Final Project: Spectrum Auction Simulation

1 Introduction

This final project option is a Spectrum Auction, which builds on the work we did in Labs 6 and 7.

2 Game Description

The model of the world, goods, bidders, and valuations is slightly different than the setup of Lab 7. In particular, whereas there we assumed the **Global Synergy Value Model** (**GSVM**), for the final project we are assuming the **Local Synergy Value Model (LSVM)**, also with national and regional bidders. This model is so-called because the regional bidders are assigned a local proximity where they prefer to operate.

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 LSVM-18 Model

LSVM-18[1] is a small model of a spectrum auction. There are 18 goods, labelled A through R. They are arranged in a grid as follows:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	O	P	Q	R

Following Scheffel et. al, we assume one national bidder and five regional bidders. Valuations are generated according to the following procedure:

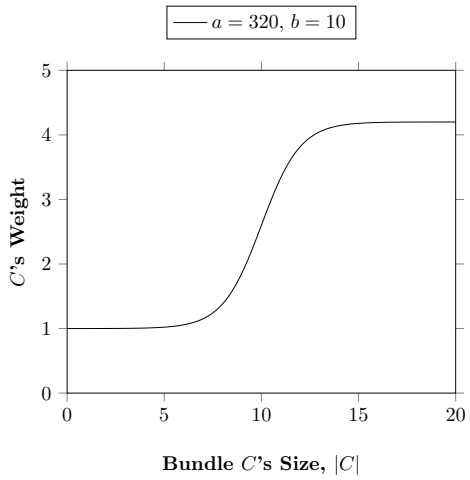
- The *national bidder* has positive base values for all 18 goods, drawn i.i.d. from $U[3, 9]$.
- Each *regional bidder* is assigned a *preferred* good drawn uniformly at random from the set of goods. The regional bidder's *proximity* consists of: their preferred good, goods that are vertically or horizontally adjacent to it, and goods that are vertically or horizontally adjacent to *those*.[2] Thus, a "preferred" good does not mean the bidder values it more than it does other goods; it is merely used to define a regional bidder's proximity. Shown below is the proximity of a regional bidder that prefers good **O**:

A	B	C	D	E	F
G	H	I	J	K	L
M	N	**O**	P	Q	R

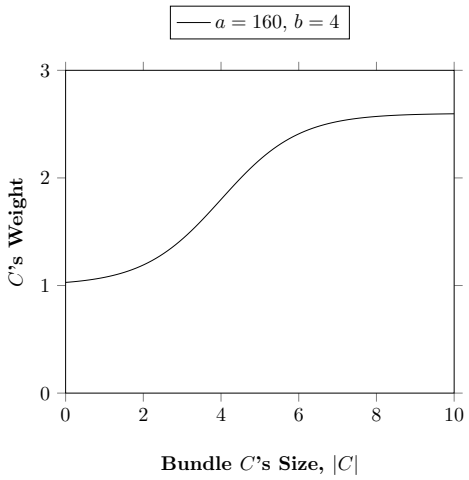
  Regional bidders have base values drawn i.i.d. from $U[3, 20]$ for goods within their proximity, and base values of 0 for goods outside it.
- Define P as a partition of the set of all goods containing maximally connected bundles. That is, for all $C \in P$, with good k in C, all goods l adjacent to k are also in C.

[1] Tobias Scheffel, Georg Ziegler, and Martin Bichler. On the impact of package selection in combinatorial auctions: an experimental study in the context of spectrum auction design. *Experimental Economics*. 15:667–692, 2012.

[2] In other words, the proximity is the set of goods with a Manhattan distance from the preferred good of at most 2.



(a) National Bidder's Bundle Valuations.



(b) Regional Bidders' Bundle Valuations.

