ArianTAC: A Supply Chain Management Agent

Mazda Ahmadi

Computer Science Department
University of Texas at Austin

Abstract. ArianTAC is a supply chain management agent for TAC-SCM environment. In this agent, three components of *production manager*, *supply manager* and*customer negotiator* is implemented separately. The *production manager* component simply manufactures computers with a greedy algorithm. *supply manager* component tries to maintain the components inventory more than a constant level and *customer negotiator* tries to bid for the most beneficial customer RFQs with the assumption that the inventory status would be unchanged.

1 Introduction

TAC-SCM is a competition of six supply chain agents in a simplified supply chain scenario[1]. ArianTAC is one the agents implemented for this scenario. One of the paradigms for creating agents in supply chain scenarios is agent based[2], which is adapted for ArianTAC.

One can assume three main sub-agents for an SCM Agent:

- Supply manager agent
- Production manager agent
- customer negotiator agent

These three agents work separately in ArianTAC architecture. The focus of the ArianTAC project was on *customer negotiator agent*, this agent works on a set rules and the agent has the ability to adapt the rules while the simulation progress.

After the introduction, there would be overview of ArianTAC approach and in section 3 the experimental results is presented and there is conclusion and future work section after that.

2 ArianTAC Approach

ArianTAC has three main components (or agents) for three different tasks. *Production manager* is a simple agent who does not hold any computer inventories, but produce computers when ever there are enough components to fill one of the customers RFQs. It prioritizes the RFQs to the favor of those with earlier deadlines. *Supply manager* requests for a big bundle of items at day 0 since the components are cheaper at that day. After that when ever there are less than 1500 (750 for less needed components) in inventory it requests for 100 (50) components [1]. If the total number of components in inventory and ordered is less than 1500 (750) it requests for immediate 100(50) items.

The focus of this project was the *customer negotiator*. This agent is implemented using a rule-based approach. For each PC, we have a paid price, which is the amount that agent has paid for its components to the suppliers and a reserve price which is what the customers have requested. For each customer RFQ, the agent give a discount (1%-100%) from its revenue ($reservePrice - paidPrice$).

This agent bid based on three parameters:

- **discount:** The percentage discount from revenue for responses to customers.
- **Free factory capacity:** The factory capacity in cycles for that specific day.
- **Success rate:** The assumed success rate which is: $\frac{number\ of\ ordered\ PCs}{number\ of\ offers}$

Free factory capacity is computed based on how much factory capacity has been left from the previous day + 2000 cycles of today. For example if the accepted RFQs for the previous day, takes 2300 cycles of factory time, free factory capacity for today is 1700 cycles.

I noticed that discount factor is related to two things: first number of customer RFQs and second (and more important) the other agents in economy. I tried to find the optimal discount factor for an overall and typical economy and then try to tune the discount factor during the game. Agent also changes the success rate during the game.

For accomplishing this, agent holds three tables. These three tables are stored in a file, so at the beginning of each game, the agent load these tables and at the end write the new tables in the same file. The tables are as follows:

- Table rfqDiscoundSuccRate shows for each total number of customer RFQs and a discount factor, what is a success rate for winning customer RFQs. At initial state, all success rates are equal to 1 and in the progress of simulation, the actual success rates are computed and averaged with the success rate from table.
- Table rfqDiscountCheck shows how many times have we computed the reward for a pair of (rfq number, discount factor). This table is used for keeping track of parts of the search space which have been explored enough. Obviously the initial values of this table is 0 and is added after computation of each pair of (rfq, discount).
- Table rfqDiscountReward shows what is the reward for a pair of (rfq number, discount). The initial value for each pair (rfq number, discount) which rfq is not equal to 50 is set to 0 and for $rfq = 50$ its set to 10. In this way, at the start point discount equal to 50 has the maximum revenue. Then during the game, agent keeps track of the revenue of each cycle and update the table when the final revenue for a pair is computed (which takes 12 days).

[1] 6 of the components are needed twice the other 4 components

The agent's progress is as follows: At each time step (i.e. a day), the agent knows the current rfq number from the customers, so looking through rfqDiscountReward table it can find the discount that maximize its revenue. Agent also needs to explore the search space, so it chooses the discount with ϵ difference from most profitable discount [2]. Agent chooses the the neighbor discount which has been less explored (i.e. has the less rfqDiscountCheck amount).

By looking at the rfqDiscountSucc table, the agent has the success rate also. It then computes the free factory capacity parameter. Then it bids for the appropriate amount of customer RFQs. It bids for the most profitable RFQs. Profit is computed as follows:

$$profit = [p * r + (1 - p) * penalty]/cycles$$

Where p is the probability of production, r is the revenue if the PC is sold at the reserve price. *penalty* is the amount of penalty that the customer has asked for and cycle is the cycles needed to build this PC. Note that the factory time required for building each type of PC varies, so it is needed to compute the profit per factory cycle.

