A Systematic Approach for Including Machine Learning in Multi-Agent Systems

José A. R. P. Sardinha, Alessandro Garcia, Carlos J. P. Lucena, Ruy L. Milidiú

TecComm Group (LES), Computer Science Department, PUC-Rio
Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, Brazil
{sardinha,afgarcia,lucena,milidiu}@inf.puc-rio.br

Abstract. Large scale multi-agent systems (MASs) in unpredictable environments must use machine learning techniques to perform their goals and improve the performance of the system. This paper presents a systematic approach to introduce machine learning in the design and implementation phases of a software agent. We also present an incremental implementation process for building asynchronous and distributed agents, which enable the combination of machine learning strategies. This process supports the stepwise building of adaptable MASs for unknown situations, improving their capacity to scale up. We use the Trading Agent Competition (TAC) environment as a case study to illustrate the suitability of our approach.

1 Introduction

Multi-Agent Systems (MASs) [1] [2] is a new technology that has been recently used in many simulators and intelligent systems to help humans perform several time-consuming tasks. Applications for e-commerce, information retrieval, and business intelligence use the MASs technology to build distributed systems over the Internet. In this context, machine learning algorithms are crucial to provide well-known strategies to support the construction of adaptable agents, especially in unpredictable, heterogeneous environments, such as the Internet.

However, the incorporation of learning techniques into large scale multi-agent systems is not a trivial task. Software engineers who design and implement realistic MASs are faced with recurring learning concerns, such as: (i) How to evaluate if the goal of a multi-agent system has been achieved? (ii) How to define the individual goal of each agent in the system and evaluate it? (iii) How the knowledge of each individual agent is going to be modeled? (iv) How this agent will acquire the knowledge? (v) How to combine the multiple used learning techniques and to distribute them to the different agents in a MAS, (vi) How to associate the learning issues with typical abstractions in agent-based software engineering? and (vii) How to specify the learning issues at an early design stage and support a smooth transition of those issues to the implementation stages?

Unfortunately, software engineers have largely relied on their experience and intuition to address the questions above during the development of realistic adaptable MASs. Research on agent-based software engineering has focused on the development of new methodologies and implementation frameworks. However, these approaches do not provide guidelines to support the incorporation of learning issues into the system in the early stage of design. Implementation frameworks [3][4] provide object-oriented APIs for MAS development, but they do not assist the handling and structuring of the learning design in a systematic way. In addition, most proposed methodologies [5] [6] are too high level and do not indicate how to master the complexity of these learning concerns through the design and implementation steps.

This observation provides the rationale for investigating how to integrate learning issues neatly within agent-oriented software engineering. This paper presents a systematic approach to support a disciplined introduction of machine learning techniques in MASs from an early stage of design. The proposed approach encompasses guidelines to both the design and implementation phases of an agent-based system. It is based on an incremental development strategy that largely relies on simulation and testing techniques.

This systematic approach emerged from our extensive experience on the development of distinct and heterogeneous MAS applications, including: (a) a multi-agent system [7][8] that uses evolutionary techniques to build offerings in a retail market, (b) an agent-based system [9] that learns to play Tic-Tac-Toe with no prior knowledge, (c) a multi-agent system [10] for the Trading Agent Competition (TAC) [15], and (d) a multi-agent system [11] for managing the paper submission and selection process in workshops and conferences.

The remainder of this paper is organized as follows. Section 2 presents a case study that involves a multi-agent system for the TAC Competition. Section 3 presents our approach to support the introduction of machine learning techniques in large scale multi-agent systems. Section 4 presents our conclusions and directions for future work.

2 The Trading Agent Competition MAS – A Case Study

The Trading Agent Competition [12] (TAC) is designed to promote and encourage high quality research into the trading agent problem. Figure 1 presents the architecture of *LearnAgents* [10], a multi agent system with modular entities that are asynchronous, distributed, reusable, and easy to interoperate. We define agent types that tackle sub problems of trading, such as price prediction, bid planning, goods allocation, bidding, among others. The system's goal is to acquire travel packages for clients with as much profit as possible. This profit is defined as the sum of the utilities of the eight clients in the TAC game, minus the costs of acquiring the travel goods in the auctions.

The Hotel Sensor Agent, Flight Sensor Agent and Ticket Sensor Agent are responsible for the market sensory and knowledge building of the system. These agents collect price information from auctions and store them in the knowledge base. The sensor agents also receive all the events from the environment, and are responsible for notifying other agents in the system with these events. The Flight and Hotel Price Predictor Agent is also responsible for knowledge building. This agent predicts hotel and flight auction prices for the knowledge base. The price predictor agent uses the price history in the knowledge to forecast auction quotes.

