Proceedings of the Third AAAI Symposium on Educational Advances in Artificial Intelligence

An Undergraduate Course in the Intersection of Computer Science and Economics

Vincent Conitzer
Duke Unversity
Durham, NC 27708, USA
conitzer@cs.duke.edu

Abstract

In recent years, major research advances have taken place in the intersection of computer science and economics, but this material has so far been taught primarily at the graduate level. This paper describes a novel semester-long undergraduate-level course in the intersection of computer science and economics at Duke University, titled "CPS 173: Computational Microeconomics."

Introduction

In recent years, major research advances have taken place in the intersection of computer science (especially AI and theory) and economics (especially microeconomic theory). There are multiple motivations for these lines of work. On the one hand, as computer systems become increasingly interconnected, their users end up competing for scarce resources, necessarily introducing economic phenomena. On the other hand, advances in computing have made the use of various novel economic mechanisms possible. The multiagent systems community has played a prominent role in these developments, as it seeks to employ techniques from economics in the design of multiagent systems, as well as to contribute to the design of new economic systems by exporting techniques from AI and multiagent systems.

In this interdisciplinary research area, much of the focus has been on *game theory*—the theory of how to act rationally/strategically in environments with other players who strategically pursue their own objectives—and the closely related theory of *mechanism design*, which concerns the design of systems that result in good outcomes when used by such strategic agents. Example applications of the former include building good computer players for games such as poker (Sandholm 2010), but also the high-stakes "games" of strategically deploying security resources in ports and air travel, for which algorithms for computing game-theoretic solutions are now used in practice (Jain et al. 2010; An et al. 2011; Shieh et al. 2012). Example applications of the latter include the design of *combinatorial auctions* (Cramton, Shoham, and Steinberg 2006), which sell multiple interrelated items at once (for example, spectrum auctions), and *sponsored search auctions* (Lahaie et al. 2007), the auctions used by major search engines to sell advertising space next to their search results. Several articles on these topics that are accessible to a general computer science audience have recently appeared in the Communications of the ACM (Varian 2008; Shoham 2008; Conitzer 2010).

Various advanced graduate courses have been taught by the researchers in this area. In this paper, I argue that some of this research has become mature enough to teach to undergraduate students in a regular course. Specifically, this paper describes a novel undergraduate course that has now been taught twice (first as a "topics" course in Fall 2007, and then as a regular course in Spring 2010) at Duke University.[1]

The rest of this paper is organized as follows. We first discuss the main challenges faced in making the course accessible to students from a variety of backgrounds, and how these were addressed by focusing on using the GNU MathProg language to model problems as linear and integer programs. We then discuss the organization of the course, the material covered in it, and which computational problems this material raises that can be solved using linear and integer programming techniques. We proceed to discuss student evaluations of the course. We conclude with some discussion of the role of the course in the computer science curriculum.

Making the course accessible to students from different backgrounds

Before discussing the details of the material taught in the course, it is useful to discuss the key difficulty that had to be addressed in designing this course. An important goal of the course is to bring together students from computer science, economics, and other backgrounds.[2] This poses a major challenge for the programming assignments: they need to simultaneously

[1] I am currently (Spring 2012) teaching this course for the third time; evaluations for this iteration are not yet available. The most closely related undergraduate course of which I am aware is David Parkes' "Economics and Computation" course taught at Harvard in Fall 2011 (`http://www.seas.harvard.edu/courses/cs186/`).

[2] Commonly listed fields for registered students include Economics (18); Computer Science (13); Mathematics (6); and Electrical and Computer Engineering (5).

1. be accessible to students with little or no previous programming experience,
2. sufficiently challenge students who do have previous programming experience, and
3. cover key computational problems across the different topics in the course.

This combination of requirements would probably constitute an overconstrained system for many courses, but it turns out that for this course, there is in fact a solution. This solution is based on the insight that many of the relevant problems are naturally modeled as *linear programs* or *(mixed) integer linear programs*. These techniques are now used by many AI researchers, and in a few cases taught in AI courses. The rest of this section discusses the use of linear and (mixed) integer linear programs in this course in more detail.

