# Final Project: Spectrum Auction

## 1 Introduction

This final project option is a Spectrum Auction, which builds on the work we did in Labs 4, 5, and 6.

## 2 Game Description

The model of the world, goods, bidders, and valuations is simply a larger version of the setup in Lab 6, which is the **Global Synergy Value Model** (**GSVM**). In particular, whereas that version, GSVM-9, had 9 goods and 5 bidders, this version, GSVM-18, has 18 goods and 7 bidders.

Additionally, you may recall that the labs were conducted as simultaneous second-price auctions. In the final project, we will instead be running **simultaneous ascending auctions** in an auction format known as **Simultaneous Multiple-Round Auctions** (**SMRA**).

### 2.1 The GSVM-18 Model

GSVM-18[1] is a small model of a spectrum auction. There are 18 goods, labelled A through R. Twelve of them are characterized as national goods, and the other six are characterized as regional. In addition, there is 1 national bidder and 6 regional bidders. The following rules and restrictions are in place:

- The bidders are interested in the various goods as indicated by the following diagram.

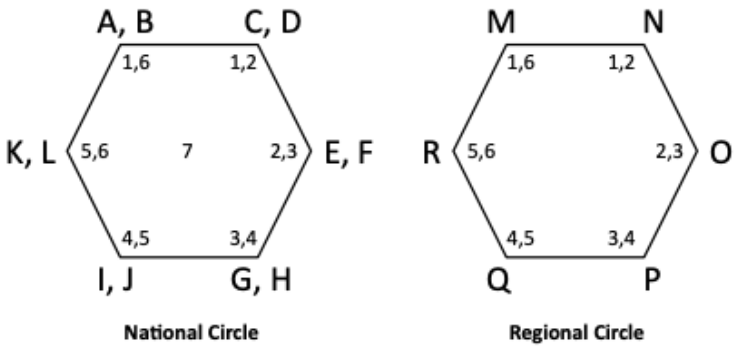



**Figure 8** **Competition structure of the Global-SVM. Regional bidders 1-6 are interested in four items from the national circle and two items from the regional circle. National bidder 7 is interested in all twelve items from the national circle (Goeree and Holt 2010).**

- Bidders may only place bids on goods they are interested in. For example, regional bidder 5 can only bid on goods {K, L, I, J, R, Q}.
- While the national bidder may bid on all twelve goods of interest, each regional bidder may only bid on up to four goods, despite being interested in six.

[1] Jacob K. Goeree and Charles A. Holt. Hierarchical package bidding: A paper & pencil combinatorial auction. *Games and Economic Behavior*. 70(1):146–169.

- There are **global complements**—the valuation of any bundle increases by an additional 20% with the addition of another good.

Valuations are drawn from the following distributions:

- **For the national bidder**:
  - U[0, 10] for items A–D and I–L
  - U[0, 20] for items E–H
- **For regional bidders**:
  - U[0, 20] for items A–D, I–L, and M–R
  - U[0, 40] for items E–H

## 2.2 Simultaneous Multiple-Round Auctions (SMRA)

The mechanism your agent will bid in for the project is a variant of Simultaneous Multi-Round Auction (SMRA). SMRA are simultaneous ascending auctions, which solicit bids in all the auctions from all the agents synchronously (i.e., in rounds). Furthermore, all the auctions end simultaneously, when there is no further bidding in any of them.

The ascending auction that will populate our SMRA mechanism is a variant of eBay's auction design. In this auction, the price in each auction in each round is given by the second-highest bid, floored by a fixed price increment $\epsilon$. Between rounds, the prices across all auctions are broadcast to all agents, as are the identities of the tentative winners.

The eBay auction rule allows for the possibility of **jump bidding**. Jump bidding means that prices need not ascend uniformly. Instead, they can jump by multiple increments, as dictated by the second-highest bid. Jump bidding gives agents an opportunity to strategize, as they can "stake out their territory" by bidding high in some auctions.

