Teaching Problem-Solving in Algorithms and AI

Lisa Torrey
St. Lawrence University
ltorrey@stlawu.edu

Abstract

This paper suggests some teaching strategies for Algorithms and AI courses. These courses can have a common goal of teaching complex problem-solving techniques. Based on my experience teaching undergraduates in a small liberal-arts college, the paper offers concrete ideas for working toward this goal. These ideas are supported by relevant studies in cognitive science and education. Together, they provide a plan for structuring lessons and assignments to help student become better problem-solvers.

Introduction

Computer science is a problem-solving discipline. Students of computer science learn to solve problems that technology creates and/or brings within our reach. Some courses in the curriculum focus more on technologies, such as programming languages and architectures, which have a naturally high rate of change. Other courses focus more on problem-solving, which is a relatively stable concept even in the face of rapid technological change.

In this sense, courses like Algorithms and Artificial Intelligence form the core of computer science. In the undergraduate curriculum, the main role of these courses is to give students a problem-solving toolbox. This is evident in the 2008 ACM report (ACM 2008). In the Algorithms section, the report calls for *"the ability to select algorithms appropriate to particular purposes and to apply them"* to problems. In the Intelligent Systems section, it calls for the ability *"to select and implement a suitable AI method"* for a problem.

Of course, there are other ways to view the role of AI in computer science. It is a field with philosophical and creative aspects as well as practical ones. However, in the context of general undergraduate education, surely one worthy goal of an AI course is to help students develop advanced problem-solving skills.

This paper discusses pedagogical strategies with that goal in mind. It suggests ways to structure lessons and assignments with deliberate attention to the development of student skills. I have used these strategies to teach undergraduates in a small liberal-arts college.

The practical realities of a small college pose several challenges for this type of project. One is that due to scheduling limitations, I teach an AI course only infrequently. Most of my examples are therefore from a more regularly-offered Algorithms course. I make efforts to point out parallel applications in AI, and because these courses do have similar roles, pedagogical ideas tend to transfer well between them.

Another challenge is that small class sizes make it impractical to conduct experiments with proper control groups and statistical comparisons. My experiences can give me a sense for what works, but this type of evidence is anecdotal. To compensate, I turn to published studies in cognitive science and education to provide more formal support for the ideas in this paper.

Teaching to the Problem

Courses like Algorithms and AI are typically organized around families of problem-solving techniques. Thus an Algorithms course typically has units on divide-and-conquer, greedy algorithms, etc. and an AI course typically has units on heuristic search, classification algorithms, etc. Most textbooks are also organized this way (Cormen et al. 2001; Russell and Norvig 2010). Few would argue against this overall structure at the level of a course or a textbook.

However, there are some convincing arguments against conducting *lessons* based on this structure. It is easy to imagine such a lesson. The instructor introduces a technique, shows how to solve a problem with it, and then asks students to repeat the procedure on assignments and exams.

Note that this familiar approach teaches students to *apply* and *implement*, but does not really teach them to *select*. It tends to tell students in advance which technique they should apply to a problem. These are extra cues they will not have when they encounter problems "in the wild" later on.

The process of selecting a problem-solving technique can be non-trivial. Michalewicz and Fogel (2004) provide a surprising example using a geometry problem from a fifth-grade math textbook. Taken out of the context of its chapter and given to adults in mathematical fields, most took more than an hour to identify the elementary methods required to solve it. It seems that application skills do not necessarily translate to selection skills. If a course does not demonstrate selection and ask students to practice it, then it bypasses an important aspect of problem-solving.

Pedagogical Strategies

In my recent problem-solving courses, I have given more attention to the selection process. The main idea of my approach is to reverse the lesson structure described above.

When I want to teach a new problem-solving technique, I begin by introducing a motivating problem. A good problem for this purpose will be easily grasped, but not easily solved by techniques that students already know, because the first thing I do with this problem is ask the students to try to solve it. During this process, we identify where the old techniques fall short. Then I return to a more traditional mode of instruction and introduce the new technique.

There are several goals to this approach. By starting with problems, it gives students the experience of tackling them without knowing in advance how they should be solved, and it makes the new technique immediately necessary and useful. By analyzing inadequate solutions, it demonstrates an important part of the selection process: determining when a technique is *not* ideal. By finishing with direct instruction, it maintains a level of guidance that I think is important for complex material.

