— Independent Work Report, Spring 2018 —

A Modular Framework for Developing, Deploying, and Evaluating Game-Theoretic Strategy Design Exercises

Andrew Wonnacott
Advisers: Professor Dave Walker, Professor Matt Weinberg

Abstract

Student interest in "Economics and Computing", Princeton's undergraduate computer science course in algorithmic game theory, has grown rapidly in recent years. This course employs small simulations to develop students' ability to use theoretical tools to analyze more real-world situations. These strategy design exercises require considerably more software infrastructure than a usual theoretical problem set would, including handout code for student self-evaluation and software auto-graders. While some frameworks for automated grading exist, none surveyed provided the specific features desired for repeatedly developing very similarly structured assignments and evaluating student submissions in the context of each other. In order to facilitate more automated grading of these assignments in this and future semesters, I have developed a framework of modular, extensible tools for building all of the necessary components of these strategy design assignments. Concurrently, in my role as a grader for COS 445, I used the framework to develop and evaluate four strategy design assignments, using assignment specifications developed with Professor Weinberg. This provides feedback from real student and instructor users which was used to improve the design. Empirical results demonstrate the importance of quality tooling to effective use of instructor time, low broken submission rates, and high student satisfaction.

Introduction

My work builds upon existing programming-based game theory exercises assigned in an existing Princeton course, COS 445 "Economics and Computing". The intention of these strategy design exercises is to teach students how to use game theory to analyze a real world situation and work rationally within a healthy incentive structure where students do not try to hurt each others' performance. Each past offering of the course has rebuilt these assignments, and all associated instructor tooling, from scratch, resulting in unnecessarily high allocation of instructor time to grading. Additionally, the course has grown rapidly to become the single largest theoretical computer science course ever taught at Princeton.[1][2]

The goals of this project are to discover to what degree the development of new strategy design exercises can be simplified or automated, to improve the student resources available during the completion of these exercises, and to create institutional memory and decrease year-to-year repetition. This was accomplished by designing a modular, extensible, and reusable framework which embodies the commonality between strategy design exercises, allowing instructors to create new exercises while changing only the necessary, assignment-specific details, and to develop advanced automated grading tools than previously existed for the course. Within this framework, I have built a collection of utilities to facilitate students' evaluation of their own strategies and to decrease strategy iteration overhead.

Through this work, I aim to improve the quality of computer science theory education from the perspectives of both the students and faculty associated with COS 445. Quantitative metrics, including software failure rates and instructor time usage, and qualitative metrics, including student and instructor satisfaction, both illustrate the improvements to the course and emphasize the importance of quality tooling to quality programming-based assignments.

[1] registrar data here

[2] More students complete COS 340 in a given year, but it is taught in both fall and spring semesters.

Problem Background

Four Princeton University departments offer courses in game theory: Economics, Politics, Mathematics, and Computer Science. While all of these courses teach the core concepts of game theory, each also connects this core to relevant examples and topics in its department. For instance, the Computer Science department's "Economics and Computing" places emphasis on the connections between game-theoretic results and algorithms, software, and other aspects of computer science. The course focuses on the relevance of game theory to matchings, auctions, cryptocurrencies, and other multi-actor computational systems.[3] It is also represented in the design of the problem sets, which, in addition to traditional proof-based exercises, contain programming-based exercises which connect course topics to prerequisite computer science skills. These programming-based exercises take a common form.

Students are presented with an iterated game, described both in the assignment handout and as a Java interface. A classic iterated game is the iterated prisoner's dilemma[4], in which two players must collaborate to receive the best outcome, but are each incentivized to defect, which results in a less preferred outcome, and have access to the other player's previous decisions when making their own decision. While this game is well-studied and models some real-world incentive structures, the course also draws on various real-world phenomenon to design iterated games, including a college admissions game from the perspective of stable matchings, a search advertisement bidding game using auction theory, an election informed by voting theory, a gerrymandering-prevention simulation inspired by fair division, and exploitation of a fictional cryptocurrency by analyzing blockchain incentives. These examples are not exhaustive, but illustrate the variety of course topics with potential for translation to iterated games.

Students then craft strategies to play the specified game, based on their own analyses of the game's equilibrium and on which strategies they expect their peers to play. These strategies are evaluated on both on their design quality and on their empirical performance. While design quality

[3] https://www.cs.princeton.edu/ smattw/Teaching/cos445sp18.htm

[4] Figure: PD game -> Java interface

is evaluated on a purely individual basis, empirical performance is measured by simulating the play of the collection of submitted student strategies against each other. For instance, for the iterated prisoner's dilemma, every student strategy plays one 'full' game, consisting of many iterations of the prisoner's dilemma, against each other student strategy. Strategies are scored based off the average payoff they achieve. For the search advertisement game, all the student strategies are concurrently asked to bid on the same search terms. They are scored based on their revenue from ad clicks less their expenditures on advertising, and each bidder knows the click-through rate for a term prior to bidding on that term. To create a more realistic simulation, students are graded solely on the performance of their own strategy, not on a curve of the performance of their peers. That is, student grades on these strategy design exercises are not zero-sum.

