Computational Agents That Learn About Agents: Algorithms for Their Design and a Predictive Theory of Their Behavior

by

José M. Vidal

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science & Engineering)
in The University of Michigan
1998

Doctoral Committee:

Associate Professor Edmund H. Durfee, Chair
Professor Robert Axelrod
Associate Professor William P. Birmingham
Associate Professor John E. Laird
Associate Professor Michael P. Wellman

ABSTRACT

Computational Agents That Learn About Agents: Algorithms for Their Design and a Predictive Theory of Their Behavior

by
José M. Vidal

Chair: Edmund H. Durfee

In this thesis, we show how to build agents that learn about agents in Multi-Agent Systems (MASs) composed of learning agents. We give a framework for describing a MAS and its agents, along with their behaviors and the errors in these behaviors. The framework is supplemented with notation that captures the knowledge of agents with recursive models of other agents. Working under the assumption that agents can generate recursive models of other agents, we design an algorithm (LR-RMM) that works by calculating an agent's expected gain in order to tell the agent when it should stop thinking about other agents and take action. We implement LR-RMM and verify its results in the Pursuit Task. We then consider agents that must learn models of other agents via observation. We extend our framework (CLRI) to capture the agents' learning abilities, regardless of the particular machine learning algorithms used by the agents, and the degree to which the agents impact each others' behaviors. CLRI predicts the correctness of the expected behavior of any learning agent in a MAS. Since agents impact each other, an agent's desired behavior might change due to changes in the other agents' behaviors, as caused by their learning. The CLRI framework takes all these changes into account when predicting the correctness of an agent's expected behavior. We confirm the CLRI predictions with experimental results from our research and from the research literature. While determining the CLRI parameter values requires an analysis of the agent's implementation, we use computational learning theory to calculate bounds on some of these parameters. These bounds apply regardless of the agent's learning algorithm. Finally, we study a specific market-based MAS in detail. We confirm the agents' behaviors, as predicted by the CLRI framework, and present other findings specific to market-based MASs, such as the fact that learning agents make the system more robust to the presence of malicious agents, and that agents can expect decreasing returns for increasing levels of strategic thinking.

To Rachel.

ACKNOWLEDGEMENTS

I would like to acknowledge the help and guidance of my advisor Ed Durfee, as well as the help I received from the other members of my committee: Robert Axelrod, William P. Birmingham, John E. Laird, and Michael P. Wellman. I also want to thank the members of the AI Laboratory for their support. I am especially indebted to Eric Glover, Marcus Huber, Jaeho Lee, Tracy Mullen, Sunju Park, and Peter Weinstein.

Of course, none of this work would have been possible without the financial support I received during my graduate career. I was lucky enough to be awarded a National Science Foundation fellowship, which covered my first years at the University of Michigan. My later years were supported by a research assistantship as part of the UMDL project, which is sponsored by the NSF/ARPA/NASA Digital Library initiative.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Imagine yourself as a merchant of apples in a small "farmers' market." Your goal as a business-minded individual is to maximize your profit. In this farmers' market, there are other similar merchants trying to sell their products to prospective buyers. Each prospective buyer is free to wander around, get a price bid from each vendor (for the particular good he is interested in) and then make a decision based on these bids, i.e., there are no "posted" prices. After a sale is made, the buyers will often tell all the vendors what everyone else bid, in the hopes that this will encourage some of them to lower their prices for the next time. The problem you face, as one of these vendors, is simple; each time a buyer asks you for a price you have to decide how much to bid. You want to bid high so as to maximize your profit, but you do not want to bid too high so that the buyer will opt to buy from somebody else.

An economist, would tell us that the prices charged by all the sellers in this market will drop down to the sellers' marginal cost. That is, if the apples you are selling are completely identical to the ones merchant Sally is selling, and there is no other reason why the buyers might prefer to buy from you rather than from Sally, then the only difference the buyers see is the price. They will, therefore, opt to buy from the merchant with the lowest price. This means that you will have to lower your price to beat, or at least match, Sally' s price. She will, in turn, lower her price. All the other merchants will also do the same. The price war will continue until the merchants' prices are so low that they only cover the cost of producing the apples. Those merchants with higher production costs will leave the market since they are not be able to lower their prices enough to match or beat the prices offered by the other merchants. Eventually, the price will reach its minimum value. The remaining merchants in the market, at this time, will all have the same cost of production since they are able to sell at this price. This final price is the marginal cost of the remaining merchants. Economists, when doing an equilibrium analysis, often assume that all the vendors of the same good have the same cost of production. Because of this assumption they can claim

that, in these types of systems, where all the goods are completely identical and the vendors have a fixed marginal cost, the price will converge to the sellers' marginal cost.

However, imagine the more realistic situation where the apples you sell are different from the apples Sally sells, e.g. your apples are sweeter than Sally's; her apples are a little tart. In fact, imagine that everybody's apples are slightly different and that buyers have individual preferences over the different merchants' apples. In this situation, it is no longer the case that the buyers will simply pick the lowest price. For example, a buyer named Bob might really dislike tartness and be willing to pay you twice what he pays Sally, since your apples are sweeter than Sally's apples. If you know this about Bob, then your best strategy is to give him a price that is twice what Sally will charge him. Of course, in order to do this you will need to know how much Sally will charge Bob. You can learn both how much Sally charges Bob, and the fact that Bob is willing to pay more for your apples, by simply observing their past behaviors and building models of Bob and Sally. Then, under the assumption that their past behaviors are good predictors of their future behaviors, you can use these models to determine your best possible bid.

This new situation is much more interesting. You have to make a decision based on what you have learned about the others, i.e., your *models* of the other people in the market. Of course, Sally might also be making her decisions based on her models of others, so you might have to model her, and her models of others. Remember also that her models will change over time as she makes new observations, so your model of 'her and her models' will also have to keep changing. You face two different problems. Your first problem is to decide how many of these nested models you should use. That is, should you consider what Sally thinks about what you think about what Bob is likely to pay for her apples? Or should you just think about what Sally thinks Bob is likely to pay for her apples? Your second problem is determining whether or not you are learning fast enough to keep up with changes in the behaviors of the other people, and what to do about it if you are not. For example, Sally might be changing her bids too fast for you to accurately predict what her next bid will be; what should you do?

1.1 Problem Statement

Now, imagine that instead of market with real people we have a market populated by software agents. Your new job is to build a software agent that takes your place in the market—buying and selling goods on your behalf. The problems you face are still the same ones you faced in the farmers' market. However, since you are implementing a software agent, you will need to solve these problems in a more formal way. How one might go about building such an agent is the motivating problem for this thesis. To put it succinctly, the

question we are interested in answering is the following:

> How do we build effective learning agents in multi-agent systems composed of other learning agents?

We want to build agents that take the best possible actions, as dictated by their individual utility functions (e.g. a merchant agent will want to take actions that maximize its profit).

Specifically, we restrict our research to the problem of designing effective agents that learn about, and *only* about, other agents. The problem of designing agents that learn about the world and what actions to take in the world is the traditional machine learning problem in Artificial Intelligence (Mitchell, 1997). We do not address traditional machine learning problems in this thesis. We simply use well-established machine learning algorithms and apply them to the problem of agents learning about agents.

Of course, the problem of building agents that learn about agents in multi-agent systems is not the same problem that traditional machine learning addresses. In order to determine how to build effective agents that learn about agents, we need to address two separate problems which do not arise in traditional machine learning research.

The first problem deals with the issues that arise when using nested agent models. Imagine an agent that can build (or has access to) models of other agents. This agent must pay some computational cost for building and/or using these models. Should this agent build and/or use models of other agents? Should it build and/or use models of the other agents' models of others? Which particular models should the agent use? How does the agent know when to stop building/using deeper nested models of other agents? These questions are all part of the nested modeling problem.

The second problem we need to address is called the moving target function problem. Imagine that the agent cannot simply generate models of other agents but must, instead, learn them via observation. The other agents are also learning. Is the agent's learning fast enough to keep up with the changes in the other agents' behaviors? What happens to a system with these learning agents? Does the system behave chaotically or does it converge? If the system converges, what is the equilibrium behavior of the agents? These questions are all part of the moving target function problem.

We were led to these two problems by our motivating problem of building effective agents that learn about agents in multi-agent systems. The two problems are, in fact, interdependent. The nested models are simply a way of structuring an agent's knowledge. With each nested model we can associate a learning problem, and each one of these learning problems might have a moving target function problem associated with it. For example, Bob might never change his behavior, while Sally might change her behavior frequently.

Therefore, when modeling Bob you simply face the traditional machine learning problem, while when modeling Sally you face the moving target function problem.

The two problems of determining what nested models to learn/use, and of the moving target function, are important problems that must be addressed before we can build effective learning agents in multi-agent systems composed of learning agents. In this thesis, we address these two important stepping stones, which provide answers to the more general problem of building effective agents that learn about agents.

1.2 Real World Significance

We believe that being able to build effective learning agents for MASs composed of learning agents is a very useful and important technique to develop in today's world. The explosion of the Internet, especially the Web, has generated wide interest in all kinds of agent systems. These systems range from digital libraries, like the University of Michigan Digital Library (UMDL) (Atkins et al., 1996), to commercial offerings like Jango (Etzioni, 1996) that scour the web for information, to systems designed for the seamless integration of legacy database systems such as the InfoSleuth project (Bayardo Jr. et al., 1997). In addition to these Internet-based agent systems, there are other more mature robotic MAS applications, such as distributed sensor networks and assembly line scheduling (Jennings et al., 1995) (Sycara et al., 1991).

The Internet has also led to an explosion in the amount of research being done on electronic commerce (e-commerce), along with a corresponding increase in the number of e-commerce applications. The UMDL itself is an early example of a system based on an e-commerce infrastructure where agents buy and sell services from each other. Other examples are the Kasbah (Chavez and Maes, 1996) project, where agents place bids on behalf of their owners using fixed bidding strategies, and the AuctionBot (Wurman et al., 1998), where the users are responsible for their bids but many different types of auctions can be instantiated. Examples of commercial auction web sites are too numerous to mention. Long listings of them can be found on many of the Web directories. All these e-commerce systems are really just MASs, some of them with humans as the agents, although the trend seems to be towards giving software agents more independence from their owners. One characteristic these systems have in common is their assumption that the agents in the MAS have the shared goal of participating in the system, but also want to maximize their individual utilities.

Up to now, of the relatively few MASs that have been built using only software/hardware agents, most have been designed by groups that are responsible for building all the agents in the system. The agents in these systems can, therefore, be built with all the knowledge

they need about the other agents in the system. Protocols can also be designed beforehand that coordinate the agents' actions. However, as more MASs are built, and as it becomes more desirable and easier for agents from different systems to talk to each other, there will be an increasing need for agents that can interact with new agents from other systems. The new MASs will be built so as to allow for the addition of these new and more interactive agents.

In these new MASs, of which the UMDL is a good example, new agents will come and go as time goes on. The agents in the system will be expected to learn about the other agents as these appear, and deal with them as any person would deal with a stranger—first exercise some caution, then lower your defenses as trust is gained. We call these types of MASs **open Multi-Agent Systems** because they are open to any agent that might want to join, and do not impose unnecessary rules on an agent's behavior (e.g., they allow agents to be selfish). This also means that the agents will probably not share a top-level goal (except, perhaps, the goal of participating in the system). Designing effective agents for open MASs means designing effective learning agents that learn about agents, the central problem addressed by this thesis.

We should also mention that the techniques presented in this thesis are also useful in traditional cooperative MASs. Specifically, they are useful in MASs where the agents cannot be built with all the prior knowledge they need about the other agents, and this knowledge cannot be given to the agents (either through communication with the agent, or by directly programming the agent) after they start operating. The agents must acquire this knowledge through observations of the other agents' behaviors. One example is a system where the agents are learning about the world. In this case, the agents' behaviors change over time and, therefore, cannot be programmed into the other agents from the start. Another example is a system where some agents are built before the behaviors of some other agents are completely specified. In this case, the first agents to be built could learn the behaviors of the other agents after these have been built and installed into the system.

Our techniques might even be useful in the analysis of results gathered from agent-based simulations. In this thesis, we present theories that predict expected system behaviors. A researcher might be able to use these to differentiate between the emergent behaviors he observes in his agent-based simulations that are general phenomena found in MASs with learning agents, and those behaviors that are specific to the particular model he is studying.

1.3 Thesis Overview

The approach we have taken for determining how to build agents that learn about agents is two-fold. For the first part, we have built a formal framework that can represent different types of learning agents and can predict their expected behaviors. The second part consists of some implementation-oriented work. This work includes an algorithm for building better nested modeling agents, and a study of learning agents in a market-based MAS. These implementations build on our framework, providing confirmation of our theoretical results and expanding our understanding of these specific MASs.

In this section, we summarize the contents of this thesis. The subsequent chapters recapitulate the ideas that follow, delving into them in more depth.

1.3.1 Formal Framework

In order to make any progress in answering our main question of how to build effective learning agents in MASs composed of other learning agents, we must first formalize what we mean by a multi-agent system. This formalization will necessarily have to make some assumptions and simplifications about the world and the agents in it. We have developed such a framework. It starts by defining a **decision function** to be the function that describes an agent's current behavior. The decision function maps each state of the world to the particular action that the agent will take in that state. We also define a similar **target function**, which is also a mapping from world states to actions. The target function describes the best possible behavior that an agent could have. That is, the target function describes the behavior of an agent that does not make any mistakes, always taking the best possible action. We assume that one such a target function exists for every agent, even though we might not be able to generate it. An agent, typically, does not have direct access to its target function (if it did, it should just set its decision function to match its target function). The agent gradually learns its target function by taking actions and receiving some payoffs indicating the correctness of its actions.

We also provide some notation for describing agents that have nested models of other agents. An agent's models of other agents can be nested to any arbitrary depth. For example, the behavior of an agent that is not aware of the fact that there are other agents in the world, and simply takes actions based on the current world state, can be described with a simple decision function. We call this agent a 0-level agent. An agent that realizes that there are other agents in the world and models each one of them with a simple decision function, while using another decision function to determine what to do given the world state and the actions of the others, is called a 1-level agent. A 2-level agent models other agents

as 1-level agents. In this way, we describe any k-level agent, where k is any non-negative integer.

We measure an agent's performance using a metric which we call the **error** of the agent, a term borrowed from Computational Learning Theory (Kearns and Vazirani, 1994). An agent's error is just the probability that the agent will take an action that is different from the action the agent's target function dictates the agent should take (i.e., the correct action). This error can also be approximated using the errors of the different nested functions, along with the amount of **dampening** that is present in the agent's knowledge. A dampening value is associated with a decision function, and is defined as a number between 0 and 1. We can intuitively explain what we mean by dampening by first considering a k-level agent that has a series of nested decision functions, corresponding to its nested models of other agents. The total error for such an agent is bounded from above by the sum of the errors at each nested level. However, if some of the levels contain decision functions with dampening, this means that the errors below those decision functions can be discounted. For example, if a decision function has a dampening value of 1, then it does not matter what input values this function gets, its output will always be the same. If, on the other hand, the decision function has a value less than 1, then the input values that this function gets will matter when determining the output value of the function. The lower the dampening value, the less the errors below the decision function can be discounted, and thus the more that the input values will matter.

Finally, we show how our notation for recursive modeling agents captures the knowledge that other researchers (Claus and Boutilier, 1997) have placed into their agents. Our ideas have also proven valuable enough that some researchers (Hu and Wellman, 1998) have found it beneficial to use our notation as a tool for describing the knowledge of their agents. We can only hope that, since our notation captures the type of nested knowledge that many researchers use, the field will use it, or an extension of it, as the standard for describing the knowledge that agents in MASs have about each other.

1.3.2 An Algorithm for Exploiting Dampening

With a framework in place, we now explore the problem of building agents that can effectively act in MASs. This problem is broken down into a series of manageable steps. We start by assuming that the agents are not learning but are, instead, "born" with functions which they can use to generate any nested model they need. Given this assumption, the question we must ask is: which of these nested models should the agent generate and use? The answer to this question is complicated by the fact that there is a time cost associated with the generation of models. That is, it takes time for the agent to generate new models

and the agent must take action within a certain time limit.

We give an algorithm called LR-RMM (Limited Rationality Recursive Modeling Method) that an agent can use to determine which of the nested models it should generate and when it should stop generating models and take an action. The algorithm makes use of the fact that there is dampening present in some of the models. The existence of such dampening means that the results given by some of the models will have no bearing on the eventual choice of action for the agent. These "useless" models are pruned away by LR-RMM so the agent does not need to spend time generating them. This pruning preserves the solution quality, while reducing the time it takes to compute which action the agent should take.

LR-RMM is based on the concept of an agent's **gain** from the generation of a deeper agent model. Since it takes time to generate a model, there is an inherent negative gain in doing so. However, the new model might lead the agent to change its mind about what action to take and, in effect, make the agent take an action that has a higher expected utility. The difference in the utilities the agent gets with the old action and with the new action (after generating the model), minus the time costs of generating the model, is the gain the agent earned from generating the model. Unfortunately, an agent can only calculate a gain *after* it has generated the model, since it needs the generated model in order to calculate the gain from generating it. Fortunately, we can get around this problem. We use a method that allows the agent to learn expectations about what action the generation of a model will lead to. These expectations are then used to determine the agent's **expected gain** from the generation of any particular nested model.

We confirm that LR-RMM works as claimed by testing it on the Pursuit Task, a traditional problem used in Distributed Artificial Intelligence. While we only test it on this one task, the algorithm can be applied to any task where an agent has a number of nested models available and needs to determine which ones to use at any given time. However, in order for the algorithm to perform well, the agent needs to be able to calculate expectations about what actions the new models will lead to. These expectations can be implemented by hand or calculated automatically with a series of test runs of the system.

1.3.3 The CLRI Formal Framework

The next step we take is to eliminate the assumption that agents already have models of others and, instead, assume that agents must learn these models of each other. We, temporarily, ignore the existence of nested models and consider all of an agent's knowledge as just one big decision function. We extend our formal framework to include a series of parameters which capture the capabilities of an agent's learning algorithm. We call this set of parameters the **CLRI** framework. The name is an acronym for the names of the

parameters used, i.e., the Change rate, Learning rate, Retention rate, and Impact.

The **change rate** of an agent is the probability that the agent will change one of the world-state-to-action mappings in its decision function to a different action, given that the action in the mapping was not the same action dictated by the agent's target function. The change rate tells us the probability that the agent will change one of its incorrect mappings to a different action. The new action need not be the correct action, as dictated by the target function. The **learning rate** of an agent is the probability that the agent will change one of its incorrect mappings to the *correct* action, as given by the target function. The **retention rate** of an agent is the probability that the agent will retain one of its correct world-state-to-action mappings. Finally, the **impact** that one agent has on another agent is defined as the probability that a change in one agent's decision function will lead to a change in another agent's target function. The impact value gives us an idea of how much each pair of agents impacts each other. An impact is defined for every ordered pair of agents in the system. In summary, the change, learning and retention rate parameters capture the agents' learning abilities, while the impact captures the amount of coupling or inter-dependence that exists between the agents in the MAS.

An important aspect of the CLRI framework is the fact that it addresses the **moving target function** problem. This problem appears when a learning agent is trying to learn a given target function but this target function keeps changing. Our framework tackles this problem for the case where the target function changes because, and only because, the other agents are changing their decision functions. In the systems we study, the other agents change their decision functions because they are also learning.

Using the CLRI framework and its parameters, we derive a series of equations that can be used to predict the expected error progression of a learning agent in a MAS composed of other learning agents. These equations can, for example, tell us whether an agent's error is expected to converge, as time goes to infinity, to some value, and what that value is. They can also tell us the value of the error after convergence for all possible combinations of the change rate, learning rate, retention rate, and impact. We have confirmed the values predicted by our equations with experimental results from economies of agents that use reinforcement learning, and with experimental results published by other researchers.

1.3.4 Learning Nested Models

Learning each one of the nested models roughly amounts to a separate instance of the moving target function problem. However, these instances are not completely independent. That is, there is a mapping between the type of knowledge that an agent designer has and the nested models he can directly implement. For example, a designer might know what his

agent should do *given* the actions of the other agents. This knowledge is captured by one of our nested decision functions. However, some of this knowledge could also be captured in some other decision functions. For example, an extreme case is when the designer knows that his agent should take action a no matter what actions the others take. This action is referred to as a dominant strategy in game theory. In this particular case, all the nested decision functions, except for the top-most one, are irrelevant since there is nothing worth learning about the other agents. In a less extreme case, we might find that the designer knows that his agent's choice of action depends on the choice of action of one particular agent, regardless of the actions taken by the other agents.

Therefore, given the designer's knowledge, we determine how that knowledge is spread out among the nested decision functions. We say that a function contains some of the designer's knowledge when that function has been correctly defined, by the designer, for some state-to-action mappings. The rest of the mappings will still have to be learned. Since learning these remaining mappings is an instance of the traditional machine learning problem (assuming a fixed target function), we can use Probably Approximately Correct (PAC) learning theory (Kearns and Vazirani, 1994) to determine an upper bound on the number of training examples that a consistent learning agent must experience before it can be said to have a PAC decision function. In order to determine this upper bound, we must measure the size of the agent's hypothesis space. For example, if an agent has a correctly defined decision function, then its hypothesis space contains just this one possible behavior. If, on the other hand, the decision function has no mappings defined, then the size of the hypothesis space is the number of possible behaviors that arise from all possible combinations of legal mappings that the decision function can have (i.e., the number of possible decision functions).

The size of the hypothesis space gives us an idea of the size of the learning problem faced by the agent. The bigger the problem, the longer it will take (in terms of learning events) for the agent to learn. We can use the size of the hypothesis space to determine a lower bound on a consistent learning agent's learning rate. This lower bound can be useful by itself, since a designer will know whether he can easily implement an agent with the given learning rate. A designer can also use the lower bounds on the learning rates of two or more possible agent designs in order to determine how these agents' learning abilities compare. The comparisons of the lower bounds are expected to lead to the same conclusions as would comparisons between the real learning times (if known), as long as the agents use similarly powerful learning algorithms.

1.3.5 Learning in a Market System

Finally, we implement an actual market-based MAS composed of learning agents with nested decision functions. The agents are distinguished as either buyers or sellers of a service. Each agent is implemented as either a 0, 1 or 2-level agent. Some of the agents' decision functions are learned using reinforcement learning or model-building; other functions are implemented directly into the agents. We run a battery of tests on this system in order to confirm the results predicted by the CLRI model, and to find out what other conclusions we can make about the use of nested modeling agents in a market system.

The model economy used is based on the University of Michigan Digital Library. The exchange protocol is simple. When a buyer wants to buy some good/service, she advertises this desire. The sellers that sell this good send her bids that reflect how much they would charge her. She picks one seller and sends him a payment of the amount of his bid. The seller then gives her the good in question.

The system does not guarantee any sort of quality of service, so the sellers are free to return a good of any quality they desire. The buyer, therefore, must assess the quality of the good received and learn from this experience. In future encounters (we assume transactions are very frequent and small), the buyer will use this knowledge when deciding which seller to buy from. The sellers, meanwhile, must determine how much they can charge for their services so that they maximize their profits, remembering that, if a seller is not chosen by a buyer, that seller does not make a sale and his profit is zero.

In this system, 0-level agents base their bids only on the set of outstanding bids for the good (e.g., 0-level buyers consider only the set of bids that the sellers send them), or on the good being sold (e.g., 0-level sellers consider only the good that some buyer advertises that she wants to buy). 0-level agents do not recognize the fact that there are other particular agents out there. They consider the bids and advertisements as independent entities. 1-level agents recognize the fact that there are other agents out there and try to model them. These models allow 1-level buyer agents to determine the kind of quality a seller is expected to offer, and they allow 1-level seller agents to determine the amount of money sellers are expected to bid, and buyers expected to pay. 2-level agents model other agents as 1-level agents and, therefore, might have a better prediction of their actions.

Since the 0-level buyers do not model the sellers explicitly, they are not able to determine when a particular seller returns a good which, the buyer believes, is of a very low quality. The 0-level buyers must use price as a signal of the expected quality of the good they are buying. This seems like a big handicap for these 0-level buyers but, in fact, it is not as bad as it might at first seem. For example, if all the sellers are returning the same quality, then there is no need for the buyers to explicitly model these sellers. On the other hand, if some

sellers are selling a higher quality, then they will tend to price it higher, in order to make a profit. The buyers can then successfully use price as a signal for the quality they expect to get (i.e., higher prices mean higher quality, on average).[1] The handicap of 0-level buyers can be seen when strategic sellers of low quality goods start pricing them higher in order to make them appear to be of a higher quality, and get a higher profit. 0-level buyers are powerless against this trickery. However, 1-level buyers can model these sellers explicitly and avoid them.

Therefore, the more 1-level buyers there are, the less the sellers are expected to gain from being strategic; the sellers should all just bid according to the quality they sell. But, if all the sellers bid according to their qualities (i.e., assuming higher quality means a higher marginal cost), then the buyers do not need to be 1-level, and incur the additional computational costs. The buyers can simply be 0-level.

This is just one example of the interesting dynamics we find in our system. In general, we find that the use of learning agents makes the economy much more robust, in the sense that it cannot be manipulated by malicious agents. On the other hand, we also find that the deeper models often do not give the agents that are using them any higher profits. But it is these same deeper models that make the economy more robust. Therefore, a robust economy needs agents that are capable of sometimes keeping nested models, but these agents should also be smart enough to realize that they do not always need to keep these deeper models. In a similar vein, we also found that there are decreasing gains from increasing modeling depths. This means that an ever-expanding escalation in modeling depths, even when ignoring computational costs, is unlikely.

We also confirmed theoretical predictions from our CLRI framework. For example, we show how price volatility is correlated to the gains of a 1-level seller over a similar 0-level seller. In order to do this, we first map price volatility to the volatility value used in the CLRI model. Also, as we explained in Section 1.3.4, we notice that the 1-level agent has a bigger CLRI learning rate than the similar 0-level agent. This means that we can use the predictive equations from the CLRI framework to determine what the expected error for these two agents will be. This error is inversely proportional to the profit that the agents get. An application of our CLRI equations shows that we expect to see a linear correlation between volatility and the gains of the agent with the higher learning rate (1-level) over the one with the lower learning rate (0-level). Experimental results with our system show the same correlation.

[1] That is, as long as both buyers and sellers agree on what "quality" means. Our system can also simulate a scenario where the agents have different opinions on the quality of certain goods.

1.4 Contributions

The main thrust of this thesis is the question of how to build effective learning agents in multi-agent systems composed of learning agents. This question led us to the development of a framework for formalizing the problem, an algorithm for determining which nested learning models to use, and a theory (CLRI) that tells us how a learning agent is expected to fare in a multi-agent system. We also studied this question empirically by building and analyzing a market-based multi-agent system with nested learning agents. The results of those experiments showed that having learning agents in such a system, apart from being an inevitable consequence of having selfish individuals build the agents, makes the system much more robust and adaptable.

The specific contributions of this thesis are:

- We develop a framework for describing learning MASs and for categorizing the agents' nested knowledge (Chapter 3). This framework explicitly represents the agent's nested models of other agents. It can be used to describe the types of knowledge that researchers in multi-agent learning are using (e.g. (Claus and Boutilier, 1997), (Hu and Wellman, 1998)). The framework builds upon the work of (Gmytrasiewicz, 1992).

- Our framework introduces the concept of knowledge dampening. That is, a model of an agent can sometimes make other agent models irrelevant (e.g., if I expect a buyer will always buy from me then I don't care what the other sellers offer him). The recognition of this fact has direct consequences in the building of learning agents.

- We design an algorithm (LR-RMM) that exploits the dampening in an agent's knowledge and allows the agent to dynamically determine which of its nested models to use (Chapter 4). LR-RMM shows how an agent can take the best possible action without spending unnecessary time "thinking."

- We provide a predictive theory and framework (CLRI) that can be used to predict the expected behavior of each of the learning agents in a MAS, given the value of a series of parameters that describe the agents' learning abilities (Chapter 5). This theory is confirmed with our experimental results and with experimental results observed by other researchers.

- We show a method for calculating the size of the hypothesis space (i.e., the size of the learning problem) for learning the different nested decision functions of a nested modeling agent (Chapter 6). This method can be used when the agents use either some form of reinforcement learning or some form of supervised learning. The size

of the learning problem is used to determine a lower bound on the learning rate of a consistent learning agent. This lower bound can, in turn, be used as a way to compare two or more possible agent designs before implementing them.

- Finally, we provide an analysis of the effects of nested learning agents within an economic MAS (Chapter 7). We show the relative advantages of keeping nested models, how having more (correct) knowledge gives an agent an advantage in highly volatile markets (i.e., those where an agent's target function changes frequently), the fact that agents can expect decreasing returns for increasing levels of modeling, and other results. Our experiments also help us to distinguish between those emergent behaviors that can be predicted by our theory versus those behaviors that are too domain-specific and can only be reproduced through implementation and agent-based simulation.

CHAPTER 2

Related Work

In this chapter we introduce previous research which this thesis either makes use of or builds upon. The reader will notice that the research comes from fields as diverse as artificial intelligence, game theory, philosophy, and the social sciences, especially economics. Such diversity is typical of studies in Artificial Intelligence (AI), but is especially pronounced in our case because of our goal of studying societies of learning agents. The study of societies has typically been carried out within the social sciences. Research in the social sciences usually deals with societies of humans, and is therefore mired in all the complications that studying humans, in all their complexity, brings to any scientific study. Luckily, we do not have to deal with complex humans but with relatively simpler computer programs that, while able to learn, are much easier to describe and analyze.

We start with an introduction of what we mean by an "agent" (Section 2.1). We further explore the idea of agents by explaining different theories and frameworks that researchers have used to describe agents (Section 2.1.1), and then delve into the problem of agents modeling agents (Section 2.1.2). It is this last section that is most relevant to the work in this thesis. Finally, we give an overview of the machine learning techniques used in this thesis (Section 2.2).

2.1 Agents

Throughout this thesis we use the term **agent**. At this moment there are several competing definitions of the word agent. For the purposes of this thesis, we assume that an agent is an autonomous (i.e., can act independently of others) and persistent entity (i.e., either a software construct or a robot). A good overview of agents and the different theories and definitions of them is given in (Wooldridge and Jennings, 1995); a more recent survey of the field appears in (Huhns and Singh, 1997).

We subdivide this section into two. First, we talk about different theories that people

have used to model, describe, and implement agents. We then present previous research into the problem of how agents can build models of agents.

2.1.1 Agent Theories— People Modeling Agents

Within the field of AI, and Distributed AI (DAI) specifically, there has been a lot of work done on building models of agents that capture the agents' mental state using formal logic. Most of the early work has tried to determine what are the basic components of an agent's mental state that need to be modeled, in order to have a useful model of the agent. These components typically include beliefs, desires, and intentions. The components are then formalized and instantiated as part of formal theories, based on some logic-based frameworks, that try to explain how the different components affect each other (Jennings, 1994) (Cohen and Levesque, 1990) (Dennett, 1987) (Bratman et al., 1988) (Werner, 1989). This type of research has emphasized the search for logical constructs and semantic theories that can represent our intuitions about what it means to have beliefs, desires and intentions. In this thesis we are not concerned with these high-level logical constructs but with the actual decisions an agent makes and how the agent can arrive at these decisions.

An interesting system that goes beyond just modeling and into the programming of agents is **Agent-Oriented Programming** (Shoham, 1993) (AOP). This system was designed as a tool for programming and describing agents in a multi-agent environment. In it, one explicitly represents the agent's beliefs, decisions, capabilities and obligations, which form the agent's mental state. The system limits the expressibility of this mental state by defining it to consist of only that which can be expressed using their extension of standard epistemic logics. One instance of AOP, described in (Shoham, 1993) and implemented by AGENT0 (Torrance and Viola, 1991), itemizes all the possible types of communication primitives and introduces operators for obligation, decision, and capability. However, it assumes that agents that are capable of establishing commitments with others, and are not otherwise obliged, will do so. Thus, unlike our approach, AOP forces certain cooperative behaviors on the agents, like the fact that agents must establish a commitment whenever they are not otherwise obliged.

We can also take a step back and consider the general problem faced by agents in a MAS. In most cases, the actions of some agents will have an effect on the action choices of others. If the agents are rational, they should try to pick the actions that they expect will bring them the biggest rewards or utilities given the actions of the others. The problem of determining what rational agents should do when faced with some joint decision has been deeply studied in **game theory** (Luce and Raiffa, 1957) (Shubik, 1985). These studies focus on determining what are the "right" actions for players of a particular game to take.

The games are represented as payoff matrices, that give each agent a utility for each joint action of all the agents.

Game theory was primarily designed as a tool for human decision-makers to make the "right" decisions given the set of possible payoffs they were expecting. What made this decision especially difficult was that the payoffs depend not only on the action the decision-maker took, but also on the actions of all the other agents, all of which where also assumed to be rational decision-makers. Since its beginnings, the science of game theory has been used in a wide variety of fields, including economics, math, psychology, etc., usually providing a way to determine how the intelligent agents (i.e., humans) involved in complex systems will behave. In the fields of AI and economics it has also been used as an inspiration for building decision methods that intelligent (i.e., rational) agents can use for making decisions based on their expected payoffs, even when these payoffs depend on the actions of other agents.

A modeling method that is based on game theory was proposed by Binmore (Binmore, 1987) (Binmore, 1988). He goes beyond just saying which are the "right" actions, and proposes a decision mechanism through which the agents can arrive at these actions. This mechanism assumes that the agents have common knowledge[1] of the payoff matrix. The algorithm Binmore provides allows the agents to reach one of the equilibrium solutions. The algorithm works by iteratively improving the agents' guessed solutions. The quality of an agent's solution, therefore, improves monotonically. Binmore also allows for the presence of probabilistic or uncertain knowledge. That is, the model can include probabilities associated with the payoff matrices of other agents, in case the agent is not completely sure what the payoff matrices of its opponents look like.

We include game theory in this overview because the insights behind this discipline are used through this thesis (e.g., the ideas of joint actions, equilibrium points, expected utility). However, we have chosen not to make the common knowledge assumption, since it implies that the agents have a great deal of knowledge about each other and there are certain MASs where it is unlikely that the agents will have this much knowledge about each others' payoffs. For example, in MASs where the agents are built by different individuals, the agents are unlikely to know much about each other since their builders are themselves unlikely to know that much about each other, especially if they are in a competitive environment. We are also interested in the question of how agents can acquire this knowledge about each other and/or how they decide which parts of it to make use of. Our rejection of the common knowledge assumption echoes the concerns that led to the development of the Recursive Modeling Method, which we introduce in Section 2.1.2.

[1] Common knowledge about fact x means that everyone knows x, and everyone knows that everyone knows x, and everyone knows that everyone knows that everyone knows x, and so on.

2.1.2 Agents Modeling Agents

More central to this thesis is the research into the question of how an agent can model another agent, i.e., how an agent can learn to predict what another agent will do, or is trying do, by simply observing the actions it takes. Several different approaches to this problem exist. We review them in the following sections.