Bundle values are computed according to the following function:[3]

$$v_i(S) = \sum_{C \in P} \left(1 + \frac{a}{100\left(1 + e^{b-|C|}\right)}\right) \sum_{k \in C} v_i(k) \tag{1}$$

In words, the goods in a bundle S are maximally partitioned in P, and then the value of S is computed as a weighted sum of the total base values of all elements C of the partition, where the weights are determined by the constants a and b and the element's size. For the national bidder, $a = 320$ and $b = 10$, while for all regional bidders, $a = 160$ and $b = 4$. Thus, for the national bidder, the weight of a bundle remains small until it is of size 6, at which point it increases rapidly, tapering off again around 12. For regional bidders, smaller bundles, say of size 2, are assigned relatively small weights; but weights increase rapidly as the bundle size approaches 6, at which point they taper off again.
N.B. Goods outside a regional bidder's proximity *can* contribute to the weighting function, if they are in the same element C of P as a good within the proximity. Thus, even though the base value of such goods is 0, regional bidders can still accrue additional value by winning goods outside their proximity.

- Bidders may bid on as many goods as they like. Indeed, regional bidders may bid outside their respective proximities, which, as already noted, can add value.
- In addition to base values, which are private information, each regional bidder's preferred good is also private information.

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 ϵ. Between rounds, the prices across all auctions are broadcast to all agents. Moreover, each agent is informed of all goods it is tentative winning. (Ties are broken randomly.)

[3] You have access to this function in the code (see `getValuation` in Section 5); you do *not* need to implement it yourself.

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 2.5. Before delving into those details, we present our version of SMRA:

Algorithm 1 The SMRA Spectrum Auction

Inputs: A set of goods G
Outputs: An allocation matrix (an assignment of goods to bidders) and pricing (one price per good)

Initialize the tentative allocation to be empty
Initialize the prices of all goods to 0

do
  Broadcast all current prices
  Send each bidder their tentative allocation
  Get bids from each bidder comprising at most one bid per good
  Prune bids that violate the **revealed preference activity rule**
  Prune all bids with any entries that do not exceed the respective good's current price by at least ϵ
  Call all remaining bids *valid*
  if the set of valid bids is non-empty **then**
    for each good $g \in G$ with at least one valid bid **do**
      Let bidder i^* be the current winner of good g, if any
      Define the set of *incremental* bids on g as the valid bids placed by bidders other than i^*
      if the set of incremental bid(s) on g is non-empty **then**
        Tentatively allocate g to a highest bidder
        if there exists a second-highest valid bid b_2 **then**
          Set the price of g to b_2
        else
          Increase the price of g by ϵ
        end if
      end if
    end for
  end if
while the set of incremental bids for at least one good is non-empty

The outcome of the auction is the final allocation and the final prices

If the highest bid for a good is not unique, a winner is chosen uniformly randomly among the highest bidders.

2.3 Examples

Below are multiple examples of auction outcomes, each from one round to the next, for an individual good. We assume for the sake of simplicity that Bidders A and B are always the two highest bidders, if they bid at all, so no other bidders come into play. Invalid bids are not shown. Incremental bids (bids by bidders other than the current winner) are highlighted in yellow.

Current Winner	Current Price*	Bid A	Bid B	New Winner	New Price
-	0	-	-	**-**	0
-	0	ϵ	-	**A**	ϵ
-	0	$c \geq \epsilon$	-	**A**	ϵ
-	0	$3+\epsilon$	$2+\epsilon$	**A**	$2+\epsilon$
A	p	-	-	**A**	p
A	p	$p+3\epsilon$	-	**A**	p
A	p	-	$p+3\epsilon$	**B**	$p+\epsilon$
A	p	$p+2\epsilon$	$p+2\epsilon$	**A or B**	$p+2\epsilon$
A	p	$p+3\epsilon$	$p+2\epsilon$	**A**	$p+2\epsilon$
A	p	$p+2\epsilon$	$p+3\epsilon$	**B**	$p+2\epsilon$

*The minimum valid bid is this value (i.e., the current price) plus ϵ. Even the current winner cannot place a bid that is not above this minimum. Once bidders beat this minimum, however, they need not bid in increments of ϵ: e.g, if the current price is 10 and the price increment is 2, any bid of 12 or greater is valid.

