Approximating the Optimal Revenue of Continuously-Valued Auctions

Anton Stengel

a senior thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Arts in Mathematics at
Princeton University

Advised by S. Matthew Weinberg

May 1, 2023

Designing an incentive-compatible auction that maximizes revenue in a multi-item setting is difficult. When bidder valuation distributions have finite support, finding the optimal revenue is a linear programming problem. But there is no similar computational solution in the continuous case. In this thesis, we present Cara, a program that approximates the optimal revenue of auctions in continuous settings.[1] We motivate Cara by trying to answer the following question: What is the optimal revenue when n additive bidders compete to purchase two items valued according to the equal revenue curve?

[1] Continuous Auction Revenue Approximator - `https://github.com/antonstengel/cara`

Thank you, Matt, for your guidance over the past year. This thesis would not have been possible without your generosity.

This paper represents my own work in accordance with University regulations.

Anton Stengel
May 1, 2023

Contents

Abstract ii

1 Introduction **1**

1.1 Problem Background 1

1.2 Results 2

1.3 Related Work 3

1.4 Roadmap 3

2 Auctions **5**

2.1 Auction Structure 6

2.2 Auction Properties 8

3 Background Theory **12**

3.1 Linear Program 12

3.2 Revenue Monotonicity 16

4 Software Usage **25**

4.1 Additional Terminology 25

4.2 Software Installation 27

4.3 Input Specifications 28

4.4 Execution 33

4.5 Output Specifications 36

5 Results **38**

5.1 Comparison to Deep Learning Methods 38

5.2 Equal Revenue Auction 39

6 Software Documentation **50**

6.1 Development Environment 50

6.2 Hyperparameters 51

6.3 Code Overview 53
6.4 Code Details 54
6.5 Further Work 56

A Timing **59**

A.1 Support Size 59
A.2 Number of Items 59
A.3 Second-Price Auctions 61

Chapter 1

Introduction

1.1 Problem Background

Consider someone who wishes to sell m items to n potential buyers. She may decide to host an auction to sell her items. There are many different ways to structure such a sale. If she is selling art, she might host an *open-outcry ascending-price auction* for each item, where potential buyers publicly call out bids and the highest bidder wins. If she is Google and needs to auction off an advertisement slot, she essentially hosts a *sealed-bid second-price auction*, where advertisers submit private bids and the highest bidder wins the slot but only pays the second-best price. In either case, the seller—or *auctioneer*—wishes to maximize her revenue. How should she structure her auction?

In this thesis, we restrict ourselves to additive sealed-bid auctions. These are auctions in which (1) a bidder's valuation for a set of items is the sum of her values for each item in the set, and (2) bidders cannot communicate and do not see each other's bids or valuations for items. When there is only one item, the optimal mechanics—the best way to structure an auction—are well-known [Mye81]. But even with just two identical items, things are hard. The best mechanism might necessitate bundling the items together or using a stochastic rule for allocation [HR15]. The difficulty in solving these problems has led auction theorists to consider computational approaches [Düt+17; RJW20]. When the valuation distributions that bidders have for items have finite support, it is easy to pose the revenue-optimization problem as a linear program, though one with an exponential number of variables. Using reduced forms makes this program tractable in certain cases [CDW11]. However, there is no such equivalence when bidder valuation distributions are continuous.

1.2 Results

1.2.1 Cara

In this thesis, we present Cara, a software program that approximates the optimal revenue of additive sealed-bid auctions. The program takes as input a continuously-valued auction setting and desired parameters for approximation. Cara then tries to lower bound the optimal revenue in two ways:

1. Cara randomly takes a number of finite-support item valuation distributions that are stochastically-dominated (see Definition 3) by the original distribution and solves for their optimal revenues.

2. Cara considers the revenue of two specific *second-price auctions*, namely bundling all items together in one second-price auction (*grand bundle*) as well as selling the items separately in m different second-price auctions (*selling separate*).

The first method uses the idea that most auction settings are *revenue monotonic*—when buyers value items higher, the auctioneer can make more money (Definition 4). We will sometimes refer to this as our *working hypothesis* throughout the thesis. It is well-known that the class of auctions we consider are not revenue monotonic in general; sometimes lowering bidders' valuations can actually increase auction revenue. However, for a variety of reasons that we discuss in Chapter 3, we think it is a reasonable assumption to use lower-valued auctions as an approximate lower bound on optimal revenue.

For the second method, since bundling and selling separate are possible ways to structure an auction, the best auction can only be an improvement. The uncertainty with Cara's results here is that we approximate the second-price auctions by sampling item valuation distributions and simulating the two auctions.

In addition to returning the best revenue it can find, Cara returns a variety other data, such as the valuation distributions of each discretized trial and confidence interval data about the second-price auctions.

1.2.2 Equal Revenue Auction

After creating Cara, we use it to approximate one auction setting in which we are particularly interested. Let the *equal revenue distribution* or *equal revenue curve* be the distribution with cumulative distribution function (CDF) of $F(x) = 1 - 1/x$ for $x \geq 1$ and $F(x) = 0$ for $x < 1$. Then we are interested in the optimal revenue when n independent and identically distributed (i.i.d.) bidders draw values for two items from the equal revenue distribution, independently across items.

We call this the *equal revenue auction setting* with n bidders. Specifically, we want to know the asymptotic behavior of the optimal revenue as n diverges. Previous research shows that it is $2n + \Omega(\log n)$ and $2n + \mathcal{O}(\sqrt{n})$.[1]

Using Cara, we empirically study the optimal revenue. Cara can match but not exceed the known $2n+\Omega(\log n)$ lower bound, but the results in Chapter 5 may provide slight evidence that the equal revenue auction setting is precisely $2n+\Theta(\log n)$, or at least $2n+o(\sqrt{n})$. This research additionally serves as a general testing ground for Cara.

1.3 Related Work

As we discuss in Chapter 3, the optimal revenue problem for finite-support auctions can be posed as a linear program, made tractable in certain cases by reduced form. This follows from the work of [CDW11]. Since the proof is constructive, a previous Princeton student took this paper and created RoaSolver, a program that solves these auctions [Shu19]. We use RoaSolver to solve the discretized trials that Cara creates. A user familiar with RoaSolver will be pleased to see that the input specifications for Cara remain similar.

There have been other computational approaches to finding the optimal revenue of multi-item auctions in the continuous setting. [Düt+17] and [RJW20] model auction settings as a multi-layer neural network and the optimal auction design as a constrained learning problem. In Chapter 5, we compare Cara's performance to these deep neural nets on a couple of simple auction settings.

1.4 Roadmap

- Chapter 2 introduces the auction format considered in the thesis and formalizes notation. It is written simply for a reader not familiar with any auction theory.
- Chapter 3 reviews related background theory for Cara. We discuss results in reduced forms that making solving finite-support auctions possible. Additionally, we prove a few simple results about revenue monotonicity and discuss our working hypothesis.
- Chapter 4 explains how to use Cara. A reader interested just in using the program may skim Chapter 2 and head directly to this chapter.
- Chapter 5 describes attempts at using Cara to solve the equal revenue auction. Additionally, it compares the results of Cara against state-of-the-art deep learning models.
- Chapter 6 documents the engineering behind Cara and overviews the codebase.

[1] See *here* for definitions of asymptotic notation.

- Appendix A gives timing data for inputs to Cara. This data may prove useful to a user; much of the difficulty in using Cara is about figuring out how to get the best revenue approximation within time constraints.

Chapter 2

Auctions

In this chapter, we formally introduce the main auction structure that will be considered throughout the thesis. We explain everything simply so that a reader without any background in auction theory may understand the material. A more familiar reader may skip this section, though it is nice to skim through in order to familiarize oneself with the notation. The material here follows closely to [Shu19], which in turn is drawn from [CDW11].

An *auction* is a structure where bidders compete to buy items by submitting bids. The auctions we are concerned with throughout this paper are different from the open-outcry ascending-price auction mentioned in Chapter 1. Here are some of the general properties of the *auction structure* we will consider:

- There may be multiple items.
- Bidders do not know how others value the items.
- Bidders submit a singular bid for each item.
- The bids are private.
- Items may be sold separately or in groups.
- Items do not necessarily go to the highest bidder.
- Bidders may pay whatever the auctioneer decides.

What we have described above is the *auction structure*. One choice of bidders and items defines an *auction setting*. For example, the equal revenue auction setting with n bidders described in Chapter 1 describes one auction setting. Within an auction setting, it is up to the auctioneer to figure out the rules—or *mechanism*—for how to distribute items and charge bidders. These rules

along with an auction setting define a single *auction*. These differences will make more sense as we formalize our language below.

2.1 Auction Structure

2.1.1 Bidders, Items, and Types

There are n bidders and m items. We label each bidder and item with a number so the bidders are $\mathcal{B} = \{1, 2, \ldots, n\}$ and the items $\mathcal{I} = \{1, 2, \ldots, m\}$. Then we can refer to a specific bidder, for example, by saying something like "bidder b" for some $b \in \mathcal{B}$.

Before the auction begins, each bidder determines her true value for the items. We can think of each of these values as the utility the bidder gains from receiving the respective item. Together, we call a bidder's m values a *type* or a *valuation* $x = (x_1, x_2, \ldots, x_m) \in \mathbb{R}_+^m$. Each x_i is the value that a bidder with type x gives to item i. The set of possible types is $\mathcal{T} \subset \mathbb{R}_+^m$, which we call the *type collection*.[1] The bidder's valuations are *additive*, which means that her valuation for a set of items is the sum of her valuation for each of the individual items in the set; a bidder with type x's valuation for a set $L \subset \mathcal{I}$ of items is $\sum_{i \in L} x_i$.

Each bidder has her own *type distribution* over $\mathcal{T}$, which says how likely she is to draw each type. We call this distribution D_b for bidder b and write $x \sim D_b$ to say that x is drawn from D_b. Since a type determines values for all items, we note that a bidder's valuations are not necessarily independent across items.

Before the auction begins, each bidder independently draws her *true type* from her predetermined type distribution. This determines the *type profile*, which we denote $\boldsymbol{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)}) \in \mathcal{T}^n$. The type profile specifies the type of each bidder, and we may say that it is drawn from the distribution $\boldsymbol{D} = \bigtimes_{b \in \mathcal{B}} D_b$; types are drawn independently across bidders.

Sometimes we want to hold one bidder constant, motivating the definition

$$\boldsymbol{D}_b(x) = D_1 \times \cdots \times D_{b-1} \times x \times D_{b+1} \cdots \times D_n$$

for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$. This is the distribution where $\boldsymbol{x} \sim D_b(x)$ is a type profile in which bidder b has type x and all other bidders draw types normally from their type distributions.

Note that we put no restrictions on the size of $\mathcal{T}$. When we consider the revenue-optimization problem as a linear program, we will restrict ourselves to the case when $|\mathcal{T}| < \infty$. However, in the continuous auctions that we will consider, there are an uncountable number of types.

[1] We use "$\subset$" to include equality, and we define $\mathbb{R}_+ = \{x \in \mathbb{R} : x \geq 0)$

2.1.2 Bidding, Allocating, and Paying

After drawing types, each bidder submits a single *sealed bid* in the form of a *reported type*; a bidder's bid is just a type $x \in \mathcal{T}$. The bids are private, known only to the bidder and auctioneer. Furthermore, bidders are not required to report their true type. From the bidding, the auctioneer receives a *reported type profile* $\boldsymbol{x} \in \mathcal{T}^n$.

After receiving the reported type profile, the auctioneer needs to allocate the items and collect payment. First, let us consider the *allocation*. Let $\mathcal{B}_\perp = \mathcal{B} \cup \{\perp\}$, where we will let $\perp$ mean "no one." Then we can think of an allocation of items as $\boldsymbol{b} = (b_1, b_2, \ldots, b_m) \in \mathcal{B}_\perp^m$. Index i of the vector tells us that item i is allocated to bidder $b_i \in \mathcal{B}_\perp$, which may be no one if $b_i = \perp$. An *allocation distribution* is a distribution over $\mathcal{B}_\perp^m$. We will only consider allocation distributions that are independent across items. Let $\mathcal{A}$ be the set of all such distributions.

The auctioneer must choose an allocation distribution given a reported type profile. We call such a function an *ex-post allocation rule*, and we write it as $\boldsymbol{\pi} : \mathcal{T}^n \to \mathcal{A}$. We say that $\pi_{bi}(\boldsymbol{x})$ is the probability that bidder b gets item i given a reported type profile $\boldsymbol{x}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\boldsymbol{x} \in \mathcal{T}^n$. Since the allocation distribution is independent across items, this suffices to fully characterize $\boldsymbol{\pi}$. So that the ex-post allocation rule outputs a valid allocation distribution, we must have

$$\pi_{bi}(\boldsymbol{x}) \geq 0 \text{ for all } b \in \mathcal{B},\ i \in \mathcal{I},\ \text{and } \boldsymbol{x} \in \mathcal{T}^n$$

and

$$\sum_{b \in \mathcal{B}} \pi_{bi}(\boldsymbol{x}) \leq 1 \text{ for all } i \in \mathcal{I} \text{ and } \boldsymbol{x} \in \mathcal{T}^n.$$

Note that the sum may be less than 1 because items need not be allocated.

The auctioneer must choose an ex-post allocation rule before the auction and disclose it to each bidder. The bidders, of course, may use this information before bidding. After the auctioneer receives the reported type profile, the auctioneer applies the ex-post allocation rule to get an allocation distribution, and then samples it to get the final allocation, $\boldsymbol{b} \sim \boldsymbol{\pi}(\boldsymbol{x})$.

The auctioneer must also decide how much each bidder pays. This takes the form of a *simple payment rule*, $\boldsymbol{s}$, specified by the n functions $s_b : \mathcal{T} \to \mathbb{R}_+$, where $s_b(x)$ gives the price b pays for reporting type x. As with the ex-post allocation rule, the auctioneer nominates a simple payment rule before the auction begins and the bidders may use this information to inform their bidding strategies.

This simple payment rule may seem odd; it does not depend on others' bids nor on the final allocation. After all, in most auctions in the real world, people pay for the items they get. Despite

this, it can be shown that using a complex payment rule, one that takes into account the full reported type profile as well as the final allocation, does not change the properties of the auction format [CDW11]. This is because each bidder has no information about the other participants' bids nor about the final allocation. Still, why bother with the simple payment rule instead? As we mentioned in Chapter 1, finding the optimal revenue for an auction setting when $|\mathcal{T}| < \infty$ is a linear programming problem. Using a simple payment rule reduces the number of variables required to describe the program and thus speeds up its runtime.

2.1.3 Summary

Here we summarize the auction structure. Both the bidders and the auctioneer know the following information before the auction begins:

- The number of bidders n and items m.
- The type collection $\mathcal{T} \subset \mathbb{R}_+^m$.
- The bidders' type distribution $\boldsymbol{D} = \bigtimes_{b \in \mathcal{B}} D_b$.

Together, we call these variables the *predefined constants* which specify the *auction setting*. Given these predefined constants, the auctioneer decides the *mechanism* of the auction, which is given by the following:

- The ex-post allocation rule $\boldsymbol{\pi}$, given by $\pi_{bi}(\boldsymbol{x})$ for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\boldsymbol{x} \in \mathcal{T}^n$.
- The simple payment rule $\boldsymbol{s}$, given by $s_b(x)$ for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$.

The bidders then see the mechanism of the auction setting, draw types from their type distributions, and submit a reported type. From here the auction is carried out without thought: the auctioneer applies the ex-post allocation rule, samples an allocation distribution to get the final allocation, and applies the simple payment rule to charge each bidders.

The auction setting along with a mechanism defines a single *auction*. Below we will discuss bidder and auctioneer behavior and define the *revenue* of an auction.

2.2 Auction Properties

2.2.1 Bidder Incentives

Each bidder wishes to maximize her value in an auction. Let us consider how much value bidder b gets out of a particular auction. Say she draws type $x \sim D_b$ and say the auction concludes with final allocation $\boldsymbol{b} = (b_1, b_2, \ldots, b_m) \in \mathcal{B}_\perp^m$. Let $\mathcal{I}_b$ be the items that bidder b receives, i.e. the indices of $\boldsymbol{b}$ that equal b. Then the value that bidder b receives from this auction is $\sum_{i \in \mathcal{I}_b} x_i - s_b(x')$,

where x' is the type she reports to the auctioneer. This is all the utility that bidder b gains from the items she receives (remember that her valuations are additive) minus the cost she incurs for participating in the auction. So far, there is no guarantee that this number is positive. The auctioneer may give her no items and charge her exorbitant prices.

But bidders cannot calculate this value before they submit their bid. They do not know each other's reported types. Thus it makes sense to calculate a bidder's *expected payoff*. This is the value of the items she receive minus the cost she pays, in expectation over the type distributions and allocation distribution, assuming all other bidders report their true type. When bidder b's true type is x and reported type is x', we define her expected payoff as

$$\begin{aligned} J_b(x, x') &= \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \boldsymbol{D}_b(x')} \left[\mathop{\mathbb{E}}_{\boldsymbol{\pi}(\boldsymbol{x})} \left[\sum_{i \in \mathcal{I}_b} x_i \right] \right] - s_b(x') \\ &= \mathop{\mathbb{E}}_{\boldsymbol{x} \sim \boldsymbol{D}_b(x')} \left[\sum_{i \in \mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right] - s_b(x') \\ &= \int_{\substack{\boldsymbol{x} \in \mathcal{T}^n \\ \boldsymbol{x}^{(b)} = x'}} \left(\sum_{i \in \mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right) dP_{\boldsymbol{D}_b(x')} - s_b(x'), \end{aligned}$$

where $P_{\boldsymbol{D}_b(x')}$ is the probability measure induced from the distribution $\boldsymbol{D}_b(x')$. Quickly parsing this, the first line is the same as the value we described above at the beginning of Section 2.2.1, but taking the expectation over type distributions and the allocation distribution. Each subsequent line just writes out explicitly one expectation.