The probability of p is computed with the assumption that the components inventory will be unchanged. So, if there is not enough component in the inventory, p is less than one and is computed with some hand tuned heuristics. Note that when the agent is processing $n\acute{t}h$ customer RFQ, it subtracts the components needed for previous customer RFQs ($total\ needed\ components\ *\ success\ rate$) and compute a another new profit and would not send a RFQ if the profit is less than 0.

further enhancements Since economies differ a lot in different aspects, success rates differ a lot between different economies, for example when playing against dummy agents, with rfq number bigger than 220 and discount = 0, the success rate is close to 1, but while playing at the class competitions, the success rate at the same situation was 0.7. In order to overcome this problem, I assumed that the rate of difference between success rates for different economies is constant. That is, if the success rate of economy one in situation A, is 1 but in economy B is 0.5, then if the success rate in economy A be 0.5 the success rate in economy B would be 0.25. This turned out to be mostly correct. The free factory capacity will cover the remaining error.

Although the revenue is also related to economy, but averaging through the similar economies makes it robust to some extent. For speeding the learning of rules, the revenue and success rate propagates to the neighbor cells. This makes the adaptation of the agent much faster.

A consideration for final binding is when ever the agent finds out, there is a big possibility that she can not use all his component in a regular way, it will give higher discounts.

[2] currently ϵ is set to 3%, it was set to 6% for some initial games for faster exploration.

3 Experimental Results

Figure 1 shows the evolved chosen discount after approximately 60 runs in different economies. As the figure shows the agent successfully learned that when the customer demand is high, the discount should be less than when the customer demand is low.

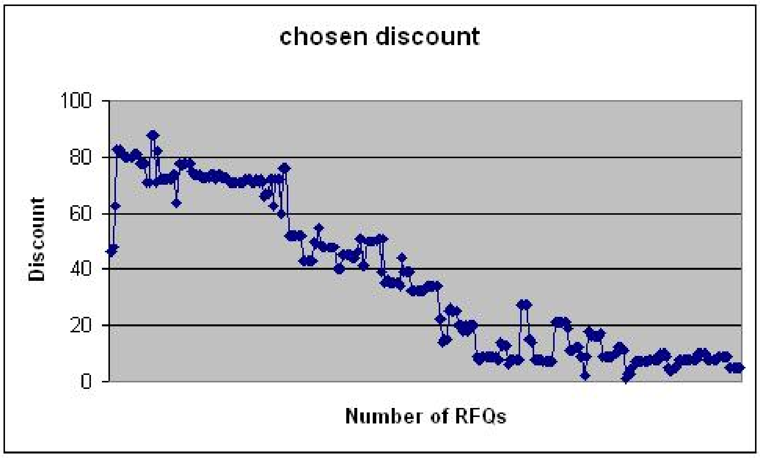

Fig. 1. Discount factor at different customer's demand (from 60 to 350 RFQs per day)

I tested the program over a period of 20 games [3]. The opponent agents where class agents who wanted to participate, although the agents in the economy changed from time to time, but the overall economy was constant to many extent. The initial values of rfqDiscountReward were 10 for discount factor equals to 50 and 0 otherwise. In this case, the first discount factor selected would be $50 - 3 = 47\%$. Checking the rfqDiscountReward function interestingly showed that the discount for the pairs with higher number of RFQs and lower discounts increased while it decreased for the cases with higher RFQs and lower discounts. This is of course as we expected.

Showing the power of ArianTAC to adapt to other economies, we tested the same ArianTAC learned after 20 games (which mentioned before) against dummies which is a totally different economy. The average of first five games showed the increase of profit from 30M$ to 40M$. [4]

I had 6 copies of ArianTAC play against each other. The economy was pretty robust and the agents got approximate same share of the overall profit. Their

[3] I shall mention at the time of test, the agent was buggy.

[4] for the both of the mentioned experimental results, the log files are deleted, and this is what I remember, but its error is less than 4M$ for sure

average profit in a high demand scenario was around 26M/$andaround1M/$ in a low demand scenario which constructs a good economy.

4 Conclusion and Future Works

ArianTAC is a simple supply chain agent with the big assumption that the three components work with out communication with each other. But, the agent seems to be successful if it had a good supply manager part. It's strong point is satisfactory coping with customer demand fluctuation (though it doesn't happen much in this scenario).

The main enhancement that should be done is in adapting the rfqDiscountReward table from one economy to another economy. It can probably be done by propagating the difference with the actual new data and the previous data in a more proper way.

Other enhancement can be reached by a better coordination between the supply manager agent and the customer negotiator agent. For having a good supply chain agent, all parts of the agent should be strong, but in ArianTAC just the customer negotiator seem to be strong, and other two parts should be improved for a successful participation in competitions.

References

1. The TAC team, TAC Supply Chain Management Specification. http:://www.sics.se/tac
2. Fox, M.S., Barbuceanu, M., and Teigen, R. Agent-Oriented Supply-Chain Management. International Journal of Flexible Manufacturing Systems, Vol. 12, No. 2/3, pp. 165-188, 2000.