Plan recognition

A lot of research has been done under the name of **plan recognition** (Huber et al., 1994) (Pollack, 1990) (Goodman and Litman, 1990) (Charniak and Goldman, 1993) (Konolige and Pollack, 1989) (Carver et al., 1984). As can be deduced by its name, the work in this area has close ties with traditional AI planning. In plan recognition, the agent doing the recognizing is assumed to have knowledge of all the plans that the other agents might be executing. The agent's task is to identify which of these plans the other agents are executing. The task is complicated by the presence of noise in the input and the possibility that the other agents are interleaving and executing two or more plans simultaneously. Each one of the different approaches to plan recognition has its specific advantages and disadvantages. Some require that the representation of the plans be modified into a form that the recognizing agent can use, while others use the same representations for planning and plan recognition. Still others use different representations but have automatic translators. Also, some approaches are better at providing good guesses quickly while other approaches take longer to form a definite answer, although once they have formed and answer the user can be much more certain that it is the correct one. Still, all of the methods assume that the recognizing agent has some knowledge of the others' plans; none of the methods tries to start with an empty knowledge base or with sketchy (i.e., not in the form of plans) information about the other agents. In this thesis we do not assume that agents have these long-range plans—the agents' behaviors are based on the current state. We also study the possibility that an agent knows nothing, or very little, about the others and has to learn this knowledge via observations. Therefore, the results from plan recognition research are not a good match for the problems we are trying to solve in this thesis.

Recursive Modeling Method

The basic modeling primitives we use were inspired by the **Recursive Modeling Method** (RMM) (Gmytrasiewicz, 1992) (Gmytrasiewicz et al., 1991) (Durfee et al., 1994) (Durfee et al., 1993). RMM provides a theoretical framework for representing and using the knowledge that an agent has about its expected payoffs and those of others. It is based on payoff matrices, just as game theory uses, but it eschews the assumption that agents have

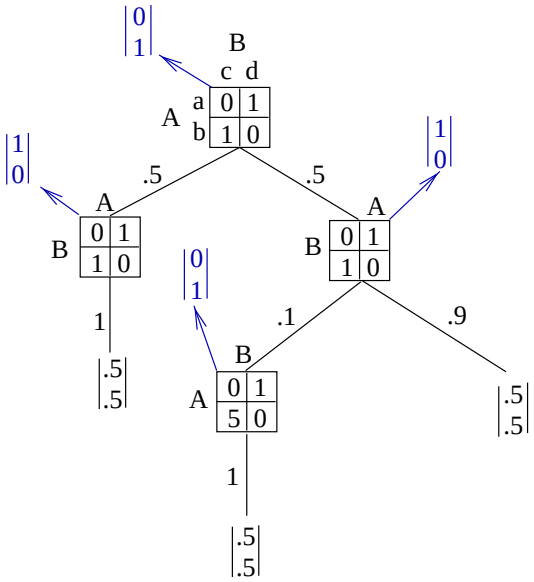


Figure 2.1: This is the hierarchy of a simple RMM game to four levels for two players A and B, annotated with the solutions at the different matrices. The bottom most element is the "zero knowledge" solution. After propagating values to the top, the solution we get tells us that the player A should pick choice b.

common knowledge about these payoffs. In order to use RMM, an agent is expected to have a payoff matrix where each entry represents the payoffs the agent expects to get given a combination of actions chosen by all the agents. Each entry in the matrix corresponds to a different combination of actions, and all possible combinations should be represented. Typically, each dimension of the matrix corresponds to one agent, and the entries along this dimension correspond to the actions the agent can take.

The agent can also model the other agents, similarly, as having payoff matrices. In fact, the agent can recursively model others as having payoff matrices and modeling others the same way, and so on, as seen in Figure 2.1. The recursive modeling only ends when the agent's knowledge depth is reached, that is, when an agent runs out of knowledge. At this point a zero knowledge model is used, which basically states that the modeling agent has no knowledge about the agent being modeled. Given this lack of knowledge, the modeling agent assigns an equal probability to each of the other agent's actions. For example, the top matrix in Figure 2.1 shows the payoffs agent A expects to get given each combination of its two actions (a and b), and B's two actions (c and d). The matrices directly below the top one show the payoffs A thinks that agent B will receive. The left matrix contains the payoffs that A thinks, with probability .5 (as shown by the edge connecting the matrices),

that B will receive. The right matrix shows the payoffs that A thinks, with probability .5, that B will receive. Notice, however, that the two payoff matrices are different. That is, A thinks that with probability .5, B will receive the payoffs given by the matrix on the left and, with probability .5, B will receive the payoffs given by the matrix on the right. At the leaf nodes of the tree we find the zero-knowledge models, which attribute an equal probability to each of the agents' actions. Since, in this case, both agents have only two possible actions to chose from, their zero-knowledge models give each one of these actions a probability of .5.

RMM provides a method for an agent to use this hierarchy of payoff matrices to determine which action or strategy[2] to take. The method works by propagating a solution strategy from the leaf nodes, where the zero knowledge model is used, to the root node. Each node models some agent and contains the matrix of payoffs that the agent receives, given the strategies played by the other agents. Once these strategies are available, we can plug them into the agent's payoff matrix, in order to determine which strategy will give the agent its highest expected payoff. This strategy can then be used to feed into the payoff matrix above this node. This calculation is repeatedly applied, from the leaves to the root, in order to get the strategy that tells the agent, whose RMM hierarchy this is, which action to take. For example, the bottommost matrix in Figure 2.1 has one leaf node under it. This leaf node contains the zero-knowledge model for agent B. We evaluate this bottommost matrix by assuming that B takes actions c and d with equal probability, and then determining what is the best strategy for A. Given this matrix, the strategy that gives A the biggest payoff is to play b. We represent this strategy with the vector $|0\ 1|$, where 0 is the probability that A will take action a, and 1 is the probability that A will take action b. This becomes the strategy the node evaluates to, as shown in the figure at the end of the arrow which comes out from the matrix. The strategy is then used as A's strategy for the matrix above this one. Notice, however, that the matrix above this one has two children. Therefore, A's strategy for this middle matrix will be .1 multiplied by the strategy which resulted from evaluating the bottommost matrix (i.e., $|0\ 1|$), plus .9 multiplied by the zero-knowledge strategy from the corresponding leaf. The sum of these two terms results in the strategy $|.45\ .55|$, which is A's strategy for this middle matrix. Given this information, we can determine that B's best strategy for this middle matrix is $|1\ 0|$. This process continues for all nodes on the tree until, at the root, we determine that A's strategy should be $|0\ 1|$. That is, A should take action b.

RMM has some limitations. Firstly, the time it takes to evaluate an RMM hierarchy

[2]A strategy can be either an action or a mixed strategy. A mixed strategy associates a probability with each action. The sum of all the probabilities must be equal to one.

grows exponentially with the number of agents and the depth of modeling. RMM does not provide a way to limit this computational time without sacrificing the quality of the solution. Secondly, the assumption that the agents have these nested payoff matrices, which reflect the internal payoffs other agents receive, is not a reasonable assumption to make in many domains. Specifically, in adversarial MASs (i.e., those with selfish agents), it is unlikely that agents will know so much about each other. The agents in these systems usually have to learn about the other agents via observations of their behaviors.

Other nested modeling methods

Another approach, similar in spirit to ours and RMM, is given in (Tambe and Rosenbloom, 1996), (Tambe, 1995). The authors use the SOAR (Laird et al., 1987) architecture, along with its ability to recursively use an agent's own code as a model of the agent (i.e., the model of the agent is, in fact, the very same code that implements the agent). The nesting achieved is very similar to the one we use in this thesis, and it also suffers from the same problems. That is, the computational costs of using deeper models grow exponentially with the number of agents and depth of modeling. The authors' solution to this problem involves the clustering of agents into groups of similar agents. The authors try to reduce the number of agents that need to be modeled by claiming that certain groups of agents are identical and so need only be modeled once. This solution is effective for their particular implementation of air-to-air combat, but is not applicable in general. We present a more general solution to this problem in Chapter 4.

A good formal framework for talking about the knowledge of agents, including the knowledge an agent might have about another agent's knowledge, is given in (Fagin et al., 1995). With this framework they can prove whether or not an agent can deduce a given statement, given its knowledge. Their notation takes the form of $K_i K_j p$, which means that agent i knows that agent j knows p. This notation echoes the idea of nested agent models that we will be using throughout this thesis. However, the semantics of their theory is too constraining for us since they assume that, if an agent knows p, then p *must* be true. That is, their agents are not allowed to know false knowledge. Given that our agents will be learning, and that there are no perfect machine learning algorithms, it is likely that some of the things that our agents know (i.e., believe) will, in fact, be wrong.

2.1.3 Limited Rationality

As we just mentioned, nested agent models are a good way for an agent to decide what it should do, but their computational costs grow exponentially with the modeling depth. We give a solution to this problem in Chapter 4. Our solution builds upon work done under

the names of limited rationality and bounded rationality.

The term **bounded rationality** was used by Simon (Simon, 1982), in the field of economics, to refer to organisms that do not try to optimize their behavior but simply try to do "good enough" or satisfice.[3] He argued that a complete theory of rationality would need to take into account organisms whose behavior is not optimal and is constrained by their processing and knowledge limitations. These ideas, which in the field of economics provided a descriptive theory, were the inspirations for the field of **limited rationality** (Russell and Wefald, 1991) within the AI community. Limited rationality addresses the issues of how agents can use their knowledge to make useful and timely decisions. That is, an agent using limited rationality algorithms can take into account its hardware, time, and knowledge limitations when making a decision. This means that the agent could be forced to take a quick action because the situation demands a quick response and the agent's computational speed cannot handle the evaluation of all pertinent data in the allotted time. On the other hand, it could mean that the agent stops deliberating about a particular action choice because it thinks that no further deliberation will make it change its mind about which action to take. We will now summarize some of the existing implementations and theories of limited rationality that try to give these capabilities to AI agents.

The issues surrounding limited rationality and its implementation in AI agents have been studied by Russell and Wefald (Russell and Wefald, 1991), whose approach is based on the scheduling of discrete computational actions using metalevel thinking. Another approach that uses **decision theory** is given by (Horvitz, 1988) in his work on formal models of rational decision making for computational agents with explicit consideration of preferences and resource availability. Other approaches to the same question lead to **anytime algorithms** (Dean and Boddy, 1988), in which the deliberation process can be stopped at any time and an answer will always be returned. The quality of this answer is expected to increase monotonically as a function of the time spent in deliberation. This work concentrates on the issues in time-dependent planning. **Deliberation scheduling** (Boddy and Dean, 1994) explicitly allocates computational resources based on the expected benefits that they will provide to the system as a whole. It assumes that these decisions can be made in a reliable and timely manner by a central controller. **Design-to-time** decision procedures (Lesser et al., 1988) are decision procedures that are generated at run time and can guarantee a result of a certain quality before a given deadline. They handle the uncertainty by, at first, trying to get as good an answer as possible and then, as time runs out, they start trading off the quality of the answer for the expected time to run the

[3]Simon defines a "satisficing" decision method as one that looks for good or satisfactory solutions instead of optimal ones (Simon, 1996).

decision procedure.

2.1.4 Multi-Agent Learning

In the past few years we have seen a growing interest, within the MAS community, in the study of MASs with learning agents (Sen, 1996), (Weiß, 1997), (Sen, 1997). This area is sometimes called multi-agent learning. For example, (Nadella and Sen, 1997) and (Terabe et al., 1997) show how agents can learn the capabilities of others via repeated interactions, but these agents do not learn to predict what actions others might take. In fact, most of the work in multiagent learning also fails to recognize the possible gains from using explicit agent models to predict agent actions. Other examples are the studies of (Shoham and Tennenholtz, 1992), and (Glance, 1993) which focus on very simple but numerous agents and emphasize their emergent behavior. Meanwhile, (Hu and Wellman, 1996) show that learning agents sometimes converge to globally sub-optimal equilibria.

In general, the primary concerns in multi-agent learning research are: what happens when all agents in a MAS system start learning? and what can we do to improve their learning abilities? These concerns closely echo the problems addressed in this thesis. All the research we have found in multi-agent learning takes one of two approaches. It either involves the construction of learning agents (using some particular learning algorithm) and the analysis of experimental runs using these agents, or it deals with the construction of new "cooperative" learning algorithms which allow agents to share their knowledge. We are interested in the former approach. In Chapter 7 we show our attempt at building a community of learning agents in a market-based MAS and our analysis of the lessons learned. However, while this type of research does lead to a lot of insights within the particular domain, and with the particular learning algorithms, it typically provides little insight outside this area. That is, it is generally impossible to take results gathered from simulation in one domain and apply them to another domain. This is why, in this thesis, we have chosen to go beyond the traditional methods of multiagent learning and build a framework that characterizes the types of knowledge that an agent can have, as well as the learning abilities of the agents. With this framework we can analyze the way one agent's learning abilities affect the effectiveness of another agent's learning algorithm.

2.1.5 Agent-Based Modeling

A system composed of many learning (or adaptive) agents is, by definition, a complex adaptive system. These types of systems are common in the natural world (e.g. the economy, nature, human societies, the human immune system, etc.) and are currently the focus of research from groups studying *complex systems* (Cowan et al., 1995). The initial stages of

this research involved building simplistic models of a complex system and showing that the complexity is the result of simple interactions among the agents in the system. This line of research is often called **agent-based modeling** because instead of using global equations, the researchers build simple models of the agents and run these models on a computer. The complexity in the system is said to be an *emergent* property of the system.

Some of the earlier work in agent-based modeling was done by (Axelrod, 1984), (Axelrod, 1997). In it, Axelrod analyzed different agents' performance (against each other) in the Prisoner's Dilemma. Other work has studied biological systems (Kauffman, 1994), and simple economic systems (Hogg and Huberman, 1991), (Huberman and Clearwater, 1995). These studies, and others like them, aim at explaining complex behaviors based on simple interactions among many agents. In this thesis our emphasis is on predicting the system's expected behavior based on the not-so-simple behaviors of machine learning agents. Our aim is to avoid complexity, not to explain it. However, explanations of where complexity can arise from are very useful when building systems that try to avoid such complexity (or, at least, to keep it in check). We have also found it useful to apply the research methods used in agent-based modeling research to our problem. That is, one of our research methodologies involves implementing learning market-based MASs and then running the programs to see what behaviors arise. These experiments lead us to the discovery of several emergent behaviors, as we show in Chapter 7.

2.2 Machine Learning

The field of machine learning is a vast area of research, containing many different techniques and approaches. For a good introductory text to the field, including presentations of the machine learning techniques used in this thesis, we refer the reader to (Mitchell, 1997). Rather than try to summarize the whole field of machine learning in this section, we instead introduce and explain the particular machine learning algorithms that we use throughout this thesis.

The work in this thesis uses the techniques of reinforcement learning (Sutton, 1988) and a particular type of reinforcement learning called Q-learning (Watkins and Dayan, 1992). In reinforcement learning the agents learn a state-to-action mapping function via their experiences with the environment. Their action/learn loop starts with an agent taking an action. After the agent takes an action, it receives some evaluation of the effectiveness of that action and moves to a new state. The evaluation the agent receives usually takes the form of a payoff. The agent, however, is not explicitly told which action was the correct one to take, or even whether or not the action it took was the correct one. The task of the reinforcement learning algorithm is to use these payoffs in order to build a successful state-to-

action mapping that tells the agent the correct action to take in each state. The algorithms of reinforcement learning consist of utility functions that assign a utility value either to each state, or to each state-to-action mapping. These utilities are updated using an update function which takes into account the old utility, the payoff the agent just received, and a discounted measure of the agent's expected future payoffs from its current state. Different reinforcement learning algorithms have been developed for different domains. Each domain makes its own assumptions about the structure of the world that the agents inhabit (e.g., the new states might be dictated by the actions of the agents, by a probability function, or by a combination of both). Given the assumptions we make about the domain our agents inhabit (Chapter 3.1.1), the reinforcement learning equations we use are rather simple. The equations associate a utility with each state-and-action pair. The agent takes the action that has the highest utility for the current state, when it is exploiting its knowledge. When the agent is, instead, exploring its possibilities, it takes random actions. The utility of a state-action pair is updated after the action has been taken and a payoff has been received. The new utility is a weighted sum of the old utility plus the payoff the agent just received. The agents keep updating their utilities forever but, as time goes on, they perform less exploring and instead exploit their knowledge more.

We note that we are not the first to use these techniques for adaptive agents in a multi-agent environment, and that other techniques have also been used. An earlier study was done by (Lin, 1992), which shows a system which allows reactive agents to improve their performances by using, among other things, reinforcement learning. Other approaches, (Weiß, 1993) (Parker, 1993) (Sian, 1991) use their own learning algorithms and concentrate their efforts on how cooperative agents can learn from each other and, therefore, learn to cooperate more effectively. In these approaches there are some strong assumptions made about the willingness of the agents to cooperate and about their trustworthiness. There are also some efforts into building agents using genetic algorithms (Grefenstette, 1992).

Q-learning has been used by (Sen et al., 1994) for implementing agents, in a multi-agent setting, that try to optimize their behaviors. However, in this work, like in all the other work we have managed to find on adaptive agents, the agents never try to do any explicit modeling of the other agents. The agents simply learn to take those actions which give them the highest expected payoffs. The fact that these payoffs actually depend, in part, on the actions of other agents is not explicitly acknowledged. Instead, the behavior the agent learns reflects both the payoffs the agent expects to get, and the actions the others are expected to take. We believe that explicitly modeling the other agents will often lead to better performance because the agent will then have to learn simpler functions. That is, the agent might only need to learn to predict the action that another agent will take,

rather than learning what action to take based on some numeric reward. Explicit models of other agents will also be useful for modeling agents that have never been seen before but that, the agent believes, are similar to other agents it already has models of. In these cases the agent can simply reuse the old models and, that way, try to predict the new agent's actions. Finally, it is highly probable that the agent will be able to get models of the other agents either by direct programming (i.e., the models would be built into the agent from the start), or by communication with other agents.

Some of the work on the emergence of conventions (Shoham and Tennenholtz, 1992) can also be seen as a form of agent learning. In this work, an agent still does not model others but does engage in some form of learning. The designers of these system create rules of encounter for the agents that guarantee that the system will reach an equilibrium (i.e., the agents will agree on a convention). However, these results apply only to communities of agents all of which have the same design, and the same learning capabilities. We are more concerned with open systems where agents can be designed by anyone, to do anything, and in any way they deem it appropriate.

2.3 Summary

In this chapter, we have presented the research that this thesis builds upon. In the next chapters we extend agent theories, particularly RMM, in order to formally describe agents with nested models in a MAS. We then use ideas from limited rationality to solve the problem of exponential growth in the evaluation of nested agent models (without sacrificing the quality of the solution). After this, we consider agents with machine learning abilities. Their abilities are modeled by our framework for describing learning agents. This framework is then used to derive equations for predicting the expected behavior of a MAS with learning agents. Finally, we implement an economic society of learning agents and, in a manner similar to agent-based modeling, analyze the system's emergent behavior. The experiments provide us with a lot of insight into the agents' expected behavior. The experiments also allow us to confirm the predictions given by our framework, and to determine exactly which of the observed behaviors were too system-specific to be predicted by the theory but are nonetheless brought to light by our experiments.

CHAPTER 3

Formally Describing a Multi-Agent System and Recursive Modeling Agents

In order to build learning agents we first need to develop a formal framework for describing these agents and the MASs they inhabit. In this chapter we present such a framework. Our framework can describe MASs that have a finite number of agents, where each agent can take any one of a finite set of actions. The MASs must also use discrete time. And the actions taken by all agents, at each time step, are assumed to be simultaneous. The framework provides a tool for measuring the effectiveness of an agent's behavior within the MAS.

While the framework can be used to describe an agent's behavior, it does not say anything about how the agent is implemented. There are many ways to implement agents but, in this thesis, we have chosen to concentrate on agents that have recursive models of other agents. In order to formally describe these agents, we present notation used for representing agents with nested models. We will not, however, delve into an agent's machine learning abilities in this chapter. Instead, we talk about the agent's knowledge, and we assume that the agent is "born" with this knowledge or has some way of generating it as needed.

We start the chapter by motivating the need for a formal framework that can describe MASs. Section 3.1.1 describes the simplifying assumptions made by our framework. Section 3.1.2 describes how we represent the world the agents inhabit. An agent's actual behavior, the best possible behavior the agent can have, and a measure of how these two differ, are all captured by the functions presented in Section 3.1.3. While these functions are used to describe an agent's behavior, an agent need not be implemented with procedures that directly represent these functions. The agent's implementation might consist of various functions, or other more complex data structures. Specifically, we consider agents that have functions which explicitly model other agents recursively. We give a notation for describing these recursive modeling agents in Section 3.2. Finally, in Section 3.3 we show

how to measure the distance between a recursive modeling agent's behavior and the agent's best possible behavior.

In Chapter 5, we will expand the framework presented in this chapter. The extended framework will allow us to make predictions about the relative merits of different learning techniques when these techniques are used in a MAS composed of other learning agents.

3.1 A Framework for Modeling MASs

We developed this framework because we wanted a method for describing different multi-agent systems. After years working with several MASs, and studying many more, we found a set of characteristics that were common to many of them. All MASs we studied have a finite number of agents, each of whom has a finite set of actions it can take. Most systems use discrete time increments and assume that all agents' actions, taken at the same time step, are effectively taken in parallel. Finally, most of the agents take actions based on their current perceptions of the state of the world and the current states of their knowledge bases. After taking actions and observing the results that they produced, the agents change their knowledge bases in order to capture any new information.

Because the agents take actions based solely on the state of the world, we can define an agent's **behavior** as the mapping from states to actions that tells us what action the agent will take in each state of the world. We will use this definition of behavior throughout this thesis.

3.1.1 Simplifying Assumptions of Our Framework

Given our studies of existing MASs, we decided that it was reasonable to design a framework that can represent: a finite set of agents, a finite set of actions for each agent, and discrete time. We also determined that a general MAS framework will need to have both *world states* and *agents*. Finally, given the many different types of MASs in existence, it seems reasonable to expect a general framework to be able to describe MASs where:

1. All agents might only perceive part of the total world state and the actions of some subset of the other agents, e.g., part of the world state might be hidden from their view, or some of the other agents' actions might not be observable.

2. The next world state the agents see is somehow correlated to the previous state, and to the set of actions taken by all agents.

3. Some of the actions an agent can take are better than others, i.e., in each world state there is a preference ordering over the set of actions that an agent can take.

4. At any moment, an agent's action is dictated not only by the world state the agent is in, but also by some other factors, e.g., a coin flip, user input.

Unfortunately, in order to develop a theory that can make useful predictions about an agent's expected behavior correctness, we had to make some simplifying assumptions that partly simplify away the previous four desiderata. Namely, the framework we present in this chapter assumes that:

1. All agents perceive the same world state. All agents can observe everyone else's actions.

2. The next world state that the agents perceive is taken from a fixed probability distribution.

3. In each world state, there is only one correct action defined for each agent. The agents do not have preferences over the incorrect actions.

4. At any moment, an agent's action is dictated solely by the world state the agent is in.

The first assumption states that agents can perceive everything. This assumption excludes MASs were the agents cannot fully perceive the world or the actions of other agents. This assumption includes MASs where agents can observe everything around them, including the actions of others. One example of such a system is a simulation domain where the agents all have the same instruments (i.e., sensors) and the information returned by these instruments is the same for each agent, regardless of the location or status of the agent. Another example is an electronic commerce application where the actions of everyone are all broadcast. Since the world for these agents includes only the other agents, our assumption is satisfied. In general, if a system broadcasts all relevant information then we can expect all agents to have access to the same world state and to the set of actions taken by all other agents. Such broadcasting systems are common in networking environments were messages are often broadcast to everyone, especially when all the agents reside in the same local area network.

The second assumption states that the new world state the agents perceive at each time is picked at random, using some fixed probability distribution. This simplification precludes us from modeling systems where the next state of the world depends either on the previous one, or on the agents' actions. However, it does allow us to model episodic systems where agents experience one episode after another, all of them independent of each other. A good example is a market system, where the episodes are the goods being sold at each time. Another example is a coordination game, where the agents take an action at each time step

in the hope that, eventually, they will all choose the same action. In this game there is really only one world state, so our second assumption trivially holds.

The third assumption says that there exists a unique correct behavior, for each agent. It, therefore, precludes systems where one cannot identify a correct action. Notice that this assumption does not state that we must know what the correct action is for any agent. It merely states that a correct action must exist. There are some systems where it is not clear that a correct action exists (e.g., in the iterated Prisoner's Dilemma it is not clear whether an agent should cooperate or defect, or for how long (Axelrod, 1984)). But, our theory will still work for these system, as long as the designers are willing to define a given behavior as the correct one (e.g., in the Prisoner's Dilemma the designers might decree that the "correct" action is to cooperate). Still, in the vast majority of systems, we expect that there will exist a correct behavior for each agent, even if this behavior cannot be determined *a priori*.

The third assumption also states that the agents do not have any preferences over their incorrect actions. This assumption rules out systems where the agents have a different utility associated with each possible action, in each world. However, one can still apply our framework to such systems in order to get some useful predictions. The framework we present in this chapter (and expand in Chapter 5) measures the correctness of an agent's behavior based on the probability that the agent will take a correct action. Since we assume that there is only one correct action in each state, our correctness measure cannot capture the more general measure given by a utility function over the set of possible actions. Therefore, the predictions made by the framework will only reflect the agent's expected behavior as measured by the probability that the agent will take the action with the *highest* utility, i.e., the correct action. While this measure does not tell us the agent's expected utility, it is still a useful but limited measure of the agent's performance.

The fourth assumption states that, at any moment, an outside observer could model an agent's behavior as a simple world-state-to-action mapping. Because of this assumption, we can say that an agent's behavior is always defined as its current world-state-to-action mapping. This assumption does not preclude the use of agents with internal state because it leaves open the possibility that the agent's world-state-to-action mapping might change over time. Having this mapping change over time can be seen as a way for the agent to hold an internal state. For example, say we have an agent with an internal state that consists of the history for the last 5 moves. This agent chooses its actions based on this internal state and the current world state. At any moment, this agent's behavior can still be described with a state-to-action mapping. The reason this mapping can be done is that, at any time t, the agent's history will be a constant. Therefore, the action the agent takes will only

depend on this constant and on the state of the world. In effect, the only variable that the agent considers is the world state. Of course, the history will change for time $t + 1$, but this change simply leads to a new constant history. The new action will then depend on the new constant and the world state. In other words, the agent's choice of action, at any moment, is a function of just the world state. In Chapter 5 we study learning agents and their changes in behavior.

Our fourth assumption holds for most domains. The few domains we can think of where it does not apply include systems where the agents use a randomness generator or use some kind of external oracle for determining which actions to take. Also, if our first assumption does not hold for a particular system, then it might seem as if the fourth assumption also does not hold. That is, if the agents cannot all perceive the same world, then it might seem to an observer that some agent is taking actions that are not dictated by the current world state. This is a violation of the first assumption and should not be confused with violations of the fourth assumption, which are rare.

While these are limiting assumptions that curtail the wide applicability of our framework, we felt they were necessary assumptions to make because they allowed us generate predictions about the MASs we study. They are also, perhaps, not as limiting as they might at first appear. For example, in Chapter 7, we study a market-based MAS (modeled after a real working system) which was successfully modeled with this framework. In Chapter 8.1 we will examine the additional research, and the extra information, needed in order to relax these simplifying assumptions and some of the other assumptions we made about the use of discrete time and a finite set of actions.

3.1.2 The World and the Agents

We define a MAS as consisting of a finite number of agents, actions, and world states. We let N denote the finite set of agents in the system. W denotes the finite set of world states that can exist. Each agent is assumed to have a set of perceptors with which it can perceive the world (e.g., a camera, microphone, bid queue, etc.). An agent maps each possible perception it can have into some actual world state w; the set of all these states is W. A_i, where $|A_i| \geq 2$, denotes the finite set of actions agent $i \in N$ can take. We assume discrete time, indexed in the various functions by the superscript t, where t is an integer greater than or equal to 0.

3.1.3 Describing Agent Behavior

In the types of MASs we are modeling, every agent i perceives the state of the world w and takes an action a_i, at each time step. As we stated in Section 3.1.1, we assume

N the set of all agents, where $i \in N$ is one particular agent.

W the set of possible states of the world, where $w \in W$ is one particular state.

A_i the set of all actions that agent i can take.

$\delta_i : W \to A_i$ the **decision** function for agent i. It tells which action agent i will take in each world.

$\Delta_i : W \to A_i$ the **target** function for agent i. It tells us what action agent i should take. It takes into account the actions that other agents will take.

$e(\delta_i) = \mathbf{Pr}[\delta_i(w) \neq \Delta_i(w) \,|\, w \in \mathcal{D}]$ the **error** of agent i. It is the probability that i will take an incorrect action, given that the worlds w are taken from the fixed probability distribution $\mathcal{D}$.

Figure 3.1: Summary of notation used for describing a MAS and the agents in it.

that every agent's behavior can be described with a simple state-to-action mapping. We also assume that there is a correct behavior for each agent. An agent's *target* behavior is composed of all of the agent's correct state-to-action mappings. In order to determine the target behavior for an agent, one would generally need to know, for each $w \in W$, the set of actions that all the other agents are going to take in that w. For example, in order for you to determine what an agent should bid in every world w of an auction-based market system, you will need to know what the other agents will bid in every world w. Luckily, we will not always need to derive specific target behaviors, since our results and predictions will be based on our measure of the correctness of an agent's behavior.

More formally, we say that each agent's behavior is represented by a **decision function**, given by $\delta_i : W \to A_i$ for agent i. This function maps each state $w \in W$ to the action $a_i \in A_i$ that agent i will take in that state.

The action that agent i *should* take in each state w (i.e., the correct action for each state w) is given by the **target function** $\Delta_i : W \to A_i$, which also maps each state $w \in W$ to an action $a_i \in A_i$. Since the choice of action for agent i often depends on the actions of other agents, the target function for i needs to take these actions into account. That is, in order to generate a target function for i, one would need to know $\delta_j(w)$ for all $j \in N_{-i}$ and $w \in W$. These $\delta_j(w)$ functions tell us the actions that all the other agents will take

in every state w. One can use these actions, along with the state w, in order to identify the best action for i to take. An agent does not usually have direct access to its target function, and the target function is not part of the agent. In fact, in Chapter 5 we will show how the agent's learning task is to change its decision function so that it matches the target function. This learning is done with the use of payoffs which the agent receives after taking each action. In Chapter 5 we also show how the target function can change over time because of changes in the other agents' decision functions.

Our measure of the correctness of an agent's behavior is given by our **error** measure. We define the error of agent i's decision function δ_i as:

$$\begin{aligned} e(\delta_i) &= \sum_{w \in W} \mathcal{D}(w) \mathbf{Pr}[\delta_i(w) \neq \Delta_i(w)] \\ &= \mathbf{Pr}_{w \in \mathcal{D}}[\delta_i(w) \neq \Delta_i(w)] \end{aligned} \tag{3.1}$$

where $\mathcal{D}(w)$ is a fixed probability distribution from which the worlds seen at each time are taken. The assumption that the new worlds are taken from a fixed probability distribution was discussed in Section 3.1.1. $e(\delta_i)$ gives us the probability that agent i will take an incorrect action. $e(\delta_i)$ is the measure we use to gauge how well agent i is performing. An error of 0 means that the agent is taking all the actions dictated by its target function. An error of 1 means that the agent never takes an action as dictated by its target function. All this notation forms our framework for describing MASs. The notation is summarized in Figure 3.1. We expand this framework on Chapter 5.

3.2 Recursive Modeling Agents

While we can use the framework presented in Section 3.1 to describe a MAS, along with an agent's behavior (δ_i) and the correctness of this behavior ($e(\delta_i)$), the framework does not tell us how the agent is implemented. For example, the agent could be implemented with a neural network that learns to match worlds w to high rewards, or it could have an oracle that tells it the actions that everyone else will take which the agent then uses for determining what action to take, or it could be implemented as a lookup table that matches each w to some particular a_i, etc. There are many different ways to implement an agent. However, in this thesis we focus our attention on agents that keep explicit models of other agents, and use these models to determine what action to take.

The use of modeling levels (Gmytrasiewicz, 1992) is an intuitive and useful way for the agent to split its knowledge. That is, an agent might be implemented as an instantiation of a simple decision function $\delta_i(w)$, or the agent might model other agents as using a decision function and use the predictions from these functions to determine what action to take, or the agent might use an arbitrarily deep nesting of functions. We call these k-level agents,

Level	Types of knowledge.
0-level	$\delta_i(w)$
1-level	$\delta_i(w, \vec{a}_{-i})$
	$\delta_{ij}(w)$
2-level	$\delta_i(w, \vec{a}_{-i})$
	$\delta_{ij}(w, \vec{a}_{-j})$
	$\delta_{ijk}(w)$

Table 3.1: Decision functions possessed by different k-level agents.

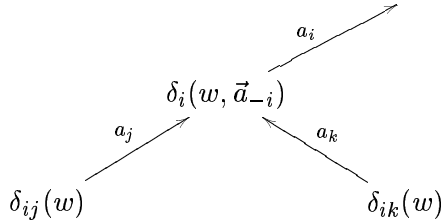


Figure 3.2: 1-level agent i determines what action to take.

where $k \geq 0$ and refers to the level of nesting that the agent uses.

3.2.1 0-level agents

We define a 0-level agent as an agent that is not capable of recognizing the fact that there are other agents in the world. The only time the other agents' presence affects a 0-level agent is when their actions lead to changes in the payoff the 0-level agent receives. A 0-level agent i is implemented with a procedure that directly instantiates the decision function $\delta_i(w)$. This function captures all the knowledge the agent has.

3.2.2 1-level agents

A 1-level agent i recognizes the fact that there are other agents in the world and that they take actions, but it does not know anything more about them. Given these facts, the 1-level agent's strategy is to predict the other agents' actions based on his models of them, and use these predictions when trying to determine its best action. A 1-level agent assumes that the other agents pick their actions by using a mapping from W to A. A 1-level agent i is, therefore, implemented with procedures that directly instantiate the functions $\delta_i(w, \vec{a}_{-i})$, and $\delta_{ij}(w)$ for all agents $j \neq i$. The agent's behavior can be described by the function

$$\delta_i(w) = \delta_i(w, \vec{a}_{-i}) \tag{3.2}$$

where

$$\vec{a}_{-i} = \{\delta_{ij}(w) \mid j \in N_{-i}\} \tag{3.3}$$

In other words, a 1-level agent's behavior can be described by a decision function that is formed by the following composition of the agent's decisions functions:

$$\delta_i(w) = \delta_i(w, \{\delta_{ij}(w) \mid j \in N_{-i}\}) \tag{3.4}$$

An example 1-level agent i modeling two other agents j and k can be seen in Figure 3.2. Here we see agent i's functions, which include its models of agents j (δ_{ij}) and k (δ_{ik}). δ_{ij}, for example, is a function that tells us what i *thinks* that j will do in every state w. This action need not be the one j takes, since i's model of j could be incorrect. That is, it is not necessary that $\delta_{ij}(w) = \delta_j(w)$ for all $w \in W$. When i needs to determine which action to take, it first evaluates its models of j and k in order to determine what actions it thinks they will take, i.e., a_j and a_k in Figure 3.2. These actions then form the vector $\vec{a}_{-i} = \{a_j, a_k\}$. Since we now have values for w and for $\vec{a}_{-i}$, we can evaluate the function $\delta_i(w, \vec{a}_{-i})$ for these values in order to get the action that i will take, i.e., a_i in Figure 3.2.

