Focusing AI Students' Attention: A Framework-Based Approach To Guiding Impasse-Driven Learning

Steven Bogaerts and David Leake
Computer Science Department
Lindley Hall, Indiana University
Bloomington, IN 47405, U.S.A.
{*sbogaert, leake*}*@cs.indiana.edu*

Abstract

Research indicates that impasse-driven learning can have important benefits for improving student mastery of material. When students recognize gaps in their understanding of a concept, attempt self-explanations to resolve the impasses, and then receive further assistance from an instructor, the concept is more likely to be learned than without the impasse or self-explanation. This paper proposes that AI programming assignments be designed to encourage useful impasses and self-explanation, by (1) guiding student attention levels to increase students' scrutiny of areas important to pedagogical goals and 2) prompting fruitful expectation failures. It presents general guidelines for managing student attention levels when designing programming assignments, and describes how this strategy can be operationalized through the use of code *frameworks*. It illustrates the approach with concrete examples from sample AI assignments.

Introduction

Research on *impasse-driven learning* (Newell 1990; VanLehn, Jones, & Chi 1992) suggests that student learning may be greatly enhanced when the student must respond to an impasse. An *impasse* can be defined as any situation in which the student is unsure how to proceed in applying a concept, realizes a mistake has been made in applying a concept, or applies the concept correctly but is unsure if the application is correct (VanLehn, Siler, & Murray 2003). In contrast, an impasse has not occurred if the student correctly applies a concept (though perhaps only in a superficially or fortuitously correct way), never realizes a mistake has been made, or is never required to actually apply the concept.

Upon reaching an impasse, student *self-explanation* (Chi *et al.* 1989), even if the resulting explanations are potentially inaccurate, helps students to think more carefully about their knowledge and to form a more accurate model of the target concept (Chi *et al.* 1989; Pirolli & Bielaczyc 1989; Ferguson-Hessler & deJong 1990). After self-explanation, further instructor explanation does help (Renkl 2002), but this benefit comes primarily if self-explanation has already occurred (Anderson, Conrad, & Corbett 1989; McKendree 1990; VanLehn, Siler, & Murray 2003). The strong association between learning and impasses has been documented for many domains (Brown & VanLehn 1980; Carroll & Kay 1988; VanLehn 1987; 1990; 1999; VanLehn & Jones 1993; VanLehn, Jones, & Chi 1992). Thus, given the demonstrated advantages of impasse-driven learning, instructors should design course materials which encourage fruitful impasses. An important question for AI education is how best to do so within the constraints faced by instructors of AI courses.

Instructors of AI courses have the difficult task of designing short- and mid-length assignments that give students experience in the depth and power of often complex AI techniques. In programming assignments, the end goal is often viewed as to be able to build a working system, without necessarily focusing on guiding how students reach that end product. When students receive little guidance on how to focus their attention within an assignment, impasses may arise in a haphazard way, or may not occur at all, and the instructor has limited control over how and what the students learn along the way. This paper discusses a technique for instructors to design programming assignments to encourage students to focus on useful concepts and foster impasse-driven learning. It argues for a strategy involving (1) managing the *attention levels* of the students on various concepts in an assignment, and (2) using the attention levels to foster fruitful impasses, by prompting students to form expectations based on their current understanding of key concepts and then leading them to confront surprising factors. In this way, gaps in important knowledge can be exposed (i.e., impasses can be reached), motivating students to search for an explanation to resolve the impasses. In particular, the scope of this paper is the use of frameworks (collections of reusable code designed to be adapted to perform a given task in multiple domains) as a valuable resource for giving instructors control over student attention in programming assignments.

The paper begins by defining three general attention levels and discussing how frameworks can assist the instructor in fostering appropriate levels of student attention to individual topics. It then provides specific examples of how this can be exploited in assignments designed to lead to useful impasses and opportunities for self-explanation.

