1 Introduction

In this lab, you will being your foray into the design of bidding heuristics for **combinatorial auctions**. A combinatorial auction is one in which bidders valuations for **bundles** (i.e., sets) of goods are non-additive. In particular, a bidder can value a bundle more than the sum of its parts (think a pillow and a pillowcase; **complements**), or less than the sum of its parts (think ammonia and bleach; **substitutes**).

During the next two labs, you will be implementing agent bidding strategies for one of the simplest combinatorial auction designs, **simultaneous sealed-bid auctions**. Simultaneous auctions are exactly what they sounds like: parallel auctions in which there are multiple goods to bid on (say m), simultaneously. In a sealed-bid auction, bidders submit private bids to the auctioneer.

At the start, the server announces the goods up for auction, and sends the agents their valuations (or access to a function that computes their values for all bundles, if there are too many to enumerate). In a sealed-bid auction, the server then waits for the agents to submit their bids, before announcing the outcome.

Auctions are challenging to analyze when goods are sold separately, but bidders' valuations are combinatorial. Even when bidders are concerned only with their own winnings (not the other bidders' winnings), it may not be possible to describe their valuations using fewer than 2^m numbers. Bidding heuristics for simultaneous auctions try to make sense of all of these numbers using a reasonable (i.e., preferably sub-exponential) number of calculations, as they search for a vector of bids that yields high utility as often as possible.

Many successful heuristics/strategies employ the following two-tiered architecture:

1. predict: build a model of the highest other-agent bids (i.e., the auctions' clearing prices)
2. optimize: solve for an (approximately) optimal set of bids, given this model

During the next two labs, you will be implementing a prediction method that generates so-called **self-confirming prices**, and an optimization routine called **LocalBid**.[1] **Self-confirming prices** in an auction are prices that are realized when all agents bid according to a two-tiered strategy (price prediction plus optimization), and the price prediction input to the optimization routine equals its output. **LocalBid** is an algorithm that updates bid for each good in turn, by computing **marginal values** given the current bid vector and predicted prices. Before writing any code, you will develop your intuition about marginal values by working through a few examples. These examples involve both complements and substitutes.

2 Setup

You can find and clone the stencil code for Lab 6 here. The stencil code includes:

- `marginal_value.py`*
- `test_marginal_value.py`*
- `local_bid.py`*
- `sample_valuations.py`

The star annotations indicate which files you will edit during this lab. The others are purely support code and/or already-implemented opponent bots for the simulations,

Please be sure to read the `README.md`. Here is an abridged version of the setup guide described there:

1. Clone the repository with the command
```
git clone https://github.com/brown-agt/lab06-stencil.git
```

[1] Michael P. Wellman, Eric Sodomka, & Amy Greenwald. Self-confirming price-prediction strategies for simultaneous one-shot auctions. *Games and Economic Behavior*, 102:339–372, 2017.

2. Create a python virtual environment and activate it.

3 Marginal Values

Recall that in a single second-price auction, the equilibrium strategy is for each bidder to bid their true valuation for the good. Why isn't this good enough in *simultaneous* second-price auctions? The reason is that a bidder does not have merely one valuation for each individual good, but rather *many* valuations for each, depending on context: i.e., the other goods in the bidder's bundle. In other words, a bidder's valuation for winning a good is a function of the other goods that bidder also wins.

Example:[2] As an example, suppose an agent values a camera and flash together at 500, and either good alone at 1. Also, suppose these two goods are sold separately in two simultaneous auctions, and that the highest other-agent bids turn out to be 200 for the camera, and 100 for the flash. If the agent were to bid only its true, but individual, values (1), it would lose both goods, obtaining utility of 0 rather than $500 - 200 - 100 = 200$. This bidding strategy is suboptimal: the agent fails to win goods it wishes it had won—and which it could have won had it bid otherwise: e.g., 500 on both goods.