3.2.3 2-level agents

A 2-level agent i also recognizes the other agents in the world and, in addition, has some information about their decision processes and previous observations. That is, a 2-level agent has beliefs about the other agents' internal procedures used for picking an action. This model of the others allows a 2-level agent to dismiss "useless" information when picking its next action, e.g., i might know that j takes the same action no matter what it thinks others will do. We say that a 2-level agent is implemented by procedures that directly implement the three decision functions $\delta_i(w, \vec{a}_{-i})$, $\delta_{ij}(w, \vec{a}_{-j})$, and $\delta_{ijk}(w)$. $\delta_{ij}(w, \vec{a}_{-j})$ captures what agent i thinks j will do given that both of them are in state w and j believes all other agents will take actions given by $\vec{a}_{-j}$. $\delta_{ijk}(w)$ captures what i thinks j thinks k will do in state w, where $i \neq j \neq k$. Notice that, in $\delta_{ijk}(w)$, it is possible that $i = k$, i.e., I can have a model of your model of me. A 2-level agent's behavior can be described with the decision function

$$\delta_i(w) = \delta_i(w, \{\delta_{ij}(w, \{\delta_{ijk}(w) \mid k \in N_{-j}\}) \mid j \in N_{-i}\}) \tag{3.5}$$

A simple way a 1-level agent can become 2-level is by assuming that "others are like him", and modeling others using the same decision functions and observations the agent, itself, was using when it was a 1-level agent. This type of process will, of course, only be effective when the other agents are indeed similar to the modeling agent. If that is the case, then this method can be used to "bootstrap" an agent to any modeling level.

3.2.4 k-level agents

We can keep defining agents with deeper models in a similar way. In a world with $|N|$ agents, a k-level agent i_1, where $k > 2$ would have the following functions:

$$\delta_{i_1}(w, \vec{a}_{-i_1}) \tag{3.6}$$

$$\bigcup_{i_2 \in N_{-i_1}} \delta_{i_1 i_2}(w, \vec{a}_{-i_2}) \tag{3.7}$$

and so on, until

$$\bigcup_{i_{k+1} \in N_{-i_k}} \cdots \bigcup_{i_3 \in N_{-i_2}} \bigcup_{i_2 \in N_{-i_1}} \delta_{i_1 i_2 \cdots i_{k+1}}(w) \tag{3.8}$$

where we use the union operator to form a set of functions. That is, a union of functions is just the set of such functions.

3.3 Calculating the Error for Recursive Modeling Agents

In Section 3.1.3 we showed how to calculate an agent's error given its decision and target functions, using Eq. (3.1). However, this equation can only be directly applied to a 0-level agent. Higher level agents, as we have just shown, generate their $\delta_i(w)$ decision functions by composing several other functions. Since these nested functions can be collapsed into a $\delta_i(w)$ function, their error could be calculated by performing this collapse and using $\Delta_i(w)$. If, on the other hand, $\Delta_i(w)$ is not available but the errors for each of the nested functions are available, then we can approximate the error for the top-level nested function from the errors of each of the k-level nested functions. In this section we show how this approximation can be accomplished.

3.3.1 Total Error for 1-level Agent

We start by remembering that the decision function for a 1-level agent i is generated by the composition:

$$\delta_i(w) = \delta_i(w, \{\delta_{ij}(w) \,|\, j \in N_{-i}\}) \tag{3.4}$$

and therefore

$$e(\delta_i) = \mathbf{Pr}_{w \in \mathcal{D}}[\delta_i(w, \{\delta_{ij}(w) \,|\, j \in N_{-i}\}) \neq \Delta_i(w)] \tag{3.9}$$

We can also calculate the errors for the two functions of a 1-level agent.

Calculating $e(\delta_i(w, \vec{a}_{-i}))$

If the agent already has a correct $\delta_i(w, \vec{a}_{-i})$ function then we have simply $e(\delta_i(w, \vec{a}_{-i})) = 0$. Otherwise, the error is given by:

$$e(\delta_i(w, \vec{a}_{-i})) = \mathbf{Pr}_{w \in \mathcal{D}}[\delta_i(w, \{\delta_j(w) \mid j \in N_{-i}\}) \neq \Delta_i(w)] \tag{3.10}$$

Notice that instead of using $\delta_{ij}(w)$, as in Eq. (3.9), we use $\delta_j(w)$—the actions the agents will take. We use $\delta_j(w)$ because, in order to calculate the error in $\delta_i(w, \vec{a}_{-i})$ we need to know the actual actions that the other agents will take, not the actions that i thinks they will take. The error $e(\delta_i(w, \vec{a}_{-i}))$ is the probability that $\delta_i(w, \vec{a}_{-i})$ leads i to take an incorrect action, even when $\vec{a}_{-i}$ (i.e., i's prediction about what all the other agents will do) is correct. This error does not address the case where $\vec{a}_{-i}$ is incorrect. We deal with this case later in this section. In order to use Eq. (3.10), for some agent i, we would need to know either, $\delta_j(w)$ for all other agents $j \neq i$, or we would need to know the action i should take given any possible combination of all the other agents' actions, in all possible world states.

Error in $e(\delta_{ij}(w))$

This error is given by:

$$e(\delta_{ij}(w)) = \mathbf{Pr}_{w \in \mathcal{D}}[\delta_{ij}(w) \neq \delta_j(w)] \tag{3.11}$$

This error is the probability that i's model of j will lead i to make an incorrect prediction as to what action j will take. It is simply the probability that the prediction that i makes about the action that j will take is incorrect. In other words, j takes a different action from the one i predicted. By its very definition, we can see that the calculation of this error requires us to know both $\delta_j(w)$, as well as $\delta_{ij}(w)$, in the same way that the calculation of an agent's error requires us to know the agent's decision and target functions.

Total 1-level error

Using the errors given by Eq. (3.11) and Eq. (3.10), we can bound $e(\delta_i)$ as such:

$$e(\delta_i) \leq 1 - (1 - e(\delta_i(w, \vec{a}_{-i}))) \prod_{j \in N_{-i}} (1 - e(\delta_{ij})) \tag{3.12}$$

This equation makes a couple of independence assumptions. The first assumption is that $e(\delta_{ij})$, for any one agent j, is independent of $e(\delta_{ik})$, for any other agent $k \neq j$. The second assumption is that $e(\delta_{ij})$ is independent of $e(\delta_i(w, \vec{a}_{-i}))$. In other words, Eq. (3.12) assumes the errors for the different decision functions are all independent of each other.

The bound given by Eq. (3.12) ignores the fact that errors in the $\delta_{ij}(w)$ can be masked by the $\delta_i(w, \vec{a}_{-i})$ knowledge such that they do not change the final decision function. A

better approximation, therefore, is:

$$\begin{aligned} e(\delta_i) \approx (1 - e(\delta_i(w, \vec{a}_{-i})))d(\delta_i(w, \vec{a}_{-i}))(1 - \prod_{j \in N_{-i}} (1 - e(\delta_{ij}))) \\ + e(\delta_i(w, \vec{a}_{-i})) \prod_{j \in N_{-i}} (1 - e(\delta_{ij})) \end{aligned} \tag{3.13}$$

where we define

$$d(\delta_i(w, \vec{a}_{-i})) = \mathbf{Pr}_{(w, \vec{a}_{-i}) \in \mathcal{S}}[\delta_i(w, \vec{a}_{-i}) \neq \delta_i(w, \vec{a}^*_{-i}(w))] \tag{3.14}$$

Here $\mathcal{S}(\{w, \vec{a}_{-i}\})$ is a probability density function over all possible $\{w, \vec{a}_{-i}\}$ pairs, and $\vec{a}^*_{-i}$ is the vector of the actions that the other agents will take.

We call $d(\delta_i(w, \vec{a}_{-i}))$ the **dampening** value of $\delta_i(w, \vec{a}_{-i})$. It is the probability that $\delta_i(w, \vec{a}_{-i})$ will dampen-out errors that occur in the deeper nested functions, i.e., those that determine the actions that go into the $\vec{a}_{-i}$ parameter of $\delta_i(w, \vec{a}_{-i})$. The first term in Eq. (3.13) multiplies this dampening value with the error in the nested functions, and with $1 - e(\delta_i(w, \vec{a}_{-i}))$, in order to determine the probability that an error in the deeper nested functions could have propagated to the top. The second term in Eq. (3.13) determines the probability that the error that gets propagated to the top was actually in $\delta_i(w, \vec{a}_{-i})$. In Eq. (3.13) we are using the approximation that, if both $\delta_i(w, \vec{a}_{-i})$ and $\delta_{ij}(w)$ are wrong, then $\delta_i(w)$ will be wrong (i.e., we are ignoring the possibility that two wrongs could make a right).

For example, assume that there are only two agents i and j. We could then simplify Eq. (3.13) to be:

$$\begin{aligned} e(\delta_i) \approx d(\delta_i(w, \vec{a}_{-i}))e(\delta_{ij}(w))(1 - e(\delta_i(w, \vec{a}_{-i}))) \\ + e(\delta_i(w, \vec{a}_{-i}))(1 - e(\delta_{ij}(w))) \end{aligned} \tag{3.15}$$

We can use these techniques for calculating the errors of an k-level agent as functions of the errors of each one of the nested decision functions. However, in order to calculate these errors we would need access to the appropriate target functions. These functions will be readily available for some MASs but not for others; it depends on the type of system and the information available to the agents. For example, in an auction-based market system, the designer usually knows the correct $\delta_i(w, \vec{a}_{-i})$—the designer knows what his agent should do given that he knows what state it is in and the actions that all the other agents are going to take.

On the other hand, the value of $d(\delta_i(w, \vec{a}_{-i}))$ is not easy to calculate since it depends on the probability distribution of the other agents' actions and worlds $(w, \vec{a}_{-i}) \in \mathcal{S}$. This information can only be gathered through repeated interactions with other agents. Luckily,

we can make use of the presence of dampening knowledge without having to calculate the actual value of the dampening. In the next chapter we present an algorithm that a recursive modeling agent can use for determining which of its nested functions to use. The algorithm achieves this feat by exploiting the presence of dampening in the agent's decision functions. The algorithm calculates the expected actions of the different agent models and uses these predictions for determining whether or not a more detailed model of a particular agent should be generated.

3.4 Summary

As we have gained experience with the framework and notation presented in this chapter, we have found the division of knowledge used by the recursive agents to be especially useful for characterizing the knowledge that these agents possess. While this division was inspired by RMM, we believe that our use of functions instead of payoff matrices makes our recursive agent models applicable to a wider variety of research. By categorizing agents as k-level agents, researchers can better understand the amount of knowledge they have put into an agent since our notation makes explicit the knowledge about the agent, about the agent's models of other agents, about the agent's models of other agents' models of others, and so on. We also believe that splitting the knowledge in this way is a natural thing for designers to want to do.

In fact, this split has already been used by many researchers. For example, (Claus and Boutilier, 1997) study Independent Learners (ILs) and Joint Action Learners (JALs), concentrating on how to implement Q-learning in these agents. After studying their work, we have determined that their IL agents are 0-level agents using Q-learning, while their JAL agents are 1-level agents that are learning both $\delta_i(w, \vec{a}_{-i})$ and $\delta_{ij}(w)$. In (Parkes and Ungar, 1997) the authors define Myopic-learning agents, which are akin to our 1-level agents, and Strategic-learning agents, which are very similar to our 2-level agents. Finally, (Hu and Wellman, 1998) have decided to use our definition of k-level agents as a way to guide their research into agent learning in market systems. In all these examples, the mapping from our definitions of k-level agents to their usage in the particular examples fails in some minor ways. For example, some of the agents have knowledge that cannot be fully captured by one of our functions. One such example is given by the Strategic-learning agents from (Parkes and Ungar, 1997). While these agents are very similar to our 2-level agents, they nonetheless incorporate a type of long-term strategic thinking that cannot be captured by our framework. A Strategic-learning agent takes into account the effects that its actions will have on the other agents' model of it, something which a normal 2-level agent does not do. Still, even in the presence or near-misses such as this one, the mappings to our recursive

agent modeling notation are fairly successful, and could serve as way for researchers to arrive at a common language for describing, at least at a high level, the types of knowledge that their agents have.

We defined an agent's error as a function over the error of the agents' different nested models. The concept of error we use comes directly from research in computational learning theory; we have merely adapted it for our use and calculated its value for recursive modeling agents. In doing this calculation, we defined the concept of knowledge dampening, which tells us how likely it is that changes in the actions chosen below some decision function will lead to changes in the action returned by the decision function. The concept of knowledge dampening applies only to nodes in trees formed by nested decision functions. It does not apply to nodes in other types of search trees, such as minimax trees, where the nodes are not decision functions. Dampening is a basic characteristic of knowledge and it is very useful to be able to explicitly talk about it. Unfortunately it is not always possible to directly estimate its value, so other methods must be developed to exploit the presence of knowledge dampening. In the next chapter we present one such method.

CHAPTER 4

An Algorithm for Exploiting Dampening Effects

In the last chapter we presented a framework for describing MASs, as well as some notation that a designer can use for structuring the knowledge that his agent possesses. The notation we presented allows us to represent the agent's models of other agents, and its models of other agents' models of others, and so on. We also mentioned that the cost of generating and maintaining these deeper models grows exponentially with the depth of nesting and with the number of agents. Luckily, the presence of knowledge dampening gives us a possible way out of this problem. Having a top-level model with dampening means that the actions predicted by the models below this one are less likely to provoke a change in the action returned by the top-level model. The bigger the dampening value, the less likely it is that the action returned by the top-level model will change because of changes in the actions predicted by the models below. This means that we can safely ignore the models of other agents that appear below a model with a high degree of knowledge dampening. For example, if my model of Sally tells me that she will take action a every time, then I do not need to worry about what she thinks others will do.

In this chapter we present an algorithm that takes advantage of dampening knowledge in order to reduce the number of deeper models that an agent must consider when deciding on which action to take. This reduction in the number of models translates into an increase in the speed at which the agent determines which action to take. We start by extending the notation for describing recursive modeling agents presented in Chapter 3.2. Please note that the extended notation we present in this chapter is not an extension of our framework for modeling MASs (Chapter 3.1), but is instead built as an addition to the notation for recursive modeling agents. The extended notation is based on the idea of *situations*, which encompass the physical state of the world that the agents are in (w), the mental states of all the agents, and the strategies and expected strategies associated with these other agents and their respective situations. This extended notation allows us to rigorously define the *gain* of continuing deliberation versus taking action. Since the gain can only be evaluated after

generating the model, and we need to know it beforehand, we had to develop a method for determining the expected gain of generating a model. This expected gain is used to guide the pruning of the nested model structure (i.e., the situation). We have implemented our approach on a canonical multi-agent problem, the Pursuit Task, to illustrate how real-time, multi-agent decision-making can be based on a principled, combinatorial model. Test results show a marked decrease in deliberation time while maintaining a good performance level.

4.1 The Recursive Modeling Method

The basic modeling primitives we use are based on the Recursive Modeling Method (RMM) (Gmytrasiewicz, 1996) (Gmytrasiewicz and Durfee, 1995) (Durfee et al., 1994) (Durfee et al., 1993) which we introduced in Chapter 2.1.2. RMM provides a theoretical framework for representing and using the knowledge that an agent has about its expected payoffs and those of others. To use RMM, an agent is expected to have a payoff matrix where each entry represents the payoffs the agent expects to get given the combination of actions chosen by all the agents. Typically, each dimension of the matrix corresponds to one agent, and the entries along it correspond to the actions that agent can take. The agent can (but need not) recursively model others as similarly having payoff matrices. In fact, the agent can recursively nest these agents' models to any arbitrary depth. The recursive modeling only ends when the agent has no deeper knowledge. At this point, a Zero Knowledge (ZK) strategy is attributed to the particular agent. If agent i models agent j with a ZK strategy, this means that i believes that j does not know what action to take. If, on the other hand, an agent does have reason to believe some actions are more likely than others, then this different probability distribution can be used (Gmytrasiewicz, 1996). RMM provides a method for propagating strategies from the leaf nodes to the root. The strategy derived at the root node is what the agent performing the reasoning should do.

4.1.1 Mapping our Recursive Modeling Agent Notation to RMM

We can map our recursive modeling agent notation from Chapter 3.2 to include RMM's payoff matrices by simply defining our decision functions as returning the actions that maximize the agent's payoff in RMM's matrix. That is, if agent i's payoff matrix is given by $U_i(w, \vec{a})$ (where $\vec{a}$ is a vector of all agents' actions) then the agent's decision function is:

$$\delta_i(w) = \arg\max_{a_i \in A_i} U_i(w, \{a_i, a_j, \ldots, a_k\}) \tag{4.1}$$

where $a_j, \ldots, a_k$ are the actions that agent i thinks the other agents will take. We also extend the concept of "action" to include the *mixed strategy*. A mixed strategy, as defined

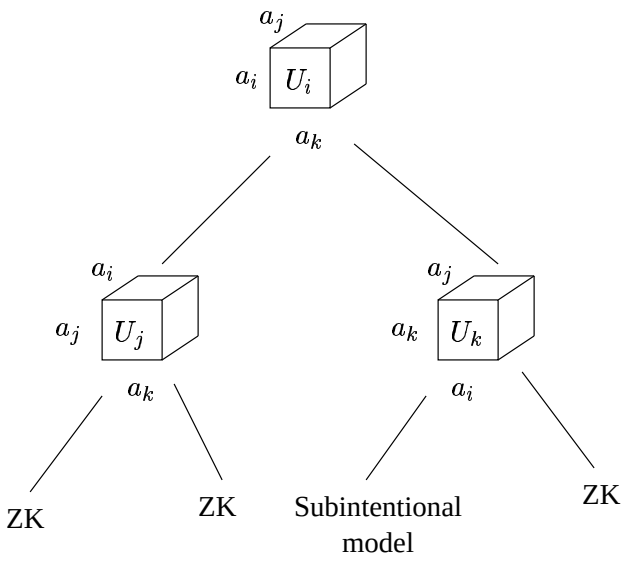


Figure 4.1: This is an example RMM hierarchy for three agents i, j, and k. The leaves of the tree will either be the Zero Knowledge (ZK) strategy or a sub-intentional model.

in game theory (Shubik, 1985), is simply a probability vector over the set of actions that an agent can take. For example, instead of just saying that agent i takes action a_i^1, a mixed strategy could say that i takes action a_i^1 with probability .8 and action a_i^2 with probability .2. The ZK strategy is another example of a mixed strategy since it says that an agent will take each action available to it with equal probability. We distinguish between a mixed strategy and our definition of an agent's behavior, by noting that a behavior is a mapping from every state to the action taken in that state, while a strategy only tells us about the action an agent will take in one particular state.

An example payoff matrix, along with its RMM hierarchy, is shown in Figure 4.1. This RMM hierarchy represents a situation from agent i's point of view and, therefore, has its payoff matrix U_i at the root. We also show i's model of j as the matrix U_j, and i's model of k as the matrix U_k. Notice that, instead of detailing the values inside the payoff matrices as we have been doing for RMM hierarchies, we have instead used variables to represent the matrices.

4.2 The Pursuit Task

In order to evaluate the algorithm we present in this chapter, we have automated the construction of payoff matrices for the *pursuit task*, in which four predators try to surround

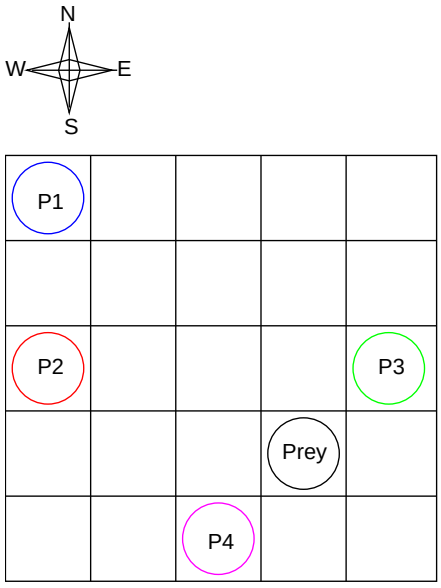


Figure 4.2: The Pursuit task. Four predators (P1, P2, P3, P4) try to capture the Prey.

a prey which moves randomly. The agents move simultaneously in a two-dimensional square grid. They cannot occupy the same square, and the game ends when either all four predators have surrounded the prey on four sides ("capture"), when the prey is pushed against a wall so that it cannot make any moves ("surround"), or when time runs out ("escape"). The pursuit task has been investigated in Distributed AI (DAI) and many different methods have been devised for solving it (Korf, 1992) (Stephens and Merx, 1990) (Levy and Rosenschein, 1992) (Gasser and Huhns, 1989). These methods either change the problem definition in order to make it more tractable, impose specific roles on the predators, spend much time computing, or fail because of lack of coordination. RMM provides another method for providing this coordination but, to avoid the time-consuming generation of nested models, we must devise a theory to support selective expansion of the hierarchy.

4.3 Situations

Our notation from Chapter 3.2 dealt with the different types of nested models that an agent might have. We now expand that notation so that we can describe which specific models the agent is using and describe an approximation function associated with each specific model. The algorithm we present in Section 4.4 is based on ideas from limited rationality. Our notation closely parallels Russell and Wefald's work (Russell and Wefald,

1991) in this area. Their research was described in Chapter 2.1.3. The differences between their algorithms and ours will be detailed later in Section 4.3.1.

The basic unit in our extended notation for recursive agent modeling is the **situation**. At any point in time, an agent is in a situation, and all the other agents present are in their respective situations. An agent's situation contains, not only the physical situation the agent thinks it is in, but also the situations the agent thinks the others are in (these situations might then refer to other situations, and so on). In this thesis, we have adopted RMM's assumption that common knowledge[1] cannot arise in practice. Common knowledge would be represented in our extended notation by having situations refer back to themselves either directly or transitively. Conversely, cycles or loops of reference between situations imply the presence of common knowledge. They are, therefore, disallowed since common knowledge cannot be achieved in many common situations of interest to us, such as in asynchronous message-passing systems (Fagin et al., 1995). Allowing agents to jump to assumptions about common knowledge can simplify coordination reasoning, since agents can use fixed-point equilibrium notions and axioms based on mutual beliefs and plans. While leaping to such assumptions can, at times, be a viable method for keeping coordination practical (Durfee, 1995), it also introduces risks that we wish to avoid.

A situation has both a physical and a mental component. The physical component refers to the physical state of the world (w) and the mental component refers to the mental state of the agent, i.e., what the agent thinks about itself and about the other agents around it. Intuitively, a situation reflects the state of the world from some agent's point of view by including what the agent perceives to be the physical state of the world and what the agent is thinking. A situation evaluates to a mixed strategy—a probability distribution over the set of actions A_i that agent i can take.

Let S be the set of situations an agent might encounter, and N the set of all relevant agents. A particular situation $s \in S$ that agent i is in, is recursively defined as:

$$s = (U_i(w, \vec{a}), f, w, \{(p, r, j) \mid \sum p = 1, r \in (S \cup \mathrm{ZK} \cup L), j \in N_{-i}\}) \tag{4.2}$$

The matrix U_i has the payoff the agent, in situation s, expects to receive for each combination of actions that all the agents might take. The matrix U_i can either be stored in memory as is, or it can be generated by a procedure (which is stored in the agent's memory) that takes as inputs any relevant aspects of the physical world, and previous history. The relevant aspects of the physical world are stored in w, the *physical* component of the situation. The rest of s constitutes the *mental* component of the situation which includes the agent's models of other agents.

[1] Common knowledge about fact x means that everyone knows x, and everyone knows that everyone knows x, and everyone knows that everyone knows that everyone knows x, and so on.

The function $f(x)$ gives the probability that the strategy x is the one that the situation s will evaluate to. It need not be a perfect probabilistic distribution; it is merely used as a useful approximation for guiding our search through the recursive models. That is, our algorithm will employ it as a heuristic for determining which recursive models to expand. We use $f(x)$ to calculate the strategy that the agent in the situation is expected to choose, using standard expected value formulas from probability theory (Luce and Raiffa, 1957). The value of this function is usually calculated from previous experience. In practice, some simplifying assumptions can be made about how to calculate (i.e., learn) this value, as we shall explain in Section 4.4.2. We also note that the knowledge encompassed by $f(x)$ is not the real knowledge, which is contained in the matrices, but is the control knowledge used only for guiding the search through the recursive models. There is no guarantee that the predictions produced by $f(x)$ will be correct. The models, on the other hand, are always considered to be correct.

A situation s also includes the set of situations which the agent in s believes the other agents are in. Each agent j is believed to be in r, with probability p. The value of r can be either a situation ($r \in S$), or a sub-intentional model $l \in L$, or, if the modeling agent has no more knowledge, it can be the Zero Knowledge strategy ($r = ZK$).

A situation s expands to a tree, as shown in Figure 4.3(a). This tree can be evaluated in order to determine the action the agent in this situation will take. That is, given that agent i is in a situation s associated with world w, the action it takes in that situation (and, therefore, the world w associated with it) can be described by evaluating the situation it is in:

$$\delta_i(w) = \text{Evaluate}(s) \tag{4.3}$$

`Evaluate`(s) is a procedure that evaluates situation s by propagating the strategies at its leaf nodes all the way to the root payoff matrix. This propagation is done in the standard way used by RMM. The procedure returns the action that the agent in s will take. Notice that, although in general a situation evaluates to a strategy, the `Evaluate` procedure is guaranteed to return an action since the agents can only take one of their five actions, i.e., a predator cannot decide to go half North and half West. As we showed in Chapter 2.1.2, the evaluation of an RMM hierarchy can only result in a single action or in an equal-probability strategy over the set of actions. If s evaluates to a single action then `Evaluate` simply returns this action. If, on the other hand, s evaluates to the equal-probability strategy then `Evaluate` chooses one of the actions and returns it. Since, in this case, all the actions are considered equally good, it does not matter which action is chosen.

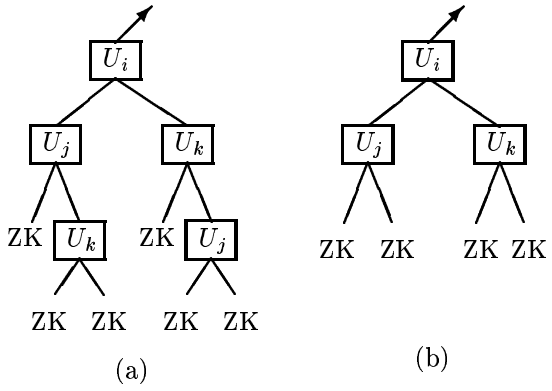


Figure 4.3: These are graphical representations of (a) a situation s (the agent in s expects to get the payoffs given by U_i and believes that its opponents will get payoffs based on U_j and U_k, respectively), and (b) a partially expanded situation t that corresponds to s.

4.3.1 Partially expanded situations

We define a *partially expanded situation* as a subset of a situation where only some of the nodes have been expanded. That is, if t is a partially expanded situation of s, then it must contain the root node of s, and all of t's nodes must be directly or indirectly connected to this root by other nodes in t. The situation t might have fewer nodes than s. At those places where the tree was pruned, t will insert the zero knowledge strategy, as shown in Figure 4.3.

We also define our basic computational action to correspond to the expansion of a leaf node l in a partially expanded situation t, plus the propagation of the new strategy up the tree. The expansion of leaf node l corresponds to the creation of the matrix U, and its placement in the position where l was. The children of the new matrix are set to the ZK strategy because we have already spent some time generating U and we do not wish to spend more time calculating a child's matrix, which would be a another step or computational action. We also do not want to set the children to their expected strategy because these expected strategies are not the true knowledge that the agent has. The expected strategies are heuristic knowledge and have errors associated with them. Setting all the leaves to the expected strategies would compound the errors, reducing the quality of the strategy that t evaluates to.

The strategy that results from propagating the ZK strategies past M is then propagated up the tree of t, so as to generate the new strategy that t evaluates to. The whole process, therefore, is composed of two stages. First we Expand l and then we Propagate the new

t	Partially expanded situation for agent i. It corresponds to the fully expanded situation s, associated with world w.
$t.a_i$	The strategy the agent i currently favors in the partial situation t. $t.a_i = \text{Evaluate}(t) = \delta_i(w)$.
$t.a_j$	The strategy that the agent i believes its opponent $j \in N_{-i}$ will favor in the partial situation t.
$t.\text{leaves}$	These are the leaf nodes of the partially expanded situation t that can still be expanded (i.e., those that are not also leaf nodes in s).
$t.\hat{a}_i^l$	The strategy agent i expects to play given that, instead of actually expanding leaf l (where $l \in t.\text{leaves}$) it simply uses its expected value, which it got from $f(x)$, and propagates this strategy up the tree.
$t.\hat{a}_j^l$	The strategies the agent expects another agent $j \in N_{-i}$ to play given that, instead of actually expanding leaf l, it simply uses its expected value.
$U_i(t.a_i, t.a_j \cdots t.a_n)$	This is the payoff agent i gets in its current situation. To calculate it, use the root matrix and let the agent's strategy be $t.a_i$ while the opponents' strategies are $t.a_j \cdots t.a_n$.

Figure 4.4: Summary of notation for partially expanded situations.

strategy up t. We will call this process Propagate_Expand(t, l), or $PE(t, l)$ for short. The procedure $PE(t, l)$ is our basic computational action.

We define the time cost for doing $PE(t, l)$ as the time it takes to generate the new matrix U, plus the time to propagate a solution up the tree, plus the time costs incurred so far. This time cost is given by:

$$\begin{aligned} TC(t,l) \quad = \quad & (c \cdot \text{size of matrix } U \text{ in } l) \; + (d \cdot \text{propagate time}) \\ & + g(\text{time transpired so far}) \end{aligned}$$

where c and d are constants of proportionality. The function $g(x)$ gives us a measure of the urgency of the task. That is, if there is a deadline at time T before which the agent must act, then the function $g(x)$ should grow much larger as x approximates T. Figure 4.4 summarizes the notation for handling the strategies associated with a particular partially expanded situation.

Using the notation from Figure 4.4, we can finally express the utility of the partially expanded situation t that agent i is in. Let t' be the situation t after expanding some leaf node l in it, i.e., $t' \leftarrow \text{PE}(t, l)$. The utility is the payoff the agent will get, given the current evaluation of t'. The expected utility of t' is the utility that an agent in t *expects* to get if he expands l so as to then be in t'.

$$U(t') = U_i(t'.a_i, t'.a_j \cdots t'.a_n) \tag{4.4}$$

$$E(U(t')) = U_i(t'.\hat{a}_i^l, t.\hat{a}_j^l \cdots t.\hat{a}_n^l) \tag{4.5}$$

Note that the utility of a situation depends on the shape of the tree below it since all $t.a_i$, $t.a_j \cdots t.a_n$ are calculated by using the whole tree. There are no utility values associated with the nodes by themselves.[2]

Unfortunately, we cannot use the utilities of situations in the traditional way, which is to expand leaves so as to maximize expected utility. Such traditional reasoning in a recursive modeling system would mean that the system would not introspect (i.e., examine its recursive models) more deeply if it thought that doing so might lead it to discover that it will not, in fact, receive as high a payoff as it thought it would before doing the introspection. Ignoring more deeply nested models because they reduce estimated payoff is a bad strategy, because ignoring them does not change the fact that other agents might take the actions that lead to these lower payoffs. This reasoning is better understood with the help of an example.

We contrast our problem with the case of using minimax for playing a board game, as studied in (Russell and Wefald, 1991). There, if a player realizes that taking move A will lead to a set of possible outcomes, none of which have higher utility than he gets when taking move B, then the player should not consider move A anymore. Within our system of recursive modeling agents, however, this result does not apply. Imagine a predator who is very close to surrounding the prey. The predator might initially decide that the others will be taking moves so that, if he moves North, the prey will be captured. A capture has a very high utility and is very desirable to the predator. However, the predator also realizes that if he thinks some more about the predator to his left, he will probably change his mind about what the predator to the left will do, and he will then prefer to go West. The resulting situation will not be a capture and, therefore, will have smaller utility. The question then is: should the predator stop thinking about what the predator to its left will do just because doing so might lead him to think his prospects for the future are worse? We do not believe so. The predator should continue to think more deeply about others as long as doing so makes him change his mind about which move to take, even when thinking more leads him to the realization that things are worse than he initially believed.[3] So, while in the case of minimax a move that was found to have lower utility was abandoned forever, in our case it is always possible that a move with lower utility will later get a higher utility because of changes in the actions we expect the other agents to take. Thus, we need a different notion

[2]This is in contrast to Russell and Wefald's work (Russell and Wefald, 1991), where each node has a utility associated with it.

[3]We can anthropomorphize this advice to say that it does not help to pretend things are better than they probably are; we must determine exactly where we stand and take the optimal action based on this assessment.

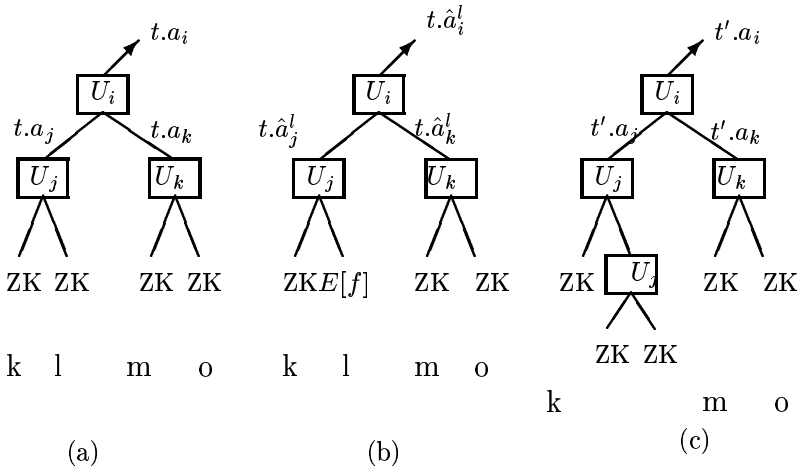


Figure 4.5: Here we can see how the algorithm proceeds when expanding a situation. We start with t in (a). For each leaf we calculate the expected gain. (b) shows how this is done for leaf l. If we do decide to expand leaf l we will end up with a new t', as seen in (c).

of what an agent can gain by examining more of its deeper nested knowledge.

The driving force for the agent's reasoning is to decide what is its best course of action. Thus, expanding a situation is only useful if, by doing so, the agent chooses a different action than it would have before doing the expansion. And the degree to which the agent is better off with the new action, rather than the original action, given what it now knows, represents the **gain** due to the expansion. The reader will remember that in Chapter 3.3.1 we defined the concept of knowledge **dampening**, which is the probability that different actions taken by other agents will not lead the agent, that has the knowledge in question, to change the action it chooses to take. If a particular U_i has high dampening this means that, no matter what the other agents do, agent i will always take the same action. This means that this agent gets no gain from determining which actions the others will take.

More specifically, the Gain of situation t', $G(t')$, is the amount of payoff an agent gains if it was previously in situation t and, after expanding leaf $l \in t$.leaves, is now in situation t'. Since $t' \leftarrow \text{PE}(t, l)$, we could also view this as $G(PE(t, l))$. The Expected Gain, $E(G(t'))$, is similarly $E(G(PE(t, l)))$, which is approximated by G(Propagate(E(Expand(t, l)))) in our algorithm. We can justify this approximation because Propagate() simply propagates the strategy at the leaf up to the root, and it usually maps similar strategies at the leaves to similar strategies at the root. So, if the expected strategy at the leaf, as returned by E(Expand(t, l)), is close to the real one, then the strategy that gets propagated up to the

root will also be close to the real one.[4] The formal definitions of gain and expected gain are:

$$\begin{aligned} G(t') &= U_i(t'.a_i, t'.a_j \cdots t'.a_n) - U_i(t.a_i, t'.a_j \cdots t'.a_n) - TC(t,l) \\ E(G(t')) &= U_i(t.\hat{a}_i^l, t.\hat{a}_j^l \cdots t.\hat{a}_n^l) - U_i(t.a_i, t.\hat{a}_j^l \cdots t.\hat{a}_n^l) - TC(t,l) \end{aligned}$$

where, as we showed in Figure 4.4, $t.\hat{a}_i^l$ is the strategy agent i expects to play given that, instead of actually expanding leaf l it simply uses its expected value, derived from $f(x)$, and propagates this strategy up the tree. Notice that $E(G(t'))$ reaches a minimum when $t.a_i = t.\hat{a}_i^l$, since $t.\hat{a}_i^l$ is chosen so as to maximize the payoff given $t.\hat{a}_j^l \cdots t.\hat{a}_n^l$. In this case, $G(t') = -TC(t,l)$. This means that the gain will be negative if the possible expansion l does not lead to a different and better strategy. In effect, the gain captures the aggregate dampening that exists on the path between the leaf l and the root node. If this dampening is 1 then the gain will just be $TC(t,l)$. If the dampening is 0 then the gain will depend on the actual expected actions. By calculating expected actions, our measure of expected gain is able to give us more information than we would get with just the dampening values.