Finally, SMRA also generally invokes an activity rule. Activity rules in ascending auctions are designed to encourage sincere bidding. Likewise, they tend to discourage exiting the auction only to re-enter later. Our particular choice of activity rule—revealed preference—is described in Section 2.3. Before delving into those details, we present the high-level auction procedure for our version of SMRA:

**Algorithm 1** Run the SMRA Spectrum Auction

```
Set the price of all goods to 0
Begin with an empty tentative allocation

do
    Broadcast all current prices and each bidder's tentative allocation
    Get bid vectors from each bidder, one bid per good
    Prune bids that do not exceed current prices by the bid increment ϵ
    Prune bids that violate the revealed preference activity rule
    Call the remaining bids valid
    if the set of valid bids is non-empty then
        for each good do
            Tentatively allocate to a highest bidder
            if there are multiple bidders then
                Update the price to the second-highest bid
            else if the good changed hands since the last round then
                Update the price to the current price plus ϵ
            end if
        end for
    end if
while the set of valid bids is non-empty
```

The final allocation and final payments are set to be the outcome that maximized revenue, maximizing over all rounds.

For example, if the current price in one of the auctions is 6 and $\epsilon = 2$, then all bidders must bid at least 8 for the good for their bids to be considered valid. If nobody bids at least 8 for the good, the price remains at 6. If multiple bidders bid above 8, the good will be tentatively allocated to a highest bidder at the price of the second-highest (valid) bid. If only one bidder bids above 8, and if that bidder is *not* currently winning the good, they will be tentatively allocated the good at price 8. If the sole bidder above 8 *is* currently winning the good, neither the current price nor the current allocation will change.

This final allocation and payment rule is inspired by Ausubel *et al.*'s clock-proxy auction.[2] The bids in the final round may not be the ones that yield the highest revenue (or welfare), because some bidders may stop bidding on some bundles if they become too expensive. In such cases, it is useful to give the auctioneer the freedom to revert back to an earlier allocation and payments to maximize revenue (or welfare).

## 2.3 The Revealed Preference Rule

The **revealed preference rule** is an activity rule which we will adopt to determine the validity of a bid in our SMRA auction. Like any activity rule, its purpose is to restrain bidders to the space of "reasonable" strategies; in particular, this rule is designed to discourage bidders from re-entering the ascending market for a good after previously exiting it.

The revealed preference rule is defined as follows: If a bidder switches its preferred demand set from bundle $S$ to bundle $T$, then it must have been that since the time when $S$ was preferred, the price of $T$ increased by less than the price of $S$. Alternatively, if the price of $T$ increased by more than the price of $S$, then a bidder cannot switch from preferring $S$ to preferring $T$.

To present the revealed preference rule formally, we introduce the following notation: Let $m$ be the number of goods (i.e., ascending auctions). Let $s < t$ be two rounds in the auction. Let $\boldsymbol{p}^s$ and $\boldsymbol{p}^t$ be vectors in $\mathbb{R}^m$

[2] Lawrence M. Ausubel, Peter Cramton, and Paul Milgrom. The Clock-Proxy Auction: A Practical Combinatorial Auction Design. In *Combinatorial Auctions*. Peter Cramton, Yoav Shoham, and Richard Steinberg (editors). MIT Press, Chapter 5, 115–138, 2006.

describing good prices at rounds $s$ and $t$, respectively. Let $\boldsymbol{x}^s$ and $\boldsymbol{x}^t$ be vectors in $\{0,1\}^m$ describing the bundles a bidder demands at rounds $s$ and $t$, respectively (i.e., all goods for which it submits a valid bid).