Attention Levels on Subconcepts

A large-scale *concept* in a course can be divided into a number of *subconcepts*. For example, the concept of version-space learning (Mitchell 1997) can be divided into subcon-

cepts such as the basic algorithm (handling positive and negative examples), attribute representation schemes (e.g., flat or hierarchical), using partially learned hypotheses, and inductive bias. An instructor leading student learning of a concept must choose the more crucial *target* subconcepts on which students should focus, and the *non-target* subconcepts to be ignored or deemphasized, at least temporarily.

For each subconcept, the instructor may guide the students' attention level, with attention level falling into one of three general categories:

1. **None** - Students are either 1) not explicitly made aware that the subconcept is relevant, 2) presented with a simplified model of the concept in which the subconcept is truly not relevant, or 3) asked to simply take some subconcepts "on faith."
2. **Shallow** - Students are instructed to examine the subconcept's basic application, but finer details are not covered.
3. **Deep** - Students are instructed to consider the subconcept's many applications and facets in great detail.

Motivations for Managing Attention Levels

By managing attention levels, the instructor can guide student learning step by step, providing greater motivation along the way than by simply presenting the assignment as a whole. The instructor can begin with a *None* attention level to a subconcept, allowing students to get a broad understanding of the relevant top-level issues. When this broad understanding is reached, they may then be instructed to consider subconcepts in more detail, at the level of *Shallow*. For example, a comparative study of different strategies for a subconcept may suggest a new approach relevant to that particular subconcept. Furthermore, at this point students may feel they understand the subconcept, only to find surprising results (an impasse), indicating a need for further study. This impasse would result from the instructor not initially explaining every detail, but rather first letting students discover subtle gaps in their understanding. Students can be encouraged to try to explain the surprising results (self-explanation). This also provides students with motivation to advance to the *Deep* attention level, if the subconcept is judged important enough for the course. The instructor time commitment for this approach is similar to more typical instruction, except that it may take place as shorter discussions over multiple class periods, allowing students to consider the subconcepts on their own at each attention level.

In the version-space learning example above, the *None* attention level may be appropriate for an overview, in which a basic algorithm can be discussed and the importance of some subconcepts can be recognized, without worrying about finer details. The instructor might later choose the *Shallow* attention level for, for example, inductive bias—making students aware of the issue and having them understand the high-level details and tradeoffs of a few basic approaches. Finally, the instructor can lead the students to focus more deeply on a subconcept, perhaps to find surprising results—that is, to encounter an impasse, leading to encouragement to explain these results and focus attention more deeply.

The following sections discuss this in more detail and also propose the use of frameworks for the management of attention levels, showing how frameworks can assist the instructor in leading misunderstanding students to relevant impasses, providing opportunities for self-explanation.

Managing Attention Levels with Frameworks

In a programming assignment designed to focus on selected target subconcepts, students who develop all code from scratch will nevertheless need sufficient understanding of even non-target subconcepts to program a complete functional system. Furthermore, in our experience, students facing an assignment with limited time seldom focus on genericity and independence of components, resulting in systems in which components may have many entangling relationships. In such a situation, attention level is not easily managed by the instructor—the student's attention must be broad enough to cover most or all of the subconcepts all the time. *Frameworks* can be a valuable tool for avoiding this difficulty and enabling instructors to guide students' focus.

Attention Levels and Frameworks

A framework is a collection of reusable code designed to be extended and applied to a variety of domains (Johnson & Foote 1988). The use of frameworks in AI classes has previously been advocated as a way to enable students to address real large-scale problems, within the constraints of a course (Bogaerts & Leake 2005a); this paper proposes that frameworks can give the instructor the control over attention levels necessary to guide misunderstanding students to impasses, and allow them to self-explain and test their explanations.

