

## 1 Introduction

In this lab, you will begin 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 2 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 sound 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 values for all bundles (or access to a function that computes those valuations, 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: i.e., the allocation and payments.

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

Over the course of the next two labs, you will be implementing a prediction method that generates so-called **self-confirming prices**, and an optimization routine called **LocalBid**.<sup>1</sup> In this lab, you will implement an optimization routine only; then next time, you will implement a price-prediction algorithm. But before writing any code, you will develop your intuition about how one might bid in simultaneous *second-price* auctions 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*`
- `localbid.py*`
- `sample_valuations.py`

The star annotations indicate which files you be editing during this lab. The others are purely support code for the lab.

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

1. Clone the repository with the command  
`git clone https://github.com/brown-agt/lab06-stencil.git`
2. Create a python virtual environment and activate it.

---

<sup>1</sup>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.

### 3 Marginal Values

You may 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:**<sup>2</sup> 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  $\mathbf{q} \in \mathbb{R}_{\geq 0}^m$ , 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  $\mathbf{q}$ ; and second, assuming  $j$  is unavailable, while all other prices are given by  $\mathbf{q}$ .

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  $\mu_{ij}(\mathbf{q})$  is given by:

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

**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 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 camera

<sup>2</sup>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.

and flash are 275 and 175, respectively, then what are the marginal values of the camera and the flash? 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  $\mathbf{b}_i$  for bidder  $i$  and a price vector  $\mathbf{q}$  of highest other-agent bids yields an allocation for bidder  $i$ : i.e.,  $x_i(\mathbf{b}_i, \mathbf{q}) = \{j \mid b_{ij} \geq q_j\}$ . 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  $\mathbf{b}_i$  as well as a price vector  $\mathbf{q}$ , is simply the difference in value between having good  $j$  and not having it: i.e.,

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

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

The LocalBid algorithm, starts from some initial bid vector  $\mathbf{b}_i$  (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  $\mathbf{b}_i$  and  $\mathbf{q}$ . LocalBid is not guaranteed to converge in general, 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  $\mathbf{b}$ , and a price vector  $\mathbf{q}$ .

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  $\mathbf{b}$ , price vector  $\mathbf{p}$ **OUTPUT:** The marginal value (MV) of good  $g_j$ 


---

```

bundle  $\leftarrow \{\}$ 
for each  $g_k \in G \setminus \{g_j\}$  do    ▷ Simulate either winning or losing  $g_k$  by comparing assumed bids and prices.
    price  $\leftarrow p_k$ 
    bid  $\leftarrow b_k$ 
    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  $\leftarrow v(\text{bundle} \cup \{g_j\}) - v(\text{bundle})$ 
return MV

```

---

### 3.4 LocalBid Revisited, in Pseudocode

The LocalBid strategy takes as input an initial bid vector  $\mathbf{b}$ , and updates this bid vector by computing marginal values until convergence. This vector can be initialized at random, but a common choice is the bidder's individual valuations for each good. Marginal values are then calculated for each good in turn 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. 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. Please 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`.

### 4.1 Calculating a Single Marginal Value

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

---

**Algorithm 2** LocalBid

---

**INPUTS:** Set of goods  $G$ , valuation function  $v$ , price vector  $\mathbf{q}$ **HYPERPARAMETERS:** NUM\_ITERATIONS**OUTPUT:**Initialize bid vector  $\mathbf{b}_{\text{old}}$  with a bid for each good in  $G$    ▷ E.g., individual valuations.**for** NUM\_ITERATIONS or until convergence **do**     $\mathbf{b}_{\text{new}} \leftarrow \mathbf{b}_{\text{old}}.\text{copy}()$    ▷ Initialize a new bid vector to the current bids.    **for each**  $g_k \in G$  **do**         $MV \leftarrow \text{CalcMarginalValue}(G, g_k, v, \mathbf{b}_{\text{old}}, \mathbf{q})$          $b_k \leftarrow MV$    ▷ Insert the marginal value into the new bid vector.    **end for**        ▷ Now update the original bid vector  $\mathbf{b}_{\text{old}}$  with the new information stored in  $\mathbf{b}_{\text{new}}$ .        ▷ The simplest update is  $\mathbf{b}_{\text{old}} \leftarrow \mathbf{b}_{\text{new}}$ , but there are other possibilities (more on this later).

▷ This is also where you can check for convergence.

 $\mathbf{b}_{\text{old}} \leftarrow \text{UpdateBidVector}(\mathbf{b}_{\text{old}}, \mathbf{b}_{\text{new}})$ **end for****return**  $\mathbf{b}_{\text{old}}$ 

---

```

Float calculate_marginal_value(
    Set[String] G,
    String good,
    Callable[[Set[String]], Float] v,
    Dict[String, Float] b,
    Dict[String, Float] p
)

```

This method should return the marginal value of `good`. The other arguments correspond to the inputs of the same name 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 `localbid.py`.

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

```

Dict[String, Float] localBid(
    Set[String] G,

```

```

    Callable[[Set[String]], Float] v,
    Dict[String, Float] p,
    Int numIterations
)

```

You should use 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  $O(Tm^2)$ . 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  $\alpha \in [0, 1]$  (typically close to 0), and then perform the update  $\mathbf{b}_{\text{old}} = (1 - \alpha)\mathbf{b}_{\text{old}} + \alpha\mathbf{b}_{\text{new}}$ . Smoothing preserves the relevance of old data, while prioritizing new data. Admittedly, smoothing is more relevant when the price vectors are sampled rather than provided up-front (which will happen in the next lab, after you generate probabilistic price predictions). Nonetheless, we encourage you to try it out this week to see how it affects the outcome.

**Extra Credit** Smoothing can be used to force the convergence of LocalBid. Otherwise, it is not guaranteed to converge. Give an example where it does not.

#### 4.2.2 Testing

`localbid.py` provides you with the functionality to test your implementation of LocalBid to see how quickly it converges, if at all. You should test it out using multiple valuation distributions.

Notice the `SampleValuations` dataclass. This represents the valuation distribution for testing purposes. When you run `localbid.py`, the valuation distribution in question, along with a generated price vector for a set of 16 goods, is passed to your `LocalBid` function, which will print out the bid vector at each iteration. Hopefully, you will observe quick convergence.

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

- `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 by another 5%.
- `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 by another 5%.
- `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 each one 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],  
}
```

## 5 To Be Continued ...

This lab does not include a competition, but these bidding strategies will serve as a foundation for you to enter auction competitions in upcoming labs, as well as in the final project.

In the next lab, you will be combining `LocalBid`, a bid optimization routine, with a price prediction method to create fully functional agents for simultaneous second-price auctions.