PRINCETON UNIVERSITY

SENIOR THESIS

Computation of Optimal Auctions

An efficient implementation of computing revenue-optimal auctions via reduced forms

Author:
Mel Y. SHU

Advisor:
S. Matthew WEINBERG

A thesis submitted in partial fulfillment of the requirements
for the degree of Bachelor of Arts

in the

Department of Computer Science

May 6, 2019

Abstract

While little is known about revenue-optimal auctions in a multi-bidder, multi-item setting, computational approaches are able to provide some insights into their structures. Cai, Daskalakis, and Weinberg describe in [CDW11] an efficient way of calculating such auctions by solving a linear program with a separation oracle, using reduced forms to make the problem more tractable. In this thesis, we present RoaSolver, a software program that implements the ideas set forth in that paper.

Acknowledgements

First and foremost, I would like to thank my advisor, Matt Weinberg, for being such a wonderful teacher and mentor, guiding me through my work this year, from selecting a topic to completing this thesis. He was always accessible and took the time and effort to ensure that this project stayed on track. It is no exaggeration to say that this thesis would not have been possible without him!

I also want to thank my small army of proofreaders: Justin Yan, Prabha Upreti, James Currah, Angela Yang, and Mandy Yang. Their sharp eyes helped catch many mistakes that otherwise would not have been corrected.

Finally, to everyone in Chapel Choir, VTone, Colonial Club, and the extended Princeton community, thank you for all the memories over the last four years.

Contents

Chapter 1

Introduction

In this chapter, we introduce the reader to the contents and structure of this thesis. We discuss the background of the problem we are concerned with, present a brief summary of our solution, and provide a roadmap of the remainder of this thesis.

1.1 Problem background

In this thesis, we present a software program called *RoaSolver*, which efficiently computes the revenue-optimal auction, given the parameters of the auction setting as input. In this section, we discuss the theoretical motivations behind the development of RoaSolver.

1.1.1 Motivation

The problem of designing revenue-optimal auctions in a multi-bidder, multi-item setting is one that auction theorists have found very difficult to tackle, and little is understood about the structure of such auctions. In light of this difficulty, computation-based methods provide a practical way to gain numerical insights into these auctions. One possible approach is to construct a linear program, specifying constraints on the parameters of the auction and optimizing for revenue, as detailed in Section 2.3.5. In theory, the revenue-optimal auction would be output as a solution to this linear program, but in practice, this approach requires an exponential number of variables with respect to the size of the input, and so quickly becomes intractable.

To combat this issue, [CDW11] proposes to use *reduced forms*, an alternate way of describing auctions that requires only a polynomial number of variables with respect to the size of the input, thereby reducing the size of the corresponding linear program that needs to be solved. The difficulty with using reduced forms is that not all reduced forms are *feasible*. That is, some reduced forms do not actually specify valid auctions. It is not immediately obvious whether a reduced form is feasible, but [CDW11] shows that feasibility can be determined by a polynomial number of constraints, and that these constraints can be checked by a *separation oracle* that runs in linearithmic time. It is therefore possible to use this separation oracle with the smaller linear program built on reduced forms to more efficiently compute the revenue-optimal auction, as detailed in Section 2.4.3.

1.1.2 Related work

The primary reference for this thesis is a paper by Cai, Daskalakis, and Weinberg, titled *A Constructive Approach to Reduced-Form Auctions with Applications to Multi-Item Mechanism Design*, which we cite as [CDW11]. As we discussed above, the paper lays down the theoretical ideas that form the main motivation for this thesis, and RoaSolver is in essence an implementation of these ideas. We encourage the reader to refer to the paper for a more complete understanding of the problem background.

The secondary reference for this thesis is a publication by Daskalakis and Weinberg, titled *Symmetries and Optimal Multi-Dimensional Mechanism Design*, which we cite as [DW12]. We cite this work to include some theoretical ideas not already covered by [CDW11].

1.1.3 Goal

The goal of this thesis is to develop a software program capable of more efficiently computing revenue-optimal auctions by implementing the separation oracle and linear program presented in [CDW11], so that auction theorists can better understand the structures behind these mechanisms.

1.2 Implementation

In this section, we give a brief overview of RoaSolver. We encourage the reader to refer to Chapter 4 for a more detailed discussion of the source code.

1.2.1 Approach

RoaSolver is an implementation of the separation oracle and linear program specified in [CDW11]. The program is written in Java and can be run as an executable JAR. Internally, we rely on the *IBM ILOG CPLEX Optimization Studio* to solve the linear program. The bulk of the source code for RoaSolver deals with setting up the constraints and separation oracle for a given problem and feeding it into the CPLEX model.

1.2.2 Results

We have checked the correctness of RoaSolver by running it on a set of small problems that are solvable by hand and verifying that the output is numerically correct. We have also checked that the program can be run for extended periods of time by testing it on inputs that took many hours to run. The program typically takes less than one second to run on inputs with under 1,000 constraints. For inputs with over 1,000 constraints, the program can take anywhere from milliseconds to minutes to run, depending on how many times the separation oracle runs, a quantity that is hard to predict. Refer to Appendix A for more detailed timing data.

1.2.3 Limitations

It is very easy to specify problem settings that take many minutes or even hours to run. Since the bulk of the run time is taken up by the CPLEX solver, there is a very limited amount of room for reducing the run time unless we can further decrease the number of variables in the linear program, or somehow reduce the number of times the separation oracle is called. In addition, while RoaSolver has produced the correct output for all reasonable inputs we have tested, on some extreme inputs, such as specifying 10,000,000 bidders, we have noticed that RoaSolver may output nonsensical figures.

1.3 Roadmap

In this section, we provide a brief overview of the contents of the remainder of this thesis, as well as the intended audience of each chapter.

- Chapter 2 is written for readers who are unfamiliar with auction theory, and establishes the core theoretical concepts from the ground up.
- Chapter 3 is written for readers who are already familiar with the material in Chapter 2, and provides instructions to run RoaSolver.
- Chapter 4 is written for readers who are already familiar with both the theory and the execution of RoaSolver, and provides documentation on the source code that may be helpful for future development.
- Appendix A contains tables detailing how long RoaSolver takes to run on inputs of varying sizes.

Chapter 2

Background Theory

In this chapter, we introduce the reader to the relevant terminology and auction theory that is required to understand RoaSolver. The material presented is predominantly drawn from [CDW11], with some notation changes for our own convenience. We make a special effort to explain everything in a detailed manner, so that a reader without any background in this area can learn the concepts put forward. A more informed reader may find it useful to briefly consult this chapter to become familiar with the vocabulary and notation used.

2.1 Introduction

In this section, we introduce auctions and discuss an auction format that the reader might already be familiar with.

2.1.1 Auctions

An *auction* is a process in which a number of bidders compete to buy one or more items by submitting bids. There are many different ways to run an auction, and therefore many different *auction formats*. Before we discuss the auction format relevant to this thesis, we first describe an example that most readers would already be familiar with, to start using some of the terminology and to provide a point of reference.

2.1.2 Auction format example

The *open ascending price auction* is a common auction format found in auction houses. In this auction format, the bidders compete to buy one item by submitting bids over a period of time. A bidder may submit a bid at any time, providing it is higher than all previous bids, and the value of all bids are disclosed to all bidders immediately. Each bid is a price that the bidder would be willing to pay for the item being auctioned. When no more bidders are willing to make bids, the auctioneer declares the winner as the bidder who submitted the highest bid. That bidder is then required to pay the value of their winning bid, and in exchange receives the item.

There are substantial differences between the open ascending price auction and the auction auction we are concerned with in this thesis. We list some, but not all, of the differences below:

- The bidders are only allowed to bid once.
- The bidders do not disclose their bids to one another.
- The bids cannot be arbitrary values but must be chosen from a set of predetermined values.
- There are potentially multiple items being sold all at once.
- The items are not necessarily given to the highest bidder.
- A bidder may be required to pay even if they do not receive an item.
- A bidder who receives an item does not necessarily pay the value of their bid for that item.

2.2 Auction format

In this section, we describe in detail the auction format that is relevant to this thesis. We define the terminology and notation that we use, and also explain how the auction is run from start to finish.

2.2.1 Auctioneer

Firstly, as a matter of convenience, we may sometimes find it useful to think of the auction as being run by an *auctioneer*. To begin with, we assume that the auctioneer only has the power to specify the parameters of the auction before it is run. Once the auction begins, however, the auctioneer simply performs the execution as specified beforehand, and does not actively participate in the auction itself, even if the language we use suggests that they have the power to change the outcome with their actions.

2.2.2 Bidders and items

There are one or more *bidders* participating in the auction, and one or more *items* the auctioneer wishes to sell to the bidders. We use n to denote the number of bidders, and m to denote the number of items. For convenience, we denote $\mathcal{B} = \{1, 2, \ldots, n\}$ and $\mathcal{I} = \{1, 2, \ldots, m\}$, so that each bidder is given the label "bidder b", for some $b \in \mathcal{B}$, and each item is given the label "item i", for some $i \in \mathcal{I}$. We sometimes use the term *participants* to refer to both the bidders and the auctioneer.

2.2.3 Valuations and types

In the setup phase before the auction takes place, each bidder must determine their *true valuation* of each of the items. The true valuation can be thought of as the actual value the bidder thinks the item is worth, or alternatively, the utility gained if they[1] were to receive that item. The true valuations are *additive*, so that from a certain bidder's point of view, the utility gained from receiving a set of items is exactly the sum of their true valuations of each item they receive.

In our auction format, each bidder's true valuation of each item is determined by what *type* they have. A type can be thought of as a collection of m valuations, one for each item, that specifies how much each item is worth. The number of possible types across all bidders is finite and denoted by c. For convenience, we denote $\mathcal{T} = \{1, 2, \ldots, c\}$, so that each type is given the label "type t", for some $t \in \mathcal{T}$. These c types are predetermined, and we refer to them collectively as the *type collection*, denoted by $\mathbf{v}$. They are defined by $v_{it} \in \mathbb{R}$, for all $i \in \mathcal{I}$ and $t \in \mathcal{T}$, indicating type t's valuation of item i. It may be useful to think of using $(v_{1t}, v_{2t}, \ldots, v_{mt}) \in \mathbb{R}^m$ to represent type t.

Each bidder has their own predetermined *type distribution* over $\mathcal{T}$ that specifies how likely they are to have each type. We use D_b to denote bidder b's type distribution, and $t \sim D_b$ to indicate that

[1]The pronouns we use for a single bidder are "they", "them", and "their".

type $t \in \mathcal{T}$ is drawn from D_b. The type distributions are defined by $p_{bt} = \mathbb{P}_{t' \sim D_b}[t' = t]$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$, indicating the probability of bidder b drawing type t. Before the auction is run, each bidder independently draws their *true type* from their type distribution, thus determining their true valuations of each of the items and completing the setup phase of the auction. It is important to note that since a type specifies valuations for all items, it is possible that a bidder's true valuations are not independently drawn across items.

For convenience, we use the term *type profile* to refer to some $\mathbf{t} = (\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_n) \in \mathcal{T}^n$, indicating a type for each of the n bidders. Thus, the *true type profile* is sampled from the distribution $\mathbf{D} = \bigtimes_{b \in \mathcal{B}} D_b$. Furthermore, we define

$$\mathbf{D}_b(t) = \bigtimes_{\substack{b' \in \mathcal{B} \\ b' < b}} D_{b'} \times (t \text{ w.p. } 1) \times \bigtimes_{\substack{b' \in \mathcal{B} \\ b' > b}} D_{b'},$$

for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$, to be the distribution where $\mathbf{t} \sim \mathbf{D}_b(t)$ draws a type profile where bidder b draws type t with certainty, and all other bidders independently draw their types from their type distribution.

2.2.4 Bidding

After the setup phase, each bidder simultaneously submits a single *sealed bid* to the auctioneer, in the form of a *reported type*. It is important to note that each bid is sealed, which means that only the bidder and the auctioneer see it. In addition, the bid itself is a type from the type collection, specifying the corresponding bidder's *reported valuation* for each item. Importantly, the bidder is not required to bid their true type. They are permitted to bid any of the c predetermined types, and so their reported valuations could differ from their true valuations. The auctioneer thus receives a *reported type profile* as the outcome of the bidding phase.

