# Automatic Generation of Problems and Explanations for an Intelligent Algebra Tutor

Eleanor O'Rourke[1](✉), Eric Butler[2], Armando Díaz Tolentino[2], and Zoran Popović[2]

[1] Northwestern University, Evanston, IL, USA
eorourke@northwestern.edu
[2] University of Washington, Seattle, WA, USA
{edbutler,ajdt,zoran}@cs.washington.edu

**Abstract.** Intelligent tutors that emulate one-on-one tutoring with a human have been shown to effectively support student learning, but these systems are often challenging to build. Most methods for implementing tutors focus on generating intelligent explanations, rather than generating practice problems and problem progressions. In this work, we explore the possibility of using a single model of a learning domain to support the generation of both practice problems and intelligent explanations. In the domain of algebra, we show how problem generation can be supported by modeling if-then production rules in the logic programming language *answer set programming*. We also show how this model can be authored such that explanations can be generated directly from the rules, facilitating both worked examples and real-time feedback during independent problem-solving. We evaluate this approach through a proof-of-concept implementation and two formative user studies, showing that our generated content is of appropriate quality. We believe this approach to modeling learning domains has many exciting advantages.

**Keywords:** ITS · Problem generation · Answer set programming

## 1 Introduction

Over the past fifty years, researchers have developed robust artificial intelligence systems that can emulate one-on-one tutoring with a human [3,6,8,29]. These *intelligent tutoring systems* provide adaptive problem progressions and personalized feedback in a variety of domains, and have been shown to produce strong learning gains in classroom studies [10,30]. However, these systems are often challenging and time-consuming to build. Researchers have explored a variety of approaches for modeling learning domains, resulting in the development of cognitive tutors [3,6], constraint-based tutors [16,18], example-tracing tutors [9], and ASSISTments [8]. However, these approaches all focus on optimizing the modeling and authoring of intelligent feedback, rather than of problems and problem progressions, an area which has been highlighted as interesting for future development [11]. While a variety of problem-generation approaches have been developed and studied [14,19,26], most depend on models of the learning domain that are very different than those used to generate intelligent explanations, making it difficult to integrate them into existing tutoring systems.

In this work, we explore the possibility of using a single underlying model to generate both practice problems and intelligent explanations. We build on prior work in problem generation [2,5,19,28] by using the logic programming language *answer set programming* (ASP) to model if-then production rules similar to those used by cognitive tutors in the domain of algebra. We show how this model can be used to generate algebra problems and all valid solutions to those problems. We also present a new method for automatically generating step-by-step explanations directly from the ASP model. We show how our explanation content can be used to create worked examples, feedback during tutored problem-solving, and a progression that gradually fades between the two.

We evaluate our approach through a proof-of-concept implementation and two formative user studies. First, we developed an application called the *Algebra Notepad* that embeds the problems, solutions, and explanations generated from our model, demonstrating how our content can be used to implement an intelligent tutor. Next, we evaluated the application through two user studies. In a study with 57 Mechanical Turk workers, we found that participants solve problems more accurately and efficiently after practicing with our tutor, demonstrating that the generated solutions and explanations are understandable. In a study with seven eighth-grade students, we found that the tutor helps learners in our target population solve problems successfully. This approach for generating problems and explanations from a single domain model has many advantages, and could support robust content generation for tutoring systems in the future.

## 2 Background

### 2.1 Modeling Learning Domains

A variety of approaches for designing and implementing intelligent tutoring systems (ITS) have been explored. Cognitive tutors provide the most sophisticated form of intelligent feedback. They represent knowledge using production rules that define if-then relationships which capture all knowledge needed to solve problems in a target learning domain, allowing the computer to solve problems step-by-step along with the student [3,6]. Cognitive tutors detect errors when the student's action does not match any production rule in the model, and most include explicitly programmed "buggy" production rules that match common mistakes and misconceptions so that these can be explained [3]. Cognitive tutors are complex, and typically include as many as 500 production rules [4].