Literature Support

My approach shares some common ground with a pedagogical movement called constructivist or problem-based learning (Wilson 1998). This paradigm is based on the claim that learning is constructed through experience, not acquired through instruction. In its purest forms, constructivist learning advocates for minimal instructor roles in the learning process. However, its critics point to evidence that some students learn better from direct instruction (Kirschner, Sweller, and Clark 2006). Since my approach retains an element of direct instruction, it is more problem-*first* than problem-*based*.

There is support for this strategy in cognitive and educational research. For example, Schwartz et al. (2005) conducted experiments with students who were studying psychological experiments on human memory. They asked some students to analyze and graph data to find patterns, while other students instead wrote a summary of an existing analysis. Both groups then attended a lecture on this material. They found that students who had done their own analysis were more likely to make correct predictions about new experiments. They also found that the lecture component was a necessary condition for this result.

Similarly, and more closely applicable to computer science, Schwartz and Martin (2004) conducted experiments in the context of a statistics course. They asked some students to invent their own procedure for solving a problem before attending a lecture and seeing worked examples. Other students received more traditional instruction, with lectures and examples only. They found that students who had done the invention activity were able to solve more varied problems later. It did not matter if students invented *correct* procedures during their activity; in fact, none of them did.

Examples

I have put the problem-first approach into practice in several units of my Algorithms course. One example is the unit on greedy algorithms. The motivating problem I have used here is *interval scheduling*: given a set of intervals, each with a start and finish time, select a maximal subset of non-overlapping intervals (see Figure 1.)

This problem works well for several reasons. It is easy to describe and comprehend, both mathematically and visually. Students intuitively suggest several greedy approaches, which sets the stage well for explaining the concept of a greedy algorithm. However, the approaches they suggest tend not to be the optimal one, which provides opportunities to analyze incorrect solutions, and makes clear the importance of doing so.

Students typically suggest the following heuristics: prefer intervals that begin earliest, are the shortest, or have the fewest conflicts. We construct counterexamples to show that all three are incorrect; this is easy for the first two and rather challenging for the last. Then either a student or I will suggest the correct heuristic, which is to prefer intervals that *end* earliest. We do a proof by induction to verify this solution. I can then discuss the greedy problem-solving technique in a more general sense.

In an AI course, a related approach could be taken in a unit on Bayesian probability. A good motivating problem would be a classic one on medical diagnosis: given the results of a disease test, the accuracy of the test, and the frequency of the disease, compute the probability that a patient actually has the disease. Students familiar with simple probability could suggest solutions, but would not be likely to arrive at the correct answer. Real or simulated data could be used to induce dissonance, which could then motivate Bayes' theorem and the correct solution.

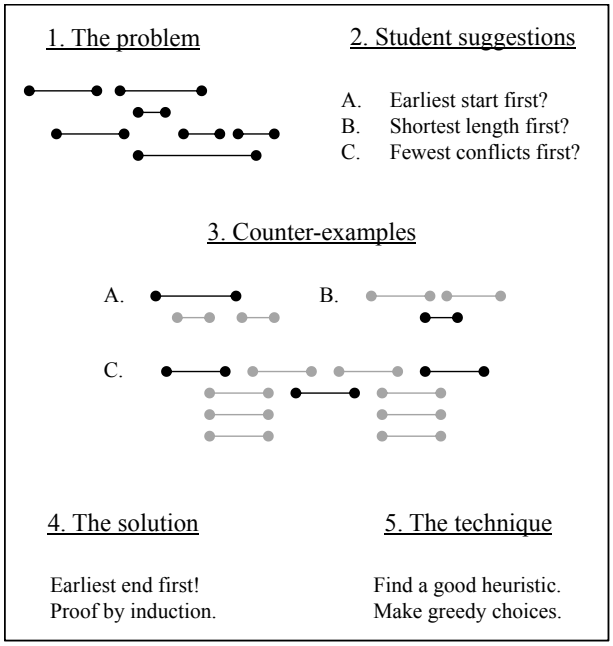

Figure 1: The interval-scheduling problem is to select a maximal set of non-overlapping intervals. This figure illustrates a problem-first lesson plan for introducing greedy algorithms using this problem.

Teaching for Transfer

Some of the problem-solving techniques taught in Algorithms and AI can be applied directly in many settings. Topics like sorting and depth-first search are straightforward in this way. Other techniques are more abstract: they provide a framework for approaching problems, but they must be tailored or adapted in some non-trivial way to be applied effectively in each new setting.