If a bidder is bidding sincerely, then at round $s$, $\boldsymbol{x}^s$ is the set of goods that is utility maximizing:

$$v(\boldsymbol{x}^s) - \boldsymbol{p}^s \cdot \boldsymbol{x}^s \geq v(\boldsymbol{x}^t) - \boldsymbol{p}^s \cdot \boldsymbol{x}^t.$$

Similarly, at round $t$, $\boldsymbol{x}^t$ is the set of goods that is utility maximizing:

$$v(\boldsymbol{x}^t) - \boldsymbol{p}^t \cdot \boldsymbol{x}^t \geq v(\boldsymbol{x}^s) - \boldsymbol{p}^t \cdot \boldsymbol{x}^s.$$

Combining these inequalities yields the revealed preference rule:

$$(\boldsymbol{p}^t - \boldsymbol{p}^s) \cdot (\boldsymbol{x}^t - \boldsymbol{x}^s) \leq 0.$$

This rule ensures that sincere bidders will not take a break from bidding part way through the auction, because if they do (i.e., if they report the empty set as their favorite bundle), and then if prices go up on some goods, then they can no longer bid on bundles that contain those goods. In particular, this rule discourages bidders from sitting out the very first round.

### 2.4 Game Specifics

- Our price increment, $\epsilon = 2.5$.
- To make sure our simulated auctions don't run indefinitely, we will cap their number of rounds. If an auction reaches a predetermined final round without having terminated, it will end abruptly at that point. This number will be drawn from an exponential distribution, and is guaranteed to be at least 30. Although this distribution is common knowledge, the draw will not be revealed to your agent, as knowing this number would facilitate sniping.[3]

## 3 Strategy and Considerations

This is a very complex game and requires several strategic considerations beyond what was in play for Labs 4, 5, and 6.

First, due to the combinatorial nature of this auction, there are an exponential number of possible bundles on which your agent can bid. This number is especially large for the national bidder, so any strategy that involves enumerating them all will not be viable. How will your agent determine which bundle to bid on, and at what prices, in an efficient manner?

Second, how high above the price thresholds will your agent bid? Due to the second-price nature of these auctions, your agent could submit high bids to secure a tentative allocation, without risking having to pay too high a price? But would doing so potentially reveal too much information to other bidders? In other words, should your agent bid high right from the start, revealing its interests to other bidders—a form of cooperative behavior—or should it lay low at first, gauging its opponents' strategies—how cooperative are they?—before bidding too aggressively: i.e., showing its cards?

Third, how will your agent's strategy differ when it is the national bidder, versus a regional bidder? There is more competition for national goods; will this affect your agent's bidding strategy? Will you incorporate the valuation distributions (which are public knowledge) to try to predict the bids of opposing agents, and ultimately the final auction prices?

[3] `urbandictionary.com` defines sniping as the act of bidding on an item on eBay literally seconds before the auction ends. This prevents anyone from outbidding you while also ensuring that you don't bump the price up too much.

In addition to these considerations, a successful agent strategy for this game will comprise many others. We suggest building a theoretical model of the game (consider starting your writeup!) before attempting to implement an agent strategy. This model will help you envision even more of the many important factors that your agent's strategy should try to take into account.

# 4 Code: Installation, Testing, and Submission

Please read the following instructions carefully, as it is essential that your code runs successfully with our game server, and that your handin is correctly formatted.

## 4.1 Installation

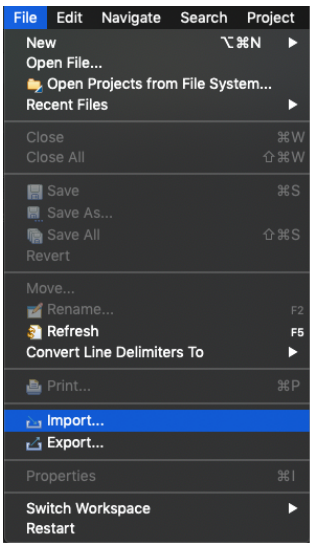
Download the stencil code from the course website. Just as in Labs 7 and 8, the stencil code is an entire Java project, complete with a package structure and a file called `pom.xml`, along with a few python scripts.

