14: Mean number of timesteps testing the sensitivity of the ad hoc agent's learning parameters. $I_{K,L}$ contains an ad hoc agent that is given K observed runs to model his teammates and L simulated runs to construct his strategy, in terrain size 9×15.

		Mean Number of Timesteps for Teams								
		Team $I_{3,300}$	Team $I_{3,500}$	Team $I_{3,700}$	Team $I_{5,300}$	Team $I_{5,500}$	Team $I_{5,700}$	Team $I_{7,300}$	Team $I_{7,500}$	Team $I_{7,700}$
Terrains	1	23.4±0.7	33.2±0.2	24.8±0.7	23.8±0.5	24.0±0.0	24.0±0.0	26.6±0.4	27.2±1.0	24.4±3.9
	2	34.2±2.7	34.4±0.6	40.8±5.4	31.2±4.8	35.8±1.9	43.4±1.0	29.4±2.2	35.2±3.2	43.9±2.5
	3	22.4±0.8	23.6±0.8	24.2±1.2	24.2±0.7	22.8±1.3	23.6±1.0	21.3±0.7	24.2±0.6	21.6±0.6
	4	39.4±1.3	37.2±2.4	38.0±2.0	39.2±1.3	37.0±0.6	38.9±0.7	38.2±1.1	37.0±2.0	36.6±1.7
	5	18.2±0.7	18.2±0.4	18.6±0.6	17.2±0.8	19.2±0.4	19.9±0.1	18.8±0.3	17.6±0.5	18.0±0.0
	6	18.4±0.9	19.4±1.5	17.4±0.9	18.8±0.9	18.0±0.8	19.4±0.9	21.4±1.0	18.6±1.0	19.4±1.0
	7	37.7±1.3	35.0±1.5	34.0±0.0	33.8±0.3	33.4±1.7	35.8±0.6	34.6±1.0	33.8±0.6	37.4±0.8
	8	20.6±0.5	19.0±1.6	20.8±1.0	22.6±0.6	19.8±0.6	21.4±0.7	22.1±1.0	21.2±1.4	20.8±0.4
	9	23.4±1.2	21.2±1.3	27.2±1.3	23.8±1.4	24.0±1.0	24.6±2.0	23.6±1.3	22.8±1.0	22.2±1.6
	10	25.5±0.7	32.4±0.4	25.5±0.5	25.6±1.4	23.0±0.0	25.3±0.7	26.1±0.7	24.3±0.4	26.4±1.1
Sum		**263.2±10.8**	**273.6±10.6**	**271.3±13.6**	**260.2±12.7**	**257.0±8.3**	**276.3±7.6**	**262.1±9.5**	**261.9±11.7**	**270.7±13.6**

Table C.15: Mean number of timesteps testing the sensitivity of the ad hoc agent's learning parameters. $I_{K,L}$ contains an ad hoc agent that is given K observed runs to model his teammates and L simulated runs to construct his strategy, in terrain size 12×20.

		Mean Number of Timesteps for Teams							
		Team $I_{3,300}$	Team $I_{3,500}$	Team $I_{3,700}$	Team $I_{5,300}$	Team $I_{5,500}$	Team $I_{5,700}$	Team $I_{7,300}$	Team $I_{7,500}$	Team $I_{7,700}$
Terrains	1	34.2±2.1	39.6±0.4	38.8±2.1	39.0±0.0	35.3±0.6	34.2±0.9	36.2±0.2	35.0±0.0	36.2±0.6
	2	18.8±0.4	18.8±0.7	20.8±1.7	20.8±0.8	19.0±0.9	21.0±0.0	21.2±0.9	21.4±0.8	18.6±0.5
	3	52.7±5.5	49.0±2.8	50.0±0.0	53.6±1.4	52.6±0.9	47.8±1.2	51.8±3.7	45.6±1.3	45.2±0.8
	4	55.1±2.6	50.4±0.6	52.0±0.0	53.6±2.3	52.2±0.8	49.4±0.7	50.4±0.9	53.1±0.7	50.6±1.5
	5	25.0±0.0	19.1±1.3	20.8±0.8	18.6±1.7	20.0±0.9	18.8±0.3	21.6±0.5	17.8±0.6	19.6±1.2
	6	43.7±2.4	41.8±0.7	43.0±0.0	44.0±0.0	45.0±1.3	45.2±2.2	50.2±2.5	46.2±1.4	45.0±0.9
	7	39.0±0.0	36.8±0.2	41.2±0.8	36.2±1.4	37.4±0.8	35.6±0.6	36.0±0.0	37.6±0.6	37.8±1.0
	8	44.2±0.8	39.0±3.8	39.8±0.9	41.6±0.8	41.6±0.7	42.0±0.0	40.0±1.0	43.2±0.9	40.2±1.3
	9	26.2±0.8	31.8±0.4	25.4±0.7	28.4±0.6	27.8±2.4	26.6±1.6	31.7±0.5	32.0±0.8	26.4±1.7
	10	34.0±1.1	35.6±0.3	36.2±0.1	36.6±1.0	35.4±0.0	36.6±0.7	34.6±1.2	36.4±0.6	35.0±1.8
	Sum	**372.9±15.7**	**361.9±11.1**	**368.0±7.1**	**372.4±10.1**	**366.3±9.2**	**357.2±8.2**	**373.7±11.3**	**368.3±7.7**	**354.6±11.3**