2.2.5 Allocation

After each of the bidders has submitted their bid, the auctioneer must decide which items to give to which bidders, specified in the form of an *allocation*. This allocation can be thought of as some $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m) \in (\mathcal{B}_\perp)^m$, where we define $\mathcal{B}_\perp = \mathcal{B} \cup \{\perp\}$. The allocation specifies that for each item $i \in \mathcal{I}$, if $\mathbf{b}_i = \perp$, that item is not given to any bidder, otherwise it is given to bidder $\mathbf{b}_i$. An *allocation distribution* is a distribution over $(\mathcal{B}_\perp)^m$ that can be sampled to obtain an allocation. In our case, we only ever consider allocation distributions that are independent across items. We use $\mathcal{A}$ to denote the set of all such allocation distributions.

An *ex-post allocation rule* specifies an allocation distribution as a function of the reported type profile. It is denoted by $\boldsymbol{\pi}^*$ and defined by $\pi^*_{bi}(\mathbf{t}) \in \mathbb{R}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$, indicating the probability with which bidder b receives item i, given a reported type profile $\mathbf{t} \in \mathcal{T}^n$. We may think of the ex-post allocation rule as the function $\boldsymbol{\pi}^* : \mathcal{T}^n \to \mathcal{A}$ such that

$$\boldsymbol{\pi}^*(\mathbf{t}) = \bigtimes_{i \in \mathcal{I}} \left(b \text{ w.p. } \pi^*_{bi}(\mathbf{t}), \text{ for all } b \in \mathcal{B}, \text{ and } \perp \text{ otherwise}\right).$$

Note that not all ways of defining $\boldsymbol{\pi}^*$ via the values of $\pi^*_{bi}(\mathbf{t})$ output valid allocation distributions. The ex-post allocation rule must also satisfy the following constraints, to ensure that this is the case:

- $\pi^*_{bi}(\mathbf{t}) \geq 0$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$.
- $\sum_{b \in \mathcal{B}} \pi^*_{bi}(\mathbf{t}) \leq 1$, for all $i \in \mathcal{I}$ and $\mathbf{t} \in \mathcal{T}^n$.

These constraints ensure that the $\pi^*_{bi}(\mathbf{t})$ are valid probabilities for an allocation distribution.

An ex-post allocation rule is used by the auctioneer to determine the actual allocation that occurs, which we refer to as the *final allocation*. This ex-post allocation rule is nominated before the auction and disclosed to all bidders, so that they may bid accordingly. After all bids are submitted, the auctioneer applies the ex-post allocation rule to the reported type profile, thereby obtaining an allocation distribution. This allocation distribution is then sampled to obtain the final allocation.

2.2.6 Payment

Finally, the auctioneer must decide what price the bidders must pay for participating in the auction, specified in the form of a *payment rule*. In our auction format, a payment rule specifies the price each bidder pays as a function of their reported type. It is denoted by $\mathbf{q}$ and defined by $q_{bt} \in \mathbb{R}$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$, indicating the price that bidder b must pay the auctioneer if they bid type t. We may think of the payment rule itself consists of n functions $q_b : \mathcal{T} \to \mathbb{R}$, one for each $b \in \mathcal{B}$, defined by $q_b(t) = q_{bt}$, so that q_b is the function that determines the price bidder b pays as a function of their reported type. Note that other bidders' reported types, as well as the final allocation, have no impact on the price that a bidder pays. We discuss this further in Section 2.5.1.

2.3 Auction behavior

In this section, we discuss how the behavior of each participant in the auction is governed by the information available to them and by their incentives.

2.3.1 Information accessibility

We first explicitly state what information is available to which participants at each point in time.

All participants have access to the following information from the very beginning, in addition to knowledge of the format of the auction as specified in Section 2.2:

- The number of bidders, n.
- The number of items, m.
- The number of types, c.
- The type collection, $\mathbf{v}$, defined by v_{it}, for all $i \in \mathcal{I}$ and $t \in \mathcal{T}$.
- Each bidder's type distribution D_b, defined by p_{bt}, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$.

We refer to these variables collectively as the *predefined constants* which define the *auction setting*.

Given these predefined constants, the auctioneer determines the exact auction to be run by specifying the following information:

- The ex-post allocation rule, $\boldsymbol{\pi}^*$, defined by $\pi^*_{bi}(\mathbf{t})$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$.
- The payment rule, $\mathbf{q}$, defined by q_{bt}, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$.

This information, referred to collectively as the *parameters* of the auction, is disclosed to all bidders before the setup phase.

Afterwards, each bidder draws their true type, which is known to them but disclosed to no other participants. With this information and knowledge of the predefined constants and the auction parameters, each bidder simultaneously submits a bid to the auctioneer. The bids are sealed and thus each bid is only seen by the corresponding bidder and the auctioneer. The auctioneer, upon receiving the reported type profile, carries out the auction specified by the parameters chosen earlier, to arrive at the final allocation.

2.3.2 Bidder incentives

Each bidder has a *payoff* or *utility* from participating in the auction. This can be loosely thought of as how much that bidder is better off after the auction, as compared to before the auction. We make a more precise definition below, but we first note that bidders prefer higher payoffs, and are thus incentivized to act in a way that increases their payoffs.

Consider a situation where the auction has already taken place, with final allocation $\mathbf{b} \in (\mathcal{B}_\perp)^m$, and where bidder b has true type t and reported type t'. Let $I_b = \{i \in \mathcal{I} : \mathbf{b}_i = b\}$ be the indices of the items that bidder b receives. Then bidder b's payoff is given by $\sum_{i \in I_b} v_{it} - q_{bt'}$, the sum of all their true valuations for the items they received, minus the price they paid.

However, the bidders do not know each other's true or reported types before they submit their bids. It therefore makes sense to calculate a bidder's *expected payoff* from submitting a certain bid. This is the value of the payoff, except in expectation over the allocation distribution and the other bidders drawing true types from their own type distributions, assuming that all other bidders bid their true types. We define bidder b's expected payoff when they have true type t and reported type t' to be

$$\begin{aligned} J_b^*(t,t') &= \mathop{\mathbb{E}}_{\mathbf{t} \sim \mathbf{D}_b(t')} \left[\sum_{i \in \mathcal{I}} \pi^*_{bi}(\mathbf{t}) v_{it} \right] - q_{bt'} \\ &= \sum_{\substack{\mathbf{t} \in \mathcal{T}^n \\ \mathbf{t}_b = t'}} \left(\prod_{\substack{b' \in \mathcal{B} \\ b' \neq b}} p_{b' \mathbf{t}_{b'}} \cdot \sum_{i \in \mathcal{I}} \pi^*_{bi}(\mathbf{t}) v_{it} \right) - q_{bt'}. \end{aligned}$$

We assume that the bidders participating in the auction wish to maximize their expected payoffs. Thus, if bidder b has true type t, they should bid a type t' that maximizes $J_b^*(t,t')$.

2.3.3 Auction properties

Of course, we must remember that the calculation of each bidder's expected payoff assumes that all other bidders bid their true types. If each bidder maximizes their expected payoff and simultaneously bids their true type, we must have $J_b^*(t,t) \geq J_b^*(t,t')$, for all $b \in \mathcal{B}$ and $t, t' \in \mathcal{T}$. This property is known as *Bayesian Incentive Compatibility*, and an auction that satisfies it is said to be *Bayesian Incentive Compatible*, or simply *BIC*. An auction that is BIC achieves a consistency as in a Bayesian Nash equilibrium. That is, from each bidder's point of view, there is no incentive to bid something other than their true type, assuming that other bidders also bid truthfully.

In addition, we would like to be able to assume that all bidders would willingly participate truthfully in the auction. That is, each bidder's expected payoff for bidding truthfully should be nonnegative, or $J_b^*(t,t) \geq 0$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$. This property is known as *Individual Rationality*, and an auction that satisfies it is said to be *Individually Rational*, or simply *IR*. An auction that is IR prohibits the auctioneer from setting arbitrarily high prices just because the bidders are forced to participate.

2.3.4 Auctioneer incentives

Since the auctioneer is responsible for specifying the parameters of the auction, they can also be thought of as having a payoff, namely the sum of the payments collected from the bidders. If $\mathbf{t} \in \mathcal{T}^n$ is the reported type profile, then $\sum_{b\in\mathcal{B}} q_{b\mathbf{t}_b}$ is the auctioneer's payoff, also known as the *revenue* of the auction.

However, the auctioneer does not know the bidders' true or reported types before specifying the auction parameters. It therefore makes sense to calculate the *expected revenue*, the revenue in expectation over the bidders reporting their true types. We define this to be

$$\begin{aligned} J &= \mathop{\mathbb{E}}_{\mathbf{t}\sim\mathbf{D}}\left[\sum_{b\in\mathcal{B}} q_{b\mathbf{t}_b}\right] \\ &= \sum_{\mathbf{t}\in\mathcal{T}^n}\left(\prod_{b\in\mathcal{B}} p_{b\mathbf{t}_b} \cdot \sum_{b\in\mathcal{B}} q_{b\mathbf{t}_b}\right). \end{aligned}$$

We assume that the auctioneer wishes to maximize this expected revenue. That is, given the predefined constants, they wish to specify an allocation rule and a payment rule that maximize J.

We must remember that the calculation of the expected revenue assumes that all bidders willingly bid truthfully. To ensure that this is the case, it is up to the auctioneer to specify an auction that has the additional properties of being BIC and IR.

2.3.5 Linear program

In this thesis, we are primarily concerned with taking the place of the auctioneer and adjusting auction parameters to maximize the expected revenue, under the condition that the auction must be both BIC and IR. We use the term *optimization problem* to refer to this maximization, and the term *revenue-optimal auction* or simply *optimal auction* to refer to the corresponding auction with the *optimal parameters*. The optimization problem can be posed as a linear program, which we describe below.

Given the predefined constants of the auction setting $n, m, c, \{v_{it}\}_{i \in \mathcal{I}, t \in \mathcal{T}}, \{p_{bt}\}_{b \in \mathcal{B}, t \in \mathcal{T}}$, we define the following variables to be used in the linear program:

- $\pi^*_{bi}(\mathbf{t}) \in \mathbb{R}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$. These variables are the probabilities specifying the ex-post allocation rule $\boldsymbol{\pi}^*$. (mnc^n variables)
- $q_{bt} \in \mathbb{R}$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$. These variables are the prices specifying the payment rule $\mathbf{q}$. (nc variables)

We impose the following constraints in the linear program:

- $\pi^*_{bi}(\mathbf{t}) \geq 0$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$. These constraints ensure that the $\pi^*_{bi}(\mathbf{t})$ are all probabilities. (mnc^n constraints)
- $\sum_{b \in \mathcal{B}} \pi^*_{bi}(\mathbf{t}) \leq 1$, for all $i \in \mathcal{I}$ and $\mathbf{t} \in \mathcal{T}^n$. These constraints ensure that $\boldsymbol{\pi}^*$ is a valid ex-post allocation rule. (mc^n constraints)
- $\sum_{\substack{\mathbf{t} \in \mathcal{T}^n \\ \mathbf{t}_b = t}} \left(\prod_{\substack{b' \in \mathcal{B} \\ b' \neq b}} p_{b'\mathbf{t}_{b'}} \cdot \sum_{i \in \mathcal{I}} \pi^*_{bi}(\mathbf{t}) v_{it} \right) - q_{bt} \geq 0$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$. These constraints ensure that the auction is IR. (nc constraints)

- $\sum_{\substack{\mathbf{t}\in\mathcal{T}^n\\ \mathbf{t}_b=t}} \left(\prod_{\substack{b'\in\mathcal{B}\\ b'\neq b}} p_{b'\mathbf{t}_{b'}} \cdot \sum_{i\in\mathcal{I}} \pi^*_{bi}(\mathbf{t})v_{it}\right) - q_{bt} \;\geq\; \sum_{\substack{\mathbf{t}\in\mathcal{T}^n\\ \mathbf{t}_b=t'}} \left(\prod_{\substack{b'\in\mathcal{B}\\ b'\neq b}} p_{b'\mathbf{t}_{b'}} \cdot \sum_{i\in\mathcal{I}} \pi^*_{bi}(\mathbf{t})v_{it}\right) - q_{bt'}$, for all $b \in \mathcal{B}$ and $t, t' \in \mathcal{T}$. These constraints ensure that the auction is BIC. (nc^2 constraints)

Finally, we nominate the objective function to be maximized:

- $\sum_{\mathbf{t}\in\mathcal{T}^n} \left(\prod_{b\in\mathcal{B}} p_{b\mathbf{t}_b} \cdot \sum_{b\in\mathcal{B}} q_{b\mathbf{t}_b}\right)$. This is the expected revenue.