4.4 Algorithm

Our algorithm starts with the partially expanded root situation, which consists only of the payoff matrix for the agent. ZK strategies are used for the situations of other agents, forming the initial leaves. The algorithm proceeds by expanding, at each step, the leaf node that has the highest expected gain, as long as this gain is greater than K. For testing purposes, we set $K = 0$, but setting K to some small negative number would allow for the expansion of leaves that do not show an immediate gain, but might lead to some gains later on. The procedure `Expected_Strategy(l)` takes as input a leaf node situation l and determines the strategy that its expansion, to some number of levels, is *expected* to return. It does this by calculating the expected value of the corresponding $f(x)$ which, the reader will remember, is the probability function that returns the probability that the strategy x is the one that the situation will evaluate to. The strategy is propagated up by the procedure `Propagate_Strategy(t,l)`, which returns the expected $t.\hat{a}_i^l$ and $t.\hat{a}_j^l \cdots t.\hat{a}_n^l$. These are then used to calculate the expected gain for leaf l. The leaf with the maximum expected gain, if it is greater than K, is expanded, a new strategy propagated, and the whole process is repeated. Otherwise, we stop expanding and return the current $t.\alpha$.

The algorithm, shown in Figure 4.6, is started by calling `Expand_Situation(t)`. Before the call, t must be set to the root situation, and $t.\alpha$ to the strategy that t evaluates to.

[4] Test results show that this approximation is good enough to increase the performance of our algorithm over simple breadth-first search, as we show in Section 4.5.1.

Expand_Situation(t)
  max_gain $\leftarrow -\infty$; max_sit $\leftarrow$nil; $t' \leftarrow t$
  For $l \in t$.leaves /* l is a structure */
    t'.leaf $\leftarrow l$
    $l.a_i \leftarrow$ Expected_Strategy(l)
    $t.\hat{a}_i^l$, $t.\hat{a}_j^l \cdots t.\hat{a}_n^l \leftarrow$ Propagate_Strategy(t, l)
    t'.gain $\leftarrow U_i(t.\hat{a}_i^l, t.\hat{a}_j^l \cdots t.\hat{a}_n^l) - U_i(t.a_i, t.\hat{a}_j^l \cdots t.\hat{a}_n^l) - TC(t, l)$
    If t'.gain $>$ max_gain Then
      max_gain $\leftarrow t'$.gain
      max_sit $\leftarrow t'$
  If max_gain $> K$ Then
    $t \leftarrow$ Propagate_Expand(max_sit)
    Return(Expand_Situation(t))
  Return($t.a_i$)

Propagate_Expand(t) /* l is included in t.leaf */
  leaf $\leftarrow t$.leaf
  leaf.$a \leftarrow$ Expand(leaf)
  $t.a_i$, $t.a_j \cdots t.a_n \leftarrow$ Propagate_Strategy(t, leaf)
  t.leaves $\leftarrow$ New_Leaves(t)
  Return(t)

Propagate_Strategy(t, l)
  Propagates $l.a$ all the way up the tree and returns the best strategy for the root situation, and the strategies for the others.

Expand(l)
  Returns the strategy we find by first calculating the matrix that corresponds to l, and then plugging the zero-knowledge solution for all the children of l.

New_Leaves(t)
  Returns only those leaves of t that can be expanded into full situations.

Figure 4.6: Limited Rationality RMM (LR-RMM) Algorithm

The strategy returned by `Expand_Situation` could then be stored away so that it can be used, at a later time, by `Expected_Strategy`. The decision to do this would depend on the particular implementation. One might only wish to remember those strategies that result from expanding a certain number of nodes or more, that is, the "better" strategies. In other cases, it could be desirable to remember all previous strategies that correspond to situations that the agent has experienced before, on the theory that it is better to have some information rather than none. This is the classic time versus memory tradeoff.

4.4.1 Time Analysis

For the algorithm to work correctly, the time spent in metalevel thinking must be small compared to the time it takes to do an actual node expansion and propagation of a new solution (i.e., a $PE(t,l)$). Otherwise, the agent is probably better off choosing nodes to expand at random without spending time considering which one might be better.

A node expansion involves the creation of a new matrix U. If we assume that there are $|N|$ agents and each one has $|A|$ possible actions, then the number of elements in the matrix will be $|A|^{|N|}$. Each one of these elements represents the utility of the situation that results after all the agents take their actions. In order to calculate this utility, it is necessary to simulate the new state that results from all the agents taking their respective actions (i.e., as dictated by the element's position in the matrix) and then calculate the utility, for some particular agent, of this new situation (see Figure 4.7). The calculation of this utility can be an arbitrarily complex function of the state, and must be performed for each of the $|A|^{|N|}$ elements in the matrix.

The next step is the propagation of the expected strategy. If we replace one of the ZK leaf strategies by some other strategy, then the time to propagate this change all the way to the root of the tree depends both on the distance between the root and leaf, and the time to propagate a strategy past a node. Because strategies are calculated by determining which one maximizes the utility, it is almost certain that any strategy that is the result of evaluating a node will be a pure strategy.[5] The only time we should see a mixed a strategy is when there are several actions, all of which return the maximum utility payoff.

If all the children of a node are pure strategies, then the strategy the node evaluates to can be found by determining the maximum of $|A|$ numbers since, in this case, the $|N|$-dimensional matrix collapses into a one-dimensional vector. If c of the children have mixed strategies, we would need to add $|A|^{c+1}$ numbers and find the maximum of $|A|$ numbers. In the worst case we have $c = |N| - 1$ so, in the worst case, we need to add $|A|^{|N|}$ numbers.

Therefore, the propagation of a solution will require at most d maximizations plus

[5] In a pure strategy, one action is chosen with probability 1, the rest with probability 0.

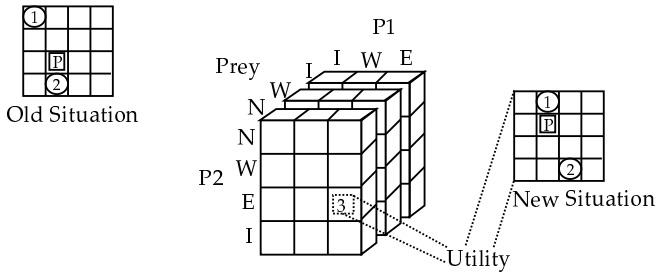


Figure 4.7: A very simplified version of the Pursuit Problem with only 3 agents and in a 4 by 4 grid. Given the old situation, each one of the elements of the matrix corresponds to the payoff in one of the many possible next situations. There is one for each combination of moves the agents might take (e.g. North, East, Idle). Sometimes, the moves of the predators might interfere with each other. The predator calculating the matrix has to resolve these conflicts when determining the new situation, before calculating its utility.

$d \cdot |A|^{c+1}$ additions, were d is the depth of the tree and $|A|^{c+1}$ is the number of additions it takes to propagate past a node. However, in most instances the propagation past a node consists only of one *max* operation, since we are unlikely to see mixed strategies. This time is small, especially when compared to the time needed for the simulation and utility calculation of $|A|^{|N|}$ different possible situations. A more detailed analysis can only be performed on an application-specific basis, and it would have to take into account actual computational times.[6]

4.4.2 Implementation Strategies

A strategy we adopt to simplify the implementation is to classify agents according to *types*. Associated with each type is a function that takes as input the agent's physical situation and generates the full mental situation for that agent. This technique uses little memory and is an intuitive way of programming the agents. It assumes, however, that the mental situation can be derived solely from the physical. This was true in our experiments but need not always be so. A counter example would be a case where the mental situation of an agent depends on its individual biases or past experiences. The function `Expected_Strategy`, which calculates the expected value of $f(x)$, can then be implemented by a hash table for the agent type. The keys to the table would be *similar* physical situations and the values would be the last y strategies played in those situations. Grouping similar situations is needed because the number of raw physical situations can be very big. The

[6]Some basic calculations, along these lines, are given in Section 4.5.1.

Method	Avg. Nodes Expanded	Avg. Time to Capture/Surround
BFS 1 Level	1	∞/Never Captured
BFS 2 Levels	4	23.4
BFS 3 Levels	13	22.4
BFS 4 Levels	40	17.3
LR RMM	4.2	18.5
Greedy	NA	41.97

Table 4.1: Results of implementation of the Pursuit problem using simple BFS to various levels (i.e., regular RMM), using our Limited Rationality Recursive Modeling algorithm, and using a simple greedy algorithm, in which each predator simply tries to minimize its distance to the prey.

definition of what situations are similar is heuristic and determined by the programmer. In our case we have chosen to represent situations from the agent's point of view, without any references to the cardinal points or any other fixed landmarks. This means that situations are automatically equivalent under rotations of 90 degrees. Also, if an agent is more than a Manhattan distance of two from the prey, we only consider the quadrant in which it is located (with respect to the agent observing it), not the actual distance. The intuition is that agents that are far from the prey do not influence the other agents' move decisions.

4.5 Implementation of Pursuit Task

The original algorithm has been implemented in a simulation of the pursuit task, using the MICE system (Montgomery and Durfee, 1990). The experiments show some promising results. The overall performance of the agents is maintained while the total number of expanded nodes is reduced by an order of magnitude.

We designed a function that takes as input a physical situation and a predator in that situation, and returns the payoff matrix for the predator in that situation. This function is used by all the predators; that is, we assumed all predators were of the same type. Notice that the existence of this function actually implies that the agents have common knowledge about their payoffs, since they can generate nested models of each other to arbitrary depths. However, this fact is is not used by the LR-RMM algorithm. The functions are used to generate the nested models to some fixed arbitratry depth and then LR-RMM uses this tree of nested models in its execution. That is, the LR-RMM algorithm does not assume that the agents have common knowledge about their payoffs.

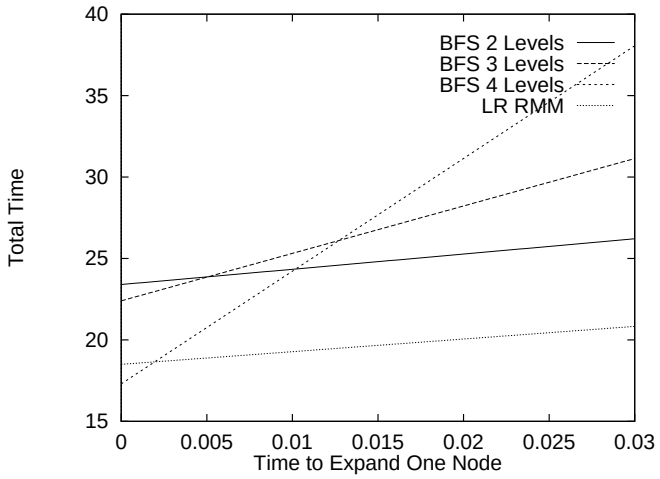


Figure 4.8: Plot of the total time that would, on average, be spent by the agents before capturing the prey, for each method. The x axis represents the amount of time it takes to expand a node as a percentage (more or less) of the time it takes the agent to take action.

A predator's payoff is the sum of two values. The first value is the change in the distance from the predator to the prey between the current situation and the new situation created after all agents have made their moves. The second value is 5^k, where k is the number of quadrants around the prey that have a predator in them in the new situation. The quadrants are defined by drawing two diagonal lines across the prey (Gasser et al., 1989; Levy and Rosenschein, 1992).

Since the matrices take into account all possible combinations of moves (5 for each predator and 4 for the prey), they are five-dimensional with a total of 2500 payoff entries. With matrices this big, even in a simple problem, it is easy to see why we wish to minimize the number of matrices we need to generate. We define the physical situation which the predator is in as its relative position to the prey and to the other predators. These situations are then generalized such that distances greater than two are indistinguishable, except for quadrant information. This generalization formula serves to shrink the number of buckets in our hash table to a manageable number (from $4.1 \cdot 10^{11}$ to around 3000). Notice also that the number of buckets remains constant no matter how big the grid is.

4.5.1 Results

Our tests were run on an 20 by 20 grid with four predators randomly placed and one prey that always started in the middle of the grid. We started with tests where the predators used simple Breadth First Search of the situation hierarchy. When using BFS to one level, the

predators could not predict the actions of the others and so, they never got next to the prey. As we increased the number of levels, the predators could better predict what the others would do, which made their actions more coordinated and they managed to capture the prey. The goal for our algorithm was to keep the same good results while expanding as few nodes as possible. For comparison purposes, we also tested a very simple greedy algorithm where each predator simply tries to minimize its distance to the prey, irrespective of where the other predators are.

We then ran our Limited Rationality RMM algorithm on the same setup. We had previously compiled a hash table which contained several entries (usually around 5, but no more than 10) for almost all situations. This was done by generating random situations and running RMM to 3 or 4 levels on them to find out the strategy. The results, as seen in Table 4.1, show that our algorithm managed to maintain the performance of a BFS to 4 levels while only expanding little more than four nodes on the average. All the results are the averages of 20 or more runs.

Another way to view these results is by plotting the total time it takes the agents, on average, to surround the prey as a function of the time it takes to expand a node. If we assume that the time for an agent to make a move is 1 unit and we let the time to expand a node be x, the average number of turns before surrounding the prey be t_s, and the average number of nodes n, then the total time T the agents spends before surrounding the prey is approximately $T = t_s \cdot (n \cdot x + 1)$, as shown in Figure 4.8. LR RMM is expected to perform better than all others except when the time to expand a node is much smaller (.002 times smaller) than the time to perform an action.

4.6 Conclusions

We have presented an algorithm that provides an effective way of pruning a recursive model so that only a few critical nodes need to be expanded in order to get a quality solution. The algorithm does the double task of delivering an answer within a set of payoff/time cost constraints, and pruning unnecessary knowledge from the recursive model of a situation. This pruning exploits the presence of dampening knowledge within the agents' models. We also extended our recursive modeling agent notation to encompass strategies, expected strategies, and expected gains from expanding a recursive model.

The experimental results proved that this algorithm works, in the example domain. We expect that the algorithm will provide similar results in other problem domains, provided that they show some correspondence between the physical situation and the strategy played by the agent. The test runs taught us three main lessons. Firstly, that it is possible and beneficial to ignore some of the deeper models. Secondly, that the models to ignore vary

depending on which state the agent is in. Our data showed that the actual nodes chosen for expansion differ from situation to situation (i.e., depending on the relative locations of the predators and prey). Thirdly, they showed us that a lot of memory is needed for handling the computation of expected strategies, even without considering the agent's mental state. This seems to suggest that more complete implementations will require a very smart similarity function for detecting which situations are similar to each other, and thus reduce the total number of dissimilar situations.

We expect the LR-RMM algorithm to work well in domains other than the Pursuit Task. The reason LR-RMM was successful in this task was because there was dampening present in the payoff matrices. That is, sometimes the payoffs made it so that it did not matter what action resulted from the evaluation of a deeper model. For example, in the Pursuit Task when one agent is very far away, its actions are not relevant to the other agents. Therefore, the other agents need not think about this agent. We can expect that LR-RMM will work well in any other domains where the agents' knowledge has some dampening value. The more dampening that exists, the more likely it is that LR-RMM will be successful.

In summary, the LR-RMM algorithm provides an answer to the question of when should an agent stop thinking about what others are thinking about what others are thinking, and so on. This problem arises naturally when we consider agents that have models of agents. While it is true that game theory addresses this problem when common knowledge is assumed, we believe that our algorithm is the first to solve it for cases where there is no common knowledge and the agents, instead, have nested models of the other agents.

CHAPTER 5

Learning in a Learning MAS—The CLRI Framework

So far we have assumed that the agents had all the knowledge they needed in order to pick their actions (i.e., all the $\delta_i(w)$'s), and that this knowledge did not change over time. However, in a significant number of MASs we find that the agents use some form of machine learning, which sometimes starts with absolutely no initial knowledge, and sometimes builds on existing knowledge the designers built into the agent.

We can model these agents by allowing the decision function $\delta_i(w)$ to vary with time $\delta_i^t(w)$. The superscript t indicates the time at which this decision function holds for agent i. The learning problem the agent faces is to change its $\delta_i^t(w)$ so that it matches $\Delta_i^t(w)$. If we imagine the space of all possible decision functions, then agent i's $\delta_i^t(w)$ and $\Delta_i^t(w)$ will be two points in this space, as shown in Figure 5.1. The agent's learning problem can then be re-stated as the problem of moving its decision function as close as possible to its target function, where the distance between the two functions is given by the error $e(\delta_i^t)$. This is the traditional machine learning problem, which we show in Figure 5.1.

However, once agents start to change their decision functions (i.e., change their behaviors) the problem of learning becomes more complicated because these changes might make the other agents' target functions change. We end up with a moving target function, as seen in Figure 5.2. In these systems, it is not clear if the error will ever reach 0 or, more generally, what the expected error will be as time goes to infinity. Determining what will happen to an agent's error in such a system is what we call the **moving target function problem**, which we address in this chapter.

Section 5.1 extends our framework from Chapter 3.1 and allows it to describe an agent's learning abilities. Section 5.2 presents an equation that can be used to predict an agent's expected error, as a function of time, when the agent is in a MAS composed of other learning agents. The equation is simplified in Section 5.3. Section 5.4 then defines the last few parameters used in our framework. Finally, the predictions made by our framework are verified with our experiments, and the experiments of others, as detailed in Section 5.7.

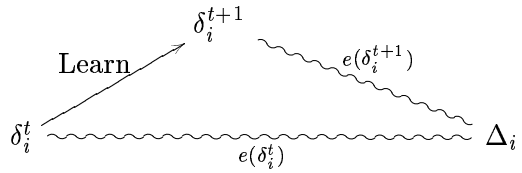


Figure 5.1: The traditional learning problem.

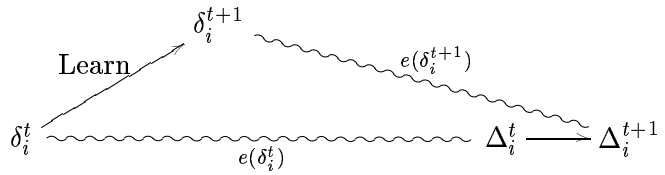


Figure 5.2: The learning problem in learning MASs.

5.1 Modeling a Learning Algorithm

We assume that the agents in the MAS are engaged in the discrete action/learn loop shown in Figure 5.3. The loop works as follows: At time t the agents perceive a world $w^t \in W$ which is drawn from a fixed distribution $\mathcal{D}(w)$. They then each take the action dictated by their δ_i^t functions; all of these actions are assumed to be taken effectively in parallel. Lastly, they each receive a payoff which their respective learning algorithms use to change the δ_i^t so as to (hopefully) better match Δ_i^t. By time $t+1$, the agents have new δ_i^{t+1} functions and are ready to perceive the world again and repeat the loop. Notice that, at time t, an agent's Δ_i^t is derived by taking into account the δ_j^t of all other agents $j \in N_{-i}$. That is, Δ_i^t is the best possible behavior for agent i at time t, given that all the other agents $j \in N_{-i}$ take actions as dictated by their respective δ_j^t's.

An agent's learning algorithm is responsible for changing δ_i^t into δ_i^{t+1} so that it better matches Δ_i^t. Different machine learning algorithms will achieve this match with different degrees of success. We have found a set of parameters that can be used to model the effects of a wide range of learning algorithms, which we present in this section.

After agent i takes an action and receives some payoff, it activates its learning algorithm. The learning algorithm is responsible for using this payoff in order to change δ_i^t into δ_i^{t+1}, making δ_i^{t+1} match Δ_i^t as much as possible. We can expect that for some w's it was true that $\delta_i^t(w) = \Delta_i^t(w)$, while for some other w's this was not the case. That is, some of the $w \rightarrow a_i$ mappings given by $\delta_i^t(w)$ might have been incorrect. In general, a learning algorithm might affect both the correct and incorrect mappings. We will treat these two cases separately.

We start by considering the incorrect mappings and define the **change rate** of the agent as the probability that the agent will change one of its incorrect mappings. Formally, we

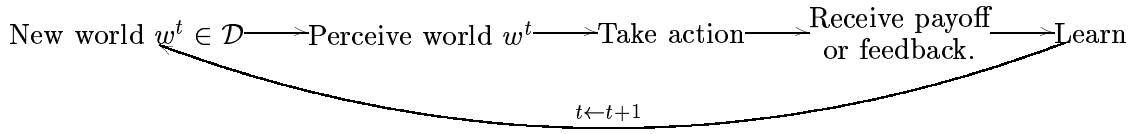


Figure 5.3: Action/Learn loop for an agent.

define the change rate c_i for agent i as

$$\forall_w \; c_i = \mathbf{Pr}[\delta_i^{t+1}(w) \neq \delta_i^t(w) \mid \delta_i^t(w) \neq \Delta_i^t(w)] \tag{5.1}$$

The change rate tells us how likely the agent is to change an incorrect mapping into something else. This "something else" might be the correct action, but it could also be another incorrect action. The probability that the agent changes an incorrect mapping to the correct action is called the **learning rate** of the agent, which is defined as l_i where

$$\forall_w \; l_i = \mathbf{Pr}[\delta_i^{t+1}(w) = \Delta_i^t(w) \mid \delta_i^t(w) \neq \Delta_i^t(w)] \tag{5.2}$$

When determining the value of l_i, for a particular agent, one must remember to take into account the fact that the worlds seen at each time step are taken from $\mathcal{D}(w)$.

There are two constraints that must always be satisfied by these two rates. Since changing to the correct mapping implies that a change was made, the value of l_i must be less than or equal to c_i, i.e., $l_i \leq c_i$ must always be true. Also, if $|A_i| = 2$ then $c_i = l_i$ since there are only two actions available, so the one that is not wrong must be right. The complementary value for the learning rate is $1 - l_i$ and refers to the probability that an incorrect mapping does not get changed to a correct one. An example learning rate of $l_i = .5$ means that, if agent i initially has all mappings wrong, it will get half of them right after the first iteration.

We now consider the agent's correct mappings and define the **retention rate** as the probability that a correct mapping will stay correct in the next iteration. The retention rate is given by r_i where

$$\forall_w \; r_i = \mathbf{Pr}[\delta_i^{t+1}(w) = \Delta_i^t(w) \mid \delta_i^t(w) = \Delta_i^t(w)] \tag{5.3}$$

We propose that the behavior of a wide variety of learning algorithms can be captured (or at least approximated) using appropriate values for c_i, l_i, and r_i. Notice, however, that these three rates claim that the $w \rightarrow a$ mappings that change are independent of the w that was just seen. We can justify this independence by noting that most learning algorithms usually perform some form of generalization. That is, after observing one world state w, and the payoff associated with it, a typical learning algorithm is able to generalize what it learned to some other world states. This generalization is reflected in the fact that the

change, learning, and retention rates apply to all w's. However, a more precise model would capture the fact that, in some learning algorithms, the mapping for the world state that was just seen is more likely to change than the mapping for any other world state.

The rates are not time dependent because we assume that agents use one learning algorithm during their lifetimes. The rates capture the capabilities of this learning algorithm and, therefore, do not need to vary over time.

Finally, we define **volatility** to mean the probability that the target function will change from time t to time $t+1$. Formally, volatility is given by v_i where

$$\forall_w \; v_i = \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w)] \tag{5.4}$$

In Section 5.4, we will show how to calculate v_i in terms of the error of the other agents. We will then see that volatility is not a constant but, instead, varies with time.

5.2 Calculating the Agent's Error

We define the error of agent i's decision function δ_i^t at time t as:

$$e(\delta_i^t) = \sum_{w \in W} \mathcal{D}(w) \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \tag{5.5}$$

where, as mentioned before, $\mathcal{D}(w)$ is a fixed probability distribution from which the worlds seen are taken. $e(\delta_i^t)$ gives us the probability that agent i will take an incorrect action. This definition of error is in keeping with the error definition used in computational learning theory (Kearns and Vazirani, 1994), and is the same one we gave in Chapter 3, except for the time superscript.

Given that we have an agent's error and his learning parameters (i.e., the change rate, learning rate and retention rate) we can derive a difference equation for determining the progression of this error over time. The agent's expected error at time $t+1$ can be calculated by considering the four possibilities of whether $\Delta_i^t(w)$ changes or not, and whether $\delta_i^t(w)$ was correct or not.

$$\begin{aligned}
E[e(\delta_i^{t+1})] = E[\sum_{w \in W} \mathcal{D}(w) \mathbf{Pr}[\delta_i^{t+1}(w) \neq \Delta_i^{t+1}(w)]] = \sum_{w \in W} \mathcal{D}(w) (\\
\mathbf{Pr}[\Delta_i^{t+1}(w) = \Delta_i^t(w) | \delta_i^t(w) = \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot (1 - r_i) \\
+ \mathbf{Pr}[\Delta_i^{t+1}(w) = \Delta_i^t(w) | \delta_i^t(w) \neq \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot (1 - l_i) \\
+ \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) | \delta_i^t(w) = \Delta_i^t(w)] \\
\cdot \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot (r_i + (1 - r_i) \cdot B) \\
+ \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) | \delta_i^t(w) \neq \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot C
\end{aligned} \tag{5.6}$$

where we define

$$B = \mathbf{Pr}[\delta_i^{t+1}(w) \neq \Delta_i^{t+1}(w) | \delta_i^t(w) = \Delta_i^t(w) \wedge \Delta_i^{t+1}(w) \neq \Delta_i^t(w) \wedge \delta_i^{t+1}(w) \neq \Delta_i^t(w)] \tag{5.7}$$

$$C = l_i + (1 - c_i)D + (c_i - l_i)F \tag{5.8}$$

$$D = \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^{t+1}(w) | \delta_i^t(w) \neq \Delta_i^t(w) \wedge \Delta_i^{t+1}(w) \neq \Delta_i^t(w)] \tag{5.9}$$

$$F = \mathbf{Pr}[\delta_i^{t+1}(w) \neq \Delta_i^{t+1}(w) | \delta_i^t(w) \neq \Delta_i^t(w) \wedge \Delta_i^{t+1}(w) \neq \Delta_i^t(w) \wedge \delta_i^{t+1}(w) \neq \Delta_i^t(w) \wedge \delta_i^{t+1}(w) \neq \delta_i^t(w)] \tag{5.10}$$

Equation (5.6) separates the four cases of whether or not δ_i^t was correct, and whether or not Δ_i^t changes from t to $t+1$. Each of these cases is represented by a term in the sum. Each of the terms is multiplied by the expected error of the case the term represents. The expected error of a case is just the probability that $\delta_i^{t+1}(w) \neq \Delta_i^{t+1}(w)$ for some particular w (as given by the summation), in the particular case. For example, the first term covers the case where $\delta_i^t(w) = \Delta_i^t(w)$ and $\Delta_i^{t+1}(w) = \Delta_i^t(w)$, for some particular w. In English, these two equalities mean that, for this w, agent i was correct at time t and the target function did not change from time t to time $t+1$. If agent i was correct for this w at time t, and the target function does not change from time t to time $t+1$, then we can expect that agent i will also be correct at time $t+1$ *unless* it changes its correct mapping to something else. The probability that it will make this change is given by 1 minus the retention rate. Therefore, the expected error for this case is simply $1 - r_i$, i.e., the probability that i will change one of its correct mappings to something else.

The second term in Eq. (5.6) covers the case where $\delta_i^t(w) \neq \Delta_i^t(w)$ and $\Delta_i^{t+1}(w) = \Delta_i^t(w)$, for some particular w. In English, these two inequalities mean that, for this w, agent i was incorrect and the target function does not change from time t to time $t+1$. If agent i was incorrect, then we can expect that it will change this incorrect mapping, so as to match $\Delta_i^t(w)$, with probability l_i. Since the target function does not change for this particular w, the new error will simply be the probability that the agent does not change its incorrect mapping so as to match $\Delta_i^t(w)$. That is, the expected error for this case is $1 - l_i$.

The third term in Eq. (5.6) covers the case where $\delta_i^t(w) = \Delta_i^t(w)$ and $\Delta_i^{t+1}(w) \neq \Delta_i^t(w)$, for some particular w. In English, these two inequalities mean that, for this w, agent i was correct and the target function changes from time t to $t+1$. Since the target function changes and the agent was correct, we can expect that the new $\delta_i^{t+1}(w)$ will be incorrect only if it is either the same as the old $\delta_i^t(w)$ (which happens with probability r_i), or if it is different from $\delta_i^t(w)$ but the new $\delta_i^{t+1}(w)$ still does not match the new target function $\Delta_i^{t+1}(w)$. The probability of $\delta_i^{t+1}(w)$ being different from $\delta_i^t(w)$, but still not matching the new target, is given by the multiplication of $1 - r_i$ and B, as given by Eq. (5.7). B is simply

the probability that the new decision function (at time $t+1$) is in error given that the old decision function (at time t) did not match the old target function, and the new target function does not match the old target function, and the new decision function does not match the old target function.

The fourth term in Eq. (5.6) covers the case where $\delta_i^t(w) \neq \Delta_i^t(w)$ and $\Delta_i^{t+1}(w) \neq \Delta_i^t(w)$, for some particular w. This case, of course, is the most complicated of all since it considers the case where the decision function was incorrect and the target function changes. The error in this case depends on the three possibilities, given in Eq. (5.8), of whether the $\delta_i^t(w)$ changes to match $\Delta_i^t(w)$, whether $\delta_i^t(w)$ does not change, and whether $\delta_i^t(w)$ changes but does not match $\Delta_i^t(w)$. For the first case, where the decision function changes to match $\Delta_i^t(w)$, which it does with probability l_i, the error is 1 since the target function also changes. For the second case, where $\delta_i^t(w)$ does not change, which happens with probability $1 - c_i$, the error is given by Eq. (5.9). For the third case, where $\delta_i^t(w)$ changes but does not match $\Delta_i^t(w)$, which happens with probability $c_i - l_i$, the error is given by Eq. (5.10).

Equation (5.6) will model any MAS whose agent learning can be described with the parameters presented Section 5.1 and whose action/learn loop is the same as we described. We can use Eq. (5.6) to calculate the successive expected errors for agent i, given values for all the parameters and probabilities. In the next section we show how this is done in a simple example game.

5.2.1 The Matching game

In this matching game we have two agents i and j each of whom, in every world w, wants to play the same action as the other one. Their set of actions is $A_i = A_j$, where we assume $|A_i| > 2$ (for $|A_i| = 2$ the equation is simpler). After every time step, the agents both learn and change their decision functions in accordance to their learning rates, retention rates, and change rates. Since the agents are trying to match each other, in this game it is always true that $\Delta_i^t(w) = \delta_j^t(w)$ and $\Delta_j^t(w) = \delta_i^t(w)$. Given all this information, we can find values for some of the probabilities in Eq. (5.6) (including values for Equations (5.7) (5.8) (5.9) (5.10)) and rewrite (see Appendix A for derivation) it as:

$$\begin{aligned} E[e(\delta_i^{t+1})] = \sum_{w \in W} \mathcal{D}(w) & (r_j \cdot \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot (1 - r_i) \\ & + (1 - c_j) \cdot \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot (1 - l_i) \\ & + (1 - r_j) \cdot \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot \left(r_i + (1 - r_i) \cdot \left(\frac{|A_i| - 2}{|A_i| - 1} \right) \right) \\ & + c_j \cdot \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot \left(1 - l_j + \frac{c_i l_j (|A_i| - 1) + l_i (1 - l_j) - c_i}{|A_i| - 2} \right)) \end{aligned} \tag{5.11}$$

We can better understand this equation by plugging some values and simplifying. For

example, lets assume that $r_i = r_j = 1$ and $l_i = l_j = 1$, which implies that $c_i = c_j = 1$. This is the case where the two agents always change all their incorrect mappings so as to match their respective target functions at time t. That is, if we had $\delta_i^t(w_1) = x$ and $\delta_j^t(w_1) = y$, then at time $t+1$ we will have $\delta_i^{t+1}(w_1) = y$ and $\delta_j^{t+1}(w_1) = x$. This means that agent i changes all its incorrect mappings to match j, while j changes to match i, so all the mappings stay wrong after all (i.e., i ends up doing what j did before, while j does what i did before). The error, therefore, stays the same. We can see this by plugging the values into Eq. (5.11). The first three terms will become 0 and the fourth term will simplify to the definition of error, as given by Eq. (5.5). Since the fourth term is the only one that is non-zero, we end up with $E[e(\delta_i^{t+1})] = e(\delta_i^t)$.

We can also let c_i and l_i (keeping $c_j = l_j = 1$) be arbitrary numbers, which gives us $E[e(\delta_i^{t+1})] = c_i e(\delta_i^t)$. This tells us that the error will drop faster for a smaller change rate c_i. The reason is that i's learning (remember $l_i \leq c_i$) in this game is counter-productive because it is always made invalid by j's learning rate of 1. That is, since j is changing all its mappings to match i's actions, i's best strategy is to keep its actions the same (i.e., $c_i = 0$).

5.3 Further Simplification

We can further simplify Eq. (5.6) if we are willing to make two assumptions. The first assumption is that the new actions chosen when either $\delta_i^t(w)$ changes (and does not match the target), or when $\Delta_i^t(w)$ changes, are both taken from flat probability distributions over A_i. By making this assumption we can find values for A, D, F and C, namely:

$$B = D = \frac{|A_i| - 2}{|A_i| - 1} \qquad F = \frac{|A_i| - 3}{|A_i| - 2} \tag{5.12}$$

which makes

$$C = \frac{|A_i| - 2 - c_i + 2l_i}{|A_i| - 1} \tag{5.13}$$

The second assumption we make is that the probability of $\Delta_i^t(w)$ changing, for a particular w, is independent of the probability that $\delta_i^t(w)$ was correct. In Section 5.2.1 we saw that in the matching game the probabilities of $\Delta_i^t(w)$ and $\delta_i^t(w)$ changing were correlated since, if $\delta_i^t(w)$ was wrong then $\delta_j^t(w)$ was also wrong, which meant j would probably change $\delta_j^t(w)$, which would change $\Delta_i^t(w)$.

However, the matching game is a degenerate example in exhibiting such tight coupling between the agents' target functions. In general, we can expect that there will be a number of MASs where the probability that any two agents i and j are correct is uncorrelated (or loosely correlated). For example, in a market system all sellers try to bid what the buyer wants, so the fact that one seller bids the correct amount says nothing about another seller's

bid. Their bids are all uncorrelated. In fact, the Distributed Artificial Intelligence literature is full of systems that try to make the agents' decisions as loosely-coupled as possible (Lesser and Corkill, 1981), (Liu and Sycara, 1995).