Example: Now suppose an agent values a Canon AE-1 at 300 and a Canon A-1 at 200, but values both cameras together at only 400. Also, suppose these two goods are sold separately in two simultaneous auctions, and that the highest other-agent bids turn out to be 275 for the AE-1 and 175 for the A-1. If the agent were to bid its true, but individual, values, it would win both goods, obtaining utility of $400 - 450 = -50$. This bidding strategy is again suboptimal: the agent wins goods it wishes it had not won—and which it would not have won had it bid otherwise: e.g., 500 on one and only one of the cameras.

In the first of these two examples, the goods are called **complements**, as they complement one another. In the second, they are called **substitutes**. In simultaneous (or sequential) auctions where goods are complements or substitutes, an alternative to bidding individual valuations is to bid **marginal values**. The marginal value of a good to a bidder is the difference between the bidder's utility assuming that they have the good, and their utility assuming they don't. As bidding individual valuations has been shown to be insufficient in combinatorial auctions, we consider the use of marginal values in bidding strategies next.

3.1 Marginal Values

Consistent with our proposed architecture (and our examples), we assume a price vector $\boldsymbol{q} \in \mathbb{R}^m_{\geq 0}$, which represents the highest other-agent bids in all m auctions. Given a bidder i, the **marginal value** of a good j to bidder i is the difference in *utility* between having good j and not having it.

To compute this marginal utility, bidder i determines its optimal utility by searching over all subsets $G \setminus \{j\}$ twice: first, assuming j is available at no cost, while all other prices are given by $\boldsymbol{q}$; and second, assuming j is unavailable, while all other prices are given by $\boldsymbol{q}$.

Formally, if each bidder i ascribes value $v_i(X)$ to $X \subseteq G$, and if $q(X) = \sum_{k \in X} q_k$, then the marginal value $\mu_{ij}(\boldsymbol{q})$ is given by:

$$\mu_{ij}(\boldsymbol{q}) = \max_{X \subseteq G \setminus \{j\}} [v_i(X \cup \{j\}) - q(X)] - \max_{X \subseteq G \setminus \{j\}} [v_i(X) - q(X)] \tag{1}$$

Question: Consider once again the setup of our first example above. Given both the camera and flash together, the bidder's value is 500; but either one of these components without the other is valued at only

[2]The examples in this lab were borrowed from Amy Greenwald and Victor Naroditskiy. Heuristics for the Deterministic Bidding Problem. *SIGecom Exchanges.* 6(1):35–44, 2006.

1. If the highest other-agent bids on the camera and flash are 200 and 100, respectively, then what are the marginal values of the camera and the flash? Is bidding marginal values a good idea in this example?

Question: Consider once again the setup of our second example, where a bidder values a Canon AE-1 at 300 and a Canon A-1 at 200, and both cameras together at 400. If the highest other-agent bids on the AE-1 and A-1 are 275 and 175, respectively, then what are the marginal values of the AE-1 and the A-1? Is bidding marginal values a good idea in this example?

Note: We will be using a slightly simplified definition of Marginal Value below for our actual implementation in preparation for LocalBid

3.2 LocalBid

The optimization problem embedded in the marginal value calculation involves optimizing utility by searching over all nearly all subsets of G, which can be very expensive (even once, let alone twice!). An alternative approach based on local search leads to a bidding heuristic called **LocalBid**. The key idea underlying LocalBid is that the combination of a bid vector $\boldsymbol{b}_i$ for bidder i and a price vector $\boldsymbol{q}$ of highest other-agent bids yields an allocation for bidder i: i.e., $x_i(\boldsymbol{b}_i, \boldsymbol{q}) = \{j \mid b_{ij} \geq q_j\}$. That is, bidder i wins all goods for which $b_{ij} \geq q_j$ at price q_j. Now the **marginal value** of a good j to bidder i, given a vector of bids $\boldsymbol{b}_i$ as well as a price vector $\boldsymbol{q}$, is simply the difference in value between having good j and not having it: i.e.,

$$\mu_{ij}(\boldsymbol{b}_i, \boldsymbol{q}) = v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \cup \{j\}) - v_i(x_i(\boldsymbol{b}_i, \boldsymbol{q}) \setminus \{j\}) \tag{2}$$

Note that prices do not come into play in this calculation, because they are fixed (on both sides of the subtraction, they equal $\sum_{k \in x_i(\boldsymbol{b}_i, \boldsymbol{q})} q_k$), so they cancel out.

