ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Computer Science Master, Course: Intelligent Agents
Students: Ivan Gavrilović, Boris Perović

Assignment 5

Programming: Implementing decentralized Coordination

A brief solution description:

The solution was developed in IntelliJ IDEA 12 IDE, using JDK 7 update 40 and the provided LogistPlatform framework. Git was used as collaboration tool and a version control system. Provided template (*auction.zip*) was used as a base for the project and agent auction-random given in it was used for comparison purposes as one of the dummy agents.

As we are supposed to operate a multi-vehicle company and compete against opponents in auctions in this assignment, the implementation consisted of two steps.

Planning:

The first step was to implement the planning algorithm using which we can calculate our marginal cost and from whose output we can make a plan that we return to the auction house in the end. Since the algorithm from the previous assignment (centralized coordination) suited these needs, we reused it in this solution. Also since tasks are added on by one to the plan in each of the rounds, in order not to recalculate the whole SLS algorithm from start each time, the best solution is preserved and used as a starting point for future calculations.

After this part is done, one can participate in an auction by bidding the marginal cost (or additionally improving it by making some sort of a formula, including marginal cost). But taking into account only calculations about its own company and plan will probably not lead to a great performance against other competitors. That's why there is a second step, which is bidding strategy development / refinement.

Bidding:

In this step we refined our strategy using information available to us through the system. This information is previous opponent bids given to us by `auctionResult()`, which we store and calculate several different things using them. Probability distribution provided by the system was not used, since there was a lot of discussion in class about feasibility and value of using it in implementation, with no conclusive remarks. We instead focused on perfecting opponent bid estimation. Since this is a closed-bid auction, in order to bid better and be more competitive, we need to adapt to opponent's previous bidding behaviour and estimate his future bids. If the situation allows, we bid just below opponent's predicted bid, therefore potentially winning the bid as profitably as we can.

The strategy also relies on the assumption that it is better to bid less in the beginning, in order to acquire more tasks, be flexible in the future. This can in a sense be looked at as "warming-up" the company.

Estimation on the opponent's bid is done in the following way. We instantiate three different SLS solutions that we will use for the opponent. Each of these SLSs is associated with the different domain of values. Reason for this is that we do not know the exact parameters for the opponent's company, and we can only guess. In the beginning, each of the domain values associated with SLS contains only vehicles belonging to a company, as tasks are added one by one. Vehicles belonging to a domain have some parameters changed compared to the vehicles from our company, other than that, they are completely the same. Their `homeCity` is randomly chosen and their capacity is between 80% and 120% of original capacity of the "base" vehicle. Purpose of this is to try to have three different setups for the opponent, and to use the arithmetic mean of the solution costs in order to have the best estimation of the opponent's marginal cost for the task. This method is very useful when the number of tasks is smaller, as the position of the vehicles may yield quite different solutions. Using the randomization of the vehicle parameters for all three SLSs gave very good results. Because of the limited time to give the bid, time for calculation is split evenly among calculation for our own marginal cost, and for calculation of the opponent's bid estimation. For finding the optimal opponent's solution, we use the mechanism of tracking the latest solution found, and start exploring its neighbourhood, as it is better than to start from random initialization.

We keep track of all accepted bids for the opponent and the tasks he won. After finding the cost for the opponent, we multiply it by *ratio* (it will be explained soon) and we deduct the total amount of his bids, thus leaving us with his approximate marginal cost. At that point we set our bid to be 85% of this calculated marginal value. If that price is below certain limit, we correct the price to the lowest possible viable value. This way we are protecting ourselves from too making harmful bids. The bottom limit is calculated as `myMarginalPrice * canAllow`, which is a factor between 0.45 and 1.

Let's see how `ratio` is calculated. Being able to get the opponent's bids is really important as we can see how much our estimation needs to be adjusted in order to guess more precisely. For every task that is auctioned, we keep track of the ratio between actual cost paid by competitor and our estimated price (in round `i`, this is er_i). This history of ratios is used to calculate the new one, for the next round, where we use it as a correction. We calculate it as arithmetic mean of previous ratio and the value that is obtained as a function of the er_i, $1 <= i <= currentRound$, where newer er_i has a greater impact (see `auctionResult()` method for details). This value should be between 0.7 and 2.5.

Another important factor is `canAllow`. This, as mentioned before, is the lowest value that company allows for the task. For every task won we increase it by 0.1, as price per tasks will change less and less as we have more tasks. We reduce this value by 0.05 every time we lose a task, in order to allow lower bids. Reason for this is that winning larger number of tasks usually brings profit, even if each of the tasks during bidding does not.

Choosing the correct bound values for the `canAllow` and `ratio` is very important. The values that were previously presented allowed our agent to win against multiple dummy agents, and every match was won with a large difference.

Obtained results:

The implementations against which we ran our implementation are: agent that always bids 0, agent that randomly chooses value between -200 and 5000 for the task, agent that uses SLS to calculate his marginal cost and always bids with 5% commission on it and `AgentTemplate` that was already provided in the source skeleton. As we can see in *Table 1* our implementation won all the matches.

Not surprisingly, agent bidding always 0 finished last. Random agent had score 6/2 which is greatly depends on the distribution of the bids. Although agent that adds the 5% commission uses SLS for the calculation of its marginal cost, it did not manage to beat the Template agent whose marginal cost is typically much greater. This just emphasizes the fact how important is to use the opponent's bids in order to calculate your own.

Name of the agent	**Win-Draw-Lose**
OurAgent	8-0-0
Random	6-0-2
Template	4-0-4
Commission	2-0-6
Zero	0-0-8

Table 1 – Results of tournament

It is valuable to see the profit won in the games between our agent and competitors. Using the good estimation of competitor's cost, `ratio` and `canAllow` factors, profit for our agent is much greater. For all of the matches `OurAgent` received profit of 101806, while its opponents received -30055 in total.

In the *Table 2* we can see how the factors are adjusting to the opponent's bids. If we are bidding too high, ratio decreases, thus our bid gets lower, but is above minimum allowed value. This table illustrates quite well the idea behind this adjustable factors and how they change.

roundNum	**canAllow**	**ratio**	**myCost**	**estimOpponent**	**bid**	**oppBid**
1	0.45	1.0	2191	2191	1863	2905
2	0.55	1.16	2710	2108	1793	3443
3	0.65	1.43	917	3178	2703	2929
4	0.75	1.51	-3527	2743	2332	3808
5	0.85	1.62	-5658	1867	1588	604
6	0.80	1.50	-5652	4695	3992	2447
…						
19	0.75	1.11	-3528	2627	2234	6236
20	0.85	1.14	-5762	3443	2927	6365

Table 2 – Execution of algorithm round by round, with details on the change of values