While the strategy design exercises assignments are popular with students[5], they also demand considerable instructor resources be dedicated to building and grading software resources, which have historically been significantly rebuilt for each assignment. Worse still, due to course staffing changes, this work has not been preserved between semesters, resulting in inefficient allocation of department resources to producing new exercises without the institutional memory or software resources previously developed. Due to the limited tooling provided in past semesters for students to test their code, many submissions did not compile or caused runtime exceptions, requiring considerable additional instructor resources.

Related Work

This work was developed to replace previous software design exercises assigned to COS 445 students[6] in Spring 2017.[7] In one problem set, students developed and implemented strategies for three different strategy design exercises related to three different classical games: Prisoner's Dilemma, Centipede, and Ultimatum. In a later problem set, an exercise placed students in the role

[5] https://reg-captiva.princeton.edu/chart/index.php?terminfo=1184&courseinfo=012095 requires CAS login
[6] including the author
[7] https://web.archive.org/web/20170916042726/https://www.cs.princeton.edu/ smattw/Teaching/cos445sp17.htm

of bidders in a generalized second-price auction.[8]

Both problem sets were implemented and graded by Cyril Zhang[9], a graduate student TA for the course. Zhang developed interfaces, dummy implementations, and grader software for each strategy design exercise.[10] However, his implementation of software for strategy design assignments was intended for grader use only. It is not modular and has to be completely rewritten for each assignment. The student interfaces were documented, but the grader software was not documented for future use. Additionally, some of the scripts and notebooks used by Zhang were not preserved and were not accessible as resources. Thus, while these resources served as inspiration for this work and provided some assignment-specific code to my work developing specific assignments, I did not build directly on Zhang's code.

Prior to Spring 2017, COS 445 was taught by Professor Mark Braverman in Fall 2012 and Spring 2014[11]. Though no documentation or archives of the course homepage from these semesters is available online or in the Internet Archive, I recovered some resources from the course account on the "Cycles" cluster of the Princeton Computer Science department and from the source code repositories used by the course staff. In both semesters, Yonatan Naamad[12] was responsible for software development.

The only resources available from Fall 2012 were collected from the course Google Code repository.[13] This repository contains incomplete Python skeleton code for a basic tournament of an unspecified game. I uploaded this code to the current course repository[14] but did not develop it further.

The Spring 2014 iteration of the course was developed on a private GitHub repository.[15] Through a clone of this repository stored on Cycles, I discovered that the course had PHP and Java imple-

[8] an auction of multiple items, one after another, where the winning bid pays the second highest bid. This is the auction format used by Google to sell sponsored search auctions.

[9] https://github.com/cyrilzhang

[10] https://github.com/PrincetonUniversity/cos445-hw/tree/s17 requires invite: contact author or department

[11] https://precourser.io/course/1184012095 requires Princeton CAS authentication

[12] https://github.com/ynaamad/

[13] https://code.google.com/archive/p/cos445-scripts/

[14] https://github.com/PrincetonUniversity/cos445-hw/tree/f12 requires invite: contact author or department

[15] https://github.com/ynaamad/cos445

mentations of strategy design exercises for Prisoner's Dilemma and another auction setting. The Java code was the student handout, which contained strategy interfaces and some testing code. The testing code was not designed to be automated for grading and was not modular. It tested manually-entered strategies against each other, and assignment-specific details, such as game structure and payoffs, were integrated within the simulation code. As a design decision I dismissed the available PHP code, which was used for course grading, due to language unfamiliarity. These resources were uploaded to the current course repository,[16] but were not reused, in part because they were located and examined only when I was given access to the course site in order to deploy my project.

Beyond COS 445 at Princeton, many universities have taught computer science courses in algorithmic game theory. However, a survey of several universities' courses, including courses at Harvard[17], Stanford[18], Cornell[19], U. Washington[20], Yale[21], U. Penn[22], Georgia Tech[23], Carnegie Mellon[24], and Duke,[25] identified only one other course with similarly structured assignments.

Harvard's Computer Science 136, "Economics and Computation",[26] assigns two programming problem sets. The first requires implementation of a peer-to-peer file sharing system similar to Bittorrent, a protocol in which incentives parallel Prisoner's Dilemma. However, the assignment is not a competition and student solutions are not differentiated based on strategic properties. The second assignment is a strategy design exercise built on the generalized second-price auction, in which student strategies vary and are graded based on their performance in a competitive environment. This is very similar to both the Spring 2014 and Spring 2017 auction exercises of COS 445 at Princeton. However, the grading structure of this assignment is quite different from the grading structure of the COS 445 strategy design exercises, and that the implementation is in

[16] https://github.com/PrincetonUniversity/cos445-hw/tree/s14 requires invite: contact author or department
[17] https://beta.blogs.harvard.edu/k108875/
[18] https://theory.stanford.edu/ tim/364a.html and https://theory.stanford.edu/ tim/f16/f16.html
[19] https://www.cs.cornell.edu/courses/cs6840/2014sp/
[20] https://courses.cs.washington.edu/courses/cse522/05sp/
[21] https://zoo.cs.yale.edu/classes/cs455/fall11/
[22] https://www.cis.upenn.edu/ aaroth/courses/agtS15.html
[23] https://web.archive.org/web/20131002121909/https://www.cc.gatech.edu/%7Eninamf/LGO10/index.html
[24] https://www.cs.cmu.edu/ arielpro/15896s16/index.html
[25] https://www2.cs.duke.edu/courses/spring16/compsci590.4/
[26] https://beta.blogs.harvard.edu/k108875/assignments/

Python. While comparison between COS 445 assignments and CS 136 assignments in a future project might provide some pedagogical value, such work would be out of scope for this project. Given that these assignments are implemented in Python, only handout (and not grading) code is available, and the assignments are graded differently, these resources were dismissed as a base for development of new resources for COS 445.