Topics like dynamic programming and heuristic search fall into this second category. A dynamic programming algorithm solves subproblems in the right order so that their solutions can be used in larger subproblems. Many problems can be effectively approached this way, but the specifics will differ in each case. Similarly, in heuristic search, a space is explored according to a heuristic that needs to be specified appropriately for a particular domain.

Particularly for this second category of techniques, instruction typically centers around examples. We demonstrate how to apply a strategy to some problems, and then ask students to apply it to others. Cognitively speaking, what this requires is *transfer of learning*.

Transfer has recently become a popular topic in AI research, so some of the ideas and terminology may be familiar. When we ask students to apply an algorithm in a way that is directly analogous to the examples they have seen, we are asking for *near transfer*. However, when an algorithm must be significantly adapted to a new problem, we may be asking for *far transfer*. Analogical reasoning may be useful in this case, but it will not be enough to solve the problem.

Research on mathematical problem-solving indicates that transfer of learning does not come easily to students. Novick and Holyoak ((Novick and Holyoak 1991)) found that it requires three stages: noticing that an old problem is relevant, determining its correspondence to the new problem, and adapting the old solution procedure to the new problem. Surprising numbers of students in their experiments struggled with all three stages, even in near-transfer activities, and even more so in far-transfer activities.

Using standardized tests, Novick and Holyoak found that mathematical ability was a good predictor of transfer success, but general analogical-reasoning ability was not. They also found that completing a near-transfer activity successfully helped students develop a better mental model (or *schema*) for the problem-solving technique, which in turn improved their success rate on a far-transfer activity.

There are several important implications of experiments like these. First, for many students, transfer is not something we can take for granted. It is the main goal of most instruction, but it is by no means a guaranteed result. Second, transfer can be improved through practice, which means that we can consciously design our courses to promote it.

Pedagogical Strategies

In my recent problem-solving courses, I have given more attention to practicing transfer. The main idea of my approach is to plan a sequence of examples, starting with near transfer and moving towards far transfer.

After showing students how to apply a new technique to an initial problem, I go on to present a second problem. A good problem for this purpose will be directly analogous to the first one, or nearly so. I ask students to attempt the new problem, and eventually we settle on a solution. Then I ask students to compare the two problems - to explain their common elements and also their differences. Based on this analysis, we develop a better description of the problem-solving technique. For more complex techniques, I repeat this process with at least one more problem that lies further away from the original.

There are several goals to this approach. By presenting a sequence of problems, it gives students guided practice with transfer. By comparing problems, it directs their attention to the common principles of the problem-solving technique, which presumably are the basis of a good schema. The mix of active learning and direct instruction follows logically from the previous section.

Literature Support

There is support for this strategy in cognitive and educational research. Gentner et al. (2003) conducted relevant experiments in the context of teaching negotiation strategies for conflict resolution. They gave all their subjects two examples to study, but one group was instructed to compare them while another group was instructed to study them one at a time. They found that the first group was more likely to transfer their learning to later problems, and could also articulate the key principles better.

One reason for this effect may be that comparisons help people distinguish between the core elements of a solution technique and the surface elements of a specific problem. Quilici and Mayer (1996) investigated this possibility in the context of teaching statistical tests. They gave one group of subjects examples that used a similar *cover story* for problems that required different test procedures, while another group received examples that used different stories. They found that the first group was more likely to choose the correct procedure for later problems, and less likely to be distracted by surface similarities between problems.

Another important effect may be to help students construct internal explanations of problems and techniques. Chi and Bassok (1989) confirmed the importance of *self-explanations* through their experiments with students in a physics course. They interviewed students while they studied mechanics examples and tried to solve related problems, and found that problem-solving success was correlated with good self-explanations of examples. Simply providing good explanations to the students did not have the same effect; they needed to construct their own.

Examples

I have put the problem-comparison approach into practice in several units of my Algorithms course. For example, in the unit on dynamic programming, I use a sequence based on the *knapsack problem*. This problem works well because it has many potential cover stories and two different structural variants (see Figure 2).