2.4 Reduced forms

In this section, we discuss *reduced-form auctions*, or simply *reduced forms*, an alternate way of specifying an auction. Reduced forms are particularly useful in the context of this thesis since they simplify the optimization problem.

2.4.1 Definition

A reduced form is an auction where the allocation is specified by an *interim allocation rule* instead of an ex-post allocation rule. An interim allocation rule is denoted by $\boldsymbol{\pi}$ and defined by the values $\pi_{bi}(t) \in \mathbb{R}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$, indicating the probability that bidder b gets item i when bidding type t, in expectation over all other bidders bidding their true types. We refer to this as the *interim probability* that bidder b gets item i when bidding type t. Note that an interim allocation rule is defined by mnc values, much fewer than the mnc^n values required for an ex-post allocation rule, a property we can use to our advantage to simplify the optimization problem.

2.4.2 Feasibility

Given an ex-post allocation rule $\boldsymbol{\pi}^*$ defined by $\pi^*_{bi}(\mathbf{t}) \in \mathbb{R}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $\mathbf{t} \in \mathcal{T}^n$, we say that $\boldsymbol{\pi}^*$ *induces* the interim allocation rule $\boldsymbol{\pi}$ defined by

$$\begin{aligned}\pi_{bi}(t) &= \mathop{\mathbb{E}}_{\mathbf{t}\sim\mathbf{D}_b(t)} \pi^*_{bi}(\mathbf{t}) \\ &= \sum_{\substack{\mathbf{t}\in\mathcal{T}^n\\ \mathbf{t}_b=t}} \left(\prod_{\substack{b'\in\mathcal{B}\\ b'\neq b}} p_{b'\mathbf{t}_{b'}} \cdot \pi^*_{bi}(\mathbf{t})\right),\end{aligned}$$

for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. Since $\boldsymbol{\pi}^*$ can be executed to determine the final allocation, it is clear that the induced interim allocation rule $\boldsymbol{\pi}$ can thus be executed in the same way. We call an interim allocation rule *feasible* if there exists a valid ex-post allocation rule that induces it, so that it can be executed to determine the final allocation.

Suppose conversely that we are given the values $\pi_{bi}(t)$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$, to define an interim allocation rule $\boldsymbol{\pi}$. Under what conditions is $\boldsymbol{\pi}$ feasible? It is shown in [CDW11] that the specific problem of determining whether an interim allocation rule is feasible can be independently[2] solved across items. That is, if we use the phrase *projected interim allocation rule* of item i to refer to the probabilities $\{\pi_{bi}(t)\}_{b\in\mathcal{B},t\in\mathcal{T}}$ that determine in isolation how item i is allocated in a single-item sense, then an interim allocation rule is feasible if and only if all of the projected interim allocation rules are feasible.

Using a result known as Border's theorem, [CDW11] shows us how to determine the feasibility of a projected interim allocation rule. Firstly, define the sets

$$\begin{aligned} S_{bi}(x) &= \left\{ t \in \mathcal{T} : \pi_{bi}(t) \underset{t'\sim D_b}{\mathbb{P}}[\pi_{bi}(t) \geq \pi_{bi}(t')] \geq x \right\} \\ &= \left\{ t \in \mathcal{T} : \pi_{bi}(t) \sum_{\substack{t' \in \mathcal{T} \\ \pi_{bi}(t) \geq \pi_{bi}(t')}} p_{bt'} \geq x \right\}, \end{aligned}$$

for all $b \in \mathcal{B}$ and $i \in \mathcal{I}$, as a function of $x \in \mathbb{R}$. For a given item $i \in \mathcal{I}$, the projected interim allocation rule of that item is feasible if and only if for all $x \in \mathbb{R}$ it is true that

$$\sum_{b\in\mathcal{B}} \sum_{t\in S_{bi}(x)} \pi_{bi}(t) \underset{t'\sim D_b}{\mathbb{P}}[t' = t] \leq 1 - \prod_{b\in\mathcal{B}} \left(1 - \underset{t\sim D_b}{\mathbb{P}}[t \in S_{bi}(x)] \right),$$

or more explicitly,

$$\sum_{b\in\mathcal{B}} \sum_{t\in S_{bi}(x)} \pi_{bi}(t) p_{bt} \leq 1 - \prod_{b\in\mathcal{B}} \left(1 - \sum_{t\in S_{bi}(x)} p_{bt} \right). \tag{2.1}$$

Each of the constraints in Equation 2.1 above is called a *Border constraint*. Due to the discrete nature

[2]Note that in contrast, the interim probability that a particular bidder receives a particular item must depend on their valuations of other items and so is not necessarily independent!

of the sets $S_{bi}(x)$, it is only necessary to check the Border constraints for the threshold values of x in the set

$$X_i = \left\{ \pi_{bi}(t) \underset{t'\sim D_b}{\mathbb{P}}[\pi_{bi}(t) \geq \pi_{bi}(t')] : b \in \mathcal{B}, t \in \mathcal{T} \right\}$$
$$= \left\{ \pi_{bi}(t) \sum_{\substack{t' \in \mathcal{T} \\ \pi_{bi}(t) \geq \pi_{bi}(t')}} p_{bt'} : b \in \mathcal{B}, t \in \mathcal{T} \right\},$$

which we define for each item $i \in \mathcal{I}$. Each X_i contains at most c values of x that need to be checked for that item. For a few more notes on Border's theorem, refer to Section 2.5.2.

As a final consequence, [CDW11] shows that, given the predefined constants of the auction setting, it is possible to define a *separation oracle*, denoted by $\mathfrak{S}$, which determines whether an interim allocation rule is feasible in linearithmic time. The separation oracle takes an interim allocation rule $\boldsymbol{\pi}$ as input, and outputs one of the following:

- A hyperplane ℓ corresponding to a broken Border constraint, to indicate that $\boldsymbol{\pi}$ is infeasible.
- The signal YES, to indicate that all Border constraints are satisfied and thus that $\boldsymbol{\pi}$ is feasible.

2.4.3 Linear program

We can rewrite the linear program posed in Section 2.3.5 in terms of an interim allocation rule instead of an ex-post allocation rule. To do this, note that bidder b's expected payoff when they have true type t and reported type t' is now given by

$$J_b(t, t') = \sum_{i \in \mathcal{I}} \pi_{bi}(t') v_{it} - q_{bt'}.$$

Also, the properties of being BIC and IR are still defined analogously. The full specification of the rewritten optimization problem is described below.

Given the predefined constants of the auction setting $n, m, c, \{v_{it}\}_{i \in \mathcal{I}, t \in \mathcal{T}}, \{p_{bt}\}_{b \in \mathcal{B}, t \in \mathcal{T}}$, we define the following variables to be used in the linear program:

- $\pi_{bi}(t) \in \mathbb{R}$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. These variables are the interim probabilities specifying the interim allocation rule $\boldsymbol{\pi}$. (*mnc* variables)

- $q_{bt} \in \mathbb{R}$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$. These variables are the prices specifying the payment rule $\mathbf{q}$. (nc variables)

We impose the following constraints in the linear program:

- $0 \leq \pi_{bi}(t) \leq 1$, for all $b \in \mathcal{B}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. These constraints ensure that the $\pi_{bi}(t)$ are all probabilities. ($2mnc$ constraints)

- $\sum_{i \in \mathcal{I}} \pi_{bi}(t) v_{it} - q_{bt} \geq 0$, for all $b \in \mathcal{B}$ and $t \in \mathcal{T}$. These constraints ensure that the auction is IR. (nc constraints)

- $\sum_{i \in \mathcal{I}} \pi_{bi}(t) v_{it} - q_{bt} \geq \sum_{i \in \mathcal{I}} \pi_{bi}(t') v_{it} - q_{bt'}$, for all $b \in \mathcal{B}$ and $t, t' \in \mathcal{T}$. These constraints ensure that the auction is BIC. (nc^2 constraints)

We add the following separation oracle:

- $\mathfrak{S}$, so that if $\boldsymbol{\pi}$ satisfies all other constraints and $\mathfrak{S}(\boldsymbol{\pi}) = \ell$, we add the broken constraint ℓ to the linear problem and iterate. Requiring $\mathfrak{S}(\boldsymbol{\pi}) = \text{YES}$ ensures that there is an ex-post allocation rule inducing $\boldsymbol{\pi}$.

Finally, we nominate the objective function to be maximized:

- $\sum_{t \in \mathcal{T}} \sum_{b \in \mathcal{B}} p_{bt} q_{bt}$. This is the expected revenue.

2.5 Further results

In this section, we address some of the important results not fully covered elsewhere.

2.5.1 Payment rules

In Section 2.2.6 we defined the payment rule to be a function for each bidder, indicating how much they should pay based on their reported type. For convenience, we refer to such a payment rule as a *simple payment rule*. It is natural to think that the payment rule should also depend on which items the bidder receives in the final allocation. After all, our intuition from many real-life auctions is that bidders pay for the items they receive. However, if we were to use a *complex payment rule* instead, so that each bidder pays a price dependent on not just their reported type, but potentially also other bidders' types as well as the final allocation, we would not change the properties of the auction format. It can be shown that since the bidders do not have any information about other bidders' bids or the final allocation when calculating their expected payoff, the expected payment using a complex payment rule depends only on the bidder's own type and is thus equivalent to a simple payment rule. As a result, by using a simple payment rule, we reduce the number of variables required to describe an equivalent auction format with identical bidder behavior.

2.5.2 Border's theorem

Equation 2.1 in Section 2.4.2 uses a modified version of Border's theorem. The original version of the theorem, as stated in [CDW11], asserts that a projected interim allocation rule of an item $i \in \mathcal{I}$ is feasible if and only if, for all $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n) \in \mathbb{R}^n$, it is true that

$$\sum_{b \in \mathcal{B}} \sum_{\substack{t \in \mathcal{T} \\ \pi_{bi}(t) \geq \mathbf{x}_b}} \pi_{bi}(t) \underset{t' \sim D_b}{\mathbb{P}}[t' = t] \leq 1 - \prod_{b \in \mathcal{B}} \left(1 - \underset{t \sim D_b}{\mathbb{P}}[\pi_{bi}(t) \geq \mathbf{x}_b]\right),$$

or more explicitly,

$$\sum_{b \in \mathcal{B}} \sum_{\substack{t \in \mathcal{T} \\ \pi_{bi}(t) \geq \mathbf{x}_b}} \pi_{bi}(t) p_{bt} \leq 1 - \prod_{b \in \mathcal{B}} \left(1 - \sum_{\substack{t \in \mathcal{T} \\ \pi_{bi}(t) \geq \mathbf{x}_b}} p_{bt}\right).$$

In each Border constraint above, the left-hand side represents the probability that item i is allocated to some bidder b whose true type t satisfies $p_{bt} \geq \mathbf{x}_b$, and the right-hand side represents the probability that such a bidder exists. Thus, it is clear that if the interim allocation rule is feasible, the Border constraints must be satisfied. Border's theorem asserts that the converse is true as well, that if the Border constraints are satisfied, then the projected interim allocation rule is feasible.

In addition, [CDW11] proves two key insights that build on this result:

- Border's theorem only deals with the feasibility of single-item allocation rules. However, the feasibility of a multi-item auction, as is relevant to our particular auction format, can be determined by assessing the feasibility of each of the projected interim allocation rules using Border's theorem.