Many universities, including Princeton, maintain extensive systems for the automatic submission and processing of student assignments. Such resources are useful in the general case of developing new programming-based assignments, but no such resources replicate the behavior implemented in this project. Most auto-grading systems are designed with an assumption that each student submission is graded in isolation from other student submissions. Such systems are not applicable to this project since the strategy design exercises involve evaluation of student strategy performance in a common environment. Though existing automatic grading systems cannot replicate the full extend of this work, I integrate existing resources for assignment submission and student feedback to perform smaller tasks within a larger system. I integrated CS DropBox[27] functionality where relevant, and modeled the leaderboard structure after review of the leaderboard implementation used in other Princeton CS classes. Generally, no such tools would leverage the high degree of structural similarity between strategy design exercises, and a dedicated project for these exercises will allow for more efficient development and evaluation.

Approach

Cyril Zhang completed the TA requirement for the PhD program last fall, so at the start of this semester it was unclear who among the graduate and undergraduate course staff would be responsible for maintaining the COS 445 strategy design exercises. Therefore, the course needed not just long-term work to improve student and instructor resources, but also short-term work to maintain and debug the same resources. These coinciding requirements inspired me to design one extensible implementation of all the reusable components for software design exercises, and to develop the

[27] https://csguide.cs.princeton.edu/academic/csdropbox

project iteratively in conjunction with its usage in the course. This software would naturally grow throughout development as demanded by the needs of the course, and the final product will naturally lower-bound the maximum degree of automation and modularity possible.

Before the first problem set of the course was assigned in this semester, Professor Weinberg and I designed a handout interface and tools to allow students to test their strategies against each other, which I then implemented. The game simulates college admissions using stable matchings. While I implemented this handout with the explicit goal of producing a tool to test out multiple different student strategies in the specified application environment, I followed the principles of modular software design taught in COS 217.[28] I factored out the code which implemented the Gale-Shapley deferred acceptance algorithm for stable matchings from the code which performed randomized trials of the stable matchings and produced performance statistics for students. When it came time to implement the second strategy design exercise, I was able to reuse code which processed results of trials, even though the nature of the trials changed from college admissions to elections. Additionally, when it came time to grade the problem set, I was able to modify only the post-processing step to compute grades. Thus, creating a new assignment required constant work in the number of framework extensions, and creating a new framework extension requires constant work in the number of assignments. The result of this process duplicates significant functionality from last year's assignments. However, because of conscious design decisions to factor out repeated code from assignment implementations, it accomplishes what previous implementations had not. It will be modular, as described above, so that new extensions such as additional tools or interfaces can be written for strategy design exercises. Throughout the semester, I can leverage my role as an undergraduate course assistant for COS 445 during the current semester in order to identify and produce such extensions. It will be reusable, both for the same assignments in future years and for new strategy design exercises implemented by future course staff with lower overhead.

In order to increase the reusability, I elected to store it in a University-owned private GitHub repository and documented the repository structure and contents. This ensures that future course

[28] Kernighan and Pike chapter 4

staff will have the physical and technical means to maintain this project after I have graduated.

Implementation

Preliminary Choices

The first key decision in the implementation of these programming assignments was language selection. Many students suggested the assignments be implemented in Python,[29], and this would be preferred by the course staff as well. Members of the Mechanism Design group are not expected to have working proficiency with Java, but many have enough experience with Python to support students. Additionally, Python's conciseness and ability to load classes from user input would be useful when implementing tools which are parameterized on user-input strategy names. Unfortunately, COS 445 only prerequisites Java experience, and not Python experience, so students reasonably expect they will only be asked to write Java code. Data from other course surveys indicate that 90% of students report themselves fluent in Java after the Princeton introductory sequence, compared to 30% for Python, confirming that the student work for the assignment must be in Java.[30]

Given this constraint, and lacking any other language selection pressures, I elected to implement as much of the assignment code as possible in Java, with limited usage of scripting languages such as Python, Bash, and GNU Make. These scripting languages were used only where Java would not suffice and only where implementation details were irrelevant to student assignment comprehension.

Java also offers convenience to assignment designers by enforcing an object-oriented design pattern. This allows instructors to communicate assignment specifications formally through a well-documented interface. Furthermore, by constructing new instances of student implementations between game rounds, instructors can programatically enforce that students do not employ learning strategies outside the parameters of the assignment.[31]

[29] per survey 1 results

[30] COS 333 survey data

[31] This requires that instructors forbid the usage of static variables, but it is trivial to verify compliance with this requirement.