Constraint-based tutors are another type of system designed to help students identify and learn from mistakes [18]. These tutors model learning domains as sets of pedagogically important constraints [14,16,17]. Rather than tracing student actions, constraint-based tutors analyze the student's current state to identify violations of model constraints [17]. These tutors typically require less authoring effort than cognitive tutors, but can only provide feedback about constraint violations rather than also providing goal-oriented feedback [16].

Finally, *peudo tutors* exhibit many of the behaviors of ITS without requiring complex modeling. Example-tracing tutors are created by demonstrating correct solutions and common mistakes for specific types of problems. These demonstrations are used to create a behavior graph that can trace learner behavior and provide feedback [9]. A downside of this approach is that content must be demonstrated or authored for each problem type; in contrast, a cognitive tutor's production rules can generalize across many different problem types [9].

### 2.2 Explanation Generation

Most tutors use hand-authored templates to generate explanations. Cognitive tutors produce next-step hints and feedback by associating an explanation template with each correct and buggy production rule in the model [3,4,29]. To fill the templates with appropriate problem-specific content, each problem must also be labeled to indicate which phrases should be inserted into the template [4]. The Cognitive Tutor Authoring Tools (CTAT) were developed to support efficient authoring of both model-tracing cognitive tutors [12] and example-tracing tutors [9]. Example-tracing tutor authors can annotate hints and feedback messages for specific problems in the system [9]. In constraint-based tutors, hand-authored feedback messages are attached directly to the constraints [17], and can either be given after each student action or at the end of the problem [16]. In contrast to cognitive tutors, these explanations are problem-independent.

### 2.3 Problem Generation

Researchers have explored problem generation for a variety of domains including word problems [19], natural deduction [1], procedural problems [2], and embedded systems [23]. Most approaches are template-based; given a general template for a type of problem, they generate more problems that fit the template. These templates can be generated automatically [1], semi-automatically [2], or manually [23]. While many of these approaches use exhaustive search or logical reasoning to generate problems, others use logic programming languages to model domains and generate problems. For example, Andersen et al. use the code coverage toolkit Pex, built on the Z3 SMT solver, to generate problems for procedural mathematics (e.g., long division) [2]. Others use ASP directly for domains such as word problems [19] or educational math puzzles [5,28].

Despite this extensive research on problem generation, most intelligent tutoring systems still rely on hand-authored problems. In a recent paper discussing areas for ITS improvement, Koedinger et al. highlight automated problem generation as an interesting area for future development [11]. In the domain of constraint-based tutors, Martin and Mitrovic developed an algorithm that can generate problems from a set of target constraints [13,14]. However, since the domain models used to generate problems are different than those used to generate explanations, integrating problem generation into tutors is non-trivial.

## 3 Implementation Approach

In this work, we explore using a single underlying model to generate both problems and explanations for an intelligent tutor in the domain of algebra. Our core approach is to model valid algebraic operations using answer set programming (ASP), which facilitates generating both new problems and all valid solutions to those problems. During the modeling process, we structure the ASP program such that explanations for each solution step can be generated directly from the code itself. We note that if a tutoring system has access to all valid solutions to a target problem, it can trace a learner's steps and compare them to those in the solutions to detect errors. Furthermore, given step-by-step explanations of each solution, a tutor can provide worked examples and also use explanations of specific steps to provide feedback in response to learner mistakes. In this section, we first describe our approach for modeling algebra in ASP, then discuss how we generate problems, solutions, and explanations from this model. Finally, we describe how we can use the ASP program to automatically detect and explain a class of misconceptions related to applying algebraic operators incorrectly.

### 3.1 Modeling Algebra in ASP

ASP programs define facts and rules that are represented in first-order logic. Answer set solvers search the space of truth assignments for each logical statement in an ASP program to produce satisfying solutions called *answer sets*, which define a set of self-consistent statements that identify a valid state of the world. ASP programs typically include three types of rules: *choice rules* that allow the solver to guess facts that might be true, *deductive rules* that allow the solver to deduce new facts from established or guessed facts, and finally *integrity constraints* that forbid certain solutions.

