— 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 Princeton's undergraduate course in algorithmic game theory, "Economics and Computing" (COS 445), 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 would a usual theoretical problem set, including handout code for student self-evaluation and software autograders. 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. 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 healthy incentive structures where students do not try to hurt each others' performance. Each past offering of the course has rebuilt these assignments from scratch, and all associated instructor tooling, 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] In order to eliminate unnecessary annual replication of previous coursework development, improve course performance at scale, and improve the course experience for the students, I develop an extensible framework to improve these strategy design exercises.

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, in the computer science department, COS 445, "Economics and Computing", places emphasis on the connections between game-theoretic results and algorithms, software, and other aspects of computer science. This is represented in the course topics, which focus 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 contain, in addition to traditional proof-based exercises, programming-based exercises which connect course topics to prerequesite 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

[1] registrar data here
[2] More students complete COS 340 in a given year, but it is taught in both fall and spring semesters.
[3] https://www.cs.princeton.edu/ smattw/Teaching/cos445sp18.htm
[4] Figure: PD game -> Java iterface

in a less preferred outcome, and have acecss to the other player's play history 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, from semester to semester, 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 equilbirium, and based on their own priors over the strategies which will be designed by their peers. These strategies are evaluated on both on their design quality and on their empirical performance. While design quality is evaluated on a purely individual basis, empirical performance is measured by simulating the play of the collection of submitted student strategies. 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 strategy, and strategies are scored based off the average payoff they achieved. For the search advertisement game, all the student strategies are concurrently asked to bid on the same search terms, and are scored based on their revenue from ad clicks (where each bidder knows the click-through rate for a term prior to bidding on that term) less their expenditures on advertising. To create a more realistic simulation, students are graded solely 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 demanded considerable instructor resources be dedicated to building and grading software resoruces, 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

[5]https://reg-captiva.princeton.edu/chart/index.php?terminfo=1184&courseinfo=012095 requires CAS login

resources previously developed. Due to the limited tooling provided in past semesters for students to evaluate their submissions, many submissions did not compile or caused runtime exceptions, requiring considerable additional instructor resources.

The goal of this project is 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 of the same work. This was accomplished by designing a reuasble framework which embodies the commonality between strategy design exercises, allowing instructors creating new exercises to change only the necessary, assignment-specific details, and which will include more 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 qualitiative 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.

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

[6]including the author

[7]https://web.archive.org/web/20170916042726/https://www.cs.princeton.edu/ smattw/Teaching/cos445sp17.htm

[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.

Both problem sets were implemented and graded by Cyril Zhang[9], a graduate student TA for the course. Cyril developed interfaces, dummy implementations, and grader software for each strategy design exercise.[10] However, this existing implementation of grader software for strategy design assignments is not modular and has to be completely rewritten for each assignment, and was designed for grader use only. The student interfaces were documented, but the grader software was not documented for future re-use. Additionally, some of the scripts and notebooks used by Cyril 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 Cyril's code.

Previous 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 respositories 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. This was uploaded to the current course repository,[14] but was not reused.

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 implementations of strategy design exercises for Prisoner's Dilemma and another auction setting. The Java code consisted of student handouts of interfaces and some testing code, but the testing code was not designed to be automated for grading and was not modular: it tested manually-enterred

[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

strategies against each other and assignment-specific details (e.g. 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 filesharing 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 the implementation is in 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

[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/

assignments are graded differently, these resources were dismissed as a base for development of new resources for COS 445 at Princeton.

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 autograding systems are designed with an assumption that student submission are graded in a sandbox, isolated 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. However, existing resources for assignment submission and student feedback in use by Princeton are used in this assignment. I integrated some CS DropBox[27] functionality where relevant, and modelled the leaderboard structure after review of the leaderboard implementation used in other Princeton CS classes. Generally, however, 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 these assignments. Thus, 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 harmonizing requirement suggest that the product should consist of one extensible implementation of all the reusable components for software design exercises, developed 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, Matt and I designed and

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

I implemented a handout interface and tools to allow students to test their strategies against each other. 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 covered in previous courses.[28] I factored out the code which implemented the Gale-Shapley deferred acceptance algorithm for stable matchings from the code which amortized results from multiple randomized trials, and from agglomerated these results to output performance statistics for students. Thus, 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 trails 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 requires 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 our 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. And, 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. This ensures that, even after I have graduated, future course staff will have access to the project. Additionally, I documented the repository structure and each program's role in the repository, as documented later in this document.