However, several default Java behaviors do not map cleanly onto the desired properties of strategy design exercises. For instance, Java objects used as function parameters are passed by mutable, nullable reference. This is in contrast to functional languages (including Ocaml and Rust), where parameters are immutable and nonnull unless otherwise explicitly specified. Current industry standards for Java code obviate both of these issues using modern language features.[32][33][34] In order to preserve backwards compatibility, these features are not implemented by default. Where possible, I have taken advantage of these features. Since the assignment specification documents that students are required not to modify parameters, instructor code can use unmodifiable (immutable) containers. However, I have not included NotNull annotations because they are not taught in the prerequisite courses and, as previously noted, these assignments should not require students to self-teach any programming skills. In my own framework code, which must be supported by future instructors, I decided to assume support for Java 9, released in fall 2016, and to use modern language features, since course instructors are at least as likely to have industry experience with modern Java code as experience with the Princeton introductory sequence.[35]

Handout Implementation

In order to design my framework, I reviewed non-modularized code from the Spring 2017 auctions exercise. This is the only assignment for which I had access to both the student handout and the grader code. After the game design is complete, the first piece of the assignment to be implemented is the student interface:[36]

```
import java.util.List;

public interface Bidder {
    // Return your bid for the current day
    // Called once per day before the auction
    public double getBid(double dailyValue);
```

[32] https://github.com/google/guava/wiki/UsingAndAvoidingNullExplained
[33] https://github.com/google/guava/wiki/ImmutableCollectionsExplained
[34] https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableList(java.util.List)
[35] https://www.techworld.com/news/developers/java-9-release-date-new-features-3660988/
[36] TODO: make this code a figure

```
    // Let you know if you won, and how much the winners paid
    // Called once per day after the auction
    public void addResults(List<Double> bids, int myBid, double myPayment);
}
```

This interface describes the necessary functions of a student strategy. Each day, they must bid on an auction of advertising slots which appear on a search results page (often called "sponsored search results"), and they assign different values to words auctioned on different days. The strategy also has access the winning bids for each of the ten sponsored results from each past day and which of those bids were its own.[37] This allows the strategy to obey the budget constraint: across all 1000 days, it may not spend more than 500. These constants are not documented in the interface as designed, only in the assignment specification. The instructors distribute a simple sample student implementation — in this case, the truthful bidder. Zhang also implemented a program used to grade the student-submitted strategies, which consisted of two methods: `main` and `trial`. The former ran 100 trials and printed out the results, averaging the outcomes across each trial for each student. The latter contained an implementation of the generalized second-price auction format described in the assignment specification.

Having reviewed a former assignment, I implemented the first assignment handout for the course this semester: the college admissions simulation described previously. In doing so, I made several implementation decisions in order to facilitate future reconfiguration of both the game being played and the post-processing of tournament results, which are representative of design decisions made throughout the assignment. In every assignment, the tournament simulation portion (instructor-written code) is broken up into game rounds, result processing and amortization, and further post-processing, such as grade assignment and output formatting. The divide between game rounds and code which processes game results was present in Zhang's auction code, as the separation between the `trial` and `main` methods. However, this prior work did minimal amortization and no advanced output formatting, and had no separation between them. While the handout code written

[37] When a strategy wins, it is also told how much it paid.

for the first assignment did minimal post-processing, this distinction was quite useful when later developing the auto-grader and additional extensions.

During the development of this handout, however, the statically-typed nature of Java raised some difficulty. Zhang's grader code had student submission names hardcoded into complex expressions, as though they had been copied in from a manually-written list formatted using `sed`. In the context of developing a handout to allow students to test multiple strategies of their own design, I determined that requiring students to modify the simulator Java code whenever they changed which strategies they wanted to test would be so cumbersome that the simulator might not have been distributed at all. Additionally, graders desire functionality to configure the list of evaluated strategies without having to manually list out all the student strategy names and to modify this list repeatedly to identify and diagnose misbehaving student strategies. Since Java is not able to type-check user-input strategy names, I determined that the most effective workaround for this issue would be to automatically rewrite the Java code before compile-time.

For this purpose, I developed a simple Python script capable of rewriting arbitrary template plaintext files using a CSV file. This script, `formatBase.py`,[38] builds a list of dictionaries, one for each row of the CSV, using the header of the CSV as the keys and the contents of the row as the values. In its input, it identifies lines which start with `//R` and end with `//`. For each such line in its input, it outputs multiple lines, each corresponding to the input line rewritten using a distinct row of the CSV by replacing each curly-enclosed key with the corresponding value. It also rewrites the key `0` with the row number from the configuration table. This script directly injects the contents of the configuration file into the Java code with no escaping, as is necessary in this application. However, this practice is extremely dangerous, since using a user-submitted configuration file would allow users to execute arbitrary code on the evaluation machine. Fortunately, this is not a security concern, since students already run arbitrary code on the evaluation machine: their submitted strategy.