To solve an algebraic equation, a learner must isolate a variable on one side by applying a sequence of operators, such as combining terms or dividing both sides by a constant. In ASP, we model operators using deductive rules and integrity constraints. Then, we use event calculus [25], a logical formulation that can represent the state of the world at multiple time steps, to model the problem-solving process. For each operator, we use deductive rules to define a set of precondition predicates that must hold for that operator to applicable at a given time step. Then, we use additional deductive rules to describe how the equation will change on the next time step if that operator is applied. For example, consider the operator for adding two like terms on the same side of the equation. This operator is only applicable when a set of preconditions hold: two terms must be on the same side of the equation, they must be added, and they must be monomial terms of the same degree. If these hold, the operator can be applied by adding the coefficients of the two monomial terms to produce a single term.

Most algebra problems have many valid solutions. In general, textbooks recommend first simplifying by canceling and combining terms, and then rearranging terms to isolate a variable on one side of the equation. We therefore group operators into five classes ordered by precedence – *cancel*, *combine*, *rearrange*, *move*, and *expand* – and define integrity constraints that force the program to explore the classes of operators in this priority order. Finally, integrity constraints are used to ensure that the final step is a valid solution.

## 3.2 Problem and Solution Generation

To generate new problems and their solutions from the ASP program, we define choice rules that set the initial problem configuration, the operators used in the solution, and predicates describing which operator is chosen at which time. Answer set solves also require that you define a finite search space, so we constrain both the size of the equations and the number of steps in the solution. In practice, novices focus on relatively simple problems, so we constrained generation to equations with a maximum of six terms per side and a maximum solution length of four steps. An answer set calculated on our ASP program produces both a problem and a sequence of valid operators that solves the problem.

This allows us to generate problem-solution pairs, but we want to generate all valid solutions to each problem. This requires some subtlety because generating all solutions is in a higher complexity class than generating a single solution. ASP is capable of solving this class of problem, and previous work has explored the specific challenge of generating math puzzles with all solutions [27], but implementing this type of model is technically challenging. Since we do not need to enforce any constraints over all problem solutions, we can take a simpler two-step approach. First, we generate a set of problem-solution pairs, and then we use a second ASP program to generate all shortest solutions for each problem.

## 3.3 Explanation Generation

Our approach for generating step-by-step explanations for each problem is to name the rules and predicates in the ASP program in such a way that explanations can be generated directly from the program itself. This allows us to produce explanations without having to write any problem- or solution-specific content, but requires structuring the ASP program differently than we would if we were not generating explanations from the code. Our explanations have two parts: we describe each operator that is applied to the equation, and we also provide strategy explanations that describe the priority of operator classes.

For operators, we first describe *why* an operator can be applied to the equation at this step, and then describe *how* to modify the equation to apply the operator. We generate explanation text from the declarative rules defined for each operator the ASP program. The precondition predicates in the operator rule define precisely why the operator can be applied, but we typically do not want to explain all predicates to the learner. For example, a predicate that states that a term cannot be added to itself is necessary for the solver, but not for the

**Algorithm 1.** ASP rules used to define the *add two terms* operator. The preconditions are separated into two tiers, one used to generate a high-level description of the precondition and a second used to provide a more detailed description. The second tier is also used to generate explanations for rules that *almost apply*.

```
applicable(T, weCanSimplifyByAddingTheseTwoTermsTogether(L, R))
    :-  weAreAddingTwoTermsWithVariablesThatHaveTheSameDegree(T, L, R).

weAreAddingTwoTermsWithVariablesThatHaveTheSameDegree(T, L, R)
    :-  _areDistinctNodes(L, R), isMonomial(T, L), isMonomial(T, R),
        areOnTheSameSideOfTheEquation(L, R),
        areBeingAdded(T, L, R), haveEqualDegrees(T, L, R),
        _isNotZero(T, L), _isNotZero(T, R).
```

learner. To handle these cases, we add an underscore to the beginning of predicate names that should not be explained. More importantly, we may want to describe multiple predicates through one high-level explanation. To handle these cases, we define our rules in ASP using a tiered approach that defines multiple levels of explanation detail for each operator. The first level produces a general explanation of why the rule applies, while the second level describes each of the predicates that must hold for the rule to apply. Algorithm 1 shows the ASP code that we used to model the *add like terms* operator in our system, which includes multiple rule definitions that provide different levels of explanation.