- Border's theorem requires checking the Border constraints for all $\mathbf{x} \in \mathbb{R}^n$, which amounts to checking about c^n constraints. However, through the well-constructed sets $S_{bi}(x)$, which are said to *shade* the reduced form, we only need to check a modified version of the Border constraints for all $x \in \mathbb{R}$, which amounts to checking about c constraints, a significant improvement.

We refer the reader to [CDW11] for more detailed proofs of these results.

2.5.3 Separation oracle

The separation oracle $\mathfrak{S}$ that we define in Section 2.4.2 determines the feasibility of an interim allocation rule by checking all the relevant Border constraints, which takes time on the order of $nmc \log(nc)$. While this is sufficient for our use of the separation oracle in the linear program, if the interim allocation rule is deemed to be feasible, it is in fact possible to describe an ex-post allocation rule inducing it in time polynomial in c and n. We refer the reader to [CDW11] for a more detailed proof of this result.

2.5.4 Bidder symmetry

In certain auction settings, it might be possible for *bidder symmetry* to exist. That is, two or more bidders may be indistinguishable since their type distributions are identical, and so relabelling these bidders in any permutation would still result in exactly the same auction setting. In this case, [DW12] shows that there exists an optimal auction that treats the indistinguishable bidders in the same symmetric way. That is, the probabilities defined by the allocation rule and the prices defined by the payment rule are identical for any bidders that are indistinguishable. Consequently, it is possible to take advantage of bidder symmetries to reduce the number of variables used to describe the auction, thereby reducing the size of the linear program. This is the primary motivation for using bidder classes in the input for RoaSolver, as detailed in Section 3.2.2.

2.5.5 Item symmetry

In certain auction settings, it might be possible for *item symmetry* to exist. That is, two or more items may be indistinguishable from each other since bidders may draw their valuations for them independently. In this case, relabelling these items in any permutation would result in exactly the same auction setting. Types whose valuations are identical under permutations allowed by this symmetry are thus equivalent. [DW12] shows that there exists an optimal auction that treats the indistinguishable items in the same symmetric way. That is, the probabilities defined by the allocation rule and the prices defined by the payment rule are identical for equivalent types. Consequently, it is possible to take advantage of item symmetries to reduce the number of variables used to describe the auction, thereby reducing the size of the linear program.

Chapter 3

Software Usage

In this chapter, we present a user manual for RoaSolver, the software program we have written to compute the optimal auction as specified by the optimization problem posed in Section 2.4.3. We provide instructions detailing how to set up a machine to execute the software program, assuming familiarity with the background theory laid out in Chapter 2. We also assume that the reader is able to use a web browser to access websites and download files, and has the basic skills to navigate the terminal on their machine.

3.1 Overview

We first give an overview of the contents of this chapter. Each section of this chapter discusses one step to run RoaSolver:

- Section 3.2 describes some of the additional terminology used in the context of this program.
- Section 3.3 describes how to install the dependencies required by RoaSolver.
- Section 3.4 describes how to create an input file for RoaSolver to read.
- Section 3.5 describes how to run RoaSolver on an input file.
- Section 3.6 describes how to read the output describing a solution generated by RoaSolver.

As a disclaimer, most of what is written in this chapter is based on running RoaSolver on a Windows 10 machine. There is no guarantee that these are the correct instructions for all other environments, so slight tweaks might be required to get RoaSolver running on other machines.

3.1.1 RoaSolver installation

RoaSolver exists as a standalone executable JAR and is available online in the RoaSolver Github repository[1]. Simply download the JAR into any local directory to be able to run it in that directory.

3.2 Additional terminology

In this section, we explain some of the language we use just for the purpose of running the program, in addition to the terminology laid out in Chapter 2.

3.2.1 Type classes

For ease of input, we think of each type as being defined by a *type class* into which it is grouped. In the setup phase of the auction, we think of each bidder as first drawing a type class, and then drawing a type from within that type class. There is a predefined distribution that determines the probabilities with which a bidder in a given type class draws each type in that type class. Crucially, the types are grouped into type classes such that this distribution is independent across items. That is, a bidder in a given type class can be thought of as drawing their valuations for each item independently.

We use $\hat{c}$ to denote the number of type classes. For convenience, we denote $\hat{\mathcal{T}} = \{1, 2, \ldots, \hat{c}\}$, so that each type class is given the label "type class $\hat{t}$", for some $\hat{t} \in \hat{\mathcal{T}}$. Each type class associates with each item a univariate distribution from which the valuations for that item are independently drawn. The univariate distributions must be finite and discrete, and are referred to as *valuation distributions*. We use $\hat{d}_{i\hat{t}}$ to denote the valuation distribution that type class $\hat{t}$ has for item i, for all $i \in \mathcal{I}$ and $\hat{t} \in \hat{\mathcal{T}}$. Thus we can think of a bidder with type class $\hat{t}$ as drawing their type from the distribution $\bigtimes_{i \in \mathcal{I}} \hat{d}_{i\hat{t}}$.

When we specify the input, we are allowed to define some valuation distributions to be used in the problem setting. We use r to denote the number of valuation distributions defined. For convenience, we denote $\mathcal{V} = \{1, 2, \ldots, r\}$, so that each valuation distribution is given the label "valuation distribution v", for some $v \in \mathcal{V}$. All valuation distributions must be defined over the

[1] Available at https://github.com/melyshu/roasolver/blob/master/roasolver.jar.

same support S. We use s to denote the number of values in S. For convenience, we denote $\mathcal{W} = \{1, 2, \ldots, s\}$, and use u_w to denote the w-th value in S, for all $w \in \mathcal{W}$. The valuation distributions are thus defined by ϕ_{vw}, for all $v \in \mathcal{V}$ and $w \in \mathcal{W}$, indicating the probability of drawing u_w from valuation distribution v. For the purpose of specifying input, it is not necessary to define any valuation distribution that always gives the same value with probability 1.

In terms of notation, we use the upper carat ($\hat{}$) to mark any variables that have been modified to use type classes instead of explicit types.

3.2.2 Bidder classes

For ease of input and computational efficiency, we think of each individual bidder as belonging to a *bidder class*. We use $\dot{n}$ to denote the number of bidder classes. For convenience, we denote $\dot{\mathcal{B}} = \{1, 2, \ldots, \dot{n}\}$, so that each bidder class is given the label "bidder class $\dot{b}$" for some $\dot{b} \in \dot{\mathcal{B}}$. We use $n_{\dot{b}}$ to refer to the number of bidders in bidder class $\dot{b}$, which we also call the *multiplicity* of that bidder class. Each complete specification $\mathbf{n} = (n_1, n_2, \ldots, n_{\dot{n}})$ indicating the multiplicities of each bidder class is called a *multiplicity profile*.

In addition, we associate with bidder class $\dot{b}$ the *type class distribution* denoted by $\hat{D}_{\dot{b}}$, defined by $\hat{p}_{\dot{b}\hat{t}} = \mathbb{P}_{\hat{t}' \sim \hat{D}_{\dot{b}}}[\hat{t}' = \hat{t}]$, for all $\dot{b} \in \dot{\mathcal{B}}$ and $\hat{t} \in \hat{\mathcal{T}}$, indicating the probability that each bidder in bidder class $\dot{b}$ draws type class $\hat{t}$. We group the bidders in classes like this so that in a given trial, bidder class $\dot{b}$ contains $n_{\dot{b}}$ indistinguishable bidders, each with the identical type class distribution $\hat{D}_{\dot{b}}$. Thus, we only need to specify one type class distribution per bidder class, rather than one type distribution per bidder.

Since the bidders within a bidder class are indistinguishable, we can use the result discussed in Section 2.5.4, which tells us that there exists an optimal auction that treats the bidders identically. That is, each bidder in the same bidder class is allocated items and charged prices in the same way. Consequently, the allocation rule and payment rule only need to be defined per bidder class, rather than per individual bidder. We use $\dot{\pi}_{\dot{b}i}(t)$ to denote the interim probabilities with which each bidder in bidder class $\dot{b}$ receives item i when reporting type t, and $\dot{q}_{\dot{b}t}$ to denote the price each bidder in bidder class $\dot{b}$ pays when reporting type t, for all $\dot{b} \in \dot{\mathcal{B}}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. We can reduce the number of variables in the linear program by solving for $\dot{\boldsymbol{\pi}}$ and $\dot{\mathbf{q}}$ instead.

In terms of notation, we use the dot $(\dot{\ })$ to mark any variables that have been modified to use bidder classes instead of individual bidders.

3.2.3 Problem settings and trials

RoaSolver takes the position of the auctioneer and computes the optimal auction from the values of the predefined constants as listed in Section 2.3.1. Since we are using type classes and bidder classes for the purpose of input, the predefined constants are derived from the variables $\dot{n}, m, \hat{c}, \{\hat{p}_{\dot{b}\hat{t}}\}_{\dot{b}\in\dot{\mathcal{B}},\hat{t}\in\hat{\mathcal{T}}}, \{n_{\dot{b}}\}_{\dot{b}\in\dot{\mathcal{B}}}$, which we refer to as the *trial arguments*. We use the term *trial* to refer to each unique specification of the trial arguments for which RoaSolver calculates the predefined constants and finds the optimal auction. The optimal parameters for that trial are referred to as a *solution*. Each input file specifies a *problem setting* that gives the values of the trial arguments for one or more trials to be run, and produces one or more solutions when read by RoaSolver. Currently, only the number of bidders in each bidder class can be varied between different trials under the same problem setting, as explained below.

To specify multiple trials under one problem setting, we associate each bidder class $\dot{b}$ with the set of its possible multiplicities, denoted by $N_{\dot{b}}$, containing all possible values of $n_{\dot{b}}$ for that bidder class. These sets are defined in the problem setting, and each multiplicity profile $(n_1, n_2, \ldots, n_{\dot{n}}) \in \bigtimes_{\dot{b}\in\dot{\mathcal{B}}} N_{\dot{b}}$ is given its own trial. All other inputs specified by a problem setting are held constant over all trials within that problem setting. Refer to Section 3.4 for a more detailed discussion on writing input for RoaSolver.

3.3 Installation of dependencies

RoaSolver requires Java to run and IBM's CPLEX library to solve linear programs. In this section, we discuss how to install both of these dependencies.

3.3.1 Java installation

RoaSolver is written in Java and therefore requires an installation of Java to run. If Java has not yet been installed, visit the Java website[2] to download the appropriate installer. In the case of a

[2]Available at `https://www.java.com/en/download/manual.jsp`.

Windows machine, select the Windows Offline (64-bit) option. Run the installer and follow the prompts to configure Java. The default installation should suffice.

Once Java has been installed, open a terminal and enter the following command:

```
java -version
```

This checks which version of Java has been installed. For reference, the output should look like:

```
java version "1.8.0_171"
Java(TM) SE Runtime Environment (build 1.8.0_171-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.171-b11, mixed mode)
```

Note that we require Version 8, indicated by the 1.8 in the first line, and a 64-bit installation, as specified in the last line.

3.3.2 CPLEX installation

RoaSolver relies on *IBM ILOG CPLEX Optimization Studio v12.9 (CJ4Z5ML)* to solve linear programs. This software is available for download from the IBM website[3] and is free for students and faculty. Click on "Get student and faculty editions for free" and follow the prompts to download the installer. The website requires registration with a `.edu` email address.

Once account registration has been completed, the software can be purchased for free after filling out a survey. Download and run the appropriate installer based on the operating system of the machine. Select the default installation configuration at each prompt, but note that *Microsoft Visual C++ 2015* is not required for RoaSolver, and that there is no need to start the IDE or open the Readme file at the end of the installation.

3.4 Input specification

In this section, we explain how to write an input file that specifies a problem setting for RoaSolver. This involves specifying the values of all input variables.