2.4 Game Specifics

- We set the price increment $\epsilon = 0.1$.
- To make sure our simulated auctions don't run indefinitely, we cap their number of rounds. If an auction reaches a predetermined final round without having terminated, it will end abruptly. This termination point will be drawn from an exponential distribution, and is guaranteed to be at least 1000. Although this distribution is common knowledge, the draw will not be revealed to your agent or any others, as knowing this number would facilitate sniping.[4]

2.5 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: A bidder is only allowed to switch their preferred demand set from S to T if, since the time when S was preferred, the price of T increased by less than the price of S. If the price of T increased by more than the price of S since any time when the bidder's demand set was 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 $\mathbb{R}^m$ 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:

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

[4] 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.

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, as this will render all future bids invalid.*

3 Strategy and Considerations

This is a very complex game and requires several strategic considerations beyond what was in play in the labs.

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? Will your agent make use of the valuation distributions (which are public knowledge) to try to predict the bids of opposing agents, and ultimately the final auction prices?

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 doing so could potentially reveal too much information to other bidders. Put another way, 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 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; how will this affect your agent's bidding strategy?

A successful agent strategy for this game will comprise these ideas and 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 TA Bots

We are providing you with access to various TA bots to test your own agent against. For starters, the Tier 1 bot, which is implemented in the `MinBidAgent` class, does the following:

- Identifies the set of goods for which its single-good value is above the minimum bid.
- Bids exactly the minimum allowable bid on each of those goods.

The `JumpBidder` agent class works as follows:

- Identifies the set of singletons (i.e., individual goods) for which its value is above the minimum bid.
- Bids a random value on each of these singletons between the minimum allowable bid and the valuation for the good.

And the `TruthfulBidder` agent class:

- Identifies the set of singletons (i.e., individual goods) for which its value is above the minimum bid.
- Bids the valuation for the good on each of these singletons.

5 Testing Arena

To test your agent *offline* against other agents, you should run your `my_agent.py` file. The main method instantiates 10 agents, including your own, and runs a local simulation, offline.

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. To do this, you simply need to configure the `LSVMArena` in the main section of `my_agent.py` file.

You can test against any number of copies of your own agent, any number of copies of our Tier 1 TA Agent and other simple agents (see Section 4), or any combination thereof. You can also create your own dummy agents to test your own agent against; but they also need to inherit from `MyLSVMAgent`.

Here are the attributes in `LSVMArena` that you are allowed and encouraged to tweak locally:

- `num_cycles_per_player` - This is how many times each player will be matched with five other random players. Six auctions will be simulated with these agents, with each agent as the National bidder exactly once. By default, this is set to 3.
- `timeout` - This is the total number of seconds that each agent has to submit their bids. In the actual auction, this will be limited to 1 second, but this is adjustable locally if you wish to test out strategies that take longer than 1 second.
- `local_save_path` - This is the path of the directory *relative* to your root directory that you want to use to save the simulated auctions in. These logs will contain histories of *all* the agents in the auction, and can be trained on as mentioned in Section B.
- `players` - Perhaps most importantly of all, agents is the list of all the agents that will be competing in the arena! This is where you can vary the configuration of your agent to test them out locally.

With the default configurations, testing locally should take no more than 5 to 10 minutes, and will usually be faster than that. Before submitting your agent, please be sure to run it locally against other local agents to ensure that it properly submits its bids.

6 Submission

Before submitting your code through gradescope, please make sure that

- The submitted agent in `agent_submission.py` is named
- You don't name any of your files `random.py` or `numpy.py` or `<Insert Common Package Name>.py`
- Please let us know if you want to import any new packages not natively provided in `agt_server` and we will install them on our end so that your code can import them in your final submission. (E.g., for Lemonade, we installed `tensorflow` for some agents' `tensorflow` code to run correctly.)

A Game API

A.1 `MyLSVMAgent` class

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

- `self.getBids()`: this function represents your agent's strategy. It is called each round, and returns a `Dict[str, float]` representing your agent's bid for each item (represented as `str`s) that it would like to bid on. The stencil code helps you split up your agent's national and regional bidder strategies.
- `self.regional_bidder_strategy()`: this function represents your agent's regional bidding strategy. It is called when your agent is the regional bidder, and returns a `Dict[str, float]` representing your agent's bid for each item (represented as an `str`) that it would like to bid on.
- `self.national_bidder_strategy()`: this function represents your national bidding strategy. It is called when your agent is the national bidder, and returns a `Dict[str, float]` representing your agent's bid for each item (represented as an `str`) that it would like to bid on.
- `self.setup()`: this is an initialization function that is called exactly once before every auction starts.
- `self.update()`: this is an update function that is called exactly once between all auction rounds.