Given a CSV file as input with the header row `netID` and a row for each student's name, the following function is thus rewritten to instantiate one implementation of each Student class listed in

[38] TODO: appendix this

the configuration file: [39]

```
// S, T, W are simulation parameters documented in the student interface
public static int[] oneEachTrial(double S, double T, double W)
{
    // Initialize students
    Student[] students = new Student[N];
    //R students[{0}] = new Student_{netID}();//
    return runTrial(students, S, T, W);
}
```

My modular design allows for data collection code to vary without additional overhead. The following method instantiates an instance of an instructor-provided strategy using reflection, in addition to the strategies specified in the configuration file. This method is useful for grading assignments. [40]

```
public static <Student_T extends Student> int[] withExtraTrial(
        Class<Student_T> clazz, double S, double T, double W)
{
    // Initialize students
    Student[] students = new Student[N+1];
    //R students[{0}] = new Student_{netID}();//
    try {
        students[N] = clazz.getConstructor().newInstance();
    } catch (ReflectiveOperationException roe) {
        roe.printStackTrace();
        System.exit(1);
    }

    return runTrial(students, S, T, W);
}
```

Given all of the components described above, I have developed a considerably improved the assignment handout. Previously, the handout contained the strategy interface and a simple implementation strategy with a sample testing method for that implementation only. This semester, assignment handouts contained not just the strategy interface and simple implementation, but also a full copy of the above game round simulation, complete with amortization by many randomized trials, which simply prints the average score of each strategy. While working on the assignment,

[39] use figure
[40] todo: figure

students can reconfigure the testing code easily by modifying the configuration file to contain whatever strategies they want to simulate. For instance, given strategies `Student_random.java` and `Student_favorites.java` which apply to random schools and their favorite schools in the above college admissions simulation, all the student would need to do to simulate 10 submissions using the former strategy and 15 using the latter would be set the configuration file to ten lines of `random` and 15 lines of `favorites`, and then run `make test`. Whereas, last year, each student would have had to manually write simulation code to run the exact collection of strategies they wanted to simulate, the new handout will automatically rewrite the simulation code to simulate those strategies using `formatBase.py` and then compile and run the simulation with no additional overhead.

Auto-Grader Design

The two most important components of each exercise are the handout code for the students and the auto-grader code for the instructors. However, in order to design the auto-grader, Professor Weinberg and I had to develop a well-specified, portable standard for grading strategy design exercises. A key constraint in the grading of these assignments is that the incentive for the students must be to try to maximize their own strategy's performance. Therefore, curving each student's grade based on the percentile of their strategy's performance is not acceptable. In this scenario, a student would be as happy to hurt their peers as to improve their own strategy. For instance, in an iterated prisoner's dilemma, if students are graded based on relative performance, then they are apathetic between both cooperating and both defecting. So there is no incentive under any circumstance to cooperate, even in the face of potentially irrational counterparties, and every student should always defect.

One feasible grading strategy, given this constraint, would be to grade student performance without any curve at all. That is, in a simulation where the best possible performance is 5000 points, and the worst possible performance is 0 points, students would receive 3/5 on the assignment if they scored 3000 points on average. However, in this case, student grades can be negatively impacted

by the decisions of their peers. For instance, in the prisoner's dilemma, for every student who always defected, every other student would have the cap on the maximum possible grade they could receive lowered. Students' grades would be dominated not by the performance of their own strategy but by the decisions of their peers. Course staff, including Professor Weinberg, agreed that this is unacceptable.

"Replacement-comparison grading", the grading rubric used in the course, was developed by Professor Weinberg and myself as a dependency of my work on this project, and satisfies the need for an incentive-correct grading rubric which could be implemented in an auto-grader. It is based on the following intuition: to grade a given student's strategies in a room with their peers, substitute each instructor in their place while making no other changes to the room, and measure whether the student does better or worse than that instructor would have in their place. The performance of a student strategy S_i among strategies $S_1 \dots S_n$, pro-rated based on the potential best and worst possible performances by any strategy in the simulation, can be captured by comparing the score x_i of S_i against $S_1 \dots S_{i-1}, S_{i+1} \dots S_n$ to the scores $y_{i,1} \dots y_{i,m}$ of a predetermined menagerie of m instructor strategies $I_1 \dots I_m$, where each $y_{i,j}$ in $y_{i,1} \dots y_{i,m}$ is the score of I_j against $S_1 \dots S_{i-1}, S_{i+1} \dots S_n$. Students whose strategies outperform all instructor-designed strategies have done better than the instructors would have at the assignment, and receive the best grades. Some instructor-designed strategies are intentionally imperfect or misguided, in order to differentiate between less ideal student-designed strategies.

This grading strategy has the correct incentives: each student will only work to maximize their chance of beating each instructor threshold, and not to sabotage their peers' performance. Since the instructor strategies used for grading are not revealed and the pool of other student submissions is unpredictable, the distribution of instructor thresholds approaches uniform, and students will attempt to beat as much of the real interval of the score range as possible, producing the same behavior as students maximizing expectation. Additionally, by amortizing student performance over many randomized trials — for instance by reassigning student and university preference orderings in the college applications game independently between trials — the law of large numbers guarantees

to students that their grade represents their expected performance without uncertainty based on variance between trials. Thus, students are not incentivized to play a lower-expectation strategy in order to increase their variance.[41] This system is not vulnerable to students attempting to sabotage instructor scores for their own benefit, since instructor scores for a given student are calculated without the play of this student.

Though the above strategy achieves all the correctness requirements, it has a fatal flaw: when there are n student strategies and m instructor strategies, and a trial with n strategies takes $t(n)$ time, this grading metric takes $O(m \cdot n \cdot t(n))$ time. After approval from Professor Weinberg, I implemented a simple optimization to bring down the time requirement to $O((m+1) \cdot t(n+1))$ time.[42] The modified replacement-comparison grading scheme measures student scores $x_1 \dots x_n$ for all students $S_1 \dots S_n$ amortized across many trials, then for each instructor strategies I_j among $I_1 \dots I_m$, measures their score y_j against all of $S_1 \dots S_n$. That is to say, instructor strategies are assigned scores based on their performance against all the students, while student strategies are assigned scores based on their performance against all the other students.