The LocalBid algorithm, starts from some initial bid vector $\boldsymbol{b}_i$ (such as each good's individual valuation), and then repeatedly iterates over all goods, updating bidder i's bid on good j to be i's marginal value for j, given $\boldsymbol{b}_i$ and $\boldsymbol{q}$. LocalBid may not converge, so it is typically run for some fixed number of iterations, or its bids are smoothed to favor more recent iterations, effectively forcing convergence.

Besides saving on computational complexity, LocalBid can also produce optimal bids in situations where bidding marginal values straight up would not.

Example: Assume a bidder ascribes a value of 2 to one or more of the m goods, and that the highest other-agent bid on each good is 1. Bidding marginal values would amount to bidding 1 on each good. Thus, in the worst case, the marginal utility bidding strategy obtains utility $2 - N < 1$.

Question: What utility would LocalBid obtain in this example? Can you devise another bidding strategy that would achieve the same in this example (even if it performs worse in other cases)?

In the next two sections, we revisit the definition of marginal value and the description of LocalBid using pseudocode, rather than math or English. Your implementations should follow this pseudocode closely.

3.3 Marginal Values Revisited, in Pseudocode

LocalBid calculates marginal values for all goods, given an *assumed bid* and an *assumed price* (i.e., highest other-agent bid) for each good. In Algorithm 1, we provide pseudocode for a key subroutine of LocalBid, namely the calculation of the marginal value of a single good $g_j \in G$, given a valuation function v, a bid vector $\boldsymbol{b}$, and a price vector $\boldsymbol{q}$.

Intuitively, marginal values are calculated by comparing the bidder's assumed bid for each good to an assumed price for that same good. Via this comparison, the bidder predicts which goods they will win, and which they will lose; if the bid is greater than the price, they win; otherwise, they lose. They then find the difference in valuations between their predicted winnings plus the good in question, and their predicted winnings as they stand. This valuation difference is the marginal value of interest. Solving for the marginal value of one good via this procedure (Algorithm 1) is linear in the total number of goods.

Algorithm 1 Calculate the marginal value of good $g_j \in G$

```
INPUTS: Set of goods $G$, select good $g_j$, valuation function $v$, bid vector $\boldsymbol{b}$, price vector $\boldsymbol{p}$
OUTPUT: The marginal value (MV) of good $g_j$

bundle ← {}
for each $g_k \in G \setminus \{g_j\}$ do    ▷ Simulate either winning or losing $g_k$ by comparing assumed bids and prices.
    price ← $p_k$
    bid ← $b_k$
    if bid > price then    ▷ The bidder wins $g_k$. Add it to the bundle of won goods.
        bundle.add($g_k$)
    end if
end for

MV ← $v(\text{bundle} \cup \{g_j\}) - v(\text{bundle})$
return MV
```

3.4 LocalBid Revisited, in Pseudocode

The LocalBid strategy takes as input an initial bid vector $\boldsymbol{b}$, and updates this bid vector by computing marginal values for each good in turn. The bid vector can be initialized at random, but a common choice is the bidder's individual valuations for each good. Marginal values are then calculated good-by-good via Algorithm 1, updating the bid vector with these values as the new assumed bids. As each update affects the marginal values of all the other goods, LocalBid repeats this process, either for a set number of iterations, or until the bid vector converges.[3] Upon termination, the bid vector represents the agent's marginal values for each good, given the rest of the assumed bids (also marginal values). These are the bids placed by LocalBid.

Recall that solving for the marginal value of one good via Algorithm 1 is linear in the total number of goods. So the runtime to calculate the marginal value of all m goods once is $O(m^2)$. Embedded within a loop, the total runtime of LocalBid is then $O(m^2 T)$, where T is the maximum number of iterations. Depending on the value of T, this can be a *massive* runtime improvement over any strategy that enumerates the valuations of all 2^m bundles, including our original marginal value calculation (Equation 1).