The agent types responsible for bid planning are the Allocator Master Agent and the Allocator Slave Agents. The slave agents calculate different scenarios based on the prices in the knowledge base. These scenarios define travel good types and quantities, and characterize a list of different strategies for the system. The Allocator Master Agent is responsible for combining all the scenarios generated from the other slave agents. These scenarios are then stored in the knowledge base.

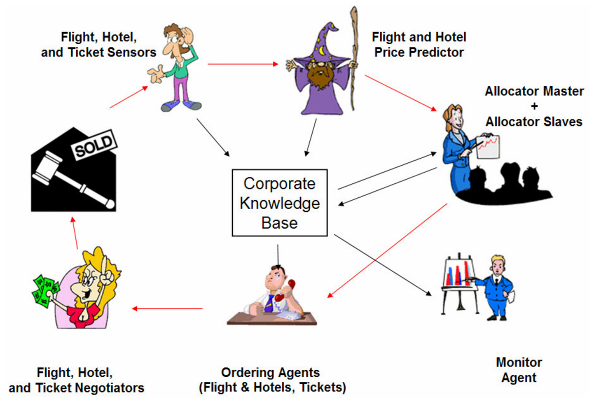

Flight, Hotel, and Ticket Sensors
Flight and Hotel Price Predictor
Allocator Master + Allocator Slaves
SOLD
Corporate Knowledge Base
Monitor Agent
Flight, Hotel, and Ticket Negotiators
Ordering Agents (Flight & Hotels, Tickets)

Fig. 1. The *LearnAgents* Architecture

The Ordering Agent is responsible for good allocation, and it decides the quantity of the required travel goods based on the scenarios in the knowledge base. This decision is based on a scenario with a high profit and a low risk. The Hotel Negotiator Agent, Flight Negotiator Agent, and Ticket Negotiator Agent are responsible for negotiating travel goods in the auctions based on the decision from the Ordering Agent. The negotiators have the goal to buy all the requested goods with the minimum expenditure. The Monitor Agent is responsible for saving data from the environment and evaluating the performance of the multi-agent system.

There was a need to introduce machine learning techniques in the Flight and Hotel Price Predictor Agent, the Allocator Master and Slave Agents, and the Hotel Negotiator Agent. For brevity reasons, we will only illustrate in section 3 the design decisions of the Flight and Hotel Price Predictor Agent. However, we present results of the performance gain obtained with the process of including machine learning algorithms in the *LearnAgents* system based on our stepwise approach.

3 Introducing Learning Techniques in Multi-Agent Systems

Mitchell [13] defines machine learning as follows: “A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*”. Consequently, machine learning techniques are normally used when performance gain is required in a system. The main goal of the proposed approach is to support a disciplined introduction of machine learning techniques into a multi-agent system. The process also permits in the implementation phase a disciplined integration of disparate machine learning algorithms in agent-based systems and their performance assessment. This systematic approach has four phases:

(i) *Systemic Goal & Performance Measure Selection*, where a systemic goal is defined and a measure of performance is selected;

(ii) *Agent Selection & Agent Learning Goal Definition*, where agents with complex plans are selected, and goals are defined for the learning algorithm;

(iii) *Agent Machine Learning Design*, where code design is defined; and

(iv) *Incremental Implementation & Performance Measurement*, where an incremental implementation is proposed with training, testing and evaluation.

Figure 2 depicts all the phases of the process. The following subsections describe in detail each of the process phases and present the application of the approach to our case study.