Linear and integer programs

A linear program consists of a set of linear inequalities over a set of variables, together with a linear objective to be maximized (or minimized). In an integer linear program (resp., mixed integer linear program), all (resp., some) of these variables are not allowed to take fractional values. We will see an example shortly. Linear programs can be solved to optimality in polynomial time (Khachiyan 1979); (mixed) integer linear programs are NP-hard to solve, but in practice can often be solved reasonably fast using various techniques such as branch-and-bound search, cutting planes, etc. For more detail on the theory of linear and integer programs (and how to solve them), see, for example, (Vanderbei 2008; Nemhauser and Wolsey 1999).

Due to the broad applicability of linear and integer programs, various solvers are available. In this course, we use the GNU Linear Programming Kit (GLPK), primarily for its open-source nature. It is available at

`http://www.gnu.org/s/glpk/`

It is not the most powerful solver, and for student projects sometimes another solver is needed to scale; however, besides that, the solver is adequate, and it is not at all difficult to switch to using another solver.

In addition, special modeling languages are available. These modeling languages allow one to specify the linear or (mixed) integer linear program "in the abstract," and it can then be solved once concrete values for the parameters of the problem are specified. The modeling language associated with the GNU Linear Programming Kit is called MathProg; it is a subset of the AMPL language.

Let us now discuss an example.

Example problem in MathProg

The following problem instance of the knapsack variety illustrates the relevant concepts. (This problem instance is also discussed in the course.) Suppose we are in a room full of precious objects. We want to maximize the total value of the objects that we take out of the room. We can carry at most 30 kilograms out of the room, and we can also carry at most 20 liters out of the room.

- There are 3 units of object A available, each of which weighs 16kg, takes up 3 liters of volume, and is worth \$11.
- There are 4 units of object B available, each of which weighs 4kg, takes up 4 liters of volume, and is worth \$4.
- There is 1 unit of object C available, which weighs 6kg, takes up 3 liters of volume, and is worth \$9.

This problem instance is naturally expressed by the following integer program:

```
maximize 11x + 4y + 9z
subject to
x <= 3
y <= 4
z <= 1
16x + 4y + 6z <= 30
3x + 4y + 3z <= 20
x, y, z >=0, integer
```

Here, x, y, and z represent the numbers of units of objects A, B, and C that are taken, respectively. If the word "integer" were dropped, the result would be a linear program in which we would be allowed to take fractions of objects with us.

A downside of the above way of writing the integer program is that it is not very transparent how it should be changed if the instance changes—for example, if one of the parameters (such as the weights) changes, if a new object is added, or a new type of capacity constraint in addition to weight and volume is added (for example, the objects may be noisy and we may have a constraint on the total noise generated by the objects we take with us, so that we can escape undetected). In the MathProg language, we can represent the problem in the abstract first, and only then add the specific data of the particular instance to be solved. To model the example problem in the MathProg language, we first specify:

```
set OBJECT;
set CAPACITY_CONSTRAINT;
```

That is, we specify that there will be some set of objects and some set of capacity constraints (such as weight and volume), but we do not (yet) specify what they are. Next, we specify what types of parameters to expect:

```
param limit{j in CAPACITY_CONSTRAINT};
param availability{i in OBJECT};
param value{i in OBJECT};
param cost{i in OBJECT, j in
CAPACITY_CONSTRAINT};
```

These represent, respectively, the limit for each capacity constraint (for example, how much total weight or volume we can carry); how many of each object are available; how valuable each object is; and how much each object contributes to each capacity constraint (for example, how heavy is object A, or how much volume does item B take up).

Next, we specify the variables of the problem, for whose optimal values the solver will solve. For each object, there is a variable indicating how many units of that object we take with us.

```
var quantity{i in OBJECT}, integer, >= 0;
```

Having defined the parameters and the variables of the integer program, we can now define the objective and the constraints. These are the same as those given above, except they are now specified in the abstract. We wish to maximize the total value of the objects that we take:

```
maximize total_value:
sum{i in OBJECT} quantity[i]*value[i];
```

There is a constraint that we cannot take more units of an object than are available:

```
s.t. availability_constraints {i in
OBJECT}: quantity[i] <= availability[i];
```

Finally, we need to specify that we cannot exceed any capacity constraint.

```
s.t. capacity_constraints {j in
CAPACITY_CONSTRAINT}:  sum{i in OBJECT}
cost[i,j]*quantity[i] <= limit[j];
```

Of course, a solver cannot solve the problem "in the abstract"; we need to instantiate it with specific values. This is done as follows (using the same values as above):

```
data;
set OBJECT := a b c;
set CAPACITY_CONSTRAINT := weight volume;
param cost: weight volume :=
a 16 3
b 4 4
c 6 3;
param limit:= weight 30 volume 20;
param availability:= a 3 b 4 c 1;
param value:= a 11 b 4 c 9;
end;
```

While this may seem more complicated than the original way of writing the integer program, it is now much easier to change the particulars of the instance: only the data after `data;` needs to be changed.