[3] Available at `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

3.4.1 Input format

The variables specifying a problem setting should be input in the following order, with spaces between values and newlines as indicated:

$$\begin{array}{lllll}
\dot{n} & m & \hat{c} & r & s \\
\\
\hat{p}_{11} & \hat{p}_{12} & \cdots & \hat{p}_{1\hat{c}} & N_1 \\
\hat{p}_{21} & \hat{p}_{22} & \cdots & \hat{p}_{2\hat{c}} & N_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\hat{p}_{\dot{n}1} & \hat{p}_{\dot{n}2} & \cdots & \hat{p}_{\dot{n}\hat{c}} & N_{\dot{n}} \\
\\
\hat{d}_{11} & \hat{d}_{12} & \cdots & \hat{d}_{1\hat{c}} \\
\hat{d}_{21} & \hat{d}_{22} & \cdots & \hat{d}_{2\hat{c}} \\
\vdots & \vdots & \ddots & \vdots \\
\hat{d}_{m1} & \hat{d}_{m2} & \cdots & \hat{d}_{m\hat{c}} \\
\\
\phi_{11} & \phi_{12} & \cdots & \phi_{1s} \\
\phi_{21} & \phi_{22} & \cdots & \phi_{2s} \\
\vdots & \vdots & \ddots & \vdots \\
\phi_{r1} & \phi_{r2} & \cdots & \phi_{rs} \\
u_1 & u_2 & \cdots & u_s
\end{array}$$

3.4.2 Input variables

Below is a complete list of the variables used to define a problem setting and their corresponding meanings. Recall that all variables except the $n_{\dot{b}}$ are held constant for all trials within the problem setting.

- $\dot{n}$ is the number of bidder classes. $\dot{n}$ must be an integer satisfying $\dot{n} \geq 1$.
- m is the number of items. m must be an integer satisfying $m \geq 1$.
- $\hat{c}$ is the number of type classes. $\hat{c}$ must be an integer satisfying $\hat{c} \geq 1$.

- r is the number of valuation distributions to be defined for use in the type classes. r must be an integer satisfying $r \geq 0$.

- s is the size of the support used to define the valuation distributions. s must be an integer satisfying $s \geq 0$.

- $\hat{p}_{\dot{b}\hat{t}}$ is the unnormalized probability with which a bidder in bidder class $\dot{b}$ draws type class $\hat{t}$, for all $\dot{b} \in \dot{\mathcal{B}}$ and $\hat{t} \in \hat{T}$. $\hat{p}_{\dot{b}\hat{t}}$ must be a float satisfying $\hat{p}_{\dot{b}\hat{t}} \geq 0$. Furthermore, the total unnormalized probability $\hat{p}_{\dot{b}} = \sum_{\hat{t} \in \hat{T}} \hat{p}_{\dot{b}\hat{t}}$ for bidder class $\dot{b}$ must satisfy $\hat{p}_{\dot{b}} > 0$, so that $\{\hat{p}_{\dot{b}\hat{t}} / \hat{p}_{\dot{b}} : \hat{t} \in \hat{T}\}$ are the normalized probabilities.

- $N_{\dot{b}}$ specifies the possible multiplicities for bidder class $\dot{b}$, for each $\dot{b} \in \dot{\mathcal{B}}$. $N_{\dot{b}}$ is a collection of integers, specified by a list of *integer ranges*, separated by semicolons (`;`). Each integer range either is a single nonnegative integer j to add that integer to the list, or has the form j~k (j immediately followed by a tilde, immediately followed by k) to add all integers between j and k, inclusive, to the list, where j, k are integers satisfying $0 \leq j \leq k$. As an example, `5;4~8;1` is a valid integer range list that specifies the integers $5, 4, 5, 6, 7, 8, 1$. Recall that a trial is run for each possible multiplicity profile $(n_1, n_2, \ldots, n_{\dot{n}}) \in \bigtimes_{\dot{b} \in \dot{\mathcal{B}}} N_{\dot{b}}$.

- $\hat{d}_{i\hat{t}}$ specifies the valuation distribution from which a bidder with type class $\hat{t}$ draws their value for item i, for all $i \in \mathcal{I}$ and $\hat{t} \in \hat{\mathcal{T}}$. Either $\hat{d}_{i\hat{t}}$ is a float u, in which case it corresponds to the distribution that gives u with probability 1, or $\hat{d}_{i\hat{t}}$ is of the form `D`v (that is, the character `D` immediately followed by v) where v is an integer satisfying $1 \leq v \leq r$. In the second case, this specifies that a bidder with type class $\hat{t}$ draws their value for item i independently from the valuation distribution v defined below.

- ϕ_{vw} is the unnormalized probability of drawing the value u_w from the valuation distribution v, for all $v \in \mathcal{V}$ and $w \in \mathcal{W}$. ϕ_{vw} must be a float satisfying $\phi_{vw} \geq 0$. Furthermore, the total unnormalized probability $\phi_v = \sum_{w \in \mathcal{W}} \phi_{vw}$ for valuation distribution v must satisfy $\phi_v > 0$, so that $\{\phi_{vw} / \phi_v : w \in \mathcal{W}\}$ are the normalized probabilities.

- u_w is the w-th value in S, the support of the valuation distributions, for all $w \in \mathcal{W}$. u_w must be a float.

Note that the input format is composed of three large matrices. It may be useful to note that each column of the first two matrices ($\hat{p}_{\hat{b}\hat{t}}$ and $\hat{d}_{\hat{i}\hat{t}}$) corresponds to a type class. Each row of the first matrix ($\hat{p}_{\hat{b}\hat{t}}$) corresponds to a bidder class, while each row of the second matrix ($\hat{d}_{\hat{i}\hat{t}}$) corresponds to an item. In the third matrix (ϕ_{vw}), each column corresponds to a value in the support, and each row corresponds to a valuation distribution.

3.4.3 Input file example

We provide an example of a correctly formatted input file. Below, we specify a problem setting with two trials. One trial has two bidders and the other trial has three bidders. In both trials, there are two items, and all the bidders are i.i.d. with each bidder valuing each item independently as either 3 or 4, with probabilities 0.4 and 0.6, respectively.

```
1 2 1 1 2

1 2;3

D1

D1

2 3

3 4
```

3.5 RoaSolver execution

In this section, we provide instructions on how to run RoaSolver on a given input file.

3.5.1 Command-line execution

Currently, RoaSolver can only be run through the command-line. Open up a terminal in the same directory as the `roasolver.jar` executable. If installation of Java and CPLEX has gone smoothly, then RoaSolver can be run by entering the following command:

```
java -jar roasolver.jar <flags/files>
```

Note that running RoaSolver on a Mac may require specifying the CPLEX installation directory by entering the following command instead:

```
java -jar roasolver.jar
    -Djava.library.path="/Applications/CPLEX_Studio129/cplex/bin/x86-64_osx"
    <flags/files>
```

The command-line arguments `<flags/files>` are a space separated list of tokens. If a token is one of the flags specified in Section 3.5.2 below, it enables the corresponding output method, otherwise it is treated as a file name to be input into the solver.

3.5.2 Output flags

Each one of the following flags enables an additional method of output:

- `-ft` specifies outputting the full descriptions of the solutions to the terminal.
- `-ff` specifies outputting the full descriptions of the solutions to individual CSV files.
- `-bt` specifies outputting brief descriptions of the solutions to the terminal.
- `-bf` specifies outputting brief descriptions of the solutions to individual CSV files.

Each problem setting only produces one brief description of the solutions for all trials under that problem setting, but each trial produces its own full description of its solution. All output files are named with the original file name as a prefix.

3.5.3 Execution example

Suppose that we have the example input in Section 3.4.3 contained in a file called `input.txt` in the same directory. Then entering the command

```
java -jar roasolver.jar input.txt -bt -ff
```

takes input from the file `input.txt`, displays a brief description of the solutions within the terminal, and writes the full descriptions of the solutions to the CSV files `input.txt_2_solution.csv` and `input.txt_3_solution.csv`, one for each trial.

3.6 Output specifications

In this section, we explain how to read the solutions output from RoaSolver. There are two possible output formats that can be specified by the flags used in the command-line, one that gives a brief description and one that gives a full description.

3.6.1 Full output format

Whenever a full description of a solution is output, the following variables are specified in the following format (separated by spaces in terminal, and by commas in a CSV file):

$$\begin{array}{lllll}
\dot{n} & m & c & & \\
\hat{t}_1 & \hat{t}_2 & \cdots & \hat{t}_c & \\
\dot{p}_{11} & \dot{p}_{12} & \cdots & \dot{p}_{1c} & n_1 \\
\dot{p}_{21} & \dot{p}_{22} & \cdots & \dot{p}_{2c} & n_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\dot{p}_{\dot{n}1} & \dot{p}_{\dot{n}2} & \cdots & \dot{p}_{\dot{n}c} & n_{\dot{n}} \\
v_{11} & v_{12} & \cdots & v_{1c} & \\
v_{21} & v_{22} & \cdots & v_{2c} & \\
\vdots & \vdots & \ddots & \vdots & \\
v_{m1} & v_{m2} & \cdots & v_{mc} &
\end{array}$$

$$J$$

$$\begin{array}{cccc}
\dot{\pi}_{11}(1) & \dot{\pi}_{11}(2) & \cdots & \dot{\pi}_{11}(c) \\
\dot{\pi}_{12}(1) & \dot{\pi}_{12}(2) & \cdots & \dot{\pi}_{12}(c) \\
\vdots & \vdots & \ddots & \vdots \\
\dot{\pi}_{1m}(1) & \dot{\pi}_{1m}(2) & \cdots & \dot{\pi}_{1m}(c) \\
\dot{q}_{11} & \dot{q}_{12} & \cdots & \dot{q}_{1c}
\end{array}$$

$$\begin{array}{cccc}
\dot{\pi}_{21}(1) & \dot{\pi}_{21}(2) & \cdots & \dot{\pi}_{21}(c) \\
\dot{\pi}_{22}(1) & \dot{\pi}_{22}(2) & \cdots & \dot{\pi}_{22}(c) \\
\vdots & \vdots & \ddots & \vdots \\
\dot{\pi}_{2m}(1) & \dot{\pi}_{2m}(2) & \cdots & \dot{\pi}_{2m}(c) \\
\dot{q}_{21} & \dot{q}_{22} & \cdots & \dot{q}_{2c}
\end{array}$$

$$\vdots$$

$$\begin{array}{cccc}
\dot{\pi}_{\dot{n}1}(1) & \dot{\pi}_{\dot{n}1}(2) & \cdots & \dot{\pi}_{\dot{n}1}(c) \\
\dot{\pi}_{\dot{n}2}(1) & \dot{\pi}_{\dot{n}2}(2) & \cdots & \dot{\pi}_{\dot{n}2}(c) \\
\vdots & \vdots & \ddots & \vdots \\
\dot{\pi}_{\dot{n}m}(1) & \dot{\pi}_{\dot{n}m}(2) & \cdots & \dot{\pi}_{\dot{n}m}(c) \\
\dot{q}_{\dot{n}1} & \dot{q}_{\dot{n}2} & \cdots & \dot{q}_{\dot{n}c}
\end{array}$$

$$\delta$$

3.6.2 Full output variables

Below is a complete list of the variables used in a full description of a solution and their corresponding meanings.

- $\dot{n}$ is the number of bidder classes. $\dot{n}$ is an integer.
- m is the number of items. m is an integer.
- c is the number of types. c is an integer.
- $\hat{t}_t$ is the type class that type t is derived from. $\hat{t}_t$ is an integer.
- $\dot{p}_{\dot{b}t}$ is the normalized probability with which a bidder in bidder class $\dot{b}$ draws type t, for all $\dot{b} \in \dot{\mathcal{B}}$ and $t \in \mathcal{T}$. $\dot{p}_{\dot{b}t}$ is a float.