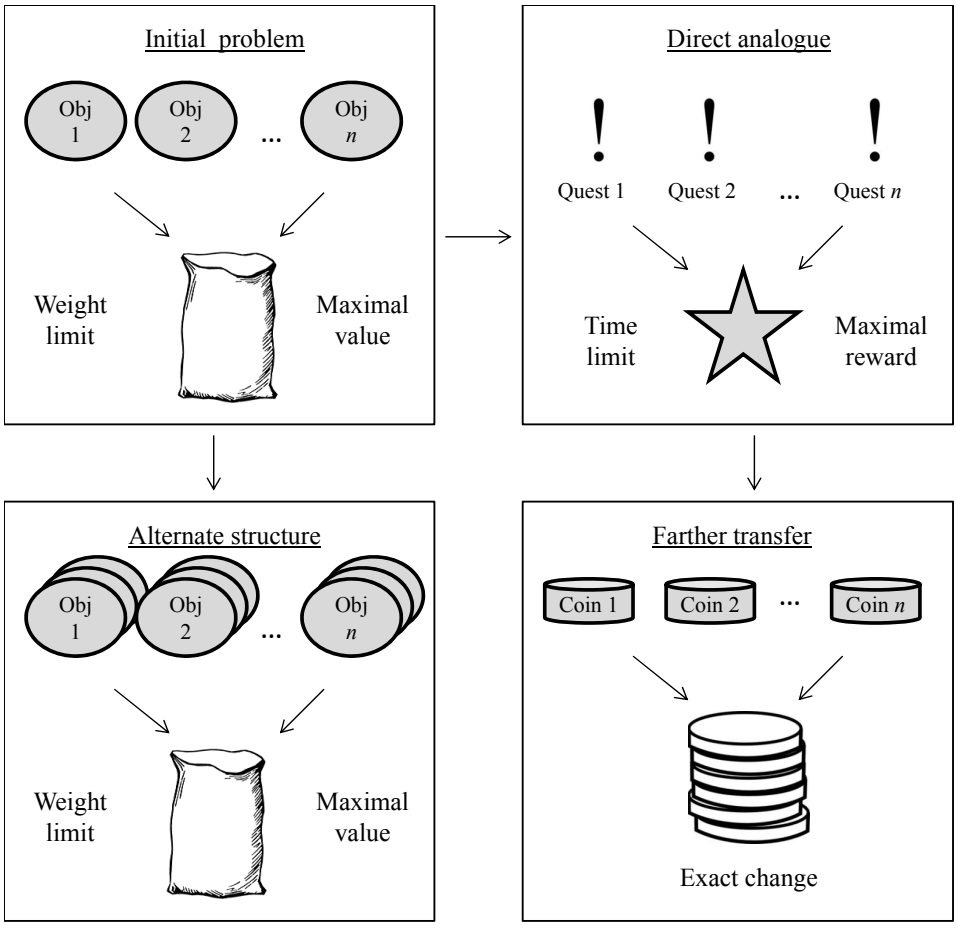

Figure 2: The knapsack problem is to put objects into a sack, maximizing total value within a weight limit. This figure illustrates related problems that provide transfer opportunities from the knapsack problem.

The standard form of the problem is to put objects in a knapsack, with the goal of maximizing the knapsack's total value within a weight limit. Objects can be available either with or without replacement, which produces the two structural variants. Dynamic programming can be used to solve both variants, but the solutions have non-trivial differences. (The former works with one dimension of subproblems while the latter works with two.)

One direct analogue of the knapsack problem is the *videogame problem*, in which one maximizes total quest rewards within a time limit. A little further away is the *coin-changing problem*, in which one makes a certain amount of change using as few coins as possible. Both of these examples have the same two variants, based on whether quests and coins are available in unlimited quantities.

These examples can be used to practice transfer in two directions: between cover stories and between structural variants. Comparisons across cover stories emphasize the important properties of the knapsack-problem family, while comparisons across variants reveal the important characteristics of dynamic programming solutions.

In an AI course, a related approach could be taken in a unit on genetic algorithms. Like dynamic programming, genetic algorithms can be applied to many problems, but effective solutions may differ substantially from one problem to the next. Students could practice specifying genetic representations, reproductive processes, and fitness functions for different domains. Comparisons across domains could give students a stronger basis for designing new genetic algorithms than any single example could provide.

Making Groups Work

After students have practiced problem-solving in a guided setting, the logical next step is independent practice, through problem sets or programming assignments. An important decision here is whether homework should be truly independent, or whether students should be allowed to collaborate. I believe that doing some work independently is important, but I support some forms of collaboration for particularly challenging assignments.

Multiple studies have suggested that cooperative learning can increase individual achievement (Johnson, Johnson, and Smith JulyAugust 1998). Within groups, students can learn from each other, and they may be forced to explain and monitor themselves to a greater degree than they would be otherwise. Groups also tend to perform at a higher level than

individuals (Heller and Hollabaugh 1992) and can therefore be given more challenging assignments.