This notion of expected payoff is only reasonable, of course, if it makes sense to assume that other bidders will bid their true type. This motivates the following definition.

2.2.2 Incentive Compatibility

Definition 1. An auction is *Bayesian incentive compatible*, or *BIC*, if it satisfies:

$$J_b(x, x) \geq J_b(x, x') \text{ for all } b \in \mathcal{B} \text{ and } x, x' \in \mathcal{T}. \tag{1}$$

This is satisfied just in case there is a Bayesian Nash equilibrium when every bidder behaves truthfully.

When an auction is BIC, each bidder has no incentive to bid other than her true type, assuming that other bidders are also truthful. Thus every participant behaving truthfully is a stable equilibrium. Throughout this thesis, we will restrict ourselves to auctions that are BIC.

Additionally, there is one other constraint on auctions that we would like to satisfy.

Definition 2. An auction is *individually rational*, or *IR*, if $J_b(x, x) \geq 0$ for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$.

An auction that is IR insures that bidders willingly participate. It prohibits the auctioneer from setting arbitrarily high prices. When an auction is both BIC and IR, we will just say that it is *incentive compatible* (IC).

In Chapter 1, we briefly mentioned the sealed-bid second-price auction. For the sake of parsing these definitions, we will quickly prove the following:

Proposition 1. *Sealed-bid second-price auctions are BIC and IR.*

Proof. The second-price auction is clearly IR because a bidder may never pay more than her bid. Consider bidder b and item i. Let y_i be the best bid for i submitted by another bidder.

If $y_i > x_i^{(b)}$, then bidder b will not get i nor will she pay anything for it if she bids truthfully. If she bids $x_i' \neq x_i^{(b)}$, then she will either not win and not pay, or win and pay y_i, losing $x_i^{(b)} - y_i$ in utility. So it is best of she bids truthfully.

If $y_i \leq x_i^{(b)}$, then bidder b will win item i and gain $x_i^{(b)} - y_i$ in utility if she bids truthfully. If she bids $x_i' \neq x_i^{(b)}$, then she will either not win nor gain any utility if $x_i' < y_i$, or she will win and still gain exactly $x_i^{(b)} - y_i$ in utility if she bids $x_i' \geq y_i$. In either case, it is best if she bids truthfully.

This holds for all items $i \in \mathcal{I}$. Then since it is best for bidder b to submit her true type no matter how others bid, it clearly is best for her to submit her true type in expectation over others bidding truthfully. This holds for all $b \in \mathcal{B}$. □

We note that it is up to the auctioneer, given the auction setting, to specify a mechanism $(\boldsymbol{\pi}, \boldsymbol{s})$ so that the auction is incentive compatible.

2.2.3 Auctioneer Incentives

The auctioneer also has incentives. She is trying to maximize the revenue of the auction. If $\boldsymbol{x} \in \mathcal{T}^m$ is the reported type profile, then $\sum_{b \in \mathcal{B}} s_b(\boldsymbol{x}^{(b)})$ is her revenue, or the total amount she charges the bidders. Similarly to a bidder's expected payoff, we define the *revenue* of an auction to be the total amount charged by the auctioneer in expectation over the type distributions and allocation distribution, assuming all bidders report their true type. The revenue is

$$\begin{aligned}\mathrm{Rev}_{(\boldsymbol{\pi},\boldsymbol{s})}(\boldsymbol{D}) &= \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\boldsymbol{D}}\left[\sum_{b\in\mathcal{B}} s_b(\boldsymbol{x}^{(b)})\right] \\ &= \int_{\boldsymbol{x}\in\mathcal{T}^n}\left(\sum_{b\in\mathcal{B}} s_b(\boldsymbol{x}^{(b)})\right) dP_{\boldsymbol{D}},\end{aligned}$$

where $P_{\boldsymbol{D}}$ is the probability measure induced from $\boldsymbol{D}$. Once again, note that this definition of revenue does not make much sense unless the auctioneer ensures the auction is incentive compatible.

Notationally, we let $\boldsymbol{D}$ encompass all of the predefined constants of the auction setting—n, m, and $\mathcal{T}$ are left implicit.

Now the auctioneer is not a real person—she serves to clarify the auction format by personifying the mechanisms of the auction. In this thesis, we are concerned with taking the place of the auctioneer. Given an auction setting, we want to find a mechanism that maximize the revenue, subject to the auction being incentive compatible. We say that the *optimal revenue* of an auction setting is the supremum of the revenue over all incentive compatible mechanisms. More precisely, the optimal revenue of an auction setting is

$$\operatorname{Rev}(\boldsymbol{D}) = \sup_{\text{IC } (\boldsymbol{\pi},\boldsymbol{s})} \operatorname{Rev}_{(\boldsymbol{\pi},\boldsymbol{s})}(\boldsymbol{D}).$$

Chapter 3

Background Theory

In this chapter, we introduce background theory relevant to Cara. A reader interested in just using the program or in understanding its engineering has no need for this section and may skip to Chapter 4. Most of the material in this chapter is drawn from [CDW11] and [HR15].

In Section 3.1, we discuss how the revenue optimization problem of an auction setting with a finite type collection can be posed as a linear program. We then overview further results that reduce the number of variables and constraints to make the the linear program tractable in certain cases.

In Section 3.2, we prove a couple of simple results on revenue monotonicity in the single bidder setting. As discussed in Chapter 1, Cara uses the working hypothesis that most auction settings are revenue monotonic—when buyers value items higher, the auctioneer can make more money. While none of the results proved in this section strictly inform Cara, we think that this section provides insight on why we think our working hypothesis is a good one and is interesting in its own right.

3.1 Linear Program

3.1.1 Initial Linear Program

Maximizing the revenue of an auction setting when the type distributions (bidder valuation distributions for items) has finite support ($|\mathcal{T}| < \infty$) is a linear programming problem. Consider this situation and let $c = |\mathcal{T}|$. Then we can characterize the type distribution $\boldsymbol{D} = \bigtimes_{b \in \mathcal{B}} D_b$ through the variables p_{bx} for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$, where p_{bx} tells us the probability with which bidder b draws type x.

The naive linear program is easy to set up. The unknown is the mechanism—the allocation rule

$\boldsymbol{\pi}$ and payment rule $\boldsymbol{s}$. The constraints are the BIC and IR inequalities. The objective function is the revenue.

More specifically, given the predefined constants $n, m, c, \{x_i\}_{i\in\mathcal{I},x\in\mathcal{T}}, \{p_{bx}\}_{b\in\mathcal{B},x\in\mathcal{T}}$ of an auction setting, we define the following variables:

- $\pi_{bi}(\boldsymbol{x})$ for all $b \in \mathcal{B}, i \in \mathcal{I}$ and $\boldsymbol{x} \in \mathcal{T}^n$. These variables specify the ex-post allocation rule. (nmc^n variables.)
- $s_b(x)$ for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$. These variables specify the simple payment rule. (nc variables.)

We impose the following $nmc^n + mc^n$ constraints to ensure that $\boldsymbol{\pi}$ specifies valid probabilities, remembering that items need not be allocated:

$$\pi_{bi}(\boldsymbol{x}) \geq 0 \text{ and } \sum_{b\in\mathcal{B}} \pi_{bi}(\boldsymbol{x}) \leq 1 \text{ for all } b \in \mathcal{B}, i \in \mathcal{I}, \text{ and } \boldsymbol{x} \in \mathcal{T}^n$$

The next nc constraints ensure that the auction is IR:

$$\sum_{\substack{\boldsymbol{x}\in\mathcal{T}^n \\ \boldsymbol{x}^{(b)}=x}} \left(\prod_{\substack{b'\in\mathcal{B} \\ b'\neq b}} p_{b'\boldsymbol{x}_{b'}} \cdot \sum_{i\in\mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right) - s_b(x) \geq 0 \text{ for all } b \in \mathcal{B},\ x \in \mathcal{T}$$

The final nc^2 constraints ensure that the auction is BIC:

$$\sum_{\substack{\boldsymbol{x}\in\mathcal{T}^n \\ \boldsymbol{x}^{(b)}=x}} \left(\prod_{\substack{b'\in\mathcal{B} \\ b'\neq b}} p_{b'\boldsymbol{x}_{b'}} \cdot \sum_{i\in\mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right) - s_b(x) \geq \sum_{\substack{\boldsymbol{x}\in\mathcal{T}^n \\ \boldsymbol{x}^{(b)}=x'}} \left(\prod_{\substack{b'\in\mathcal{B} \\ b'\neq b}} p_{b'\boldsymbol{x}_{b'}} \sum_{i\in\mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right) - s_b(x')$$

for all $b \in \mathcal{B}$ and $x, x' \in \mathcal{T}$. The objective function to maximize is the expected revenue:

$$\sum_{\boldsymbol{x}\in\mathcal{T}^n} \left(\prod_{b\in\mathcal{B}} p_{b\boldsymbol{x}^{(b)}} \cdot \sum_{b\in\mathcal{B}} s_b(\boldsymbol{x}^{(b)}) \right)$$

While this linear program successfully finds the optimal revenue of any auction setting when $|\mathcal{T}| < \infty$, the number of variables and constraints grows exponentially with n.

3.1.2 Reduced Forms

A *reduced-form auction*, or a *reduced form* for short, is an auction that is specified by an *interim allocation rule* as opposed to an ex-post allocation rule. An interim allocation rule $\boldsymbol{\pi}^*$ is defined by $\pi^*_{bi}(x)$ for all $b \in \mathcal{B}$, $i \in \mathcal{I}$ and $x \in \mathcal{T}$, where $\pi^*_{bi}(x)$ specifies the probability that bidder b receives item i given that she reports type x. An interim allocation rule is simpler in that allocations for bidder b only depend on what she bids. Thus an interim allocation rule specifies nmc variables and constraints, as opposed to the nmc^n of an ex-post allocation rule.

Given an ex-post allocation rule $\boldsymbol{\pi}$, we say that it induces the interim allocation rule $\boldsymbol{\pi}^*$ when

$$\begin{aligned}\pi^*_{bi}(x) &= \mathop{\mathbb{E}}_{\boldsymbol{x}\sim D_b(x)}\left[\pi_{bi}(\boldsymbol{x})\right] \\ &= \sum_{\substack{\boldsymbol{x}\in\mathcal{T}^n \\ \boldsymbol{x}^{(b)}=x}} \left(\prod_{\substack{b'\in\mathcal{B} \\ b'\neq b}} p_{b'\boldsymbol{x}_{b'}} \cdot \pi_{b'i}(\boldsymbol{x}) \right)\end{aligned}$$

for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $x \in \mathcal{T}$. The induced rule π^*_{bi} specifies the probability that bidder b receives item i, in expectation over all other bidders behaving truthfully. When considering a bidder's expected payoff or the BIC or IR constraints, we only ever need the induced allocation rule. It is easy to see this by looking at the linear program written above. After all, we always consider situations in expectation over other bidders behaving truthfully. We say that an interim allocation rule is *feasible* if there is an ex-post allocation rule that induces it—that is, if it corresponds to a real auction mechanism.

In the linear program above, if we replace the ex-post allocation rule with the set of feasible interim allocation rules, we will be maximizing an equivalent objective function. The issue, however, is knowing when an interim allocation rule $\boldsymbol{\pi}^*$ is feasible. We use the term *projected interim allocation rule* for item i to refer to just the probabilities for that item, i.e. $\{\pi^*_{bi}(x)\}_{b\in\mathcal{B},x\in\mathcal{T}}$. [CDW11] shows that an interim allocation rule is feasible if and only if the projected rule is feasible for all $i \in \mathcal{I}$. They show that a projected rule for item i is feasible just in case we have

$$\sum_{b\in\mathcal{B}} \sum_{\substack{x\in\mathcal{T} \\ \pi^*_{bi}(x)\geq z_b}} \pi^*_{bi}(x)p_{bx} \leq 1 - \prod_{b\in\mathcal{B}} \left(1 - \sum_{\substack{x\in\mathcal{T} \\ \pi^*_{bi}(x)\geq z_b}} p_{bx} \right) \text{ for all } \boldsymbol{z} = (z_1, z_2, \ldots, z_n) \in \mathbb{R}^n.$$

For one choice of $\boldsymbol{z} \in \mathbb{R}^n$, we call the inequality above a Border constraint. The left side is the probability that item i is allocated to a bidder b with true type x satisfying $p_{bx} \geq z_b$, and the right side is the probability that such a bidder exists. Since we are required to check Border constraints for all $\boldsymbol{z} \in \mathbb{R}^n$, we may have to check up to c^n constraints. However, [CDW11] also proves that instead of having to check the Border constraints for all $\boldsymbol{z}^n$, it is sufficient to check

$$\sum_{b\in\mathcal{B}} \sum_{x\in S_{bi}(z)} \pi^*_{bi}(x)p_{bx} \leq 1 - \prod_{b\in\mathcal{B}} \left(1 - \sum_{x\in S_{bi}(z)} p_{bx} \right) \text{ for all } z \in \mathbb{R},$$

where we define the sets

$$S_{bi}(z) = \left\{ t \in \mathcal{T} : \pi^*_{bi}(x) \sum_{\substack{x'\in\mathcal{T} \\ \pi^*_{bi}(x)\geq\pi^*_{bi}(x')}} p_{bx'} \geq x \right\} \text{ for all } b \in \mathcal{B} \text{ and } i \in \mathcal{I}$$

as a function of $z \in \mathbb{R}$. It is only necessary to check the Border constraint at the threshold values

of z, which amounts to at most c values of z that need to be checked for each item i.

Finally, [CDW11] proves that given an auction setting, it is possible to create a *separation oracle* $\mathfrak{S}$ which determines whether an interim allocation rule is feasible in linearithmic time. If the inputted rule is not feasible, $\mathfrak{S}$ return a hyperplane ℓ which corresponds to a broken Border constraint.

3.1.3 Reduced Form Linear Program

We now rewrite the linear program described in Section 3.1.1 using reduced forms. Most of the difference is because $J_b(x, x')$—bidder b's expected payoff for reporting type x' while having type x—is now

$$\sum_{i \in \mathcal{I}} \pi^*_{bi}(x') x_i - s_b(x')$$

instead of

$$\sum_{\substack{\boldsymbol{x} \in \mathcal{T}^n \\ \boldsymbol{x}^{(b)} = x'}} \left(\prod_{\substack{b' \in \mathcal{B} \\ b' \neq b}} p_{b' \boldsymbol{x}_{b'}} \sum_{i \in \mathcal{I}} \pi_{bi}(\boldsymbol{x}) x_i \right) - s_b(x').$$

Specifically, given the predefined constants $n, m, c, \{x_i\}_{i \in \mathcal{I}, x \in \mathcal{T}}, \{p_{bx}\}_{b \in \mathcal{B}, x \in \mathcal{T}}$ of an auction setting, we define the following variables:

- $\pi^*_{bi}(x)$ for all $b \in \mathcal{B},\ i \in \mathcal{I}$ and $x \in \mathcal{T}$. These variables specify the interim allocation rule. (nmc variables.)
- $s_b(x)$ for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$. These variables specify the simple payment rule. (nc variables.)

The constraints are the following:

- $0 \leq \pi^*_{bi}(t) \leq 1$ for all $b \in \mathcal{B},\ i \in \mathcal{I}$, and $x \in \mathcal{T}$. These constraints ensure valid probabilities. ($2mnc$ constraints.)
- $\sum_{i \in \mathcal{I}} \pi^*_{bi}(x) x_i - s_b(x) \geq 0$ for all $b \in \mathcal{B}$ and $x \in \mathcal{T}$. These constraints ensure the auction is IR. (nc constraints.)
- $\sum_{i \in \mathcal{I}} \pi^*_{bi}(x) x_i - s_b(x) \geq \sum_{i \in \mathcal{I}} \pi^*_{bi}(x) x_i - s_b(x')$ for all $b \in \mathcal{B}$ and $x, x' \in \mathcal{T}$. These constraints ensure that the auction is BIC. (nc^2 constraints.)

The objective function is the same:

$$\sum_{\boldsymbol{x} \in \mathcal{T}^n} \left(\prod_{b \in \mathcal{B}} p_{b \boldsymbol{x}^{(b)}} \cdot \sum_{b \in \mathcal{B}} s_b(\boldsymbol{t}^{(b)}) \right)$$

Additionally, we add the separation oracle $\mathfrak{S}$ to ensure that we only consider feasible interim allocation rules. If $\boldsymbol{\pi}^*$ satisfies all other constraints and $\mathfrak{S}(\boldsymbol{\pi}^*) = \ell$, we add the broken constraint

ℓ to the linear program and iterate.

It is this reduced form linear program that RoaSolver solves. We discuss how Cara interacts with RoaSolver in Chapters 4 and 6.

3.2 Revenue Monotonicity

In Chapter 1, we described high level how Cara approximates the optimal revenue of an auction setting. Cara attempts to lower bound the revenue by finding the optimal revenue of finitely-valued auctions in which bidders have lower values for items. If our original continuous distribution is $\boldsymbol{D}$, then this amounts to finding a distribution $\boldsymbol{D'} = \bigtimes_{b \in \mathcal{B}} D'_b$ with finite support such that $\boldsymbol{D}$ first-order stochastically dominates $\boldsymbol{D'}$. Here are a few equivalent definitions of stochastic dominance:

Definition 3 (Stochastic dominance)**.** A distribution D_2 *stochastically dominates* D_1 if and only if

- $F_{D_1}(x) \leq F_{D_2}(x)$ for all $x \in \mathbb{R}$, where F denotes the CDF.
- There exists a coupling $(X_1, X_2) \sim (D_1, D_2)$ such that $X_1 \leq X_2$ pointwise in the sample space.
- $\mathbb{E}_{X \sim D_1}[u(X)] \leq \mathbb{E}_{X \sim D_2}[u(X)]$ for every nondecreasing real-valued u.