In both of the above methods, `self.get_min_bids()` represents the **minimum allowable bid** for each respective item, *not* the current price. Specifically, `self.get_min_bids()` contains the current prices plus ϵ.

Be careful with this, because if any single one of your bids is too low, your entire bid bundle will be rejected. And due to the revealed preference rule, "sitting out" a round can have dire consequences.

We provide a few useful tools for checking the validity of your bids, as described next.

A.2 Useful Inherited Methods

A.2.1 Information Methods

The following methods collect information about your agent. All are inherited by your agent.

- `self.get_regional_good()` - returns a string that is the regional good of your bidder if you are the regional bidder. If you are not, then this returns None.
- `self.get_goods()` - returns a set of strings representing the goods in the auction, namely {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R"})
- `self.is_national_bidder()` - returns a boolean indicating whether or not the agent is a national bidder or not. True if you are the national bidder and False if you are a regional bidder.
- `self.get_shape()` - returns a tuple that corresponds to the shape of the game. For LSVM18 this returns $(3, 6)$.
- `self.get_num_goods()` - returns an integer that corresponds to the size of the game. For LSVM18 this returns 18.
- `self.get_goods_to_index()` - returns a `Dict[str, Tuple[int, int]]` that maps good names to indices on a 3 x 6 grid.
- `self.get_tentative_allocation()` - returns a set of strings representing the names of the goods that your agent is tentatively allocated.
- `self.get_current_round()` - returns the current round of the auction.
- `self.get_goods_in_proximity()` - returns the names of the goods that are within a agent's proximity if they are the regional bidder. Otherwise, it returns all goods.
- `self.proximity(arr = None, regional_good = None)` - returns a masked numpy ndarray of `arr` that only includes elements of `arr` that are a distance of two away from the regional good; the rest are zeroed out. If no specific regional good is specified, the function uses the regional good assigned to your agent. Similarly, if no array is specified, the function uses the valuations assigned to your agent.

A.2.2 Utility Calculation Methods

Here are a few methods that can be used to track your agent's tentative utility as the auction progresses:

- `self.calc_total_valuation(bundle=None)` - returns the total valuation of a given bundle of goods (set of strings) using the formula listed above. If no bundle is specified, then this function calculates the total valuation for your agent's tentative allocation.
- `self.calc_total_prices(bundle=None)` - returns the total cost of a given bundle of goods (set of strings). If no bundle is specified, then this function calculates the total cost of your agent's tentative allocation.
- `self.calc_total_utility(bundle=None)` - returns the total utility of a given bundle of goods (set of strings). If no bundle is specified, then this function calculates the total utility for your agent's tentative allocation.

A.2.3 Bidding Methods

The following methods can be used to generate and verify your bids. We *strongly* encourage you to make use these functions, especially `is_valid_bid_bundle`.

- `self.get_valuation_as_array()` - returns valuations as a numpy ndarray.
- `self.get_valuation(good)` - given a `good` (as a string), returns the good's valuation.
- `self.get_valuations(bundle=None)` - given a `bundle` (set of goods), returns a dictionary mapping each good in the bundle to its valuation. If no bundle is specified, this function returns a map from every good to its valuation.
- `self.get_min_bids_as_array()` - returns a numpy ndarray of minimum bids which is epsilon greater than current prices.
- `self.get_min_bids(bundle=None)` - given a `bundle` (set of goods), returns a dictionary mapping each goods in the bundle to the minimum allowable bid, which is the current price plus epsilon. If no bundle is specified, this function returns a map from every good to its minimum allowable bid.
- `self.is_valid_bid_bundle(my_bids)` - returns `true` if `my_bids` is a valid bid bundle (dictionary from goods to bids) to submit, by performing the following checks:
  1. None of your bid amounts are `None`.
  2. All bids in `myBids` are at least the minimum allowable bid, as indicated in `my_bids`.
  3. 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 running this check; if your bid bundle passes, it is guaranteed to be accepted by the server.
- `self.clip_bids(my_bids)` - given a dictionary `my_bids` mapping goods to bids, clips each of the bids to the minimum allowable bid for that good.
- `self.clip_bid(good, bid)` - returns the `bid` clipped to the minimum allowable bid for that `good`.