[28] Kernighan and Pike chapter 4

Implementation

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 parametrized on user-input strategy names. Unfortunately, COS 445 does not have Python experience as a course prerequisite, only Java experience. Additionally, survey data from other course surveys suggests that 90% of students report themselves fluent in Java after the Princeton introductory sequence, compared to 30% for Python.[30]

Thus, the whole system under consideration is constrained: it must support student strategy implementations in Java. Given this constraint, and lacking any other language selection pressures, I elected to implement as much of the assignments as possible in Java, with limited usage of scripting languages such as make, Python, and bash. These scripting languages were used where Java would not suffice and only for tasks for which 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 an assignment specification formally through a well-documented interface. Furthermore, by constructing new instances of the student implementations between game rounds, instructors can programatically enforce that students do not employ learning strategies outside the parameters of the assignment.[31]

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,

[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.

nullable reference. This is in contrast to functional languages (including Ocaml and Rust), where parameters are immutable and not null 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 prerequesite 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]

In order to design my framework, I reviewed unmodularized code from a past strategy design exercise, the Spring 2017 auctions exercise. This is the only assignment to which I had access to the full implementation of the student handout (having taken the course) and the grader code (preserved by Professor Weinberg). The first piece of the assignment to be implemented, after the game design is complete, 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);

    // 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);
}
```

[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

This interface describes what students' strategies must do: they must, each day, bid on an auction of advertising slots which appear on a search results page (often called "sponsored search results"), and they value the word being auctioned each day at a different value. The strategy also has access the winning bids for each of the ten sponsored results from each past day, and knows if it won any auctions itself, and if so how much it paid. 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. Cyril also implemented a program used to grade the student-submitted strategies. This program had 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 the game being played and of the postprocessing of the 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 postprocessing, such as grade assignment and output formatting. The divide between game rounds and code which processes game results was present in Cyril'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 for the first assignment did minimal postprocessing, this distinction was quite useful when later developing the autograder and additional extensions.

During the development of this handout, however, the statically-typed nature of Java raised some difficulty. Cyril's 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 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 strategies to be evaluated at grading time, without having to manually list out all the student strategy names, and to easily modify this list 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`, 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. Then, it rewrites its input by finding lines which start with `//R` and end with `//`, and replaces each such line with, for each row of the CSV, the line rewritten by replacing each curly-enclosed key with the corresponding value from that row. 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 the configuration file: [37]

```
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);
}
```

[37] use figure

In the above code from the college applications simulator, `S`, `T`, and `W` are configuration parameters for the simulation which were documented in the student interface. For contrast, the following method also instantiates an instance of an instructor-provided strategy using reflection. This illustrates potential variation in data collection code as enabled by my modular design, and is used in the autograder described below. [38]

```
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. This testing code is not hard-coded to any implementation of the student strategy interface, but instead can be configured by the student 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`.

[38] todo: figure

The handout will rewrite the simulation code to simulate those strategies using `formatBase.py` and then compile and run the simulation.

The two most important components of each exercise are the handout code for the students and the autograder code for the instructors. However, in order to design the autograder, 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, created to satisfy the need for an incentive-correct grading rubric which could be implemented in an autograder. It is based on the following intuition: 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 $S_1 \dots S_{i-1}, S_{i+1} \dots S_n$. That is, 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. 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.

The grading strategy described above has the correct incentives: each student will attempt to maximize their chance of beating each instructor threshold. 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, which is the same behavior as 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. So students are not incentivized to play a lower-expectation strategy in order to increase their variance, or any other higher-order moment of their strategy. The only concern would be if a student 'played mean' to hurt instructor scores, but since instructor scores for a given student are calculated without the play of this student, students could not improve their own grade by doing so.

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.[39] The modified replacement-comparison grading scheme measures student scores $x_1 \dots x_n$

[39] 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.

for all students $S_1 \ldots S_n$ amortized across many trials, then for each instructor strategies I_j among $I_1 \ldots I_m$, measures their score y_j against all of $S_1 \ldots S_n$. That is, instructor strategies are assigned scores based on their performance against all the students, and 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 them, not just their peers. Thus, if a student 'played mean' and hurt instructor scores, they might perform better. However, among a course of over 100 students, the impact of any individual student on any instructor's score is assumed to be negligible. The strength of this assumption needs to be verified during the development of an assignment.

The goal of this project is not to build a handout, or an autograder, 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 in e.g. a college admissions simulation
- data processing methods, e.g. `oneEachTrial` which builds a list containing each configuration-specified strategy once[40] and `oneEachTrials` (not shown above) which runs `oneEachTrial` a large number of times and averages the results
- postprocessing methods, e.g. to output 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[41], and a corresponding method `withExtraTrials` which averages the above. These are very similar to the existing data processing methods.

