Supply Chain Management Sales Using XCSR

Maria A. Franco
Grupo de Inteligencia Artificial
Universidad Simón Bolívar
Caracas, Venezuela
maria@gia.usb.ve

Ivette C. Martínez
Grupo de Inteligencia Artificial
Universidad Simón Bolívar
Caracas, Venezuela
martinez@ldc.usb.ve

Celso Gorrín
Grupo de Inteligencia Artificial
Universidad Simón Bolívar
Sartenejas, Venezuela
celso@gia.usb.ve

ABSTRACT

The Trading Agent Competition in its category Supply Chain Management (TAC SCM) is an international forum where teams construct agents that control a computer assembly company in a simulated environment. TAC SCM involves the following problems: to determine when to send offers, to determine final sales prices of offered goods and to plan factory and delivery schedules. The main goal of this work was to develop an agent called TicTACtoe, using Wilson's XCSR classifier system to decide the final sales prices. We develop an adaptation to the classifier system, that we called blocking classifiers technique, which allows the use of XCSR in an environment with parallel learning. Our results show that XCSR learning allows generating a set of rules that solves the TAC SCM sales problem in a satisfactory way. Moreover, we found that the blocking mechanism improves the performance of the XCSR learning in an environment with parallel learning.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Design, Experimentation, Management

Keywords

Supply Chain Management, TAC SCM, Classifier Systems, XCSR

1. INTRODUCTION

The supply chain management embodies the management of all the process and information that moves along through the supply chain from the supplier to the manufacturer right through to the retailer and the final customer. The supply chain management nowadays is one of the most important industrial activities. Planning the activities through the supply chain is vital to the competitiveness of manufacturing enterprises. According to [3], "while today's supply chains are essentially static, relying on long-term relationships among key trading partners, more flexible and dynamic practices offer the prospect of better matches between suppliers and customers as market conditions change".

The Trading Agent Competition of Supply Chain Management (TAC SCM) was designed to expose the participants to the typical challenges present in the dynamic supply chain. These challenges include competing for the components provided by the suppliers, managing the inventory, transforming components into final products and competing for the customers. These problems can be classified into three main problems: purchases, production and sales.

David Pardoe and Peter Stone made experiments applying different learning techniques in sales decisions of TAC SCM agents[6]. One of their main conclusions was that winning offers in TAC SCM is a very complex problem because the winning prices may vary very quickly. This is why this work affirms that taking decisions based on previous states of the actual game is inaccurate, while using information taken from a lot of previous games shows better results.

The goal of this work is to present an approach to the TAC SCM problem using an evolutionary reinforcement learning technique. We specifically use XCSR to solve one of the most important sales problems: pricing the components in order to compete over the market and maximize the profit at the same time.

2. TAC SCM

TAC SCM competition was designed by a team of researchers from the e-Supply Chain Management Lab at Carnegie Mellon University and the Swedish Institute of Computer Science (SICS)[3]. In this contest each team has to develop an intelligent agent capable of handle the main supply chain management problems (which orders accept, decide the sale price for products, compete over the market, among others).

Each agent competes against other agents in a simulation that lasts 220 days and includes customers and suppliers to deal with. The main goal of the competitors is to maximize the final profit by selling assembled computers to the customers. The profit of an agent is calculated by subtracting production costs to the earned money. This profit reflects in the amount of money the agents have at the end of the game, which indicates which agent is the winner.

Each TAC SCM simulation has three actors: customers who buy computers, manufacturers (agents) who produce and sell computers, and suppliers who provide the components to the manufacturers. A detailed description of these actors can be found in [3].

At the beginning of each day the agent receives “request for quotes” (also known as RFQs) from the customers. Then the agent decides which ones to accept and which should be the final offer price. After the agent sends its offers, it should receive the actual orders from the customers (the offers that were accepted). Then the agent decides when to produce and deliver the order, and most importantly, how much components it should buy to accomplish the production schedules. In order to buy the components, the agent sends the suppliers RFQs. When the agents receive offers from the suppliers they place orders and wait until the components are delivered.

Each team that competes in a TAC SCM game should construct a manufacturer agent that has to deal with the main decisions of the supply chain management: how much components it should buy, when to produce an order and which RFQs it should accept and which will be the final price. In this work, these problems will be referred as the purchase problem, the production problem and the sales problem.

We approached the sales problem using an evolutionary reinforcement learning technique. The other problems were solved using simple static strategies (explained in Section 4) in order to evaluate the impact of the learning system on the sales problem.