Below, we present pseudocode for LocalBid.

4 Time to Code

Hopefully, you recall from your introductory computer science courses that one should demonstrate their understanding of a problem by writing test cases (i.e., examples of inputs and correct outputs) before coding anything. Please test your understanding of LocalBid's marginal value calculation (Equation 2) by developing a few simple test cases. You can refer to the code in `test_example_case` in `test_marginal_value.py` to see an example we created. You should then develop three new test cases of your own before proceeding to code the algorithm by filling out `test_student_case_1`, `test_student_case_2`, and `test_student_case_3`.

[3]Convergence is not guaranteed in general, and may have to be forced.

Algorithm 2 LocalBid

```
INPUTS: Set of goods $G$, valuation function $v$, price vector $\boldsymbol{q}$
HYPERPARAMETERS: NUM_ITERATIONS
OUTPUT:

Initialize bid vector $\boldsymbol{b}_{\text{old}}$ with a bid for each good in $G$    ▷ E.g., individual valuations.

for NUM_ITERATIONS or until convergence do
    $\boldsymbol{b}_{\text{new}} \leftarrow \boldsymbol{b}_{\text{old}}$.copy()    ▷ Initialize a new bid vector to the current bids.
    for each $g_k \in G$ do
        $\boldsymbol{b}_{\text{new},k} \leftarrow$ CalcMarginalValue($G, g_k, v, \boldsymbol{b}_{\text{old}}, \boldsymbol{q}$)    ▷ Insert the marginal value into the new bid vector.
    end for

    ▷ Now update the original bid vector $\boldsymbol{b}_{\text{old}}$ with the new information stored in $\boldsymbol{b}_{\text{new}}$.
    ▷ The simplest update is $\boldsymbol{b}_{\text{old}} \leftarrow \boldsymbol{b}_{\text{new}}$, but there are other possibilities (more on this later).
    ▷ This is also where you can check for convergence.
    $\boldsymbol{b}_{\text{old}} \leftarrow$ UpdateBidVector($\boldsymbol{b}_{\text{old}}, \boldsymbol{b}_{\text{new}}$)
end for

return $\boldsymbol{b}_{\text{old}}$
```

4.1 Calculating a Single Marginal Value

Once you are satisfied with your test cases, and hence your understanding of marginal values, navigate to `marginal_value.py`. Here, you will implement the following method:

```
def calculate_marginal_value(
    goods: set[str],
    selected_good: str,
    valuation_function: Callable[[set[str]], float],
    bids: dict[str, float],
    prices: dict[str, float]
) -> float
```

This method should return the marginal value of `good`. The other arguments correspond to the inputs in the pseudocode in Algorithm 1.

When you have finished your implementation, run `test_marginal_value.py`. This will run your implementation on our test cases and yours. Make sure they all pass before you move on to implementing LocalBid.

4.2 Implementing LocalBid

Now that you can calculate the marginal value of a single good, you have implemented the most important part of the machinery used by LocalBid. Next, using this code as a subroutine, you will implement the whole LocalBid strategy to produce an entire bid vector of marginal values.

Navigate to `local_bid.py`.

You will be implementing the LocalBid strategy to create an iterative, (hopefully) converging calculation of marginal values. To proceed, you should write the following method:

```
def local_bid(goods: set[str],
    valuation_function: Callable[[set[str]], float],
    price_vector: dict[str, float],
    num_iterations: int = 100
) -> dict[str, float]
```

You should call your own subroutine to calculate a single good's marginal value. You can do this by calling `calculate_marginal_value`.

To copy a bid vector, remember use its `copy` method.

Recall from Section 3.4 that LocalBid's runtime over T iterations is $O(T\,m^2)$. Considering the level of improvement this strategy provides over exponential-time alternatives, LocalBid agents can perform many iterations during their search, possibly from many different initializations of the bid vector. But how many iterations is enough to guarantee convergence? And are there some valuation functions for which LocalBid isn't guaranteed to converge at all?