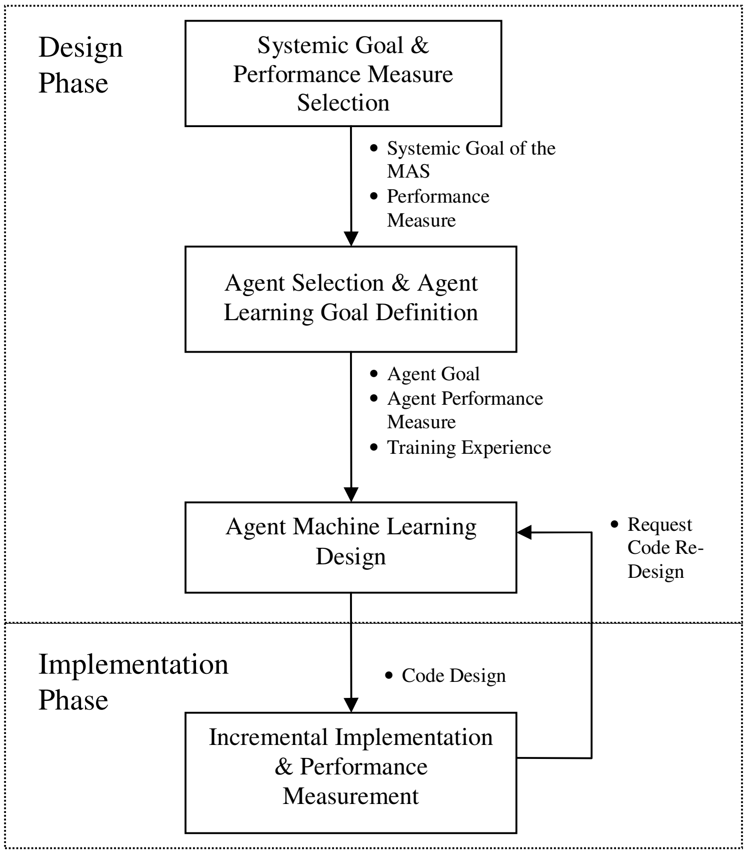

Design Phase
Systemic Goal & Performance Measure Selection
• Systemic Goal of the MAS
• Performance Measure
Agent Selection & Agent Learning Goal Definition
• Agent Goal
• Agent Performance Measure
• Training Experience
Agent Machine Learning Design
• Request Code Re-Design
Implementation Phase
• Code Design
Incremental Implementation & Performance Measurement

Fig. 2. The process for introducing Learning Techniques

3.1 Systemic Goal and Performance Measure Selection

Software engineers must define in this phase two central elements related to learning issues: (i) the *Systemic Goal*, SG, and (ii) the *Systemic Performance Measure*, SP, which measures the system's performance gain. The systemic goal is the highest-level goal of the system, which is usually captured in the initial phase of a usual development process of multi-agent system. The systemic performance measure is the mechanism to evaluate the goal achievement. As a consequence, it is directly derived from the systemic goal. For example, these elements are defined in *LearnAgents* as: (i) SG: acquire travel packages for clients with as much profit as possible; (ii) SP: average score. The average score counts how much profit is being achieved in the system.

The goal abstraction is central in our approach since it is essential to determine the performance measures for each software agent (Section 3.2) and the learning techniques in later design stages (Section 3.3). As a consequence, we must also decompose the systemic goal in many subgoals. Note that the modeling of goal hierarchies is a common activity in MAS methodologies. However, the idea here is to detect and model the learning-specific goals. Goals work as a unified abstraction to connect the learning concerns and other basic concerns of the MAS at hand. These goals can be either derived from goals already defined for the basic functionalities of the agents or new ones that are associated with the systemic goal.

These subgoals are associated to agents, and create a system based on specialized agents. This process helps to reduce the complexity of the problem. For example, the goal of the *LearnAgents* is to acquire travel packages for clients with as much profit as possible. This profit is defined as the sum of the utilities of the eight clients in the TAC game, minus the costs of acquiring the travel goods in the auctions. This goal is presented in figure 3, and is decomposed in two other subgoals: build a knowledge base for decision making; and negotiate travel packages based on this knowledge base.

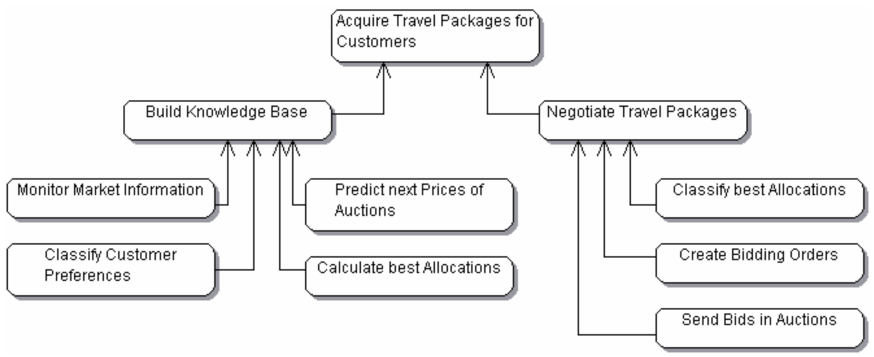

Acquire Travel Packages for Customers
Build Knowledge Base
Negotiate Travel Packages
Monitor Market Information
Predict next Prices of Auctions
Classify Customer Preferences
Calculate best Allocations
Classify best Allocations
Create Bidding Orders
Send Bids in Auctions

Fig. 3. The *LearnAgents* goal diagram

The knowledge base is built with the current prices of the auctions (sub goal: Monitor Market Information), the clients preferences - expressed as a utility table for each travel package (sub goal: Classify Customer Preferences), the future prices of the auctions (sub goal: Predict Next Prices of the Auctions), and a list of different scenarios based on the prices and client's preferences (sub goal: Calculate Best Allocations).