3. XCSR

XCSR is a version of the of the XCS first described by Wilson[9] with the only difference that it accepts real numbers as inputs[10]. The inputs of the decision we wanted to make were all real and the decisive thresholds needed to be found dynamically. We also decided to use XCSR because the rule system can constantly adapt to new environments using a fixed rate of exploration and the rules that it generates are interpretable by human beings.

4. TICTACTOE

TicTACtoe is our approach to the TAC SCM problem. It was initially developed as Franco and Gorrín’s undergraduate thesis [4]. TicTACtoe has three modules: Purchase, Production and Sales (see Figure 1). Each module manages one of the sub-problems involved into the Supply Chain Management. Every module takes its own decisions using information from the environment and the other modules. On the next subsections we focus on the details of these modules.

In addition to these modules, we developed a structure that we called agenda, which acts as an organizer for the agent. The agenda keeps track of: orders scheduled for production, possible order commitments, actual produced orders[1] and possible future inventory [2]. This allows to record decisions taken by the agent and to consider events that would happen in the future, in order to make more accurate decisions.

[1]This variation on the production schedules is do to the lack of components.

[2]The future possible inventory is based on suppliers offers already accepted by the agent.

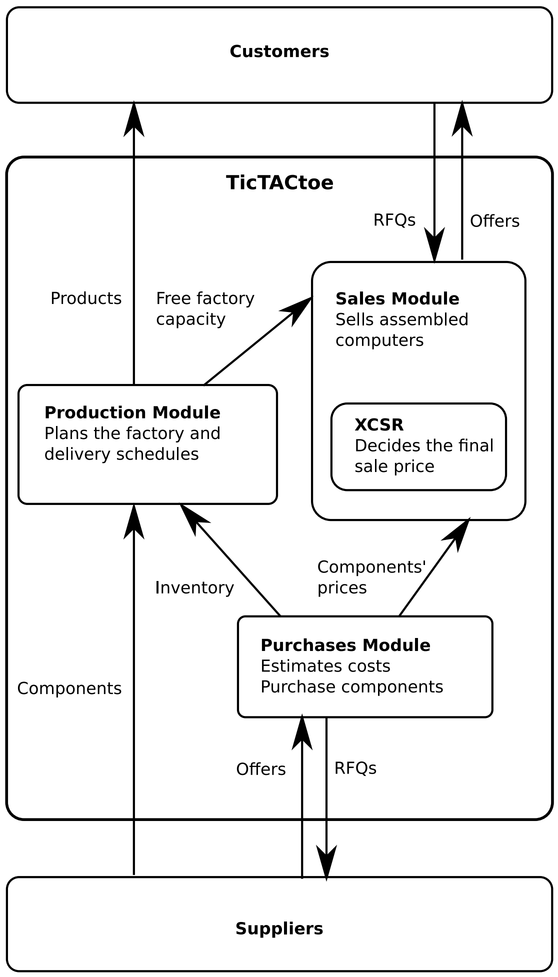


Figure 1: TicTACtoe’s Architecture

4.1 Purchases

The purchase module is in charge of sending RFQs to the suppliers in order to get the current prices and accept the offers from them in order to buy components for production.

4.1.1 Suppliers RFQ creation

First the agent needs to calculate how many components are needed for production within the next ten days. These calculations are based on the actual inventory, orders scheduled for the next ten days and component orders that have been placed already. The agent always sends the RFQ to the favorite supplier for the needed component, which is the one who has given the best prices lately. Instead of sending RFQs with a certain quantity, the agent only asks the other suppliers for the current prices in order to update them and update the favorite supplier if necessary.

The favorite supplier is selected in order to get lower prices. This is based in the idea that the state of a supplier does not change drastically, so if a supplier gives an agent the best price, probably it would be continue giving the best prices for some time. There is only one favorite supplier for each one of the components.

4.1.2 Accepting the offers

When supplier offers arrive, TicTACtoe accepts first complete offers and then the ones that vary the quantity. When suppliers are not able to deliver the products the agent asks for, the supplier sends two types of adjusted offers: the ones that vary the quantity and the ones with a later due date. Once the order is set, the agent adds a record for the components arrival in the agenda to calculate the future inventory. The updates on the base price are calculated by a weighted average as shown in equation 1:

$$P_d^c = S_d^c \cdot \alpha + P_{d-1}^c \cdot (1 - \alpha) \tag{1}$$

where P_d^c is the base price for component c in day d, S_d^c is the supplier's price for component c in day d and α is a constant for weighting.

The agent also updates the base price and the favorite supplier for each one of the components.

4.2 Production

The production module is in charge of scheduling the productions of the active orders (the orders that are waiting to be elaborated or delivered).