We also wanted to explain the problem-solving strategy of first simplifying the equation and then rearranging terms, which is represented through the priority of the five operator classes. We designed a dialog that presents these classes through a sequence of question-answer pairs, which start by asking whether an operator in each class can be applied (e.g. "can we combine?"). This question is answer either *no* ("no we cannot combine") or *yes* ("yes we can combine these terms"). In cases where an operator class can be applied multiple times, we note this in the response ("we can combine multiple terms"). This text is generated from five templates, one for the question and one for each possible response. The templates are populated with the current operator class and the ids of the terms to which the operator is applied. Figure 1 shows the sequence of explanations generated for applying the *combine like terms* operator to an example equation.

Our approach for automatically generating explanations requires authoring the ASP program with these explanations in mind, by abstracting predicates into multiple levels and naming the rules and predicates such that they will produce clear and understandable descriptions. While this requires significant up-front authoring effort, once the program is written explanations can be automatically generated for any problem and solution generated by the model.

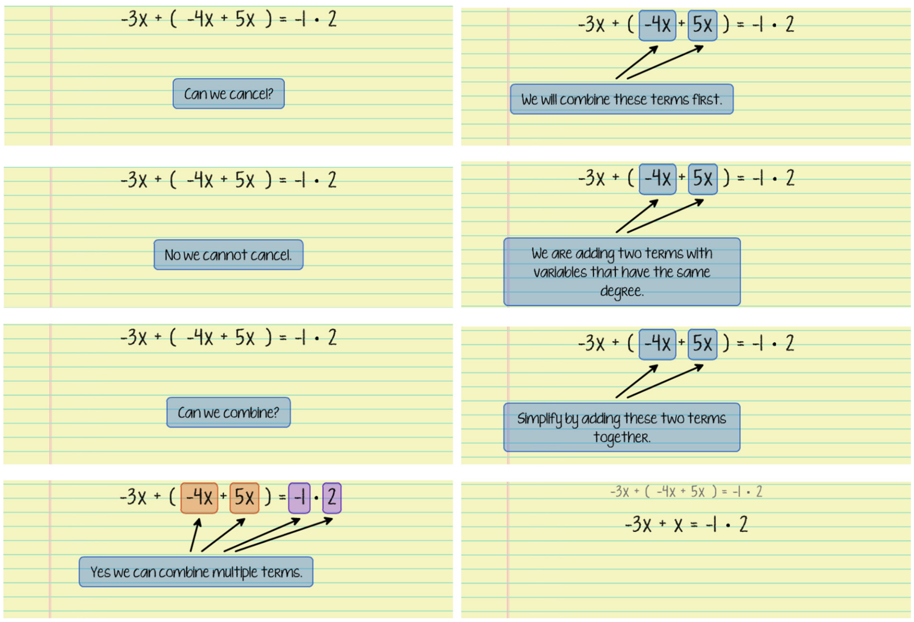
-3x + ( -4x + 5x ) = -1 • 2
Can we cancel?
We will combine these terms first.
No we cannot cancel.
We are adding two terms with variables that have the same degree.
Can we combine?
Simplify by adding these two terms together.
Yes we can combine multiple terms.
-3x + x = -1 • 2

**Fig. 1.** An explanation sequence generated for the *add like terms* operator that describes problem-solving strategy and how and why the operator can be applied.