The only difference between the initial and modified replacement-comparison schemes is that each student can have some small impact on the thresholds used to grade themselves, not just their peers. As an unfortunate consequence of this simplification, if a student was able to hurt instructor scores, they might perform better. However, since students cannot identify when they play against an instructor, as opposed to a peer, they must do so by playing to sabotage all their peers' scores. When multiple students try to sabotage instructor scores, they also sabotage each other, and all student scores will be correspondingly lower. In this situation, the damage to the grader strategies' scores is representative of lower potential performance by any student strategy. Thus, the modified replacement-comparison scheme is sufficient.

[41] The same analysis holds for all higher moments of their strategy.

[42] So long as t grows slower than $n!$, this is a performance improvement; for practical reasons, all simulations used in this course take polynomial time in the number of students.

Auto-Grader Implementation

The goal of this project is not just to build a handout and an auto-grader, but to build a framework in which the development of the grader as described above requires no modification of assignment-specific code. Consider the components of the simulation handout:

- `runTrial` method which takes a list of strategies and simulates their behaviors, e.g. in a college admissions simulation
- data processing methods, e.g. `oneEachTrial`, which builds a list containing each configuration-specified strategy once,[43] and `oneEachTrials` (not shown above), which runs `oneEachTrial` a large number of times and averages the results
- post-processing methods, e.g. outputting a CSV file containing student names and their average performance in the simulation

Only the first of these contains assignment-specific code. In order to implement replacement-comparison grading, I implemented the following methods:

- two new data processing methods: `withExtraTrial`, which builds a list containing each configuration-specified strategy once, and one extra strategy passed in as a parameter[44], and a corresponding method `withExtraTrials`, which averages the above. These are very similar to the existing data processing methods
- A new post-processing method that computes grades for each student given how many instructor strategies each student strategy outperformed and a lookup table to translate this win-count to an assignment grade. This method, `writeGrades`, is included as appendix [45]

```
public static void writeGrades () {
    final int numTrials = 1;
    //R netids[{0}] = "{netID}";//
    // defaults as specified in assignment handout PDF
    final double S = 100, T = 100, W = 10;
    // Compute x_i
    double[] oneEachTrials = oneEachTrials(100, S, T, W);
```

[43] see figure above
[44] see figure above
[45] TODO: make into figure/appendix

```
        // Instructor strategies
        List<Class<? extends Student>> ourStrats =
            new ArrayList<Class<? extends Student>>();
        ourStrats.add(Student_REDACTED.class);
        // a total of twelve such lines like the above line
        double[] winsToPoints = new double[]{
            0, 0.5, 1.0, 1.2, 1.5, 1.8, 2, 3, 3.5, 4.0, 4.5, 4.8, 5
        };
        // Compute \sum_j \delta_{x_i > y_j} for all i
        int[] wins = new int[N];
        for (Class<? extends Student> strat : ourStrats) {
            double[] res = withExtraTrials(strat, numTrials, S, T, W);
            for (int i = 0; i < N; ++i) {
                if (res[i] > res[N]) {
                    ++wins[i];
                }
            }
        }
        // Output grades as csv
        System.out.println("netID,points");
        for (int i = 0; i < N; ++i) {
            System.out.println(netids[i] + "," + Double.toString(oneEachTrials[i])
                               + "," + Double.toString(winsToPoints[wins[i]]));
        }
    }
```

Given only these two changes, the handout code has been transformed into an auto-grader. Additionally, neither of these changes required alteration of the `runTrial` method, so minimal additional work is necessary for grading an assignment once the handout has been made. Only filling in the instructor strategy names and scoring rubric is needed.

In summary, each assignment consists of a strategy interface, implementations of that strategy, a list of those implementations, a `runTrial` method which specifies how to evaluate those strategies, methods which run specific subsets of strategies, methods that post-process the results of those trials to provide low-latency feedback or for auto-grading, and automating these steps with a Makefile to configure, compile, and run the assignment simulation. [46] [47] [48]

[46] TODO redraw using Powerpoint
[47] TODO add formatBase.py config csv
[48] TODO use this diagram when I talk about modularization earlier