This module organizes the active orders by due date and by penalty in case they are not delivered on time. The agent verifies each one of them checking if there is enough inventory of products to deliver the order. If the agent has enough products the order is delivered. This strategy is used by PhantAgent[8] to avoid extra storing charges. In case there are not enough components to deliver the order, the agent verifies if the order is beyond the latest possible delivery date[3]. In this case the customer would not receive the order anymore and the agent needs to cancel it and free all the components and products associated with this order in order to use them to fulfill other orders.

If there are not enough products to fulfill the order but the customer can still wait for it, the agent tries to produce it. However the agent only produces an order if the order is marked in the agenda to be produced that day.

To produce an order scheduled for a specific day the agent checks if there are enough components. If a component in inventory is taken into account to fulfill the order, the agent reserves it in order to not use it in future orders.

When there are not enough components to produce the desired quantity, the agent produces the maximum quantity allowed. If the agent cannot produce an order completely, it schedules the order for the next day.

At the end of the day the production module determines the number of late orders and the number of active orders and sends this information to the sales module in order to adjust the quantity of free cycles the agent can offer. This makes the agent save cycles to use them in late order production.

4.3 Sales

The sales module is in charge of pricing the products and dealing with the customers. This module checks everyday the customer's RFQs and sends offers to the ones that meet the following characteristics: have a reserve price higher that its base price[4] and have a due date earlier than the end of the simulation. Then, for each one of these RFQs the agent uses a set of rules generated using a XCSR, that determines the discount factor over the reserve price[5]. The implementation of XCSR is explained in detail in section 5.

The final offer price is determined by equation 2, where *BasePrice* is the calculated cost of the product based on recent experiences, *ReservePrice* is the reference price determined by the customer and d is the discount factor determined by the XCSR.

$$\mathit{OfferPrice} = \mathit{BasePrice} + \mathit{Revenue} \cdot (1 - d) \tag{2}$$

$$\mathit{Revenue} = \mathit{ReservePrice} - \mathit{BasePrice} \tag{3}$$

Once the agent calculates the offer price for each one of the RFQs, the agent sorts them by revenue, and checks if there are enough free cycles to produce them.

In order to save production cycles for future orders, that would need to be delivered earlier (as done in [7]). The agent always tries to produce an order as late as possible according to its due date.

On the other hand, free cycles for a day are multiplied by a factor between 0 and 1, inversely proportional to the quantity of late orders that the agent has. This helps the agent to get on schedule again, by leaving some cycles for the production of late orders. If there are enough free cycles to produce an order on the chosen day, the agent sends an offer in response to the RFQ, and updates the agenda adding the possible commitment and reserving the cycles needed for its production[6] In case the agent does not have enough free cycles, it verifies if the next day's non-reserved inventory is enough to fulfill this order, in order to schedule the order for delivery on the next day.

Customers only accept the best-priced offer for each RFQ they send. If a customer accepts an agent offer, then the commitment is rewritten in the agenda as an order. On the contrary, if a customer rejects an offer, the commitment is removed from the agenda to free all the components and cycles associated with it.

5. XCSR INSIDE TICTACTOE

One of the most important decisions in the supply chain management is to decide the final price for the products, because it should be low enough to win the order and at the same time high enough to maximize the agents profit.

The decision taken by the XCSR is the final price discount the agent should give to an order, which is done in the Sales Module. The Sales Module uses the XCSR library through two methods:

getPriceDiscountFactor(): Gets the state from the environment, finds the match set and the action that should take effect, and associates the action set to the RFQ, in order to reward it later.

rewardAndRunGA(actionset, reward): Rewards the action set and saves the error information for future statistics.

[3]The latest possible deliver date is determined when the customer sends the first RFQ.

[4]The base price is calculated by the agent as the sum of the estimated prices of all the components needed to produce the product.

[5]The reserve price is the maximum price a customer is willing to pay for an order.

[6]This helps the agent to calculate correctly how much free cycles are left on a day for the production of further orders.

5.1 Classifiers structure

In the following sections we explain the structure used to represent the TAC SCM sales problem using real inputs and discrete actions.

5.1.1 Condition

There are values known by the agent that provide important information for the agent's decisions. It is impossible to include all this values in the classifiers structure so we had to pick the ones we though were the most important for the decision we wanted to take. After analyzing a lot of values we decided that the features of the classifier would be:

x_1 Rate of late orders over the total of active orders. This determines how much work is late and how convenient is to make a good offer that can become an order later.

$$x_1 = \frac{lateOrders}{totalOrders} \tag{4}$$

x_2 Rate of the factory cycles that remain unused the day before. This helps the agent determine if it should raise or lower the price discount. For example, if the factory is full, the agent should give low discounts in order to try to finish with the all the active orders it already has, before getting new ones.