We will be using the Apache Maven build system for this project. Put simply, Maven allows us to export our Java projects into runnable JAR files in a standardized fashion (as defined in `pom.xml`, which acts as a configuration file for Maven).

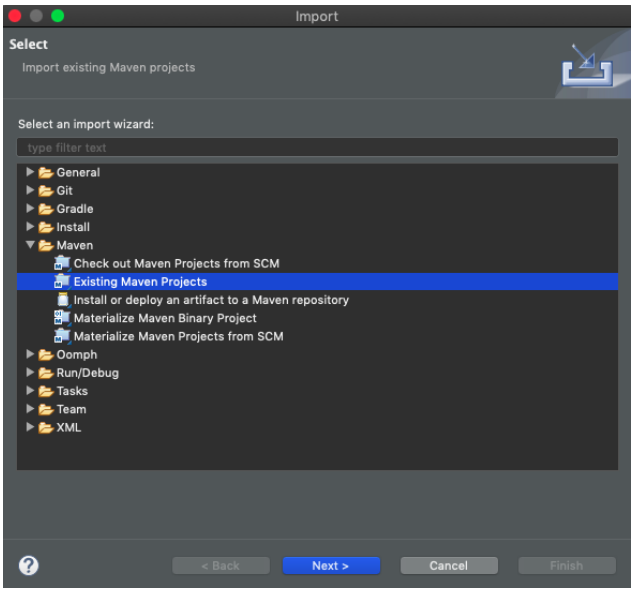
If you do not already have Maven installed, please download and install it. If you are working on a department machine or via FastX, you will already have Maven installed, which you can verify by running `mvn -version`.

If you have not installed Eclipse on your machine, you should download that as well.

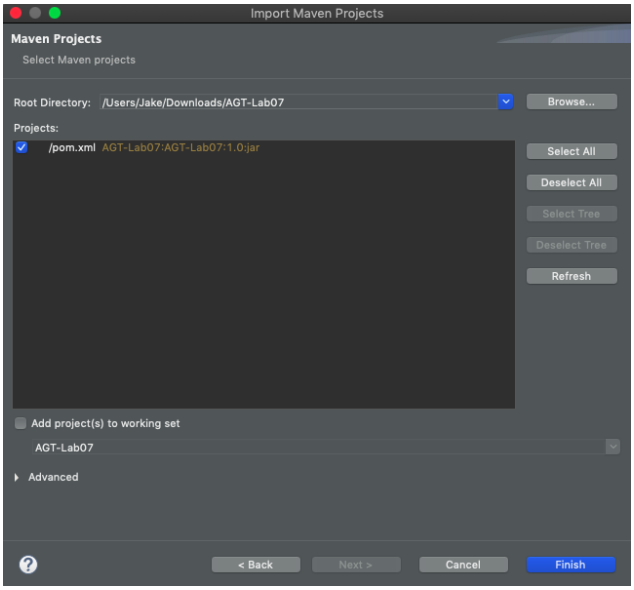
Once you have installed Maven and unzipped the stencil code, open Eclipse and select **File** → **Import**.



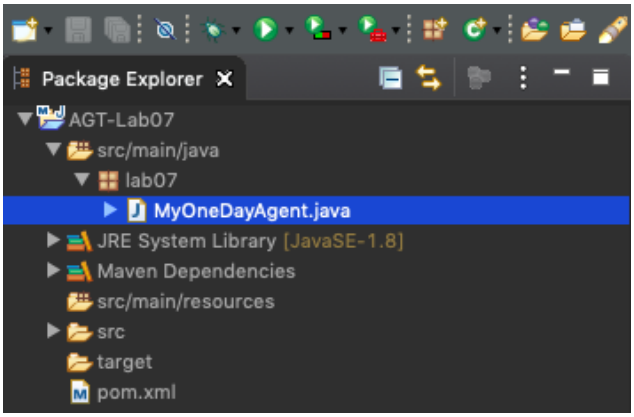
Then, select **Maven** → **Existing Maven Project**.