However, it is clear that student collaboration also involves potential risks. A student may dominate the group and do the majority of the work, thereby short-changing the others. Alternatively, a student may fail to contribute sufficiently, leaving the majority of the work to others. In a more fair but still undesirable scenario, groups charged with working together can instead decide to split up the work.

All of these situations can mean that some students learn less from homework than was intended. They can also make it difficult to grade students fairly. Instructors are faced with deciding whether the benefits are worth the risks.

Pedagogical Strategies

In my recent courses, I have found two models for group work that have potential for achieving some of the benefits while avoiding many of the risks. They both provide students with specific guidelines for collaboration and include elements of individual responsibility. I believe these properties have been crucial to their success.

Pair programming is one form of group work that I have used. Obviously, this model involves two students working together on a programming project. The students are given two specific roles to play: *driver* and *navigator*. The driver is the one at the keyboard typing code, but the navigator is also actively involved - making suggestions, catching mistakes, and looking up information. Furthermore, the students are required to switch back and forth between these roles frequently.

For assignments that involve paper-based problem-solving rather than programming, I have used a different form of group work that I call *design sessions*. I present a problem in class and ask students to design a general solution approach. They do so in groups of three, which are composed differently every time. I check that groups are on the right track in a brief discussion with the class as a whole. Throughout this in-class session, students may only write high-level conceptual notes to themselves. For homework, I have them write up detailed solutions individually.

Although they differ in their details, these models have similar goals. By establishing an official collaboration, they aim to allow students to take on challenging tasks and learn from each other. By providing a high degree of structure and specific individual expectations, they try to help students avoid developing counterproductive habits.

Literature Support

Pair programming is an idea that originated in industry as the central practice of agile software development. In that setting, there is evidence that it can lead to better software design and fewer defects (Cockburn and Williams 2000). Over the past decade, pair programming has also gained advocates in educational settings. It has been used most frequently in introductory courses, where there is evidence that it can produce better programs and boost retention rates in the major (McDowell et al. 2006).

Pair programming has not been shown to have any reliable effects on individual exam scores in introductory courses (Brereton, Turner, and Kaur 2009). However, it is unclear to what degree the exams in these studies emphasized problem-solving tasks, as opposed to basic recall and application tasks. It is also unclear whether results in introductory courses will translate to more advanced ones like Algorithms and AI. There is room for more research to evaluate the effects of pair programming in these contexts.

Design sessions are largely my own invention, but they have elements in common with an activity called think-pair-share, an active-learning technique designed to increase student engagement in lectures (Lyman 1981). In think-pair-share, students are asked to consider a question individually and discuss it in pairs, and then some pairs are asked to share their thoughts with the class.

Design sessions could be considered an adaptation of this technique for complex problem-solving. There is support for using a group size of three: two students can be too few to generate ideas in the face of a complex problem, and more than three can be too many to allow everyone to participate (Heller and Hollabaugh 1992). The individual writeups are intended to ensure that all students fully engage with the problem, even though high-ability students are likely to take the lead during the in-class sessions.

Examples

I have used pair programming in a recent Software Development course, and I have used design sessions in a recent Algorithms course. Both were quite popular among the students; in both courses, students have made unsolicited comments expressing their enjoyment of these activities. My informal evaluation of the experience was that students sought my help less frequently, and submitted better work than I would have normally expected, given the challenging nature of the assignments.

I have learned two main guidelines for applying these activities effectively. First, it is important to use them only for tasks that are sufficiently challenging. For simpler assignments, students may not need to take the activity (and its rules) seriously. Second, it is important to describe the rules clearly to students, emphasizing the key features of the activities and explaining their purpose. Strict enforcement of the rules is rarely possible, but compliance is more likely when students fully understand the expectations.

Pair programming is clearly more applicable to programming-intensive courses like Software Development, while design sessions apply better to courses like Algorithms that involve more traditional problem-solving. AI courses are among the few that could naturally make use of both. Topics in the AI course are typically a mixture of abstract problem-solving and concrete implementation. Both of these models for student collaboration could therefore be applicable.

Last Day of Class

The final lesson of a course may be spent in a rush to finish material and complete administrative tasks. Or perhaps it includes a summary lecture, tying together all the course units into a coherent whole, or an inspirational speech about

the future of the field. All of these are worthy activities, but I have one final suggestion for making the most of the last day of class.