$$x_2 = \frac{freeFactoryCapacity}{totalFactoryCapacity} \tag{5}$$

x_3 Rate of the base price over the reserve price indicated by the customer. This represents how profitable an order would be. The agent discards the cases when the base price is higher than the reserve price.

$$x_3 = \frac{basePrice}{reservePrice} \tag{6}$$

x_4 The number of days between the actual date and the day the order should be delivered. This indicates how much time the agent has to produce and deliver an order. This value is scaled between 0 and 1 considering that the due dates are, at most, 12 days after the actual date.

$$x_4 = \frac{(dueDate - day)}{12} \tag{7}$$

x_5 The actual day of the simulation normalized by the maximum number of days a game has. This value is very important because there are different situations as the days go by. For example, in the middle of a simulation components start to be scarce and their prices start rising. But these variations during the simulation depends on the decisions the agents are taking so this is an important feature for the classifier, because it helps the agent take decisions according to the changes in the simulation.

$$x_5 = \frac{day}{220} \tag{8}$$

All the features are scaled between 0 and 1, to use these values as upper and lower bounds. This is because the limitation of the library explained in Section 6.1.

5.1.2 Action

Our implementation of XCSR has 10 actions that represent the different discounts over the reserve price; i.e., the difference, expressed in a percentage, between the base price and the reserve price determined customers. The different discounts go from 0% to 90% with 10% steps.

5.1.3 Reward

The reward is determined by the profit obtained through a RFQ, scaled by the amount of money that implied its fabrication when the agent sent the offer. There are three different moments when the agent can reward an action set.

When the offer is not accepted by the customer. In this case the RFQ didn't make the agent earn or lose any money and the reward is zero.

When the order is delivered. In this case we consider the money earned by the sale and the money lost because of the penalty (if the order was delivered late). The profit and loss are scaled by the investment of the agent calculated based on the base price of the product. The reward in this case is calculated using the equation 9.

$$reward = (profit)^2 - (loss)^2 \tag{9}$$

where

$$profit = \frac{offeredPrice}{basePrice} \tag{10}$$

$$loss = \frac{\max(day - duedate, 0) \cdot penalty}{basePrice \cdot quantity} \tag{11}$$

When we scale the profit and the loss using the base price we obtain a percental value of the money earned with the order. We could think that a good approximation for the reward function is to subtract the expenses to the profit; but this gave us the net earnings that are not the same for the different products. Products have different prices due difference on theirs production cost, but they can have the same profit margin. Therefore, it is more appropriate rewarding a classifier based on the gained percentage.

When the order is canceled without being delivered. In this case the agent didn't produce the order on time and it only produces losses for the agent thanks to the penalties the agent had to pay to the customer. The money invested in the production is not considered as an expense, because these products can be used to fulfill another order. In this situation, the reward is calculated by the equation 12, which is very similar to equation 9 eliminating the term corresponding to the profit.

$$reward = -(loss)^2 \tag{12}$$

In equations 9 and 12, the terms corresponding to the profit and the loss are squared so the situations where they are significantly greater give the classifiers a stronger reward.

6. IMPLEMENTATION DETAILS

The XCSR library implementation is based on Butz's XCS library (Version 1.0)[1]. We adapted this library to XCSR using a lower and upper bound notation proposed by Loiacono[5].

6.1 Don't care

The don't care in our library was implemented as the absence of the lower or the upper bound, depending on the allele we want to modify. To implement don't cares we had to put a restriction to the data: all the features should be bounded between 0 and 1. Putting a don't care in an allele, is equivalent to put 0 or 1 depending if it is a lower bound or an upper bound. On this way, we open the range to the maximum limit and the allele would classify all the states.

6.2 Classifier Subsumption

The subsumption rules in Butz's library were oriented to boolean features, therefore its was necessary to redesign these rules, so they adapt to our classifier structure, where all the features are bounded between 0 and 1.

The rules used were the same rules used by Loiacono in [5] where a classifier is more general than another if all the ranges of the first classifier contains the second one. For example, (l_i, u_i) subsumes (l_j, u_j) if $u_i > u_j \wedge l_i < l_j$. The actions of the classifier should be the same for the subsumption to occur.

6.3 Dynamic population generation

The Butz's library had a dynamic population generation method, in which the population started empty and each time the algorithm generated an action set, it inserted new classifiers into the population until all the actions were covered. In other words, there is at least one classifier for each possible action. If there was no classifier in the population for a specific action, covering was made and the new classifier was inserted into the population. The advantage of this technique is that the population grows dynamically as the states occur in the experiment.