Using MathProg for programming assignments

The course has two types of assignments: written assignments, and "programming" assignments using MathProg. ("Programming" is in quotation marks here because there are many aspects of programming that are not captured in MathProg modeling assignments; but, on the other hand, there are quite a few aspects that are.) Doing all the programming assignments in MathProg satisfies the three criteria layed out before:

1. For a mathematically well-prepared student, it is feasible to learn to do these assignments without any prior experience in programming. It should be pointed out that students with prior programming experience do have a significant advantage on these assignments; on the other hand, the students without programming background often have significant economics background, giving them a significant advantage on other assignments.
2. Students with previous programming experience—notably, computer science students—are still challenged by these assignments. This is in part because linear and integer programs do not receive much attention in most computer science curricula. Linear programs may be covered in algorithms courses; the techniques for solving (mixed) integer programs have much in common with techniques introduced in AI courses, but the connection is usually not made. In any case, computer science students tend to be unfamiliar with these modeling languages.
3. Many of the problems in the course can be tackled with these techniques, as discussed in more detail below.

Course organization

This section concerns the organization of the course.

Basic aspects

Prerequisites. In order to accommodate a broad audience, the course's prerequisites have been kept to a minimum. The main prerequisite is background in probability theory.

Materials. The book used for the course is *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, by Shoham and Leyton-Brown (Shoham and Leyton-Brown 2009). This book does a good job of covering the topics in the course, with the exception of some of the topics in Part 1 (see below). For those, separate reading assignments are given. Many research papers are also listed for optional reading, and various supplementary materials are made available for Part 0 (see below). All the slides for the course are publicly available on the course website at `http://www.cs.duke.edu/courses/spring10/cps173/`

Grading and assignments. The course is graded as follows: participation: 10%, programming assignments: 15%, written assignments: 15%, midterm: 15%, small project: 20%, final exam: 25%.

As the course progresses, the focus shifts from programming assignments to written assignments. The motivation for front-loading the programming assignments is two-fold:

1. Doing the programming assignments forces students to think very precisely about how to formulate problems as linear or integer programs, in particular about distinguishing between parameters and variables, whether to sum over a particular set or not, etc. After completing some of these programming assignments, they are much less likely to make conceptual mistakes on written assignments.
2. Especially for the students without programming background, it helps to have concrete experience using the optimization software before deciding on the projects they do for the course. Also, if students choose to use the GLPK software for their projects, then this will already get them to use it on an ongoing basis later in the course, lessening the benefit of making them do additional programming assignments at that point in the course.

Projects may be undertaken in teams of one, two, or three students. There are no hard requirements on the type of project, other than that they be related to the course material and, ideally, creative.

Technical content

Apart from an introductory lecture that gives an overview of the course, and a concluding lecture that discusses additional real-world applications, the course is organized into four main parts.

Part 0: Basic techniques. This part introduces linear and integer programming, focusing on getting students comfortable with the MathProg language and the GLPK solver. In class, we cover examples such as the one given above, and practice formulating various other problems (including the popular Sudoku puzzle) as linear or integer programs, as well as using GLPK to solve them. The students are also given their first programming assignment, in which, among other tasks, they must formulate a problem in the MathProg language by themselves. An extremely brief overview of basic techniques from theoretical computer science, including analyzing the runtime of algorithms and a basic introduction to the complexity of computational problems, is also given in this part of the course.