Note: Please leave in the print statements provided in the stencil code; these are meant to provide you with a way to eyeball whether or not the marginal values are converging.

4.2.1 Design Choices

There are two particular design choices in our implementation that may affect the speed of convergence of LocalBid. The first is the initialization of the bid vector. You can choose to initialize it with random values; with the individual valuations for each good; or with anything else you like. Initializing to the individual valuations may speed up convergence in certain special cases. (Think about when and why.) Nevertheless, we encourage you to experiment with different initializations to see how they affect the outcome.

The second design choice is the method of updating the bid vector after each iteration. The simplest way to update is to simply replace the old bid vector with the new one. Another possible idea is called *smoothing*, where you choose some parameter $\alpha \in [0, 1]$, and then perform the update $\boldsymbol{b}_{\text{old}} = (1 - \alpha)\boldsymbol{b}_{\text{old}} + \alpha \boldsymbol{b}_{\text{new}}$. Smoothing preserves the relevance of old data, while prioritizing new data. Smoothing can be used to force the convergence of LocalBid, which is otherwise not guaranteed to converge.

Exercise: Design an example for which LocalBid does not converge.

As above, we encourage you to experiment with different values of α to see how they affect convergence. Do you observe convergence with a constant value of α? If not, you can decay α according to some schedule, such as $O(1/t)$, where t is the current iteration.

4.2.2 Testing

`local_bid.py` provides you with the functionality to test your implementation of LocalBid to see how quickly it converges, if at all.

We built a `SampleValuations` class to represent valuation distributions, which you should use for testing purposes. When you run `localbid.py`, a valuation distribution, along with a (randomly generated) price vector, is passed to your `localBid` function, which then prints out the bid vector it computes at each iteration. Hopefully, you will observe quick convergence.

You should run `localbid.py` using the following four `SampleValuations` values:

- `SampleValuations.additive_valuation` is the simplest valuation of all. The value of any bundle is the sum of the individual valuations of the goods in the bundle.

- `SampleValuations.complement_valuation` is a valuation with a *global complement*. It is like an additive valuation, but adding any additional good to a bundle increases its valuation.
- `SampleValuations.substitute_valuation` is a valuation with a *global substitute*. It is like an additive valuation, except that adding any additional good to a bundle decreases its valuation.
- `SampleValuations.randomized_valuation` is a valuation for which there exists a separate, randomized, complement or substitute for each possible combination of goods. Since there are random values involved, be sure to run this one a few times.

See if you can get them all to converge.

To be sure your implementation is correct, check that when using `additive_valuation`, your bid vector converges to:

```
BidVector: {
    [A : 70.0], [B : 55.0], [C : 85.0], [D : 50.0],
    [E : 15.0], [F : 65.0], [G : 80.0], [H : 90.0],
    [I : 75.0], [J : 60.0], [K : 40.0], [L : 80.0],
    [M : 90.0], [N : 25.0], [O : 65.0], [P : 70.0],
}
```

The `AdditionalSampleValuations` class represents more valuation distributions, which you can use for testing purposes.

5 Self-Confirming Price Predictions

In the above tests, we provided your LocalBid agent with a (random) price vector as input. But bidding agents are ordinarily responsible for predicting prices themselves. Indeed, your next goal is to extend your agent, which so far uses LocalBid to optimize, so that it first builds price predictions.

5.1 Learning in Self-Play

A key component of successful AI game-playing programs, dating back to one of the earliest, a Checker-playing program written by Arthur Samuel in 1959[4]—is learning in self-play. As the name suggests, learning in this way means that an agent plays against itself repeatedly, each time improving its behavior slightly.

In particular, in a symmetric two-player game like Poker, the agent assumes that its opponent is playing its own strategy, and learns how to respond to that strategy. Then, during the next iteration, it assumes its opponent's strategy is the response it learned during the previous iteration, and learns a new a response to that response; and so on. This process continues until convergence (or forced convergence), at which point the algorithm has learned a (near) symmetric equilibrium: i.e., a strategy that is a (near) response to itself.