A framework consists of several *components*, which correspond to one view of the logical chunks of the overall concept. An important advantage of using frameworks is the ability of the user to either completely ignore or closely examine the intricacies of any particular component—that is, to have any attention level. This is possible due to the framework property of *minimizing component dependencies*: the major components of a framework should have minimal assumptions about implementation details of other components. This not only makes component implementations generic and swappable, but also allows the user to consider one component in relative isolation from the details of the others. Thus the minimal component dependencies property of frameworks allows the attention level of any given component to be freely managed according to interests.

Attention Levels for Components In mapping subconcepts to components, the attention levels can be defined more specifically as follows:

1. **None** - Students may not even be aware that the component is in use, because it has been specified automatically by another component or serves as a default when no overriding component is provided.
2. **Shallow** - Students consider the high-level behavior of a component, but not its implementation details.
3. **Deep** - Students propose or implement a component extension with new functionality to address a given issue in

incomplete or insufficient code.

Decreased attention to some components allows the student to focus on other components and how they pertain to the specific academic question of interest.

Guiding Student Attention Levels with Components Instructors can take explicit steps to influence students' attention levels on various components, to satisfy course goals. To prompt the *None* attention level, instructors might:

- Provide a basic system implementation which uses the instructor's chosen component.
- Provide only in compiled format any components to which the instructor does not wish the students to attend.

To prompt the *Shallow* attention level, the instructor might:

- Instruct students to examine the high-level differences between components.
- Instruct students to select a component for use in their system and explain their choice.
- Instruct students to conduct a high-level analysis of system performance for each of several provided component implementations.
- Again, some components could optionally be provided only in compiled format, but rather than ignoring them as in *None*, students would be expected to make use of them and analyze the results, at a high level.

To prompt the *Deep* attention level, the instructor might:

- Start with any of the actions of the *Shallow* attention level. Instruct students to then hypothesize improvements and optimizations and provide intuitive justification.
- Instruct students to construct new component implementations to compare performance of their hypotheses.

Issues In Mappings Between Components and Subconcepts Ideally, a framework addressing a given concept will have a one-to-one mapping between components and target subconcepts. However, even in a well-designed framework, this may not always be the case, depending on the instructor's chosen breakdown of the concept into subconcepts. For example, the instructor might break down a concept such that each subconcept is a different algorithm, relevant to only part of a component, or to multiple components.

If the breakdown leads to subconcepts corresponding to only part of a component, then the irrelevant portion could be given a different attention level. If the breakdown leads to subconcepts spanning multiple components with newly-introduced dependencies, then each of these components must have a similar attention level in order to fully study the subconcepts. Thus, conclusions made about guiding attention levels given one-to-one component-subconcept mappings still apply, though they may be weakened depending on the amount of deviation from a one-to-one mapping.

Example: Guiding Impasse-Driven Learning About Case Base Maintenance

As discussed previously, instructor guidance of student attention levels can encourage useful impasses and concomitant self-explanation. To illustrate, two example assignments are provided below, covering aspects of case base maintenance in CBR, and version-space learning.

As a case study in the utility of frameworks to facilitate guidance of attention levels, we first focus on an example for teaching case-based reasoning (CBR). CBR is an AI problem-solving approach in which a system stores prior problem-solving experiences (cases) so that they can be applied and adapted to suggest a solution for a new problem. (Kolodner 1993) provides a textbook introduction. The CBR processing cycle starts from a problem description entered into the system. Similar past cases are retrieved and adapted to propose a solution to the problem. It is evaluated by some external process, with failures returned for additional adaptation, and successes stored as new cases to be used for future problem solving. IUCBRF (Bogaerts & Leake 2005b), a freely available open-source framework, written in Java, was used for the assignment.

Shallow Examination of Maintenance Policies

This assignment was developed by the authors and given to a graduate-level artificial intelligence class.

Assignment Description The initial assignment helps students gain experience in *case base maintenance*—the revision of case base contents or organization to improve system performance. As case base sizes increase and CBR systems receive long-term use, maintenance becomes an important concern. Consequently, maintenance has received considerable research attention (e.g., (Leake *et al.* 2001)).