## 3.4 Misapplied Rules

One potential benefit of this modeling approach is that it provides an opportunity to automatically generate rules that describe learner misconceptions. Many intelligent tutors detect and respond to common misconceptions, typically using hand-authored "buggy" production rules [3,17]. One class of misconceptions is misapplied rules. For example, given the equation $2x * 5x = 100$, a student could mistakenly think the add-like-terms operator applies to $2x$ and $5x$, since they are monomials of the same degree on the same side of the equation. However, they are being multiplied, not added, so the operation is not valid. Modeling algebraic operators in ASP allows us to automatically detect and reason about a subset of such misapplied rules. Each operator has several predicates which make up the precondition. If nearly all the predicates hold (e.g., there are two terms, both monomials of same degree) but one such predicate is missing (e.g., terms are not being added), then such a rule *almost applies*.

Given the set of predicates that define when an operator is applicable, we can produce an exhaustive list of all rules that almost apply for a given equation by searching for those that apply when a single predicate is omitted from the rule body. As with correctly applied operators, we can automatically generate explanations for operators that almost apply from the rules themselves. We take the name of the omitted predicate and negate it (e.g., "are being added" in Algorithm 1 becomes "are not being added"). The structure and consistency of our rule names makes this negation straightforward, placing "not" after the first "is" or "are" that appears in the rule name. To explain a rule that almost applies, the system generates the text "it looks like we can ⟨*operator name*⟩, but we cannot because ⟨*negated predicate that was omitted*⟩" using a template.

Traditionally, buggy production rules are hand-authored. In contrast, our modeling approach allows us to automatically detect a wide set of misapplied rules. While these only cover a subclass of potential misconceptions, they can be generated fully automatically. We hypothesize that, with data from learners, it would be possible to determine which of the generated misapplied rules are likely to occur in practice. Then, such rules could be used to preemptively explain common misconceptions or provide feedback in response to learner mistakes.

## 4 Formative Evaluation

The central contribution of this work is our approach for generating problems, solutions, and step-by-step explanations from a single model of a learning domain. To evaluate the content that can be produced using this approach, we first developed a proof-of-concept implementation in the domain of algebra. We modeled algebraic problem solving in ASP as described above, and then developed an interactive algebra tutor on top of the content generated by this model. We show that our tutor can provide step-by-step worked examples, can give real-time feedback during independent problem solving, and can support a problem progression that gradually fades between the two types of scaffolding. To evaluate the proof-of-concept tutor and its content, we conducted formative user studies with Mechanical Turk workers and eighth-grade students.

### 4.1 Proof-of-Concept Implementation

Many design decisions go into the development of any tutoring system [29]. Our goal in this work is not to study any particular instructional approach, but rather to show the variety of scaffolds that can be implemented with our generated content. We developed an interactive tutor we call the *Algebra Notepad* that uses a gesture-based interface to emulate solving equations on paper (see Fig. 2). The application uses problems, solutions, and explanations that were generated by our ASP model offline. We implemented a scaffolded problem progression that gradually fades between step-by-step demonstrations of correct solutions [7,15,20] and independent problem solving with mistake feedback [3,6,29], a pedagogical approach known as faded worked examples [21,22,24]. Our fading policy has five levels. In *Level 1*, learners walk through example solutions step-by-step, viewing our generated strategy and operator explanations (see Fig. 1). In *Level 5*, learners solve problems independently while the system compares their steps to the correct solutions. The system displays sparkly stars in response to correctly applied operators, and gives tiered feedback messages when steps are incorrect. The remaining three levels blend these extremes, for example showing explanations but requiring the learner to perform the operations on their own.

## 4.2 Mechanical Turk Study

First, we conducted study on Mechanical Turk with the goal of recruiting a relatively large number of users to try the *Algebra Notepad* application. We used this study to evaluate whether our scaffolds helped participants solve problems, and whether the generated explanations are understandable. We generated a static progression of nine problems for this study, six of which required a minimum of four steps to reach a solution and three of which required three steps. We used the fading policy described in the previous section; the progression started with one worked example at *Level 1*, followed by two problems each at *Level 2, 3, 4* and *5*. Participants took a three-problem test before and after using the *Algebra Notepad*, and completed a short survey about the experience at the very end.