- $n_{\dot{b}}$ is the number of bidders in bidder class $\dot{b}$, for all $\dot{b} \in \dot{\mathcal{B}}$. $n_{\dot{b}}$ is an integer.
- v_{it} is the value of item i to a bidder with type t, for all $i \in \mathcal{I}$ and $t \in \mathcal{T}$. v_{it} is a float.
- J is the expected revenue of the auction. J is a float.
- $\dot{\pi}_{\dot{b}i}(t)$ is the interim probability that each bidder in bidder class $\dot{b}$ gets item i when bidding type t, for all $\dot{b} \in \dot{\mathcal{B}}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. That is, for each bidder in bidder class $\dot{b}$, $\dot{\pi}_{\dot{b}i}(t)$ is the probability that they get item i, under the condition that they report type t, and in expectation over all other bidders bidding truthfully. $\dot{\pi}_{\dot{b}i}(t)$ is a float.
- $\dot{q}_{\dot{b}t}$ is the price that each bidder in bidder class $\dot{b}$ pays when reporting type t, for all $\dot{b} \in \dot{\mathcal{B}}$ and $t \in \mathcal{T}$. $\dot{q}_{\dot{b}t}$ is a float.
- δ is the time in seconds that RoaSolver takes to run this trial. δ is a float.

For readability, there are multiple newlines separating matrices, and in the case of terminal output, floats are truncated to 4 decimal places.

3.6.3 Full output example

We provide an example of the full descriptions of the solutions that are produced by running RoaSolver on the example input in Section 3.4.3 with a `-ft` flag. Note that there are two trials specified. The output from the first trial is:

```
1 2 4
      1        1        1        1
  0.1600   0.2400   0.2400   0.3600 2
  3.0000   3.0000   4.0000   4.0000
  3.0000   4.0000   3.0000   4.0000



  7.3440


  0.0800   0.2800   0.8800   0.5800
  0.0800   0.8800   0.2800   0.5800
  0.4800   4.2800   4.2800   4.2800


  0.0092
```

The output from the second trial is:

```
1 2 4
       1        1        1        1
  0.1600   0.2400   0.2400   0.3600 3
  3.0000   3.0000   4.0000   4.0000
  3.0000   4.0000   3.0000   4.0000



  7.7606


  0.0085   0.0832   0.7792   0.3472
  0.0085   0.7792   0.0832   0.3472
  0.0512   3.3579   3.3579   2.6859


  0.0064
```

3.6.4 Brief output format

Whenever a brief description of the solutions is output, the following variables are specified in the following format (separated by spaces in terminal, and by commas in a CSV file):

$$\begin{array}{cccccc} n_{11} & n_{12} & \cdots & n_{1\dot{n}} & J_1 & \delta_1 \\ n_{21} & n_{22} & \cdots & n_{2\dot{n}} & J_2 & \delta_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ n_{T1} & n_{T2} & \cdots & n_{T\dot{n}} & J_T & \delta_T \end{array}$$

3.6.5 Brief output variables

Below is a complete list of the variables used in a brief description of the solutions and their corresponding meanings.

- T is the number of trials run. T is an integer.
- $n_{\tau\dot{b}}$ is the value of $n_{\dot{b}}$ used in trial τ, for all $\dot{b} \in \dot{\mathcal{B}}$ and $1 \leq \tau \leq T$. $n_{\tau\dot{b}}$ is an integer.
- J_τ is the expected revenue of the auction in trial τ, for all $1 \leq \tau \leq T$. J_τ is a float.
- δ_τ is the time in seconds that RoaSolver takes to run trial τ, for all $1 \leq \tau \leq T$. δ_τ is a float.

Note that this output has headings, and in the case of terminal output, floats are truncated to 4 decimal places.

3.6.6 Brief output example

We provide an example of the brief description of the solutions that is produced by running RoaSolver on the example input in Section 3.4.3 with a `-bt` flag.

```
n_1      rev     time
  2   7.3440   0.0998
  3   7.7606   0.0075
```

Chapter 4

Software Documentation

In this chapter, we present a software development guide for RoaSolver, intended to provide documentation and act as a reference for any developer who wishes to work closely with the source code. We assume that the developer is already familiar with the background theory and the execution of the software as discussed in Chapters 2 and 3. We describe the setup of the development environment and discuss the style and structure of the code in detail.

4.1 Development environment

In this section, we describe one way of setting up a development environment for RoaSolver.

4.1.1 System specifications

To provide a point of reference, RoaSolver was developed on a Dell XPS 15 9550 running Microsoft Windows 10 Education Version 10.0.17763 Build 17763. The code was written in Java with the Java Development Kit (JDK)[1] Version 8u121, using Visual Studio Code[2] Version 1.33.1 for code editing, Apache Maven[3] Version 3.6.0 for project management, and Git[4] Version 2.8.1 for version control. Developing RoaSolver on other systems or setups may be a different experience to what is described in the rest of this chapter.

[1] Available at `https://www.oracle.com/technetwork/java/javase/downloads/index.html`.
[2] Available at `https://code.visualstudio.com/download`.
[3] Available at `https://maven.apache.org/download.cgi`.
[4] Available at `https://git-scm.com/downloads`.

4.1.2 Repository setup

The full repository containing the source code is available on GitHub[5]. Clone the repository onto a local machine. One way to do this is to open up a terminal in the directory that is to contain the repository, and enter the command:

```
git clone https://github.com/melyshu/roasolver.git
```

The repository should be cloned into a directory named `roasolver`. For convenience, we use `roasolver` to refer to this root directory.

The repository needs to be set up to use the CPLEX library. If CPLEX is not yet installed, refer to the instructions in Section 3.3.2. After CPLEX has been installed, Maven must be configured to have it registered as a dependency. This can be done by opening up a terminal in `roasolver` and entering the command:

```
mvn install:install-file -DgroupId=cplex -DartifactId=cplex -Dversion=12.9
    -Dpackaging=jar -Dfile="C:\Program
    Files\IBM\ILOG\CPLEX_Studio129\cplex\lib\cplex.jar"
```

On a Mac, the `-Dfile` flag should be `/Applications/CPLEX_Studio129/cplex/lib/cplex.jar` instead. This configuration only needs to be completed once.

4.1.3 Repository structure

As a brief guide to navigating the repository, we describe the function of a few important files.

- `roasolver\src\main\java\edu\princeton\cs\roasolver\RoaSolver.java` is the Java file that contains the source code for RoaSolver.
- `roasolver\testing` contains the test inputs as discussed in Section 4.5.
- `roasolver\pom.xml` is the configuration file for Maven, specifying dependencies and build instructions.
- `roasolver\readme` provides minimal instructions on how to set up and build the code.

[5] Available at `https://github.com/melyshu/roasolver`.

4.1.4 Debugging procedure

Whilst writing RoaSolver, debugging was primarily done through the debugging tools provided in the editor, Visual Studio Code. To compile `RoaSolver.java` manually, open up the terminal in a directory that is to contain the compiled files, and enter the following command:

```
javac -sourcepath <sourcepath> RoaSolver.java -d .
```

The command-line argument `<sourcepath>` is the path to the `roasolver\src\main\java` directory of the repository. To run RoaSolver following the compilation, simply enter the following command in the same directory:

```
java edu.princeton.cs.roasolver.RoaSolver <files/flags>
```

Input is specified as described in Section 3.5.1.

4.1.5 Build process

The build process is automated by Maven, and can be executed by opening up a terminal in `roasolver` and entering the command:

```
mvn clean compile assembly:single
```

This produces build files in the directory `roasolver\target`, which should contain the executable JAR `roasolver.jar`.

4.2 Code overview

In this section, we discuss the overall structure and purpose of the code within `RoaSolver.java`.

4.2.1 RoaSolver class

The `RoaSolver` class is the primary class in `RoaSolver.java`. Each `RoaSolver` object represents a trial of a problem setting. The input variables specifying the arguments for the trial are encapsulated by the `RoaSolverInput` inner class, and the output variables specifying the solutions are encapsulated by the `RoaSolverOutput` inner class. Each trial is solved by passing a `RoaSolverInput` object into the `RoaSolver` constructor, and retrieving the outputs by calling the `getOutput()` method.

The entry point for the final executable JAR is the `main()` method in the `RoaSolver` class, which locates the files and parses them with the help of the `RoaSolverParser` inner class to obtain the arguments for the trials. Each trial is then solved by creating a corresponding `RoaSolver` object. The solution is written to the specified locations with the help of the methods in the `RoaSolverOutput` class.

Internally, the `RoaSolver` constructor sets up a CPLEX model in the `cplex` instance variable, then constructs and solves the linear program using the `solve()` helper method, which specifies the CPLEX model through the `addIRConstraints()`, `addBICConstraints()`, `addObjective()`, and `addSO()` helper methods.

4.2.2 RoaSolverParser class

The `RoaSolverParser` inner class is responsible for parsing the input files. Each `RoaSolverParser` object represents the information specifying a problem setting, obtained from reading an input file. The input file must be passed into the constructor as a `Scanner` object. Internally, the `RoaSolverParser` constructor relies on a collection of helper methods to read, validate, and normalize the input.

4.2.3 RoaSolverInput class

The `RoaSolverInput` inner class encapsulates the input arguments creating a `RoaSolver` object. Each `RoaSolverInput` object represents the arguments specifying one trial within a problem setting. The static method `getInputs()` expands type classes into explicit types and calculates multiplicity profiles for all trials given a `RoaSolverParser` object. Internally, this is done by repeatedly building up incomplete types and incomplete multiplicity profiles until all possible permutations are covered. This process is covered in more detail in Sections 4.4.1 and 4.4.2.

4.2.4 RoaSolverOutput class

The `RoaSolverOutput` inner class encapsulates the output containing the solution produced by a `RoaSolver` object. Each `RoaSolverOutput` object represents both the collection of expanded input arguments as well as the parameters of the optimal auction, for a single trial. The methods within this class provide ways to format the solution according to the output formats as specified in Section 3.6.

4.2.5 SeparationOracle class

The `SeparationOracle` inner class executes the role of the separation oracle within the CPLEX model. The `main()` method performs the work of the separation oracle, adding any broken constraint it finds to the CPLEX model. Internally, the static constant `EPSILON` specifies the margin of error within which constraints should be satisfied, and is used primarily to avoid cases where precision-related issues cause a constraint to become broken. The two classes `PiNode` and `XNode` contained within the `SeparationOracle` class are used to encapsulate data for sorting purposes.

4.2.6 RoaSolverError class

The `RoaSolverError` inner class encapsulates the errors internal to the `RoaSolver` class. Most foreseeable exceptions are thrown as a `RoaSolverError` object so that they can be caught and dealt with conveniently.

4.3 Notational conventions

In this section, we discuss some of the notational conventions adopted in `RoaSolver.java` for the sake of increasing code readability and making typos easier to find.

4.3.1 Variable naming

Generally, we try to keep the notation consistent between the names of the variables used in mathematical specifications in this thesis and the names of the variables used within the code. Thus, for example, the number of items is denoted by m in mathematical contexts, and `m` in code. However, there are some notable exceptions:

- `b` is the number of bidder classes, denoted elsewhere by $\dot{n}$.
- `t` is the number of type classes, denoted elsewhere by $\hat{c}$.

Variables with a corresponding mathematical symbol are denoted with snake case (for example, `q_sol_bcm`), while other variables are named with camel case (for example, `startTime`).

4.3.2 Array notation

For vectors and other multidimensional arrays, we adopt the practice of appending the length of the array in each dimension to the end of the variable name, separated by an underscore. For example, `n_b` is the name of the integer array that contains the multiplicity of each bidder class. Recall that we use $n_1, n_2, \ldots, n_{\dot{n}}$ to denote the multiplicities in a mathematical context. Since this corresponds to a one-dimensional array with $\dot{n}$ entries in that dimension, we append the corresponding symbol, `b`, to the end of the variable name.

4.3.3 Index notation

When we wish to access an array at a certain index or set of indices, we name the indexing variable based on the size of the dimension it is indexing. For example, in the two-dimensional double array `p_bc`, we usually use `bi` and `ci` as the indexing variables for each dimension, so that `p_bc[bi][ci]` is the value at those indices.

As a result, we frequently use the following idiom to iterate over an array:

```
for (int bi = 0; bi < b; bi++) {
    for (int ci = 0; ci < c; ci++) {

        // do something with each p_bc[bi][ci]
    }
}
```

If the size of a dimension is not fixed or is unknown, we use the variable `y` to denote this size, and the variable `yi` to denote the index, except in the particular case of bidder class multiplicities. We use `z` to denote the number of possible multiplicities for a given bidder class, and `zi` to denote the index.