Finally, navigate to, and select, the stencil project and make sure `pom.xml` is selected. Select **Finish**.



This will create an Eclipse project for this lab. Your task resides in the file `MySpectrumAuctionAgent.java` in the package `agent` under `src/main/java`. Feel free to create and use any other packages, classes, etc. within the project, as long as you do not change the location or name of `MySpectrumAuctionAgent.java`.




## 4.2 Local Testing (and Training)

If you would like to test out your agent against other agents, run your `MySpectrumAuctionAgent` file in Eclipse. The main method instantiates 10 agents, including your own, and runs an offline simulation (in order to not have to deal with server delays and wait times). This simulation consists of 42 iterations of the Spectrum Auctions game.

As with any programming project, you should test your programs extensively. In addition to the usual unit tests, etc., you should play test games against other agents. You can test against 9 copies of your own agent, 9 copies of our Tier 1 TA Agent (see Section 6), or any combination thereof. You can also create your own dummy agents to test your own agent against; just create a class extending `AbsSpectrumAuctionAgent`. To set the agents to test against, you should edit the agent types in `src/main/resources/offline_test_config.json`.

If you design a strategy that involves training, you can submit a pre-trained agent. Then, to use a pre-trained agent that reads its input from a file, please place the file in `src/main/resources` and read it using `getResourceAsStream()`; see this guide. By including your file in the resource folder, you will ensure that it is included in the executable JAR we use to run your submission; file-reading is thus independent of the specifics of the filesystem.

## 4.3 Submission

You will be submitting your code via the course handin script.

First, if you are not already working on a department machine or via FastX, you will need to transfer your files to the department filesystem. We have provided a utility to do this upload for you. You are free to transfer the files any way you'd like—this utility is just an attempt to simply your life.[4]

To use this utility, from your project's root directory, run `python3 upload.py`.[5] It will transfer your files to `/course/cs1951k/student/<cslogin>/SpectrumAuction`. (We have created student directories for you so that you don't need to use up tons of space in your personal filesystem.)

Next, SSH to Brown, and navigate to your project's root directory (if you used the upload script, `/course/cs1951k/student/<cslogin>/SpectrumAuction`).

[4] Well, not really. Just this very very small part of it.
[5] To run this script, you will need Python 3. You may also need to install it, as well as the `pysftp` library, via `pip`.

To make sure that your agent will work as intended when being run the way we run handins, we have provided a script that launches the server, along with your agent and 9 opponent bots. You should run this script to make sure your agent connects to the server properly, doesn't crash, and properly submits its bids.

The command is `/course/cs1951k/pub/2020/SpectrumAuction/test`. It should be run from your project's root directory, and it runs 5 iterations of the game, and should take about one minute, or less (but it may take a bit longer the first time, if Maven needs to download any packages).

If it succeeds, you'll be able to see the results of your game against the bots. As such, feel free to also use this script to debug your agent's strategy.

Finally, run `/course/cs1951k/pub/2020/SpectrumAuction/submit` to submit your agent. We only store your most recent handin, so if you wish to update your agent and submit a newer version at some later date, just repeat these steps and your submission will be replaced.

# 5 Game API

## 5.1 `MySpectrumAuctionAgent` class

Your task is to fill in the following methods of the `MySpectrumAuctionAgent` class:

- `getNationalBids(Map<String, Double> minBids)`: your strategy as the national bidder. This is called each round of the ascending auction; you should return a `Map<String, Double>` representing your bid for each item (represented as `String`s) that you would like to bid on.
- `getRegionalBids(Map<String, Double> minBids)`: your strategy as the national bidder. This is called each round of the ascending auction; you should return a `Map<String, Double>` representing your bid for each item (represented as `String`s) that you would like to bid on.