The problem is that multi-item auctions are not in general *revenue monotonic* in either of the following ways:

Definition 4 (Revenue monotonicity)**.** Suppose we have some number of bidders n and items m as well as a type collection $\mathcal{T}$. Let $(\boldsymbol{\pi}, \boldsymbol{s})$ be the mechanisms of an auction. Let $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ be distributions over $\mathcal{T}$ such that $\boldsymbol{D}_1$ is stochastically dominated by $\boldsymbol{D}_2$.

Then the *mechanism* $(\boldsymbol{\pi}, \boldsymbol{s})$ is revenue monotonic if

$$\mathrm{Rev}_{(\boldsymbol{\pi}, \boldsymbol{s})}(\boldsymbol{D}_1) \leq \mathrm{Rev}_{(\boldsymbol{\pi}, \boldsymbol{s})}(\boldsymbol{D}_2).$$

The setting $(n,\ m,\ \mathcal{T})$ is revenue monotonic if

$$\mathrm{Rev}(\boldsymbol{D}_1) \leq \mathrm{Rev}(\boldsymbol{D}_2).$$

The latter usage of revenue montonicity, that $\mathrm{Rev}(\boldsymbol{D}_1) \leq \mathrm{Rev}(\boldsymbol{D}_2)$, is the working hypothesis that RoaSolver uses.

In this section, we will only consider situations in which there is a single bidder ($n = 1$). Even here, we will show examples in which an auction is not revenue monotonic. But all is not to despair. These examples will show that non-monotonic auctions are generally weird. We will also prove

a couple of positive results, namely identifying two circumstances in which revenue monotonicity holds:

- When the mechanism is deterministic and symmetric.
- When the payment rule is submodular.

While we only consider the case when $n = 1$, we hope this section can provide some general intuition about revenue monotonicity.

3.2.1 Preliminaries

First, we review some notation. We will have one bidder with valuations $x = (x_1, x_2, \ldots, x_m) \in \mathcal{T} \subset R_+^m$ drawn from D. A mechanism is given similarly to Chapter 2, but adapted for one bidder. It is given by a pair (π, s) where $\pi : \mathcal{T} \to [0,1]^m$ is the allocation rule and $s : \mathcal{T} \to \mathbb{R}_+$ is the payment rule. The bidder payoff for reporting type x' when having valuations x is $J(x, x') = \pi(x') \cdot x - s(x')$. We will abuse notation and just write $J(x)$ when the bidder bids x truthfully, i.e. when $x' = x$. We use the same definitions of BIC and IR and say that a mechanism is incentive compatible (IC) when it is both BIC and IR. We similarly have $\text{Rev}_{(\pi,s)}(D) = \mathop{\mathbb{E}}_{x \sim D}[s(x)]$. and $\text{Rev}(D) = \sup_{\text{IC } (\pi,s)} \text{Rev}_{(\pi,s)}(D)$.

We call the range $M = \{(\pi(x), s(x)) : x \in \mathcal{T}\} \subset [0,1]^m \times \mathbb{R}_+$ of a mechanism (π, s) a *menu*. It is easy to see that giving the menu M of an IC mechanism and letting the bidder choose an option on the menu is equivalent to running the auction normally; the option she will choose is the same one she must be given under BIC conditions.

Finally, we will restrict ourselves to *seller-favorable* mechanisms, which are IC mechanisms for which it is not possible to increase the revenue without changing the bidder's payoff:

Definition 5 (Seller-favorable mechanism)**.** An IC mechanism (π, s) is *seller-favorable* if there does not exist another IC mechanism (π', s') such that

1. $s'(x) \geq s(x)$ for all $x \in \mathcal{T}$ with strict inequality for some x.
2. $\pi'(x) \cdot x - s'(x) = \pi(x) \cdot x - s(x)$ for all $x \in \mathcal{T}$.

When trying to maximize the revenue of an auction setting, it is clear that these are the only mechanisms that matter. Restricting ourselves to seller-favorable mechanisms helps with differentiability issues ([HR15] Appendix). In the case of mechanisms described by a menu, seller-favorability implies that when a buyer is indifferent between multiple options, the most expensive menu option is chosen.

In the rest of this chapter, when considering multiple type distributions, they will always be over

the same type profile $\mathcal{T}$. Additionally, always assume that we have $x, y \in \mathcal{T}$.

3.2.2 Monotonicity for One Item

Proposition 2. *Consider the case of one item ($m = 1$). When D_1 is stochastically-dominated by D_2, we have $\text{Rev}_{(\pi,s)}(D_1) \leq \text{Rev}_{(\pi,s)}(D_2)$ for all IC mechanisms (π, s).*

Proof. Fix an IC mechanism (π, s). First we will show that s is nondecreasing. BIC gives us the inequalities

$$\pi(y) \cdot y - s(y) \geq \pi(x) \cdot y - s(x) \text{ and } \pi(x) \cdot x - s(x) \geq \pi(y) \cdot x - s(y).$$

Moving things around, we have

$$(\pi(y) - \pi(x))y \geq s(y) - s(x) \geq (\pi(y) - \pi(x))x. \tag{1}$$

Subtracting the right side of Equation (1) tells us that $(\pi(y) - \pi(x))(y - x) \geq 0$. When $y > x$, then $\pi(y) - \pi(x) > 0$. Since $x \geq 0$, then the right side of (1) is nonnegative when $y > x$ and so $s(y) - s(x) \geq 0$ or $s(y) \geq s(x)$. Thus s is nondecreasing.

From our definition of stochastic dominance, we know then that $\mathop{\mathbb{E}}_{x \sim D_1}[s(x)] \leq \mathop{\mathbb{E}}_{x \sim D_2}[s(x)]$, which is precisely the desired result. □

It follows from Proposition 2 that revenue monotonicity holds for the supremum, i.e. that $\text{Rev}(D_1) \leq \text{Rev}(D_2)$.

3.2.3 Non-Monotonicity Examples

Example 1. Consider an auctioneer who offers the following menu for two items:

$\pi(x_1, x_2)$	$s(x)$
(0, 0)	0
(1, 0)	1
(0, 1)	2

It is easy to see that this auction mechanism is not revenue monotonic. When the bidder's valuations are $(0, 3)$, she will take the second item and pay 2. But if her valuation increases to $(3, 3)$, she will buy the first item and pay 1. Even if both of her valuations increase, the auctioneer may still lose money. Say her valuations increase to $(4, 4)$, once again she will take the first item and the auctioneer will only collect 1.

The issue here is that the auctioneer is offering a cheap option (item 1) and an expensive option (item 2). When the bidder's valuation for the cheap option increases enough relative to the

expensive option, she will switch from the expensive choice to the cheap one, yielding less revenue for the auctioneer. As with all of our non-monotonic examples, they cannot be deterministic and symmetric or submodular. In this toy example, it is easy to see how being deterministic and symmetric or submodular would fix the issues (see definitions in Section 3.2.4). If the menu were symmetric, then there would not be a cheap option. If the menu were submodular, then the bundle of items would be offered for an affordable price, mitigating the loss of switching to the cheap option.

One might say that this is just a silly mechanism. The more interesting question is whether the optimal revenue might be non-monotonic in a given auction setting. Here we give a slightly more involved example, which is from [HR15].

Example 2. For $0 \leq \alpha \leq 1/12$, let D_α be the type distribution over two items given by

$$D_\alpha = \begin{cases} (1,1) & \text{with probability } 1/4 \\ (1,2) & \text{with probability } 1/4 - \alpha \\ (2,2) & \text{with probability } \alpha \\ (2,3) & \text{with probability } 1/2 \end{cases}.$$

It is clear that D_{α_2} stochastically dominates D_{α_1} for $\alpha_2 > \alpha_1$; we are just moving some mass away from $(1,2)$ towards $(2,2)$ as we increase α. However, we actually have $\text{Rev}(D_{\alpha_1}) > \text{Rev}(D_{\alpha_2})$.

We claim that for all $0 \leq \alpha \leq 1/12$, the optimal mechanism is given by the following menu:

$\pi(x_1, x_2)$	$s(x)$
(0, 0)	0
(1, 0)	1
(0, 1)	2
(1, 1)	4

Remembering that ties go to the seller, the revenue of this mechanism for D_α is

$$\frac{1}{4}s(1,1) + \left(\frac{1}{4} - \alpha\right) s(1,2) + \alpha s(2,2) + \frac{1}{2}s(2,3)$$

or

$$\left(\frac{1}{4}\right) \cdot 1 + \left(\frac{1}{4} - \alpha\right) \cdot 2 + \alpha \cdot 1 + \left(\frac{1}{2}\right) \cdot 4 = \frac{11}{4} - \alpha,$$

where we just multiply the probability of the bidder drawing a given type by the revenue that that type raises.

It is not hard to show that this is the maximum revenue achievable for D_α, but neither is it very enlightening. Taking a system of six inequalities derived from IR and BIC constraints, we fiddle

with them to get the following:

$$-\left(\frac{3}{4}-3\alpha\right)\pi_2(1,1)-2\alpha\pi_1(1,2)+\left(\frac{1}{4}-3\alpha\right)\pi_2(1,2)+2\alpha\pi_1(2,2)+\pi_1(2,3)+\frac{3}{2}\pi_2(2,3)$$
$$\geq \frac{1}{4}s(1,1)+\left(\frac{1}{4}-\alpha\right)s(1,2)+\alpha s(2,2)+\frac{1}{2}s(2,3).$$

The right side is exactly the revenue of our mechanism. Setting π's to 0 or 1 in order to maximize the left side, we see that it is

$$\left(\frac{1}{4}-3\alpha\right)+2\alpha+1+\frac{3}{2}=\frac{11}{4}-\alpha.$$

Thus this is the maximum revenue achievable. A reader interested in the exact inequalities should read Proposition 3 in [HR15].

Once again, the issue is that the buyer switches from an expensive option (item 2) to a cheap option (item 1) as her valuations increase from $(1,2)$ to $(2,2)$. It is easy to see why being submodular would fix this issue; she would instead switch to the bundle and the auctioneer would not lose money.

3.2.4 Classes of Monotonic Mechanisms

In this section, we prove two different classes of mechanisms that are revenue monotonic. First, mechanisms that are deterministic and symmetric. Second, mechanisms that have a submodular payment rule.

Before we begin with the first class, let us define exactly what we mean by deterministic and symmetric. A mechanism (π, s) is deterministic if its menu is a subset of $\{0,1\}^m \times \mathbb{R}_+$. Incentive compatibility implies that each allocation in $\{0,1\}^m$ has a unique price: if $\pi(x) = \pi(y)$, then $s(x) = s(y)$. If this were not the case and say $s(x) > s(y)$, then a bidder with type x would rather report y and BIC would be violated. Thus a deterministic menu has at most 2^m options—one for each subset of the items $\mathcal{I}$, and we can think of the payment rule as just specifying a price for each available subset of items. We abuse notation and just write s_I for some set of items $I \subset \mathcal{I}$.

A deterministic mechanism is *symmetric* if all the goods are treated identically: if the prices of a subset of items only depends on the number of items, not which precise ones. This implies that the bidder's payoff function J is invariant under permutation. Additionally, this means that the mechanism can be fully specified by an auctioneer simply posting a price for the number of items that the bidder wishes to buy. We abuse notation again and let s_k be the price for the bidder to buy $0 \leq k \leq m$ items.[1] It will always be clear how we are using s.

Theorem 1. *Let D_1 be a distribution over $\mathcal{T}$ in which there exists a deterministic and symmetric*

[1] If a certain number of items is not available, we can just set $s_k = \infty$.

IC mechanism that achieves the optimal revenue. Then we have $\text{Rev}(D_1) \leq \text{Rev}(D_2)$ *whenever* D_2 *stochastically dominates* D_1*.*

For the proof of Theorem 1, we will need the mechanism to be seller-favorable. While there is always a seller-favorable optimal mechanism, it is not trivial to show that there is one that is deterministic and symmetric. Thus we will use the following proposition without proof.

Proposition 3 ([HR15] Proposition 16, Corollary 6)**.** *Let* D *be a distribution over* $\mathcal{T}$ *in which there exists a deterministic and symmetric IC mechanism that achieves the optimal revenue. Then there is a seller-favorable deterministic and symmetric IC mechanism that achieves the optimal revenue.*

Proof of Theorem 1. Using Proposition 3, let (π, s) be an optimal seller-favorable deterministic and symmetric IC mechanism for D_1. First we will show that s is nondecreasing. Similarly to the proof of Proposition 2, the result then follows from the definition of stochastic dominance.

Let $x, y \in \mathcal{T}$ satisfy $x < y$. Without loss of generality, assume that the vectors x and y are both in nonincreasing order (i.e. $x_1 \geq x_2 \geq \dots$), which we can do because the mechanism is symmetric. Define

$$J_i(x) = x_1 + \cdots + x_i - s_i \text{ and } J_i(y) = y_1 + \cdots + y_i - s_i,$$

which are the best payoffs for i items that x and y can receive, respectively. Let α and β be the number of items that are sold to types x and y, respectively. Then $J(x) = J_\alpha(x)$ and $s(x) = s_\alpha$, and likewise for a bidder with type y. We want to show that $s(x) \leq s(y)$, or $s_\alpha \leq s_\beta$. There are two cases.

If $\beta \geq \alpha$, we can write

$$\begin{aligned} s_\beta - s_\alpha &= (-x_{a+1} - \cdots - x_\beta - s_\alpha - (-s_\beta)) + (x_{\alpha+1} + \cdots + x_\beta) \\ &= J_\alpha(x) - J_\beta(x) + (x_{\alpha+1} + \cdots + x_\beta). \end{aligned}$$

The BIC constraint tells us that $J_\alpha(x) \geq J_\beta(x)$. Since $x \geq 0$, then we have that $s_\beta - s_\alpha \geq 0$ or $s_\beta \geq s_\alpha$.

If $\beta < \alpha$, instead we write

$$\begin{aligned} J_\alpha(y) - J_\beta(y) &= y_{\beta+1} + \cdots + y_\alpha - s_\alpha - (-s_\beta) \\ &= (x_{\beta+1} + \cdots + x_\alpha - s_\alpha - (-s_\beta)) + (y_{\beta+1} - x_{\beta+1}) + \cdots + (y_\alpha - x_\alpha) \\ &= J_\alpha(x) - J_\beta(x) + (y_{\beta+1} - x_{\beta+1}) + \cdots + (y_\alpha - x_\alpha). \end{aligned}$$

Since $x \leq y$, we must have that $J_\alpha(y) - J_\beta(y) \geq 0$ or $J_\alpha(y) \geq J_\beta(y)$. But we know that $J_\alpha(y) \leq J_\beta(y)$ from the BIC constraint so we must have $J_\alpha(y) = J_\beta(y)$. Thus both sets of size α and β

maximize y's revenue. The seller-favorable conditions tells us that $s_\beta \geq s_\alpha$ because β is the size of the set that is actually sold to type y. Thus we have shown that s is nondecreasing.

Then we have

$$\text{Rev}(\text{D}_1) = \text{Rev}_{(\pi,s)}(D_1) = \mathbb{E}_{x\sim D_1}[s(x)] \leq \mathbb{E}_{x\sim D_2}[s(x)] = \text{Rev}_{(\pi,s)}(D_2) \leq \text{Rev}(D_2).$$

The first equality comes from assumption and Proposition 3, the second by definition, the third by the fact that s is nondecreasing and the definition of stochastic dominance, the fourth by definition, and the fifth because (π, s) is only one candidate mechanism for the optimal revenue of D_2. □

We will now turn to prove a similar result to Theorem 1 but for mechanisms that have a submodular payment rule. We say that a deterministic mechanism (π, s) is *submodular* when it satisfies $s_I + s_J \geq s_{I\cup J} + s_{I\cap J}$ for all sets of items $I, J \subset \mathcal{I}$, where we remember that s_I is the price that the deterministic mechanism assigns to the set I of items. Submodularity is a generalization of subadditivity (we see that it obtains exactly when $I\cap J = \emptyset$) that captures the notion of diminishing returns, which is very natural in many domains.

We want to generalize submodularity to nondeterministic mechanisms. Consider an arbitrary mechanism (π, s). A *pricing function* $p : [0, 1]^m \to \mathbb{R}_+$ assigns a price to every possible allocation lottery $g \in [0, 1]^m$. A pricing function induces the menu $\{(g, p(g)) : g \in [0, 1]^m\}$ that has payoff $\sup_{g\in[0,1]^m} (g \cdot x - p(g))$ when the bidder has type x. We say that p is a *pricing function for the mechanism* (π, s) if they both have the same choices and payoffs for all $x \in \mathcal{T}$. This occurs when $p(\pi(x)) = s(x)$ for all $x \in \mathcal{T}$ and

$$J(x) = \pi(x) \cdot x - s(x) = \pi(x) \cdot x - p(\pi(x)) = \sup_{g\in[0,1]^m} (g \cdot x - p(g)) . \tag{2}$$

This restricts the price of every used lottery $\pi(x) \in \pi(\mathcal{T}) = \{\pi(x) : x \in \mathcal{T}\}$ just to $s(x)$, but leaves some freedom on other allocation lotteries.