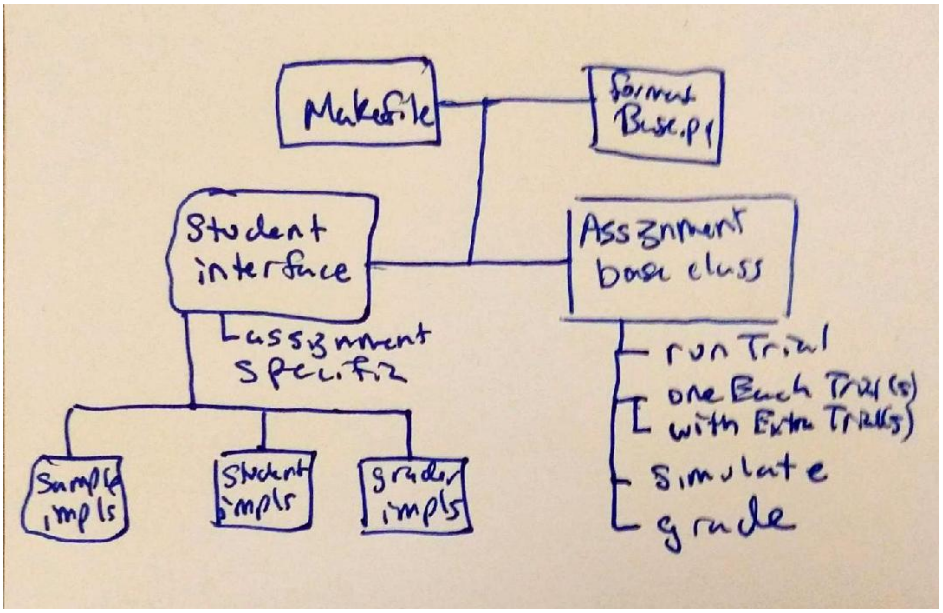

Figure 1: TODO: This is a gray image.

In addition to easily altering an assignment handout into an auto-grader, my framework also makes it easy to build new assignments. To adapt the college admissions exercise described above into the elections simulation of the second problem set of COS 445 this spring, only a few changes were necessary.

Specifically, I had to develop changes only to assignment components which were not modified between the handout and the grader. The written assignment specification describes a new strategy format, so a new Java interface file was necessary. Of course, all the implementations of this interface were also written from scratch for the new assignment, all of which are fairly concise but express assignment-specific strategies. Finally, the `runTrial` method of the simulator had to be rewritten to simulate an election instead of college admissions. No semantic changes were necessary to the data processing or post-processing methods for either the handout or application.

Development of Further Tooling

As currently implemented, each assignment was distributed as a standalone zip file through the course website. Even though significant code was shared between the two assignments described above, they were distributed as separate files. While further modularization would allow code that changes each assignment — describing how to simulate a game round — to be separated out from

code which changes between the handout and the auto-grader, I determined during assignment development that maintaining separate code bases for each assignment has significant advantages. Some simulation parameters, such as the budget constraint or number of auctioned items in an auction, vary between assignments. Often, strategies are evaluated based on their performance relative to baseline instructor strategies in several different regimes of simulation parameters. This will involve tweaks to the grader code outside of the scope described above. While these changes are minimal and necessary, keeping details of grading separate between assignments proves useful in the real world.

Thus far, I have illustrated the strengths of my project at performing tasks which were envisioned from the outset. The strength of modular implementation also comes from the ability to adapt to changing demands easily in real time. In this way, I was able to extend my framework throughout the semester in order to address my needs as a course assistant to remedy issues with the strategy design exercises. As shown below,[49] the handout code enabled students to verify that their code compiled and ran. However, several students submitted non-functioning strategies due to environmental differences, demonstrating the classic issue "Works On My Machine".

Other computer science courses at Princeton employ an online tool in the CS DropBox called "Check Submit". This tool allows instructors to specify tests to be run on department-configured machines using student code, and enables students to evaluate the performance of their submission against these tests without the concern that the software environment on their machine might not be compatible. I enabled "Check Submit" for COS 445 strategy design exercises as a part of this work by developing simple Bash script and Makefile which adapts my assignment framework to this new setting. My "Check Submit" script copies in the assignment handout from a predefined location into the student submission folder, detect the student submission name, and write the `formatBase.py` configuration file to run their submission. Now, without any modification of the assignment code, every strategy design exercise automatically has DropBox "Check Submit" functionality possible.

This confirms to students that their strategy will not be incompatible with the grading environment,

[49] results section ref here

improving student satisfaction with the course. However, students would still benefit from more realistic testing. Thus, I developed a leaderboard so students could better understand how a strategy would perform against more realistic collection of peer strategies. First, I initialized a new DropBox assignment for students to submit strategies to the leaderboard, though these strategies need not be their final assignment submission. I implemented the leaderboard by writing a simple Bash script which scrapes all student submissions from the DropBox, automatically writes the configuration file to run every students' submission together, and runs the simulation provided in the student handout as above. Because the simulation output is an easy-to-parse table, I can automatically reformat it into an HTML table and place this in a web page accessible through the department web servers. The script to update the leaderboard runs through a `cron` job at a customizable interval.

While this leaderboard provides additional feedback, there are unavoidable issues when students submit broken code. Since all student submissions are run together, one non-compiling or exception-throwing submission disables the leaderboard for all students. I worked around this issue by disabling assertions, which were used in `runTrial` to automatically halt simulation whenever a student submission performed behavior in violation of the assignment specification. With this modification, the code automatically falls back to silently overriding illegal values from student strategies with sane but selfless defaults so that the leaderboard is correctly generated even when some student submissions are faulty. Some student code also caused compilation failures, which are more difficult to work around. We enabled the "Check Submit" button to leaderboard so that students could check their leaderboard strategies would compile in the department environment. This mostly resolved the issue.

Results

This project has already achieved minimum viable product by providing a framework for implementation of strategy design exercises in the current year of COS 445. These assignments and the framework I developed are documented and available at the course GitHub site.[50] Thus, they will

[50] https://github.com/PrincetonUniversity/cos445-hw/tree/s18 requires invite

be available for reuse in future offerings of the course.