The negotiation of travel goods depends on the following actions: selection of a scenario with a high profit and a low risk (sub goal: Classify Best Allocations), definition of the number and type of travel goods to be bought based on the selected scenario (sub goal: Create Bidding Orders), and the price definition of the bids for the auctions (sub goal: Send Bids in Auctions).

In the agent identification process, we associate a sub goal in the goal diagram to one or more agents in the system. Table 1 illustrates the mapping of the sub goals to agent types in figure 1. We will select agents in the following phase that use machine learning to achieve its individual goal and the systemic goal defined above.

Sub Goal	Agent
Monitor Market Information	Hotel Sensor Agent, Flight Sensor Agent, Ticket Sensor Agent
Classify Customer Information	Hotel Sensor Agent
Predict Next Prices of the Auctions	Price Predictor Agent
Calculate Best Allocations	Allocator Master, Allocator Slaves
Classify Best Allocations	Ordering Agent
Create Bidding Orders	Ordering Agent
Send Bids in Auctions	Hotel Negotiator Agent, Flight Negotiator Agent, Ticket Negotiator Agent

Tab. 1. Mapping of the sub goals to agents

3.2 Agent Selection and Agent Goal Definition

The Agent Selection and Agent Goal Definition phase selects the agents in the system that perform complex plans and need a machine learning technique to improve the performance of the system. The goal of this phase is to establish a well-defined *Agent Learning Problem*, and defines three features for each selected agent: (i) *Learning Goal*, G; (ii) *Performance Measure*, P, which measures the performance improvement of the agent; and (iii) a *Training Experience*, E, which defines the knowledge acquisition process in learning.

The Flight and Hotel Price Predictor Agent is one of the agents selected for receiving a machine learning technique in the *LearnAgents*. The *Agent Learning Problem* for this agent is:

- G : predict future auction quote prices for hotel rooms;
- P : error between the predicted and real price; and
- E : use history of quote prices to build prediction knowledge.

3.3 Machine Learning Design

We present in this section a straightforward design that enables reuse and an easy maintenance. The process of maximizing the performance measure defined in the prior phases, normally, requires the test of different algorithms. Figure 4 presents a class diagram of the Flight and Hotel Price Predictor Agent using an object-oriented design pattern [14]. This design can be reused and refined to different contexts and applications.

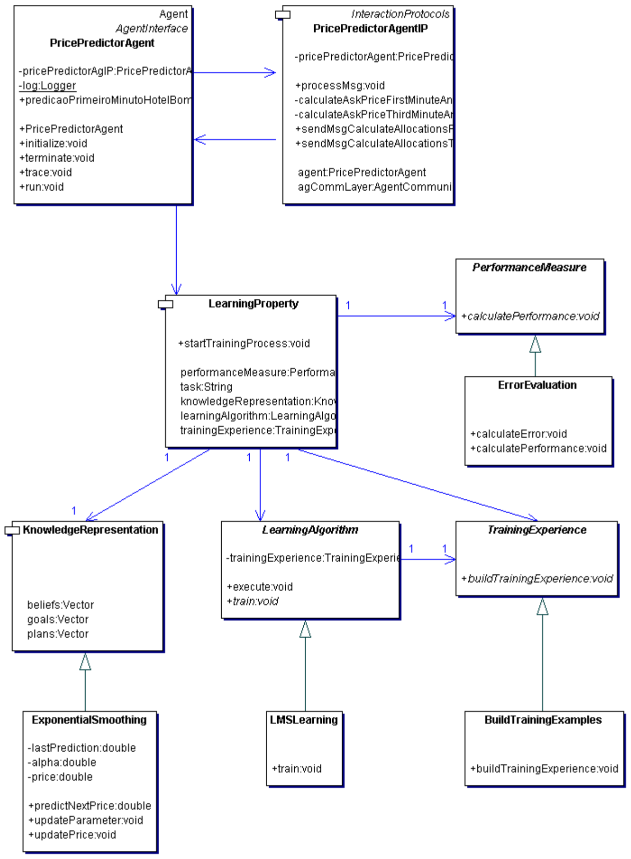


Fig. 4. The The Class Diagram of the Price Predictor Agent

The classes *PricePredictorAgent* and *PricePredictorAgentIP* are specialized classes that code the software agent's basic services, such as: sensory of the environment, event handling, message handling, etc.