On the other hand, the population has a limited size and covering all the actions yields to the loss of old classifiers through the deletion algorithm when inserting the new ones. This might be very unfavorable, on advanced execution stages when the classifiers have evolved, because the algorithm could erase better classifiers and in this way the global fitness of the population decreases. This is why we decided to change the population generation method so it was used until the population reaches the size limit. After that, when covering is necessary we only generate one rule with a random action. This helps finding better solutions to the problem. Nevertheless, we continue inserting and deleting individuals through the genetic algorithm over the action set.

6.4 Crossover operators

We implemented a two-point crossover operator between conditional ranges that generates new individuals with valid ranges. This means that only the points between an upper bound and a lower bound can be chosen. This crossover operator is equivalent to boolean two-point crossover operator, because the crossover only crosses full conditions. This guarantees that the ranges of the new individuals are not empty or false, because they are combinations of the parent ranges.

6.5 Additional adaptations

Additional adaptations were necessary to include XCSR in our TAC SCM agent TicTACtoe due to the characteristics of the problem.

6.5.1 Parallel learning and the blocking mechanism

In our classifier system, the reward of an action set depends on the amount of money the agent wins o loses when making the corresponding offer. The agent only knows this value a few days after using the classifier. This differs from the classic problems solved by XCS (e.g. boolean multiplexer), where the reward arrives immediately after using a classifier.

Considering the delayed reward, it is necessary to save the action set associated to the order in order to give these classifiers a reward when the agent gets the final result for the order.

Since we are interested in continuing learning while a classifier waits for its reward, classifiers were used in multiple learning iterations parallel to each other. This characteristic, in addition to the delayed reward, presents a new problem to us which we called parallel learning.

The parallel learning problem occurs when a classifier that is waiting for a reward is selected for deletion or subsumption. Since these mechanisms could be executed by any parallel learning iteration, it could erase this classifier based on information that is not updated. Consequently, the knowledge represented by this classifier and its upcoming rewards are lost.

In order to avoid the deletion of the classifiers expecting a reward to come based on incomplete information, we implemented a simple counting semaphore. In this system, each classifier has a counter that indicates the number of rewards the classifier is expecting (a single classifier participates in a lot of decisions each day and needs to wait a reward for each one of them). This way, we only consider for deletion the classifiers that are not blocked, the ones that have their counter in zero.

We had to add also another important restriction in the subsuming mechanism: a classifier can not be subsumed if it is blocked, because its information is not entirely up to date to become part of another classifier. The blocked classifiers may participate in all the other mechanisms like crossover, mutations, exploration and exploitation.

6.5.2 Variable exploitation and exploration rates

We also changed the use on the exploitation and exploration mechanisms in the base library. This library interleaves between exploitation and exploration, rewarding the classifiers only during the exploration and taking learning statistics only during exploitation.

In this problem, the classifier system learns while the agent competes in a simulation. Since the decisions taken by the XCSR affects the final result, regardless of whether it was determined by exploration or exploitation, we changed the algorithm in order to reward the classifiers in both cases.

We also changed the action selection method so the exploration and exploitation were stochastic with a rate starting at zero and growing linearly while days go by in the simulation. This makes the algorithm explore a lot at the beginning of the learning, and according the time passes, the algorithm exploits more and explores less. This rate increases until it reaches a threshold, and then this rate remains constant. A linear increment was chosen for simplicity.

Param.	Value	Param.	Value	Param.	Value
α	0.1	β	0.2	δ	0.1
ν	5	θ_{GA}	25	ϵ_0	10
θ_{del}	20	χ	0.8	μ	0.04
$p_{\#}$	0.1	p_I	10.0	ϵ_I	0
F_I	0.01	θ_{sub}	20	θ_{mna}	1
N	1000				

Table 1: Parameters used in TicTACtoe's XCSR

6.6 XCSR Parameters

The parameters used in our implementation of XCSR are shown in Table 1. The meaning of these parameters is explained in [2]. Also, the sources of TicTACtoe can be found in `http://www.gia.usb.ve/~maria/tictactoe`.

7. EXPERIMENTS AND RESULTS

We designed two experiments to test the effectiveness of the proposed mechanisms. In these experiments each agent plays separately against five dummy agents, taken from the server library.

In each experiment agents ran 40 games. In the first game the TicTACtoe's classifiers population was empty. At the end of each game the population was saved, and at the beginning of the next game it was recovered and established as the initial population. All data presented in figures corresponds to the last 25 games. The first 15 games were taken as the training stage.

Each games length is 220 simulations days, each simulation day duration is 5 seconds[7], considering than none of the agents would need more time to complete all its daily actions.

The agents' performance was evaluated using two measures:

(a) Final result: the final amount of money in the agent's bank account. This final result indicates how much money the agent earned and how profitable were its investments.