4.4 Code walkthrough

In this section, we provide a more detailed description of selected parts of `RoaSolver.java`.

4.4.1 Type class expansion

The first half of the static method `getInputs()` in the `RoaSolverInput` inner class expands all type classes into explicit types. This means, for each type class $\hat{t}$, calculating the support of the distribution $\bigtimes_{i \in \mathcal{I}} \hat{d}_{i\hat{t}}$, along with the associated probabilities, so that we end up with the values c, $\{\hat{t}_t\}_{t \in \mathcal{T}}$, $\{\dot{p}_{\dot{b}t}\}_{\dot{b} \in \dot{\mathcal{B}}, t \in \mathcal{T}}$, which appear at the top of the full output specified in Section 3.6.1.

To begin with, we use the variables `t_temp_c`, `p_temp_cb`, and `v_temp_cm` to accumulate, for each explicit type we have completely calculated so far, its associated parent type class, probabilities, and valuations, respectively. We expand each type class in turn. The variables `v_curr_ym` and `p_curr_y` are used to accumulate the *incomplete types* and probabilities we have calculated so far as we iterate through the items. That is, at the beginning of iteration `mi`, `v_curr_ym` and `p_curr_y` contain all possible combinations of valuations of the first `mi` items, along with the corresponding probabilities. During that iteration, a copy of each incomplete type is made for each possible valuation for item `mi`, and updated to take into account this additional valuation and associated probability. The updated incomplete types and probabilities are held in `v_next_ym` and `p_next_y` until the end of the iteration, when they replace `v_curr_ym` and `p_curr_y` for the next iteration. We continue in this manner until we have iterated through all items, thereby completely calculating all the types within the type class. The final probabilities for all the types in the type class are then scaled by the probabilities given by the type class distribution. At this point, all information about the newly expanded types is saved into `t_temp_c`, `p_temp_cb`, and `v_temp_cm`, and the next type class is expanded. Once all type classes have been expanded, we convert these into the arrays `t_c`, `p_bc`, and `v_cm`.

4.4.2 Multiplicity profile expansion

The second half of the static method `getInputs()` in the `RoaSolverInput` class deals with the expansion of all multiplicity possibilities into multiplicity profiles. That is, we calculate all the members of the set $\bigtimes_{\dot{b} \in \dot{\mathcal{B}}} N_{\dot{b}}$ so that each one may be given a trial.

We use the variable `n_curr_yb` to accumulate the *incomplete multiplicity profiles* we have calculated so far as we iterate through the bidder classes. That is, at the beginning of iteration `bi`, `n_curr_yb` contains all possible combinations of multiplicities of the first `bi` bidder classes. During that iteration, a copy of each incomplete multiplicity profile is made for each possible multiplicity for bidder class `bi`, and updated to take into account this additional multiplicity. The updated incomplete multiplicity profiles are held in `n_next_yb` until the end of the iteration, when they replace `n_curr_yb` for the next iteration. We repeat this process until we have iterated through all bidder classes, and calculated all multiplicity profiles. Finally, for each multiplicity profile, we create a `RoaSolverInput` object specifying the arguments for a trial.

4.4.3 Separation oracle

The `main()` method in the `SeparationOracle` class performs the role of the separation oracle within the CPLEX model, checking the Border constraints to ensure that the interim allocation rule is feasible. Since we use bidder classes, the separation oracle runs slightly differently to the description provided in Section 2.4.2, so we provide an updated specification first.

Given $\dot{n}, m, c, \{\dot{p}_{\dot{b}t}\}_{\dot{b}\in\dot{\mathcal{B}},t\in\mathcal{T}}, \{n_{\dot{b}}\}_{\dot{b}\in\dot{\mathcal{B}}}, \{\dot{\pi}_{\dot{b}i}(t)\}_{\dot{B}\in\dot{\mathcal{B}},i\in\mathcal{I},t\in\mathcal{T}}$, we first define the *x-values* to be

$$\dot{x}_{\dot{b}i}(t) = \dot{\pi}_{\dot{b}i}(t) \sum_{\substack{t'\in\mathcal{T} \\ \dot{\pi}_{\dot{b}i}(t)\geq\dot{\pi}_{\dot{b}i}(t')}} \dot{p}_{\dot{b}t'}, \tag{4.1}$$

for all $\dot{b} \in \dot{\mathcal{B}}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. We define the sets

$$\dot{S}_{\dot{b}i}(x) = \{t \in \mathcal{T} : \dot{x}_{\dot{b}i}(t) \geq x\}, \tag{4.2}$$

for all $\dot{b} \in \dot{\mathcal{B}}$ and $i \in \mathcal{I}$, as a function of $x \in \mathbb{R}$. We also define the sets

$$\dot{X}_i = \left\{\dot{x}_{\dot{b}i}(t) : \dot{b} \in \dot{\mathcal{B}}, t \in \mathcal{T}\right\}, \tag{4.3}$$

for all $i \in \mathcal{I}$, containing for the threshold values of x at which at least one of the sets in $\{\dot{S}_{\dot{b}i}(x)\}_{\dot{b}\in\dot{\mathcal{B}}}$ changes. The interim allocation rule $\dot{\pi}$ is feasible if and only if for all $i \in \mathcal{I}$ and $x \in \dot{X}_i$ it is true that

$$\sum_{\dot{b}\in\mathcal{B}}\sum_{t\in\dot{S}_{bi}(x)} n_{\dot{b}}\dot{\pi}_{\dot{b}i}(t)\dot{p}_{\dot{b}t} \leq 1-\prod_{\dot{b}\in\dot{\mathcal{B}}}\left(1-\sum_{t\in\dot{S}_{\dot{b}i}(x)}\dot{p}_{\dot{b}t}\right)^{n_{\dot{b}}}. \tag{4.4}$$

We use $\dot{\mathfrak{S}}$ to denote the separation oracle that takes $\dot{\pi}$ as input, checks the Border constraints above to determine the feasibility of the interim allocation rule, and outputs either a hyperplane ℓ corresponding to a broken constraint, or the signal YES to indicate that all constraints are satisfied.

Within the `main()` method of the `SeparationOracle` class, we implement the separation oracle, checking constraints by iterating through each item independently in turn. For item `mi`, we first compute the corresponding x-values. We do this by then iterating through the bidder classes. For bidder class `bi`, we create a `PiNode` object for each type `ci`, and use the `PiNode` objects to sort the types in ascending order based on their value of `pi_mbc[mi][bi][ci]`. We then calculate the x-values corresponding to each type for this bidder class by iterating through the `PiNode` objects in this order, keeping track of the variables `pi_curr` and `p_sum`, which represent the highest value of `pi_mbc` seen so far and the running total of the summation on the right-hand side of Equation 4.1, respectively. This allows us to efficiently calculate the x-values, which we store in `XNode` objects. The `XNode` captures the x-value, as well as the value of `bi` and `ci` corresponding to that x-value. All `XNode` objects for item `mi` are collected in `xn_y`, corresponding to the set defined in Equation 4.3.

We then check the Border constraints in Equation 4.4 for item `mi` by iterating through the `XNode` objects in `xn_y` in descending order based on their x-value. This is equivalent to constructing the sets defined in Equation 4.2 at each threshold value by incrementally making them bigger, and incrementally updating our running values of both sides of the inequality so that we do not need to calculate the entire expression each time. We use the variables `xn_curr_y` to keep track of the `XNode` objects seen so far and `x_curr` to keep track of the lowest x-value seen so far, corresponding to the particular x-value with which we are about to test Equation 4.4. We use the variables `lhs_sum` to keep track of the running sum of the left-hand side of Equation 4.4 at this current x-value, `ln_rhs_prod` to keep track of the natural log of the running value of the product on the right-hand side of Equation 4.4, and `rhs_b` to keep track of the running values of the individual multiplicands within the product on the right-hand side of Equation 4.4. At each new `x_curr` value, we update all of these variables to reflect the new state of the inequality after iterating through all `XNode` objects with that x-value, equivalent to adding the corresponding types to the corresponding sets defined in Equation 4.2. At this point, we check the constraint. If we find a broken constraint at any point in time, we use the `XNode` objects in `xn_curr_y` to figure out the variables to use to construct the broken constraint within the CPLEX model, and immediately stop checking additional constraints.

4.4.4 Linear program

The `solve()` helper method in the `RoaSolver` class constructs the linear program as a CPLEX model, setting up all variables and constraints as well as specifying the separation oracle and objective function. Since we use bidder classes, the linear program is slightly different to the one specified in Section 2.4.3, so we rewrite the full specification of the updated linear program below.

Given the derived inputs to the trial $\dot{n}, m, c, \{n_{\dot{b}}\}_{\dot{b}\in\dot{\mathcal{B}}}, \{\dot{p}_{\dot{b}t}\}_{\dot{b}\in\dot{\mathcal{B}},t\in\mathcal{T}}, \{v_{ti}\}_{t\in\mathcal{T},i\in\mathcal{I}}$, we define the following variables to be used in the linear program:

- $\dot{\pi}_{\dot{b}i}(t) \in \mathbb{R}$, for all $\dot{b} \in \dot{\mathcal{B}}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. These variables are the interim probabilities specifying the interim allocation rule $\dot{\boldsymbol{\pi}}$, and are declared in the body of `solve()`. ($m\dot{n}c$ variables)

- $\dot{q}_{\dot{b}t} \in \mathbb{R}$, for all $\dot{b} \in \dot{\mathcal{B}}$ and $t \in \mathcal{T}$. These variables are the prices specifying the payment rule $\dot{\mathbf{q}}$, and are declared in the body of `solve()`. ($\dot{n}c$ variables)

We impose the following constraints in the linear program:

- $0 \leq \dot{\pi}_{\dot{b}i}(t) \leq 1$, for all $\dot{b} \in \dot{\mathcal{B}}$, $i \in \mathcal{I}$, and $t \in \mathcal{T}$. These constraints ensure that the $\dot{\pi}_{\dot{b}i}(t)$ are all probabilities, and are specified by the variable declarations in the body of `solve()`. ($2m\dot{n}c$ constraints)

- $\sum_{i\in\mathcal{I}} \pi_{\dot{b}i}(t)v_{it} - q_{\dot{b}t} \geq 0$, for all $\dot{b} \in \dot{\mathcal{B}}$ and $t \in \mathcal{T}$. These constraints ensure that the auction is IR, and are specified in the helper method `addIRConstraints()`. ($\dot{n}c$ constraints)

- $\sum_{i\in\mathcal{I}} \pi_{\dot{b}i}(t)v_{it} - q_{\dot{b}t} \geq \sum_{i\in\mathcal{I}} \pi_{\dot{b}i}(t')v_{it} - q_{\dot{b}t'}$, for all $\dot{b} \in \dot{\mathcal{B}}$ and $t, t' \in \mathcal{T}$. These constraints ensure that the auction is BIC, and are specified in the helper method `addBICConstraints()`. ($\dot{n}c^2$ constraints)

We add the following separation oracle:

- $\dot{\mathfrak{S}}$, so that if $\dot{\boldsymbol{\pi}}$ satisfies all other constraints and $\dot{\mathfrak{S}}(\dot{\boldsymbol{\pi}}) = \ell$, we add the broken constraint ℓ to the linear problem and iterate. The separation oracle ensures that there is an ex-post allocation rule inducing $\dot{\boldsymbol{\pi}}$, and is specified in the helper method `addSO()`.

Finally, we nominate the objective function to be maximized:

- $\sum_{t\in\mathcal{T}} \sum_{\dot{b}\in\dot{\mathcal{B}}} n_{\dot{b}} p_{\dot{b}t} q_{\dot{b}t}$. This is the expected revenue, and is specified in the helper method `addObjective()`.

4.5 Testing

In this section, we discuss some of the ways we have performed tests on RoaSolver.