The class called *KnowledgeRepresentation* is an abstract class of the data structure of the agent's knowledge. The monitor of the agent's performance measure (defined in section 3.2) is coded as an abstract class called *PerformanceMeasure*. The learning algorithm is an abstract class called *LearningAlgorithm*. The example generator of the agent's training experience (defined in section 3.2) is modeled as an abstract class called *TrainingExperience*.

The concrete classes *ExponentialSmoothing*, *ErrorEvaluation*, *LMSLearning*, and *BuildTrainingExamples* are the classes that respectively implement the abstract classes *KnowledgeRepresentation*, *PerformanceMeasure*, *LearningAlgorithm*, and *TrainingExperience*.

Several events can trigger the agent learning [13], including the execution of internal agent actions, throwing of exceptions, messages exchanged between agents, and events sensed in the external environment. The concrete classes *PricePredictorAgent* and *PricePredictorAgentIP* access the class *LearningProperty* to trigger the agent learning, and this class is the main interface to the learning pattern.

3.4 Implementation, Training, Testing and Evaluation

We are normally faced with three key aspects in the implementation phase of the selected software agents in section 3.2: (i) knowledge representation; (ii) learning algorithm; and (iii) training set used by the learning algorithm.

The first decision determines exactly what type of knowledge will be learned. This knowledge can be modeled as a function F that receives a state S and determines an action A, or $F: S \rightarrow A$. However, it may be very difficult in general to perfectly learn this function (representation of the knowledge), and normally we reduce the complexity and transform the problem to only learn some approximation of this selected function. We must then choose a reasonable representation of this knowledge for the agent. This best representation can be described as a linear weighted function, a collection of rules, a neural network, or a quadratic polynomial function. This design choice normally involves an important tradeoff because we would like to pick a representation that is as close as possible to the ideal knowledge. However, an expressive function requires more training data in the training phase.

The knowledge representation for the Price Predictor Agent is also modeled as a function called *NextAskPrice* that accepts the ask price A of the last game instances and produces the next ask price N ($NextAskPrice: A \rightarrow N$). The Price Predictor Agent uses a simple representation to describe the approximate knowledge. The following formula called Exponential Smoothing [15] is used to predict the next price: $PredictedAskPrice(n) = \alpha * AskPrice(n-1) + (1 - \alpha) * PredictedAskPrice(n-1)$, where α is a number between 0 and 1; and n is the n-th game instance. This formula is code in the class *ExponentialSmoothing* in figure 4. A Least Mean Squares (LMS) algorithm is used to adapt the value of α: $\alpha(n) = \alpha(n-1) + \beta * (AskPrice(n-1) - PredictedAskPrice(n-1))$, where β is a learning rate. This algorithm is coded in the class *LMSLearning* in figure 4.

A training set is required to build the knowledge of the agent, and we must select a process that selects this training set. Training examples can be obtained through a direct or indirect experience. In the direct experience, the designer can carefully select the best training examples that lead to a good approximate knowledge. In the indirect experience, the approximate knowledge must suggest actions that will lead to already known states that improve the performance of the system, and unknown states that guide to new experiences. The exploration is important for indirect learning agents that are willing to discover much better actions for the long run. The class *BuildTrainingExamples* has code that executes a query in a database with auctions prices for games already played, and implements the *Training Experience* defined in section 3.2. The *ErrorEvaluation* class implements the performance measure also defined in section 3.2.

3.4.1 Incremental Development, Testing and Integration of Intelligent Agents

Instead of building the multi agent system with all intelligent agents in a single stage, we propose an incremental development. The first version of the multi agent system is composed of simple agents that do not use a machine learning algorithms. This first step is important to test the communication of the agents and the interaction with the environment. The phases depicted in figure 5 illustrate the process of integration of the selected agents in section 3.2 in order to improve the *Systemic Performance Measure* defined in section 3.1.

The incremental development starts with the code removal of one of the agents selected in the section 3.2. This agent code is built with the classes defined in section 3.3. We then test the agent individually to measure the *Performance Measure* defined in section 3.2, before re-integrating it back in the system. This test is built by the system engineer, and is normally small software modules that generate test cases. Code errors of the intelligent module are found at this point.