  As stated in Section 2.1, as the regional bidder, your bid bundle may only contain up to four goods. If this rule is broken, your bid will be rejected by the server.

In both of the above methods, `minBids` represents the **minimum allowable bid** for each respective item, *not* the current price. Specifically, `minBids` contains the current prices plus $\epsilon$.

Furthermore, `minBids` will be filtered so that its keys correspond only with the items for which you are eligible to bid, based on the diagram in Section 2.1. However, you are still responsible for trimming the size of your bid bundle down to 4 or fewer as a regional bidder.

Be careful with this, because if any single one of your bids is too low, or on an ineligible good, your entire bid bundle will be rejected. And due to the revealed preference rule, "sitting out" a round can have dire consequences. We have provided a few useful tools for checking the validity of your bids; see Section 4.2.2.

In addition, we have provided the following utility methods that you may choose to fill in, if you find them useful:

- `onAuctionStart()`: called at the beginning of each simulation of the Spectrum Auction. If you are keeping any per-simulation data, this is where you would want to initialize or reset it.
- `onAuctionEnd(allocations, payments, tradeHistory)`: at the end of an auction simulation, gives you all of the information from the game, for you to use however you want. Unlike the tentative allocations you have access to during the game, this contains information on *all agents'* bidding history and final allocations and payments. Entries in these variables that correspond with your agent in particular will have a key or agent ID of `this.getAgentBackend().getPublicID()`.

### 5.2 Useful inherited methods

You may find the following methods useful in the Spectrum Auctions. These are all inherited by your agent.

- `int getBidderPosition()`: returns your bidder position, a number 1-7. This corresponds with the bidder types in the diagram in Section 2.1.
- `double getValuation(Collection<String> goods)`: returns your agent's valuation for a bundle of goods. For simplicity have also provided a version of this method which takes in a single `String`, so you can easily determine your single-item valuations.
- `boolean isEligible(String good)`: returns `true` if you are eligible to bid on the specified good; returns `false` otherwise.
- `boolean isValidBidBundle(Map<String, Double> myBids, Map<String, Double> minBids)`: returns `true` if `myBids` is a valid bid bundle to submit to the server. Performs the following checks:

  1. None of your bid amounts are `null`.
  2. You are eligible to bid on all goods in `myBids`.
  3. If you are a regional bidder, `myBids` has size 4 or less.
  4. All bids in `myBids` are at least the minimum allowable bid, as indicated in `minBids`.
  5. Your bid bundle satisfies the revealed preference rule (we check this by maintaining records of past good prices, and of your past bids).

  We *highly* recommend using this, as if your bid bundle passes this check, it is guaranteed to be accepted by the server.
- `Set<String> getTentativeAllocation()`: gets the set of goods currently tentatively allocated to your agent.
- `double clipBid(String good, double bid, Map<String, Double> minBids)`: returns `bid` clipped to be above the minimum allowable bid for `good`. In other words, returns `max(bid, minBids[good])`.
- `int getCurrentRound()`: returns a number indicating the current round number in the ascending auctions. Is 0-indexed.
- `Map<String, Double> getAllMinBids()`: returns a map containing the minimum bids for *every* good, not just those your agent is eligible for. This can be used to predict opponent strategies, but you should use the `minBids` parameter to determine the set of goods you want to bid on, since if you submit a bid with an ineligible good, it will be rejected.

## 6 Tier 1 Agent

We have provided you with access to our Tier 1 TA Bot implementation to test your own agent against. The Tier 1 bot is implemented in the `Tier1SpectrumAuctionAgent` class, and does the following:

- As the national bidder, identifies the set of goods for which its single-good valuation is above the minimum bid. Bids at exactly the minimum bid for each of those goods.
- As a regional bidder, identifies the set of goods for which its single-good valuation is above the minimum bid, then restricts its interest to the four goods with the highest single-good valuations. Bids at exactly the minimum bid for each of those goods.