We collected data from 57 Mechanical Turk workers, who provided informed consent and were paid for their time. First, we measured whether their problem-solving performance improved after practicing with the *Algebra Notepad*. A repeated measures ANOVA showed that participants performed better on the post-test ($F(1, 56) = 8.02$, $p < 0.01$), with a mean score of 2.4 out of three correct on the pre and 2.6 on the post. We also counted the number of steps used in correct solutions, and found that participants used fewer steps on the post-test ($F(1, 50) = 80.06$, $p < 0.0001$), with a mean of 5.1 steps per problem on the pre and 4.3 steps on the post. These findings suggest that practicing with the algebra tutor improved the correctness and efficiency of participant's solutions.

We also analyzed log data from the *Algebra Notepad* to measure how participants performed during independent problem solving (fading *Level 5*). We found that participants applied the correct operator on the first try in 84.5%

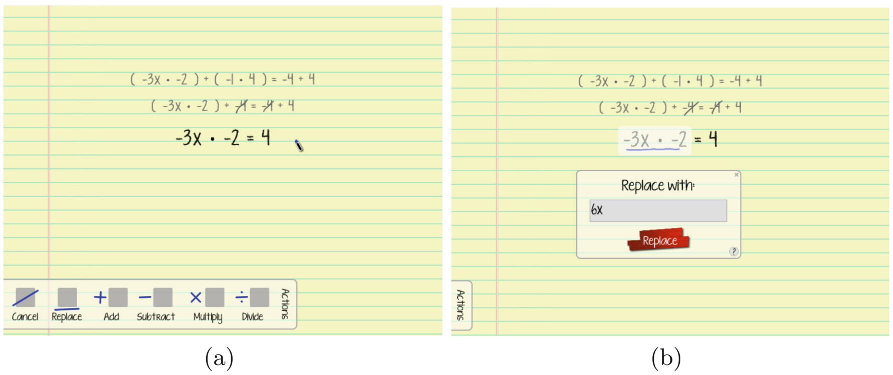

**Fig. 2.** Screenshots of the *Algebra Notepad* application. The interface displays each step on a separate line, and learners manipulate equations using gestures, as shown in the actions bar in (a). (b) shows a *replace* gesture.

of cases. Using tiered feedback, they applied the correct operator on the second or third try in 10.3% of cases, and only needed the system to perform the operation for them in 5.1% of cases. This shows that the generated feedback helped participants reach correct solutions most of the time. On the final survey, participants agreed that the explanations *"were clear and understandable"* and *"helped me solve problems"*, rating these statements an average of 4.8 and 4.7 on a six-point Likert scale respectively. When responding to a question asking what they thought of the explanations, one participant said *"I thought they were great. It has been years since I've done algebra and the explanations on the notepad refreshed by memory and improved my ability to solve problems correctly."*

### 4.3 Student Study

Since adults are not the target population of our *Algebra Notepad*, we conducted a second informal user study with seven eighth-grade students at a local Boys & Girls club to confirm that learners can successfully interact with the content generated with our model. In this study, students used the *Algebra Notepad* application to complete a static progression of 20 problems. The first nine problems in the progression were identical to those used in the Mechanical Turk study, and the same fading policy was used. However, we added an additional 11 problems at *Level 5*, where students worked on problems independently and were given feedback in response to mistakes as needed. The seven students all completed the progression of 20 problems. We analyzed their log data, and saw that for the 13 problems in *Level 5* that required independent problem-solving, students applied the correct operator on the first try in 81.4% of cases. The applied the correct operator on the second or third try, with the help of the tiered feedback, in 16.3% of cases. They only needed the system to apply the correct operator in 2.3% of cases. This suggests that most students were able to use the explanations and corrective feedback generated through our model to identify and apply the correct operators while solving algebra problems.