A.2.4 Game Reports

To facilitate online learning in the agents we provide a helper method that returns a `game_report` object containing all of the historical information available to the agent.

- `self.get_game_report()` - returns a `game_report` containing all the historical information the agent sees throughout the run of an auction

`game_report` objects come equipped with the following helper methods. For convenience, these methods have also been shortcutted to the agent itself, so, for example, `self.get_game_report().get_util_history` is the same as `self.get_util_history()`.

- `self.get_util_history()` - returns a list containing your agent's utility given your agent's tentative allocations in each of the previous rounds of bidding during an auction.

- `self.get_previous_util()` - returns your agent's most recent utility given its tentative allocation in the previous round of the auction.
- `self.get_bid_history()` - returns a list containing your agents' bids (formatted as an ndarray) in each of the previous rounds of the auction.
- `self.get_previous_bid()` - returns your agent's bids in the previous round of the auction (formatted as an ndarray).
- `self.get_bid_history_map()` - returns a list containing your agent's bids in each of the previous rounds of the auction (formatted as a dictionary).
- `self.get_previous_bid_map()` - returns your agent's bids in the previous round of the auction (formatted as a dictionary).
- `self.get_price_history()` - returns a list containing the prices (formatted as an ndarray) in each of the previous rounds of the auction.
- `self.get_current_prices()` - returns the auction's current prices (formatted as an ndarray).
- `self.get_price_history_map()` - returns a list containing the prices in each of the previous rounds of the auction (formatted as a dictionary).
- `self.current_prices_map()` - returns the auction's current prices (formatted as a dictionary).
- `self.get_winner_history()` - returns a list containing the tentative winners (formatted as an ndarray of objects) in each of the previous rounds the auction.
- `self.get_previous_winners()` - returns the tentative winners in the previous round of the auction (formatted as an ndarray of objects).
- `self.get_winner_history_map()` - returns a list containing the tentative winners (formatted as a dictionary) in each of the previous rounds the auction.
- `self.get_previous_winners_map()` - returns the set of tentative winners in the previous round of the auction (formatted as a dictionary).

A.2.5 Conversion methods

Since many of the methods, like `self.get_min_bids()`, which return a dict also have ndarray versions, like `self.get_min_bids_as_array()`, here are two methods to help convert between the two!

- `self.map_to_ndarray(map, object=False)` - translates from a map from goods to values in `map` to an ndarray. This defaults to creating an ndarray of float values if object is false; but if object is true, this creates an ndarray of python objects, enabling you to store arbitrary information in the ndarray.
- `self.ndarray_to_map(arr)` - translates from an ndarray (`arr`) to a map from goods to values in `arr`.

B Training from Saved Data

If you design a strategy that requires training, you can submit a pretrained agent, by which we mean an agent that reads a precomputed data file. To use a pretrained agent that reads its input from a file, you should save the file locally using the provided `path_from_local_root` function. Here is an example of how to use it in your `update` function, for example, if you save and read from `text.txt`:

```
def update(self):
    # To save a file locally to test.txt, for example, please use the path_from_local_root
    # method so this filepath is preserved later on when run in the actual competition

    self.filename = path_from_local_root('test.txt')
    weight = random.random()
```

```
# Writing to the file
with open(self.filename, 'w') as file:
    [Insert your file writing code here]

# Reading from the file
with open(self.filename, 'r') as file:
    [Insert your file reading code here]
```

Furthermore, if you specify the `local_save_path` in your `LSVMArena`, every single arena run will be saved as a `.json.gz` file that can be read later to train an agent. A short code snippet for how you might be able to interface with these files is provided in the `process_saved_game` and `process_saved_dir` functions.