In my recent problem-solving courses, I have made time for the class to discuss some particularly realistic problems on the last day. Good problems for this purpose are not recognizably attached to any particular unit, and may not be perfectly solvable by any single technique. Ideally, they are real problems with some impact on the students' world.

I ask students to consider these problems in the context of the entire course. We list the types of techniques they have learned, discuss how they might apply, and sketch potential approaches. In essence, this is a large-scale selection activity, with motivations derived from the first section of this paper. It is also an engaging way to summarize a course.

Each year at my university, hundreds of first-year students are asked to identify and rank their top three choices of first-year seminars. They are then assigned into sections, taking these preferences into account as much as possible within capacity restrictions of the sections. I present this problem at the end of my Algorithms course, and it allows us to touch on nearly every unit in the course. By the end, we design several reasonable approaches using backtracking, local search, and network flow.

Conclusions

This paper discusses teaching in Algorithms and AI courses from the perspective of making students better problem-solvers. It makes concrete pedagogical suggestions for structuring lessons and assignments to focus on developing student skills. These suggestions spring from my own experience, but they are supported by studies in cognitive science and education.

I have proposed four strategies that fit together into an overall plan. The first is to introduce students to a problem before presenting its solution technique. The second is to help students practice performing transfer to similar problems. The third is to provide effective frameworks for student collaboration in challenging assignments. The fourth is to use real-world problems as summary activities.

Because these strategies operate at the level of individual lessons and assignments, they are applicable without any major course restructuring. They are not difficult to implement, though the active-learning elements do take up more class time than the traditional lecture-based approach. They are no doubt easiest to implement in small classes, but should not be impossible in large ones.

There is not yet a large body of research on teaching in courses like Algorithms and AI. Most educational research in computer science focuses on introductory programming. However, insights from other problem-solving disciplines can offer useful information. It is my hope that research specific to teaching upper-level computer science will be a growing area in the future.

References

ACM. 2008. Computer science curriculum 2008: Report from the interim review task force. Technical report.

Brereton, P.; Turner, M.; and Kaur, R. 2009. Pair programming as a teaching tool: a student review of empirical studies. In *Proceedings of the 22nd Conference on Software Engineering Education and Training*.

Chi, M., and Bassok, M. 1989. Learning from examples via self-explanations. In Resnick, L., ed., *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*. Routledge.

Cockburn, A., and Williams, L. 2000. The costs and benefits of pair programming. In *eXtreme Programming and Flexible Processes in Software Engineering*.

Cormen, T.; Leiserson, C.; Rivest, R.; and Stein, C. 2001. *Introduction to Algorithms*. MIT Press.

Gentner, D.; Loewenstein, J.; and Thompson, L. 2003. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology* 95(2):393–408.

Heller, P., and Hollabaugh, M. 1992. Teaching problem solving through cooperative grouping. *American Journal of Physics* 60(7):627–644.

Johnson, D.; Johnson, R.; and Smith, K. July/August 1998. Cooperative learning returns to college: What evidence is there that it works? *Change* 27–35.

Kirschner, P.; Sweller, J.; and Clark, R. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist* 41(2):75–86.

Lyman, F. 1981. The responsive classroom discussion: The inclusion of all students. In Anderson, A., ed., *Mainstreaming Digest*. University of Maryland.

McDowell, C.; Werner, L.; Bullock, H.; and Fernald, J. 2006. Pair programming improves student retention, confidence, and program quality. *Communications of the ACM* 49(8):90–95.

Michalewicz, Z., and Fogel, D. 2004. *How to Solve It: Modern Heuristics*. Springer.

Novick, L., and Holyoak, K. 1991. Mathematical problem solving by analogy. *Journal of Experimental Psychology* 17(3):389–415.

Quilici, J., and Mayer, R. 1996. Role of examples in how students learn to categorize statistics work problems. *Journal of Educational Psychology* 88:144–161.

Russell, S., and Norvig, P. 2010. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Schwartz, D., and Martin, T. 2004. Inventing to prepare for future learning: The hidden efficiency of encouraging original student production in statistics instruction. *Cognition and Instruction* 22(2):129–184.

Schwartz, D.; Bransford, J.; and Sears, D. 2005. Efficiency and innovation in transfer. In Mestre, J., ed., *Transfer of Learning from a Modern Multidisciplinary Perspective*. Information Age Publishing.

Wilson, B., ed. 1998. *Constructivist Learning Environments: Case Studies in Instructional Design*. Educational Technology Publications.