Part 1: Expressive marketplaces. This part introduces several types of marketplaces from the recent research literature that require nontrivial computational problems to be solved in order to run. The first example is that of a combinatorial auction, already mentioned in the introduction. In a combinatorial auction, multiple items are simultaneously for sale. A bidder can bid on individual items, but is also able to bid on a bundle of items. For example, a bidder can express: "I am willing to pay $500 for the desktop computer and the monitor together, but individually they are worthless to me." Once the auctioneer has collected all the bids, she faces a computationally hard *winner determination problem* (Lehmann, Müller, and Sandholm 2006): which bids should she accept in order to maximize the total value of the accepted bids, under the constraint that no item can be allocated to two different bidders? This problem can be naturally formulated as an integer program, as follows (in MathProg). In this formulation, `accepted[j]` is 1 if bid j is accepted, and 0 otherwise; `contains[i,j]` is 1 if bid j includes item i, and 0 otherwise; and `value[j]` is simply the value of bid j.

```
set BIDS;
set ITEMS;

var accepted{j in BIDS}, binary;
param contains{i in ITEMS, j in BIDS};
param value{j in BIDS};

maximize reward: sum{j in BIDS}
value[j]*accepted[j];

s.t. feasible{i in ITEMS}: sum{j in BIDS}
contains[i,j]*accepted[j] <= 1;
```

There are many variants of combinatorial auctions, such as combinatorial reverse (procurement) auctions and combinatorial exchanges (Sandholm et al. 2002), as well as combinatorial auctions that allow bidders to express their bids in richer *bidding languages* (Boutilier and Hoos 2001; Nisan 2006). To address these variants, various changes to the above integer program formulation must be made, which provides good interactive examples in class as well as tasks for a programming assignment.

We also discuss the idea of expressive financial marketplaces, in which complex securities can be traded (for example, an option that pays out if the sum of two stocks' values crosses a threshold). Here we face optimization problems similar to the winner determination problem discussed above—for example, when such securities are offered to us at various prices, which ones should we buy to maximize our guaranteed profit? Again, this can be cast as an integer program. This particular application tends to appeal to students interested in careers in finance (often economics students).

We then discuss *barter exchanges*, in which goods are traded without money changing hands. The main example here is that of a *kidney exchange*. The idea of a kidney exchange is as follows. Sometimes, a patient in need of a kidney has a willing donor, but they are incompatible. In this case, it may be possible to find another patient-donor pair in the same situation, such that patient 1 is compatible with donor 2 and patient 2 with donor 1, so that they can swap. More complex arrangements are also possible (e.g., a cycle of three patient-donor pairs). The basic optimization problem here is as follows: given the compatibility relations within a set of patient-donor pairs, find an arrangement that results in as many patients receiving compatible kidneys as possible. (Of course, a donor cannot be used unless the corresponding patient receives a kidney.) Integer program formulations for this problem are discussed in a paper by Abraham *et al.* (Abraham, Blum, and Sandholm 2007) that is on the reading list for the course.

Finally, we cover a small amount of voting theory in the course. Voting theory usually concerns the following setting: there is a set of alternatives, and each voter ranks these alternatives according to her preferences. Then, according to some rule, a winning alternative is chosen—or, more ambitiously, an aggregate ranking of all alternatives is generated. The latter has applications outside voting proper: for example, consider executing the same query on multiple search engines, and aggregating the results into a single ranking. Of particular interest here is the *Kemeny rule* (Kemeny 1959), which minimizes the number of disagreements between the aggregate ranking and the input rankings. There are several important justifications for using the Kemeny rule (Young and Levenglick 1978; Young 1995), but unfortunately its outcomes are NP-hard to compute even when only four rankings need to be aggregated (Dwork et al. 2001). Nevertheless, an integer program formulation can be quite effective (Conitzer, Davenport, and Kalagnanam 2006).

Part 2: Game theory. In many of the applications in Part 1, the participants would benefit from thinking strategically about what actions they take in the mechanism—for example, what to bid in a combinatorial auction. This is especially tricky because what is optimal for one bidder to bid can depend on what other bidders bid. *Game theory* concerns exactly this question: what does it mean to act

rationally/strategically in an environment with other rational/strategic players? Game theory is the topic of Part 2 of the course.

The main goal of this part of the course is to give students a basic background in game theory, which is also essential for understanding mechanism design in Part 3. Part 2 covers standard concepts from game theory, including representing games in normal and extensive form, as well as solution concepts (i.e., definitions of what it means to "solve" a game) such as (iterated) dominance, minimax strategies, Nash equilibrium, subgame perfect Nash equilibrium, Stackelberg strategies, etc. The presentation differs from a standard game theory course in that (1) the focus is on discrete models, and (2) for each solution concept we discuss how to compute solutions (which, in each case, can be done using linear or mixed integer linear programs). These computational problems are motivated by applications to computer poker and the strategic allocation of security resources, as discussed in the introduction.