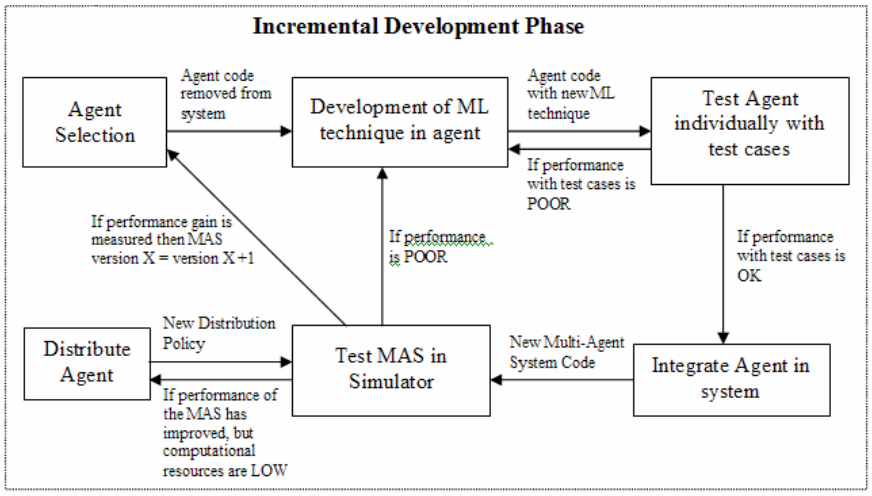

Incremental Development Phase
Agent Selection
Agent code removed from system
Development of ML technique in agent
Agent code with new ML technique
Test Agent individually with test cases
If performance with test cases is POOR
If performance gain is measured then MAS version X = version X+1
If performance is POOR
If performance with test cases is OK
Distribute Agent
New Distribution Policy
Test MAS in Simulator
New Multi-Agent System Code
Integrate Agent in system
If performance of the MAS has improved, but computational resources are LOW

Fig. 5. The Incremental Development Phase

The re-integration of the agent is done after the test cases confirm that the machine learning algorithm works correctly. These tests cases do not guarantee any performance improvement of the system. Therefore, a performance test is done with the multi agent system and the new intelligent agent. A simulator of the environment is required for this phase, and we only give a new version to the multi agent system if the *Systemic Performance Measure* defined in section 3.1 has improved. When performance gain is not achieved, the system engineer must take one of the two decisions: (i) Modify the code and test again; or (ii) Re-model the design as shown in figure 2.

However, there are cases when a performance gain is achieved but the new intelligent agent starts to use too much computational resources such as CPU and memory. This slows down the system, and the responses of the system are delayed. Consequently, we propose the distribution of this new agent to another CPU in the network.

3.4.2 Incremental Development of the *LearnAgents*

The first version of the *LearnAgents* was built with only reactive agents [1]. Our main goal at this point was to test the communication between the agents and the execution of the system in the environment. Although we were not expecting a good average score, a benchmark was executed to measure the evolution of the performance gain as shown in table 2. The benchmark used in table 2 is calculated against 7 simulated competitors called dummy agents [12]. The system plays 100 games in the benchmark against the dummies in order to evaluate the average performance and to smooth the variations in client preferences. The simulator of the real competition can be downloaded in the TAC web site[12].

The first selected agent was the Allocator Agent (version 1.0 in table 1). The agent calculates different scenarios based on the prices in the knowledge base. These scenarios are travel good allocations (only flights and hotels) that are calculated using an integer programming solver called XPRESS [16]. The solver is executed not only in search of the optimal allocation, but as many best allocations as the solver can optimize in 25 seconds. After testing the new Allocator Agent with the test cases, we also modified the Ordering Agent with a new heuristic. The Ordering Agent had to decide the quantity of the required travel goods and the maximum price of the bids based on the allocations produced by the Allocator Agent. The multi agent system obtained a score of 1372 in the benchmark against 7 dummies.

The Negotiator Agent was the next selected agent for adding of an intelligent module. This module uses a min-max procedure [17] and an evaluation function calculated by a neural network [13] and a reinforcement learning [13] technique. The Negotiator Agent's goal is to send optimal bids to the auctions with the minimum expenditure. Therefore, the agent adapts the weights of the neural network and searches for an evaluation function that produces these optimal bids. The benchmark of the version 2.0 was 1752.