This second assumption we are trying to make can be formally represented by having Eq. (5.14) be true for all pairs of agents i and j in the system.

$$\mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w) \wedge \delta_j^t(w) = \Delta_j^t(w)] = \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_j^t(w) = \Delta_j^t(w)] \quad (5.14)$$

Once we make these two assumptions we can rewrite Eq. (5.6) as:

$$\begin{aligned} E[e(\delta_i^{t+1})] = \sum_{w \in W} \mathcal{D}(w)(&\mathbf{Pr}[\Delta_i^{t+1}(w) = \Delta_i^t(w)] \cdot (\mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot (1 - r_i) \\ &+ \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot (1 - l_i)) \\ &+ \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w)] \cdot (\mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot \left(r_i + (1 - r_i) \cdot \left(\frac{|A_i| - 2}{|A_i| - 1}\right)\right) \\ &+ \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot \left(\frac{|A_i| - 2 - c_i + 2l_i}{|A_i| - 1}\right))) \end{aligned} \quad (5.15)$$

Some of the probabilities in this equation are just the definition of v_i, the others simplify to the agent's error. This means that we can simplify Eq. (5.15) to:

$$\begin{aligned} E[e(\delta_i^{t+1})] = 1 - r_i + v_i &\left(\frac{|A_i| r_i - 1}{|A_i| - 1}\right) \\ &+ e(\delta_i^t) \left(r_i - l_i + v_i \left(\frac{|A_i|(l_i - r_i) + l_i - c_i}{|A_i| - 1}\right)\right) \end{aligned} \quad (5.16)$$

Eq. (5.16) is a difference equation that can be used to determine the expected error of the agent at any time by simply using $E[e(\delta_i^{t+1})]$ as the $e(\delta_i^t)$ for the next iteration. While it might look complicated, it is just the function for a line $y = mx + b$ where $x = e(\delta_i^t)$ and $y = e(\delta_i^{t+1})$. Using this observation, and the fact that $e(\delta_i^{t+1})$ will always be between 0 and 1, we can determine that the final convergence point for the error is the point where Eq. (5.16) intersects the line $y = x$. The only exception is if the slope equals -1, in which case we will see the error oscillating between two points.

By looking at Eq. (5.16) we can also determine that there are two "forces" acting on the agent's error: volatility and the agent's learning abilities. The volatility tends to increase the agent's error past its current value while the learning reduces it. We can better appreciate this effect by separating the v_i terms in Eq. (5.16) and plotting the v_i terms (volatility) and the rest of the terms (learning) as two separate lines. By definition, these will add up to the line given by Eq. (5.16). We have plotted these three lines and traced a sample error progression in Figure 5.4. The error starts at .95 and then decreases to eventually converge to .44. We notice the learning curve always tries to reduce the agent's error, as confirmed

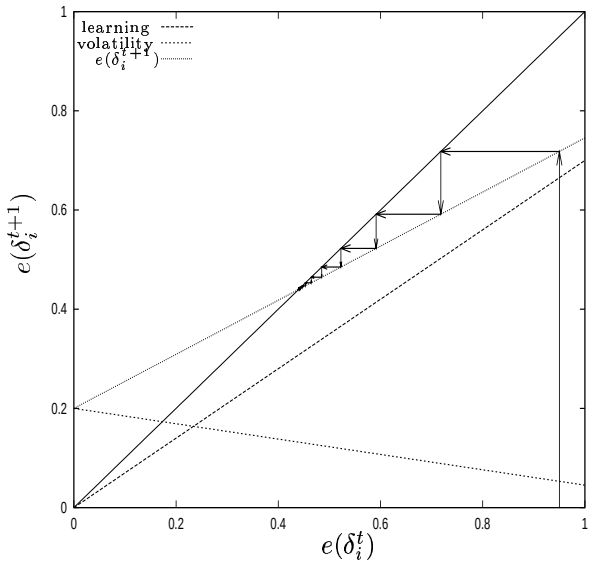


Figure 5.4: Error progression for agent i, assuming a fixed volatility $v_i = .2$, $c_i = 1$, $l_i = .3$, $r_i = 1$, $|A_i| = 20$. The error converges to .44.

by the fact that its line always falls below $y = x$. Meanwhile, the volatility adds an extra error. This extra error is bigger when the agent's error is small since, any change in the target function is then likely to increase the agent's error.

5.4 Volatility and Impact

Equation (5.16) is useful for determining the agent's error when we know the volatility of the system. However, it is likely that this value is not available to us (if we knew it we would already know a lot about the dynamics of the system). In this section we determine the value of v_i in terms of the other agents' changes in their decision functions. That is, in terms of $\mathbf{Pr}[\delta_j^{t+1} \neq \delta_j^t]$, for all other agents j.

In order to do this we first need to define the **impact** I_{ji} that agent j's changes in its decision function have on i's target function.

$$\forall_{w \in W}\ I_{ji} = \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) \mid \delta_j^{t+1}(w) \neq \delta_j^t(w)] \tag{5.17}$$

We can now start to define volatility by first determining that, for two agents i and j

$$\begin{aligned} \forall_{w \in W}\ v_i^t &= \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w)] \\ &= \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) \mid \delta_j^{t+1}(w) \neq \delta_j^t(w)] \cdot \mathbf{Pr}[\delta_j^{t+1}(w) \neq \delta_j^t(w)] \\ &+ \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) \mid \delta_j^{t+1}(w) = \delta_j^t(w)] \cdot \mathbf{Pr}[\delta_j^{t+1}(w) = \delta_j^t(w)] \end{aligned} \tag{5.18}$$

The reader should notice that volatility is no longer constant; it varies with time (as recorded by the superscript). The first conditional probability in Eq. (5.18) is just I_{ji}. The second one we will set to 0, since we are specifically interested in MASs where the volatility arises *only* as a side-effect of the other agents' learning. That is, we assume that agent i's target function changes only when j's decision function changes. For cases with more than two agents, we similarly assume that one agent's target function changes only when some other agent's decision function changes. That is, we ignore any other outside influences that might make the agent's target function change.

We can then simplify Eq. (5.18) and generalize it to N agents, under the assumption that the other agents' changes in their decision functions will not cancel each other out, making δ_i^t stay the same as a consequence. v_i^t then becomes

$$\begin{aligned} \forall_{w \in W}\ v_i^t &= \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w)] \\ &= 1 - \prod_{j \in N_{-i}} (1 - I_{ji}\mathbf{Pr}[\delta_j^{t+1}(w) \neq \delta_j^t(w)]) \end{aligned} \tag{5.19}$$

We now need to determine the expected value of $\mathbf{Pr}[\delta_j^{t+1}(w) \neq \delta_j^t(w)]$ for any agent. Using i instead of j we have

$$\begin{aligned} &\forall_{w \in W}\ \mathbf{Pr}[\delta_i^{t+1}(w) \neq \delta_i^t(w)] \\ &\quad = \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_i^{t+1}(w) \neq \Delta_i^t(w) \,|\, \delta_i^t(w) \neq \Delta_i^t(w)] \\ &\quad + \mathbf{Pr}[\delta_i^t(w) = \Delta_i^t(w)] \cdot \mathbf{Pr}[\delta_i^{t+1}(w) \neq \Delta_i^t(w) \,|\, \delta_i^t(w) = \Delta_i^t(w)] \end{aligned} \tag{5.20}$$

where the expected value is:

$$E[\mathbf{Pr}[\delta_i^{t+1}(w) \neq \delta_i^t(w)]] = c_i e(\delta_i^t) + (1 - r_i) \cdot (1 - e(\delta_i^t)) \tag{5.21}$$

We can then plug Eq. (5.21) into Eq. (5.19) in order to get the expected volatility

$$E[v_i^t] = 1 - \prod_{j \in N_{-i}} 1 - I_{ji}(c_j e(\delta_j^t) + (1 - r_j) \cdot (1 - e(\delta_j^t))) \tag{5.22}$$

We can use this expected value of v_i^t in Eq. (5.16) in order to find out how the other agents' learning will affect agent i. In MASs that have identical learning agents (i.e., their c, l, r, and I rates are all the same and they start with the same initial error) we can replace the multiplier in Eq. (5.22) with an exponent of $|N| - 1$. We use this simplification later in Section 5.7.2.

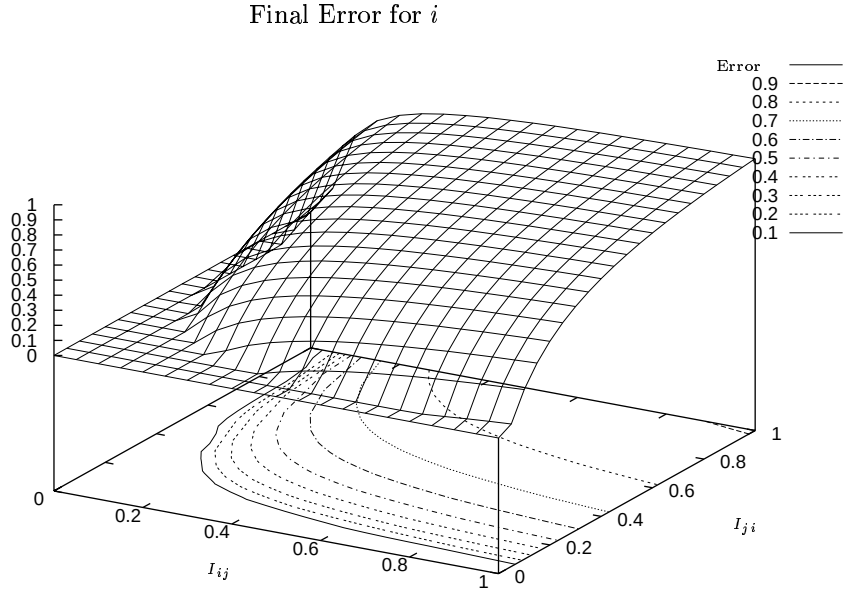


Figure 5.5: Plot of Final Error for agent i, given $l_i = l_j = .2$, $r_i = r_j = 1$, $c_i = c_j = 1$, $|A_j| = |A_i| = 20$.

5.5 An Example with Two Agents

In a MAS with just two agents i and j, we can use Eq. (5.22) to rewrite Eq. (5.16) as

$$\begin{aligned} E[e(\delta_i^{t+1})] &= 1 - r_i + I_{ji}(c_j e(\delta_j^t) + (1 - r_j) \cdot (1 - e(\delta_j^t))) \left(\frac{|A_i| r_i - 1}{|A_i| - 1} \right) \\ &+ e(\delta_i^t) \left(r_i - l_i + I_{ji}(c_j e(\delta_j^t) + (1 - r_j) \cdot (1 - e(\delta_j^t))) \right. \\ &\left. \cdot \left(\frac{|A_i|(l_i - r_i) + l_i - c_i}{|A_i| - 1} \right) \right) \end{aligned} \tag{5.23}$$

We can now use Eq. (5.23) to plot values for one particular example. Let us say that $l_i = l_j = .2$, $c_i = c_j = 1$, $r_i = r_j = 1$, $|A_j| = |A_i| = 20$ and we let the impacts I_{ij} and I_{ji} vary between zero and one. Figure 5.5 shows the final error, after convergence, for this situation. It shows an area where the error is expected to be below .1, corresponding to low values for either I_{ij}, I_{ji} or both. This area represents MASs that are loosely coupled, i.e., one agent's change in behavior does not significantly affect the other's target function. In these systems we can expect that the error will eventually[1] reach a value close to zero. We see that as the impact increases the final error also increases, with a fairly abrupt transition

[1] Notice that we are not representing how long it takes for the error to converge. This can easily be done and is just one more of the parameters our theory allows us to explore.

between a final error of 0 and bigger final errors. This abrupt transition is characteristic of these types of systems where there are tendencies for the system to either converge or diverge, and both of them are self-enforcing behaviors. Notice also that the graph is not symmetric—I_{ij} has more weight in determining i's final error than I_{ji}. This result seems counterintuitive, until we realize that it is j's error that makes it hard for i to converge to a small error. If I_{ij} is high then, if i has a large error then j's error will increase, which will make j change its decision function often and make it hard for i to reduce its error. If I_{ij} is low then, even if I_{ji} is high, j will probably settle down to a low error and as it does i will also be able to settle down to a low error.

If we were about to design a MAS we would try to build it so that it lies in the area where the final error is zero. This way we can expect all agents to eventually have the correct behavior. We note that a substantial percentage of the research in DAI and MAS deals with taking systems that are not inherently in this area of near-zero error and designing protocols and rules of encounter so as to move them into this area, e.g., (Rosenschein and Zlotkin, 1994).

The fact that the final error is 1 for the case with $I_{ij} = I_{ji} = 1$ can seem non-intuitive to readers familiar with game theory. In game theory there are many games, such as the "matching game" from Section 5.2.1, where two agents have an impact of 1 on each other. However, it is known (Binmore, 1988) that, in these games, two learning agents will eventually converge to one of the equilibria (if there are any), making their final error equal to 0. This is certainly true, and it is exactly what we showed in Section 5.2.1. The same result is not seen in Figure 5.5 because the figure was plotted using our simplified Equation. (5.16), which makes the simplifying independence assumption given by Eq. (5.14). This assumption cannot be made in games such as the matching game because, in these games, there is a correlation between the correctness of each of the agents actions. Specifically, in the matching game it is always true that both agents are either correct, or incorrect, but it is never true that one of them is correct while the other one is incorrect, i.e., either they matched, or they did not match.

Another view of the system is given by Figure 5.6 which shows a vector plot of the agents' errors. We can see how the bigger errors are quickly reduced but the pace of learning decreases as the errors get closer to the convergence point. Notice also that an agent's error need not change in a monotonic fashion. That is, the error can get bigger for a while before it starts to get smaller.

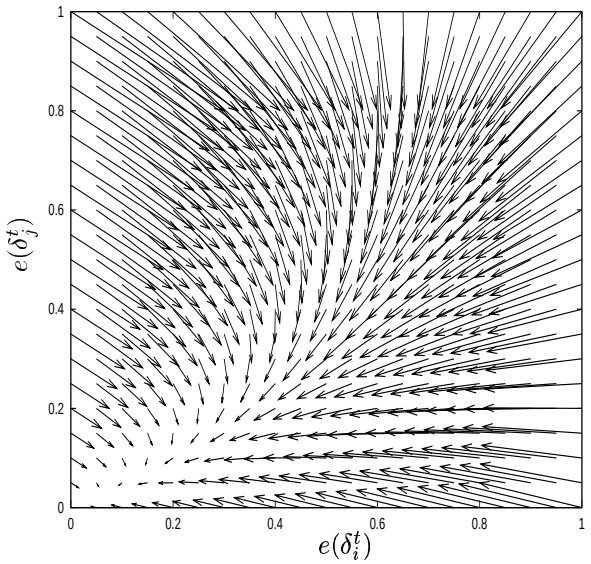


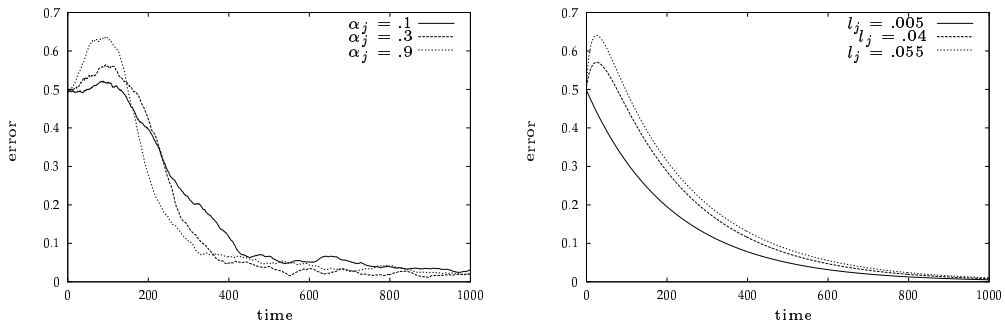
Figure 5.6: Vector plot for $e(\delta_i^t)$ and $e(\delta_j^t)$, where $|A_i| = |A_j| = 20$, $l_i = l_j = .2$, $r_i = r_j = 1$, $c_i = .5$, $c_j = 1$, $I_{ij} = .1$, $I_{ji} = .3$.

5.6 A Simple Application

In order to demonstrate how our theory might be used, we tested it on a simple market-based MAS. The game consists of three agents, one buyer and two seller agents i and j. The buyer will always buy at the cheapest price—but the sellers do not know this fact. The sellers can bid any one of 20 prices in an effort to maximize their profits. In this system we had one good being sold ($|W| = 1$).

As predicted by economic theory, the price in this system settles to the sellers' marginal cost, but it takes time to get there due to the learning inefficiencies. The sellers we use are 0-level reinforcement learning agents (Vidal and Durfee, 1998a). We experimented with different α_j rates[2] for the reinforcement learning of agent j, while keeping $\alpha_i = .1$ fixed, and plotted the running average of the error of agent i. A comparison is shown in Figure 5.7. Figure 5.7(a) gives the experimental results for three different values of α_j. It shows i's average error, over 100 runs, as a function of time. Since both sellers start with no knowledge, their initial actions are completely random which makes their error equal to .5. Then, depending on α_j, i's error will either start to go down from there or will first go up some and then down. Eventually, i's error gets very close to 0, as the system reaches a market equilibrium.

[2] α is the relative weight the algorithm gives to the most recent payoff. $\alpha = 1$ means that it will forget all previous experience and use only the latest payoff to determine what action to take.



(a) Experiment

(b) Theory

Figure 5.7: Comparison of observed and predicted error.

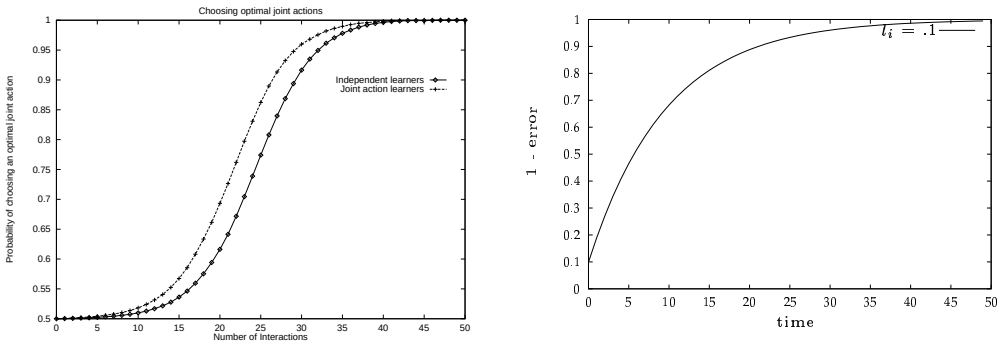
We tried to predict this behavior using Eq. (5.23). Based on the game description, we set $|A_i| = |A_j| = 20$, since there were 20 possible actions. We let $r_i = r_j = 1$ because, in reinforcement learning with fixed payoffs once an agent is taking the correct action it will never change its decision function to take a different action. The agent might, however, still take a wrong action but only when its exploration rate dictates it.

We then let $I_{ij} = I_{ji} = 1/10$ based on the rough calculation that each agent has an equal probability of bidding any one of the 20 prices. If i is bidding 20 then, if j changes its bid price, i will have to change its price with probability 19/20 (i.e., it will change it unless j just changed to 20). If i is bidding 19 then this probability is 18/20, and so on. The average of all of these probabilities is 1/10. A more precise calculation of the impact would require us to actually run the system and find it via experimentation.

Finally, we chose $l_i = l_j = c_i = c_j = .005$ for the first curve (i.e., the one that compares with $\alpha_j = .1$). We knew that for such a low α_j the learning and change rate should be the same. The actual value was chosen via experimentation. The resulting curve is shown in Figure 5.7(b). At this moment, we do not possess a formal way of deriving learning and change rates from α-rates.

For the second curve ($\alpha_j = .3$) we knew that, since only α_j had changed from the first experiment, we should only change l_j and c_j. In fact, these two values should only be increased. We found their exact values, again by experimentation, to be $l_j = .04$, $c_j = .4$. For the third curve we found the values to be $l_j = .055$, $c_j = .8$.

One difference we noticed between the experimental and the theoretical results is that the experimental results show a longer delay before the error starts to decrease. We attribute this delay to the agent's initially high exploration rate. That is, the agents initially start by taking all random actions but progressively reduce this rate of exploration. As the explo-



(a) Experiment

(b) Theory

Figure 5.8: Comparing theory (b) with results from Claus and Boutilier (a) (Claus and Boutilier, 1997).

ration rate decreases the discrepancy between our theoretical predictions and experimental results is reduced.

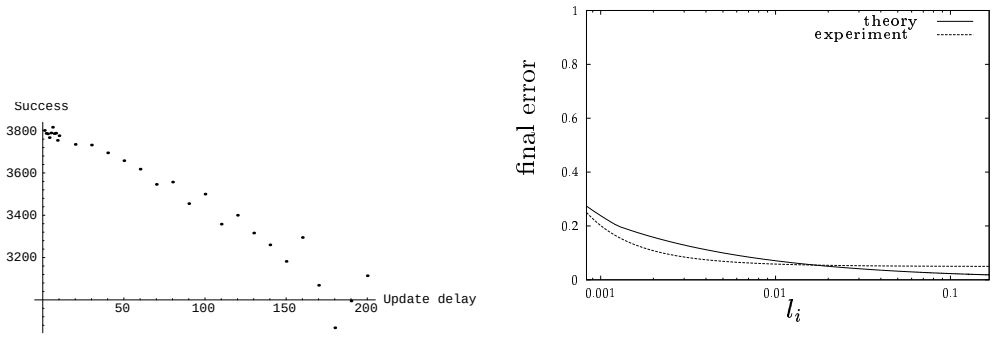
In summary, while it is true that we found l_j and c_j by experimentation, all the other values were calculated from the description of the problem. Even the relative values of l_j and c_j follow the intuitive relation with α_j that, as α_j increases so does l_j and (even more) c_j. We believe that this experiment provides solid evidence that our theory can be used to approximately determine the quantitative behaviors of MASs with learning agents.

5.7 Application of our Theory to Experiments in the Literature

In this section we show how we can apply our theory to experimental results found in the AI and MAS literature. While we will often not be able to completely reproduce the authors' results exactly, we believe that being able to reproduce the flavor and the main quantitative characteristics of experimental results in the literature shows that our theory can be widely applied and used by practitioners in this area of research.

5.7.1 Claus and Boutilier

In (Claus and Boutilier, 1997), Claus and Boutilier study the dynamics of a system that contains two reinforcement learning agents. Their first experiment put the two agents in a matching game exactly like the one we describe in Section 5.2.1 with $|A_i| = |A_j| = 2$. Their results showed the probability that both agents matched (i.e., 1 - $e(\delta_i^t)$) as time progressed.



(a) Original Experiment

(b) Theory and Experiment

Figure 5.9: Comparing theory (b) with results from Shoham and Tennenholtz (a) (Shoham and Tennenholtz, 1997).

Since they were using two reinforcement learning agents, it was not surprising that the curve they saw, seen in Figure 5.8(a), was nearly identical to the curve we saw in our experiments with the two buying agents (Figure 5.7(a) with $\alpha_j = \alpha_i = .1$, except upside-down).

We can reproduce their curve using our equation for the matching game Eq. (5.11). The results can be seen in Figure 5.8(b). Our theory again fails to account for the initial exploration rate. We can, however, confirm that by time 15 their Boltzmann temperature (the authors used Boltzmann exploration) had been reduced from an initial value of 16 to 3.29 and would keep decreasing by a factor of .9 each time step. This means that by time 15 the agents were, indeed, starting to do more exploitation (i.e., reduce their error) while doing little exploration.

5.7.2 Shoham and Tennenholtz

In (Shoham and Tennenholtz, 1997), Shoham and Tennenholtz investigate how learning agents might arrive at social conventions. The authors introduce a simple learning algorithm (strategy-selection rule) called *highest cumulative reward* (HCR) which their agents use for learning these conventions. Shoham and Tennenholtz also provide the results of a series of experiments using populations of learning agents. We try to reproduce the results they present in their Section 4.1 where they study the "coordination game" which is similar to our matching game, but with only two actions.

The experiment in question involves 100 agents, all of them identical and all of them using HCR. At each time instant the agents take one of two available actions. The aim is

for every pair of chosen agents to take the same action as each other. The authors do not state how the agents are paired up so we will assume that they are all randomly matched. The agents update their behavior (i.e., apply HCR) after a given delay. The authors try a series of delays (from 0 to 200) and show that increasing the update delay decreases the percentage of trials where, after 1600 iterations, at least 95% of the agents reached a convention. The authors show surprise at finding this phenomenon. Their results are reproduced in Figure 5.9(a) (cf. Figure 1 in their paper).

The number of actions for all agents is easily set to $|A_i| = 2$, which implies that we must have $l_i = c_i$. By examining HCR, it is easy to determine that $r_i = 1$ (i.e if an agent took the right action, it will only get more support for it). At first intuition, one's impulse is to set $I_{ij} = 1$ for every pair of agents i and j. However, since there are 100 of them and only pairs of them interact at every time instance, the real impact is $I_{ij} = 1/99$.

We will now convert from their units of measurement into ours. In Figure 5.9(a) we can see that their x-axis is called the *update delay*, which we will refer to as d. This value is the number of time units that pass before the agent is allowed to learn. For $d = 0$ the agent learns after every interaction (i.e., on every time t), while for $d = 200$ the agent takes the same action for 200 time instances and only learns after every 200 iterations. This means that we must set $l_i = \frac{1}{p(d+1)}$ where $p > 0$. The value of p depends on their learning algorithm's performance, but we know that it must be a small number (< 50) greater than 0. Through some experimentation we settled on $p = 6$ (other values close to this one give similar results). Since in their graph they look at $0 \leq d \leq 200$, we must then look at l_i where $\frac{1}{1206} \leq l_i \leq \frac{1}{6}$ Finally, we find the value of d in terms of l_i to be

$$d = \frac{1}{pl_i} - 1 \tag{5.24}$$

The y-axis of Figure 5.9(a) is the *success*, i.e., number of trials, out of 4000, where at least 95% of the agents reached a convention. We will refer to this value as s. We know that in $s/4000$ percentage of the trials *at least* 95% of the agents have error close to 0 (i.e., reaching a convention means that the agents take the right action almost all the time), and for the rest of the trials the error was greater. We can approximately map this to an error by saying that in $s/4000$ of the trials the error was 0 (a slight underestimate), while in $1 - s/4000$ of the trials the error was 1 (a slight overestimate). We add these two up (the 0 makes the first term disappear) and arrive at an equation that maps s to $e(\delta_i^t)$.

$$e(\delta_i^t) \approx \left(\frac{4000 - s}{4000}\right) \tag{5.25}$$

The mapping from d to s is given by their actual data. Their data can be fit by the

following function:

$$s = 3900 - 4d - \frac{(d-100)^2}{100} \tag{5.26}$$

Plugging Eq. (5.24) into Eq. (5.26), and the result into Eq. (5.25), we finally arrive at a function that maps their experimental results into our units:

$$\text{Final error} = \frac{4000 - \left(3900 - 4(1/pl_i - 1) - \left(\frac{(1/pl_i - 1 - 100)^2}{100}\right)\right)}{4000} \tag{5.27}$$

for the range $\frac{1}{1206} \leq l_i \leq \frac{1}{6}$.

Now that we have values for c_i, l_i, r_i, I_{ij}, $|A_i|$, a range for l_i and an equation that maps their experimental results into our units, we can plot both functions, as seen in Figure 5.9(b). The x-axis was plotted on a log-scale in order to better show the shape of the experiment curve, otherwise it would appear mostly as a straight line. For our theory curve we used Equations (5.16) and (5.22), and iterated for 1600 time units, just like in the experiment, and plotted the error at that point. For the experiment curve we used Eq. (5.27). We plotted both of these curves in the specified range for l_i. The reader will notice that our theory was able to make precise quantitative predictions. The maximum distance from our theory curve to the experimental curve is .05, which means that our predictions for the final error were, at worst, within 5% of the experimental values. Also, an error of about 5% was introduced when mapping from their success percentage s to our error.

5.7.3 Others

There are several other examples in the literature where we believe our theory can be successfully applied. In (Ishida, 1997) chapter 3.7, Ishida gives results of an experiment where two agents try to find each other in a 100 by 100 grid. He shows that if the grid has few obstacles it is faster if both agents move towards each other, while if there are many obstacles it is faster if one of the agents stays still while the other one searches for it. We believe that the number of obstacles is proportional to the change rate that the agents experience and, perhaps, to the impact that they have on each other. When there are no obstacles the agents never change their decision functions (because their initial Manhattan heuristics lead them in the correct path). As the number of obstacles increases, the agents will start to change their decision functions as they move, which will have an impact on the other agent's target function. If, however, one of them stays put, this means that his change rate is 0 so the other agent's target function will stay still and he will be able to reach his target (i.e., error 0) quicker.

Notice that the problem of a moving target that Ishida studies is different from the problem of a moving target function which we study. It is, however, interesting to note

their similarities and how our theory can be applied to some aspects of that domain.

Another possible example is given by Sen et. al. in (Sen et al., 1994). They show two Q-learning agents trying to cooperate in order to move a block. The authors show how different α rates (β in their paper) affect the quality of the result that the agents converge to. This quality roughly corresponds to our error, except for the fact that their measurements implicitly consider some actions to be better than others, while we consider an action to be either correct or incorrect. This discrepancy would make it harder to apply our theory to their results but we still believe that a rough approximation is possible.

5.8 Summary

We have presented a framework for studying and predicting the behavior of MASs composed of learning agents. We believe that this framework captures the most important parameters that describe an agents' learning and the system's rules of encounter. Various simple applications of the theory were given, along with a comparison with experimental results using reinforcement learning agents. The theoretical predictions were shown to closely match the experimental results. We also reproduced experimental results published by others. These results allow us to more confidently state the effectiveness and accuracy of our theory in predicting the expected error of machine learning agents in MASs.

However, since our theory describes an agent's behavior at a high-level (i.e., the agent's error), it is not capable of making system-specific predictions (e.g., predicting the particular actions that are favored). These types of system-specific predictions can only be arrived at by actually implementing populations of such agents and testing their behaviors. This is the approach we will take in Chapter 7.

Finally, while we have given some examples as to how learning rates can be determined for particular machine learning implementations, we do not have any general method for determining these rates. However, we can, under certain circumstances, get a lower bound for the learning rate. We show how to determine this bound in Chapter 6.

CHAPTER 6

Learning Recursive Agent Models

In Chapter 5, we presented the CLRI framework which used a set of parameters to model the learning abilities of an agent. We did not commit to any specific implementation of the agent's knowledge since the CLRI framework works under the understanding that an agent's behavior can always be described by a decision function $\delta_i^t(w)$. It does not matter whether or not the knowledge is implemented as a series of recursive agent models.

In order for the CLRI framework to be applied to a particular MAS system, we need to determine values for the learning, change, and retention rates, along with the impact. These rates depend on the particular learning algorithms the agents use. Therefore, determining their exact values will depend on the specific implementation of the agents. We determined these values, in Chapter 5, for a couple of example MASs. It would, however, be useful to be able to approximate these values, specifically the learning rate, without having to delve into the actual agent implementation. That is, as designers of agents for MASs, we would like to have a way of quickly estimating how fast two or more possible agent designs (e.g., a 0-level reinforcement learning agent and a 1-level supervised learning agent) learn, without having to go through a detailed analysis of the implementation of their learning algorithms.

In this chapter, we concentrate exclusively on recursive modeling agents, as described in Chapter 3.2. We show how to calculate the size of a recursive modeling agent's learning problem, which we then use to get a lower bound on the agent's learning rate, and for comparing two similarly powerful learning agents. We start by calculating the *sample complexity* of an agent's learning problem, as defined in computational learning theory (Kearns and Vazirani, 1994). The sample complexity gives us a loose upper bound on the number of examples (i.e., events) a consistent learning agent must observe before arriving at a Probably Approximately Correct (PAC) hypothesis, assuming that the target function stays fixed. We then use this sample complexity to derive a lower bound on a consistent learning agent's learning rate l_i. In general, higher sample complexities correspond to lower learning rates, and *vice versa*. We also point out that, while these are only loose bounds,

they are very useful for comparing two or more agents which have similarly powerful learning algorithms.

However, in order to apply computational learning theory results to our recursive modeling agents, we must first show how we can consider these agents, who are learning several decision functions at the same time, as simple learning agents. We do this by assuming that the learning of each of the nested decision functions takes place in parallel. That is, an agent that is learning nested models can be thought of as simply having several learning problems (e.g., learning the model of agent x, learning the model of agent y's model of z, etc.) all of which occur effectively in parallel. All the learning can be done in parallel since we continue to assume that, after each time step, the agent can observe the actions of all the other agents. We also assume that there is enough time between time t and $t+1$ for the agent to update all its models of other agents. This amounts to ignoring the computational costs of learning.

Typically, however, an agent with nested models will start off "knowing" some of these nested models, while other models will be learned via observation and/or experience. The fact that an agent has more knowledge means that there is less for it to learn, so it will probably have a smaller sample complexity. In this chapter, we also show how to calculate the size of the hypothesis space (and, therefore, the sample complexity) for recursive modeling agents that already have some built-in knowledge. For example, we show how a designer that has $\delta_{ij}(w)$ knowledge can use it to reduce the size of the hypothesis space for a 0-level agent that he plans to build.

6.1 Applying Computational Learning Theory to Nested Modeling Agents

In Chapter 5, we gave an equation that can be used to calculate the expected error of any k-level agent, given certain information about the MAS and the learning capabilities of the agents in it. However, while the equation is accurate, it is also impossible to calculate the error at time t directly. The only way to determine how long it will take for the error to drop below some value ϵ is to iterate the error functions until the desired result is reached.

We now show how to apply computational learning theory to determine upper bounds on the time it takes to learn the different types of knowledge, i.e., the time it takes for the error to drop below some value ϵ. We then translate these upper bounds into lower bounds on the learning rate of a consistent learner.[1] The results we give have the advantage that they

[1] A consistent learner is an agent that, once it has learned some state-to-action mapping, will not change this mapping until some further evidence shows that the mapping is incorrect. That is, it never changes mappings that it believes to be correct (Kearns and Vazirani, 1994). In the CLRI framework a consistent learner is an agent with a retention rate of 1.

can be quickly applied to a particular agent in order to get a rough idea of the magnitude of the learning problem the agent faces, given the knowledge it has and the knowledge it must learn. While these values are only bounds on the time and learning rate, not their actual values, the bounds can still be used to compare the relative performance of two or more different agents before implementing them.

We start by characterizing the magnitude of the learning problem for the 0, 1 and 2-level agents. If, for example, we have a 0-level agent that does not have perfect knowledge (i.e., its δ_i^t does not match the target Δ_i^t function), then we know that some or all of its $w \rightarrow a_i$ mappings must be wrong and need to be learned. There are many different machine learning algorithms that the agent can use.

In machine learning research, a distinction is usually made between supervised and reinforcement learning agents (Russell and Norvig, 1995). Supervised learning agents are those that have a supervisor or teacher that tells the agent what action it should have done, right after the agent takes an incorrect action. That is, a supervised learning agent i observes world w, takes action $\delta_i^t(w)$, and then is told by his teacher that he should have done $\Delta_i^t(w)$. Reinforcement learning agents are those that get some kind of reward or payoff after taking an action. The reinforcements are bigger for correct actions (i.e., those given by $\Delta_i^t(w)$) than for incorrect actions. Intuitively, we can see that supervised learning agents will have an easier time (i.e., require fewer examples for) learning the appropriate behavior since they are told exactly what to do after each incorrect action. The reinforcement learning agents, on the other hand, must try all possible actions in order to determine which one has the highest reward.

If a 0-level agent is using some form of supervised learning, then it is trying to learn one of $|A|^{|W|}$ possible 0-level models. If, instead, it is using some form of reinforcement learning, then it is trying to learn one of $|R|^{|W|\cdot|A|}$ possible models, where R is the set of rewards or payoffs it gets ($|R| \geq 2$). All this means is that, if the agent is being taught which actions are better, then it just needs to learn the mapping from state w to action a. Alternatively, if it gets a reward for each action, in each state, then it needs to learn the mapping from state-action (w, a) pairs to their rewards in order to determine which actions lead to the highest reward.