Definition 6 (Submodular mechanism)**.** A mechanism (π, s) is submodular if it is has a payment function p that satisfies

$$p(g_1) + p(g_2) \geq p(g_1 \vee g_2) + p(g_1 \wedge g_2) \tag{3}$$

for all $g_1, g_2 \in \pi(\mathcal{T})$.[2]

Theorem 2. *Let D_1 be a distribution over $\mathcal{T}$ in which there exists a submodular IC mechanism that achieves the optimal revenue. Then we have* $\text{Rev}(D_1) \leq \text{Rev}(D_2)$ *whenever D_2 stochastically dominates D_1.*

Similarly to Theorem 1, we will need the mechanism to be seller-favorable. We use the following

[2]We let $\vee$ and $\wedge$ be the coordinatewise maximum and minimum of a vector, respectively.

proposition without proof.

Proposition 4 ([HR15] Corollary 20)**.** *Let D be a distribution over $\mathcal{T}$ in which there exists a submodular IC mechanism that achieves the optimal revenue. Then there is a seller-favorable submodular IC mechanism that achieves the optimal revenue.*

Proof of Theorem 2. Using Proposition 4, let (π, s) be an optimal seller-favorable submodular IC mechanism for D_1. First we will show that s is nondecreasing. The result then follows from the definition of stochastic dominance.

Let p be the pricing function for (π, s) that satisfies (3). Let $x, y \in \mathcal{T}$ satisfy $x \leq y$ and let $g_x = \pi(x)$ and $g_y = \pi(y)$ for convenience. Then we work to show s is nondecreasing, or that $p(g_x) \leq p(g_y)$. From (2), we can write

$$g_x \cdot x - p(g_x) \geq (g_x \wedge g_y) \cdot x - p(g_x \wedge g_y) \tag{4}$$

for x, where the inequality holds because $g_x \wedge g_y$ is one choice in the supremum over $[0,1]^m$. We can similarly write

$$g_y \cdot y - p(g_y) \geq (g_x \vee g_y) \cdot y - p(g_x \vee g_y) \tag{5}$$

for y. Because $0 \leq g_y \leq g_x \vee g_y$ and $0 \leq x \leq y$, replacing the y in (5) with x decreases the right-hand side more than the left, and we have

$$g_y \cdot x - p(g_y) \geq (g_x \vee g_y) \cdot x - p(g_x \vee g_y). \tag{6}$$

We can add (4) and (6) to yield

$$(g_x + g_y) \cdot x - p(g_x) - p(g_y) \geq (g_x \wedge g_y + g_x \vee g_y) \cdot x - p(g_x \wedge g_y) - p(g_x \vee g_y).$$

Using the general fact that $g_x + g_y = g_x \wedge g_y + g_x \vee g_y$, we can cancel out and multiply by -1 to get

$$p(g_x) + p(g_y) \leq p(g_x \wedge g_y) + p(g_x \vee g_y).$$

But because of the submodularity condition (3), this must be an equality and therefore so are all of the equations above. Looking at (6) now with an equality, we see that $g = g_x \vee g_y$ must be a maximizer for the payoff $J(y)$ of type y. Because (π, s) is seller-favorable, this alternative mechanism that uses $g_x \vee g_y$ can only be worse for the auctioneer, so we have

$$p(g_x \vee g_y) \leq p(g_y), \tag{7}$$

where we recall that $p(g_y) = p(\pi(y)) = s(y)$. Now we claim that

$$p(g_x) \leq p(g_x \vee g_y), \tag{8}$$

which we can see because

$$g_y \cdot y - p(g_y) \geq (g_x \vee g_y) \cdot y - p(g_x \vee g_y) \geq g_y \cdot y - p(g_x \vee g_y),$$

where the first inequality is precisely (5) and the second because $g_y \leq g_x \vee g_y$ and $0 \leq y$. Combining (7) and (8) finally gives us that s is nondecreasing:

$$s(x) = p(g_x) \leq p(g_x \vee g_y) \leq p(g_y) = s(y).$$

Then we have

$$\text{Rev}(D_1) = \text{Rev}_{(\pi,s)}(D_1) = \mathbb{E}_{x \sim D_1}[s(x)] \leq \mathbb{E}_{x \sim D_2}[s(x)] = \text{Rev}_{(\pi,s)}(D_2) \leq \text{Rev}(D_2),$$

for identical reasons to the proof of Theorem 1. □

3.2.5 Discussion

[HR15] also shows an example in which an auction setting with i.i.d. items is not revenue monotonic. This example does not provide intuition for what causes the lack of monotonicity. In fact, it provides intuition that simple auction settings are generally approximately monotonic; the example uses an exhaustive search to find a case where increasing the type distributions decreases the optimal revenue from 408,189,937/5,875,650 to 30,614,162,731/440,673,750, a decrease of 0.00027%.

Additionally, it is not hard to see that selling separate and grand bundling are both monotonic mechanisms. Then as far as they approximate the optimal revenue of an auction setting well, the auction setting must be close to revenue monotonic. For example, [HN17] shows that with two independent items, selling separate guarantees at least 50% of the optimal revenue in the worst case. When there are $k > 2$ items, selling separate yields at least a $c/\log^2 k$ fraction of the optimal revenue.

These, of course, are just theoretical worst bounds. In our testing of simple auction settings with Cara, we have yet to see a case when the program seems to overestimate the optimal revenue.

Chapter 4

Software Usage

In this chapter, we overview how to use Cara. The sections chronologically explain the steps needed to properly work with the program:

- Section 4.1 describes additional necessary terminology.
- Section 4.2 explains how to install software dependencies.
- Section 4.3 describes how to write an input text file.
- Section 4.4 explains how to execute Cara.
- Section 4.5 describes different possible output formats.

4.1 Additional Terminology

We use slightly different notation for the input specifications than described in Chapter 2. Specifically, we group types and bidders both into classes, motivated by ease of input as well as computational efficiency. We put a dot above variables that have been modified for bidder or type classes.

4.1.1 Type Classes

Instead of directly drawing types, bidders will first draw a *type class* and then draw a type from within the type class. We let $\dot{c}$ be the number of type classes, which is finite. Let $\dot{\mathcal{T}}$ be the set of all type classes, which we can label as $\{1, 2, \ldots, \dot{c}\}$. Each type class defines a fixed distribution over the items. Type classes are grouped in a way such that the distribution is *independent* across items. Thus a type class just associates a continuous univariate distribution with each of the m items. So if we let $\dot{d}_{i\dot{t}}$ be the distribution with which type class $\dot{t}$ values item i, we can think of a

bidder in type class $\dot{t}$ as drawing her type from $\bigtimes_{i \in \mathcal{I}} d_{i\dot{t}}$. We note that each bidder in type class $\dot{t}$ draws her valuations independently of other bidders.

Instead of having to input $\dot{c}m$ many distributions, we define a list of distributions that can be used by all the type classes. We let r denote the number of *continuous valuation distributions* and let ϕ_i be the ith continuous valuation distribution. Then each type class $\dot{t}$ essentially just assigns a certain ϕ_i for each of the m items.

Even though type classes define distributions that are independent across items, it is still possible for bidders to draw valuations dependently across items. This is because bidders do not have a fixed type class, but first draw their type classes according to a certain distribution. It is this uncertainty over the type classes that allows for dependence across items.

4.1.2 Bidder Classes

We think of each bidder as being part of a *bidder class*. There are $\dot{n}$ bidder classes, which we label as $\dot{\mathcal{B}} = \{1, 2, \ldots, \dot{n}\}$. We let $N_{\dot{b}}$ be the number of bidders in bidder class $\dot{b}$, which we call a *multiplicity*. A *bidder multiplicity profile* is a complete description, $(n_1, n_2, \ldots, n_{\dot{n}}) \in \bigtimes_{\dot{b} \in \dot{\mathcal{B}}} N_{\dot{b}}$, of multiplicities for each bidder class. Each bidder class specifies a distribution over the type classes, which we call $\dot{D}_{\dot{b}}$. This is defined by $p_{\dot{b}\dot{t}}$, which is the probability that bidder class $\dot{b}$ draws type class $\dot{t}$. Each bidder in bidder class $\dot{b}$ independently draws a type class from $\dot{D}_{\dot{b}}$. Thus bidder classes group together identical bidders.

Using bidder classes has a few advantages. First is ease of input. A user only needs to input a distribution over type classes per bidder class, as opposed to per bidder.

The second benefit is computational efficiency. [DW12] shows that there exists an optimal auction that treats identical bidders in the same way. This means that they are given the same probabilities in the ex-post allocation rule and assigned the same payments from the simple payment rule. Thus there exists an optimal auction where these rules are only defined per bidder class, as opposed to per bidder. This reduces the number of variables and constraints in the linear program, which speeds up its runtime.

Finally, because of the reasons above, RoaSolver can efficiently solve for multiple multiplicity profiles within the same auction setting. For example, imagine using Cara to approximate the equal revenue auction. Instead of needing a different input each time we vary the number of bidders, Cara may take as input multiple multiplicities for the one bidder class and solve for their optimal auctions simultaneously. This both makes inputting auction settings easier and speeds up computation.

4.2 Software Installation

Cara requires Python and a few Python packages to run. Additionally, RoaSolver needs Java and IBM's CPLEX library.

4.2.1 Cara Installation

The full repository exists on Github.[1] It can be downloaded from the website or cloned locally through the command:

```
git clone https://github.com/antonstengel/cara.git
```

For the rest of the thesis, we will assume that the user has navigated to the cloned directory, whose root is `cara/`. Whenever we refer to a file or directory from here on out, we will reference it in terms of `cara/`.

`cara/cara.py` is the script that executes Cara. Additionally, `cara/cara/` contains necessary modules and a RoaSolver JAR. The rest of the files and directories in `cara/` are not necessary for running the program. When running the program, a directory `temp/` will be created and text files will be written to and deleted from it.

4.2.2 Java Installation

RoaSolver requires Java to be installed, which can be done from the Java website.[2] Run the installer and follow the prompts. Once Java has been installed, check that it is working properly by opening a terminal and running "`java -version`".

For context, this is what the results look like on my device:

```
(thesis) antonstengel@Antons-MacBook-Air cara % java -version
openjdk version "11.0.16.1" 2022-08-12
OpenJDK Runtime Environment Temurin-11.0.16.1+1 (build 11.0.16.1+1)
OpenJDK 64-Bit Server VM Temurin-11.0.16.1+1 (build 11.0.16.1+1, mixed mode)
```

4.2.3 CPLEX Installation

RoaSolver uses *IBM ILOG CPLEX Optimization Studio v12.9 (CJ4Z5ML)* to solve the linear programs. The software is available from the IBM website using this[3] guide. Make sure to install the correct version of CPLEX. The software is free for students and faculty. Select the default

[1] https://github.com/antonstengel/cara
[2] https://www.java.com/en/download/
[3] https://www.ibm.com/support/pages/downloading-ibm-ilog-cplex-optimization-studio-v1290

installation, but note that *Microsoft Visual C++ 2015* is not required. If the CPLEX installation is not working properly, we recommend reading [Shu19], which gives more details on the installation process.

4.2.4 Python Installation

Cara uses a few Python packages. They are listed in `cara/requirements.txt` along with the version used at the time of development. It is doubtful that there will be dependency errors between these packages in future versions.

We describe one of the many ways of installing Python along with the required packages. We will install everything using conda, an open source package and environment management system. Conda can be installed from the Anaconda website.[4] Once the download is complete, open the graphical installer and follow the prompts. Verify that conda is installed by running "`conda --version`" in the terminal. It may be necessary to restart the terminal after the installation is complete.

If it is successful, conda should have installed the latest version of Python with it. Check that Python is properly installed with "`python3 --version`". It may work just to run `python` instead of `python3`. If everything is successful, create a new environment with "`conda create --name MYENV`" where `MYENV` is the name of your environment. Then activate your environment with "`conda activate MYENV`". The command line should show the active environment in parentheses. For example, my prompt shows the following:

```
(thesis) antonstengel@Antons-MacBook-Air cara %
```

From here, install the packages either manually or using `cara/requirements.txt`. To install the required packages easily, first install pip with "`conda install pip`". Then run "`pip install -r requirements.txt`" to install the necessary Python dependencies. To check that the proper packages are installed, run "`conda list`" to view the packages installed in the activated conda environment.

4.3 Input Specifications

4.3.1 Input Format

Cara takes as input a text file. Here is the format:

[4] `https://www.anaconda.com/products/distribution`

$$
\begin{array}{cccccccc}
\dot{n} & m & \dot{c} & r & t & a & & \\
p_{11} & \dots & p_{1\dot{c}} & N_1 & & & & \\
\vdots & \ddots & \vdots & \vdots & & & & \\
p_{\dot{n}1} & \dots & p_{\dot{n}\dot{c}} & N_n & & & & \\
d_{11} & \dots & d_{1\dot{c}} & & & & & \\
\vdots & \ddots & \vdots & & & & & \\
d_{m1} & \dots & d_{m\dot{c}} & & & & & \\
\phi_1 & L_1 & U_1 & T_1 & S_1 & P_{11} & \dots & P_{1*} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\phi_r & L_r & U_r & T_r & S_r & P_{r1} & \dots & P_{r*}
\end{array}
$$

Each asterisks corresponds to a variable length, unrelated to any other astericks. For example, "$*$" does not need to be the same integer in P_{1*} and P_{r*}, despite how the graphic above makes it seem. Additionally, Cara can parse any number of spaces between each variable, which can be useful for visual separation. Finally, underscores may be used within numerical values as a visual aid for thousand markers.

4.3.2 Input Variables

The variables shown above are the following:

- $\dot{n}$ is the number of bidder classes. It is a positive integer.
- m is the number of items. It is a positive integer.
- $\dot{c}$ is the number of type classes. It is a positive integer.
- r is the number of continuous valuation distributions. It is a positive integer.
- t is the number of discretized trials to run. It is a nonnegative integer.
- a is the number of second-price auction trials to run. It is a nonnegative integer.
- p_{ij} is the unnormalized probability with which bidder i chooses type class j. Each entry is a nonnegative float. The dimension of this matrix is $\dot{n} \times \dot{c}$.
- N_i is the possible bidder multiplicities for bidder i. A semicolon separates different numbers, and a tilde between two numbers takes the full range, inclusive of endpoints. For example, "`4;10;6~8`" gives the bidder multiplicities of $4, 10, 6, 7, 8$.
- d_{ij} is the valuation that type class j gives to item i. It is either a float—the valuation—or of the form "`Dk`" where k means that the value is drawn from the the kth valuation distribution. The dimension of this matrix is $m \times \dot{c}$.

- ϕ_i is the ith continuous valuation distribution. This is a CDF satisfying $\lim_{x \to L_i^+} \phi_i(x) = 0$ and $\lim_{x \to U_i^-} \phi_i(x) = 1$. I.e. it vanishes at its lower end and approaches 1 towards the upper end. It must be a string consisting of the characters "`0123456789+-*/()x`" with no spaces. Exponentiation is given by "`**`". Additionally, it is possible to input the inverse of the CDF, which we call the *probability point function* (PPF). This is done by appending a vertical bar "`|`" to ϕ_i along with the PPF, which uses the same set of characters above but with "`y`" instead of "`x`". For example, "`1-1/x|1/(1-y)`" is a valid input, as is just "`1-1/x`" or "`1-1/(x**(1/2))`". Inserting the PPF is not necessary, but it speed up the computation of the second-price auctions substantially (see Appendix A.3).

- L_i is the lower end of the support for continuous valuation distribution ϕ_i. It is a nonnegative float. In the examples of ϕ_i, we should have $L_i = 1$.

- U_i is the upper end of the support for continuous valuation distribution ϕ_i. It is either a float or one of "`inf`" or "`-inf`". In the examples of ϕ_i, we should have $U_i =$ `inf`.

- T_i is the type of discretization to use for continuous valuation distribution ϕ_i. It is a string. Valid discretization types are described below in Section 4.3.3.

- S_i is the number of desired point masses in the support of the discretized version of the continuous valuation distribution ϕ_i. It is a positive integer. Note that the total number of points in the overall support of the discretized auction setting is the sum of all the unique values from the discretized support of each of the ϕ_i.

- P_i are the parameters for discretization T_i, described below.

4.3.3 Discretization Options

Each discretization type T_i enumerated below describes how values in the support of the continuous valuation distribution ϕ_i are drawn. The masses are always assigned to the discretized support using the same algorithm, which yields the discretized distribution that is stochastically-dominated (Definition 3) by ϕ_i and closest-fitting to ϕ_i. This is done by collapsing the probabilities of ϕ_i leftward into each point mass. We note that L_i must be in the support for the discretized distribution to be stochastically-dominated by ϕ_i. Observe Figure 4.1 for a visual example.

We list the discretization options from simplest to most complex below. The functionality of "Fixed" and "Random" are both contained in "Random and Fixed," but the latter is clunkier so it is often nicer to use the former two.