(b) Received offers: the number of offers that turned into orders because the customers accepted them. This value indicates the percentage of the market the agent served. This is directly linked to the decision taken by the XCSR, because if the agent gives a better price, it receives more orders

The combination of these measures will show how effective was the XCSR learning, considering that we want to learn a discount strategy that maximizes the agent's revenue by winning profitable and manageable orders. There are also additional measures considered in the experiments like factory utilization, interests, penalties, components costs and storage costs.

7.1 Experiment 1: TicTACtoe's performance

The goal of this experiment is compare three different strategies to determine the price discount. These strategies are: learning using XCSR (`L-TicTACtoe`), `Random`, and `Static`. All this strategies were tested using the base version

[7]The standard parameters for the games are 220 simulation days with a duration of 15 seconds.

		Penalties (US$)	Interest (US$)	Factory utilization (%)
L-TicTACtoe	μ	412.041	241.354	69.120
	σ	703.405	93.497	8.053
Random	μ	5.596.269	66.143	85.560
	σ	6.221.391	247.714	6.904
Static	μ	10.535.055	-781.717	93.44
	σ	10.402.476	594.105	1.529
Dummy	μ	882.427	37.235	34.560
	σ	1.539.077	93.975	4.282

Table 2: Mean (μ) and variance (σ) of the penalties, interests and factory utilization of TicTACtoe agent using different pricing strategies and the dummy agent.

of TicTACtoe. In this experiment, we also compare `L-TicTACtoe` with the dummy agent provided by the server.

The learning version of TicTACtoe, `L-TicTACtoe`, uses an exploitation rate of 30%, a population size of 1000 classifiers and uses the blocking algorithm. The other versions of TicTACtoe involved are `Random` and `Static`. The first one decides the price discount randomly while the second one gives a discount on day d as follows:

$$discount(d) = \begin{cases} 80\% & \text{if } freeFactoryCapacity_{d-1} > 80\%, \\ 10\% & \text{if } freeFactoryCapacity_{d-1} < 5\%, \\ 30\% & \text{all other cases.} \end{cases}$$

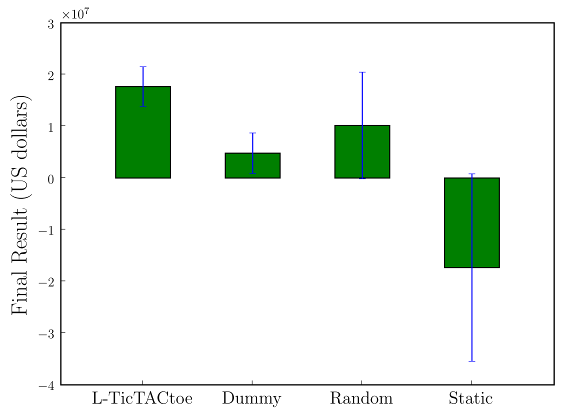
where $freeFactoryCapacity_{d-1}$ is the percentage of free factory capacity on simulation day $d-1$. These "naive" rules try to avoid factory saturation raising prices every time the number of free factory cycles goes below 5% and tries to attract customers when this value goes over 80%.

7.1.1 Results

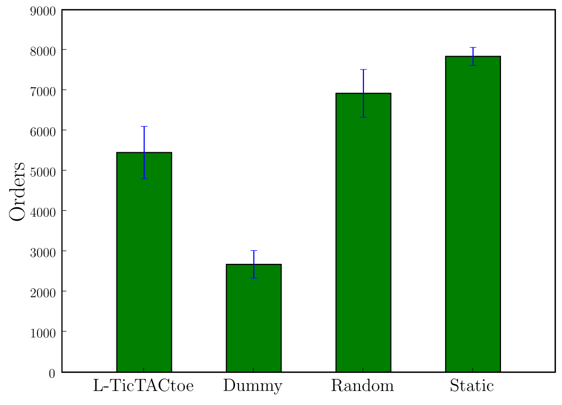
Figure 2(a) shows the global performance of the agents. These results clearly show that `L-TicTACtoe` outperforms `Random` and `Static`, with an average result twice as high as `Random` and four times higher that the dummy agent. Considering that these agents only differ in their pricing strategy, it is evident that a change on this strategy affects the global performance of the agent.

Even though we could expect that `L-TicTACtoe` manages more orders than the other agents, Figure 2(b) reveals that `Random` and `Static` win more offers. Nevertheless, Table 2 indicates that `Random` and `Static` are delivering more orders late; therefore, incurring in more penalties. These results show that the pricing strategy of these agents is less advantageous because they commit to orders which they cannot deliver on time and hence they are penalized.