As the goals of a project naturally define evaluation criteria, I have evaluated my project on the basis of each those goals: decreasing the work to build each assignment within a semester and the instructor time allocated to assignment implementation, eliminate the inter-year assignment reconstruction, improve available student resources, and in doing so improve student and instructor satisfaction.

- discover modularity and decrease assignment to assignment rewriting
  - code rewriting measurements
  - instructor time usage
- improve student resources
  - list resources created
  - polling data about student usage
  - empirical results about student error rates "Illegal submissions" constitute student strategies which do not compile, throw exceptions, violate assertions, or otherwise behave outside of the parameters of the simulation.[51] The illegal submission rate measures the percentage of students who were unable to develop a quality strategy for reasons other than lack of mastery of the course content.
- create institutional memory and decrease year-to-year rewriting
  - cant measure until next year, but set up for it
- increase student and instructor satisfaction
  - polling data about student usage

Conclusions and Future Work

Through this work, my goal has been to improve the quality of present and future computer science theory education, from the perspectives of both the students and faculty associated with COS 445.

[51] Poorly performing strategies which obey the specification are legal.

Student and instructor feedback indicates that my work has supported high quality strategy design exercises in the current, Spring 2018, offering of COS 445. Ease of development and grading of the later assignments in the course confirms that I made substantial progress in simplifying and automating the instructor experience. Low rates of illegal student submissions confirm that my work has allowed students to focus on core assignment content without experiencing undue difficulties with assignment structure or available tooling. I have achieved my goals in the present time-frame.

I intend to continue to use these tools as a returning undergraduate course assistant when this course is taught in Spring 2019. However, I will graduate at the end of that semester and will be unavailable in subsequent offerings of COS 445. I cannot yet measure my success at preventing this work from being repeated in any future year, but Professor Weinberg has confirmed that these resources will be provided to future course staff tasked with operating the strategy design exercises.

In Spring 2019, while reusing these resources, I will further simplification and automation is supported within the existing framework as I am able. While currently assignments are distributed independently, so no code may be shared between assignments, I will revise the assignments to share one implementation of the handout post-processor and auto-grader post-processor. This will complicate the assignment handout process slightly, but will simplify code reuse and iteration. Earlier assignments were built off of earlier versions of the infrastructure and will need to be modified where they differ from newer versions when they are reused, and sharing code between assignments will prevent this inconsistency from developing again.

Recently, I became aware of a Java interface for loading classes at runtime. This technology, called ClassLoader,[52] was implemented in the Spring 2014 handout code. I was unable to use ClassLoaders initially because I did not gain access to the Spring 2014 code until later in the semester. This simple modification will eliminate Python dependency currently required by `formatBase.py` and allow students to better understand and modify handout simulation code.

Though few other universities currently teach courses in algorithmic game theory as undergraduate computer science, the field is growing and such courses will inevitably become more common. As

[52] https://docs.oracle.com/javase/9/docs/api/java/lang/ClassLoader.html

they do, they will look to COS 445 for resources, along with Stanford CS 269I, C.M.U. 15–196, and Harvard CS 136. It is my hope that my code could be understood and assigned such courses, with modification to suit the topics covered.

Acknowledgements

First of all, I would like to thank Professor Weinberg for his leadership in designing the written specifications and grading rubrics for the strategy design exercises, and for his dedication to teaching and organizing COS 445. His dedication to the course engaged me as a student last year and was invaluable to my work this year. I would like to thank Cyril Zhang for his work developing the implementation of the strategy design exercises I completed as a student, which inspired and served as a basis for this work. Thanks to Jérémie Lumbroso for providing me with the University-owned GitHub workspace in which my project has been stored in order to preserve it for future semesters of COS 445.

By grading student write-ups for strategy design exercises, Leila Clark, Wei Hu, Heesu Hwang, Dylan Mavrides, Andreea Măgălie, Divyarthi Mohan, Ariel Schvartzman, Matheus Venturyne, and Evan Wildenhain provided me feedback about the quality of the handouts produced from my work and passed along student feedback. These details provided crucial insights as to the clarity of the resources I have developed and allowed me to provide targeted documentation where my code was unclear. Thanks also to all those students of COS 445 Spring 2018 for their feedback on the assignments used in the course, which informed my development of this project.

I created this work within the framework of Professor Dave Walker's Independent Work Seminar, IW 09: "You Be The Prof". Professor Walker kept me organized and on schedule, and his guidance helped me identify and describe the topic of this work. I also appreciate the feedback I received throughout the semester from my peers in Professor Walker's seminar, Noah Beattie-Moss, Mel Shu, Greg Umali, and Alex Xu.

I would like to thank Leila Clark and Kate Frain for their assistance revising this paper, and for their patience with my creative comma usage. Kate also rescued me in my darkest hours with the

delivery of hot chocolate.

Ethics

Where there was potential for overlap between my role as an undergraduate course assistant for COS 445 and my development of this project, I have ensured that no hours were billed for work which I have described as a part of my independent work above. Thus, I am not being awarded both payment as a course assistant and credit as an enrolled student for the same actions. Though students completed assignments developed from and optionally provided feedback for these assignments while enrolled in COS 445, these interactions were overseen and approved by Professor Weinberg. This paper represents my own work in accordance with university regulations.