## 5 Discussion and Conclusion

In this work, we contribute a new approach for using a single underlying model of a learning domain to generate problems, step-by-step solutions, and explanations. We describe our process for modeling algebra in answer set programming, and show how the model can be used to generate new problems and all solutions to those problems. We also introduce a new method for automatically generating explanations directly from the model, and show how this content can be used to support step-by-step worked examples, feedback in response to mistakes during independent problem solving, and a progression that gradually fades between the two. We evaluated our approach by developing a proof-of-concept implementation of an intelligent tutor that uses content generated by our model, and we show that both adult users and eighth-grade students can interact with our explanations to successfully solve algebra problems.

We believe this modeling approach has exciting potential for supporting robust automated content generation for intelligent tutoring systems in the future. While this work focuses on the domain of algebra, ASP can be used to model any domain that can be represented through if-then relationships, where learners determine when rules or conditions apply and take actions in response. Logic programming languages have already been used to model diverse learning domains such as math procedures [2], word problems [19], and game puzzles [5,28]. While this work takes an important first step towards understanding how to construct an intelligent tutor around an ASP model, it has a number of limitations. We hope future research can continue this line of work, in particular expanding on our formative evaluation to determine whether content generated using this approach can effectively support student learning in real-world contexts.

## References

1. Ahmed, U.Z., Gulwani, S., Karkare, A.: Automatically generating problems and solutions for natural deduction. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI 2013, pp. 1968–1975. AAAI Press (2013). http://dl.acm.org/citation.cfm?id=2540128.2540411
2. Andersen, E., Gulwani, S., Popović, Z.: A trace-based framework for analyzing and synthesizing educational progressions. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2013, pp. 773–782. ACM, New York (2013). https://doi.org/10.1145/2470654.2470764
3. Anderson, J.R., Corbett, A.T., Koedinger, K.R., Pelletier, R.: Cognitive tutors: lessons learned. J. Learn. Sci. **4**(2), 167–207 (1995)
4. Anderson, J.R., Pelletier, R.: A development system for model-tracing tutors. In: Proceedings of the International Conference of the Learning Sciences, pp. 1–8 (1991)
5. Butler, E., Andersen, E., Smith, A.M., Gulwani, S., Popovic, Z.: Automatic game progression design through analysis of solution features (2015)
6. Corbett, A., Koedinger, K.R., Anderson, J.R.: Intelligent tutoring systems. In: Helander, M., Landauer, T.K., Prahu, P. (eds.) Handbook of Human-Computer Interaction, 2nd edn, pp. 849–874. Elsevier Science, Amsterdam (1997)
7. van Gog, T., Paas, F., van Merriënboer, J.J.: Process-oriented worked examples: improving transfer performance through enhanced understanding. Instr. Sci. **32**(1–2), 83–98 (2004). https://doi.org/10.1023/B:TRUC.0000021810.70784.b0
8. Heffernan, N.T., Heffernan, C.L.: The assistments ecosystem: building a platform that brings scientists and teachers together for minimally invasive research on human learning and teaching. Int. J. Artif. Intell. Educ. **24**(4), 470–497 (2014). https://doi.org/10.1007/s40593-014-0024-x
9. Koedinger, K.R., Aleven, V., Heffernan, N., McLaren, B., Hockenberry, M.: Opening the door to non-programmers: authoring intelligent tutor behavior by demonstration. In: Lester, J.C., Vicari, R.M., Paraguaçu, F. (eds.) ITS 2004. LNCS, vol. 3220, pp. 162–174. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30139-4_16
10. Koedinger, K.R., Anderson, J.R., Hadley, W.H., Mark, M.A.: Intelligent tutoring goes to school in the big city. Int. J. Artif. Intell. Educ. **8**, 30–43 (1997)