Part 3: Mechanism design. Part 3 of the course concerns mechanism design. This part draws on both Parts 1 and 2, because it can be used to design the market mechanisms in Part 1 in such a way that they work well when the participants act strategically according to concepts from game theory, as explained in Part 2.

Part 3 starts by introducing Bayesian games, which can be used to model settings in which a player may not know another player's preferences exactly. We return to the topic of auctions (starting with single-item auctions), analyzing specific auction designs using the theory of Bayesian games. We then turn to the general theory of mechanism design, introducing Vickrey-Clarke-Groves mechanisms (Vickrey 1961; Clarke 1971; Groves 1973), which incentivize the participants to bid truthfully and which result in efficient allocations. To tie back into Part 1, we discuss in detail how these mechanisms can be applied to combinatorial auctions. (In the first, "topics" iteration of this course, we also managed to cover *automated* mechanism design, in which the problem of finding an optimal mechanism is itself cast as a linear or mixed integer program (Conitzer and Sandholm 2002); in the second iteration of the course, unfortunately, there was no time to cover this material.)

Course evaluation analysis

Below are the average scores on the student course evaluation criteria that seem to be the most relevant for the purpose of assessing whether the material covered is appropriate as a regular undergraduate course. (Omitted are the criteria that are more instructor-specific—with the exception of the overall instructor score—as well as progress criteria that were indicated as being less relevant for this particular course). For each criterion, the average score for the first, "topics" version of the course is listed first, and the score for the second version, with a regular course number, is listed second. The department's overall scores for the corresponding semesters are reported in parentheses. All scores are out of 5.0.

- *Overall: The quality of this course:* 4.8, 4.2 (4.0, 4.1)
- *Overall: The quality of the instruction (Inst. 1):* 4.8, 4.6 (4.0, 4.0)
- *Characteristics: Difficulty of the subject matter:* 3.8, 3.4 (3.7, 3.8)
- *Characteristics: Intellectual stimulation:* 4.7, 4.4 (3.9, 4.1)
- *Dynamics: Course requirements/expectations were clear:* 4.6, 4.4 (4.1, 3.9)
- *Dynamics: Methods of evaluating student work were fair and appropriate:* 4.8, 4.8 (4.1, 3.9)
- *Progress: Understanding fundamental concepts and principles:* 4.8, 4.6 (4.1, 4.2)
- *Progress: Learning to apply knowledge, concepts, principles, or theories to a specific situation or problem:* 4.8, 4.8 (4.1, 4.2)

The course quality score especially dropped noticeably upon moving from a "topics" course to a regular course. While it is hard to establish conclusively what the causes of this drop were, from the comments on the evaluation forms, it does seem that the students had higher expectations the second time. In particular, in the "comments" sections, students made more suggestions the second time for how the course could be improved. Notably, students with prior experience in game theory commented that it was good to take a class on its integration with computer science, but suggested shifting the focus further towards mechanism design and applications, to increase complementarity with existing courses in microeconomics that cover game theory.

Overall, though, the course evaluation scores seem to support the claim that the material is appropriate for a regular undergraduate course.

Conclusion

I believe that this course has some potential to bring new students into computer science and AI. A large number of new undergraduate majors should not be expected from it: as an advanced course, it tends to attract students in later years of the program that have already chosen a major. Also, of the students from other (non-CS) majors that take the course, many (but not all) already have some background in computer science. On the other hand, anecdotally, at least one of the course's students from another field (Economics+History) decided to take an introduction-to-programming course in a later semester. It is also worth noting that the course attracts mathematically talented students who often have an interest in graduate school, which may provide them with further opportunities to study computer science and AI.

Overall, the course provides undergraduates a concrete opportunity to apply computational thinking to another field. In fact, in her seminal article on computational thinking (Wing 2006), Wing already notes the example of economics, pointing out that "computational game theory is changing the way economists think." It is my hope that this course not only proves that an undergraduate-level course in the intersection of computer science and economics makes

sense, but also provides guidance in the design of new undergraduate courses in which computational thinking is applied to other disciplines.