If we have a 1-level agent that is wrong, then the problem could be either in its $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$, or in its $\delta_{ij} : w \rightarrow a_j$ functions. An interesting case is where we assume that the former knowledge is already possessed by the agent. This case is interesting because many MASs have the characteristic that, if an agent knows what actions all the other agents will take, then it can easily determine what action it should take. For example, in an auction-based market system, if I know what all the other agents (both buyers and sellers)

are going to bid, then I can determine how much I should bid in order to maximize my profit. In general, we can expect to find many 1-level agents that have $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$ knowledge built into them.

So, if we assume that $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$ is always correct, this leaves $\delta_{ij} : w \rightarrow a_j$, for all $j \in N_{-i}$, as the only source of discrepancy. Since we have assumed agents can observe each other's actions in all states, we can assume that they learn this knowledge using some form of supervised learning. That is, the observed agent is the teacher because it "tells" others what it does in each w. Therefore, in learning this knowledge the 1-level agent will have $|A|^{|W|}$ possible models of the other agents to choose from. We assume that the 1-level agent will be learning the models of each of the other agents in parallel. That is, after each time step it updates each of its models of all the other agents. We ignore the fact that this requires more computations as we are only interested in how many iterations (time steps) the agent must go through before having a good decision function.

It should be intuitive that the learning problem that such a 1-level agent has is of the same magnitude as the one the 0-level agent using supervised learning has (since both have to learn a mapping from states to actions), but smaller than the reinforcement 0-level agent's problem (since he has to learn the reward associated with each state-action pair). However, we can make this statement a bit more formal by noticing that we can use the size of the hypothesis space to determine the *sample complexity* (Kearns and Vazirani, 1994) of the learning problem. This give us a rough idea of the number of examples that a PAC-learning algorithm would have to see before reaching an acceptable hypothesis (i.e., model).

We start by remembering that the agent's error is given by Eq. (3.1). We also let γ be the upper bound we wish to set on the probability that i has a bad model (i.e., one with $e(\delta_i^t) > \epsilon$). The sample complexity m is then defined as:

$$m \geq \frac{1}{\epsilon}(\ln \frac{|H|}{\gamma}) \tag{6.1}$$

where $|H|$ is the size of the hypothesis (i.e., model) space. m provides a bound on the number of examples or learning opportunities sufficient for any consistent learner to learn any target hypothesis in H. A number m of training examples is sufficient to assure that any consistent hypothesis will be probably (with probability $1 - \gamma$) approximately (with probability ϵ) correct, under the assumption that the target function does not change. We cannot use the sample complexity to actually determine how long it will take for an agent to reach a PAC model, however, since the sample complexity can substantially overestimate the number of samples needed. This is a well known limitation of the sample complexity measure (Mitchell, 1997). However, we can use it to compare the relative performance of two or more learning agents with roughly equivalent learning algorithms.

6.1.1 Bounding Learning Rate with Sample Complexity

While we cannot map the sample complexity m to a particular learning rate l_i, we can use it to put a lower bound on the learning rate for a consistent learning agent. That is, we can find a lower bound for the learning rate of an agent who does not forget anything it has seen, and who is trying to learn a fixed target function. Since the agent does not forget anything it has seen, we can deduce that its retention rate must be $r_i = 1$. Then, since $r_i = 1$ and the target function is not changing, we can determine that its expected error at time t will be:

$$E[e(\delta_i^{t+1})] = e(\delta_i^t) \cdot (1 - l_i) \tag{6.2}$$

We can solve Eq. (6.2) for any time n in order to get:

$$E[e(\delta_i^n)] = e(\delta_i^0) \cdot (1 - l_i)^n \tag{6.3}$$

We now remember that, after m time steps, we expect (with probability $1 - \gamma$) the error to be less than ϵ. Since Eq. (6.3) only gives us an expected error, not a probability distribution over errors, we cannot use it to calculate how likely it is that the agent will have that expected error. That is, we cannot calculate the "probably" (γ) part of probably approximately correct. We will, therefore, assume that the γ chosen for m is small enough so that it will be safe to say that, after m time steps, the error is less than ϵ. In a typical application one uses a small γ because it guarantees a high degree of certainty on the upper bound of the error.

Since we can now safely say that, after m time steps, the error is less than ϵ, we can then deduce that l_i should be small enough such that, if $n = m$, then $E[e(\delta_i^n)] \leq \epsilon$. This is expressed mathematically as:

$$e(\delta_i^0) \cdot (1 - l_i)^m \leq \epsilon \tag{6.4}$$

We solve this equation for l_i in order to get:

$$l_i \geq 1 - \left(\frac{\epsilon}{e(\delta_i^0)}\right)^{1/m} \tag{6.5}$$

This equation is not defined for $e(\delta_i^0) = 0$. However, given our assumption of a fixed target function and $r_i = 1$, we already know, from Eq. (6.2), that if an agent starts with an error of 0 it will maintain this error of 0 for any future time $t > 0$. Therefore, in this case, the choice of a learning rate has no bearing on the agent's error.

Equation (6.5) gives us an lower bound on the learning rate that a consistent learner must have, given that it has sample complexity m, and based on an error ϵ and a sufficiently

small γ. A designer of an agent that uses a reasonable learning algorithm can expect that, if his agent has sample complexity m (for ϵ error), then his agent will have a learning rate of at least l_i, as given by Eq. (6.5). Furthermore, if a designer is comparing two possible agent designs, each with a different sample complexity but both with similarly powerful learning algorithms, he can calculate learning rates for both of them and compare their relative performance with these rates. We will show an example of how this is achieved in Section 6.1.3.

6.1.2 Comparing 0-level and 1-level Agents

Given Eq. (6.1), we can plug in values for one particular case, in order to prove the following theorem:

Theorem 1 *In a MAS where an agent can determine which move it should take given that it knows what all other agents will do, and the agents use some form of reinforcement learning, we find that the sample complexity of the 1-level agents' learning problem is* $O(\ln(|A|^{|W|}))$, *while for the 0-level agents' it's* $O(\ln(|R|^{|A|\cdot|W|}))$ *which is always bigger than the 1-level agent's sample complexity.*

Proof. We saw before that $|H| = |A|^{|W|}$ for the 1-level agent, $|H| = |A|^{|W|}$ for the 0-level with supervised learning, and $|H| = |R|^{|A|\cdot|W|}$ for the 0-level with reinforcement-based learning. From these we can determine that the 1-level agent's sample complexity will be less than the 0-level reinforcement agent as long as $|R| > |A|^{1/|A|}$, which is always true because $|R| \geq 2$ and $|A| > 0$. ∎

This theorem tells us that, in this case, the 1-level agent's learning problem is simpler than the 0-level agent's problem. This case is particularly interesting because it corresponds to MASs where the designer knows "what the agent should do if the agent knows what everyone else will do" and chooses to implement this knowledge directly into the agent. The agent then learns models of the other agents via observations. We have found this case interesting because it describes the knowledge available in many MASs, including auction-based market systems where, if I know what all the other agents (both buyers and sellers) are going to bid, then I can determine how much I should bid in order to maximize my profit.

Applying Theorem 1 to our CLRI framework implies that the 1-level agent in question will probably have a higher learning rate than the 0-level agent. The 1-level agent has fewer behaviors to chose from, therefore the probability that it will choose the correct behavior is higher than for the 0-level agent (that is, assuming the learning algorithms are comparable, i.e., they make about the same number of lucky guesses).

Level	Knowledge	Supervised Learning	Reinforcement Learning
0-level	$\delta_i(w)$	$\|A_i\|^{\|W\|}$	$\|R_i\|^{\|A_i\|\cdot\|W\|}$
1-level	$\delta_i(w, \vec{a}_{-i})$	$\|A_i\|^{\|A_1\|\cdots\|A_{i-1}\|\|A_{i+1}\|\cdots\|A_n\|\|W\|}$	$\|R_i\|^{\|A_1\|\cdots\|A_n\|\|W\|}$
	$\delta_{ij}(w)$	$\|A_j\|^{\|W\|}$	$\|R_j\|^{\|A_j\|\|W\|}$
2-level	$\delta_i(w, \vec{a}_{-i})$	$\|A_i\|^{\|A_1\|\cdots\|A_{i-1}\|\|A_{i+1}\|\cdots\|A_n\|\|W\|}$	$\|R_i\|^{\|A_1\|\cdots\|A_n\|\|W\|}$
	$\delta_{ij}(w, \vec{a}_{-j})$	$\|A_j\|^{\|A_1\|\cdots\|A_{j-1}\|\|A_{j+1}\|\cdots\|A_n\|\|W\|}$	$\|R_j\|^{\|A_1\|\cdots\|A_n\|\|W\|}$
	$\delta_{ijk}(w)$	$\|A_k\|^{\|W\|}$	$\|R_k\|^{\|A_k\|\|W\|}$

Table 6.1: Size of the hypothesis spaces $|H|$ for learning the different sets of knowledge, depending on whether the agent uses supervised or reinforcement learning. A_i is the set of actions agent i can do, n is the number of agents, R_i is the set of rewards for agent i in the given learning problem, and W the set of possible world states.

In fact, we can calculate the size of the hypothesis space $|H|$ for learning all the different decision functions, as summarized in Table 6.1. This table, along with Eq. (6.1), can be used to determine the sample complexity of learning the different decision functions for any agent that uses any form of supervised or reinforcement learning. In this way, we can compare two agents to determine which one will have the more accurate models, on average. Note that, if an agent already has some knowledge (e.g. the 1-level agent we just talked about) then we should not count the complexity of learning it. Also, the reader will notice that some of these complexities are independent of the number of agents ($|N|$). We can make this assumption because we assumed that all agent actions are seen by all agents. This means that an agent can build $w \rightarrow a$ models of each of the other agents, in parallel, and assume everyone else can also do the same. However, the actual computational costs will increase linearly with the number of agents in the system, since the agent will need to maintain a separate model for each one of them. The sample complexities rely on the assumption that, between each action, there is enough time for the agent to update all its models.

A designer of an agent for a MAS can consult Table 6.1 to estimate how long his agent will take to learn accurate models, given different combinations of implemented versus learned knowledge, and supervised versus reinforcement learning algorithms. The designer would first use the table to get the size of the hypothesis space $|H|$ and then use Eq. (6.1) to determine the agent's sample complexity m. The sample complexity gives the designer an upper bound on the number of iterations that a consistent learner needs to experience in order to reach a PAC model. Furthermore, he can also use Eq. (6.5) to get a lower bound

on his agent's learning rate.

As we mentioned before, these could be loose bounds. Real agents will probably perform much better than what is indicated by these bounds. Fortunately, if the designer is comparing two (or more) agents that use similarly powerful learning algorithms, then it is likely that the over/underestimates will be of a similar magnitude. If they are, then a comparison between the two bounds will still lead to the same result as a comparison between the two actual times. The designer can, therefore, use this method to compare two possible agent designs. The quality of the results he derives will depend on the degree to which the learning algorithms are similarly powerful.

Finally, the designer should always remember that we are comparing the number of *iterations* that it takes for an agent to reach a PAC model, not the amount of computation. As we mentioned earlier, we are assuming that all agents have enough time between time t and $t+1$ to update all models. A 1-level agent will probably have to engage in a lot more computation between time steps than a 0-level agent. This extra computation is ignored by our tables since we are only interested (in this chapter) in how many iterations it will take for an agent to reach a PAC model.

6.1.3 Example Application

For example, say I am trying to decide whether to build a 0-level reinforcement learning agent or a 1-level supervised learning agent with $\delta_i(w, \vec{a}_{-i})$ knowledge built-in. I first have to look at my domain and determine the number of actions, states, etc. that the agents will have. These numbers will vary from system to system. A typical set of values (which are representative of those used by our agents in the next chapter) is $|R_i| = 10$, $|A_i| = |A_j| = 20$ and $|W| = 20$. I can then use these values, along with Table 6.1, in order to determine the size of two agents' hypothesis spaces. For the 0-level, agent we have $|R_i|^{|A_i| \cdot |W|} = 10^{20 \cdot 20}$. For the 1-level agent we get $|A_j|^{|W|} = 20^{20}$ (since $\delta_i(w, \vec{a}_{-i})$ is known, the 1-level does not need to learn it). Since the size of the hypothesis space for the 0-level is bigger than for the 1-level, i.e., $10^{400} > 20^{20}$, we can expect the 1-level agent to learn a fixed target function faster than the 0-level agent.

We can also calculate the sample complexity for these two agents. As an example, we can set $\epsilon = .05$ and $1 - \gamma = .98$, then use Eq. (6.1) to derive the sample complexity m for these two agents. With these values, for the 0-level agent we get a value of $m \geq 18500$, and for the 1-level we get $m \geq 2799$. These are the upper bounds on the number of iterations we can expect it will take for the two agents to reach a PAC model (i.e., a low error). If the two learning algorithms we use are comparable (as they are in our experiments) then the actual number of iterations should also be comparable, although lower. Of course, there are

no guarantees. If, say, one of the learning algorithms we implement is so good that it can learn its target function from observing just one example, then the sample complexity will tell us nothing. However, we expect that most learning algorithms will have a performance, as measured by the time steps it takes them to reach a PAC model, that is proportional to their sample complexities.

Given these sample complexities, we can use Eq. (6.5) to determine a lower bound on the agents' learning rates. We first set $e(\delta_i^0) = e(\delta_j^0) = .95$, since this is the error we can expect for an agent with a randomly chosen decision function, i.e., just by guessing randomly, the agents can expect to get 5% of their actions correct. For the 0-level agent we have $l_i \geq 1 - .05^{1/18500}$, which works out to be $l_i \geq .00016$. For the 1-level agent we get $l_i \geq .001$. We can expect the learning rate for an actual implementation of these agents to be higher than these values suggest. Like the sample complexities from which they are derived, the lower bounds on the learning rates can also be used to compare the two agents. We expect the agent with the larger learning rate to learn faster than the other one. The agent with the larger learning rate will, therefore, reach its fixed target function faster and will arrive at an error of 0 sooner than the other one.

We can now implement these agents in order confirm the results that PAC learning theory predicts. We implement the 0-level agent using Q-learning (Watkins and Dayan, 1992). The 1-level agent learns $\delta_j(w)$ by simply remembering what j has done in the past in each world and predicting that it will do the same in the future. The worlds $w \in W$ that both agents experience at each time step are taken from flat probability distributions. Both agents are consistent learners, so the predictions from our tables should apply to them.

Figure 6.1 shows the average error, from 100 runs, for the two agents as a function of time. We notice how the error for the 1-level agent drops much more quickly (i.e., after fewer examples) than for the 0-level agent, as predicted by the sample complexity and learning rate comparisons. The error for the 0-level agent stays at the same level for awhile because the 0-level agent is using reinforcement learning and, therefore, must test all actions before determining which one has the highest reinforcement. This initial error level (95%) is the error we can expect for an agent with a randomly chosen decision function.

Figure 6.2 shows the percentage of the runs (from 100 runs) that have agents with an error greater than .05, as a function of time. This percentage reflects the "probably" $(1-\gamma)$ part of a probably approximately correct model. That is, when less than 2% of the runs have an error greater than .05, we can say that, by that time, any agent will probably (with probability of 2%) have an error less than .05. We notice that the 1-level agents reduce their errors faster than the 0-level agents. The point at which these curves drop below 2% should correspond to the sample complexity (m) values we calculated before. But, as we have

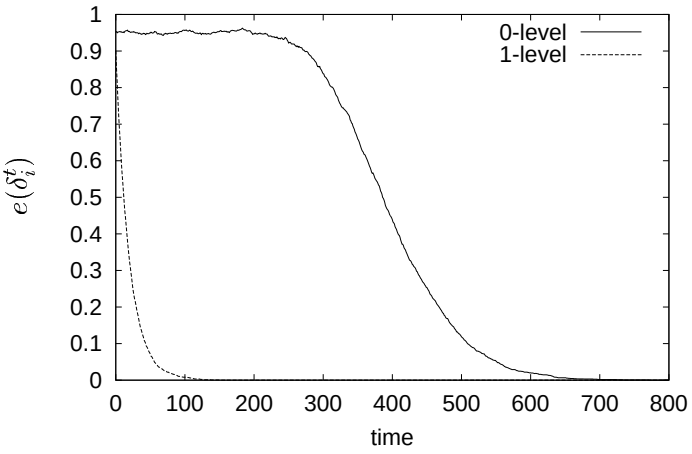


Figure 6.1: Average error for 100 runs.

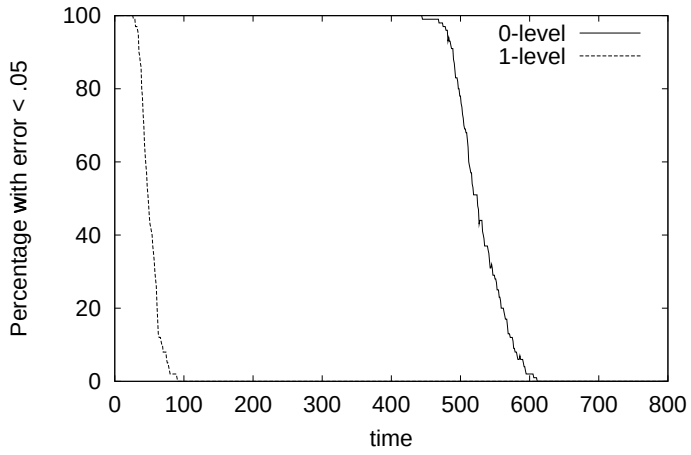


Figure 6.2: Percentage of agents with $e(\delta_i^t) > .05$.

explained, the m values are merely upper bounds. As we can see from this example, our agents can do better than the worst case predicted by their sample complexities, although their performances relative to each other remain the same.

The fact that the 1-level agent learns faster than the 0-level agent means that it requires fewer time steps in order to closely approximate the target function. These results assume that the target function is fixed. However, if the target function was moving, we should also expect the faster learning agent to do better than the slower agent since the faster learning agent can get closer to the moving target function. That is, each time the target function moves, the faster learning agent can move a greater distance and, therefore, get close to the target function. The slower learning agent cannot move such long distances and, therefore, might remain farther away from the target function. This will happen only when the target

function is not moving too fast, so as to nullify the gains from faster learning. We will explain this behavior, both experimentally and theoretically, in Chapter 7.5.3.

6.2 Further Refinement

We can further refine Table 6.1 by noticing that if a designer knows, for example, the value of $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$, then he can actually apply this knowledge when building a 0-level agent. The use of this knowledge can result in a reduction in the size of the hypothesis space for the 0-level agent. For example, if I know that I will always take action a in state w, no matter what the other agents do, then I already know what action to take in state w, so I do not need to learn this piece of knowledge. In general, the reduction can be accomplished by looking at the $\delta_i(w, \vec{a}_{-i})$ function and determining which $w \rightarrow a_i$ pairings are impossible and eliminating these from the hypothesis space of the 0-level modeler. This weeding-out process will get rid of all the impossible combinations. It is equivalent to the iterated dominance method used in game theory. We can think of the process as a way to eliminate possible entries from the table $T_i : W \rightarrow A_i$ of all possible mappings. There are $|A_i|^{|W|}$ elements in the table T_i. The process involves the following rules:

1. If there does not exist an $\vec{a}_{-i} \in \vec{A}_{-i}$ such that $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$ (i.e., the action a_i is never taken in state w, regardless of what the others do), then remove all $w \rightarrow a_i$ mappings from the table T, for this w.

2. If for all $\vec{a}_{-i} \in \vec{A}_{-i}$ it is true that $\delta_i : (w, a_{-i}) \rightarrow x$ (i.e., the agent takes the same action x in state w, no matter what the others do), then, for this w, remove all $w \rightarrow y$ mappings from T, where $y \in A_i$ is any action other than x (i.e., $y \neq x$).

After the application of the previous two rules, we are left with a new table $T_i : W_i' \rightarrow A_i$ with $|W_i'| \leq |W|$, and each $w' \in W_i'$ has a set $A_i^{w'}$ associated with it. We can then determine that, if the new 0-level modeler uses supervised learning, the size of its hypothesis space will be:

$$\prod_{w \in W'} |A_i^w|$$

If, on the other hand, it uses reinforcement learning, the size of its hypothesis space will be:

$$|R_i|^{|T_i|}$$

where the size of $|R_i|$ might also be smaller (especially if $|A_i|$ is smaller since, if the agent is allowed to take fewer actions we can usually expect that it will get fewer possible rewards). Table 6.2 summarizes the size of the hypothesis spaces for learning the different types of

Level	Knowledge	Supervised Learning	Reinforcement Learning
0-level	$\delta_i(w)$	$\prod_{w \in W'_i} \lvert A_i^w \rvert$	$\lvert R_i \rvert^{\lvert T_i \rvert}$
1-level	$\delta_i(w, \vec{a}_{-i})$ $\delta_{ij}(w)$	Previously Known $\lvert A_j \rvert^{\lvert W \rvert}$	Previously Known $\lvert R_j \rvert^{\lvert A_j \rvert \lvert W \rvert}$
2-level	$\delta_i(w, \vec{a}_{-i})$ $\delta_{ij}(w, \vec{a}_{-j})$ $\delta_{ijk}(w)$	Previously Known $\lvert A_j \rvert^{\lvert A_1 \rvert \cdots \lvert A_{j-1} \rvert \lvert A_{j+1} \rvert \cdots \lvert A_n \rvert \lvert W \rvert}$ $\lvert A_k \rvert^{\lvert W \rvert}$	Previously Known $\lvert R_j \rvert^{\lvert A_1 \rvert \cdots \lvert A_n \rvert \lvert W \rvert}$ $\lvert R_k \rvert^{\lvert A_k \rvert \lvert W \rvert}$

Table 6.2: Size of the hypothesis spaces $|H|$ for learning the different sets of knowledge, given that the designer already has $\delta_i(w, \vec{a}_{-i})$ knowledge.

Level	Knowledge	Supervised Learning	Reinforcement Learning
0-level	$\delta_i(w)$	$\prod_{w \in W'} \lvert A_i^w \rvert$	$\lvert R_i \rvert^{\lvert T_i \rvert}$
1-level	$\delta_i(w, \vec{a}_{-i})$ $\delta_{ij}(w, \vec{a}_{-j})$	Previously Known $\prod_{w \in W'_j} \lvert A_j^w \rvert$	Previously Known $\lvert R_j \rvert^{\lvert T_j \rvert}$
2-level	$\delta_i(w, \vec{a}_{-i})$ $\delta_{ij}(w, \vec{a}_{-j})$ $\delta_{ijk}(w)$	Previously Known $\lvert A_j \rvert^{\lvert A_1 \rvert \cdots \lvert A_{j-1} \rvert \lvert A_{j+1} \rvert \cdots \lvert A_n \rvert \lvert W \rvert}$ $\lvert A_k \rvert^{\lvert W \rvert}$	Previously Known $\lvert R_j \rvert^{\lvert A_1 \rvert \cdots \lvert A_n \rvert \lvert W \rvert}$ $\lvert R_k \rvert^{\lvert A_k \rvert \lvert W \rvert}$

Table 6.3: Size of the hypothesis spaces $|H|$ for learning the different sets of knowledge, given that the designer already has $\delta_{ij}(w, \vec{a}_{-j})$ knowledge.

knowledge given that the designer uses his knowledge of the $\delta_i : (w, \vec{a}_{-i}) \rightarrow a_i$ function to reduce the hypothesis spaces of other types of knowledge.

Similarly, if the designer also knows $\delta_{ij}(w, \vec{a}_{-j})$, he creates a reduced table T_j for all other agents. The new hypothesis spaces will then be given by Table 6.3.

6.2.1 Example Application

Say we applied this new refinement technique to the example from Section 6.1.3. We took the $\delta_i(w, \vec{a}_{-i})$ knowledge from the 1-level agent and used it to reduce some of the possibilities in the 0-level agent's $\delta_i(w)$. That is, we used our weeding-out process to eliminate all impossible $w \rightarrow a$ combinations in $\delta_i(w)$. This leads us to new values for the 0-level agent of $|A_i| = 4$, $|R_i| = 2$, $|W_i| = 4$ (where $T_i : W_i \rightarrow A_i$), while the 1-level agent kept its old values of $|A_j| = 20$, $|W| = 20$.

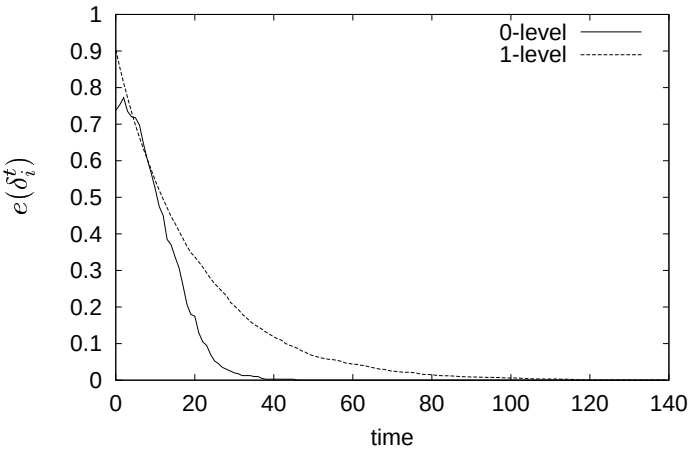


Figure 6.3: Average error for 100 runs. 0-level agent has reduced $|H|$.

With these new numbers we can now follow the same process as in the previous example, in order to calculate the sample complexities for the two agents. The 0-level agent will have $m \geq 299$, while the 1-level agent keeps the old sample complexity of $m \geq 2799$. We notice that now the 0-level agent has a lower sample complexity than the 1-level and, therefore, we expect it to reach a PAC model in fewer iterations than the 1-level agent.

This can be confirmed experimentally by implementing these agents as we did before. Figure 6.3 shows the average error, from 100 runs, as a function of time. We notice that the error for the 0-level drops earlier than for the 1-level agent. The same behavior can also be observed in Figure 6.4, where we show the percentage of the runs (out of 100 runs) where agents have an error greater than .05. In it, we see that the PAC model is reached earlier by the 0-level agents than by the 1-level agents.

The sample complexity predictions are confirmed by both experiments. One difference between this experiment and our previous one is that now the 0-level agent learns the quickest, instead of the 1-level agent, as before. This makes perfect sense since it is the 0-level agent that has the smaller sample complexity. It should also provide a stern warning against making any all-encompassing statements about the relative merits of k-level agents. For example, it might seem tempting to conclude that "all 1-level agents learn faster than 0-level agents", especially given our previous example and theorem. However, while this statement might usually be true, we have just seen one case were it is not true, so such statements should be avoided. The relative performance of two agents (i.e., their learning rates) should always be determined with the use of our tables.

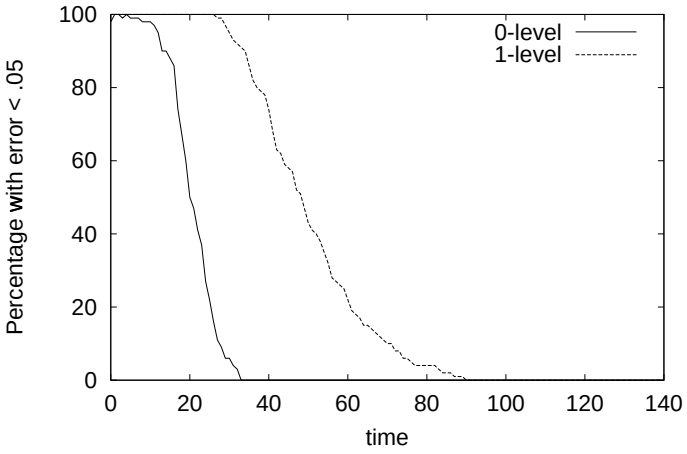


Figure 6.4: Percentage of agents with $e(\delta_i^t) < .05$. 0-level agent has reduced $|H|$.

6.3 Summary

In this chapter we have used computational learning theory to calculate the sample complexities for learning the different nested decision functions. Using these sample complexities, we then derived a lower bound on the learning rate of a consistent learning agent that uses these nested decision functions. This lower bound can be calculated for any recursive modeling agents. The bounds for different agents can be used either by themselves (e.g. a designer might know that any learning rate bigger than x would work, so a lower bound on the learning rate might suffice), or they can be used to compare two or more possible agent designs.

We also gave a theorem that showed, for one interesting case which covers auction-based market systems, that the 1-level agent has a smaller learning problem than a similar 0-level agent. From this we also concluded that in this case, given comparable learning algorithms, the 1-level agent will have a higher learning rate (l_i) than the 0-level agent. We use this result in the next chapter when analyzing experimental results on the profits acquired by 0 and 1-level agents in an economic MAS. We also showed how an agent designer can use our tables to determine the expected learning performance of different k-level agents. The time it takes for the agents to learn good models, as predicted by our tables, was confirmed with experimental results.

We then extended the initial table and built Tables 6.2 and 6.3, which give the size of the hypothesis spaces for a wide number of agents for which the designer has used all

the knowledge he has in order to reduce, as much as possible, the sizes of the hypothesis spaces. These tables provide tighter bounds than Table 6.1 since they take into account the dampening that might exist in some of the knowledge. These tables were applied to our initial example and their predictions were verified with experimental results.

Finally, we remind the reader that the results derived from these tables give us an upper bound on the number of iterations the agent is expected to experience before arriving at a good-enough model under the assumption that the target function is not changing. As such, they should only be used either for comparing two possible agents (under the assumption that they use similarly powerful learning techniques), or for estimating learning rates which would then be used in the CLRI framework. The CLRI framework can then tell us the agent's expected error over time by taking into account the possibility that the agent's target function is changing because the other agents are learning.

CHAPTER 7

Learning in a Market System

In this chapter we study an example market-based MAS. This example domain came about because of our involvement in the University of Michigan Digital Library (UMDL). The UMDL is an open MAS where agents buy and sell information/services from each other (Durfee et al., 1997). One of the reasons this type of system was chosen as a framework for the UMDL was because the implementation of an economy would encourage both content and service providers to offer their services in the UMDL. The system would attract the participation of third-party agents and, as more agents joined, the attraction would become even greater.

The use of a market system is also a good choice because economists have studied markets for many years and can tell us the types of behaviors to expect. They tell us, for example, that in a competitive market at equilibrium the sellers' price will be his marginal cost, and that this will be the equilibrium price. However, these predictions are based on the assumption that the players are humans (i.e., rational agents) and the goods are well defined. In the UMDL the agents are computer programs and they buy/sell information goods and services, e.g., encyclopedia articles, query-answering service, thesaurus service. These agents might not be as smart as humans, and the services rendered by different agents might be of different qualities. In other words, the assumptions made by economists might not be valid in the UMDL.

For example, a seller might realize that the buyers are not paying attention to the quality of service they receive, so he might decide to give them a lower quality service and pocket the difference in the cost. In the real world the buyers would quickly notice the seller's substandard service and deny him their business, foiling his plans for a quick dollar. If we want this type of robust behavior in the UMDL, we need to give the agents some learning abilities. Once we let the agents model other agents, we immediately get into the nested models problem. For example, a seller might start to model other sellers and their models of the buyers in order to predict what the sellers will bid, and bid just below them, thereby

making the highest possible profit and denying the other sellers any revenue. It is not clear when these deeper nested models start losing their effectiveness. Meanwhile, if the other sellers are also learning and changing their behaviors, this means that the agent's models of each one them will be chasing a moving target. It is not clear if the agent's models will ever get close to their target and stay close. In this chapter, these two questions, i.e., what models to use and when does the system converge, are answered via experiments and confirmed with the CLRI framework.

We start the chapter with a description of the UMDL, in Section 7.1, giving the reasons why it is a special kind of economy and some of the traditional economic results do not apply. A model that captures the essential aspects of the commerce protocol is described in Section 7.2. The agent's and their implementations are presented in Section 7.3. Finally, the test results are given in Section 7.4 and analyzed in Section 7.5.

7.1 Description of the UMDL

The UMDL project is a large-scale, multidisciplinary effort to design and build a flexible, scalable infrastructure for rendering library services in a digital networked environment. In order to meet these goals, we chose to implement the library as a collection of interacting agents, each specialized to perform a particular task. These agents buy and sell goods/services from each other, within an artificial economy, in an effort to make a profit. Since the UMDL is an open system, which will allow third parties to build and integrate their own agents into the architecture, we treat all agents as purely selfish.

7.1.1 Implications of the Information Economy

Information goods/services, like those provided in the UMDL, are very hard to compartmentalize into equivalence classes that all agents can agree on. For example, if a web search engine service is defined as a good, then all agents providing web search services can be considered as selling the same good. It is likely, however, that a buyer of this good might decide that seller $s1$ provides better answers than seller $s2$. We cannot possibly hope to enumerate the set of reasons an agent might have for preferring one set of answers (and thus one search agent) over another, and we should not try to do so. It should be up to the individual buyers to decide what items belong to the same good category, each buyer clustering items in possibly different ways.

This situation is even more evident when we consider an information economy rooted in some information delivery infrastructure (e.g., the Internet). There are two main characteristics that set this economy apart from a traditional economy.

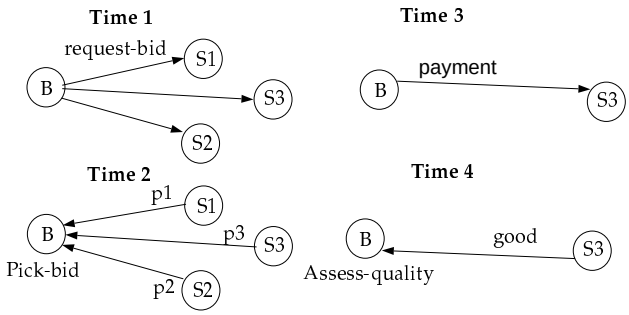


Figure 7.1: View of the protocol. We show only one buyer B and three sellers $S1$, $S2$, and $S3$. At time 1 the buyer requests bids for some good. At time 2 the sellers send their prices for that good. At time 3 the buyer picks one of the bids, pays the seller the amount and then, at time 4, she receives the good.

- There is virtually no cost of reproduction. Once the information is created it can be duplicated virtually for free.
- All agents have virtually direct and free access to all other agents.

If these two characteristics are present in an economy, it is useless to talk about supply and demand, since supply is practically infinite for any particular good and available everywhere. The only way agents can survive in such an economy is by providing value-added services that are tailored to meet their customers' needs. Each provider will try to differentiate his goods from everyone else's while each buyer will try to find those suppliers that best meet her value function. In this chapter, we build agents that can achieve these goals by learning models of other agents and making strategic decisions based on these models. These techniques can also be applied, with variable levels of efficacy, to traditional economies.

7.2 A Simplified Model of the UMDL

In order to capture the main characteristics of the UMDL, and to facilitate the development and testing of agents, we have defined an "abstract" economic model. We define an economic society of agents as one where each agent is either a *buyer* or a *seller* of some particular good. The set of buyers is B and the set of sellers is S. These agents exchange goods by paying some price $p \in P$, where P is a finite set. A buyer is capable of assessing the quality of a good received and giving it some value $q \in Q$, where Q is also a finite set.

The exchange protocol, seen in Figure 7.1, works as follows: When a buyer $b \in B$ wants

to buy a good g, she advertises this fact. Each seller $s \in S$ that sells that good gives his bid in the form of a price p_s^g. The buyer then picks one of these and pays the seller. All agents are made aware of this choice along with the prices offered by all the sellers. The winning seller then returns[1] the specified good. Note that there is no law that forces the seller to return a good of any particular quality. For example, an agent that sells web search services returns a set of hits as its good. Each buyer of this service might determine the service's quality based on the time it took for the response to arrive, the number of hits, the relevance of the hits, or any combination of these and/or other features. Therefore, it would usually be impossible to enforce a quality measure that all buyers can agree with.