The initial assignment involves a comparison of four maintenance policies:

1. *NullMaintenance* - New cases are never added, and cases are never deleted.
2. *AlwaysAddMaintenance* - New cases are always added, and cases are never deleted.
3. *UnusedRemovedMaintenance* - New cases are always added. Periodically, cases used "infrequently" according to a given measure may be removed.
4. *ThresholdBasedAdditionMaintenance* - A new case is added if its distance from the closest case in the case base surpasses a fixed threshold. Cases are never deleted.

This assignment targets each student's attention level for the maintenance component to be *Shallow*, with all other CBR components targeted to be *None*. This was facilitated by the IUCBRF framework's provision of minimally-dependent non-target components.

The instructor intentionally did not examine maintenance policies in full detail in advance. Students were instructed to first hypothesize performance tradeoffs between solution quality (which generally increases with more cases) and retrieval efficiency (which generally decreases with more cases), using each of the four policies. They were instructed to next run experiments using the experimentation facilities of IUCBRF with the University of California-Irvine *letter* dataset (Newman *et al.* 1998), and to analyze the results.

Initial Results and Explanations Most students formed similar hypothesizes: that *NullMaintenance* would never learn anything new, and thus over time solution quality would be inferior to the other two policies; that *AlwaysAddMaintenance* would provide good solution quality, but adding all new cases to the case base would gradually decrease overall efficiency due to retrieval costs (the "swamping utility problem"); that *UnusedRemovedMaintenance* might have slightly lower solution quality than *AlwaysAddMaintenance*, but over time would lead to much better retrieval efficiency, as unused cases would not be kept to slow the system down; and, finally, that *ThresholdBasedAdditionMaintenance* would improve solution quality as more cases are added, without the problems in retrieval efficiency of *AlwaysAddMaintenance*.

Reaching an Impasse The students' hypotheses are essentially correct, save for one crucial flaw, making the actual system performance quite different. Results for retrieval efficiency (that is, trends for retrieval times) were as expected. However, students were surprised to find that their experimental results showed no real improvement in solution quality for *UnusedRemovedMaintenance*, giving approximately the same results as *NullMaintenance*. *AlwaysAddMaintenance* and *ThresholdBasedAdditionMaintenance* did improve in solution quality to some degree, but again not as much as expected. Students, having thought they understood the material, reached an impasse upon finding that system performance did not match their expectations.

Reacting to the Impasse One way to respond to the impasse would be for the instructor to explain what went wrong. We would expect this to be somewhat effective, in that having reached the impasse should tend to make students more prepared to attend to an explanation than they were without the impasse, as shown by research discussed in the introduction. However—unless students attempted to explain themselves rather than waiting for the instructor (as might be expected from stronger students, but not weaker ones)—this approach would fail to take advantage of the benefits of self-explanation. Consequently, the assignment itself required students to explain the results and offer some kind of resolution for their surprises.

Most students hypothesized that the case space was already well-covered by the initial case base, i.e., that adding more cases was not necessary, therefore explaining the limited or nonexistent improvement in solution quality.

Interestingly, this explanation is itself incorrect. When this is revealed to students, they have a newfound motivating curiosity beyond what existed at the start of the assignment, when their assumption was that the results would be fairly obvious and fit hypotheses. This impasse and self-explanation were necessary for the students to learn more about case base maintenance. By focusing students' attention level on a single component without the need to consider other components (thanks to the use of a framework), the instructor allowed students to zero in on key points.

Although the students' explanations of the impasse were inaccurate, by forming their self-explanations students were forced to think more carefully about their understanding to come up with a logical conclusion—a process that has been shown to facilitate learning (VanLehn, Siler, & Murray 2003). At this point, the true issue was discussed in class, because instructor explanation can be helpful after a failed self-explanation (Renkl 2002). The following section explains the points raised in this discussion.