Acknowledgments

I gratefully acknowledge NSF Awards IIS-0812113, IIS-0953756, and CCF-1101659, as well as an Alfred P. Sloan fellowship, for support. I also thank my department and university for supporting me in developing this course.

References

Abraham, D.; Blum, A.; and Sandholm, T. 2007. Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, 295–304.

An, B.; Pita, J.; Shieh, E.; Tambe, M.; Kiekintveld, C.; and Marecki, J. 2011. GUARDS and PROTECT: Next generation applications of security games. *SIGecom Exchanges* 10(1):31–34.

Boutilier, C., and Hoos, H. 2001. Bidding languages for combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, 1211–1217.

Clarke, E. H. 1971. Multipart pricing of public goods. *Public Choice* 11:17–33.

Conitzer, V., and Sandholm, T. 2002. Complexity of mechanism design. In *Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 103–110.

Conitzer, V.; Davenport, A.; and Kalagnanam, J. 2006. Improved bounds for computing Kemeny rankings. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 620–626.

Conitzer, V. 2010. Making decisions based on the preferences of multiple agents. *Communications of the ACM* 53(3):84–94.

Cramton, P.; Shoham, Y.; and Steinberg, R. 2006. *Combinatorial Auctions*. MIT Press.

Dwork, C.; Kumar, R.; Naor, M.; and Sivakumar, D. 2001. Rank aggregation methods for the web. In *Proceedings of the 10th World Wide Web Conference*, 613–622.

Groves, T. 1973. Incentives in teams. *Econometrica* 41:617–631.

Jain, M.; Tsai, J.; Pita, J.; Kiekintveld, C.; Rathi, S.; Ordóñez, F.; and Tambe, M. 2010. Software assistants for randomized patrol planning for the LAX airport police and the Federal Air Marshals Service. *Interfaces* 40(4):267–290.

Kemeny, J. 1959. Mathematics without numbers. *Daedalus* 88:575–591.

Khachiyan, L. 1979. A polynomial algorithm in linear programming. *Soviet Math. Doklady* 20:191–194.

Lahaie, S.; Pennock, D. M.; Saberi, A.; and Vohra, R. V. 2007. Sponsored search auctions. In Nisan, N.; Roughgarden, T.; Tardos, E.; and Vazirani, V., eds., *Algorithmic Game Theory*. Cambridge University Press. chapter 28.

Lehmann, D.; Müller, R.; and Sandholm, T. 2006. The winner determination problem. In Cramton, P.; Shoham, Y.; and Steinberg, R., eds., *Combinatorial Auctions*. MIT Press. chapter 12, 297–317.

Nemhauser, G., and Wolsey, L. 1999. *Integer and Combinatorial Optimization*. John Wiley & Sons.

Nisan, N. 2006. Bidding languages for combinatorial auctions. In Cramton, P.; Shoham, Y.; and Steinberg, R., eds., *Combinatorial Auctions*. MIT Press. chapter 9, 215–231.

Sandholm, T.; Suri, S.; Gilpin, A.; and Levine, D. 2002. Winner determination in combinatorial auction generalizations. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 69–76.

Sandholm, T. 2010. The state of solving large incomplete-information games, and application to poker. *AI Magazine* 31(4):13–32. Special Issue on Algorithmic Game Theory.

Shieh, E.; An, B.; Yang, R.; Tambe, M.; Baldwin, C.; DiRenzo, J.; Maule, B.; and Meyer, G. 2012. PROTECT: A deployed game theoretic system to protect the ports of the United States. In *Proceedings of the Eleventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*.

Shoham, Y., and Leyton-Brown, K. 2009. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Shoham, Y. 2008. Computer science and game theory. *Communications of the ACM* 51(8):74–79.

Vanderbei, R. J. 2008. *Linear Programming: Foundations and Extensions*. Springer, third edition.

Varian, H. R. 2008. Designing the perfect auction. *Communications of the ACM* 51(8):9–11.

Vickrey, W. 1961. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance* 16:8–37.

Wing, J. M. 2006. Computational thinking. *Communications of the ACM* 49(3):33–35.

Young, H. P., and Levenglick, A. 1978. A consistent extension of Condorcet's election principle. *SIAM Journal of Applied Mathematics* 35(2):285–300.

Young, H. P. 1995. Optimal voting rules. *Journal of Economic Perspectives* 9(1):51–64.