It is thus up to the buyer to assess the quality q of the good received. Each buyer b also has a value function $V_b^g(p, q)$ for each good $g \in G$ that she might wish to buy. The function returns a number that represents the value that b assigns to that particular good at that particular price and quality. Each seller $s \in S$, on the other hand, has a cost c_s^g associated with each good he can produce. Since we assume that costs and payments are expressed in the same units (i.e., money) then, if seller s gets paid p for good g, his profit will be $\text{Profit}(p, c_s^g) = p - c_s^g$. The buyers, therefore, have the goal of maximizing the value they get for their transactions, while the sellers have the goal of maximizing their profits.

7.3 Learning Recursive Models

Agents placed in the economic society we just described will have to learn, typically via trial and error, what actions give them the highest expected payoffs, i.e., value or profit, and under which circumstances. In this section we will present techniques that these agents might use to maximize their payoffs.

An important question we wish to answer is: when do agents benefit from having deeper nested models of other agents? It seems intuitive that, ignoring computational costs, the agents with more complete models of others will always do better. We find this to be usually true; however, while there are instances when it is significantly better to have deeper models, there are also instances when the difference is barely noticeable, and instances when it is better to ignore deeper models that are imperfect. These instances are defined in part by the set of other agents present, the agents' capabilities and preferences, and the dynamics of the system. In order to determine precisely what these instances are, and in the hopes of providing a more general framework for studying the effects of increased agent-modeling capabilities within our economic model, we have defined a set of techniques that our agents can use for learning and using models, which we describe in this section. The agents we use

[1] In the case of agent/link failure, each agent is free to set its own timeouts and assess the quality of the never-received good accordingly. Bids that are not received in time will, of course, not be considered.

are 0-level, 1-level and 2-level agents, as described in Chapter 3, but with added learning abilities.

7.3.1 Populating the Knowledge

If agents of different recursive modeling depths had to learn all the knowledge then, since the 0-level agents have a lot less knowledge to learn, they would learn it much faster. However, in the economic domain, it is likely that the designer has additional knowledge which could be incorporated into the agents. The agents we built incorporated extra knowledge along these lines.

We decided that 0-level agents would learn all their knowledge by tracking their actions and the payoffs they got. These agents, therefore, receive no extra domain knowledge from the designers and learn everything from experience. 1-level agents, on the other hand, have *a priori* knowledge of what action each one should take given the actions that others will take. That is, while they try to learn knowledge of the form $\delta_{ij}(w)$ by observing the actions others take (i.e., in a form of supervised learning where the other agents act as tutors), they already have knowledge of the form $\delta_i(w, \vec{a}_{-i})$. In our economic domain, it is reasonable to assume that agents have this knowledge since, in fact, this type of knowledge can be easily generated. That is, if I know what all the other sellers are going to bid, and the prices that the buyer is willing to pay, then it is easy for me to determine which price to bid. We must also point out that, in this domain, the $\delta_i(w, \vec{a}_{-i})$ knowledge cannot be used by a 0-level agent (we explain how a 0-level agent can use $\delta_i(w, \vec{a}_{-i})$ knowledge in Chapter 6.2). If this knowledge had said, for instance, that from some state w agent i will only ever take one of a few possible actions, then this knowledge could have been used to eliminate impossibilities from the $\delta_i(w)$ knowledge of a 0-level agent. However, this situation never arises in our domain because, as we shall see in the following sections, the states used by the agents permit the set of reasonable actions to always be equal to the set of all possible actions. That is, we cannot use $\delta_i(w, \vec{a}_{-i})$ to permanently eliminate any action from an agent's set of possible actions.

The 2-level agents learn their $\delta_{ijk}(w)$ functions from observations of others' actions, under the already stated assumption that there is common knowledge of the fact that all agents see the actions taken by all. The rest of the knowledge, i.e., $\delta_{ij}(w, \vec{a}_{-j})$ and $\delta_i(w, \vec{a}_{-i})$, is built into the 2-level agents a priori. As with 1-level agents, we cannot use the $\delta_{ij}(w, \vec{a}_{-j})$ knowledge to add $\delta_{ij}(w)$ knowledge to a 1-level modeler, because other agents are also free to take any one of the possible actions in any state of the world. There are many reasonable ways to explain how the 2-level agents came to possess the $\delta_{ij}(w, \vec{a}_{-j})$ knowledge. It could be, for instance, that the designer assumed that the other designers would build 1-level

Level	Form of Knowledge	Method of Acquisition
0-level	$\delta_i(w)$	Reinforcement Learning
1-level	$\delta_i(w, \vec{a}_{-i})$	Previously known
	$\delta_{ij}(w)$	Learn from observation
2-level	$\delta_i(w, \vec{a}_{-i})$	Previously known
	$\delta_{ij}(w, \vec{a}_{-j})$	Previously known
	$\delta_{ijk}(w)$	Learn from observation.

Table 7.1: Summary of the forms of knowledge that the different agents have or are trying to learn.

agents with the same knowledge we just described. This type of recursive thinking (i.e., "they will do just as I did, so I must do one better"), along with the obvious expansion of the knowledge structure, could be used to generate k-level agents, but so far we have concentrated only on the first three levels.

The different nested decision functions, and their forms of acquisition, are summarized in Table 7.1. In the following sections, we talk about each one of these agents in more detail and give some specifics on their implementations. Our current model emphasizes transactions over a single good, so each agent is only a buyer or a seller, but cannot be both.

7.3.2 Agents with 0-level Models

Agents with 0-level models must learn everything they know from observations they make about the environment, and from any payoffs they get. In our economic society this means that buyers see the bids they receive and the good received after striking a contract, while sellers see the request for bids and the profit they made (if any). In general, these agents get some input, take an action, then receive some payoff. This framework is the same framework used in reinforcement learning, which is why we decided to use a form reinforcement learning (Sutton, 1988) (Watkins and Dayan, 1992), for implementing learning in our agents.

Both buyers and sellers will use the equations in the next few sections to determine what actions to take. But, with a small probability ϵ, they will choose to explore, instead of exploit, and will pick their actions at random (except for the fact that sellers never bid below cost). The value of ϵ is initially 1 but decreases with time to some empirically chosen, fixed minimum value $\epsilon_{\min}$. That is,

$$\epsilon_{t+1} = \begin{cases} \gamma\epsilon_t & \text{if } \gamma\epsilon_t > \epsilon_{\min} \\ \epsilon_{\min} & \text{otherwise} \end{cases}$$

where $0 < \gamma < 1$ is some annealing factor.

Buyers with 0-level models

A buyer b will start by requesting bids for a good g. She will then receive all bids for good g and will pick the seller:

$$s^* = \arg\max_{s \in S} f^g(p_s^g) \tag{7.1}$$

This function implements the buyer's $\delta_b(w)$ which, in this case, can be rephrased as $\delta_b(p_1 \ldots p_{|S|})$. The function $f^g(p)$ returns the value the buyer expects to get if she buys good g at a price of p. It is learned using a simple form of reinforcement learning, namely:

$$f_{t+1}^g(p) = (1-\alpha) f_t^g(p) + \alpha \cdot V_b^g(p,q) \tag{7.2}$$

Here α is the learning parameter. Within the reinforcement learning literature α is often called the learning rate. We will, however, refer to it as the learning parameter so as to avoid any confusion with the learning rate l_i defined in the CLRI framework. p is the price b pays for the good, and q is the quality she ascribes to it. The learning parameter is initially set to 1 and, like ϵ, is decreased until it reaches some fixed minimum value $\alpha_{\min}$.

Sellers with 0-level models

When asked for a bid, the seller s provides a bid that is greater than or equal to the cost[2] for producing the good being sold (i.e., $p_s^g \geq c_s^g$). The seller chooses the price, from the set of all possible prices, that maximizes his expected profit:

$$p_s^* = \arg\max_{p \in P} h_s^g(p) \tag{7.3}$$

Again, this function encompasses the seller's $\delta_s(g)$ knowledge, where we now have that the states are the goods being sold $w = g$, and the actions are prices offered $a = p$. The function $h_s^g(p)$ returns the profit s expects to get if he offers good g at a price p. It is also learned using reinforcement learning, as follows:

$$h_{t+1}^g(p) = (1-\alpha) h_t^g(p) + \alpha \cdot \text{Profit}_s^g(p) \tag{7.4}$$

where

$$\text{Profit}_s^g(p) = \begin{cases} p - c_s^g & \text{if his bid is chosen} \\ 0 & \text{otherwise} \end{cases} \tag{7.5}$$

[2]We could just as easily have said that the price must be strictly greater than the cost.

7.3.3 Agents with 1-level Models

The next step is for an agent to keep 1-level models of the other agents. This means that it has no idea of what the interior (i.e., "mental") processes of the other agents are, but it recognizes the fact that there are other agents out there whose behaviors influence its payoffs. The agent, therefore, can only model others by looking at their past behaviors and trying to predict, from them, the other agents' future actions. The agent also has knowledge, implemented as functions, that tells it what action to take, given a probability distribution over the set of actions that other agents can take. In the actual implementation, as shown below, the $\delta_i(w, \vec{a}_{-i})$ knowledge takes into account the fact that the $\delta_{ij}(w)$ knowledge is constantly being learned and, therefore, is not correct with perfect certainty.

Buyers with 1-level models

A buyer with 1-level models can now keep a history of the qualities she ascribes to the goods returned by each seller. She can, in fact, remember the last Q qualities returned by some seller s for some good g, and define a probability density function $q_s^g(x)$ over the qualities x returned by s (i.e., $q_s^g(x)$ returns the probability that s returns an instance of good g that has quality x). This function provides the $\delta_{bs}(g)$ knowledge. She can use the expected value of this function to calculate which seller she expects will give her the highest expected value.

$$\begin{aligned} s^* &= \arg\max_{s \in S} E(V_b^g(p_s^g, q_s^g(x))) \\ &= \arg\max_{s \in S} \frac{1}{|Q|} \sum_{x \in Q} q_s^g(x) \cdot V_b^g(p_s^g, x) \end{aligned} \tag{7.6}$$

The $\delta_b(g, q_1 \cdots q_{|S|})$ is given by the previous function which simply tries to maximize the value the buyer expects to get. The buyer does not need to model other buyers since they do not affect the value she gets.

Sellers with 1-level models

Each seller will try to predict what bid the other sellers will submit (based solely on what they have bid in the past), and what bid the buyer will likely pick. A complete implementation would require the seller to remember past combinations of buyers, bids and results (i.e., who was buying, who bid what, and who won). However, it is unrealistic to expect a seller to remember all this since there are at least $|P|^{|S|} \cdot |B|$ possible combinations.

However, a 1-level seller can be implemented by having it remember the last Q prices accepted by each buyer b for each good g, and forming a probability density function $m_b^g(x)$ with these prices. The probability density function $m_b^g(x)$ returns the probability that b

will accept (pick) price p for good g. The expected value of this function provides the $\delta_{sb}(g)$ knowledge. Similarly, the 1-level seller will also need to remember other sellers' last Q bids for good g and use these bids to form $n_s^g(y)$. The probability function $n_s^g(y)$ gives the probability that s will bid y for good g. The expected value of this function provides the $\delta_s(g)$ knowledge. The seller s can now determine which bid maximizes his expected profits.

$$p^* = \arg\max_{p \in P}(p - c_s^g) \cdot \prod_{s' \in \{S-s\}} \sum_{p' \in P} \begin{cases} n_{s'}^g(p') & \text{if } m_b^g(p') \leq m_b^g(p) \\ 0 & \text{otherwise} \end{cases} \tag{7.7}$$

Note that this function also does a small amount of approximation by assuming that s wins whenever there is a tie[3]. The function calculates the best bid by determining, for each possible bid, the product of the profit and the probability that the agent will get that profit. Since the profit for lost bids is 0, we only need to consider the cases where s wins. The probability that s will win can then be found by calculating the product of the probabilities that his bid will beat the bids of each of the other sellers. The function approximates the $\delta_s(g, p_b, p_1 \cdots p_{|S|})$ knowledge.

7.3.4 Agents with 2-level Models

The intentional models we use correspond to the functions used by agents that use 1-level models. The agents' $\delta_i(w, \vec{a}_{-i})$ knowledge has again been expanded to take into account the fact that the deeper knowledge is learned and might not be correct. The $\delta_{ijk}(w)$ knowledge is learned from observation, under the assumption that there is common knowledge of the fact that all agents see the bids given by all agents.

Buyers with 2-level models

Since the buyer receives bids from the sellers, there is no need for her to try to out-guess or predict what the sellers will bid. She is also not concerned with what the other buyers are doing since, in our model, there is an effectively infinite supply of goods. The buyers are, therefore, not competing with each other and do not need to keep deeper models of others.

Sellers with 2-level models

A seller will model other sellers as if they were using the 1-level models. That is, he thinks they will model others using state-to-action mappings and make their decisions using the equations presented in Section 7.3.3. He will try to predict their bids and then try to

[3]The complete solution would have to consider the probabilities that s ties with 1, 2, 3, ... other agents. In order to do this we would need to consider all $|P|^{|S|}$ subsets.

find a bid for himself that the buyer will prefer more than all the bids of the other sellers. His model of the buyer will also be an intentional model. He will model the buyers as though they were implemented as explained in Section 7.3.3. A seller, therefore, assumes that it has the correct intentional models of other agents.

The algorithm he follows is to first use his models of the sellers to predict what bids p_i they will submit. He has a model of the buyer $C(s_1 \cdots s_n, p_1 \cdots p_n) \rightarrow s_i$, that tells him which seller she might choose given the set of all individual bids p_i, each p_i submitted by the respective seller s_i. The seller s_j uses this model to determine which of his bids will bring him higher profit, by first finding the set of bids he can make that will win:

$$P' = \{p_j | p_j \in P, s_j = C(s_1 \cdots s_j \cdots s_n, p_1 \cdots p_j \cdots p_n)\} \tag{7.8}$$

And from these finding the one with the highest profit:

$$p^* = \arg\max_{p \in P'}(p - c_s^g) \tag{7.9}$$

These functions provide the $\delta_s(g, p_b, p_1 \cdots p_{|S|})$ knowledge.

7.3.5 Capturing k-level Learning Abilities with CLRI Framework

Now that we have described how our k-level agents are implemented, we try to capture their learning abilities using the c_i, l_i, r_i, and I_{ij} parameters from our CLRI framework. The specific values for these parameters will depend on the specific exploration rates, α-rates, and size of Q. Their true values could be determined, or at least approximated, with a careful analysis of the agents' learning algorithm. However, since we are interested in how k-level agents perform against x-level agents (where $k \neq x$), we only need to quantify how their respective CLRI parameters differ.

Our 0-level agents use reinforcement learning on $w \rightarrow a$ pairs. The 1-level agents, on the other hand, build models of others agents $\delta_{ij}(w)$. These models are built in parallel for each agent $j \in N_{-i}$. The 1-level agents also have some knowledge built into them in the form of the function $\delta_i(w, \vec{a}_{-i})$. This situation is identical to the one mentioned in Theorem 1 (Chapter 6), which told us that the size of the 0-level agent's hypothesis space is bigger than for the 1-level agent. The 0-level agent needs to learn more since it starts with less knowledge. This means that we can expect the 0-level agent to have a smaller learning rate than a 1-level agent. It will take longer for the 0-level agent to arrive at the correct behavior since it has more behaviors to choose from.

The change rates c_i for both 0 and 1-level agents will be similar since these rates depend on the learning parameters (α) and the exploration rates of the agents. Finally, we can determine that the retention rate r_i for both types is equal to 1 for the time after the

agents' initial exploratory phase, i.e., after they have stopped taking random actions to determine the reinforcement they receive for each action and are into their exploitation phase. The retention rate for the exploitation phase is 1 because, in reinforcement learning with fixed reinforcements, once an agent takes the action with the highest reinforcement he will only get this same reinforcement back and, therefore, continue to take this action. We ignore the initial exploratory phase since it only occurs during the initial stages of the experiment.

7.4 Tests

To determine how much better (or worse) agents with deeper models fare, we have implemented a society of the agents described above which we run to test our hypotheses. In all tests, we have 5 buyers and 8 sellers. The buyers all have the same value function $V_b(p, q) = 3q - p$, which means that if $p = q$ then the buyers will prefer the seller that offers the higher quality. The quality that they perceive is the same only on average, i.e., any particular good might be thought to have quality that is slightly higher or lower than expected. Each seller has a cost equal to the quality of the good it returns. This cost assignment supports the common sense assumption that quality goods cost more to produce. A set of these buyers and sellers is what we call a *population*. We tried various populations; within each population we keep constant each agent's modeling level, each agent's value assessment function, and the qualities returned by each agent. The tests involve a series of such populations, each one containing agents with different modeling levels, and/or sellers with different qualities/costs. We also set $\alpha_{\min} = .1$, $\epsilon_{\min} = .05$, and $\gamma = .99$. We have performed 100 runs for each population of agents, each run consisting of 10000 iterations of the protocol, i.e., market transactions. The lessons presented in the next section are based on the averages of these 100 runs.

7.5 Lessons

From our tests we can discern several lessons about the dynamics of different populations of agents. Some of these lead to methods that can be used to make quantitative predictions about agents' performance, while others make qualitative assessments about the types of behaviors we might expect. We detail these in the next subsections, and summarize them in Table 7.6.

7.5.1 Micro versus Macro Behaviors

In all tests, we find the behavior for any particular run does not necessarily reflect the average behavior of the system. The prices have a tendency to sometimes reach temporary stable points. These *conjectural equilibria*, as described in (Hu and Wellman, 1996), are instances when all of the agents' models are correctly predicting the others' behavior and, therefore, the agents do not need to change their models or their actions. These conjectural equilibrium points are seldom global optima for the agents. If one of our agents finds itself at one of these equilibrium points, since the agent is always exploring with probability ϵ, it will in time discover that this point is only a local optimum (i.e., it can get more profit selling/buying at a different price) and will change its actions accordingly. Only when the price is an equilibrium price[4] do we find that the agents continue to forever take the same actions, leaving the price at its equilibrium point.

In order to understand the more significant macro-level behaviors of the system, we present results that are based on the averages from many runs. While these averages seem very stable, and a good first step in learning to understand these systems, in the future we will need to address some of the micro-level issues. We do notice from our data that the micro-level behaviors (e.g., temporary conjectural equilibria, price fluctuations) are much more closely tied, usually in intuitive ways, to the agents' learning parameters (α) and exploration rates (ϵ). That is, bigger values for both of these lead to more price fluctuations and shorter temporary equilibria.

7.5.2 0-level Buyers and Sellers

This type of population is equivalent to a "blind" auction, where the agents only see the price and the good, but are prevented from seeing who the buyers or sellers were. As expected, we find that an equilibrium is reached as long as all the sellers are providing the same quality. This is the case for population 1 in Figure 7.2. Otherwise, if the sellers offer different quality goods, the price fluctuates as the buyers try to determine which price is correlated to the highest expected value (according to their value functions), and the sellers try to determine which price[5] the buyers favor. In these populations, the sellers offering higher qualities, at higher costs, lose money. Meanwhile, sellers offering lower qualities, at lower costs, earn some extra income by selling their low quality goods to buyers that expect, and are paying for, higher quality.

As more sellers start to offer lower quality, we find that the mean price actually *increases*,

[4]That is, p is an equilibrium price if every seller that can sell at that price (i.e., those whose cost is less than p) does.

[5]Remember, a seller is constrained to return a fixed quality. It can only change the price it charges.

Population	Seller 1	2	3	4	5	6	7	8
1	8	8	8	8	8	8	8	8
2	8	8	8	8	8	8	7	8
3	8	8	8	8	8	6	7	8
4	8	8	8	8	5	6	7	8
5	8	8	8	4	5	6	7	8
6	8	8	3	4	5	6	7	8
7	8	2	3	4	5	6	7	8
8	1	2	3	4	5	6	7	8

Table 7.2: Qualities returned by each seller in the different populations. All agents are 0-level. These populations are used in Figures 7.2 and 7.3.

evidently because price acts as a signal for quality and the added uncertainty makes the higher prices more likely to give the buyer a higher value. We see this in Figure 7.2, where population 1 has all sellers returning the same quality while in each successive population more agents offer lower quality, as detailed in Table 7.2. The price distribution for population 1 is concentrated on 9, but for populations 2 through 6 it flattens and shifts to the right, increasing the mean price. It is only by population 7 that the price starts to shift back to the left, thus reducing the mean price, as seen in Figure 7.3. That is, it is only after a significant number of sellers start to offer lower quality that we see the mean price decrease.

7.5.3 0-level Buyers and Sellers, Plus One 1-level Seller

In these experiments we explore the advantages that a 1-level seller has over a similar 0-level seller. The advantage is non-existent when all sellers return the same quality (i.e., when the prices reached an equilibrium as shown in population 1 in Figure 7.4), but increases as the sellers start to diverge in the quality they return. In order to make these findings useful when building agents, we need a way to make quantitative predictions as to the benefits of keeping 1-level models. It turns out that these benefits can be predicted, not by the population type as we had first guessed, but by the price volatility.

We define *price volatility* as the number of times the selling price changes from one transaction to the next, divided by the total number of transactions. Figure 7.5 shows the linear relation between volatility and the percentage of times the 1-level seller wins. The slope of this line can be easily calculated and the resulting function can be used by a seller agent for making a *quantitative* prediction as to how much he would benefit by switching

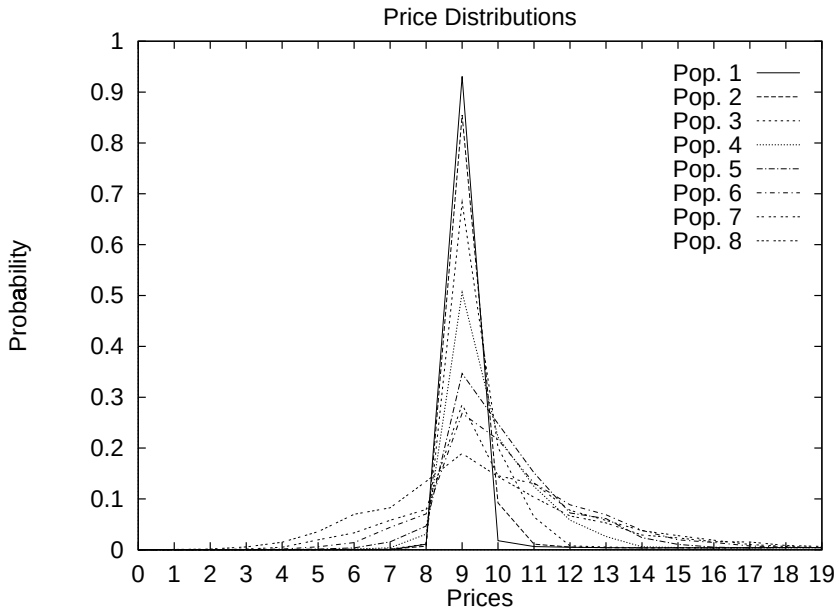


Figure 7.2: Price distributions for populations of 0-level buyers and sellers. The qualities returned by the sellers in each population are shown in Table 7.2.

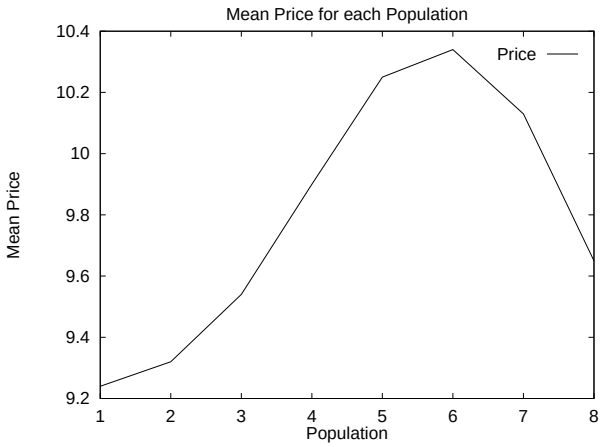


Figure 7.3: Mean price for the populations from Table 7.2.

Population	Seller 1	2	3	4	5	6	7	8
1	2	2	2	2	2	2	2	2
2	2	3	2	2	2	2	2	2
3	2	3	4	2	2	2	2	2
4	2	3	4	5	2	2	2	2
5	2	3	4	5	6	2	2	2
6	2	3	4	5	6	7	2	2
7	2	3	4	5	6	7	8	2

Table 7.3: Qualities returned by each seller in the different populations. Agent 1 is 1-level. Agents 2–8 are 0-level. A seller's cost is equal to the quality it returns. These populations are used in Figures 7.4, 7.5, 7.7.

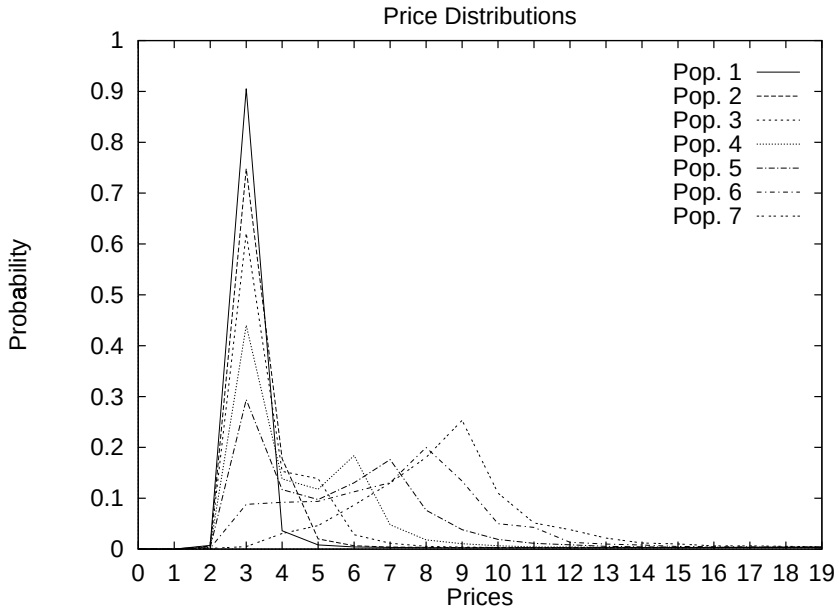


Figure 7.4: Price distributions for populations of 0-level buyers and 0-level sellers plus one 1-level seller. The populations are described in Table 7.3.

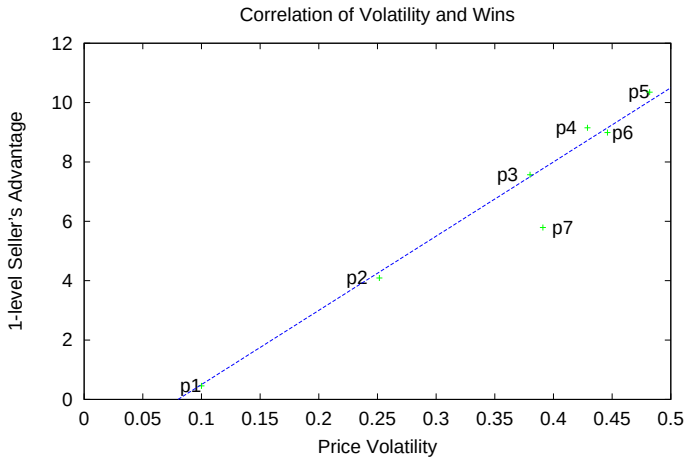


Figure 7.5: Scatter plot of volatility versus the difference between the percentage of time that the 1-level seller wins minus the percentage for the similar 0-level seller. The populations are described in Table 7.3.

to 1-level models. That is, he could measure price volatility, multiply it by the appropriate slope, and the resulting number would be the percentage of times he would win. However, for this to work the agent needs to know that all buyers and sellers are 0-level modelers because different types of populations lead to different slopes. Also, slight changes in the agents' learning parameters ($.02 \leq \epsilon_{\min} \leq .08$ and $.05 \leq \alpha_{\min} \leq .2$) lead to slight changes in the slopes, so these would have to be taken into account if the agent is actively changing its parameters.

We also want to make clear a small caveat, which is that the price volatility that is correlated to the usefulness of keeping 1-level models is the volatility of the system *with* the agent already doing 1-level modeling. Fortunately, our experiments show that having one agent change from 0-level to 1-level does not have a great effect on the volatility as long as there are enough (i.e., more than five) other sellers so that one agent's actions cannot influence the market.

Price volatility is strongly related to our volatility v_i measure in the CLRI framework. Namely, price volatility will always be greater than or equal to v_i, and it will be proportional to v_i. Remember that v_i is the probability that the agent's target function will change. If the clearing price is not changing this means that the system has reached an equilibrium and all agents are bidding the same price. At equilibrium, the agents' target functions do not change. Each agent's target function simply states that the agent should bid the equilibrium price. However, if the price starts to change this means that the agents' target

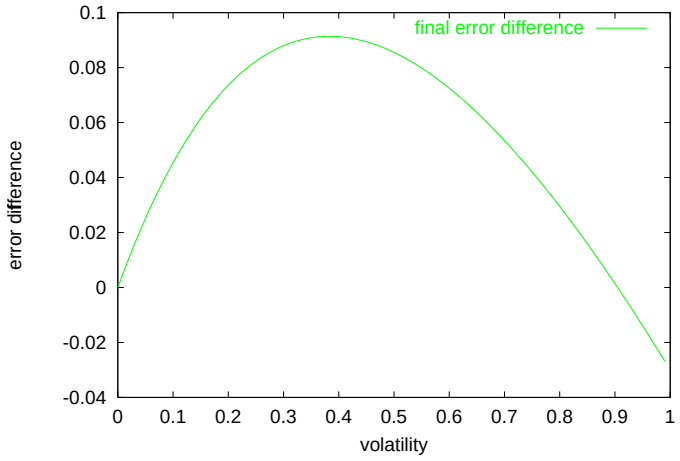


Figure 7.6: Final error as a function of volatility as predicted by the CLRI framework. We plot the difference in the final error between two agents i and j with different learning rates $l_i > l_j$ ($l_i = .9$, $l_j = .6$).

functions *might* change. The target functions will not change every time the equilibrium price changes because the agents do not change their behaviors instantly after seeing just one different price. Their learning algorithms are designed to smooth over possible noise in the input.

Now that we know that v_i will be smaller and proportional to price volatility, and given that we can map all the other experimental parameters to variables in the CLRI framework, we must ask if the results from Figure 7.5 can be reproduced using the CLRI equations. Figure 7.6 shows the results predicted by CLRI. We notice that, for small values of v_i, there is a linear correlation between v_i and the difference in the final errors of the two agents with different learning rates. This is the same correlation we observed in the experimental results shown in Figure 7.5.

We also notice that for bigger values of v_i this correlation is reversed. What happens in these cases is that the 1-level agent, with the higher learning rate, gets close to its target function each time. But, since the volatility is so high, the target function then moves very far away from the 1-level agent's decision function. It is at this moment that the error is determined. So, at every time instant the 1-level agent's error will be very big because the target function will always move far from the decision function. The 0-level agent, on the other hand, does not get close to the target function. But since the target function is

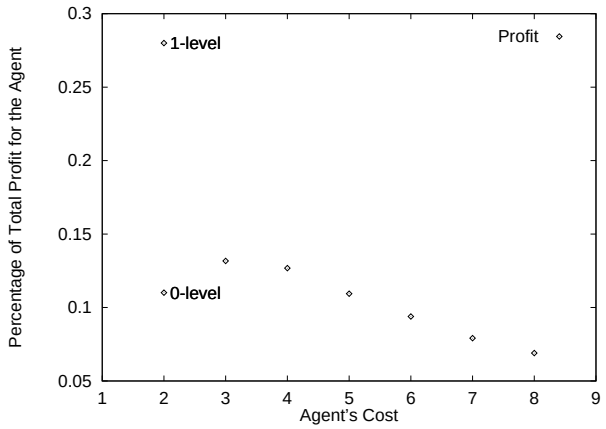


Figure 7.7: Each agent's profit percentage in population 7 from Table 7.3.

varying wildly it will, by chance, be close to the decision function some of the time.[6] This application of the CLRI framework tells us that, when there is really high volatility it is best for an agent to not change its behavior rather than aggressively trying to match its target function.

Finally, in all populations where all the buyers are 0-level agents, we have seen that it really pays for the sellers to have low costs because this allows them to lower their prices to fit almost any demand. Since the buyers have 0-level models, the sellers with low qualities and costs (please remember that in these experiments a seller's cost equals the quality the seller returns) can raise their prices when appropriate, in effect "pretending" to be the high-quality sellers, and make an even more substantial profit. This extra profit comes at the cost of a reduction in the average value that the buyers receive. In other words, the buyers get less value because they are only 0-level agents and are less able to detect the sellers' deception. In the next section we will see how this is not true for 1-level buyers.

Of course, the 1-level sellers are more successful at this deception strategy than the 0-level sellers. Figure 7.7 shows the percentage of the profit accrued by each one of the agents in population 7 (from Table 7.3), as a function of the agent's cost. We can see how the 0-level agents' profits decrease as the agents' costs increase, and how the 1-level agent's profit is much higher than the profit of the 0-level agent with the same cost. We also notice that, since the 0-level agents are not as successful as the 1-level at taking advantage of their low costs, the first 0-level seller (i.e., the one with a cost of 2) has lower profit than the 0-level seller with cost of 3.

[6]This is akin to the example of a broken clock being correct twice a day, while one that is slow is never correct.

Population	Seller 1	2	3	4	5	6	7	8
1	8	8	8	8	8	8	8	8
2	8	8	8	8	8	8	7	8
3	8	8	8	8	8	6	7	8
4	8	8	8	8	5	6	7	8
5	8	8	8	4	5	6	7	8
6	8	8	3	4	5	6	7	8
7	8	2	3	4	5	6	7	8

Table 7.4: Qualities returned by each seller in the different populations. Seller 1 is 1-level. Sellers 2–8 are 0-level. The buyers for these populations are 1-level. These populations are used in Figure 7.8.

7.5.4 1-level Buyers and 0 and 1-level Sellers

In experiments with 1-level buyers and 0 and 1-level sellers we find that the buyers have the upper hand. The buyers quickly identify those sellers that provide the highest quality goods and buy exclusively from them. The sellers do not benefit from having deeper models; in fact, Figure 7.8 shows how a 1-level seller's profit is less than that of a similar 0-level seller because the 1-level seller tries to charge higher prices than the 0-level seller. The 1-level buyers do not fall for this trick—they have learned what quality to expect, and buy more from the lower-priced 0-level seller(s). We have here a case of erroneous models—1-level sellers assume that buyers are 0-level, and since this is not true, their erroneous deductions lead them to make bad decisions. To stay a step ahead, sellers would need to be 2-level in this case.

In Figure 7.8, the first population has all sellers returning a quality of 8 while by population 7 they are returning mostly different qualities, as given by Table 7.4. We notice in Figure 7.8 that the difference in profits between the 0-level seller and a 1-level seller, both of whom return the same quality, increases with successive populations. This is explained by the fact that, in the first population, all seven 0-level sellers are returning the same quality, while by population 7 only the 0-level pictured (i.e., seller 8) is still returning quality 8. This means that his competition, in the form of other 0-level sellers returning the same quality, decreases for successive populations. Meanwhile, in all populations there is only one 1-level seller who has no competition from other 1-level sellers. To summarize, the 0-level seller's profit is always higher than the similar 1-level seller's, and the difference increases as there are fewer other competing 0-level sellers who offer the same quality.

