1 Introduction

In this lab, you will be designing agent strategies for simultaneous auctions, and implementing them in our TradingPlatform so they can compete against your classmates' agents. In particular, your agents will be bidding in the **GSVM-9 Simultaneous Auction**, in both first-price and second-price varieties.

This lab is completely open ended. The previous two labs were intended to provide you with insights into simultaneous auction strategy, which you can draw on today for inspiration, when you apply your knowledge to tackle the bidding problem in a complicated auction.

2 Setup

You can find and download the stencil code for Lab 7 from the course website. Once everything is set up correctly for this lab, you should have a project with the following Java files, all under `src/main/java` in the package `brown.user.agent.lab07`:

- `IPricePrediction.java`
- `MyPricePrediction.java`
- `MyGSVMFirstPriceAgent.java`
- `MyGSVMSecondPriceAgent.java`
- `CompetitionRunner.java`

3 The GSVM-9 Auction

The **Global Synergy Value Model** for nine goods **GSVM-9**[1] auction was designed to study the bidding behavior of human subjects in laboratory experiments that mimic wireless spectrum auctions.

In GSVM-9, there are 4 bidders and 9 goods, labelled **A** through **I**. Among the 4 bidders, 1 is a *national bidder* and 3 are *regional bidders*. The goods are designated such that goods **A** through **F** are national goods, while goods **G** through **I** are regional goods. The national bidder is interested in and eligible to bid on only the national goods, while each regional bidder is interested in and eligible to bid on 4 of the 6 national goods, plus one regional good (each regional bidder has their own region).

The valuation distributions for each bidder with respect to each good are detailed in the table below. Blank entries in the table indicate that the respective agent is ineligible to place a bid for the respective good.

	Goods								
	National						**Regional**		
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
National Bidder	15	15	30	30	15	15			
Regional Bidder 1	20	20	40	40			20		
Regional Bidder 2			40	40	20	20		20	
Regional Bidder 3	20	20			20	20			20

- **Individual good's valuations are drawn uniformly between 0 and the number in the table.**
- **Each additional good increases the value of the bundle by 20% (*global complement*).**
- **The national bidder may bid on (and win) up to 6 goods.**

[1] Martin Bichler, Zhen Hao, and Gediminas Adomavicius. Coalition-based pricing in ascending combinatorial auctions. *Information Systems Research*. 28(1):159–179, 2017.

- **Regional bidders may only bid on (and win) up to 3 goods, despite being eligible for 5.**

The global complement (i.e., the 20% bonus) is *additive* not *multiplicative*. Thus, if you win a bundle of n goods, say $g_1, g_2, \ldots, g_n$, then your bundle's value would be

$$v(\{g_1, g_2, \ldots, g_n\}) = [1 + 0.2(n-1)] \left(\sum_{i=1}^{n} v(g_i) \right) \quad .$$

4 Your Task

This lab comprises two competitions: a first-price and a second-price GSVM-9 Auction. In order to do compete, you must create both a national-bidder strategy and a regional-bidder strategy, for both auctions.

Each competition will consist of 100 simulations against your classmates' agents. During each simulation, you will be randomly matched up with opponents, at which point roles are randomly assigned. In particular, you will always have an equal probability of playing each role. Thus, over 100 rounds, you should play each of the four roles on average 25 times. (More simulations would of course provide a more fair indication of the strength of your overall strategy.)

The TAs will run two sets of two competitions. Each set will consist of a first-price and a second-price GSVM auction. While you are required to compete in all competitions, how you choose to allocate your time among them is your choice. If you feel your intuitive understanding of second-price auctions is stronger than that of first-price, you might choose to focus on building an agent for the former, to increases your chances of winning; or to focus on the latter, to improve your understanding. Again, the choice is yours.

5 Your Strategy

GSVM-9 is a very rich auction design. As you brainstorm about what make a bidding strategy effective in this auction, you might reflect on the following points:

- Because of the global complement, winning multiple goods is very significant; should an agent risk bidding over their individual good valuations to try to secure a multi-good bundle? *How can marginal values be used to navigate this situation?*
- The regional bidders are restricted to bidding on only 3 goods; if their valuations for the national goods exceeds those of their regional good, should they forgo the guaranteed win of the regional good to try to grab the more highly-coveted national good? *How should the regional bidders balance the different levels of competition for the different types of goods?*
- *Should your strategy be tailored somehow to treat national and regional goods differently?*
- In second-price auctions, where an agent does not pay its bid, bidding marginal values that exceed a good's predicted price might be a reasonable strategy. But this strategy seems much less compelling in first-price auctions, where winning agents pay their bids. *How should your agent's bidding strategy differ in first- vs. second-price auctions?*

6 Your Agents

In both `MyFirstPriceGSVM9Agent.java` and `MySecondPriceGSVM9Agent.java`, you are responsible for filling in the following methods:

- `IBidVector nationalBid(Set<IItem> goods)` represents your agent's bidding strategy when it is the national bidder. This method should return an `IBidVector`, mapping each good to your agent's bid for that good. The only rules here are that your bids must be non-negative, and that this method must terminate within a half-second. Note that `goods` will only contain the national goods (i.e., the goods your agent is eligible to bid on), so you need not worry about removing ineligible goods.
- `IBidVector regionalBid(Set<IItem> goods)` represents your agent's bidding strategy as a regional bidder. This method should also return an `IBidVector` of non-negative values within a half-second. Again `goods` will only contain the goods which your agent is eligible to bid on. However, as a regional bidder, your agent is only allowed to bid on 3 of those goods, so your resulting `IBidVector` can contain *at most three bids*. If your code violates this constraint, your agent's bid will be rejected, and a message informing you of this outcome will be printed to the console.

To create an `IBidVector`, use `IBidVector bids = new BidVector();`.

`IBidVector` then provides the following methods:

- `setBid` sets the bid for a specific good.
- `getBid` gets the bid for a specific good.
- `contains` checks if there exists a bid for the specified good.
- `size` returns the number of bids in the bid vector.
- `remove` removes your bid for a specified good.

6.1 Tools and Support

We have provided several tools for you to use when implementing your strategy:

- The method `this.valuation()` gets a variable storing your agent's valuation for any bundle of goods (including singleton bundles).
  You can use this variable by calling its `getValuation` method on an `ICart`, which is the TradingPlatform's representation of a bundle of goods. To get the valuation of a single good, simply create a single-good `ICart`.
  Recall that to get the valuation of a bundle of goods, you should use the following pattern:

```
ICart cart = new Cart();
for (IItem good : bundle) {
    cart.addToCart(good);
}
double v = this.valuation().getValuation(cart);
```

- If you would like to use a strategy that is similar to the one you developed in the previous lab, we have provided a way for you to maintain a price prediction. The instance variable `this.prediction` is automatically updated with the prices of each good at the end of each auction, and is written to disk. It is also loaded by your agent at the beginning of each auction. It is an instance of the class `MyPricePrediction`. If you wish to use it, you will need to navigate to `MyPricePrediction.java` and implement the `sample` and `addRecords` methods.
  Note: While this price prediction may be instantiated as a histogram price distribution, as in the previous two labs, feel free to represent price predictions in any way you like. For example, you might try to model some dependencies in price predictions across goods, rather than assume independence.
  Moreover, if you think it would be worthwhile to learn some quantity other than prices (such as opponents' bid distributions), feel free to mimic the code by which predictions are updated, saved, and loaded each round (see Section 6.2).
- To train your agent before bidding, simply run the `MyGSVM[First/Second]PriceAgent` in Eclipse, which will launch a 100-round simulation, with exactly the same details as the competition, except that

it will be against 3 identical copies of your agent. This will allow your agent to learn a price prediction (or whatever else you might train it to learn).
Note: Even if you'are not using a trainable/learning agent, you can still use this functionality to simulate your agent and test out different strategies.
Another Note: If you would like to increase the length of these simulations, you may edit the configs in `src/main/resources/input_configs`. In particular, look for the `numTotalRuns` variable.

- Finally, if you think that the ideas we developed in the previous two labs (e.g., marginal values, LocalBid, self-confirming price predictions, etc.) will be useful, you are free to reuse code from previous labs.

6.2 Optional Methods

If you wish to implement a training-based strategy like SCPP that begins with a simulation phase, we have provided you with a basic framework for doing so. It consists of the following methods:

- `saveInfo` is called after each simulation. It writes your agent's price prediction, or any other learned variable(s), to disk, so that it may be loaded by the other agents in the training phase. In the stencil code, it currently writes `this.prediction` to disk; feel free to change this method, or to comment it out if you are not using a training phase.
- `loadInfo` is called before each bid request. It loads your agent's price prediction, or any other learned variable(s), from disk, so that your agent has access to the most up-to-date information. It is meant to be called by all agents in the training phase. In the stencil code, it currently reads `this.prediction` from disk; feel free to change this method, or to comment it out if you are not using a training phase.
- `update` is called after each round of the simulation (i.e., after each auction). It is where you can implement a learning rule, which your agent can use to update its price prediction, and any other learned variable(s). This method takes as input two parameters: `IMarketPublicState`, which contains information about the bids and outcomes of the previous round of the simulation; and `int simulationCount` which tells you the current simulation round, so that you can implement periodic updates (as in SCPP) if you so choose. The current stencil code inserts the price of each good from the `SimulationReport` into `this.prediction`; feel free to change this method, or to comment it out if you are not using a training phase. The stencil code also includes calls to methods of the `IMarketPublicState` to obtain the lists of bids and outcomes from the previous round. Your `update` method can/should build on these calls. Since `update` is called every round, it's an ideal place to call `this.saveInfo()` to write your predictions to disk.