ANAC 2020 BIUDODY agent report

When started working on ANAC we firstly tried to see which of the built-in agent is having the best performance and found that the Decentralized Agent is the winner in this case. Therefore, we focused on improving the component that the Decentralized agent is using and after multiple version and tests, we managed to win against this agent in 75% of the cases. We did win against all other built-in agent in much higher percentages and tried to test our code with other components but decided, based on the result, to improve the components that the Decentralized Agent is using.

The Agent is using an enhanced version of the MeanERPStrategy predicts the execution rate per buyer or seller, meaning for every buyer or seller at the world we will maintain a structure that is holding his average execution rate and quantity. These parameters will then go to our controller as the mean of all expected quantities and in this way if negotiation were made with one agent it will affect accordingly and not fairly distributed. This also gives our agent to maintain different parameters per buy or sell.
We also used gradient descend in order to learn the best parameter for the production_cost_factor argument which is needed in order to decide what is the acceptable unit price.

We didn't implement a collusive version of our Agent and therefore it doesn’t play well in this version of the tournament.

Simultaneous negotiations coordination.

At Controller level: the final agent acts according to DecentralizingAgent's coordination strategy, which is performing independent negotiations with AspirationNegotiator and declining negotiations when target quantity is reached. It is worth mentioning that we did try some novel approaches: we implemented a step-sync controller that waits before accepting a negotiation and after a while selects the best outcomes from the waiting list. Another controller we tried estimated a "fair price" with respect to previous agreements and declined new offers that were too far from the fair price. Unfortunately, none of our attempts shoed any improvement, thus they were not used in the final agent.

Utility function.

The agent's utility function calculates an initial utility value *u*, a trustworthiness factor *t*, and then the final utility is set to:

$$u - |u * (1 - t)|$$

The initial utility is calculated by the DecentralizingAgent's utility function, which is the LinearUtilityFunction with (1, 1, 10) parameters for seller, and (1, -1, -10) parameters for buyer.

In a negotiation with agent A, the trustworthiness factor represents how much we trust agent A, in percentage. It is calculated by:

$$0.2 * (1 - \text{breach level}) + 0.8 * (1 - \text{breach probability})$$

When *breach level* and *breach probability* are taken from the last published financial report of agent A.

Production Strategy

At every step an agent signs on several buy and sell agreements. Our goal was to create a strategy that minimizes the difference between buy and sell quantities. To achieve that we first based our Agent on the Demand-Driven-Production-Strategy. Doing that promised that the agent will always have minimal quantity it needs for the sell agreements that were already signed. Secondly, we created a model that tries to predict if there is a need to increase production for future deals.

To create a production-classifier we created hundreds of simulations and for each of one, we collected information from the environment throughout the whole game. The final classifier was trained as a binary classifier. Given the information collected from the last 3 steps it predicts if there is a need to increase production with respect to the Demand-driven-Strategy. Since the feature space was limited, we used a linear SVM model. We split the simulation-data to train and test with ration of 80/20 respectively and got 82% accuracy over the test set.

To apply the model an agent loads the model at initialization time. At each step the Agent collects information from the environment, commits production according to the Demand-Driven-Strategy and then it produces K more items if the production-classifier returns a positive answer.