11. Koedinger, K.R., Brunskill, E., de Baker, R.S.J., McLaughlin, E.A., Stamper, J.C.: New potentials for data-driven intelligent tutoring system development and optimization. AI Mag. **34**(3), 27–41 (2013)
12. Koedinger, K.R., Heffernan, N.: Toward a rapid development environment for cognitive tutors. In: Proceedigns of the International Conference on Artificial Intelligence in Education, pp. 455–457. IOS Press (2003)
13. Martin, B., Mitrovic, A.: Automatic problem generation in constraint-based tutors. In: Cerri, S.A., Gouardères, G., Paraguaçu, F. (eds.) ITS 2002. LNCS, vol. 2363, pp. 388–398. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-47987-2_42
14. Martin, B.I.: Intelligent tutoring systems: the practical implementation of constraint-based modelling. Ph.D. thesis, University of Canterbury (2001)
15. McLaren, B.M., Lim, S.J., Koedinger, K.R.: When and how often should worked examples be given to students? New results and a summary of the current state of research. In: Love, B.C., McRae, K., Sloutsky, V.M. (eds.) Cognitive Science Society, pp. 2176–2181. Cognitive Science Society, Austin (2008)
16. Mitrovic, A.: Fifteen years of constraint-based tutors: what we have achieved and where we are going. User Model. User-Adapt. Interact. **22**(1–2), 39–72 (2012). https://doi.org/10.1007/s11257-011-9105-9
17. Mitrovic, A., Koedinger, K.R., Martin, B.: A comparative analysis of cognitive tutoring and constraint-based modeling. In: Brusilovsky, P., Corbett, A., de Rosis, F. (eds.) UM 2003. LNCS (LNAI), vol. 2702, pp. 313–322. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44963-9_42
18. Ohlsson, S.: Constraint-based student modeling. In: Greer, J.E., McCalla, G.I. (eds.) Student Modelling: The Key to Individualized Knowledge-Based Instruction, vol. 125, pp. 167–189. Springer, Heidelberg (1994). https://doi.org/10.1007/978-3-662-03037-0_7
19. Polozov, O., O'Rourke, E., Smith, A., Zettlemoyer, L., Gulwani, S., Popović, Z.: Personalized mathematical word problem generation. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence, IJCAI (2015)
20. Renkl, A.: Learning from worked-out examples: a study on individual differences. Cogn. Sci. **21**(1), 1–29 (1997)
21. Renkl, A., Atkinson, R.K., Große, C.S.: How fading worked solution steps works - a cognitive load perspective. Instr. Sci. **32**(1–2), 59–82 (2004). https://doi.org/10.1023/B:TRUC.0000021815.74806.f6
22. Renkl, A., Atkinson, R.K., Maier, U.H., Staley, R.: From example study to problem solving: smooth transitions help learning. J. Exp. Educ. **70**(4), 293–315 (2002). http://www.jstor.org/stable/20152687
23. Sadigh, D., Seshia, S.A., Gupta, M.: Automating exercise generation: a step towards meeting the MOOC challenge for embedded systems. In: Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education, WESE 2012, pp. 2:1–2:8. ACM, New York (2013). https://doi.org/10.1145/2530544.2530546
24. Salden, R.J.C.M., Aleven, V.A.W.M.M., Renkl, A., Schwonke, R.: Worked examples and tutored problem solving: redundant or synergistic forms of support? Top. Cogn. Sci. **1**(1), 203–213 (2008). https://doi.org/10.1111/j.1756-8765.2008.01011.x
25. Shanahan, M.: The event calculus explained. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 409–430. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48317-9_17

26. Singh, R., Gulwani, S., Rajamani, S.: Automatically generating algebra problems. In: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (2012)
27. Smith, A., Butler, E., Popović, Z.: Quantifying over play: constraining undesirable solutions in puzzle design. In: Proceedings of the 8th International Conference on the Foundations of Digital Games (2013)
28. Smith, A.M., Andersen, E., Mateas, M., Popović, Z.: A case study of expressively constrainable level design automation tools for a puzzle game. In: FDG 2012: Proceedings of the Seventh International Conference on the Foundations of Digital Games. ACM, New York (2012)
29. VanLehn, K.: The behavior of tutoring systems. Int. J. Artif. Intell. Educ. **16**, 227–265 (2006)
30. Vanlehn, K., et al.: The Andes physics tutoring system: five years of evaluations. In: In Proceedings of the 12th International Conference on Artificial Intelligence in Education, pp. 678–685. IOS Press (2005)