Observe that the methodology at the heart of this training loop mimics our two-tiered agent architecture for bidding in auctions, namely first predict, and then optimize. When learning in self-play, an agent *predicts* its opponent strategy, which it takes to be its own strategy, and then *optimizes* against it. Likewise, we can wrap a training loop around our bidding agent architecture, so that our agent predicts that its opponents will bid according to its own strategy, and it then computes a response to their collective behavior.

When learning in self-play is used to learn to play games like checkers, chess, and poker, the game is simulated many many times, with the opponent's strategy dictating the game's dynamics, and the agent learning a

[4]If you are interested in learning about the history of AI game-playing programs, we refer you to a AAAI 2020 Panel video moderated by Professor Greenwald.

best response in this environment. Our auctions are one-shot; so our agent's objective is *not* to learn a best response to the game dynamics implied by the opponents' strategies. Rather, the opponents' strategies in an auction imply a distribution over prices—or, more accurately, a distribution over highest other-agent bids—and our agent's objective is to learn this price distribution, against which it can optimize its response.

5.2 Learning Self-Confirming Price Predictions in Self-Play

Your agent will learn price predictions in self-play by repeatedly simulating multiple auctions, in which its bidding strategy (LocalBid) is employed by all the bidders—hence, *self-play.* More specifically, your agent will collect price data during each simulation. As these data are meant to summarize the behavior of the *other* agents in the simulation—not the learning agent itself—the relevant statistics are the highest bids on each good among all the *other* agents (i.e., not including the learning agent itself).

To learn price predictions, you should simulate the auction some number of times (say M), assuming a set of agents who bid according to an input optimization routine, given the current price prediction. This simulation process generates M data points, each of which is an auction outcome, comprising a vector of winning prices. These data points are then input into a learning algorithm, which incorporates the new price data into the old price prediction to learn a new price prediction. This entire process repeats for some number of iterations, until, hopefully, the price predictions converge.

One key implementation detail is how to incorporate the new price data into the old price prediction to generate a new price prediction. One simple method is to simply average all the newly collected simulation price data $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_M$ into a new price prediction $\overline{\boldsymbol{p}}$. Alternatively, this new price prediction $\overline{\boldsymbol{p}}$ can be smoothly incorporated into the old price prediction $\widehat{\boldsymbol{p}}^{t-1}$ via smoothing: i.e., at iteration t,

$$\widehat{\boldsymbol{p}}^t = (1-\alpha)\widehat{\boldsymbol{p}}^{t-1} + \alpha\overline{\boldsymbol{p}}$$

Over the course of learning, the hope is that $\widehat{\boldsymbol{p}}^t$ will converge, leaving the agent with an accurate prediction that it can provide to its optimization routine (e.g., LocalBid). Convergence is not guaranteed for constant values of α, however; a decay schedule may be necessary as in the case of LocalBid.

Self-confirming price predictions (SCPPs) are realized when all agents bid according to a two-tiered strategy (price prediction plus optimization), and the input to the optimization routine equals its output. In other words, the optimization routine forms a symmetric equilibrium (i.e., it is a best response to itself).

Below, we provide pseudocode for this learning procedure, which entails simulating an auction many times. The learned prediction is updated `NUM_ITERATIONS` times, with an actual update triggered only after a fixed number of simulations, namely `NUM_SIMULATIONS_PER_ITERATION`.

6 Implementing an SCPP/LocalBid Agent

Your final task in this lab is to implement an SCPP/LocalBid agent. Navigate to `scpp_agent.py`.

For starters, take a look at `get_action()`. This method returns the agent's next bid vector, which it finds by running `local_bid` to compute marginal values given a vector of self-confirming prices.