The next selected agent to include the intelligent module was the Price Predictor Agent. This agent uses a price history in the knowledge base to forecast ask prices. The technique implemented in the intelligent module is called moving average [15]. This agent is important for the system, because it helped to deal with uncertainty of the auction prices. Consequently, the Allocator Agent could now calculate scenarios based on these predicted prices. This version 3.0 now scored 2224 in the benchmark

***LearnAgents* Version**	**Agent Selected / Intelligent Module**	**Average Score**	**Games Played**
0.0	---	-1855	100
1.0	Allocator / Solver – Integer Programming Model, Ordering Agent / Heuristic	1372	100
2.0	Hotel Negotiator / Minimax, N.N. and Reinforcement Learning	1752	100
3.0	Price Predictor (hotel) / Moving Average	2224	100
4.0	Allocator (Ticket) /Solver - Integer Programming Model, Ordering Agent / Heuristic	2856	100
5.0	Allocator / Solver – Integer Programming Model (second version)	3370	100
6.0	Price Predictor (flight) / Maximum Likelihood	3705	100

Tab. 2. The Evolution of the MAS and the Performance Gain

We then decided to improve the Allocator Agent in the version 4.0 of the system. The flight and hotel goods were the only allocations calculated by the integer programming solver. We extended the model and included the entertainment tickets, and modified the Ordering Agent heuristic to decide the quantity of tickets to buy based on these new allocations. We executed the benchmark and achieved a score of 2856. This agent improved the overall score, but the system started to present delays in some responses to the market. This was clear evidence that CPU resources were low, so we created an Allocator Slave to solve the problem. This new agent also used an integer programming solver that calculated the tickets allocation, and was now executing in another CPU.

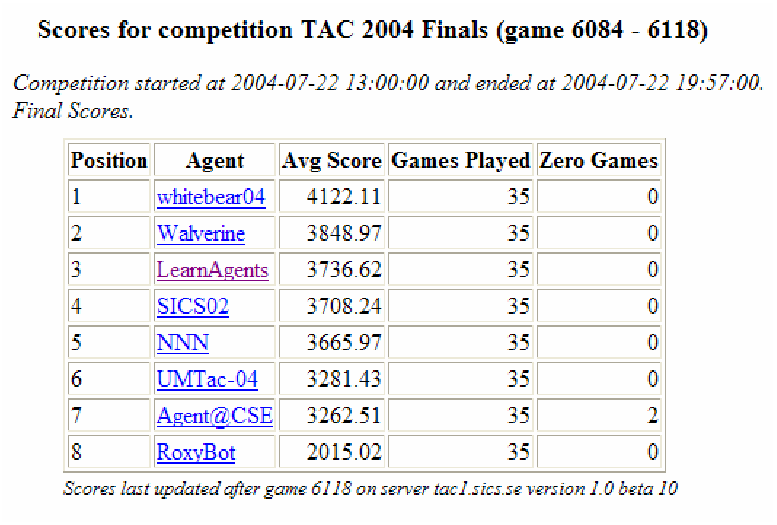
The Allocator Agent was again selected for improvements in version 5.0. The integer programming model was modified to include some parameters for tuning purposes. The travel good allocations are now calculated based on these parameters. We can select parameters that produce allocations with more hotel rooms from Shoreline Shanties [12] (the bad hotel) then the Tampa Towers [12] (the good hotel) in this version. The benchmark with the best parameter configuration produced a score of 3370.

Our last selected agent was the Price Predictor. This agent was only predicting hotel auction prices and we now wanted the prediction of flight auction prices. The flight auction prices are modified according to a stochastic function. The process uses a random walk that starts between $250 and $400 and is perturbed every 10 seconds. We included an intelligent module based on a technique called maximum likelihood [13] to predict a parameter of the stochastic function that was not revealed directly to the agents. The predicted parameter is then used to define the expected value of the flight auction prices. This benchmark achieved a score of 3705 in this 6.0 version.

4 Final Comments

The importance of the learning property in today's software systems is reflected by the support provided for this property in existing tools [18][19] and implementation frameworks [3]. However, software engineers have largely relied on their experience and intuition in order to develop adaptable MASs. This paper presents a systematic approach for building intelligent agents with machine learning techniques. These agents are now able to perform complex plans, adapt their beliefs and achieve predefined goals. In complex and open environments with many cooperating agents, it is important to have a system that is able to adapt to unknown situations. Learning techniques are crucial to the development of multi-agent systems since they provide well-known strategies to support the construction of adaptable agents.

The incremental development was extremely important for the development process of the *LearnAgents*. If all the techniques were added at once, we would never be able to evaluate the individual performance gain of each technique. This step-by-step process also prevents the development of agents that deteriorate the performance, because we only include techniques that present improvements. The process presents an easy method to add new agents with different techniques, and minimizes error development in concurrent and distributed agents.



Scores for competition TAC 2004 Finals (game 6084 - 6118)

Competition started at 2004-07-22 13:00:00 and ended at 2004-07-22 19:57:00.
Final Scores.