[40] see figure above
[41] see figure above

- A new postprocessing method that computes grades for each student given how many instructor strategies each student strategy beat and a lookup table.[42]

```
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);

    // 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 autograder. Additionally, neither of these changes required alteration of the `runTrial` method, so minimal additional work (only filling in the instructor strategy names and scoring rubric) is necessary for grading an assignment once the handout has been made.

[42] make into figure/appendix

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 postprocess the results of those trials to provide low-latency feedback or for autograding, and hooking all of the above together a `Makefile` to configure, compile, and run the assignment simulation. [43] [44]

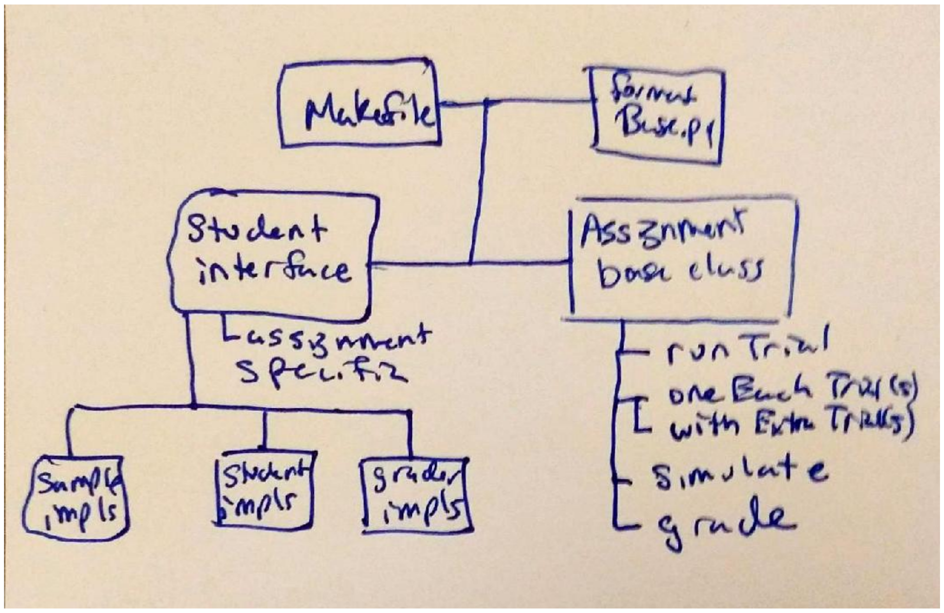

Figure 1: This is a gray image.

In addition to easily altering an assignment handout into an autograder, 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. The written assignment specification describes a new interface, so a whole new 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. And finally, the `runTrial` method of the simulator had to be rewritten to simulate an election instead of college admissions, and some file names had to be changed. No semantic changes were necessary to the data processing or postprocessing methods for either the handout or application.

[43]TODO redraw using Powerpoint
[44]TODO add formatBase.py config csv

As currently implemented, each assignment was distributed as a standalone zip file through the course website. Thus, even though significant code was shared between the two assignments described above, they were distributed as separate files. While further modularization would allow assignment-specific code describing how to simulate a game round to be separated out from code which changes within, but not between, assignments, I determined during assignment development that maintaining separate codebases for each assignment has significant advantages. Some simulation parameters, such as the budget constraint or number of auctioned items in an auction, the variance of student preferences in an admissions simulation, or the number of polling rounds in an election, 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 proved 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 comes also 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,[45] 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 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

[45] results section ref here

setting. All the script has to do is copy 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, 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, which 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 cronjob 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. Then, the code automatically fell back to silently overriding illegal values from student strategies with sane but selfless defaults, and 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, and this mostly resolved the issue.

The full implementation available at the course GitHub site.[46]

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

Results

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
- 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

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.

prisoner's dilemma + search auctions

Future Work

Some of the implementations of my infrastructure could be simplified using ClassLoaders, which would replace some of the Python code. This improvement was inspired by the implementation of ClassLoaders in the Spring 2014 student handout code.

would like to not use separate files even if lots of the code is the same - complicates handout but simplifies code reuse and iteration

Earlier assignments were built off of earlier versions of the infrastructure and will need to be ported to newer versions when they are reused: do next year

will be reused in two years without my oversight

other universities? matt through academic connections

Acknowledgements

* matt for assignment design lead * course staff, especially Andreea, Ariel, Evan, Heesu, Leila, Divya, Matheus, Wei, Dylan, for grading student writeups, providing me feedback and passing along student feedback * david walker for iw sem (helping me stay organized and on track, helping me identify my topic) * iw sem peers Greg, Alex, Mel, Noah for feedback * jeremie for creating github repo * kate for providing writing help and moral support

Ethics

* assignments used in the course were used by the executive decision of Professor Weinberg, * potential conflicts of interest * honor code