You should notice two important variables:

- `learned_prices` represents the price prediction $\boldsymbol{p}_{\text{old}}$ that the agent has learned so far
- `curr_prices` represents the price prediction $\boldsymbol{p}_{\text{new}}$ that is constructed from current simulation data

The SCPP agent also has several useful methods, such as `get_valuations()`, which retrieves valuations

Algorithm 3 SCPP

```
INPUTS: Set of goods $G$, optimization routine $\sigma$, valuation distribution $F_i$, initial price prediction $\boldsymbol{p}_{\text{old}}$
HYPERPARAMETERS: NUM_ITERATIONS, NUM_SIMULATIONS_PER_ITERATION
OUTPUT: A learned price prediction

for NUM_ITERATIONS or until convergence do
    Initialize $\boldsymbol{p}_{\text{new}}$ to store the results of NUM_SIMULATIONS auctions
    for NUM_SIMULATIONS_PER_ITERATION do
        For each agent $i \in [n]$, draw a valuation function $v_i$ from $F_i$.
        Simulate an auction, with each agent playing $\sigma(v_i, \boldsymbol{p}_{\text{old}})$.
        Store the resulting prices in the new prediction $\boldsymbol{p}_{\text{new}}$.
    end for

     ▷ This is also where you can check for convergence
    $\boldsymbol{p}_{\text{old}} \leftarrow$ update($\boldsymbol{p}_{\text{old}}, \boldsymbol{p}_{\text{new}}$)    ▷ Learn new prices from the simulation data stored in $\boldsymbol{p}_{\text{new}}$
end for

return $\boldsymbol{p}_{\text{old}}$
```

from the auction server upon each new simulation of the auction. This corresponds to the step in the SCPP pseudocode where valuations are sampled from their distributions.

Task: Fill in the `update` method to implement SCPP. Your implementation should follow the pseudocode above; you will also find some helpful comments in the code to guide you. Once this method is filled in, you will have a LocalBid agent capable of learning a self-confirming price prediction in simultaneous auctions.

7 Running your Agent

This week, your agent will compete in simultaneous second-price auctions against "truthful" agents: i.e., agents that bid their individual valuations. But before doing so, you will train your agent to learn self-confirming price predictions in self-play.

When running `scpp_agent.py`, you can run it with an argument `--mode`. It should be set to `TRAIN` when training your agent, and `RUN` when running it in a local competition or live auction.

7.1 Training your Agent

Run `scpp_agent.py`, making sure that `--mode` is set to `TRAIN`. This will launch a 500-simulation training phase, in which your SCPP/LocalBid agent trains in self-play: i.e., against multiple copies of itself. Each time an agent updates its `learned_prices`, they will be written to disk, to ensure that the agent optimizes using its current price prediction. Once training completes, the final learned price predictions are likewise written to a file to be loaded when the agent is run in a local competition or a live auction.

At the end of training, the AGT_SERVER will output a utility report (similar to those output after playing the repeated games of our first few labs). The utility values across agents may be quite close to each other, and your agent may not come in first place. This outcome is perfectly fine, as your agent is playing against itself, so you would expect equal outcomes—up to the randomness in the simulations, of course!

7.2 Running your Agent

Run `scpp_agent.py` again, but this time set `--mode` to `RUN`. This will launch another 100-simulation run of the same auction, except this time your trained agent will be competing (locally) against two truthful agents, rather than copies of itself. At the start, your agent will load the price prediction created during the training phase. Hopefully, these learned prices are good enough to estimate your agent's marginal values well, so that LocalBid with these predictions can outperform the truthful agents.

Once again, the AGT_SERVER will output a utility report. Your agent should outperform the other agents, earning average utility that is (slightly) higher than that of the other agents over the course of the 100 runs.

Question: Does your agent's performance against the truthful agents vary with the structure of the agents' valuations? That is, how does your agent perform when valuations are additive compared to when they exhibit complementary or substitutable structure?

8 To Be Continued . . .

This lab does not include a live competition—since everyone implemented the same bidding strategy—but today's bidding strategy will serve as a foundation for you to enter more complicated auction competitions in upcoming labs, as well as in the final project.