Position	Agent	Avg Score	Games Played	Zero Games
1	whitebear04	4122.11	35	0
2	Walverine	3848.97	35	0
3	LearnAgents	3736.62	35	0
4	SICS02	3708.24	35	0
5	NNN	3665.97	35	0
6	UMTac-04	3281.43	35	0
7	Agent@CSE	3262.51	35	2
8	RoxyBot	2015.02	35	0

Scores last updated after game 6118 on server tac1.sics.se version 1.0 beta 10

Fig. 6. The Scores for the Finals of the TAC 2004

The figure 6 presents the final results of the *LearnAgents* in the 2004 TAC Classic tournament. The tournament had 14 participants from universities, research institutes and technology companies from around the world (USA, Brazil, France, Sweden, Netherlands, Israel, Macau, China, Japan, etc.). The *LearnAgents* finished the tournament in the third place. An interesting observation is the similarity between the best score in the simulation process (3705) and the score in the final round of the competition. Although dummy agents use very naïve strategies, our system presents a strategy that is able to adapt to different competitors in an efficient manner.

This process emerged from the long-term application of our method to different multi-agent systems. We believe there is a need for a software engineering process for the disciplined introduction of learning properties in software agents through different development stages. This systematic approach helps the development team of a MAS to include machine learning techniques in adaptive environments, and consequently, leverage the performance of the system.

References

1. Wooldridge, M.: Intelligent Agents. In: G. Weiss. Multiagent systems: a modern approach to distributed artificial intelligence. The MIT Press, Second printing, 2000.
2. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co, 1999.
3. Telecom Italia Lab. JADE Programmer's Guide. http://sharon.cselt.it/ projects/ jade/ doc/ programmersguide.pdf, Feb. 2003.
4. Kendall, E.; Krishna, P.; Pathak, C.; Suresh, C.: A Framework for Agent Systems. In: Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (editors), John Wiley & Sons, 1999.
5. Zambonelli, F.; Jennings, N. R.; Wooldridge, M.: Developing multiagent systems: the Gaia Methodology. ACM Transactions on Software Engineering and Methodology, 12 (3) 317-370, 2003.
6. Giunchiglia, F.; Mylopoulos, J.; Perini, A.: The tropos software development methodology: processes, models and diagrams. Proceedings of the first international joint conference on Autonomous agents and multiagent systems, 2002.
7. Milidiu, R.L.; Lucena, C.J.; Sardinha, J.A.R.P.: An object-oriented framework for creating offerings. 2001 International Conference on Internet Computing (IC'2001) June 2001.
8. Sardinha, J.A.R.P.: VGroups – Um framework para grupos virtuais de consumo. Master's dissertation – Departamento de Informática – PUC-Rio. March 2001.
9. Sardinha, J.A.R.P.; Milidiú, R. L.; Lucena, C. J. P.; Paranhos, P. M.: An OO Framework for building Intelligence and Learning properties in Software Agents. Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003) at ICSE 2003, Portland, USA, May 2003.
10. Sardinha, J.A.R.P.; Milidiú, R.L.; Lucena, C.J.P.; Paranhos, P.M.; Cunha, P.M.: LearnAgents - A multi-agent system for the TAC Classic. Poster Session at The Third International Joint Conference on Autonomous Agents & Multi Agent Systems (Trading Agent Competition), July 2004, New York, USA.
11. Garcia, A.: From Objects to Agents: An Aspect-Oriented Approach. Doctoral Thesis, PUC-Rio, Computer Science Department, Rio de Janeiro, Brazil, April 2004.
12. TAC web site.: http://www.sics.se/tac.
13. Mitchell, T. M.: Machine Learning. McGraw-Hill, 1997. ISBN 0070428077.

14. Sardinha, J.A.R.P.; Garcia, A.F.; Milidiú, R.L.; Lucena, C.J.P.: The Agent Learning Pattern. Fourth Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'04. August, 2004, Fortaleza, Brazil.
15. Bowerman, B. L.; O'Connell, R. T: Forecasting and Time Series: An Applied Approach. Thomson Learning, 3rd edition, 1993. ASIN: 0534932517.
16. Dash Optimization. http://www.dashoptimization.com
17. Russell, S.; Norvig, P.: Artificial Intelligence. Prentice Hall, 1995. ISBN 0-13-103805-2.
18. Computer Associates (CA) CleverPath web site: *http://www.ca.com/*
19. DB2 Business Intelligence web site: http://www-306.ibm.com/software/data/db2bi/