An alternative to discussing the explanation would be to guide additional individual student investigation in a followup assignment. This assignment would have a different attention level, revealing the true source of the results while illustrating a fundamental but easy-to-overlook case-based reasoning principle, as discussed below.

Deep Examination of Maintenance Policies

A possible followup assignment follows the same template as the initial one, but at the *Deep* attention level, with instructions to look more carefully at the implementations of the maintenance policies, hypothesize what is wrong, implement revised policies, and test them. For students who are still stuck after they determine experimentally that their hypotheses do not hold, the following hint should be sufficient: *A new case is simply a problem description and a conclusion formed from the case base. Could the conclusion ever be wrong?*

This question brings students to another impasse—their expectations and understandings of what is and is not important are challenged. In attempting to answer that question and implement a resolution, they learn that they must change the partially inaccurate model of understanding they have for case base maintenance.

The answer to the question is, *yes, a conclusion formed from the case base could be wrong, if the most similar prior cases do not propose appropriate solutions*. Here the problem is that CBR systems require feedback about the results they generate, but the provided maintenance policies all violate a key principle of case addition, that cases only be added after they have been verified to be correct. (Providing maintenance policies with this deficiency can be seen as prompting a useful impasse by providing code with a carefully selected bug to study and diagnose, a strategy which can apply even outside of the use of frameworks.) In a CBR system, this verification typically takes place by either trying things out in the "real world" or perhaps by running a verification algorithm (able to verify, but not find, correct solutions). In these particular experiments, the data set used provides the "real world" answer for comparison. None of the maintenance policies checked for the quality of the system's solution, so any new case that was added had a chance of containing an incorrect solution. This erroneous case could suggest erroneous solutions for additional problems. As a result, the solution quality does not necessarily increase, and in fact could decrease, given an increasing case base size—as happened for the selected dataset.

These sample assignments show how an instructor can aid students in forming these conclusions on their own, by managing attention levels, as facilitated by using and extending a code framework. What begins as an "obvious" analysis of maintenance policies leads to an impasse, an incorrect self-explanation, guidance by the instructor, and the oppor-

tunity to diagnose and correct deficiencies in maintenance policies (while also correcting deficiencies in student understanding). Through the use of a framework with minimal component dependencies, students were able to have a *None* attention level to every component but maintenance, allowing them to focus their work and target the key concept of the assignment. When their *Shallow* attention level leads to an impasse, they have greater motivation to determine what is wrong, and attempt to provide a self-explanation to do so. With the instructor's guidance, further work can be encouraged to correct the self-explanation and lead them to the *Deep* attention level on the maintenance component.

Example: Version-Space Learning

Consider another example of using a framework to guide student attention levels to encourage useful impasses and self-explanation. This time, the domain is version-space learning, as discussed briefly above. The issues and example domain discussed here are from (Mitchell 1997).

Initial Assignment Suppose a framework has been built for version-space learning, such that students can quickly build a basic system for a given domain. Students initially have a *Shallow* attention level on the basic algorithm, and *None* on the other subconcepts such as attribute representation, handling of partially learned hypotheses, and inductive bias. That is, students are aware of the basic algorithm and its behavior, but have not considered its implementation, nor some of the finer points of version-space learning. This focus of attention is possible through the framework enabling students not to focus on non-target subconcepts, via the minimal component dependencies property.

Consider the domain that asks the question, again from (Mitchell 1997), "Is it a good day to do water sports?", given attributes such as temperature, humidity, and wind speed. Students are presented with a set of targets for the system to learn. This set contains only targets that are within the defined hypothesis space–for example, conjunctions of attributes (e.g., "warm and moderately windy days are the only good water sports days). The assumption that the targets fall within a limited hypothesis space, rather than one that makes no assumptions on form, is the *inductive bias*, required for version-space learning to make the "inductive leap" to generalize from examples. Through the use of a framework, students are able to consider experimentally the issue of learnability without having to worry about all the implementation details of the system. Students run experiments with the system and observe that, given enough examples, version-space learning is able to learn all targets in this set.