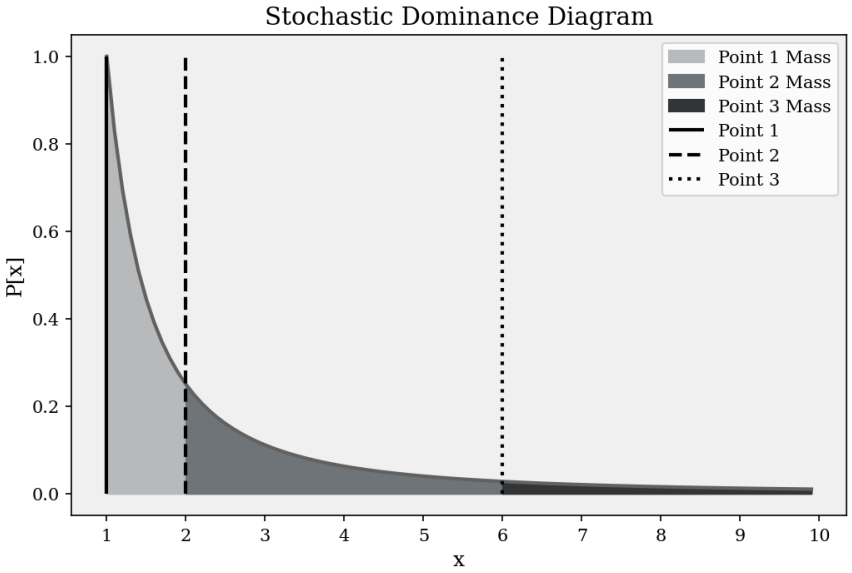

Figure 4.1: Suppose the distribution with CDF $1 - 1/x$ is discretized into a support with points at $x = 1,\ 2$, and 6. Then mass is assigned to each point by collapsing area to the left in the PDF.

N/A

This option is for no discretization, if the user only wishes to run the second-price auctions. It can be called with T_i equal to "`N/A`". It does not take any parameters, P_i, and it is not necessary to input S_i. The number of discretized trials, t, must be 0.

Fixed

This option simply takes a fully specified support as input. It can be called with T_i equal to "`fixed`" or "`f`". There are S_i parameters, each corresponding to a desired point in the support. Each parameter is a nonnegative float.

Random

This option randomly selects $S_i - 1$ points between $P_{i,1}$ and $P_{i,2}$ as well as one point at L_i. It can be called with T_i equal to "`random`" or "`r`". Both of the parameters are nonnegative floats.

Random and Fixed

This option combines randomly drawn points from a specified number of different regions along with a number of fixed points. It can be called with T_i equal to "`random-with-fixed`" or "`rf`". Here we describe the parameters, thought it may be easier to just look at the example below.

- The first two parameters, $P_{i,1}$ and $P_{i,2}$, are the number of desired fixed and random points, respectively. Both are integers and they must satisfy $P_{i,1} + P_{i,2} = S_i$.

- The third parameter, $P_{i,3}$, is the number of desired different regions from which to draw random points. It is an integer. We let $x = P_{i,3}$ for the rest of this section for notational convenience.
- The next x parameters, $P_{i,4},\ P_{i,5},\ \ldots,\ P_{i,4+x}$, are the number of random points desired in each of the x regions. These parameters must sum to $P_{i,2}$, the number of desired randomly-drawn points. Each is an integer.
- The next $2x$ parameters, $P_{i,5+x},\ P_{i,6+x},\ \ldots, P_{i,5+3x}\ P_{i,6+3x}$ give the lower and upper bounds of each of the x different random regions. They alternate between lower and upper and progress through the x regions. I.e. $P_{i,5+x}$ and $P_{i,6+x}$ give the lower and upper bounds of the first random region, and $P_{i,5+3x}$ and $P_{i,6+3x}$ the lower and upper bounds of the last region. Each is a float.
- The final $P_{i,1}$ parameters, $P_{i,7+3x},\ \ldots,\ P_{i,7+3x+P_{i,1}}$, specify the desired fixed points. Each is a float.

This is much easier to see from example. Consider the following:

```
1-1/x 1 inf    rf 10 3 7   2 6 1   11.5 20   100 200   1 3.5 1_000_000
```

This says that we want 10 points in our support, 3 of them fixed and 7 randomly determined. There are 2 regions from which to randomly draw points. In the first region, draw 6 values. In the second, draw 1. The first region is between 11.5 and 20. The second region is 100 to 200. The three fixed points in every randomized trial are 1, 3.5, and 1,000,000.

4.3.4 Example Inputs

Because there is a lot going on here, we stop now to describe some example inputs.

Example 3. Say we want to run 1000 trials of the second-price auctions on the equal revenue auction. We might input:

```
1 2 1 1 0 1_000
1 2;20;40;60;80;100
D1
D1
1-1/x|1/(1-y)   1 inf   N/A
```

Example 4. Say we want to figure out the optimal revenue of the auction setting with 10 i.i.d. bidders competing to purchase 3 i.i.d. items, where each bidder independently values each item uniformly in $[0, 1]$. Perhaps the most intuitive thing to do would be to discretize the one bidder

class into a certain number of point masses uniformly in $[0, 1]$. Let us choose 5 points. If we want to do 50 trials of this, here would be the proper input text:

```
1 3 1 1 50 0
1 10
D1
D1
D1
x   0 1   random 5    0 1
```

Example 5. Say we have two different kinds of bidders competing to purchase two different items. The first kind of bidders values the first item according to the equal revenue curve and the second item uniformly in $[0, 100]$, independently across items. The second bidder values the first item with CDF $1 - 1/x^2$ and the second item deterministically with value 5. We are interested in the cases when there are either 15 or 20 of the first type of bidder, as well as when there are 5 or 10 of the second type of bidder. We might input:

```
2 2 2 3 100 1_000
1 0 15;20
0 1 5;10
D1 D3
D2 5
1-1/x|1/(1-y)   1 inf   rf 3 2 1   1 1   1 200   1 1_000_000
x/5|5*y         0 5     r  2               0 5
1-1/(x**2)      1 inf   rf 3 2 1   1 1   1 200   1 1_000_000
```

This says to take the described auction setting and discretize according to the following manner:

- Discretize $1-1/x$ and $1-1/x^2$ into three point masses, at 1, 1,000,000, and one value chosen uniformly in $[1, 200]$.
- Discretize $x/5$ into two points, at 1 and one uniformly in $[0, 5]$.
- Additionally, approximate the second-price auction candidates across 1000 trials.

4.4 Execution

Cara is executed by running `cara/cara.py` in the command line. In the same directory must exist `cara/cara/`, which contains the necessary modules and a RoaSolver JAR.

4.4.1 Execution Format

Here is the form of the execution line:

```
python3 cara.py [-f] [-c] [-m] [-t] [-o {a,w}] input_files [input_files ...]
```

`input_files` takes a variable number of input text files. "`*`" can be used as the wildcard character.

Cara always writes to stdout the best revenue achieved across the discretized trials and the second-price auctions for each bidder multiplicity profile.

4.4.2 Flags

Here are the flags.

- `-f` or `--full`. This flag takes no parameters and writes a CSV for each `input_file[i]` that contains the optimal revenue and discretized support for each discretized trial. Additionally, it writes another CSV that contains the revenue of both of the second-price auctions along with confidence interval data.
- `-c` or `--caffeine`. This flag takes no parameters. It stops the system from sleeping until the job is complete. The display may still turn off.
- `-m` or `--more`. This flag takes no parameters. It writes a text file for each `input_file[i]` that contains the *intermediate representation* of each of the t discretized trials. The intermediate representation of a discretized trial is the properly-formatted string that is passed into RoaSolver. Its structure is very similar to the input format described in Section 4.3, but uses finite-support distributions as opposed to our continuous ones. Its full specifications are described in Section 3.4 of [Shu19].
- `-t` or `--timing`. This flag takes no parameters. It writes two CSVs for each `input_file[i]`. One CSV gives the time that Cara takes for each discretized trial. The other CSV gives the time that RoaSolver takes for each bidder multiplicity profile in each discretized trial. The times sum to more than the total execution time because Cara runs discretized trials in parallel across different CPU cores.
- `-o` or `--output`. This flag takes a parameter, which must be "`a`" or "`w`". When writing text files or CSVs, it lets Cara either append (`a`) or overwrite (`w`) existing files. Cara will warn the user and quit if it tries to write to an existing file without this flag being specified.

4.4.3 Example Execution

`cara/analysis/examples/` contains three files, one for each of the example inputs in Section 4.3.4.

To get the full results for each input file written to CSVs, we can run:

```
python3 cara.py analysis/examples/* -f -c
```

On my device, the terminal looks like:

```
(thesis) antonstengel@Antons-MacBook-Air cara % python3 cara.py analysis/examples/*.txt -f -c

All input files are:
analysis/examples/example-3.txt
analysis/examples/example-4.txt
analysis/examples/example-5.txt

analysis/examples/example-3.txt results:
SPA: 100%|████████████████████████████████████████████████| 6000/6000 [00:13<00:00, 449.58it/s]
Discretization: 0it [00:00, ?it/s]
rev_2      4.8301
rev_20    39.4661
rev_40    87.9459
rev_60   132.3863
rev_80   172.5036
rev_100  246.1369

analysis/examples/example-4.txt results:
SPA: 0it [00:00, ?it/s]
Discretization: 100%|████████████████████████████████████████| 50/50 [12:14<00:00, 14.69s/it]
rev_10  2.3987

analysis/examples/example-5.txt results:
SPA: 100%|████████████████████████████████████████████████| 200/200 [00:02<00:00, 85.58it/s]
Discretization: 100%|████████████████████████████████████████| 50/50 [00:11<00:00,  4.24it/s]
rev_15_5    63.4477
rev_15_10   93.4477
rev_20_5    74.5969
rev_20_10  104.5969
```

Additionally, four files have been written:

- `cara/analysis/examples/example-3-res-spa.csv`
- `cara/analysis/examples/example-4-res.csv`
- `cara/analysis/examples/example-5-res-spa.csv`
- `cara/analysis/examples/example-5-res.csv`

The precise contents of these CSVs are described below.

Cara provides a progress bar for the second-price auction simulation and for the discretized trials. The discretized trials are generally run in parallel. Oftentimes many trials have been completed before children workers report back, so the progress bar can get frozen. When the `[-f]` flag is activated, Cara will write trial results to the results CSV live, which gives a more accurate picture of how quickly Cara is solving the input specifications.

trial	rev_15_5	rev_15_10	rev_20_5	rev_20_10	sup_1	sup_2	sup_3	sup_4	sup_5	sup_6
1	21.03	21.06	26.03	26.06	0.00	1.00	2.72	66.55	143.87	1000000.00
2	61.78	91.78	72.37	102.37	0.00	1.00	3.31	84.39	144.79	1000000.00
3	21.04	21.09	26.04	26.09	0.00	1.00	2.94	47.90	102.91	1000000.00
4	21.04	21.10	26.04	26.10	0.00	1.00	2.25	90.21	96.66	1000000.00
5	21.03	21.07	26.03	26.07	0.00	1.00	2.63	23.26	135.68	1000000.00
6	45.77	75.77	50.05	80.05	0.00	0.05	1.00	1.88	184.99	1000000.00
7	62.61	92.61	73.48	103.48	0.00	1.00	3.11	134.26	158.38	1000000.00
8	21.13	21.29	26.13	26.29	0.00	1.00	1.86	30.29	96.88	1000000.00
9	25.00	30.00	30.00	35.00	0.00	1.00	1.65	12.15	108.89	1000000.00
10	48.16	78.16	53.16	83.16	0.00	1.00	3.16	52.35	117.81	1000000.00

Table 4.1: Results CSV for Example 5 in Section 4.3.4

4.5 Output Specifications

When Cara is run, it writes text files to a directory called `temp/`. It will create the directory if it does not already exist. After completion, Cara deletes each of the files it wrote.

For a certain `input_file[i]`, let `OUTPUT` be `input_file[i]` without the file extension. For example, if the file being parsed is `data/trial-1.txt`, we have `OUTPUT = data/trial-1`. We list all the possible files that may be written below. When we have a flag in a parenthetical after the filename, this means that the file is only outputted when the flag is used.

- `OUTPUT-res.csv (-f)`. This CSV contains a table of revenues and supports for each discretized trial. The first column says which trial it is. The next $\prod_{\dot{b}\in\dot{\mathcal{B}}} N_{\dot{b}}$ columns each correspond to the revenue of a bidder multiplicity profile. For example, "`rev_15_5`" is the revenue of the bidder multiplicity profile where the number of bidders in the first bidder class is 15 and the second bidder class is 5. After the revenues, each column corresponds to one value in the support of the discretized trial. For each trial, the support values are increasing. Rows are written live as Cara trials finish. Cara generally solves trials in parallel across CPU cores, so the results are not necessarily in order. When the trials are done, Cara deletes the rows that were written and rewrites them in the correct order. Table 4.1 shows the beginning of the output for Example 3 in Section 4.3.4

- `OUTPUT-res-spa.csv (-f)`. This CSV contains a table of revenues for the second-price auctions. The first $\prod_{\dot{b}\in\dot{\mathcal{B}}} N_{\dot{b}}$ columns each correspond to a different bidder multiplicity profile. The first two rows give the average revenues of the second-price auction simulations. The next four rows give 95% confidence interval data about the means using the t-distribution. Table 4.2 shows the output for Example 3 in Section 4.3.4.

- `OUTPUT-res-int.txt (-m)`. This text file contains the intermediate input that is passed to RoaSolver. The intermediate representation is described in Section 4.4.2.

- `OUTPUT-results-time-gen.csv (-t)`. This CSV gives the time that Cara takes for each

	rev_15_5	rev_15_10	rev_20_5	rev_20_10
rev_spa_separate	21.53	19.50	21.68	22.39
rev_spa_bundle	19.49	17.53	19.51	20.37
0.95_conf_low_separate	18.15	15.94	18.78	18.81
0.95_conf_high_separate	24.91	23.07	24.59	25.96
0.95_conf_low_bundle	16.10	14.04	16.57	16.83
0.95_conf_high_bundle	22.88	21.03	22.46	23.92

Table 4.2: Second-price auctions results CSV for Example 5 in Section 4.3.4

discretized trial. There are just two columns, `trial` and `time`.

- `OUTPUT-results-time-pre.csv (-t)`. This CSV gives the time that RoaSolver takes to solve each bidder multiplicity profile in each discretized trial. The first column is `trial` and the next $\prod_{\dot{b}\in\dot{\mathcal{B}}} N_{\dot{b}}$ columns each correspond to the time it takes for a certain bidder multiplicity profile.

Chapter 5

Results

In Section 5.1, we compare the results of Cara to state-of-the-art deep learning models on a few simple auction settings [Düt+17; RJW20].

In Section 5.2, we test how well Cara does on the equal revenue auction setting. Cara is able to match the $2n+\Omega(\log n)$ lower bound with both the second-price auction simulations and discretized trials, but is unable to do better. We believe this exploration may provide slight evidence that the equal revenue auction is $2n + \Theta(\log n)$ or at least $2n + o(\sqrt{n})$.

5.1 Comparison to Deep Learning Methods

[Düt+17] and [RJW20] model auction settings as multi-layer neural networks and the optimal auction design as a constrained learning problem. Both papers document experimental results across a range of simple auction settings. In this section, we compare Cara against their results when the bidder valuation distributions are additive—after all, that is what Cara can do.

Both papers look at the additive setting where n i.i.d. bidders value m items uniformly in $[0, 1]$, for a few choices of $n \times m$. [Düt+17]'s deep learning model is called RegretNet. [RJW20]'s primary model is ALGnet.

In Table 5.1 we compare Cara's performance to RegretNet and ALGnet. We breakdown Cara's revenue estimates into the discretized trials as well as second-price auctions. The value is "N/A" for the discretized trials when Cara could not handle the input. With ten items, RoaSolver was unable to run because CPLEX could not solve the linear program. Selling separate and the grand bundle are "N/A" when there is only one bidder because they do not specify valid auction mechanisms in these cases. There is no standard deviation data available for RegretNet.

$(n \times m)$	Discretization	Selling Separate	Grand Bundle	Cara	RegretNet	ALGnet
1×2	0.536	N/A	N/A	0.536	0.554	0.555 ($\pm$0.0019)
1×10	N/A	N/A	N/A	N/A	3.461	3.487 ($\pm$0.0135)
2×2	0.876	0.667 ($\pm$0.0020)	0.768 ($\pm$0.0021)	0.876	0.878	0.879 ($\pm$0.0024)
3×10	N/A	4.998 ($\pm$0.0043)	5.000 ($\pm$0.0038)	4.999	5.541	5.562 ($\pm$0.0308)
5×10	N/A	6.667 ($\pm$0.0035)	5.457 ($\pm$0.0032)	6.667	6.778	6.781 ($\pm$0.0504)

Table 5.1: Comparison of optimal revenue approximations between Cara and deep learning models.

In the 1×2 setting, the true optimal revenue is 0.550 [Düt+17]. RegretNet and ALGnet both slightly overestimate the optimal revenue, while Cara lower bounds it, as expected. In the 2×2 case, Cara does very similarly to the deep learning models. When there are 10 items, the best she can do is return the better of selling separately and grand bundling.

This is not a very fair comparison for Cara, however. The program is made to aid researchers quickly approximate the revenue of auction settings in a variety of situations. Her results for Table 5.1 took thirty minutes running locally on a MacBook Air. The only upstart cost is specifying the input file. For the 2×2 setting, it was the following:

```
1 2 1 1 100 100_000
1 2
D1
D1
x|y   0 1   random 14    0 1
```

On the other hand, RegretNet and ALGnet were trained on NVIDIA compute clusters. A different network was trained for each auction setting. While these deep neural networks provide novel ways of formulating the structure of multi-item auctions, they are not good tools to aid researchers on-the-fly.

5.2 Equal Revenue Auction

In Section 1.2.2, we briefly defined the equal revenue auction. Here we do it again.

Definition 7 (Equal revenue distribution)**.** The *equal revenue distribution* ($\mathcal{ER}$) is the distribution with CDF

$$F(x) = \begin{cases} 1 - 1/x & x > 1 \\ 0 & x \leq 1 \end{cases}.$$

We say that the *equal revenue auction setting* with n bidders is the setting in which n i.i.d. bidders draw values for two items from the equal revenue distribution, independently across items. In our

notation, we write $(x_1^{(b)}, x_2^{(b)}) \sim \mathcal{ER}^2$ for $b \in \mathcal{B}$.

Specifically, we want to know the asymptotic behavior of the optimal revenue as n diverges. Previous research shows that it is $2n + \Omega(\log n)$ and $2n + \mathcal{O}(\sqrt{n})$.

5.2.1 Grand Bundle

Cara easily achieves the lower bound, but not because of fancy discretizations; the grand bundle has revenue of $2n + \Theta(\log n)$. Let us show this. First we will find an open form equation for the the grand bundle revenue. Then we will simulate it using Cara. We make use of the following proposition, which follows a proof in [BW19]. This proposition gives the CDF of the sum of a bidder's valuations in the equal revenue auction.

Proposition 5.

$$\mathop{\mathbb{P}}_{x\sim\mathcal{ER}^2}[x_1 + x_2 \geq a] = \frac{2}{a} + \frac{2\log(a-1)}{a^2}$$

Proof. There are two ways for us to have $x_1 + x_2 \geq a$. We might have $x_1 \geq a - 1$, as x_2 must be at least 1. Or we have $x_1 < a - 1$ and $x_2 > a - x_1$. Then we have

$$\begin{aligned}
\mathop{\mathbb{P}}_{x\sim\mathcal{ER}^2}[x_1 + x_2 \geq a] &= \frac{1}{a-1} + \int_1^{a-1} \left(\frac{1}{z^2}\right)\left(\frac{1}{a-z}\right) dz \\
&= \frac{1}{a-1} + \left[\frac{z\log(a-z) + a - z\log z}{a^2 z}\right]_{z=1}^{z=a-1} \\
&= \frac{2}{a} + \frac{2\log(a-1)}{a^2},
\end{aligned}$$

where we leave out some algebra that took an embarrassing amount of time to figure out. □

Now we are ready to find the revenue of grand bundling. Let $v^{(i)}$ be the ith largest valuation that one of the n bidders has for the grand bundle. Let N_a be the number of bidders who value the bundle at least a, i.e. all $b \in \mathcal{B}$ satisfying $x_1^{(b)} + x_2^{(b)} \geq a$. For the second-price auction to have revenue at least a, we must have $N_a \geq 2$. Of course,

$$\mathbb{P}[N_a \geq 2] = 1 - \mathbb{P}[N_a = 1] - \mathbb{P}[N_a = 0].$$

There are exactly n ways for $N_a = 1$ and we have $N_a = 0$ when all bidders fail to reach a. So we

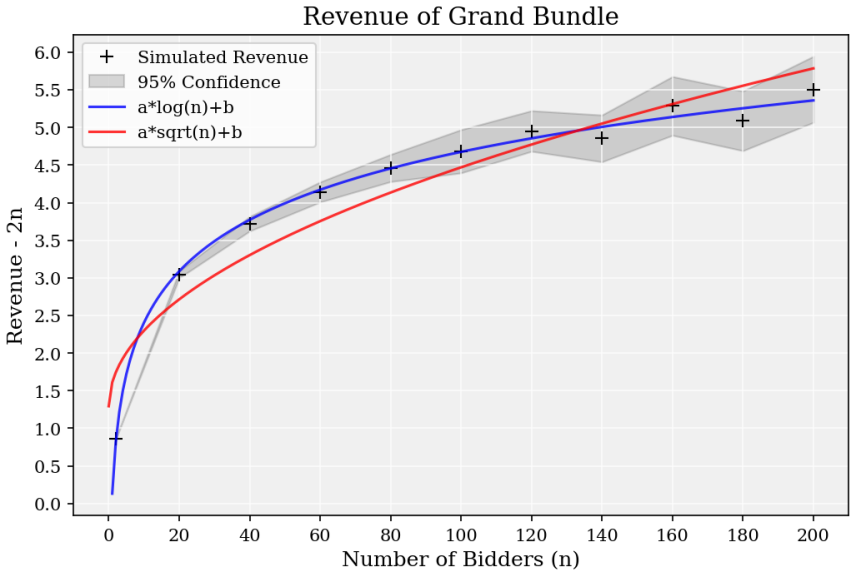

Figure 5.1: Simulated revenue of the equal revenue auction with grand bundle.

can write

$$\begin{aligned}\mathbb{P}\left[v^{(2)} \geq a\right] &= \mathbb{P}[N_a \geq 2] \\ &= 1 - \mathbb{P}[N_a = 1] - \mathbb{P}[N_a = 0] \\ &= 1 - n\left(1 - \frac{2}{a} - \frac{2\log(a-1)}{a^2}\right)^{n-1}\left(\frac{2}{a} + \frac{2\log(a-1)}{a^2}\right) \\ &\quad - \left(1 - \frac{2}{a} - \frac{2\log(a-1)}{a^2}\right)^{n}.\end{aligned}$$

Now the revenue of the grand bundle is $\mathbb{E}[v^{(2)}] = \int_0^\infty \mathbb{P}[v^{(2)} > a]da$, or

$$\int_0^\infty 1 - n\left(1 - \frac{2}{a} - \frac{2\log(a-1)}{a^2}\right)^{n-1}\left(\frac{2}{a} + \frac{2\log(a-1)}{a^2}\right) - \left(1 - \frac{2}{a} - \frac{2\log(a-1)}{a^2}\right)^{n} da.$$

Deriving a closed-form expression for this integral is hard. [Bar19] writes a program to evaluation the integral, finding that it is $\approx 2n + \log n + 0.17$. Instead, we use Cara to simulate the grand bundle itself. Running 20,000,000 trials for each number of bidder, we graph the results in Figure 5.1. We subtract $2n$ off the revenue so that it is easy to see the lower-order behavior. Additionally, we plot the closest fitting logarithmic and square root functions by minimizing the squared error, which are the following:

Square Root		Logarithm	
Fit	MSE	Fit	MSE
$0.32\sqrt{n} + 1.30$	0.1577	$0.99\log(n) + 1.32$	0.0102

The mean squared error is substantially lower for the logarithmic fit. Since this does not show

conclusively that the grand bundle achieves $2n + \Theta(\log n)$, an interested reader should check out [BW18].

5.2.2 Discretization Attempts

All of the analysis done in Section 5.2.2 is contained in the following locations:

- `cara/analysis/notebooks/era-discretizations.ipynb`
- `cara/analysis/data/era-discretizations/`

Capturing $2n$ growth

We have used Cara to simulate the $2n + \Omega(\log n)$ lower bound on the grand bundle. Now we want to see if we can do any better by discretizing the equal revenue distribution. This task may be difficult because we are not just trying to get a broad idea of the revenue; we already know it is $2n + \Omega(\log n)$ and $2n + \mathcal{O}(\sqrt{n})$. Instead, we want to see if our discretized trials can capture the lower-order term.

We have priors that the $2n$ of revenue can be captured by a single point mass. Our intuition comes from the following mathematical sketch. Say that all n bidders draw their values for each of their items independently from the distribution

$$D_\epsilon = \begin{cases} 0 & \text{w.p. } 1-\epsilon \\ 1/\epsilon & \text{w.p. } \epsilon \end{cases}$$

for some $\epsilon > 0$. This is the distribution we would give if we want to capture all of the equal revenue distribution at a point $x = 1/\epsilon$. Let X be the highest valuation for either of the items by one of the n bidders. Then naively we might imagine the optimal revenue of this auction setting to be around $\mathbb{E}[X]$. We can write

$$\begin{aligned} \mathbb{E}[X] &= 0\mathbb{P}[X = 0] + (1/\epsilon)\mathbb{P}[X = 1/\epsilon] \\ &= (1/\epsilon)\mathbb{P}[X = 1/\epsilon] \\ &= (1/\epsilon)\left(1 - (1-\epsilon)^{2n}\right). \end{aligned}$$

By union bound we know

$$\mathbb{P}[X = 1/\epsilon] \leq \sum_{\substack{b \in \mathcal{B} \\ i \in \mathcal{I}}} \mathbb{P}\left[x_i^{(b)} = 1/\epsilon\right] = 2n\epsilon.$$

Additionally, we know

$$\mathbb{P}[X = 1/\epsilon] = 1 - (1-\epsilon)^{2n} \geq 2n\epsilon - \binom{2n}{2}\epsilon^2 \to 2n\epsilon \text{ as } \epsilon \to 0.$$

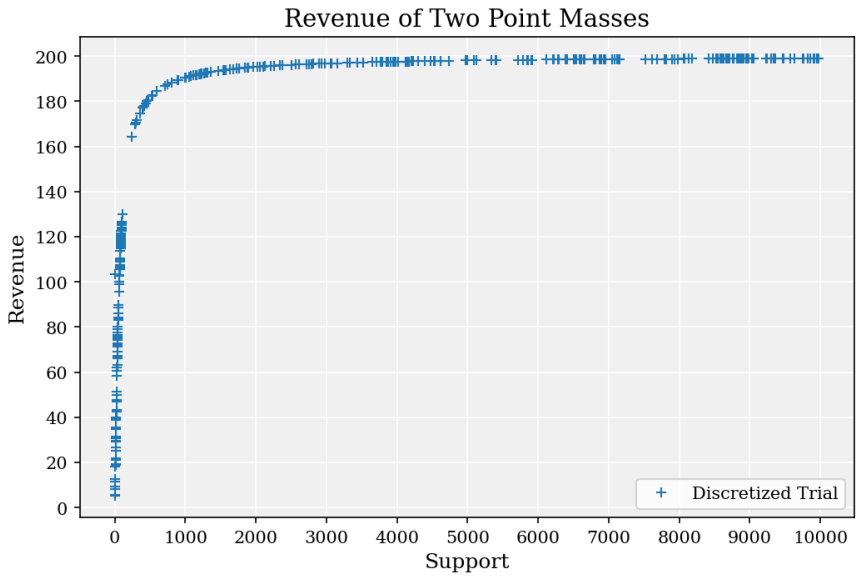

Figure 5.2: Approximating the equal revenue auction ($n = 100$) with two point masses, one at x=1 and the other varying along the horizontal axis.

The equality is true because the probability that at least one bidder values an item at $1/\epsilon$ is one minus the probability that none do. The inequality comes from the Bonferroni inequalities, which are just a generalization of the union bound. Thus we know

$$\mathbb{E}[X] \to (1/\epsilon)2n\epsilon = 2n \quad \text{as} \quad \epsilon \to 0.$$

We do not have to settle with this expected value. Instead, we simulate the results on Cara and do indeed see $2n$ of the revenue is captured by a single large point mass. The results are shown in Figure 5.2. Additionally, we observe that the revenue increases monotonically with the point mass, which is also true of $\mathbb{E}[X]$.

Capturing lower-order growth

[Kot+19] shows that when facing a single buyer, the optimal revenue can be approximated arbitrarily well by discretized distributions with a single large point in the support. While this only formally applies to one bidder, we use this as intuition; our one outlier value captures $2n$ of revenue and so we must look at the lower end of the support to capture the lower-order term that is $\Omega(\log n)$ and $\mathcal{O}(\sqrt{n})$.

Additionally, from the sketch above, we think this lower-order term is the revenue gained from the lower-valued item. To figure out the best areas to capture the second item's distribution, we run randomized trials with ten point masses in varying ranges. There is always one point at $x = 1$ so that the discretized distribution is stochastically-dominated by the equal revenue curve.

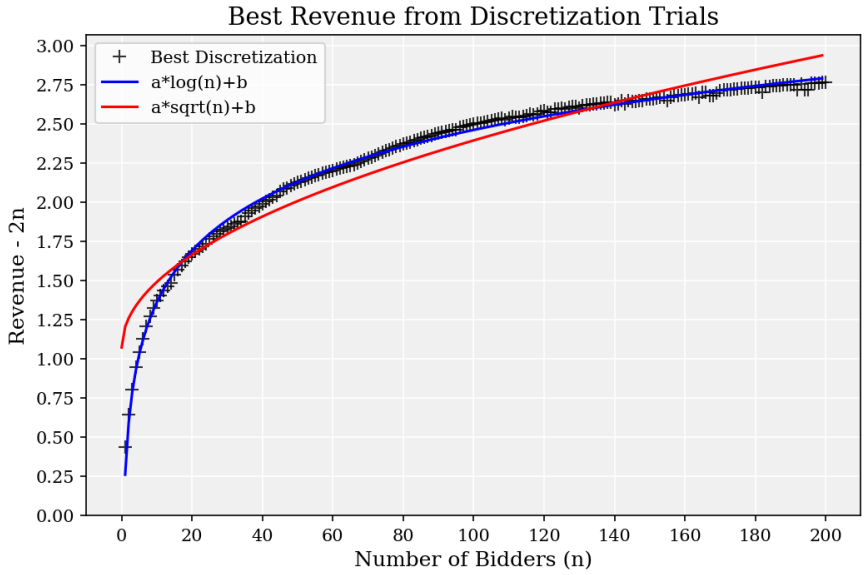

Figure 5.3: Approximating the equal revenue auction using Cara. The auction setting was discretized into ten points. Over many trials, the best revenue for each n is graphed.

Additionally, we always choose one point at $x = 1{,}000{,}000$ to capture the $2n$ of revenue from the highest-value item.

After getting a sense of the best areas to discretize, we run many more trials. (For $n = 100$, for example, the best support values seem to be somewhere in $[0, 90]$). In Figure 5.3, we graph the best revenues we found for each $n \leq 200$. We call these the results from the *general discretized trials*. We additionally plot the best logarithmic and square root fits. They are the following:

Square Root		Logarithm	
Fit	MSE	Fit	MSE
$0.13\sqrt{n} + 1.07$	0.0164	$0.48\log(n) + 0.26$	0.0008

The logarithmic fit is much better. However, the best revenues from 5.3 are still lower than that of the grand bundle.

We want to see if we can find a discretized auction that outperforms the grand bundle for at least one n. We fix $n = 100$ bidders and carry out the following algorithm:

1. Begin with two points $1 \in \hat{S}$ and $1{,}000{,}000 \in \hat{S}$.
2. For all $s \in (1, \infty)$, find the optimal revenue of the discretized auction with support of $\hat{S} \cup s$. (It is assumed that we always assign stochastically-dominated masses according to the rule described in Section 4.3.3 and Figure 4.1.)
3. Add the $s \in (1, \infty)$ to $\hat{S}$ that maximizes the revenue ($\hat{S} \leftarrow \hat{S} \cup s$).

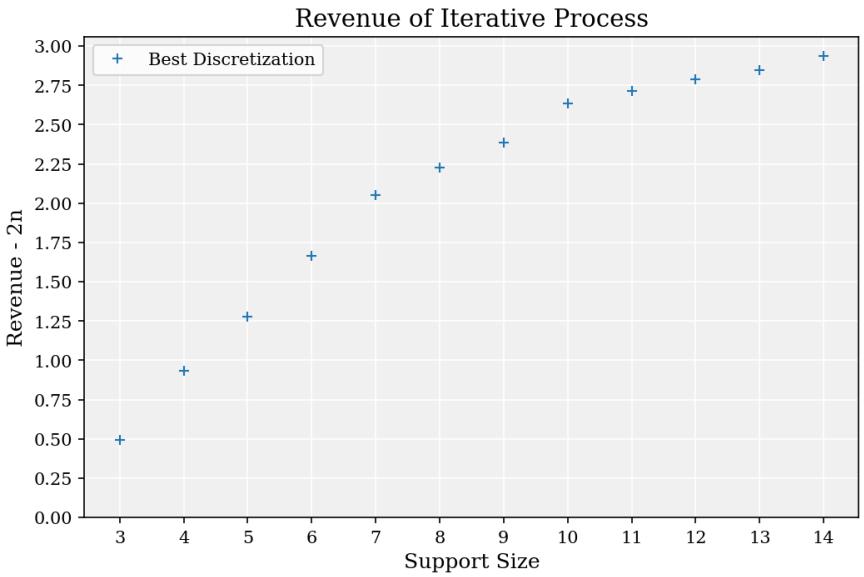

Figure 5.4: The discretized revenue achieved on the equal revenue auction for $n = 100$ when iteratively adding the best point we can find to the support.

4. Repeat steps (2) and (3) until we have fourteen points ($|\hat{S}| = 14$).

We call this algorithm the *iterative process*. At each iteration, we add one more point to the discretized support, choosing the point that maximizes the revenue. Using Cara to carry out the algorithm, we end up with the following values:

1	1,000,000	33.65	17.15	50.18	3.11	64.41	78.50	6.67	74.81	40.85	31.24	81.89	2.00

The values are listed in the order in which they were added. In Figure 5.4, we graph the revenue achieved at each number of points in the support

In Figure 5.5, we show the details of the iterative process. On the horizontal axis of each plot, we vary the ith value in the support of $\mathcal{ER}$, holding constant the first $i-1$ values given in the list above. For each sampled value, we find the optimal revenue of the discrete auction setting, which is the vertical axis. The dotted red lines show where the $i-2$ frozen values in the support are (excluding $x = 1{,}000{,}000$). Looking at these figures, we get a sense of the topology of the discretized space, which is interesting in its own right.

We believe that the appearance of stacked lines that begin in "Revenue of 8 Point Masses" is due to numerical precision issues. These stacked lines occur in all of the previous plots too, but are difficult to see because of the spacing of the vertical axis. Additionally, the number of trials decreases as we increase the size of the support because they take longer to run.

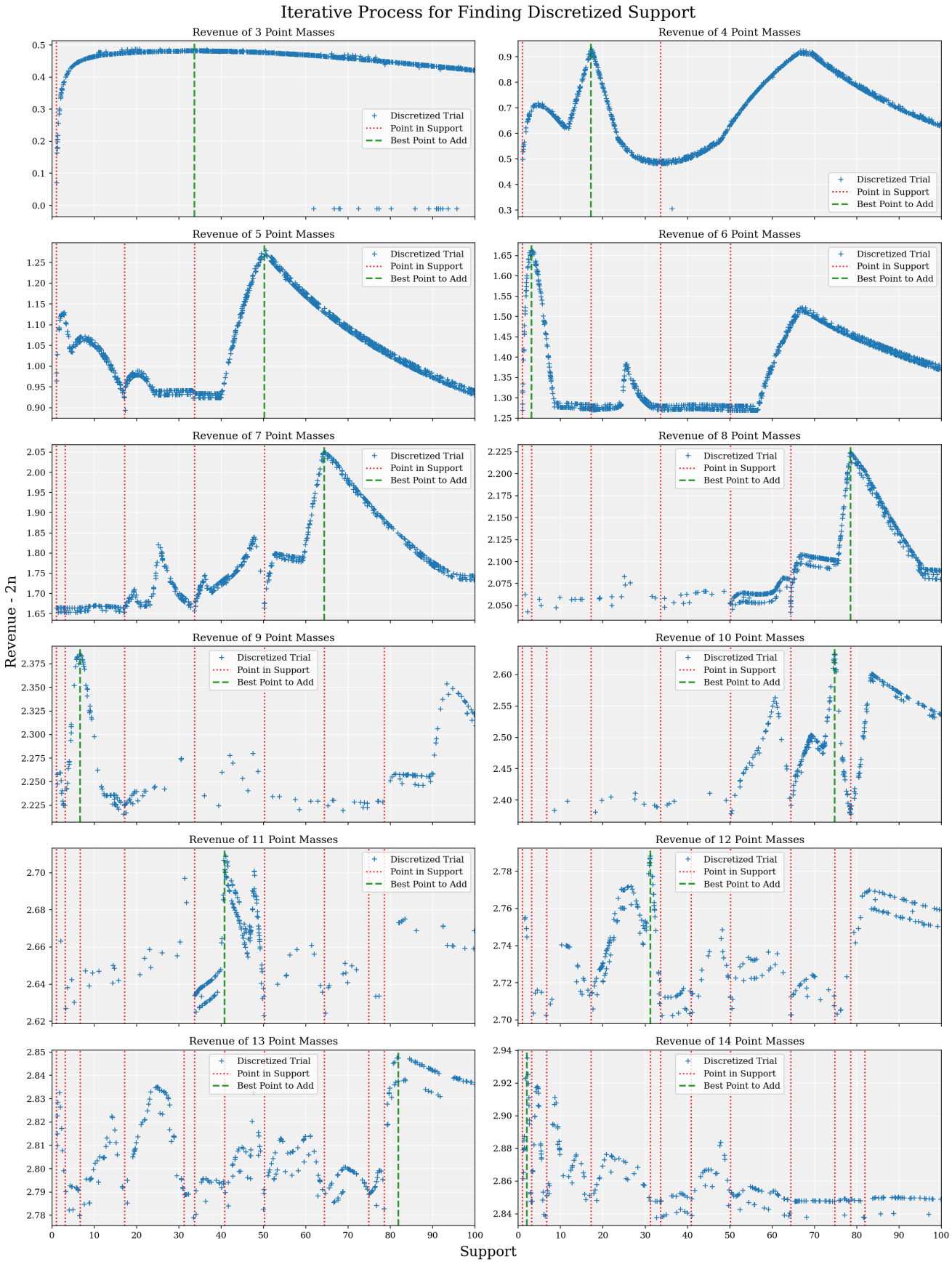

Figure 5.5: Adding values to the support iteratively for the equal revenue auction with $n = 100$ bidders.

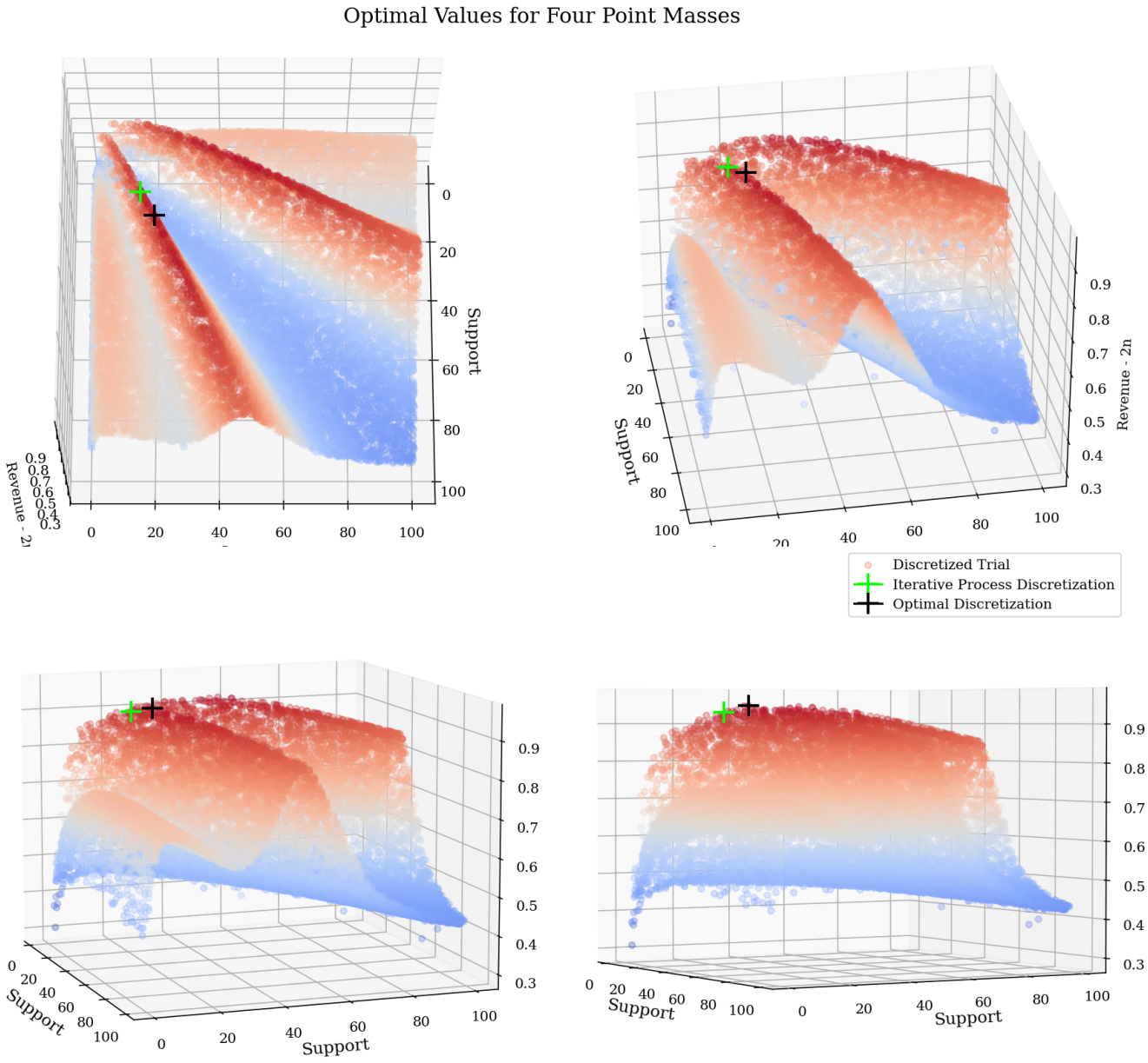

Figure 5.6: Testing if the iterative process is optimal for four point masses. Two point masses are fixed at $x = 1$ and $x = 1{,}000{,}000$ and the rest vary in trials across the two horizontal axes. The vertical axis shows the revenue in excess of 200 for each trial. The green cross, near optimal, is the point chosen by the iterative process.

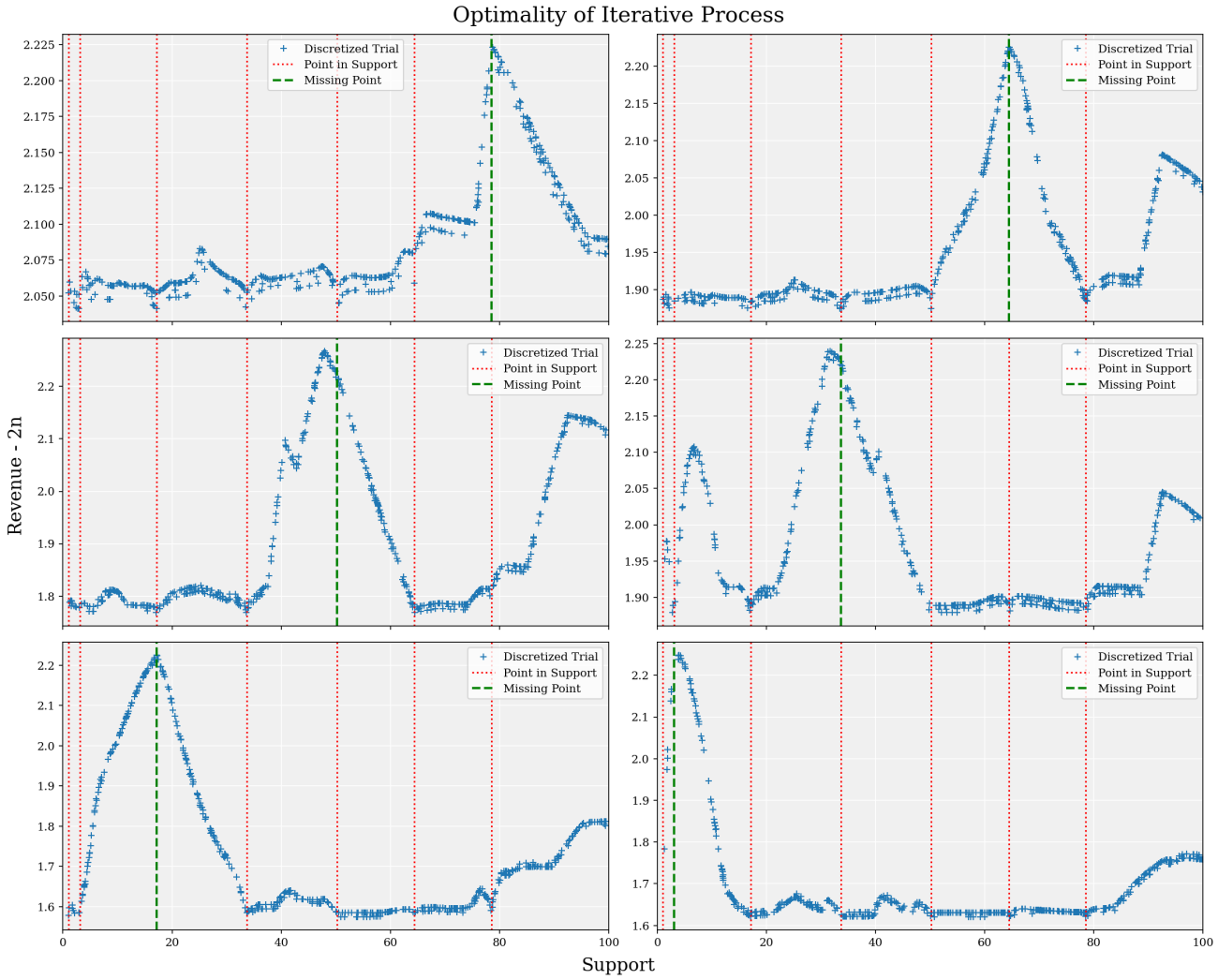

Figure 5.7: Attempting to see if the discretized supports of the iterative process are optimal. The top-left plot shows the eighth point that was actually found. Each other plot removes one of the point masses and tests whether it is the best value to add back.

Discussion

This iterative process, of course, does not necessarily find the optimal fourteen point masses to discretize the equal revenue auction for $n = 100$; repeatedly choosing the best point on the margin does not necessitate finding the best set of points together. To get a sense of how optimal our iterative process is, we do a few things.

First, we see how close to optimal it is when the support size is four. We choose four points because it is tractable to approximately find the best support. In Figure 5.6, we vary two of the values in the support (holding constant $x = 1$ and $x = 1{,}000{,}000$). We compare the optimal support found from 20,000 trials to the one found from the iterative process. They are very similar.

Second, we want to see if this optimality remains as the support gets bigger. Since we do not have the compute to carry out the experiment of Figure 5.6 for much larger supports, we do the following. We consider the case of a support with eight point masses, using the eight points that the iterative process found. Then for each point in the support, we remove it and search to find the best eighth point to add back. In each case, the best point to add back was very close to the one that was just removed. This gives us some confidence that the iterative process finds among the best possible discretizations. This test is shown in Figure 5.7.

Finally, we compare the support found from the iterative process (with support size 10) to the best support found from the general discretized trials. This is Figure 5.8. We see that they are quite similar, further supporting the thesis that the iterative process is approximately optimal.

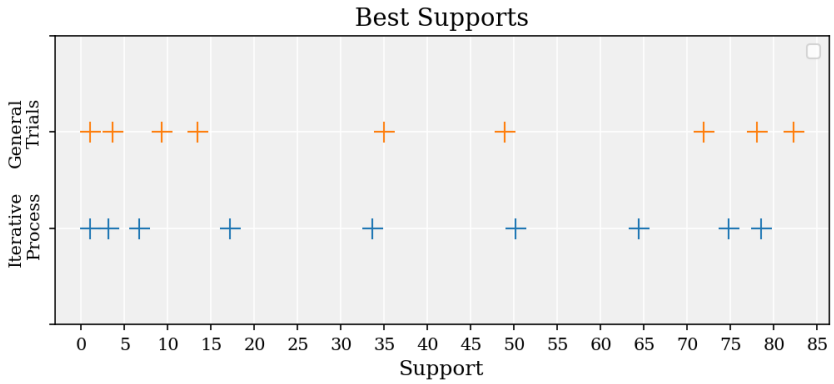

Figure 5.8: Comparing the best supports found from the iterative process and the general discretized trials, when the support size is ten.

If the iterative process is close to optimal, then what do the results in Figure 5.4 mean? If we accept the working hypothesis that this auction setting is revenue monotonic, then the revenue must be monotonically increasing as the support of the iterative process grows. It is interesting to think about how this revenue compares to that of the grand bundle as we increase the support size.

The grand bundle versus discretization represents a straightforward trade-off. The grand bundle's advantage is that it does not lose any revenue to bad approximations of the equal revenue curve. The discretization's advantage is that it gets to find the optimal mechanism (for its approximated distribution). If the revenue of the discretization stays worse than the grand bundle as we approximate the equal revenue curve better and better, it means that the grand bundle is not losing much—if any—revenue in its inability to search for the best mechanism. Therefore it provides evidence that the grand bundle itself is a near optimal mechanism. Thus we think the results in this section may slightly improve the belief that the equal revenue auction is $2n + \Theta(\log n)$, or at least $2n + o(\sqrt{n})$.

Chapter 6

Software Documentation

In this chapter, we overview Cara's codebase.

- Section 6.1 discusses the environments used when developing Cara as well as overviews the repository structure.
- Section 6.2 provides more information on the input text file than Section 4.3, specifically in defining hyperparameters that can be useful when debugging.
- Section 6.3 gives an overview of the different modules and classes.
- Section 6.4 details interesting aspects of the codebase.
- Section 6.5 discusses possible further work.

6.1 Development Environment

6.1.1 System Specifications

For reference, here are some specifications of the device that were used to develop Cara:

- 2022 M2 Macbook Air running macOS Ventura 13.0.1.
- Python 3.11.2.
- Conda and Pip 22.9.0.
- Git 2.37.1 (Apple Git-137.1).
- Python packages described according to `cara/requirements.txt`.

6.1.2 Repository Setup

The full repository exists on Github.[1] To clone it locally, run:

```
git clone https://github.com/antonstengel/cara.git
```

The repository will be cloned into a directory named `cara`. The directory contains the following files and folders, among others:

- `cara/cara.py` is the main script used to run Cara.
- `cara/cara/` is the package that contains all modules on which `cara/cara.py` depends. Additionally, it contains a RoaSolver JAR for Cara to use.[2]
- `cara/temp/` contains the temporary intermediate representations while Cara is executing.
- `cara/analysis/examples/` contains all examples from Chapter 4.
- `cara/analysis/notebooks` and `cara/analysis/data` contain all analysis from Chapter 5.

To run the program, all that is needed is for `cara/cara.py` and `cara/cara/` to be in the same directory. A `temp/` directory will be automatically created when Cara is executed if it does not already exist.

6.1.3 Java Development

A user who wishes to modify RoaSolver should read Chapters 3 and 4 of [Shu19]. Shu documents the development environment he used as well as debugging and build procedures. One piece of advise for macOS users—if Maven is having difficulty configuring CPLEX as a dependency, try adding the flag `-DgeneratePom=true` to the Maven configuration command that Shu describes:

```
mvn install:install-file -DgroupId=cplex -DartifactId=cplex
    -Dversion=12.9 -Dpackaging=jar -DgeneratePom=true
    -Dfile="/Applications/CPLEX_Studio129/cplex/lib/cplex.jar"
```

6.2 Hyperparameters

Before we delve more into the codebase, it is necessary to briefly discuss the input text format. In Section 4.3, we did not tell the whole truth. The full input file format is

[1] https://github.com/antonstengel/cara

[2] This may be the wrong place for the JAR, but it is convenient for a user if all they need is *cara.py* and *cara/cara/* together to run the program.

$$
\begin{array}{cccccccc}
\dot{n} & m & \dot{c} & r & t & a & & \\
p_{11} & \dots & p_{1\dot{c}} & N_1 & & & & \\
\vdots & \ddots & \vdots & \vdots & & & & \\
p_{\dot{n}1} & \dots & p_{\dot{n}\dot{c}} & N_n & & & & \\
d_{11} & \dots & d_{1\dot{c}} & & & & & \\
\vdots & \ddots & \vdots & & & & & \\
d_{m1} & \dots & d_{m\dot{c}} & & & & & \\
\phi_1 & L_1 & U_1 & T_1 & S_1 & P_{11} & \dots & P_{1*} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\phi_r & L_r & U_r & T_r & S_r & P_{r1} & \dots & P_{r*} \\
h_1 & \dots & h_* & & & & &
\end{array}
$$

where the $h_1, \dots, h_*$ is new. These are desired hyperparameters. Each hyperparameter is optional, and is of the form "`NAME=VALUE`" for valid "`NAME`" and "`VALUE`". Additionally, there may be any number of blank lines between the last two lines, i.e. between the line that begins with ϕ_r and the line that begins with h_1.

All current hyperparameters and their default values are in `cara/cara/hyperparameters.py`. These parameters deal with things like deciding which RoaSolver JAR to use and how to round values before and after they are passed into RoaSolver. Much of the development process of Cara was unearthing numerical precision issues. Here is a workflow that was common during development:

- Observe numerical precision issues in a certain auction setting.
- Create a directory `roasolvers/` that contained three different JARs, `roasolver-1.jar`, `roasolver-2.jar`, and `roasolver-3.jar`. Each of these JARs contains slightly different math for how RoaSolver handles its linear programming constraints.
- Rerun the auction setting with three different input files in an `issues/` directory, each of which sets `roasolver=roasolvers/roasolver-i.jar` for a different i.

Doing this is much more convenient than manually changing the code between input files; all auction settings can be run with one command:

```
python3 cara.py issues/* -f -m -c -t
```

Additionally, it is easy to keep track of which results used which RoaSolver because they are listed in the input text files.

6.3 Code Overview

There are three primarily classes used in Cara. Distribution is in `cara/cara/distribution.py`, Parser in `cara/cara/auction_parser.py`, and Auction in `cara/cara/auction.py`. The outward-facing script is `cara/cara.py`.

6.3.1 Distribution Class

The Distribution class inherits from SciPy's `stats.rv_continuous` class, which is a generic continuous random variable class meant for subclassing.[3] Each Distribution instance takes as argument a string corresponding to a distribution's CDF as well as the end points of the support (see Section 4.3 for more details). Each discretization option has a method which takes as input the parameters described in Section 4.3 and returns a tuple of the masses and support.

Two things related closely to Distribution are currently not handled within the class. First, combining the discretized supports of each valuation distribution is done in `Auction.discretize()`. Second, using the optional PPF (see Section 4.3) to speed up distribution sampling is dealt with in `Auction.run_spa()`.

6.3.2 Parser Class

Each Parser instance takes as argument a string corresponding to the input described in Section 4.3. It then parses the input, setting all instance variables. The Parser class has no methods and returns nothing.

6.3.3 Auction Class

The Auction class is where the majority of Cara's logic is handled. Each Auction instance takes as argument a Parser object and stores all relevant data as instance variables. It has a few public-facing methods:

- `Auction.discretize()` discretizes the auction setting once, updating the instance variables.
- `Auction.output()` returns a properly formatted string that serves as input for RoaSolver. The Auction instance must be discretized first. This string that serves as input to RoaSolverwe call the *intermediate representation of a discretized trial*.
- `Auction.run_spa()` runs the second-price auctions and returns a DataFrame of the results (see Section 4.5 for output specifications). The method simulates both selling separately and as a grand bundle `self.a` times each, returning the mean as the revenue of the auction.

[3] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.rv_continuous.html`

The auctions are simulated by inverse transform sampling the given valuation distributions. When there is no PPF inputted, this is done through SciPy's `stats.rv_continuous` class, which is much slower. Additionally, this method returns confidence interval data from the samples using the t-distribution.

- `Auction.run()` takes as input the runtime flags (see Section 4.4) as well as a string for the desired output filenames. It calls `Auction.run_spa()`, storing the results of the second-price auctions. Then it runs `self.t` discretized trial, each time (1) randomly discretizing the valuation distributions with `Auction.discretize()`, (2) using `Auction.output()` to pass that data into RoaSolver, (3) parsing RoaSolver's response, and (4) outputting the results to stdout as well as writing to the proper files.

6.3.4 Cara Script

`cara/cara.py` is the point of contact for a user. It parses input files and runtime flags, checking if files already exist. Then it iterates through the input files, creating a new Parser and Auction instance for each file.

6.4 Code Details

Here we highlight a couple of interesting engineering details. The whole codebase is well documented and should not be too hard to understand on its own.

6.4.1 Notational Conventions

We use the same names for variables in the codebase as are used for the input specification described in Section 4.3. The only difference is that we drop the dot on $\dot{n}$ and $\dot{c}$, and we write out ϕ as `phi`.

We append the dimensions to the names of lists and multidimensional arrays. We do this by adding an underscore after the name and then the dimensions of the array in order. For example, the variables p_{ij} for $0 \leq i \leq \dot{n}$ and $0 \leq j \leq \dot{c}$ become `self.p_nc` in the code.

When indexing arrays, we use `i` and prepend the dimension. For example, the following idiom appears often in the code:

```
for mi in range(self.m):
    for ci in range(self.c):
        # do something with self.d_mc[mi][ci]
```

6.4.2 Combining Discretized Distributions

`Auction.discretize()` discretizes each valuation distribution `self.phi_r[ri]` according to its discretization type `self.T_r[ri]` and parameters `self.P_r[ri]`, giving us a list of `self.r` panda Series, which we call `dists_r`. Each Series is the masses for the rth discretized valuation distribution indexed by its support.

RoaSolver must take as input a single suport `self.u_s` that has `self.s` many point masses in it. The masses are given by a matrix `self.phi_rs`, where row i gives the masses for the ith valuation distribution.[4]

This is not being mentioned because of any engineering difficulty, but only because there is not a uniquely correct way to combine `dists_r` to get `self.u_s` and `self.phi_rs`. For example, say that the contents of `dists_r` are the following:

`dists_r[0]`

support	mass
1	0.2
5	0.4
10	0.6

`dists_r[1]`

support	mass
1	0.5
3	0.5

`dists_r[2]`

support	mass
1	0.9
100	0.1

We have `self.u_s = (1,3,5,10,100)`. Perhaps the most natural way to write `self.phi_rs` is the following:

`self.phi_rs`

	1	3	5	10	100
0	0.2	0	0.4	0.6	0
1	0.5	0.5	0	0	0
2	0.9	0	0	0	0.1

Here we just stack the discretization of each valuation distribution. However, in the same five support values, we could get more precision by take the stochastically-dominated masses for each valuation distribution on `self.u_s`; i.e. we could fill in the 0's. This would give more tightly-fit discretized valuations in the same-sized overall support. However, this would in some sense less honor the different individual discretizations, and if RoaSolver's time depends not just on the size of `self.u_s`, but also on the number of nonzero entries in `self.phi_rs`, the former method may be best.

Currently, Cara uses the former method as it seems to work better experimentally. This section's purpose is not to fret over a small design choice in the codebase, but just to highlight that there

[4] We made sure here to use the same notation as found in [Shu19]

are many of these decisions in Cara that had to be made and can be more methodically optimized.

6.4.3 Multiprocessing

Cara runs RoaSolver in parallel across CPU cores using the Python package Pathos,[5] speeding up computation significantly (See Table A.1). The standard Python multiprocessing package only serializes functions from the top level of a module, making it difficult to split tasks in parallel within a Class method. Pathos, on the other hand, uses Dill, which has more comprehensive serialization.

When each discretized trial causes a very large linear program for CPLEX, Cara may crash the user's device if the trials are split in parallel, due to virtual memory usage. To get around this, the first trial is always run alone and if it is completed quickly enough, then the rest is split into parallel. If not, all discretized trials are run sequentially.

Most of `Auction.run()` deals with splitting the computation in parallel. First, all intermediate representations that are to be passed into RoaSolver (one file per discretized trial), are stored as text files in `cara/temp/`. Then the task being parallelized (`run_trial()`) calls RoaSolver with one of the temporary files and parses the results.

When `self.args.full=True`, results are written live to the results file as children workers finish in the `ProcessingPool()`. Thus the trial revenues and supports are not necessarily written in the correct order. After all children workers are done, `Auction.run()` deletes the last `self.t` lines of the results file and rewrites them in the correct order. We note that if two children workers finish at the same approximate time, they may both write to the same line of the results file, messing up its formatting.

6.5 Further Work

6.5.1 Correctness

When passing to RoaSolver through the intermediate representation, currently Cara rounds down all support values and and the mass of the largest value in the support. Technically, rounding down the final mass raises all the other ones because RoaSolver weights them to sum to one. In some auction settings, this could make it so that the discretized distribution is not stochastically dominated by the original continuous one. Since the precision is high, this rounding seems not to make much difference at all, but it is worth exploring more.

[5] `https://pypi.org/project/pathos/`

6.5.2 Performance

In Appendix A, we give a brief overview of timing data about Cara. It would be nice to rigorously explore the relationship between different variables and execution time, as this is one of the most important considerations for someone using Cara.

Additionally, some inputs seem to crash the device running Cara when trials are solved in parallel due to memory constraints. This is circumvented currently by running the first trial alone and then only running the rest in parallel if it was fast enough (see Section 6.4.3). It may be worthwhile to have the program monitor its memory usage and figure out more precisely when this happens and how to deal with it.

6.5.3 User interface

It would be convenient to set up Cara as a single executable.

It would be nice to clean up the structure of the modules and turn them into an unified PyPI package for anyone to use.

We could rethink the input format. Currently, Cara only takes a text file with a hard-to-parse structure. The only variable in the auction setting that is easy to vary is the number of bidders. If a user wants to change any other variables, she must write different text files. It would be nice for the Cara modules to have a unified abstraction that stores all the desired auctions to run across varying variables and parameters. Then we could rethink the input text file format or perhaps create a web app that walks a user through her desired outcome.

6.5.4 Returned Data

There are two pieces of data that Cara does not provide the user that would be useful. First, the program does not return each discretized auction's actual optimal mechanism. Second, the program does not precisely return the full discretized distribution; it just returns the support that all of the discretized valuation distributions use. It would not be too difficult to add both of these pieces of data.

6.5.5 Upper Bounding

Cara currently only lower bounds the optimal revenue. It would be easy to provide upper bounds as well, by running discretized auctions that stochastically dominate the original. Then the program could provide ranges within which it is reasonable to think the optimal revenue is. The issue with upper bounding is handling putting a point at infinity.

6.5.6 Revenue Monotonicity

It would be interesting to see if we could look at the optimal mechanisms of each of the discretized trials to see if revenue monotonicity holds. Perhaps there is some way of measuring how approximately deterministic and symmetric or submodular the mechanism is that would bear on our confidence that the revenue of the discretized trial lower bounds the optimal revenue of the auction setting in question.

6.5.7 Smart Approximations

Currently, Cara is a naive tool; she just carries out the precise instructions given in the parameters of the discretization type. It would be nice to add some smart abilities to Cara. Perhaps the iterative process described in Section 5.2.2 could be automated with a searching algorithm for each new point. Cara could give live results as to the current best revenue, and the user could just terminate the program whenever she wants. There are many other ways in which we could make Cara automate the approximation schemes. All of these options require changing the current engineering substantially.

Appendix A

Timing

A.1 Support Size

In Table A.1, we give the average time Cara takes to run a discretized trial for the equal revenue auction setting as we vary the number of point masses. Each discretized trial finds the best auction for each of the following number of bidders: 1, 20, 40, 60, 80, 100, 120, 140, 160, 180, and 200. We fixed the NumPy Random seed to 0 so that both the linear and parallel attempts were solving for identical discretized auctions.

Here is the input text, where `[X]` is the size of the support:

```
1 2 1 1 20 0
1 1;20;40;60;80;100;120;140;160;180;200
D1
D1
1-1/x   1 inf   rf [X] 2 [X]-2   1 [X]-2   1 100   1 1_000_000


parallel=[True/False]
```

A.2 Number of Items

In Table A.2, we give the average time Cara takes to run a discretized trial of the auction setting described in Section 5.1 with two bidders while varying the number of items. More specifically, we have two i.i.d. bidders competing to purchase m i.i.d. items that they value uniformly in $[0, 1]$, independently across items. We discretize the one continuous valuation distribution into three

Support Size	Linear Time (s)	Parallel Time (s)
1	0.43	0.22
2	0.45	0.25
3	0.51	0.24
4	0.62	0.32
5	1.18	2.12
6	24.72	1.26
7	44.34	3.05
8	76.45	6.13
9	56.14	12.87
10	86.49	32.53
11	122.57	72.10
12	348.84	228.49
13	1462.09	853.43

Table A.1: Average discretized trial runtime for the equal revenue auction setting while varying the size of the support.

Items	Time (s)
1	0.19
2	0.21
3	1.18
4	6.02
5	N/A

Table A.2: Average discretized trial runtime for the auction setting described in Section 5.1 while varying the number of items.

points uniformly chosen in $[0, 1]$. For five items, we stopped Cara's execution after three hours. It is easy to see that increasing the number of items quickly makes the problem intractable.

Here is the input text for when the number of items is three:

```
1 3 1 1 1 0
1 2
D1
D1
D1
x 0 1 random 3 0 1
```

Bidders	PPF Time (it/s)	No PPF Time (it/s)
2	837.16	176.00
10	739.93	50.93
20	647.01	26.92
30	567.39	18.32
40	514.65	13.77
50	471.11	11.19
60	428.57	8.24
70	389.85	7.89
80	365.20	7.00
90	339.86	6.26
100	309.65	5.64

Table A.3: Average number of simulated second-price auction trials per second on the equal revenue auction setting while varying the number of bidders.

A.3 Second-Price Auctions

In Table A.3, we give the average number of second-price auction trials per second that Cara can simulate on the equal revenue auction setting as we vary the number of bidder. In the "PPF" column, we include the PPF (inverse of CDF) of the equal revenue distribution by writing `1-1/x|1/(1-y)` for the input distribution (see Section 4.3). In the "No PPF" column, we do not include the PPF, forcing Cara to simulate the second-price auctions by sampling the distribution using SciPy's `stats.rv_continuous` class, which is substantially slower.

Here is the input text for the "PPF" column, where `[X]` is the number of bidders:

```
1 2 1 1 0 1_000
1 [X]
D1
D1
1-1/x|1/(1-y)    1 inf    N/A
```

Bibliography

[Mye81] Roger B. Myerson. "Optimal auction design". In: *Mathematics of Operations Research* (1981).

[CDW11] Yang Cai, Constantinos Daskalakis, and S. Matthew Weinberg. "A Constructive Approach to Reduced-Form Auctions with Applications to Multi-Item Mechanism Design". In: (2011). URL: https://arxiv.org/abs/1112.4572.

[DW12] Constantinos Daskalakis and S. Matthew Weinberg. "Symmetries and Optimal Multi-Dimensional Mechanism Design". In: *Proceedings of the 13th ACM Conference on Electronic Commerce*. EC '12. Association for Computing Machinery, 2012, pp. 370–387. ISBN: 9781450314152. URL: https://doi.org/10.1145/2229012.2229042.

[HR15] Sergiu Hart and Philip Reny. "Maximal revenue with multiple goods: Nonmonotonicity and other observations". In: *Theoretical Economics* (2015). URL: http://www.ma.huji.ac.il/hart/papers/monot-m.pdf.

[Düt+17] Paul Dütting et al. "Optimal Auctions through Deep Learning". In: *CoRR* abs/1706.03459 (2017). arXiv: 1706.03459. URL: http://arxiv.org/abs/1706.03459.

[HN17] Sergiu Hart and Noam Nisan. "Approximate revenue maximization with multiple items". In: *Journal of Economic Theory* 172 (Nov. 2017), pp. 313–347. URL: https://doi.org/10.1016/j.jet.2017.09.001.

[BW18] Hedyeh Beyhaghi and S. Matthew Weinberg. "Optimal (and Benchmark-Optimal) Competition Complexity for Additive Buyers over Independent Items". In: *CoRR* abs/1812.01794 (2018). arXiv: 1812.01794. URL: http://arxiv.org/abs/1812.01794.

[Bar19] Rebecca Barber. "Bounding the Competition Complexity for Additive Buyers over Two Independent Items". In: *Princeton University Undergraduate Thesis* (2019). URL: https://www.dropbox.com/sh/29v01hshy74666w/AADUGkN_auxs3OjZhmRfXhs1a?dl=0&preview=thesis_v18.pdf.

[BW19] Hedyeh Beyhaghi and S. Matthew Weinberg. "Optimal (and Benchmark-Optimal) Competition Complexity for Additive Buyers over Independent Items". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2019. Association for Computing Machinery, 2019, pp. 686–696. ISBN: 9781450367059. URL: https://doi.org/10.1145/3313276.3316405.

[Kot+19] Pravesh Kothari et al. "Approximation Schemes for a Buyer with Independent Items via Symmetries". In: *CoRR* abs/1905.05231 (2019). arXiv: 1905.05231. URL: http://arxiv.org/abs/1905.05231.

[Shu19] Mel Shu. "Computation of Optimal Auctions". In: *Princeton University Undergraduate Thesis* (2019). URL: https://github.com/melyshu/roasolver.

[RJW20] Jad Rahme, Samy Jelassi, and S. Matthew Weinberg. "Auction learning as a two-player game". In: *CoRR* abs/2006.05684 (2020). arXiv: 2006.05684. URL: https://arxiv.org/abs/2006.05684.