Respecting the interests of the agents, we can observe on Table 2, that `Static` gets negative interests. In other words, this agent had to pay the bank for having a negative balance in its bank account. This indicates that the strategy taken by `Static` is deficient, because it incurs in negative balances on most of the simulation days. On the other hand, `L-TicTACtoe` is the agent that earns more interests from the bank and presents the lowest variance. This shows that this agent has a more stable behavior in terms of the bank account balances.



(a) Final Result



(b) Received Orders

Figure 2: Comparison of the dummy agent and the TicTACtoe agent using different pricing strategies.

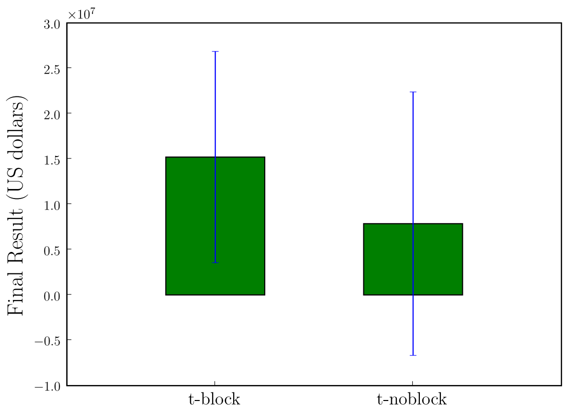
Regarding the factory utilization, we can appreciate that the agents `Random` and `Static` achieve a higher factory utilization. High factory utilization suggests a proficient management of the productive capacity. However, the penalties obtained by these agents demonstrate that these agents are surpassing their production capacity. `L-TicTACtoe` does not use the factory as much as these agents, but still presents a better solution to this problem because it served efficiently a considerable portion of the market.

Finally, through this experiment we can confirm that the strategy used by `L-TicTACtoe` improves the global performance of our solution to the TAC SCM problem. Furthermore, the static and random strategies show poor results as a consequence of the incapacity to adapt themselves to new situations. Results of the agent with XCSR learning indicate that we have accomplished the goal of creating an evolutionary rule system for the sales strategy in a TAC SCM agent.

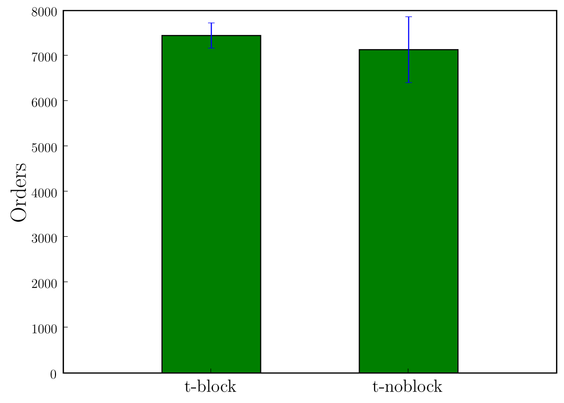
7.2 Experiment 2: Classifiers blocking

In this experiment we compare the performance of the TicTACtoe agent with and without the blocking classifier technique described in Section 6.5.1. In order to keep the agents as similar as possible, both versions of TicTACtoe used an exploitation rate of 70% and a population size of 1000 classifiers. The goal of this comparison was to prove the contribution of this technique. The agent that does not block classifiers allows erasing classifiers freely, ignoring if they are waiting for a reward. The results of this experiment will show if this simplification leads to information loss when the learning is in parallel.

7.2.1 Results



(a) Final Result



(b) Received Orders

Figure 3: Comparison of the performance of TicTACtoe with and without the blocking classifiers technique.

Figure 3(b) shows that `t-block` (L-TicTACToe with blocking) receives 312 more orders than `t-noblock`(L-TicTACtoe without blocking). This difference seems small and is not strong enough to make any assumptions on the agents' performance. However, Figure 3(a) shows that `t-block` duplicates the amount of money of `t-noblock` at the end of the simulation. This difference in the final balance is explained by the high penalties obtained by `t-noblock` (Table 3). The penalties indicate that this agent does not develop a set of appropriate rules to determine the final sale price for an RFQ. Moreover, `t-noblock` makes offers at very low prices to orders that have a high penalty set by the

		Interests (US$)	Penalties (US$)	Component costs (%)	Storage costs (%)
t-block	μ	156903	4791249	84.954	1.136
	σ	249618	6357427	2.573	0.210
t-noblock	μ	-34502	8084307	86.689	1.269
	σ	407400	7679561	6.843	0.262

Table 3: Performance of `t-block` and `t-noblock` according to the mean (μ) and variance (σ) of the interests, penalties, component costs and storage costs.

clients and are very difficult to produce because of the lack of the required components. When this agent offers products at low prices, it obtains a lot of orders, but most of them do not represent a profitable portion of the market considering its penalties.