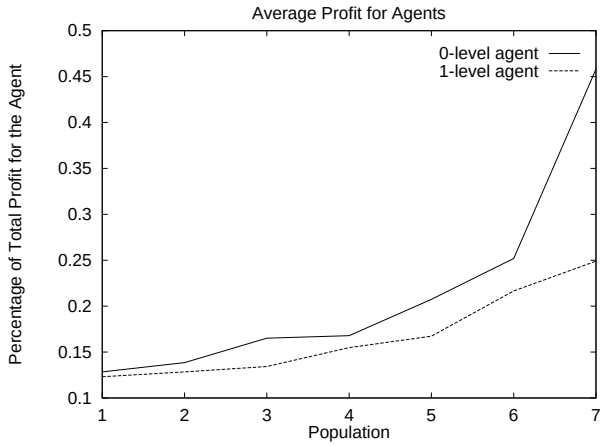


Figure 7.8: Profit of the 1-level seller 1 and the 0-level seller 8, both with the same costs and quality, in populations with 1-level buyers. The populations are given in Table 7.4.

7.5.5 0-level Buyers and Several 1-level Sellers

We have shown how 1-level sellers do better, on average, than 0-level sellers when faced with 0-level buyers, but this is not true anymore if too many 0-level sellers decide to become 1-level. Figure 7.9 shows how the profits of a 1-level seller decrease as he is joined by other 1-level sellers, given 0-level buyers. The populations used in Figure 7.9 are given by Table 7.5. The qualities offered by all sellers are kept constant for all populations, while the number of 1-level seller agents increases for successive populations.

Figure 7.9 shows that, if there are more than four 1-level sellers in the population the 0-level seller is actually making more profit than the similar 1-level seller. The 1-level seller's profit decreases because, as more sellers change from 0 to 1-level, they are competing directly with it since they are offering the same quality and are the same level. Notice that the 1-level seller's curve flattens after four 1-level sellers are present in the population. The reason is that the next sellers to change over to 1-level return qualities of 3 and 4 (corresponding to populations 6 and 7), respectively, so that they do not compete directly with the seller pictured. His profits, therefore, do not keep decreasing.

For this test, and other similar tests, we used a population of sellers that produce different qualities because, as explained in Section 7.5.3, if they had returned the same quality then an equilibrium would have been reached which would prevent the 1-level seller from making a significantly greater profit than the 0-level seller.

Population	Quality Returned							Modeling Level						
	Seller							Seller						
	1	2	3	4	5	6	7	1	2	3	4	5	6	7
1	2	2	2	2	2	3	4	0	0	0	0	0	0	0
2	2	2	2	2	2	3	4	0	1	0	0	0	0	0
3	2	2	2	2	2	3	4	0	1	1	0	0	0	0
4	2	2	2	2	2	3	4	0	1	1	1	0	0	0
5	2	2	2	2	2	3	4	0	1	1	1	1	0	0
6	2	2	2	2	2	3	4	0	1	1	1	1	1	0
7	2	2	2	2	2	3	4	0	1	1	1	1	1	1

Table 7.5: Qualities returned by each seller, and their modeling level, in the different populations. These populations are used in Figure 7.9.

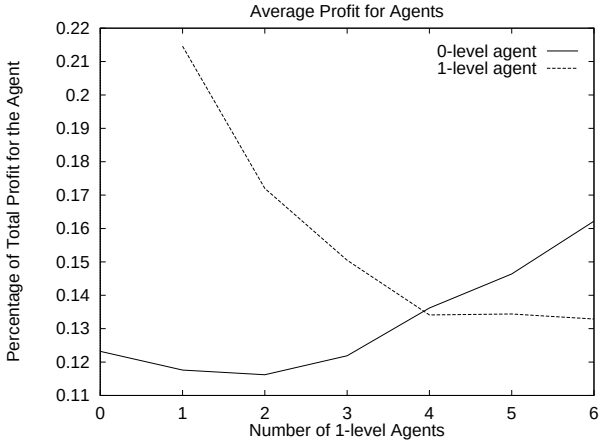


Figure 7.9: Profit of 1-level seller 2 and the similar 0-level seller 0 as a function of the number of 1-level sellers in the population. The populations used are given by Table 7.5.

7.5.6 1-level Buyers and 1 and 2-level Sellers

Assuming that the 2-level seller has perfect models of the other agents, we find that he wins an overwhelming percentage of the time. This is true, surprisingly enough, even when some of the 1-level sellers offer slightly higher quality goods. However, when the quality difference becomes too great (i.e., greater than 1), the buyers finally start to buy from the high quality 1-level sellers. This case is very similar to the cases from Section 7.5.3 with 0-level buyers and 0 and 1-level sellers. In this case, however, it is much more computationally expensive to maintain 2-level models. On the other hand, since these 2-level models are perfect, they are better predictors than the 1-level, which explains why the 2-level seller wins 95% of the time while the 1-level seller from Section 7.5.3 won, at most, 30% of the time.

Still, in both this case and the case with 0-level buyers and 0 and 1-level sellers, we noticed that the $k+1$-level sellers did better than the k-level sellers when faced with k-level buyers. With only these few data points, we venture to speculate that this might be a pattern that applies to bigger k values. That is, we expect that $k+1$-level sellers will always do better than k-level sellers when faced with k-level buyers. But, the advantages should disappear if either the buyers or the k-level sellers become $k+1$-level.

7.6 Conclusions

We have presented a method for the development of agents with incremental modeling/learning capabilities, in an economic society of agents. These agents were built, and the execution of different agent populations has lead us to the discovery of the lessons summarized in Table 7.6. The discovery of price volatility as a predictor of the benefits of deeper models confirmed the predictions made by the CLRI framework. The price volatility measure can also serve as a very useful guide for deciding how much modeling capability to build into an agent. This decision could either be done prior to development or, given enough information, it could be done during runtime.

We are also encouraged by the fact that increasing the agents' capabilities changes the system in ways that we can recognize from our everyday economic experience. Our results also showed how sellers with deeper models fare better, in general, even when they produce less valuable goods. This means that we should expect those types of agents to, eventually, be added into the UMDL.[7] Fortunately, the advantage that sellers with deeper models have is diminished with the addition of buyers that also keep deeper models. Such buyers make the UMDL system robust, in the sense that it discourages "deviant" sellers from joining.

[7] If not by us, then by a profit-conscious third party.

Buyers	Sellers	**Summary of Lessons**
0-level	0-level	Equilibrium reached only when all sellers offer the same quality. Otherwise, we get oscillations. Mean price increases when mean quality offered decreases.
0-level	Any	Sellers have big incentives to lower quality/cost.
0-level	0-level and one 1-level	1-level seller beats others. Quantitative advantage of being 1-level predicted by volatility and price distribution.
0-level	0-level and many 1-level	1-level sellers do better, as long as there are not too many of them.
1-level	0-level and one 1-level	Buyers have upper hand. They buy from the most preferred seller. 1-level sellers are usually at a disadvantage.
1-level	1-level and one 2-level	Since 2-level has perfect models, it wins an overwhelming percentage of time, except when it offers a rather lower quality.

Table 7.6: In all cases the buyers had identical value and quality assessment functions. Sellers were constrained to always return the same quality.

Sellers that try to take advantage of these buyers will be ignored by them, so they will not be able to make many sales and will, eventually, have to leave the system. That is, as long as they cannot change their identity and pretend to be a completely new agent.

The learning abilities we described have already been implemented into UDML agents and their behaviors within the UMDL are the same as reported in this chapter (this should not be surprising given that both share exactly the same machine learning code). In the real UMDL, rather than having agents broadcast their needs to all other agents, there are auction agents that sell goods, as described by the UMDL service ontology (Weinstein and Alloway, 1997). Guided by the results of this research we have built agents that use price volatility, along with their profit averages, to determine when new auctions should be started for a particular service. The creation of these auctions allows 1-level agents to go back to being 0-level agents without losing any of their profit/value. Currently, the UMDL pays for the cost of creating these new auctions but, if a protocol is developed that allows individual agents to bear part of the costs, our agents have all the information they need in order to decide whether or not they want to create a new auction and how much they are willing to pay for its creation.

In the long run, another offshoot of this research could be a better characterization of the types of system environments and how they allow/inhibit "cheating" behavior in differ-

ent agent populations. That is, we saw how, in our economic model, agents are sometimes rewarded for behaviors that do not seem to be good for the community as a whole (e.g., when some of the sellers raised their prices while lowering the qualities they offered). These rewards, we found, start to diminish as the other agents become "smarter." We can intuit that the agents in these systems will eventually settle on some level of nesting that balances their costs of keeping nested models with their gains from taking better actions (Kauffman, 1994). It would be very useful to characterize the types of equilibria that different combinations of environments and agent learning abilities might lead to.

CHAPTER 8

Conclusions

In this thesis, we set out to explore the problem of how to build agents that use and learn models of other agents. We approached this problem by first introducing a formal framework which allowed us to describe the behavior of an agent, and to characterize the degree to which its behavior is correct. We also provided some notation for describing the nested models an agent might have of other agents.

Working under the initial assumption that agents can generate correct models of other agents (with some computational cost), we used our framework to present an algorithm (LR-RMM) that allows an agent to determine when it should stop generating nested models of other agents and, instead, take an action. We applied this algorithm to the Pursuit Task and showed how the LR-RMM agent performed its task just as well as other agents that generated a lot deeper models of the other agents.

Next, we assumed that the agents had to learn their models of others via observation, instead of simply generating them from previous knowledge. This new situation brought about what we referred to as the moving target function problem. That is, if the other agents are also learning and changing their behaviors, will my agent ever achieve the correct behavior? We addressed this problem by developing the CLRI framework. This framework allows an agent designer to describe different agents' learning abilities and their impacts on each other, and to then apply our equations (namely Equations (5.6) and (5.16)) in order to determine the expected errors of the agents in a system composed of learning agents. We demonstrated the use of our framework by showing how a designer can determine if his agent is expected to ever behave correctly and, if it never behaves correctly, how bad its expected behavior will be.

We performed a computational complexity analysis on the problem of learning models, and built tables that agent designers can use to determine the sample complexity of a nested learning agent's problem. These values can also be used to get a lower bound on an agent's learning rate (l_i), which can then be used in the CLRI framework, and to compare two

similar agents in order to determine their relative learning abilities.

Finally, we used agent-based simulations to test real learning agents in a market-based system. This allowed us, first, to confirm some of the predictions made by the CLRI framework and show its application to a real-world problem. Second, it allowed us to study other behaviors specific to market-based MASs. In Table 7.6 we summarized our findings from experiments with k-level learning agents in a market system.

The contributions of this thesis can be summarized by the following bullet points:

- We provide a basic framework for describing learning agents and MASs composed of these agents, as well as some notation for categorizing an agent's nested knowledge about other agents (Chapter 3). Our notation is explicit about the agent's nested models of other agents. We show how our notation can be used to describe the types of knowledge that researchers, such as (Claus and Boutilier, 1997), (Hu and Wellman, 1998), and ourselves are using. The framework and notation also introduce formal definitions for an agent's *error* and for knowledge *dampening*.

- We give an algorithm (LR-RMM) that exploits the dampening in an agent's knowledge and allows the agent to dynamically determine which of its nested models to use (Chapter 4). The algorithm allows an agent to take the best possible action without spending unnecessary time "thinking." We have demonstrated the use and the efficacy of this algorithm in an application of the Pursuit Task.

- We have formulated a predictive theory and framework (CLRI), built as an extension of our basic framework. The CLRI framework can be used to determine the expected behavior of a learning agent in a MAS composed of other learning agents, given the value of some parameters which describe all the agents' learning abilities (Chapter 5). The CLRI framework has been confirmed with our experimental results and with experimental results observed by other researchers. It is also a first step towards a more general predictive theory for complex adaptive systems composed of learning agents.

- We give a method for calculating the sample complexity (i.e., the size of the learning problem) for learning the different nested decision functions of a recursive modeling agent (Chapter 6). This method can be used when the agent uses either some form of reinforcement learning or some form of supervised learning. The resulting sample complexity can be used to provide a lower bound on the CLRI learning rate l_i for that specific learning problem. We have demonstrated how to use this method to determine which of two different agent designs is better. These predicted results were verified with experiments.

- Finally, we have presented an analysis on the effects of nested learning agents within an economic MAS (Chapter 7). The system adheres to our initial assumptions (from Chapters 3 and 5) that agents can observe each other's actions and can update all their models in between time steps. In this system, we showed the relative advantages of keeping nested models, how having more (correct) knowledge gives an agent an advantage in highly volatile markets, the fact that agents can expect decreasing returns for increasing levels of modeling (i.e., "strategic thinking") because of the characteristics of the system, and other results. Our experiments identified emergent behaviors that can be predicted by our theory, and those behaviors that are too domain-specific and can only be found through implementation and agent-based simulation.

In summary, this thesis provides the designer of learning agents for MASs with LR-RMM, an algorithm for dynamically determining when his agent should stop thinking about others and take action, and with the CLRI framework, which the designer of learning agents can use to predict the MAS's expected behavior (and his agent's expected behavior) before implementation. Furthermore, we confirmed the CLRI framework with experiments in an economic domain and studied, in depth, the behavior of nested learning agents in a market-based MAS. These experiments taught us that these systems provide decreasing returns for increasing amounts of strategic thinking, but are only robust when some amount of learning is done by the agents. The results lead us to the conclusion that, in open market-based MASs, such as the one we studied, the system will behave robustly only if there are learning agents present, although it is not necessary for all agents to be learning all the time.[1]

8.1 Current Limitations and Simplifying Assumptions

There are several assumptions made through this thesis that will likely limit the applicability of our results to certain domains. These assumptions are also obvious pointers to areas of further research.

In Chapter 3.1.1 we made four simplifying assumptions. The first assumption was that all agents perceive the same world state and are able to observe every other agent's actions. This assumption allowed us to build agents that learn about other agents. However, in many real-world domains, there are situations were an agent cannot determine exactly which world state it is in, or is unaware of the actions of other agents, or has some uncertainty as to the specific actions taken by other agents. A typical example involves robot agents, where one agent's "view" of another might be partially blocked because of physical obstacles, or

[1] On the other hand, learning might not be needed in closed systems, such as ones where the agents must return services of a certain quality or they are punished in some way (e.g., by elimination from the system or by receiving a penalty), since the rules of encounter in these systems are strict enough to disallow cheating.

partially obscured due to noise in the agent's sensors. In domains such as this one, it will be harder for an agent to learn models of others. This could be reflected as a lower learning rate and retention rate in the CLRI framework, but such a solution does not capture all the nuances of the situation. For example, physical obstacles have a definite location so their effect on the learning rate will not be the same for all worlds w. Another possible way to address this limitation is by assuming that a function (perhaps, a probabilistic function) exists that maps from one agent's perception of the world state, into another agent's perception of the world state. Such a function could probably be added to our framework without it losing much of its predictive power. However, deeper study of the relevant issues is needed.

The second assumption was that the next world state the agents perceive is taken from some fixed probability distribution. In a more general framework we would expect the next world state to depend on the past and current world state, and on the actions of other agents. This type of general framework is also usually seen in physical or robotic domains. In these domains it is unlikely that the new world state can be taken from a fixed probability distribution. However, there are other episodic domains where this assumption does hold. The market-based system presented in Chapter 7 is an example of one such domain. Other examples are given by the matching game of Chapter 5.2.1, and the coordination game described in Chapter 5.7.2. In all of these, the agents are presented with a new problem (i.e., world state) at each time step. The agents then take an action in response to this problem and a new problem is presented to them. In these type of episodic systems there is little or no correlation between the world states seen at times t and $t+1$. These systems obey our simplifying assumption.

A possible area of further research is the expansion of the framework so that it can handle distributions that change over time. That is, rather than having a fixed distribution over world states $\mathcal{D}(w)$, we could make the distribution vary over time like the other parameters (i.e., $\mathcal{D}^t(w)$). We would also need to characterize how this distribution will change, given the current and past world states and the set of actions taken by all agents. Luckily, in many systems this transition is well defined. For example, in the Pursuit Task it is easy to determine what the new world state will be given the current world state (the state of the board) and the actions of all the agents. We would, however, need to change the predictive equations in the CLRI framework in order to handle these time-dependent distributions functions. It is not clear to us, at this time, whether this is feasible or not.

The third simplifying assumption was that, as done in computational learning theory, the agents' actions are either correct or incorrect. We excluded the possibility of a middle-ground of semi-correct actions. It is, however, true that some real-world systems implement

agents that have different utilities associated with each possible action, in each world. We could easily extend our definition of *error* and have it take into account the utilities associated with each action. However, if we made this extension, we would not be able to use the equations from our CLRI framework. A new version of the CLRI equations would have be designed that uses an utility-based error definition. We have only started to research the possibility of creating this new predictive framework.

The fourth, and final, simplifying assumption we made was that, at any moment, an agent's actions are dictated solely by the world state the agent is in. As we mentioned, this does not mean that the agent cannot have an internal state. Since the agent's decision function can vary over time, it can be considered to contain the internal state of the agent. We consider this assumption to be the least limiting of the four assumptions. The only instances, we can think of, where it will be limiting are systems in which the agents often use a random number generator, consult some outside oracle, or use other information that is outside the world state.

We have also been assuming that the agents learn at discrete intervals. This does not mean that time itself is discrete. It only means that the agents must discretize time and take actions and receive their payoffs at certain discrete time intervals. Of course, when building agents that discretize time, one must make sure that the intervals are small enough so that the agents can take actions when they need to. The size of these intervals will depend on the speed at which events in the system take place.

Another assumption we made was the fact that there is enough time between time steps t and $t+1$ for the agents to update all their models. The validity of this assumption rests on the quantitative relation between the agents' computational speed and the size of the time steps. If a given domain has slow learners, we could increase the delay between time steps, for these agents, in order to allow them enough time to learn. This means that the agents would not be able to use the information from the intervening time steps. Therefore, this solution simply reduces this problem to the previous problem of agents not being able to observe other agents' actions.

In the experiments in Chapter 7 we assumed that all the agents had $\delta_i(w, \vec{a}_{-i})$ knowledge. That is, they all knew what action to take given the set of all actions taken by other. This assumption was justified in our economic domain. However, there are other systems, including other market-based systems, where this assumption cannot be made. In these cases the agents will either have to learn this knowledge, or will be limited to being 0-level agents.

8.2 Future Work

In the future, we hope to expand on the CLRI framework and show how it can capture other learning mechanisms. We might be able to generate theorems that can map implementation parameters from some machine learning algorithm into CLRI parameters. For example, we can envision a set of equations that map the parameters used in reinforcement learning to CLRI parameters, allowing for some error margins. Another simple way to expand the current CLRI framework is to consider error distributions instead of just expected errors. There is nothing in the current CLRI equations that prevents us from using distributions instead of expected errors. However, we have not done the necessary experimental and theoretical verifications. If, as we suspect, error distributions can be used instead of expected errors, then this will give us a more precise understanding of the expected system dynamics. For example, we could determine the probability that the agent will end up in any specific error range.

Another interesting area of improvement is in the timing for the learning framework. In the CLRI framework, an agent takes an action using δ_i^t and gets a reward based on Δ_i^t. The agent then updates to δ_i^{t+1} which is, the agent hopes, a better match for Δ_i^t. The agents try to learn Δ_i^t because that is the function on which their feedback/rewards are based. The agents do not get any feedback from Δ_i^{t+1} at time t. However, in some systems a designer might have knowledge that he can use to get some feedback, at time t, on what Δ_i^{t+1} will be. He might even use this knowledge to implement an agent that learns Δ_i^{t+1} instead of Δ_i^t. These systems present an interesting situation. In general, a designer with this knowledge would know how all the agents will change their behaviors after all agents have taken action and received their feedbacks, and how the other agents' behavioral changes will affect his agent's target function. Knowing how other agents update their behaviors might be easy. For example, a designer might know that the other agents use the same learning algorithm his agent uses.[2] Knowledge about how changes in the other agents' behaviors affect an agent's target function might be hard to determine, and this knowledge is likely to need a large knowledge base. Still, we think a deeper analysis of the feasibility and impact of these "predictive" agents will be interesting.

The implementation of learning agents that model other agents also lays the foundation for future studies of distributed recommender systems (Resnick and Varian, 1997) where agents communicate with each other in order to determine who to interact (e.g., buy/sell) with. That is, while current recommender systems are centralized (i.e., all information is stored in one machine), we envision a MAS were each of the software agents has preferences

[2]Notice that this differs from k-level agents which model other agents based on the other agents' *past* behaviors.

about who to interact with, and communicates these preferences to others, becoming a recommender of other agents. We could, of course, also have recommenders of recommender agents, and so on. The agents' interactions in the MAS would form a web of preferences, which would reflect the biases of the individual agents. If properly designed, this system would allow any one agent to quickly find the specific goods/services it desires while, at the same time, heavily discouraging "deviant" agents that do not contribute to the social welfare. In order to carry out this research, we would first have to formally describe the type of social welfare we are interested in. Intuitively, we want systems where agents spend most of their time doing their domain tasks (producing goods/services), rather than thinking strategically about the market. However, this intuition needs to be formalized.

While the CLRI framework does work for a wide number of MASs and learning techniques, we realize that developing a general enough framework to capture all MASs might well be impossible. We believe, however, that the CLRI framework covers a significant number of MASs. It also points the way to how a general framework might appear, if one did exist. The parameters we chose, i.e., the change rate, retention rate, learning rate and impact, are general enough so that they apply to a wide variety of systems, but also have very precise definitions within our framework. These precise definitions allow us to make quantitative predictions. We are not claiming that these parameters are all that we will ever need, or even the correct ones to use in a general theory. We do, however, show that our approach is viable for a limited, but varied, set of implementations, and we claim that more research into similar frameworks and/or extensions that cover a wider range of learning algorithms will lead to a better understanding of MASs composed of learning agents and, perhaps, complex adaptive systems.

APPENDICES

APPENDIX A

Derivation for Matching Game

If we can assume that the action chosen when an agent changes $\delta_i^t(w)$ and the result does not match $\Delta_i^t(w)$ (for some specific w) is taken from a flat probability distribution, then we can say that:

$$B = \frac{|A_i| - 2}{|A_i| - 1} \tag{A.1}$$

We will now show how to calculate C in in (5.11). We start by remembering that

$$C = l_i + (1 - c_i)D + (c_i - l_i)F \tag{5.8}$$

For the matching game we can set these values to be

$$D = 1 - l_j \tag{A.2}$$

$$F = l_j + (1 - l_j)\left(\frac{|A_i| - 3}{|A_i| - 2}\right) \tag{A.3}$$

First, since having $|A_i| = 2$ implies that $c_i = l_i$, this means that for this case we have

$$C = l_i + (1 - c_i)(1 - l_j) \tag{A.4}$$

For the case where $|A_i| > 2$, which is the case we are interested in, we can plug in the values for D and F and simplify C to be:

$$C = 1 - l_j + \frac{c_i l_j(|A_i| - 1) + l_i(1 - l_j) - c_i}{|A_i| - 2} \tag{A.5}$$

BIBLIOGRAPHY

Atkins, D. E., Birmingham, W. P., Durfee, E. H., Glover, E. J., Mullen, T., Rundensteiner, E. A., Soloway, E., Vidal, J. M., Wallace, R., and Wellman, M. P. (1996). Toward inquiry-based education through interacting software agents. *IEEE Computer*.

Axelrod, R. (1997). *The Complexity of Cooperation*, chapter Evolving New Strategies, pages 10–29. Princeton University Press.

Axelrod, R. M. (1984). *The Evolution of Cooperation*. Basic Books.

Bayardo Jr., R. J., Bohrer, W., Brice, R., Cichocki, A., Fowler, J., Helal, A., Kashyap, V., Ksiezyk, T., Martin, G., Nodine, M., Rashid, M., Rusinkiewicz, M., Shea, R., Unnikrishnan, C., Unruh, A., and Woelk, D. (1997). InfoSleuth: Agent-based semantic integration of information in open and dynamic environments. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2).

Binmore, K. (1987). Modeling rational players, part 1. *Economics and Philosophy*, 3:179–214.

Binmore, K. (1988). Modeling rational players, part 2. *Economics and Philosophy*, 4:9–55.

Boddy, M. and Dean, T. L. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285.

Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355.

Carver, N. F., Lesser, V. R., and McCue, D. L. (1984). Focusing in plan recognition. In *Proceedings of the National Conference on Artificial Intelligence*, pages 42–48. AAAI Press.

Charniak, E. and Goldman, R. P. (1993). A bayesian model of plan recognition. *Artificial Intelligence*, 64:53–79.

Chavez, A. and Maes, P. (1996). Kasbah: An agent marketplace for buying and selling goods. In *First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 75–90.

Claus, C. and Boutilier, C. (1997). The dynamics of reinforcement learning in cooperative multiagent systems. In *Proceedings of Workshop on Multiagent Learning*. AAAI Press. `http://www.cs.ubc.ca/spider/cebly/Papers/multirl.ps`.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

Cowan, G. A., Pines, D., and Meltzer, D., editors (1995). *Complexity: Metaphors, Models and Reality*. Addison Wesley.

Dean, T. and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings AAAI-88*.

Dennett, D. C. (1987). *The Intentional Stance*. Bradford Books/MIT Press.

Durfee, E. H. (1995). Blissful ignorance: Knowing just enough to coordinate well. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 406–413.

Durfee, E. H., Gmytrasiewicz, P. J., and Rosenschein, J. S. (1994). The utility of embedded communications and the emergence of protocols. In *Proceedings of the 13th International Distributed Artificial Intelligence Workshop*.

Durfee, E. H., Kiskis, D. L., and Birmingham, W. P. (1997). The agent architecture of the University of Michigan Digital Library. *IEE Proceedings on Software Engineering*, 144(1):61–71.

Durfee, E. H., Lee, J., and Gmytrasiewicz, P. J. (1993). Overeager reciprocal rationality and mixed strategy equilibria. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*.

Etzioni, O. (1996). Moving up the information food chain: Deploying softbots on the world wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1322–1326, Menlo Park. AAAI Press / MIT Press.

Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. (1995). *Reasoning About Knowledge*. MIT Press.

Gasser, L. and Huhns, M. N., editors (1989). *Distributed Artificial Intelligence*, volume 2 of *Research Notes in Artificial Intelligence*. Pitman.

Gasser, L., Rouquetter, N. F., Hill, R. W., and Lieb, J. (1989). Representing and using organizational knowledge in distributed AI systems. In Gasser, L. and Huhns, M. N., editors, *Distributed Artificial Intelligence*, volume 2, pages 55–78. Morgan Kauffman Publishers.

Glance, N. S. (1993). *Dynamics with Expectations*. PhD thesis, Stanford University.

Gmytrasiewicz, P. J. (1992). *A Decision-Theoretic Model of Coordination and Communication in Autonomous Systems (Reasoning Systems)*. PhD thesis, University of Michigan.

Gmytrasiewicz, P. J. (1996). On reasoning about other agents. In Wooldridge, M., Müller, J. P., and Tambe, M., editors, *Intelligent Agents Volume II*, Lecture Notes in Artificial Intelligence, pages 143–155. Springer-Verlag.

Gmytrasiewicz, P. J. and Durfee, E. H. (1995). A rigorous, operational formalization of recursive modeling. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 125–132.

Gmytrasiewicz, P. J., Durfee, E. H., and Wehe, D. K. (1991). A decision-theoretic approach to coordinating multiagent interactions. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*.

Goodman, B. A. and Litman, D. J. (1990). Plan recognition for intelligent interfaces. In *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, volume 1. IEEE Computer Society Press.

Grefenstette, J. J. (1992). The evolution of strategies in multiagent environments. *Adaptive Behavior*, 1(1):67–89.

Hogg, T. and Huberman, B. A. (1991). Controlling chaos in distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1325–1332. (Special Issue on Distributed AI).

Horvitz, E. (1988). Reasoning under varying and uncertain resource constraints. In *Proceedings AAAI-88*.

Hu, J. and Wellman, M. P. (1996). Self-fulfilling bias in multiagent learning. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 118–125.

Hu, J. and Wellman, M. P. (1998). Online learning about other agents in a dynamic multi-agent system. In *Proceedings of the Second International Conference on Autonomous Agents.*

Huber, M. J., Durfee, E. H., and Wellman, M. P. (1994). The automated mapping of plans for plan recognition. In de Mantaras, R. L. and Poole, D., editors, *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, pages 344–351, San Francisco, CA, USA. Morgan Kaufmann Publishers.

Huberman, B. A. and Clearwater, S. H. (1995). A multiagent system for controlling building environments. In Lesser, V., editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 171–176, San Francisco, CA. MIT Press.

Huhns, M. N. and Singh, M. P., editors (1997). *Readings in Agents.* Morgan Kaufmann.

Ishida, T. (1997). *Real-Time Search for Learnig Autonomous Agents.* Kluwer Academic Publishers.

Jennings, N. R. (1994). Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, 8(3):223–250.

Jennings, N. R., Corera, J. M., and Laresgoiti, I. (1995). Developing industrial multi-agent systems. In *Proceeding of the First International Conference on Multi-Agent Systems*, pages 323–340. MIT Press.

Kauffman, S. A. (1994). Whispers from Carnot: The origins of order and principles of adaptation in complex nonequilibrium systems. In Cowan, G. A., Pines, D., and Meltzer, D., editors, *Complexity: Models, Metaphors and Reality*, pages 83–136. Addison Wesley.

Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory.* MIT Press.

Konolige, K. and Pollack, M. E. (1989). Ascribing plans to agents: Preliminary report. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 2, pages 924–930. AAAI Press.

Korf, R. E. (1992). A simple solution to pursuit games. In *Proceedings of the 11th International Distributed Artificial Intelligence Workshop.*

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, pages 1–64.

Lesser, V. R. and Corkill, D. D. (1981). Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(1):81–96.

Lesser, V. R., Pavlin, J., and Durfee, E. H. (1988). Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–62.

Levy, R. and Rosenschein, J. S. (1992). A game theoretic approach to the pursuit problem. In *Proceedings of the 11th International Distributed Artificial Intelligence Workshop.*

Lin, L.-J. (1992). Self-improving reactive agent based on reinforcement learning. *Machine Learning*, 8:293–321.

Liu, J.-S. and Sycara, K. P. (1995). Multiagent coordination in tightly coupled real-time environments. In Lesser, V., editor, *Proceedings of the First International Conference on Multi–Agent Systems.* MIT Press.

Luce, R. D. and Raiffa, H. (1957). *Games and Decisions.* Dover Publications, Inc., New York.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

Montgomery, T. A. and Durfee, E. H. (1990). Using MICE to study intelligent dynamic coordination. In *Proceedings of IEEE Conference on Tools for AI.*

Nadella, R. and Sen, S. (1997). Correlating internal parameters and external performance. In Weiß, G., editor, *Distributed Artificial Intelligence Meets Machine Learning*, pages 137–150. Springer.

Parker, L. E. (1993). Learning in cooperative robot teams. In *Proceeding of the International Joint Conference on Artificial Intelligence.*

Parkes, D. C. and Ungar, L. H. (1997). Learning and adaptation in multiagent systems. In Sen, S., editor, *Multiagent Learning: AAAI Workshop papers*, pages 47–52. Technical Report WS-97-03.

Pollack, M. E. (1990). Plans as complex mental attitudes. In Cohen, P. R., Morgan, J., and Pollack, M. E., editors, *Intentions in Communication*, chapter 5, pages 77–103. MIT Press.

Resnick, P. and Varian, H. R. (1997). Recommender systems. *Communications of the ACM*, 40(3):56–58.

Rosenschein, J. S. and Zlotkin, G. (1994). *Rules of Encounter.* The MIT Press, Cambridge, Massachusetts.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Russell, S. and Wefald, E. (1991). *Do The Right Thing.* The MIT Press, Cambridge, Massachusetts.

Sen, S., editor (1996). *Working Notes from the AAAI Symposium on Adaptation, Co-evolution and Learning in Multiagent Systems.*

Sen, S., editor (1997). *Collected Papers from the AAAI Workshop on Multiagent Learning.* AAAI Press.

Sen, S., Sekaran, M., and Hale, J. (1994). Learning to coordinate without sharing information. In *Proceedings of the Twelfth National Conference on Artificial Intelligence.*

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60:51–92.

Shoham, Y. and Tennenholtz, M. (1992). Emergent conventions in multi-agent systems: Initial experimental results and observations. In Nebel, Bernhard; Rich, Charles; Swartout, W., editor, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 225–231, Cambridge, MA. Morgan Kaufmann.

Shoham, Y. and Tennenholtz, M. (1997). On the emergence of social conventions: modeling, analysis, and simulations. *Artificial Intelligence*, 94(1):139–166.

Shubik, M. (1985). *Game theory in the social sciences.* The MIT Press, Cambridge, Massachusetts.

Sian, S. S. (1991). Adaptation based on cooperative learning in multi-agent systems. In Demazeau, Y. and Muller, J.-H., editors, *Decentralized AI*, volume 2, pages 257–272. Elsevier Science.

Simon, H. A. (1982). *Models of Bounded Rationality*, volume 2. MIT Press.

Simon, H. A. (1996). *The Sciences of the Artificial.* MIT Press, third edition edition.

Stephens, L. M. and Merx, M. B. (1990). The effect of agent control strategy on the performance of a DAI pursuit problem. In *Proceedings of the 9th International Distributed Artificial Intelligence Workshop.*

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Sycara, K. P., Roth, S. F., Sadeh, N., and Fox, M. (1991). Resource allocation in distributed factory scheduling. *IEEE Expert*, 6(1):29–40.

Tambe, M. (1995). Agent and agent-group tracking in a real-time dynamic environment. In Lesser, V., editor, *Proceedings of the First International Conference on Multi–Agent Systems*, pages 368–375, San Francisco, CA. MIT Press.

Tambe, M. and Rosenbloom, P. S. (1996). Architectures for agents that track other agents in multi-agent worlds. In Wooldridge, M., Müller, J. P., and Tambe, M., editors, *Intelligent Agents Volume II*, Lecture Notes in Artificial Intelligence, pages 156–170. Springer-Verlag.

Terabe, M., Wasio, T., Katai, O., and Sawaragi, T. (1997). A study of organizational learning in multi-agent systems. In Weiß, G., editor, *Distributed Artificial Intelligence Meets Machine Learning*, pages 168–179. Springer.

Torrance, M. C. and Viola, P. A. (1991). The AGENT0 manual. Technical Report CS-TR-91-1389, Stanford University, Department of Computer Science.

Vidal, J. M. and Durfee, E. H. (1994). Agent modeling methods using limited rationality. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, page 1495.

Vidal, J. M. and Durfee, E. H. (1995). Recursive agent modeling using limited rationality. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 125–132.

Vidal, J. M. and Durfee, E. H. (1996a). The impact of nested agent models in an information economy. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 377–384.

Vidal, J. M. and Durfee, E. H. (1996b). Using recursive agent models effectively. In Wooldridge, M., Müller, J. P., and Tambe, M., editors, *Intelligent Agents Volume II*, Lecture Notes in Artificial Intelligence, pages 171–196. Springer-Verlag.

Vidal, J. M. and Durfee, E. H. (1998a). Learning nested models in an information economy. *Journal of Experimental and Theoretical Artificial Intelligence: Special Issue Learning in DAI Systems*, 10(3).

Vidal, J. M. and Durfee, E. H. (1998b). The moving target function problem in multi-agent learning. In *Proceedings of the Third International Conference on Multi-Agent Systems.*

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Weinstein, P. and Alloway, G. (1997). Seed ontologies: growing digital libraries as distributred, intelligent systems. In *Proceedings of the Second ACM International Conference on Digital Libraries.*

Weiß, G. (1993). Learning to coordinate actions in multi-agent systems. In *Proceeding of the International Joint Conference on Artificial Intelligence.*

Weiß, G., editor (1997). *Distributed Artificial Intelligence meets Machine Learning.* Springer.

Werner, E. (1989). Cooperative agents:a unified theory of communication and social structure. In Gasser, L. and Huhns, M. N., editors, *Distributed Artificial Intelligence*, volume 2, pages 3–36. Morgan Kaufman.

Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2).

Wurman, P. R., Wellman, M. P., and Walsh, W. E. (1998). The michigan internet auctionbot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents.*