Reaching and Reacting to an Impasse Students are then given an additional set of targets, not conforming to the original hypothesis space, (e.g., disjunctive targets such as "either warm days or hot and not humid days are good water sports days") and asked to speculate and experimentally check the learnability of these targets. If they answer that they are learnable, then they will be at an impasse upon finding otherwise. Many students, however, may recognize that they are not learnable given the current inductive bias. The instructor may then guide the students to pay closer attention (increase their attention level) on the issue of inductive bias, a topic on which they previously did not need to focus since a default bias was provided by the framework in the system implementation. Students might then suggest expanding the hypothesis space to contain all possible hypotheses (e.g., all arbitrary boolean functions over the attributes), i.e., to attempt to create an unbiased learner—and in so doing, reaching the *Deep* attention level for inductive bias.

At this point, students reach an impasse. They find to their surprise that a version-space learner is unable to generalize from the examples using this expanded hypothesis space. They can attempt to provide a self-explanation for this, followed by additional discussion by the instructor on the inability of an unbiased learner to generalize beyond the training examples. Students can then attempt a resolution, such as changing rather than eliminating the inductive bias.

Thus the instructor has used a framework to allow students to initially focus on the basic algorithm, without concerns about inductive bias. Then, activities guided by the instructor allowed a change in attention level to address finer points, bring about a useful impasse, and provide opportunities for self-explanation and class discussion.

Potential for Broader Use

There are several AI resources that can provide the kind of code framework useful for applying the strategies of this paper. The AIMA code repository (Russell & Norvig 2002) presents discussion and code for various AI topics. WEKA (Witten & Frank 2000) is a framework of machine learning algorithms, commonly used by researchers, industrial scientists, and educators, much in the same way as IUCBRF.

Some frameworks define domains, freeing the student from having to attend to domain implementation details. For example, (Hill & Alford 2004) describes a highly-configurable agent-based war environment to which broad categories of AI techniques can be applied. Other examples include the "Wumpus" world of avoiding dangers and seeking benefits ((Russell & Norvig 1995), credited to Michael Genesereth) and the Eden microworld (Paine 2000).

Some collections of applets for demonstrations of AI topics also exist, such as CISpace (Conati *et al.* 1999) and AIxploratorium (Greiner & Schaeffer 2001). They may be quite useful for the *None* and *Shallow* attention levels, but are more limiting for assignments requiring a *Deep* attention level. This is because these allow exploration within the confines of the applet functionality, but have limited possibilities for deep examination in the implementation of new components to be used in the applet.

Conclusion

Much research has demonstrated the benefits in learning that come with impasses and student attempts at self-explanation guided by an instructor, and some learning environments have been developed with this in mind (e.g., (Burke & Kass 1996)). This paper has argued that managing attention levels can assist instructors of artificial intelligence courses in bringing about these impasses and self-explanations, and has

presented strategies for operationalizing this general principle. In particular, this paper has shown that code frameworks can be a useful tool for instructors to manage student attention levels, and has presented concrete examples and guidelines for their use. The minimal component dependencies property of frameworks makes it possible for students to have widely varying attention levels on different components of a system while still enabling them to work with a complete system. It is this possibility that gives the instructor significant power to bring about useful impasses.

References

Anderson, J. R.; Conrad, F. G.; and Corbett, A. T. 1989. Skill acquisition and the LISP tutor. *Cognitive Science* 14(4):467–505.

Bogaerts, S., and Leake, D. 2005a. Increasing AI project effectiveness with reusable code frameworks: A case study using IUCBRF. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Bogaerts, S., and Leake, D. 2005b. IUCBRF: A framework for rapid and modular CBR system development. Technical Report TR 617, Computer Science Department, Indiana University, Bloomington, IN.