In Table 3 we can observe that `t-noblock` gets negative interests from the bank, while the `t-block` gets positive interests. This implies that, on average, the agent that does not block classifiers incurs in debts, while the other agent maintains a positive balance in its bank account. This factor, in addition to the penalties, explains why in Figure 3(a) the agent `t-noblock` ends with less money than agent `t-block`.

According to the costs, we can notice that the agent that blocks classifiers spends less money on components and storage than the other agent. Considering that both agents have the same purchase strategy, we can affirm that the agent that blocks classifiers involves itself with orders that have lower production costs and higher profits.

It is also important to analyze the experience of the XCSR system in each agent. The experience is a measure of the classifier usage; it indicates how many times a classifier has been used.

In Figure 4 we can observe that the mean experience of `t-block`'s classifiers is higher than the mean experience of `t-noblock`. This pattern occurs because `t-noblock` allows erasing classifiers anytime based on incomplete and inaccurate information, i.e. rules that are still waiting for a reward that will determine if the classifier performed well. Consequently, classifiers that could lead to good decisions are erased before the reward arrives, and their knowledge is completely lost. The blocking classifier technique increments the global experience of the population when XCSR, improving the learning performance.

The results of this experiment shows that agents that uses the blocking classifiers technique inside their XCSR achieve a better performance, supporting its significance on parallel learning situations.

8. CONCLUSION

We designed and implemented a supply chain management agent for the TAC SCM problem. Our agent solves the production and the purchases sub-problems using static strategies, while it solves the sales sub-problem using a dynamic strategy. The purchase strategy is based on the acquisition of components considering production commitments for the next simulation days. The production strategy is based on manufacturing goods prioritizing orders according to their expected profits and due dates.

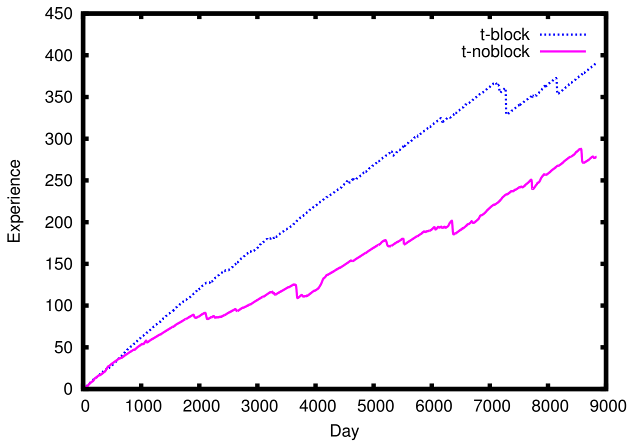


Figure 4: Mean experience of the XCSR population during 8800 days (40 simulations)

Our agent implements a dynamic sales strategy builds on Wilson's XCSR classifier systems. Through the XCSR mechanism we obtained a suitable set of rules for the TAC SCM sales problem. This set of rules showed to work better than the strategies used for control.

As our initial solution for the TAC SCM sales problem presents a parallel learning problem, we introduce a blocking classifier technique. We showed that the use of this technique improves learning performance under a parallel learning scenario. Moreover, we showed that the use of this technique yields to more experienced populations and a faster evolution of the classifier system.

9. REFERENCES

[1] M. Butz. Illigal java-xcs - lcs web, 2006.

[2] M. V. Butz and S. W. Wilson. An algorithmic description of XCS. *Lecture Notes in Computer Science*, 1996:253–??, 2001.

[3] J. Collins, R. Arunachalam, N. Sadeh, J. Eriksson, N. Finne, and S. Janson. The Supply Chain Management Game for the 2007 Trading Agent Competition, 2006.

[4] M. Franco and C. Gorrín. Diseño e implementación de un agente de corretaje en una cadena de suministros en un ambiente simulado, 2007.

[5] D. Loiacono. Evolving rules with XCSF: Analysis of generalization and performance. Tesi di Laurea, Politecnico di Milano, Facoltà di Ingegneria dell' Informazione, 2004.

[6] D. Pardoe and P. Stone. Bidding for customer orders in TAC SCM: A learning approach, June 03 2004.

[7] D. Pardoe and P. Stone. An autonomous agent for supply chain management. 2006.

[8] M. Stan, B. Stan, and A. M. Florea. A dynamic strategy agent for supply chain management. 2006.

[9] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[10] S. W. Wilson. Get real! XCS with continuous-valued inputs. *Lecture Notes in Computer Science*, 1813:209–222, 2000.