4.5.1 Correctness

In order to ensure that RoaSolver produces the correct output, we have run the program on some simple inputs, specified by the files contained in `roasolver\testing\correctness`. These inputs specify problem settings small enough that we can solve them by hand using existing ideas from auction theory. Where possible, we have checked that all output variables are numerically correct, but at a minimum, we have at least verified that all expected revenues are numerically correct.

4.5.2 Validation

In order to ensure that input validation is working as intended, we have run the program on some erroneous inputs, specified by the files contained in `roasolver\testing\validation`. Each of these files is an invalid input for a different reason, and corresponds to a different exception that can be thrown in the `RoaSolverParser` class. We have checked that all errors are caught and trigger a helpful error message.

4.5.3 Stress

In order to ensure that RoaSolver can handle larger problems, we have run the problem on some larger inputs, specified by the files contained in `roasolver\testing\stress`. Each of these files specifies a problem setting with 100 trials, with each trial taking a few minutes to solve. We have successfully run RoaSolver on all problem settings to completion, taking many hours for each file.

4.5.4 Timing

In addition to the input files described above, we provide a way of measuring the performance of RoaSolver with the help of the Java program `RoaSolverTiming`, the source code for which can be found in the file `roasolver\testing\RoaSolverTiming.java`. This program generates random inputs and aggregates timing data when RoaSolver is run on these inputs. We have run `RoaSolverTiming` using the batch scripts provided in `roasolver\testing` to gain some insight into how long RoaSolver takes to solve inputs of varying sizes. The timing data is contained in corresponding CSV files within `roasolver\testing`, and is also reproduced in Appendix A.

In order to run `RoaSolverTiming`, we must first compile it by opening up a terminal in the `roasolver\testing` directory and entering the command:

```
javac -cp ..\..\target\roasolver.jar RoaSolverTiming.java
```

Then we may execute it by entering the command:

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming <b> <m> <c> <T>
   <filename>
```

The input arguments must be specified as follows:

- `<b>` specifies the possible values for the number of bidder classes.
- `<m>` specifies the possible values for the number of items.
- `<c>` specifies the possible values for the number of types.
- `<T>` specifies the number of trials for each unique combination of (`b`, `m`, `c`).
- `<filename>` specifies the name of the file to which the output is to be written.

The command-line arguments `<b>`, `<m>`, and `<c>` are specified as a list of integer ranges, in the same syntax used to define N_b for input into RoaSolver, as detailed in Section 3.4.2.

4.6 Future work

In this section, we detail some of the possible directions of future development.

4.6.1 Correctness

On certain inputs with extreme values, as may be found in `roasolver\testing\error`, where we specify a bidder class with 10,000,000 bidders, we have found that RoaSolver may output nonsensical figures. Since we have only noticed this phenomenon in rare cases where large numbers are specified, investigating this issue has not been a high priority. Consequently, while we think this is due to the limited amount of computational precision available, the exact cause and extent of this problem is unknown. We have assumed for the time being that a typical user who does not need to specify such extreme inputs will not encounter this issue. Ideally, the program should not produce outputs that are incorrect, or it should at least provide a warning that the outputs may not be accurate.

4.6.2 Performance

The timing data in Appendix A only gives a brief overview of the performance of RoaSolver. It may be worth the trouble to figure out a more precise relationship between the size of the input and the time required to find the solution, so that it becomes clear how to reduce run time further. At a glance, it seems that for trials of the same size, the run time has a lot of variance, and is heavily dependent on the number of times the separation oracle happens to run.

4.6.3 Code refactoring

Currently, the entirety of the source code for RoaSolver lies in the one file `RoaSolver.java`, containing about 1500 lines of code. Navigating and maintaining such a long file can become difficult, and is in general considered to be bad practice. To make future development more manageable, the code should be refactored into smaller files that are self-contained and easier to edit.

4.6.4 Item symmetry

It is possible to reduce the number of variables in the linear program by taking advantage of item symmetry, as discussed in Section 2.5.5. Since the CPLEX solver takes up the majority of the run time, and we are only able to control the size of the input going into the CPLEX model, taking advantage of item symmetry has the greatest potential to increase the performance of RoaSolver. However, detecting item symmetry is not a simple problem and as a result we have so far avoided implementing this optimization.

4.6.5 Input and output

It may be worthwhile to rethink how the program deals with input and output, since there is most certainly room for improving the user experience. Currently, input and output files have different formats, and it is not possible to specify the names of the output files. Furthermore, RoaSolver does not check for existing files, and so if it attempts to write to a file that already exists, it either silently overwrites it, or produces an error indicating that the file was unable to be written, both of which are undesirable outcomes.

Bibliography

[CDW11] Yang Cai, Constantinos Daskalakis, and S. Matthew Weinberg. "A Constructive Approach to Reduced-Form Auctions with Applications to Multi-Item Mechanism Design". In: *CoRR* abs/1112.4572 (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*. 2012. URL: http://doi.acm.org/10.1145/2229012.2229042.

Appendix A

Timing Data

In this appendix, we provide tables of timing data indicating how long RoaSolver takes to solve randomly generated inputs of varying sizes, prouced from running `RoaSolverTiming` via the batch script `roasolver\testing\timing\timing.bat` in the code repository. We show the command that could be used to run the test for each table. Each row in each table details the information for a set of trials that are solved by RoaSolver, but they may not be representative of all inputs of that size. As usual, $\dot{n}$ denotes the number of bidder classes, m denotes the number of items, c denotes the number of types, and T denotes the number of trials of that size that were run. For all tables below, we ran 10 trials of each size. The columns "#var." and "#con." show the number of variables and the number of constraints within the linear program for all trials of that size. The columns "#so" and "time(s)" show the number of times the separation oracle is called and the time in seconds taken to solve the linear program, averaged over all trials of that size.

A.1 Varying the number of bidder classes

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming
   1;2;4;8;16;32;64;128;256;512;1024;2048;4096;8192;16384;32768;65536 1 1 10
   timing_01.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
1	1	1	10	2	4	1.0	0.016
2	1	1	10	4	8	2.0	0.005
4	1	1	10	8	16	2.0	0.004
8	1	1	10	16	32	2.0	0.005
16	1	1	10	32	64	2.0	0.006
32	1	1	10	64	128	2.0	0.007
64	1	1	10	128	256	2.0	0.007
128	1	1	10	256	512	2.0	0.007
256	1	1	10	512	1,024	2.0	0.011
512	1	1	10	1,024	2,048	2.0	0.018
1,024	1	1	10	2,048	4,096	2.0	0.036
2,048	1	1	10	4,096	8,192	2.0	0.088
4,096	1	1	10	8,192	16,384	2.0	0.223
8,192	1	1	10	16,384	32,768	2.0	0.722
16,384	1	1	10	32,768	65,536	2.0	2.618
32,768	1	1	10	65,536	131,072	2.0	11.906
65,536	1	1	10	131,072	262,144	2.0	103.581

TABLE A.1: `timing_01.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming
   1;2;4;8;16;32;64;128;256;512;1024;2048;4096;8192 3 3 10 timing_02.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
1	3	3	10	12	30	1.0	0.014
2	3	3	10	24	60	9.2	0.008
4	3	3	10	48	120	37.4	0.013
8	3	3	10	96	240	107.6	0.042
16	3	3	10	192	480	959.1	4.900
32	3	3	10	384	960	2,902.2	82.833
64	3	3	10	768	1,920	1,970.6	30.129
128	3	3	10	1,536	3,840	78.4	0.611
256	3	3	10	3,072	7,680	1.6	0.073
512	3	3	10	6,144	15,360	2.1	0.264
1,024	3	3	10	12,288	30,720	305.7	130.891
2,048	3	3	10	24,576	61,440	1.1	1.853
4,096	3	3	10	49,152	122,880	2.0	13.016
8,192	3	3	10	98,304	245,760	1.6	42.394

TABLE A.2: `timing_02.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 1;2;3;4;5 10 10 10
   timing_03.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
1	10	10	10	110	310	1.0	0.019
2	10	10	10	220	620	651.1	0.786
3	10	10	10	330	930	1,393.4	2.649
4	10	10	10	440	1,240	3,095.0	23.149
5	10	10	10	550	1,550	8,730.3	174.592

TABLE A.3: `timing_03.csv`

A.2 Varying the number of items

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 1
    1;2;4;8;16;32;64;128;256;512;1024;2048;4096;8192;16384;32768;65536;131072
    1 10 timing_04.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
1	1	1	10	2	4	1.0	0.014
1	2	1	10	3	6	1.0	0.004
1	4	1	10	5	10	1.0	0.004
1	8	1	10	9	18	1.0	0.004
1	16	1	10	17	34	1.0	0.003
1	32	1	10	33	66	1.0	0.006
1	64	1	10	65	130	1.0	0.006
1	128	1	10	129	258	1.0	0.005
1	256	1	10	257	514	1.0	0.007
1	512	1	10	513	1,026	1.0	0.007
1	1,024	1	10	1,025	2,050	1.0	0.011
1	2,048	1	10	2,049	4,098	1.0	0.023
1	4,096	1	10	4,097	8,194	1.0	0.058
1	8,192	1	10	8,193	16,386	1.0	0.195
1	16,384	1	10	16,385	32,770	1.0	0.668
1	32,768	1	10	32,769	65,538	1.0	2.499
1	65,536	1	10	65,537	131,074	1.0	9.708
1	131,072	1	10	131,073	262,146	1.0	101.661

TABLE A.4: `timing_04.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 3
   1;2;4;8;16;32;64;128;256;512;1024 3 10 timing_05.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
3	1	3	10	18	54	5.6	0.019
3	2	3	10	27	72	23.5	0.013
3	4	3	10	45	108	52.9	0.015
3	8	3	10	81	180	58.2	0.027
3	16	3	10	153	324	160.1	0.151
3	32	3	10	297	612	225.8	0.340
3	64	3	10	585	1,188	21.8	0.034
3	128	3	10	1,161	2,340	64.7	0.267
3	256	3	10	2,313	4,644	2,251.3	94.891
3	512	3	10	4,617	9,252	3,748.1	493.957
3	1,024	3	10	9,225	18,468	60.4	12.563

TABLE A.5: `timing_05.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 10 1;2;3 10 10
    timing_06.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
10	1	10	10	200	1,300	1,154.6	2.281
10	2	10	10	300	1,500	5,246.1	55.669
10	3	10	10	400	1,700	8,645.0	272.091

TABLE A.6: `timing_06.csv`

A.3 Varying the number of types

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 1 1
   1;2;4;8;16;32;64;128;256;512 10 timing_07.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
1	1	1	10	2	4	1.0	0.012
1	1	2	10	4	10	1.0	0.005
1	1	4	10	8	28	1.0	0.006
1	1	8	10	16	88	1.0	0.007
1	1	16	10	32	304	1.0	0.007
1	1	32	10	64	1,120	1.0	0.024
1	1	64	10	128	4,288	1.0	0.160
1	1	128	10	256	16,768	1.0	1.870
1	1	256	10	512	66,304	1.0	14.033
1	1	512	10	1,024	263,680	1.0	160.950

TABLE A.7: `timing_07.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 3 3 1;2;4;8;16;32 10
    timing_08.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
3	3	1	10	12	24	4.0	0.014
3	3	2	10	24	54	8.2	0.008
3	3	4	10	48	132	99.8	0.034
3	3	8	10	96	360	256.3	0.172
3	3	16	10	192	1,104	848.9	1.504
3	3	32	10	384	3,744	3,539.7	39.463

TABLE A.8: `timing_08.csv`

```
java -cp ..\..\target\roasolver.jar;. RoaSolverTiming 10 10 1;2;3;4;5;6 10
    timing_09.csv
```

$\dot{n}$	m	c	T	#var.	#con.	#so	time (s)
10	10	1	10	110	220	11.0	0.019
10	10	2	10	220	460	53.5	0.031
10	10	3	10	330	720	501.6	0.564
10	10	4	10	440	1,000	693.4	2.536
10	10	5	10	550	1,300	2,625.5	29.629
10	10	6	10	660	1,620	4,232.9	110.779

TABLE A.9: `timing_09.csv`