Brown, J. S., and VanLehn, K. 1980. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science* 4:379–426.

Burke, R., and Kass, A. 1996. Retrieving stories for case-based teaching. In Leake, D., ed., *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI Press. 93–109.

Carroll, J. M., and Kay, D. S. 1988. Prompting, feedback, and error correction in the design of a scenario machine. *International Journal of Man-Machine Studies* 28:11–27.

Chi, M.; Bassok, M.; Lewis, M.; Reimann, P.; and Glaser, R. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science* 13:145–182.

Conati, C.; Gorniak, P.; Hoos, H.; Mackworth, A.; and Poole, D. 1999. CISpace: Tools for learning computational intelligence. Accessed October 22, 2004 at http://www.cs.ubc.ca/labs/lci/CIspace/.

Ferguson-Hessler, M. G. M., and deJong, T. 1990. Studying physics texts: Differences in study processes between good and poor solvers. *Cognition and Instruction* 7:41–54.

Greiner, R., and Schaeffer, J. 2001. The AIxploratorium: A vision for AI and the web. In *Proceedings of the International Joint Conference On Articial Intelligence Workshop On Effective Interactive AI Resources*.

Hill, J. M. D., and Alford, K. L. 2004. A distributed task environment for teaching artificial intelligence with agents. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 224–228. New York, NY, USA: ACM Press.

Johnson, R., and Foote, B. 1988. Designing reusable classes. *Journal of Object-Oriented Programming* 1(5):22–35.

Kolodner, J. 1993. *Case-Based Reasoning*. San Mateo, CA: Morgan Kaufmann.

Leake, D.; Smyth, B.; Wilson, D.; and Yang, Q., eds. 2001. *Maintaining Case-Based Reasoning Systems*. Blackwell. Special issue of *Computational Intelligence*, 17(2), 2001.

McKendree, J. 1990. Effective feedback content for tutoring complex skills. *Human Computer Interaction* 5:381–413.

Mitchell, T. 1997. *Machine Learning*. New York: McGraw Hill.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, MA: Harvard.

Newman, D.; Hettich, S.; Blake, C.; and Merz, C. 1998. UCI repository of machine learning databases.

Paine, J. 2000. Using Java and the web as a front-end to an agent-based artificial intelligence course. Available at http://www.j-paine.org/teaching_with_agents.html, accessed November 16, 2005.

Pirolli, P., and Bielaczyc, K. 1989. Empirical analyses of self-explanation and transfer in learning to program. In *Proceedings of the eleventh annual conference of the Cognitive Science Society*, 450–475.

Renkl, A. 2002. Worked-out examples: Instructional explanations support learning by self-explanations. *Learning and Instruction* 12(5):529–556.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.

Russell, S., and Norvig, P. 2002. Artifical intelligence: A modern approach. Accessed October 22, 2004 at http://aima.cs.berkeley.edu/.

VanLehn, K.; Jones, R.; and Chi, M. 1992. A model of the self-explanation effect. *The Journal of the Learning Sciences* 2(1):1–59.

VanLehn, K., and Jones, R. M. 1993. Learning by explaining examples to oneself: A computational model. In Chipman, S., and Meyrowitz, A., eds., *Cognitive Models of Complex Learning*. Boston, MA: Kluwer Academic. 25–82.

VanLehn, K.; Siler, S.; and Murray, C. 2003. Why do only some events cause learning during human tutoring? *Cognition and Instruction* 21(3):209–249.

VanLehn, K. 1987. Learning one subprocedure per lesson. In *Artificial Intellgience*, volume 31. 1–40.

VanLehn, K. 1990. *Mind Bugs: The Origins of Procedural Misconceptions*. MIT Press.

VanLehn, K. 1999. Rule learning events in the acquisition of a complex skill: An evaluation of cascade. *Journal of the Learning Sciences* 8(2):179–221.

Witten, I., and Frank, E. 2000. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann.