A Survey of Automated Programming Hint Generation: The HINTS Framework

JESSICA MCBROOM, IRENA KOPRINSKA, and KALINA YACEF,
University of Sydney, School of Computer Science

Automated tutoring systems offer the flexibility and scalability necessary to facilitate the provision of high-quality and universally accessible programming education. To realise the potential of these systems, recent work has proposed a diverse range of techniques for automatically generating feedback in the form of hints to assist students with programming exercises. This article integrates these apparently disparate approaches into a coherent whole. Specifically, it emphasises that all hint techniques can be understood as a series of simpler components with similar properties. Using this insight, it presents a simple framework for describing such techniques, the Hint Iteration by Narrow-down and Transformation Steps framework, and surveys recent work in the context of this framework. Findings from this survey include that (1) hint techniques share similar properties, which can be used to visualise them together, (2) the individual steps of hint techniques should be considered when designing and evaluating hint systems, (3) more work is required to develop and improve evaluation methods, and (4) interesting relationships, such as the link between automated hints and data-driven evaluation, should be further investigated. Ultimately, this article aims to facilitate the development, extension, and comparison of automated programming hint techniques to maximise their educational potential.

CCS Concepts: • **Applied computing** → **Computer-assisted instruction**; • **Computing methodologies** → *Artificial intelligence;*

Additional Key Words and Phrases: Automated hint and feedback generation, computer science education, artificial intelligence in education, intelligent tutoring systems

ACM Reference format:
Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A Survey of Automated Programming Hint Generation: The HINTS Framework. *ACM Comput. Surv.* 54, 8, Article 172 (October 2021), 27 pages.
https://doi.org/10.1145/3469885

1 INTRODUCTION

Automated tutoring systems, which provide educational materials and feedback to students without direct teacher involvement, offer promising approaches to delivering scalable and high-quality programming education. One fundamental aspect of these systems is the provision of hints and

Irena Koprinska and Kalina Yacef contributed equally to this research.

Authors' addresses: J. McBroom, I. Koprinska, and K. Yacef, School of Computer Science, University of Sydney, NSW 2006, Australia; emails: jmcb6755@uni.sydney.edu.au, irena.koprinska@sydney.edu.au, kalina.yacef@sydney.edu.au.

0360-0300/2021/10-ART172 $15.00
https://doi.org/10.1145/3469885

guidance to students working on programming tasks. Specifically, automated hints can help students progress in their learning by providing instant and relevant feedback to correct their mistakes and help them advance through exercises. In recent years, numerous techniques for producing programming hints have been developed, including approaches aimed at scaling up instructor feedback [14, 15, 26], extracting patterns from peer data [22, 23, 35], identifying particular dynamic or static issues with student programs [9], automatically generating personalised paths to solutions [11, 19, 41, 45], or combinations of these [38, 39].

While this diversity of approaches offers a great range of potential options for improving feedback, it also presents a difficult challenge—namely, it is difficult for instructors and researchers to decide which techniques are most effective for different situations. To address this issue, many studies have employed a range of different evaluation methods, including user studies and surveys [3, 11, 15, 23, 26], comparisons with experts [38, 41], analysis using historical data [7, 9, 11, 14, 19, 22, 27, 35, 39, 45], or other technical evaluations [27]. Researchers have also conducted comparison studies on small subsets of techniques [38, 40]. However, the time-consuming nature of evaluations, complexity of hint techniques and the apparently disparate nature of hints produced by these techniques (e.g., for different purposes, programming languages, and students) precludes a comprehensive study of every technique to achieve unambiguous answers. In addition, this difficulty is exacerbated as more hint techniques and variations are developed. Overcoming these challenges requires a clearer theoretical perspective to draw these techniques together into a simple, coherent picture. Such a perspective could motivate more focused questions for empirical studies to investigate, facilitate the sharing of ideas across hint techniques and provide teachers and researchers with insight into the range of available approaches and their relationships with each other. As such, it would act as an important step toward discovering the most useful techniques for different situations, to maximise the effectiveness of programming hints.

As a step toward addressing these challenges, this article surveys and develops key theoretical ideas behind recent work from the last five years (2014–2018) on generating automated hints for programming exercises. Specifically, it draws together hint techniques into a common framework that is highly generic and modular, yet simple. In addition, it discusses recurring themes and investigates how these can inform future work. As such, it acts as a guide for understanding the important developments, challenges and future directions in the field of automated programming hints, with the ultimate goal of maximising the potential of automated tutoring systems.

This article is set out as follows. Section 2 discusses related reviews and surveys, with an emphasis on how this work contributes to the field. Section 3 defines the scope of this survey. Section 4 presents the **Hint Iteration by Narrow-down and Transformation Steps (HINTS)** framework for understanding hint generation techniques, then Section 5 surveys such techniques in the context of this framework. Finally, Section 6 concludes with a discussion of key insights and future directions in the field.

2 RELATED SURVEYS AND REVIEWS

Previous work on categorising automated programming feedback has generally used one of three criteria to distinguish between feedback classes:

(1) the *technique* used (i.e., how the feedback is produced). For example, **Markov Decision Processes (MDPs)**, Bug libraries, or test cases could be considered different techniques for producing hints.
(2) the *nature* of the feedback (i.e., the type of information it reveals to students). For example, the feedback may be directed toward different parts of the program (syntax, layout, output),

may be very specific or general, or may be targeted in different ways (e.g., toward mistakes or toward next steps to try).

(3) the *input* required (i.e., which data are used to produce the feedback). For example, the feedback may be produced using model solutions, peer data or test cases as input, and the format of these data could be different, too. For instance, programs could be input as **abstract syntax trees (ASTs)** or lines of code.

In Reference [24], a survey of adaptive programming feedback, feedback is categorised based on its nature. Specifically, it is divided into the classes "yes/no," "syntax," "semantic," "layout," and "quality," where"yes/no" feedback reveals whether work is correct, and the other classes reveal information about syntactic, semantic, layout, or quality (e.g., efficiency) issues, respectively. In contrast, in Reference [49], which reviews static analysis approaches to producing feedback for Java programs, tools are classified based on the input used. For example, tools are classified based on the number of files they accept (e.g., "Single File vs. Multi File Analysis") and the program representations they require as input (e.g., "Trees vs. Graphs," and "Source Code vs. Byte Code Analysis").

In Reference [21] (extended from Reference [20]), all three types are used separately to label programming tools. The tools are labelled based on the technique used (e.g., "Model Tracing," "Data Analysis," "Program Transformations"), the nature of the feedback (e.g., "Knowledge about Task Constraints," "Knowledge about Concepts," "Knowledge about Mistakes") and also separately based on their "adaptability" (i.e., the input required by the system, such as "Solution Templates," "Model Solutions," or "Test Data").

Since the purpose of these previous surveys has generally been to compare feedback tools, they focused on dividing feedback into general categories so these tools could be distinguished. In contrast, while our work also uses the technique to classify feedback and also makes reference to the input and nature of the feedback, our focus is not on assigning feedback techniques to categories, but instead on building one simple and integrated picture of these. Our work considers the individual components that comprise these techniques, and utilises these to draw out insights about the nature of hint generation. As such, it conducts a deeper exploration into the nature of programming hints.

Our work is situated more broadly in the area of automated programming tutors. Other reviews in this area have considered techniques for automated assessment (e.g., Reference [1] and Reference [17]), or approaches to tutoring that can be supported by AI techniques [25], such as "example," "simulation," or "dialogue"-based approaches. In addition, some work has focused on the general features of programming tutors, such as the programming languages taught, or primary and supplementary features [8]. While some of these reviews also reference automated programming feedback, it is generally brief or not the main focus.

Our work also relates to reviews on automated software repair and debugging. For example, [30] presents a bibliography of automated software repair techniques and [47] surveys algorithmic debugging strategies. While our work also reviews some techniques for debugging and repairing programs, the focus is on producing hints for students in an educational context, where different resources may be available to the system, such as data from peers or teachers.

3 SCOPE

The scope of this survey is recent (2014–2018) techniques for generating automated hints for programming exercises:

(1) We consider a *programming hint* to be any type of feedback that improves a student's knowledge of how to complete a programming exercise. For example, it may help them to identify

mistakes in their program, suggest potential ways to proceed, recommend concepts to revise or clarify the task requirements. However, it does not, for example, include feedback aimed at encouragement, emotional support or coding style.

(2) We define *automated hints* very broadly to be any hints where no human intervention is required between the time the hint is requested and the time the hint is given. As such, it can still include hints generated using historical peer data, pre-written teacher hints, or other resources produced by people, so long as human intervention is not required when the actual hint is produced. However, it does not include peer-to-peer hints or teacher hints written *after* the hint is requested.

Note that this article focuses on methods for *generating* hints. For this reason, work on hint timing (e.g., based on student emotion [4, 51]), restricting hint availability (e.g., to prevent hint overuse [2]), hint delivery (e.g., through conversational agents), or deciding which hint technique to use for a particular student (e.g., using student models) are beyond the scope of this article.

Note also that, while the focus of this article is automated hints and not automated grading, sometimes papers on automated grading are discussed if the grading technique could also be used to produce hints. For example, test cases can be used to grade student programs, but can also be used to give hints about the types of inputs the program is failing on.

We include all papers that fit this scope from major conferences and journals that frequently publish work on this topic: International Conference on Artificial Intelligence in Education, International Journal of Artificial Intelligence in Education, International Conference on Intelligent Tutoring Systems, and International Conference on Educational Data Mining and Journal of Educational Data Mining. Specifically, we include all full-length papers that focus on and sufficiently detail a hint generation technique in accordance with the above specifications. We also include a broad range of other interesting work within our scope found through general searches and backward/forward reference chaining. Note that if two papers detail the same technique (e.g., a conference and journal paper), then we include the more developed version.

4 THE HINTS FRAMEWORK FOR GENERATING PROGRAMMING HINTS

A diverse range of approaches to generating automated programming hints have been proposed in recent years. These approaches are based on a variety of ideas and techniques, including machine learning, the utilisation of peer or teacher data, debugging techniques, and other methods focused on diagnosing errors or discovering potential improvements in student programs. In addition, these techniques result in hints in various forms, including hints that highlight errors, direct student actions, recommend additional materials, or provide other forms of support.

To understand how this multitude and variety of hint techniques fit together, existing approaches based on general categories are limited, for the following reasons:

(1) categorisation approaches based on the general *technique* employed are problematic, because techniques are so readily combined. For example, even if two approaches were fundamentally different, then it would be possible to integrate the output from both, thereby producing a third approach that fit neither category. Indeed, difficulties with this type of categorisation have been reported in previous investigations such as [21], where around 28% of surveyed tools did not suit a particular technique category.

(2) categorisation approaches based on the *nature* of hints produced are also problematic, because hint type does not necessarily correspond to the technique used. For example, a hint to delete a line of code could be produced by identifying it in a library of common bugs, as a frequent action taken by peers or as a step toward a model solution—all different techniques. Conversely, even if a problematic line of code was identified in the same way, then the final

form of hints could be quite different: They could explicitly instruct the student to delete the line, recommend reading materials relevant to it or perhaps identify its rough vicinity to focus the student's attention. As such, vastly different hint techniques can produce hints of a similar nature, and similar hint techniques can produce hints of vastly different nature. For these reasons, the final form of hints is also not ideal for understanding hint generation techniques.

(3) categorisation approaches based on the *input* are also problematic, because the same input can be processed in many different ways to achieve vastly different results. For example, data from peers could be used to find common paths to a solution or mined to find common buggy patterns.

If these categories are not sufficient, then this suggests that hint techniques may be too complex to allow for easy categorisation when considered in their entirety. However, hint techniques are often comprised of many smaller steps that are each simpler than the entire technique. This prompts the following question:

Can we develop a simple framework to describe all hint techniques by considering the smaller steps of which they are comprised?

A framework describing how automated hints are produced would be an important step toward understanding how these techniques relate and differ from each other, finding ways to extend and improve them, and developing methodologies for evaluating and comparing them. As such, it would act as an important step toward realising the full potential of automated hint systems.

This section, and the following sections, argue that such a framework is indeed possible. Moreover, they demonstrate that recent hint approaches are built up from just two simple operations applied iteratively. Considering the diversity of techniques, this is both an important result and tool for understanding fundamental ideas behind automated programming hints.

Before presenting the framework, Section 4.1 introduces three examples of automated hint techniques and highlights key similarities in the processes that comprise them. This will act as motivation for the key ideas behind the HINTS framework for describing automated hint techniques in general, which will be presented in Section 4.2.

4.1 Themes of Hint Generation Techniques

To explore several key ideas relevant to hint techniques, we first begin with a discussion of three examples of such techniques. Though these do not exemplify all approaches, they are illustrative of their diversity, and the similarities they exhibit will act as a basis for the general framework discussed in the next section. The examples are as follows:

(MB) MistakeBrowser [15]. Using a database of program transformations learned from peer data, the system searches for a subset of these transformations that correct a student's program. If successful, it then presents the student with hints written in advance by teachers for this particular set (cluster) of transformations.

(SFL) Spectrum-based fault localisation [9]. Test cases are run against a student's program, comparing its output to the expected output on various inputs. After this, parts of the program (program spectra) that are used when the tests pass and fail are compared, and functions associated with failed tests are flagged and highlighted to the student as hints.

(SC) SourceCheck [41]. An incorrect student program is matched to the closest known solution based on a distance measure defined by the authors. The edits needed to convert the student's program to this solution are then presented as hints.

Table 1. Hint Levels for Each Hint Generation Technique

Example	Input	Intermediate Hint	Final Hint
MB	All teacher hints, all transformations, student program, correctness test	Transformations that correct student program	Most relevant teacher-written hint
SFL	Student program outputs, expected outputs	Passed/failed tests	Problematic functions
SC	All solutions, student program	Closest solution	Edits to solution

The first column shows the input to the system. This is then processed to produce a new intermediate hint (second column). Finally, the intermediate hint is processed to produce another final hint (third column).

While these examples all appear to be quite different, they all share an important similarity: they build up hints in *levels* of increasing complexity. That is, they begin with simple hints, which are then developed into more sophisticated hints through an iterative process. For example, in (SFL) the final hint (highlighting a particular function) is produced using the results of test cases, which themselves could act as hints. Indeed, test cases are commonly used in programming tutors [21]. Moreover, these test case results are produced by comparing the output of the student's program to the expected output and, again, these themselves can act as simple hints revealing information about the student's program or the task respectively to the student. In this way, the hint technique can be considered a series of smaller steps, producing increasingly sophisticated hints. The levels of hints for all examples are shown in Table 1.

Surprisingly, not only can each of these examples be divided into simpler steps, but also these steps are remarkably similar: In each case, they involve deriving hints by *narrowing down* a set of hints from an earlier level. For example, in (MB) the most appropriate teacher hint is derived by narrowing down the set of all teacher hints. In (SC), the closest solution is derived from the set of all solutions. In all cases, this *narrowing down* operation also follows a similar pattern: Hints are selected from the earlier set based on their relevance to the student's program and/or some quality criterion. For example, in (MB) the most appropriate transformations are selected from the set of all peer transformations based on whether (a) they can be applied to the student's program (relevance to the student's program) and (b) they are able to correct the program (quality). In (SFL) the important problematic functions are derived from the set of all functions based on their associations with passed and failed tests of the student's program (i.e., their relevance to the student program). In this way, the steps used to build up hints all follow a similar process. The steps for all examples are summarised in Table 2.

At this stage, the structural similarities between these three hint techniques may appear to be coincidental. However, this is not the case: They are, in fact, characteristic of recent hint techniques and can provide valuable insights into hint generation. These similarities will act as an essential basis for the framework describing automated hints presented in the next section.

Note that, in some of the examples shown in Table 2, the hint from an earlier level is not explicitly a set of hints, but is easily converted to one through simple *transformations*, which change the way the data are represented. For example, in (SFL) the student program is *transformed* into a set of functions. In (SC) the solution and student program are *transformed* into a set of smaller parts to find differences. In these cases, the transformations are trivial, but other techniques can involve more sophisticated transformations. As such, *transformation steps* are also an important part of hint techniques and will be discussed in more detail in the next section.

Table 2. The Steps Used to Derive Hints in All Three Technique Examples

Example	Hint Data	Description
MB	All peer transformations	Select transformations that (1) can be applied to the student program (*relevance*) and (2) can correct the student program (*quality*).
	All teacher hints	Select teacher hints that are attached to the transformations that correct the student program (*relevance*).
SFL	All student program output	(To determine incorrect output) select the output that differs from expected output (*quality*)
		(To determine correct output) select output that matches the expected output (*quality*)
	Student program	Select functions from the student program that are mostly associated with incorrect output (*quality*)
SC	All solutions	Select solution that is closest to the student program (*relevance*)
	Solution	(To discover code to add) select parts of solution that don't match the student program (*relevance*)
	Student Program	(To discover code to delete[a]) select parts of the student program that don't match the solution (*quality*)

Each step involves narrowing down a set of hints (hint data) from an earlier level. This is achieved by selecting hints from the set that (a) are relevant to the student's program and/or (b) satisfy a quality criterion.
[a]There are also hints to move code described in the article, which can considered deletions followed by insertions.

4.2 The HINTS Framework for Hint Generation

In the previous section, three examples of hint generation techniques were presented, with each able to be divided into a series of steps that produced hints in *levels*. These steps involved *narrowing down* and *transforming* hints from earlier levels. Here we present the HINTS framework, which is based on these ideas.

The HINTS framework, shown diagrammatically in Figure 1, is described as follows:

(1) The system begins with a pool of data as input, which we call *hint data*, since they are used to produce hints. Input hint data can include datasets produced by peers or teachers; features of the help-seeking student's program or submission history; or a correctness test, such as test cases, desired static program features or a model solution. While these input data are usually processed before being given to students as hints, note that they could potentially be used as a hint without further processing. For example, expected input/output from test cases could be given to students to help them understand the task requirements, and an entire dataset of teacher-written hints could be given to increase their awareness of common errors.

(2) Hint data from the pool can be processed to produce new hint data. This is achieved by applying one of two steps:

(a) a *narrow-down* step. This involves taking a set of hint data and selecting a subset of this data based on a relevance criterion and/or a quality criterion. The relevance criterion stipulates that the subset of data should be relevant to some feature of the student's program. The quality criterion, in contrast, stipulates that hints should be of high-enough quality (based on some measure). For example, in (MB) from Section 4.1, the set of all

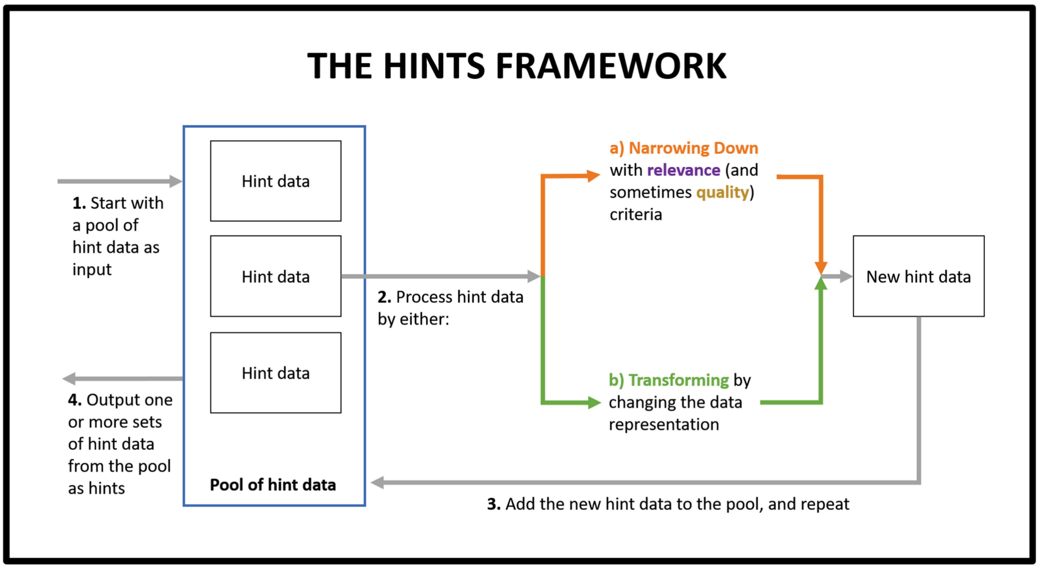
THE HINTS FRAMEWORK
Hint data
Hint data
Hint data
Pool of hint data
1. Start with a pool of hint data as input
2. Process hint data by either:
a) Narrowing Down with relevance (and sometimes quality) criteria
b) Transforming by changing the data representation
New hint data
3. Add the new hint data to the pool, and repeat
4. Output one or more sets of hint data from the pool as hints

Fig. 1. The HINTS framework for describing automated hint generation techniques. A small pool of hint data is given as input, and new hint data are produced by transforming or narrowing down existing hint data. Finally, some hint data are selected to be given as hints.

peer transformations was narrowed down to a subset based on the following criteria: (a) (relevance) that the transformations could be applied to (i.e., were relevant to) the student's program and (b) (quality) that they were able to produce a correct solution. As another example, consider a system that takes a large set of peer program states and narrows these down to only the most popular reachable state from the current student's program. Then, the relevance criterion would be that the state is reachable from the current student's state and the quality criterion would be that the next state is popular.

(b) a *transformation* step, which involves changing the way hint data are represented. This could include dividing the hint data into parts, representing it with a different data structure or converting it to a standard form. For example, a model solution to a programming task could be divided into a set of functions, represented with a different data structure, such as an AST [22], or converted to a canonical (standard) form through semantics-preserving transformations [11]. In addition, a set of student submissions could be divided into groups based on their output [35], or represented as a graph with nodes for programs and edges for transitions between them.

(3) Once new hint data are produced, they can be added to the pool of available hint data, and the proceeding two steps can be applied iteratively to produce increasingly complex hints.

(4) One or more *sets of hint data* from the final pool can be selected and offered to students as *hints*.

5 AN EXAMPLE-GUIDED SURVEY OF HINT METHODS

In the previous section, the HINTS framework was presented to describe automated programming hint techniques. This section now reviews recent work in the context of this framework, showing how hint techniques fit into HINTS, and also how they relate to each other. Specifically, this review progresses through a series of discussions, guided by example hint techniques from recent work. In each discussion, an example that is different from anything discussed so far is presented and related to the HINTS framework. Then, other work extending upon or relevant to this example is introduced and discussed. Finally, all techniques introduced in the discussion are related to previously introduced techniques, and the process is repeated.

In general, Section 5.1 discusses ideas on how a hint system can select next-steps from a pool of existing steps. Section 5.2 then extends upon this by discussing how next-steps can be automatically generated using a goal. Following this, Section 5.3 investigates the uses of program features, including how direct and indirect comparisons can be used to produce hints, and how features can be attached to pre-written teacher feedback. Finally, Section 5.4 discusses ideas for automatically repairing programs to produce hints.

Note that, following the discussion in Section 4, this review aims to combine the advantages of both general and step-based categorisations of hint techniques. Recall that general categorisations of hints can provide perspective on techniques as a whole, but are limited in that they do not capture the step-based nature of hint techniques. As such, this section uses general categories (i.e., the above discussion topics) to introduce hint techniques, but also breaks them down into individual steps using HINTS. This allows for many additional insights, such as a method for visualising all techniques together, which will be discussed in detail in the next section.

Also note that the examples in this section were chosen for their diversity. In particular, they came from different research groups, venues and years, varied in their complexity and implementation and were based on different ideas for producing hints. They were also chosen, because they were different from the three examples already discussed in Section 4. The discussion topics were chosen by organising the remaining literature into groups based on the most similar example technique, allowing for overlaps, and drawing out the key ideas of each group to discuss. As such, the topics were chosen to cover all of the surveyed literature, while providing a diverse range of examples to relate to HINTS.

5.1 Selecting Next Steps from Peer or Teacher Program States

A number of hint systems utilise existing data, such as peer actions or teacher solutions, to produce hints for help-seeking students. This survey begins by considering a subset of such systems, which convert existing data into program states, then select from these the most appropriate next state for the current student. While the selected next state may be given directly to students as hints, note that some systems, such as ITAP [45], involve additional steps before or after the next state is selected, and these further steps will be discussed in later sections.

5.1.1 Example: Hint Factory. Originally presented in the context of a logic tutor [48] and later adapted to the domain of programming [16, 35, 39], the Hint Factory is a technique that uses peer data and a MDP to produce next-step hints for students. To produce hints, program submissions made by peers are transformed into a state space where each state represents a particular class of programs, and edges connecting the states represent how students transition between them (thereby indicating paths to various solutions). A student's submission is then matched to a state in this state space, and the MDP is used to determine the best next state for that student. Figure 2 shows the general steps involved in the Hint Factory approach, and how these fit into the HINTS framework.

5.1.2 Discussion. One key decision when producing a hint system such as this is how to select the best next state. In the Hint Factory approach, an MDP was used, but other options have also been explored in recent work. These include finding the most common next-state of peers in a similar position [7] ("Code-based" technique), using a scoring function that accounts for popularity, correctness and distance from the student's current state [44], finding the path with the shortest

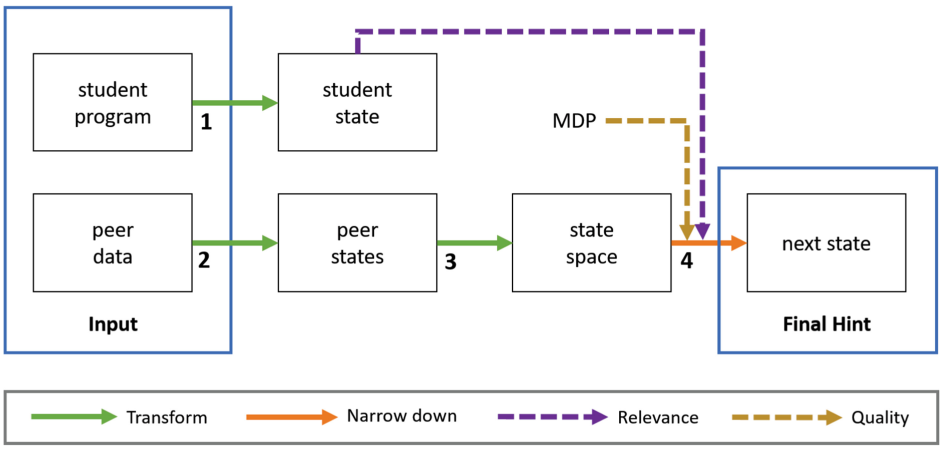

Fig. 2. A visualisation of the Hint Factory technique, using the HINTS framework (the colours correspond to the colours in Figure 1). Input: A student's program and peer data (i.e., programs submitted by peers). Output hint: The best next state for the student's program. Processes:

(1) The student's program is *transformed* into some general state.
(2) Similarly to (1), peer programs are *transformed* into states. (Note that, for efficiency purposes, this step is usually done before a student requests a hint, but could also be done after. In this article, we focus on the automated process rather than the timing).
(3) The peer states are *transformed* into a state space.
(4) The set of all states in the state space are *narrowed down* to the best next one for the current student. This best state is chosen using a *relevance* criterion (the next step must be reachable from the student's current state in the network) and also a *quality* criterion (it must be the best next step from that position as determined by a MDP).

expected time to a solution [38] ("Poisson Path"), finding the most likely path an average student would take from the current state [38] ("Independent Probable Path") or using a custom distance measure to select the closest state [41] (SourceCheck, solution matching step). With respect to the HINTS framework, each of these selection techniques could replace the MDP in Figure 2 as the quality criterion. Note that the selected next state can be a combination of existing states, rather than a directly existing one, as in the Continuous Hint Factory [33], where the weighted sum of peer edits is used to select the best next state.

While there is currently no definitive evidence to suggest which of the these techniques are most effective at producing hints, some comparison studies have been conducted. In Reference [38], the authors' suggested "Poisson Path" and "Independent Probable Path" techniques most closely matched next-steps chosen by human experts. However, the programming exercises tested were simple and there was variation in technique performance across exercises. In addition, agreement or disagreement with experts does not necessarily imply high- or low-quality hints. These types of techniques were also compared in Reference [40].

Another key decision when producing a hint system such as this is how to represent student programs as states. In recent work, many variations have been presented. These include representing programs as ASTs [7], based on their output [16, 35] ("worldstates"), based on a standardised syntactic form, known as a *canonical form* [35] ("codestates") or based on their components [39][1]

[1]Note that in this case each component of a student's program is used to represent it in a different state space, so multiple next steps can be generated on each of the parts.

("root paths"). Programs may also be represented by the inputs they were tested on by students [7] (input hints), or as points in a continuous space [33]. Such representations often govern the nature of the hint given to students (e.g., if output is used, then the hints would suggest the next output to aim for. If input is used, then the hints might suggest the next input to try testing the program on). Each of these representation steps are considered to be transformations under HINTS, and would replace steps 1 and 2 in Figure 2.

Since there can be large variations in the programs written by students, using general states to represent these programs can help to reduce the size of the state space, and increase the probability of a match being found. However, the more general the states become, the less information there is available in the next-step. For example, in Reference [35], the "worldstates" were more general than the "codestates" (there was less variation in the output than the syntax), so the state space was smaller. However, knowing the next "worldstate" would only give a student information on how the output should change, and not the syntax. While work such as this has provided insight into the relationship between state type and hint availability, it is still unclear how the balance between next step availability and next step information content impacts on the quality of hints. In any case, when deciding upon a representation, these factors should be considered.

5.2 Generating Next Steps toward a Goal

In the previous section, the discussed techniques involved narrowing down a set of existing program states to just the most appropriate ones for the current student. Here, the discussion is extended to systems that *produce* their own next states, as opposed to selecting them, by working toward some *goal* program. Note that there are other approaches to producing next steps that do not involve a goal program, but these will be discussed in later sections.

5.2.1 Example: Program Strategies in AskElle. In Reference [11], the authors present a Haskell programming tutor, AskElle, which produces automated hints by using model solutions to a programming exercise. In particular, it converts sets of model solutions written by teachers into steps leading to these solutions, called *programming strategies*. This is achieved through the use of a strategy language defined by the authors, which specifies how parts of a solution may be built up from others. The generated steps are in the form of a context free grammar, and can be used to parse a student's program to find potential next steps. Note that the student programs are *normalised* (converted to a standard form) to increase the chance of matching the program to a step. Once potential next steps are found, teacher annotations associated with those steps are also given as hints, and any functions appearing in these annotations are automatically linked to external web pages with further information. Figure 3 shows how this hint method fits into the HINTS framework.

Note that, in this example, three different types of hints are produced—next steps, relevant teacher annotations and relevant web links—as shown in Figure 3. These have all been included for completeness, but the next steps component is the focus of this section. The ideas behind the remaining components will be discussed later in Section 5.3.

Also note that AskElle employs a second separate technique for generating hints, property-based testing, which will also be discussed in Section 5.3.

5.2.2 Discussion. Notice first that AskElle produces its own steps from model solutions, instead of using existing steps. Similar approaches to this that use model solutions include that in Reference [19], which also uses program strategies, and AutoTeach [3], where a solution is automatically converted into steps of increasing detail, called *hint levels*. The process for generating hints in Reference [19] fits into HINTS in the same way as the strategy technique in AskElle, but the details of the individual steps are different, since the hints are generated for different programming languages, and the web link step is omitted. AutoTeach, however, is quite different, because the

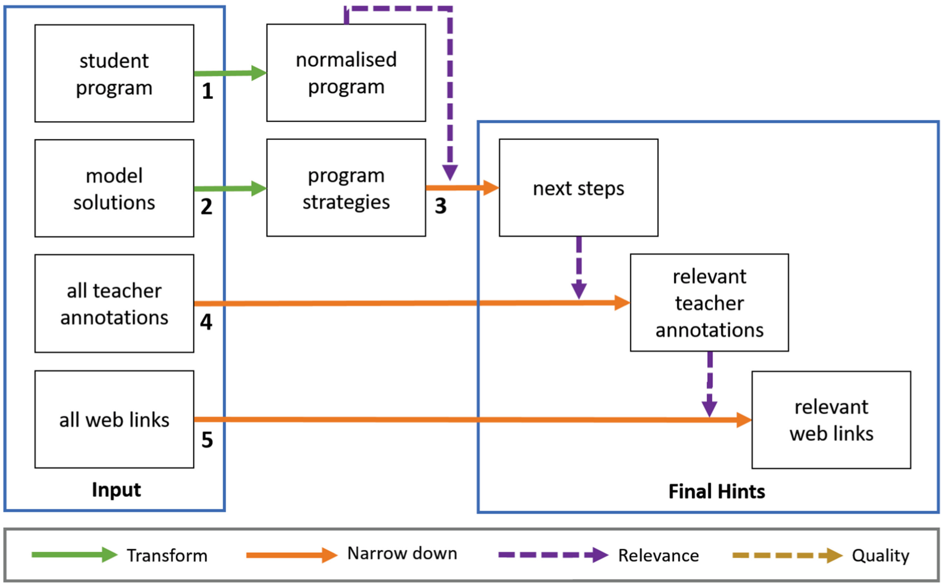

Fig. 3. A visualisation of how the program strategy hint technique in AskElle [11] fits into the HINTS framework. Input: A student's program, model solutions written by teachers, a set of annotations written by teachers to act as feedback and a set of possible links to documentation (whether explicit or implicit). Output hints: Possible next steps a student can take, with relevant teacher annotations and links to documentation. Processes:

(1) A student's program is *transformed* into a standardised form using transformations that affect syntax but not output.
(2) Model solutions are *transformed* into a series of steps called *program strategies* using a strategy language.
(3) The program strategies are *narrowed down* to just the next steps *relevant* to the current student's normalised program.
(4) Teacher annotations are *narrowed down* to just the ones *relevant* to the selected next steps
(5) The set of all possible web links are *narrowed down* based on a *relevance* criterion (that they link to documentation about prelude functions appearing in the selected teacher annotations).

student program is not used in the hint generation process at all. Instead, the steps toward the solution are given incrementally to students as they request more hints. In addition to this, the details of how model solutions are converted into steps are different: Solution steps are generated using customisable visibility rules defined by the authors instead of a strategy language, and there is only one path to one solution. In all of these systems, however, the result is a series of steps that can lead from a blank submission to correct programs.

While AskElle, AutoTeach, and Reference [19] generate the set of hint steps in advance, another option is to wait until the student's program is known, then find the direct edits between that program and a goal program. This is done in the "edit inference" step of SourceCheck [41], where the edits include deletion, movement, reordering, and insertion of code to direct students to a solution. Note that this example was discussed previously to motivate the framework, so a description of how it fits into the framework can be seen in Table 2 in Section 4.1. A similar process is also involved in the hint generation process of ITAP [45] and Reference [46] where, after the closest solution has been found, edits between this solution and the student's are generated and then, in the case of ITAP, further processed to produce next-steps.

One advantage of pre-computing steps to a solution is that these steps can be more easily attached to teacher-authored hints, since they are known in advance. Indeed, this is the case in all three of AskElle, AutoTeach, and Reference [19]. However, a disadvantage is that it can be more difficult to deal with mistakes—if a student makes a mistake, he or she may no longer be on a path to a solution, meaning a next-step hint cannot be produced using that solution. It is possible to adapt the technique by including buggy solutions, so that the system can recognise mistakes if a student is on a path to these [10], but this means the teacher must anticipate the types of mistakes students will make. Edit-based approaches can still produce next-steps if there are mistakes by simply identifying parts of the solution not in the student's program.

One advantage of these techniques over the ones in the previous section is that they do not require existing steps. However, this also makes them sensitive to the input model solutions. In particular, if a student's attempt is completely different from any known model solutions, then hints may either be unavailable or of low quality. For example, in the worst case, an edit-based approach might instruct a student to delete everything then re-write the solution. The program strategies approach might not be able to match a student program to a step toward a known solution, leaving no next-step hint. As such, these approaches are most effective when the model solutions are reflective of the different strategies students can take to solve a problem.

It is interesting to note that, when model solutions are not sufficiently close to the student's program, teachers using these systems must consider a tradeoff between hint quality and hint availability. By setting restrictions on when hints can be produced (e.g., only when the model solution is close enough or only when the hint increases the number of tests passed, as in ITAP), teachers can increase the average relevance or usefulness of hints, but this will also reduce their availability. Conversely, if restrictions are removed, then hint availability can be increased but at the cost of quality. For example, if a student's program cannot be matched on a path to a solution using the program strategies approach, then they can still be given a worked example to some solution—just perhaps not the one they were aiming for. Since the purpose of a hint system is to produce hints, high hint availability is clearly desirable. However, some recent research has suggested that poor quality hints can discourage students from seeking further help [42], so the balance between hint availability and quality in these systems is an important consideration that requires further investigation.

Note that the step generation techniques discussed in this section can be used in combination with the step selection techniques discussed in the previous section. Specifically, after a set of steps is automatically generated, an appropriate next step can be selected from these, as in AskElle. In addition, after an appropriate next step is selected, the difference between this state and the student's current state may be too large, so the automated generation techniques can use this as a goal to automatically generate smaller steps. A good example of how these ideas can be combined is ITAP. Here the nearest solution is first selected. This solution is then changed slightly to better match the student, using an idea discussed later in Section 5.4. Finally, smaller "micro" edits are *generated* toward the solution.

5.3 Comparing Program Features

So far, we have explored two important hint technique ideas: the selection of next steps from past data and the generation of next steps using a goal. We now turn to another important idea in hint generation: The utilisation of *program features*, such as output or structure, to produce hints. Specifically, we discuss how features of a student program can be compared to expected features to produce hints based on dissimilarity, or to assign pre-written teacher feedback to the student. Note that these are frequently paired with ideas from other sections to produce complex and interesting hint techniques.

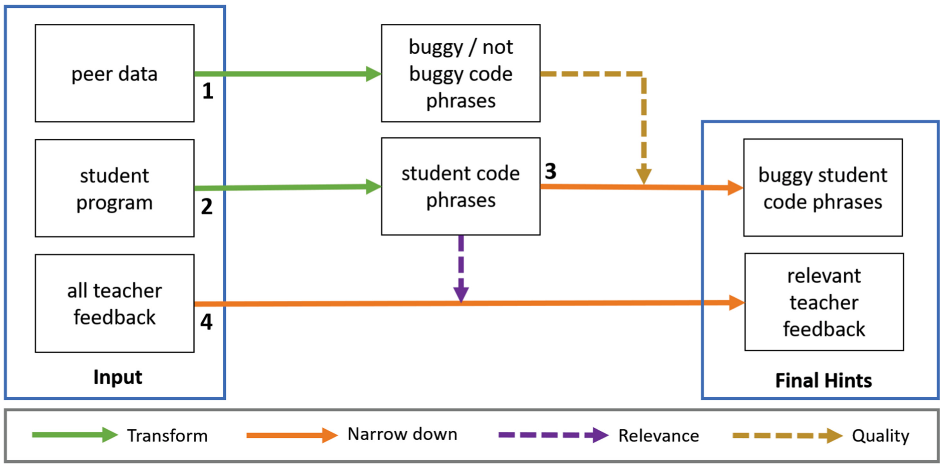

Fig. 4. A visualisation of how the hint techniques in Codewebs [32] fits into the HINTS framework. Input: Peer data, a student's program, and teacher feedback to be triggered by particular program pieces (code phrases). Output hints: Locations of potential bugs in student programs and relevant teacher feedback for the student's program. Processes:

(1) Peer programs are *transformed* into a set of buggy and non-buggy code phrases. This is done by transforming all peer programs into ASTs and then, using a technique for discovering probabilistic equivalence classes presented by the authors, collecting similar subtrees together. Specific code phrases are then identified as buggy or not based on their association with incorrect or correct peer programs.
(2) Similarly to the peer programs, a student's program is *transformed* into a set of code phrases.
(3) The student code phrases are *narrowed-down* to specific buggy phrases based on a *quality* criterion (that they have been identified as buggy in the peer data).
(4) Teacher feedback is *narrowed down* to just the feedback *relevant* to the student code phrases.

5.3.1 Example: Codewebs Engine. In Reference [32], the authors present a technique for extracting patterns, called *code phrases*, from large numbers of peer programs. They then show how these patterns can be used to identify bugs and scale-up teacher feedback to provide hints to new students. Specifically, they show how particular patterns can be attached to teacher feedback, or automatically identified as buggy by considering their relative frequency in incorrect peer programs. When these patterns are then identified in a new student's program, the teacher feedback or a bug warning can then be automatically offered to the student. A visualisation of how this technique fits into HINTS is given in Figure 4.

5.3.2 Discussion. Notice in this example that program features, in the form of code phrases, were used to generate hints. Program features can include anything directly derivable from the program, such as its output, intermediate states or syntactic patterns. They can also be different representations of the program, such as its AST, dependence graph or canonical (standardised) form. In addition, they can be more abstract features, such as which next step is best for the program, or which transformations correct it.

When discussing how program features can be used to generate hints, it is useful to consider one of the simplest approaches first. Specifically, a list of expected features can be compared to a list of actual features of a student's program, and the differences can then be highlighted as hints. This is the idea behind the very widely used feedback approach for programming [20], test cases, where the student's output is compared to the expected output to find differences. Further examples are given in Table 3.

Table 3. Examples of How Features of a Student Program Can Be Directly Compared to a Set of Expected Features in Order to Produce Hints

Feature	Comparison Example(s)
Concepts	In Reference [7], the "concepts" present in a student's program (e.g., if statements, for loops) are compared to expected concepts (i.e., the concepts frequently present in correct peer submissions). Any expected concepts missing from the student's program are then given as *concept hints*.
Properties	In the property-based testing technique in References [11, 18] (AskElle) the properties of a student's program are compared to the properties it is expected to satisfy. Information about any unsatisfied properties is then given as hints.
Syntax	In Reference [41] (SourceCheck) and Reference [45] (ITAP) the syntax of a student's program is compared to the expected syntax (model solution). The differences are then used to produce hints in the form of syntactic edits. Interestingly, since these techniques can also be viewed as guiding a student toward a goal (see Section 5.2), they are good examples of how hint techniques can be viewed from different perspectives, and how different hint ideas can relate.

Extending upon the idea of comparing features to a set of correct features, sometimes features are only expected under certain conditions. This is a key idea behind constraint-based tutoring systems [29], which check student programs against a set of *constraints*. These constraints involve a *satisfaction condition* (i.e., the expected features) and a *relevance condition* (i.e., the condition under which these features are expected). In Reference [27], patterns are matched against student programs using constraints and dependence graphs to produce hints.

Note that the comparison between features can vary in directness. For instance, in the Codewebs example, every potentially buggy pattern in a student's program is not reported directly to the student. Instead, the AST structure is considered so buggy patterns closer to the leaves take precedence over their ancestors. Similarly, in Reference [9] (spectrum-based fault localisation) and Reference [34], the student's output and execution trace is compared to the expected output and execution trace (respectively), and the differences are used to infer which parts of the program contain bugs. As such, the incorrect output and intermediate values are not directly reported, but rather the program parts associated with them.

Notice that the Codewebs example also introduces a second approach to generating hints from features. Namely, teacher feedback can be attached to particular features (e.g., code phrases), then automatically scaled-up to any students whose programs have these features. This is in contrast to the previously discussed techniques, where the features of a student's program were compared to expected features, either directly or indirectly. Some further examples of using features to scale-up teacher hints are given in Table 4 along with the Codewebs example.

If these teacher hint examples were represented diagrammatically under HINTS, then all examples would include Step 4 of Codewebs (see Figure 4), where teacher feedback is narrowed down to relevant teacher feedback based on some feature of the student's program. They would also either include a transformation step similar to Step 2 of Codewebs to represent the student program in terms of its features, or a series of steps to produce those features. For example, MistakeBrowser finds features (i.e., corrections) by first repairing the student program, using steps described in the next section. Note that the process of writing teacher hints for particular features is itself not automated. However, these techniques can still be considered automated by our definition in Section 3 if the teacher hints are available, pre-written, as input. As such, this manual aspect is treated as an input through the HINTS framework, rather than an automated step.

Table 4. Examples of How Teacher Feedback Can Be Attached to Particular Features of a Student Program in Order to Produce Relevant Hints

Feature	Teacher Hint Example(s)
Output	ViDA [26], CSF^2 [14]: Teacher hints are written for various incorrect outputs, or sets of output, on different test cases. When the output of a student's program matches one of these known cases, the corresponding hint is given to the student.
Corrections	MistakeBrowser [15]: Teacher feedback is attached to clusters of peer programs that are corrected by the same transformations (see Section 5.4 for details of how these are found). When a new student's program can also be corrected by one of these transformations, the teacher feedback for that cluster is given.
Strategy	Reference [12]: Teacher feedback is written for different algorithmic strategies. These strategies are identified based on the intermediate values of some expressions in the program. When a student submits an incorrect program following a particular strategy, the teacher feedback associated with that strategy is given.
Patterns	Codewebs [32]: Teacher feedback is written to address particular bugs in peer data. Then, when these bugs are identified in a new student's program through patterns (code phrases), the same teacher feedback can be given to the new student. Reference [6] also suggests scaling-up teacher feedback using patterns from Codewebs.

Just as the comparison-based ideas could be indirect, note that techniques for scaling-up teacher feedback can also be indirect. For example, in Reference [37], teacher hints are used as supervised labels to train a classifier to predict correct hints from program features. This classifier can then be used to propagate hints to new programs. As such, hints are not directly generated from the program features, but instead from a model trained on the features.

While this section discusses techniques for producing hints by comparing features or attaching them to teacher hints, note that there are many other uses of program features. In fact, almost every hint technique uses program features in some way. For example, in Section 5.1, where the techniques involved selecting next steps, program features were often used to produce program states. The techniques discussed in the next section will also make use of program features. As such, even though this section discusses two specific applications of program feature extraction, the set of possibilities is not limited to these applications.

5.4 Automatically Repairing Programs

In the previous sections, many interesting ideas for producing hints were discussed. These included selecting next steps, generating next steps using a goal, and also utilising the program features to produce hints. This section builds on these by considering another important idea. In particular, it considers techniques for automatically repairing student programs to to produce feedback.

5.4.1 Example: SYNFIX. In Reference [5], the authors present a method for automatically correcting syntactic errors in student programs using machine learning and peer data. Specifically, the peer data are used to train a **recurrent neural network (RNN)** to predict correct sequences of program pieces (tokens). When a new student requires help, the RNN can then be used to find new or alternative tokens to insert near the error location to correct the program. Corrections found in this way can then be used to produce hints for the student.

5.4.2 Discussion. In this example, first notice that there are two important steps. First, the student programs are represented by their features (i.e., as sequences of tokens), similarly to the

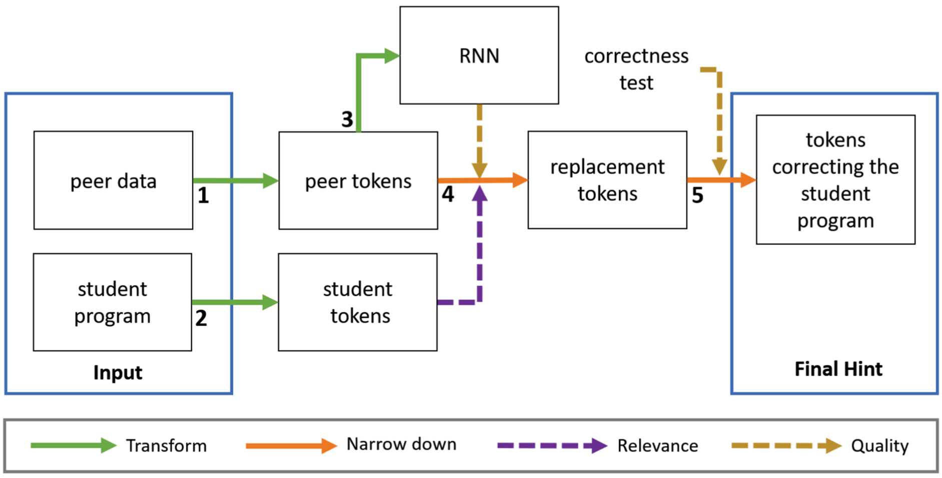

Fig. 5. A visualisation of how the program strategy hint technique in SYNFIX [5] fits into the HINTS framework. Input: Peer data and a student's program. Output hint: Token corrections to fix syntax errors in the student's program Processes:

(1) Peer programs are *transformed* into sequences of tokens, such as "if", "==", "exp", and so on.
(2) Similarly, a student's program is *transformed* into a sequence of tokens
(3) An RNN is trained on the sequences of correct peer tokens to learn correct patterns. As such, the correct peer tokens are *transformed* into a model for correct token sequences.
(4) Peer tokens are *narrowed down* to a series of replacement tokens based on a *quality* criterion (that they are predicted by the RNN) and a *relevance* criterion (that they follow on from other parts of the student program near the syntax error location).
(5) The sequence of replacement tokens (which can be thought of as a set of subsequences) is *narrowed down* to a single subsequence based on a *quality* criterion (that it corrects the student program).

techniques in the previous section. Second, a model (RNN) is built using machine learning to represent how these features should be correctly combined (i.e., by learning correct token sequences). Similar techniques using machine learning are summarised in Table 5 along with this one.

It is interesting to note that these examples are highly related to the feature comparison techniques discussed in the previous section. Recall that the previous techniques involved comparing the features of a student's program to the expected features, either directly or indirectly. These examples are similar, but the comparison is even more indirect, because the expected features are now abstracted to a model first. For example, in SYNFIX, the sequences of tokens in a student's program are not directly compared to a list of expected token sequences, but they are input into a model that outputs expected token sequences.

As an alternative to the machine learning techniques discussed so far, which build a model to correct programs, another option is to perform a *search* for possible corrections. This involves first defining some search space of possible program features (e.g., expressions or edits), and also some method for testing correctness (e.g., test cases). Then, starting with the student's current program, a search can be automatically performed over the search space to find edits that will satisfy the correctness test. Some examples of this are given in Table 6. Note that these search techniques relate to *program synthesis*—the task of automatically creating a program that satisfies some criteria. The difference between the techniques from an educational context, compared to

Table 5. Examples of Hint Techniques That Involve the Use of Machine Learning to Produce a Model to Represent Correct Program Features

Paper(s)	Model of Correct Features	Correction Technique
RLAssist [13]	Here, the program features are sequences of actions that correct the program. The model is an agent trained through reinforcement learning to learn correct sequences of actions. The learning process can be sped up by giving the agent examples of correct action sequences ("expert demos").	The agent is used to produce actions that will correct the student's program.
[22]	The model is a set of rules for predicting correct and incorrect programs using AST patterns. In Reference [22], these rules are obtained by training a rule learner on correct and incorrect peer programs. In Reference [31], they are produced through argument-based machine learning (ABML). Note that, in the case of Reference [31], the model would be treated as input under HINTS, since ABML is not automated)	The rules are used to identify buggy AST patterns in the student's program, or good patterns to include in the program
SYNFIX [5]	An RNN, which is trained on sequences of correct peer tokens.	The RNN is used to find tokens that will correct the student's program near the error location.

other areas, though, is that resources produced by teachers or peers can be utilised to correct programs.

It is interesting to note a relationship between the two methods of correcting student programs discussed in this section, and the two methods of generating steps toward a goal discussed in Section 5.2. Recall in Section 5.2 that steps could either be generated in advance before the student's program was seen, or they could be produced "on the fly" by finding edits between the student's program and the goal. Here, again, there is a choice between producing a model of correct features in advance, or waiting until the student's program is known to perform a search. In some sense, the techniques are quite similar in nature, but in this section the goal is to pass all the test cases or successfully compile the program, and in Section 5.2, the goal was to reach a model solution.

Once these techniques have been used to find corrections, there are many possible uses for these corrections when producing hints. For example, in MistakeBrowser [15], the corrections are used to scale-up teacher hints. In Reference [36] and Reference [23], the locations of the corrections are suggested to students as places to focus on. An interesting exploration of many different possible uses can be found in Reference [50] where, for example, the authors suggest running the corrected code, then highlighting differences in the behaviour of this (i.e., the intermediate states) and the student's program.

Since all of these techniques aim to correct student programs, one important consideration when comparing these techniques is how many programs they are able to correct. It is difficult to directly compare these techniques in this regard, because they have been evaluated on different types of programs and serve different purposes (e.g., some correct syntax errors, and others logic errors). In addition, the types of evaluations performed on them have been different. For example, some

Table 6. Examples of Hint Techniques That Involve a Search to Automatically Correct Student Programs

Paper	Search Space	Search Technique
[36]	The search space is defined by a domain specific language (DSL), which defines the possible expressions. This is created by mining peer data for expressions that are used more than 10 times.	The search begins at the student's program, which is slowly modified in steps. In each step, the current best program is modified to satisfy more input/output examples using the DSL. When all input/output examples are satisfied, the search is complete.
[23]	The search space consists of a set of rewrite rules learnt from past student data. Probabilities are assigned to each of these re-write rules based on their prevalence in the peer data.	The search involves applying rewrite rules or combinations of them in order of increasing probability on the new student's program, until a correct solution is found or the system times out. In this way, more common solutions are explicitly favoured, and shorter solutions are implicitly favoured.
Mistake Browser [15]	The search space consists of sets of transformations learned from peer program attempts.	The search involves trying different transformations until the program is corrected.
ITAP [45]	(Note that this is a single part of a longer process). The search space comes from the closest known solution to a student's program. Specifically, the powerset (all possible subsets) of edits between this solution and the student's program is computed, and this forms the search space.	The search involves applying each of the subsets of edits to the student program, until the closest solution is found.

authors evaluated how often the discovered corrections were later used by students. However, keeping in mind that the contexts were different, of the techniques that *were* evaluated for the percentage of programs they could correct, the results varied greatly from around 27% to 87%. These figures are shown in Table 7. Note that, since less changes must be made to programs that are almost complete, one would expect these techniques to be most effective when a student's program is almost correct.

6 INSIGHTS FROM SURVEYING HINT TECHNIQUES UNDER THE HINTS FRAMEWORK

In the previous section, recent work on automated hints was surveyed in the context of HINTS. This section now presents some key insights resulting from this survey. In particular, it discusses how all of the previously introduced ideas can be integrated together into a single, coherent picture. In addition, it argues that the smaller components that comprise hint techniques should be considered when designing, communicating and evaluating hint systems. It explicitly demonstrates how viewing techniques in terms of their components can lead to important insights on connections between the field of automated programming hints and other areas, such as data-driven evaluation.

Table 7. Reported Percentage of Programs Able to Be Corrected through Automatic Repair Techniques

Paper	Correction % (to the nearest %)	Context: Types of Corrections Being Made
Misktake Browser [15]	87% of students	Logic errors in simple programs. (e.g., "*Repeated (720 students): takes as parameters a unary function f and a number n, and returns the nth application of f. For example, repeated(square,2)(5) returns square(square(5)), which evaluates to 625.*")
Reference [36]	65% of attempts	Logic errors in programs written for a game, called "Code-Hunt"
SYNFIX [5]	32% of programs, with partial corrections for an additional 6%	Syntax errors in programs from an online introductory programming course
RLAssist [13]	27% of programs, 40% of error messages	Syntax errors in programs from an introductory programming course

Moreover, it explores how the similarities between hint techniques can offer insight into the nature of hint generation. Finally, it argues that the multitude of possible hints techniques producible from smaller steps is immense, and that this necessitates further work on evaluation methods.

6.1 It Is Possible to Fit Together Hint Techniques into a Coherent Picture Using Their Components

After having reviewed some key ideas from hint generation techniques in Sections 5.1–5.4, it is now possible to combine these ideas together into a single, coherent picture. This can be done by considering all of these techniques in the context of the HINTS framework—as a series of simpler steps—and then depicting similar steps together. By doing this, it is possible to gain insight into the current state of the field, including the relationships between existing techniques and the potential for future development.

We show this general picture in Figure 6. Note that the figure was created by producing diagrams similar to Figures 2 to 5 for all surveyed techniques, then merging them by grouping similar steps and depicting these with a single arrow. The particular groupings are discussed in more detail below, along with examples and references to further discussion in Section 5.

(1) model solutions → steps. Model solutions can be transformed into steps in the form of program strategies (AskElle [11] and Reference [19]) or hint levels (AutoTeach [3]) (see Section 5.2).
(2) peer data → general features. Peer programs can be represented by their features for various purposes. For example, to form states in a state space, they can be represented by their AST form [7], output [16, 35], canonical form [35], components [39], point in space [33] or inputs they were tested in Reference [7] (see Section 5.1). Also see Section 5.3 for further discussion of features.
(3) student program → student features. This is similar to (2), except the step is applied to the help-seeking student's program instead of peer programs.

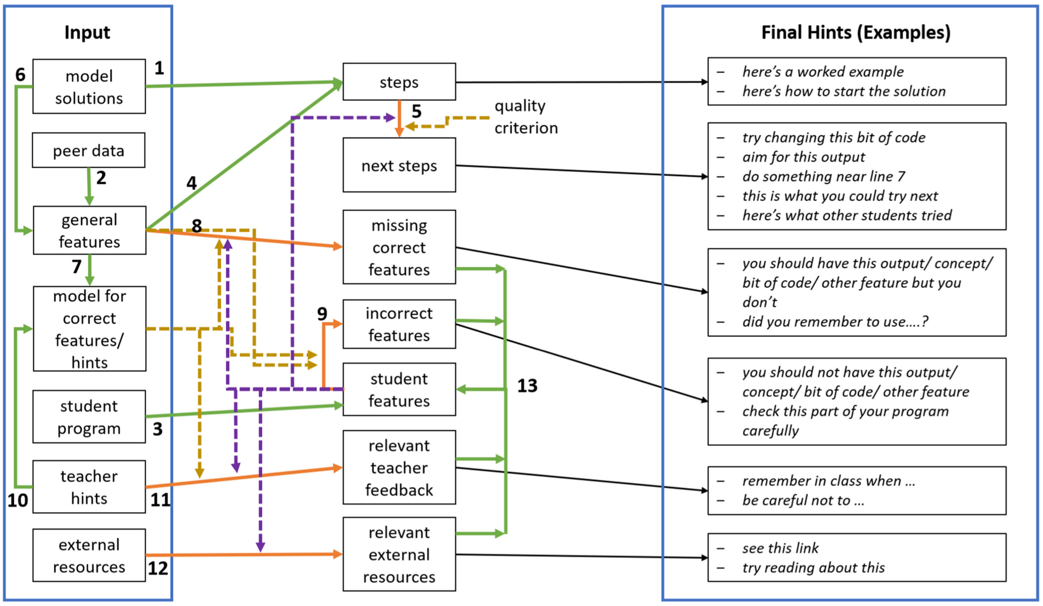
Input
model solutions
peer data
general features
model for correct features/ hints
student program
teacher hints
external resources
1
2
3
4
5
6
7
8
9
10
11
12
13
steps
next steps
quality criterion
missing correct features
incorrect features
student features
relevant teacher feedback
relevant external resources
Final Hints (Examples)
– here's a worked example
– here's how to start the solution
– try changing this bit of code
– aim for this output
– do something near line 7
– this is what you could try next
– here's what other students tried
– you should have this output/ concept/ bit of code/ other feature but you don't
– did you remember to use....?
– you should not have this output/ concept/ bit of code/ other feature
– check this part of your program carefully
– remember in class when ...
– be careful not to ...
– see this link
– try reading about this

Fig. 6. A general diagram summarising the main ideas of the surveyed papers and the relationships between them in the context of the HINTS framework. The leftmost blue box indicates types of hint data that can be given as input. The rightmost blue box gives examples of hints that can be produced from the indicated hint data. Green arrows denote transformation steps according to HINTS. Orange arrows indicate narrow-down steps, with any sources of quality or relevance criteria marked by gold or purple arrows respectively. Example techniques using each transformation and narrow-down step are given in the main text of this section, along with the sections in which they are discussed.

(4) general features → steps. Program states can be collected together into a state space, as in [7, 16, 33, 38, 39, 44]. Note that if a technique involves steps (2) and (4), then they can be treated as a single transformation step, depending on where the emphasis is being placed (see Section 5.1).

(5) steps → next steps. A set of potential next-steps can be narrowed down to just the ones relevant to the current student's program state. This can involve various quality criteria based on MDPs [16, 35, 39] and other approaches [7, 33, 38, 41, 44] (see Section 5.1). These can also come from program strategies [11, 19] (see Section 5.2).

(6) model solutions → general features. Model solutions can be divided into pieces to form a set of features (i.e., parts of programs).

(7) general features → model for correct features. A model can be built on the general features to learn which features are correct/ incorrect using, for example, reinforcement learning [13], rules [22], or an RNN [5] (see Section 5.4).

(8) general features → missing correct features. A set of general expected features can be compared to features of the student's program. Expected features not in the student's program can be identified as missing correct features. For example, these include edits to a solution in References [41, 45, 46] that involve adding code (see Section 5.2). It could also include missing concepts [7] or properties [11, 18] (see Section 5.3). The set of features can also be searched until some correctness test is passed as in References [15, 23, 36, 45], or a model of correct features can be used to narrow them down as in References [5, 13, 22, 31].

(9) student features → incorrect features. Similarly to (8), student features can be compared to expected features, and the features in the student's program that are not expected can be identified as incorrect, either directly or indirectly. For example, edits for deleting code in [41, 45, 46] (see Section 5.2). These can also be based on buggy patterns [32], constraints [27], output [9], or execution traces [34] (see Section 5.3).

(10) teacher hints → model for correct hints. Teacher hints attached to particular program features can be transformed into a model to predict correct hints, as in Reference [37] (see Section 5.3).

(11) teacher hints → relevant teacher hints. Teacher feedback can be narrowed down to just the feedback relevant to the student's features. These can include output [14, 26], edits that correct the program [15] or strategy [12] (see Section 5.3).

(12) external resources → relevant external resources. External resources, such as web links [11], can be narrowed-down to just those relevant to the student's features (see Section 5.3).

(13) student features cycle. Any next steps, missing correct features, and so on, can be treated as features of the student program, which can be used to build more hints.

6.2 During the Design, Communication, and Evaluation of Hint Systems, the Smaller Components That Comprise Hint Techniques Should Be Considered

The fact that hint techniques are comprised of many smaller steps suggests that these steps should play a key role in the design, communication and evaluation of hint systems. In recent work, there have been many interesting techniques and ideas presented about automated hint generation. However, without considering these techniques as a series of smaller processes that can be modified and re-purposed, we can miss opportunities to integrate them and utilise them in new work. As such, when communicating and designing new techniques, we should consider the different steps comprising these techniques, the possible variations on these steps and whether these steps could be used in different contexts for different purposes. In addition, when evaluating hint systems, we should not only consider the effectiveness of entire techniques, but also evaluate the individual components and how choices of these affect the overall quality of hint systems.

6.3 The Connection between Hint Generation Techniques and Data-driven Hint Evaluation Techniques Should Be Investigated Further

Recently, along with the development of hint generation techniques, there has been much interest in hint *evaluation* techniques. In particular, there has been a focus on *data-driven* evaluation methods, which utilise historical student data to evaluate hint systems. Since evaluation methods serve a different purpose from hint generation techniques, one would expect these two kinds of techniques to be quite different. However, by considering hint techniques in the context of our framework, as a series of simpler components, it is possible to observe some interesting relationships between hint techniques and data-driven evaluation methods.

To see these relationships, first consider the narrow-down step of the HINTS framework. Recall that this step involves narrowing down a set of hint data to just the most appropriate subset for the current student, using relevance or quality criteria. As such, any hint technique with a narrow-down step is, in some sense, performing an evaluation of the hint data to decide what to select. For example, the SourceCheck [41] hint technique (discussed previously) involves a narrow down step where all peer solutions are narrowed down to just the closest one. To do this, each of the potential solutions must be "evaluated" for their quality, by checking their distance from the current student's program. By considering hint techniques in the context of HINTS, we can thus begin to notice a link between hint techniques and the general idea of evaluation.

Now consider an actual data-driven evaluation method, presented in Reference [45] to evaluate ITAP (which was introduced previously). This technique involves computing the number of edits needed to correct a student's program (i.e., its distance from the student's program). In essence, just as in SourceCheck, the solution is being evaluated based on how close it is to the student's program. While these techniques are not exactly the same, since the distance measure is different, this indicates there is an important relationship between this evaluation technique and the narrow-down step of SourceCheck.

We can observe another example of the correspondence between hint generation techniques and hint evaluation techniques by considering MistakeBrowser from Reference [15], and a different evaluation method for ITAP in Reference [45]. In MistakeBrowser, the hint system narrows down a large set of transformations to just a small number by "evaluating" whether or not they are able to correct a student's program. Similarly, in Reference [45], in addition to path length, there is an evaluation to check whether a series of hints (in the form of edits, which can be thought of as transformations) would actually lead students to a solution. As such, in both cases, transformations are evaluated based on whether they correct the student's program, suggesting an important link between these techniques.

Note that it is not only narrow-down steps that seem to correspond to evaluation techniques, but also transformation steps. Recall that these steps involve changing the representation of the data to produce hints. For example, in Reference [35], which uses the HintFactory approach (see Section 5.1), programs are transformed into states (i.e., *worldstates*) based on their output, which are then transformed into a state-space. This relates to the evaluation method presented in Reference [28], where peer programs are clustered and transformed into a state-space so that teachers can visualise how students complete an exercise. Two additional examples are described in Table 8, along with a summary of the examples already discussed.

The link between transformation steps and data-driven evaluation techniques is perhaps not so surprising, considering that these evaluation techniques must convey information about the hint system to a teacher in an understandable way. This will often involve techniques for automatically representing large volumes of data in a coherent way, or transforming them into more accessible forms. Since this is also often the purpose of transformation steps in hint techniques, this suggests a correspondence between them.

While a full survey of evaluation methods is beyond this scope of this article, there is clearly an important link between hint generation and data-driven evaluation methods, which would be a worthwhile avenue for further investigation. Perhaps there is not only potential to connect different hint techniques, but also to use their steps to improve evaluation methods, or to use the steps of evaluation methods to extend hint techniques.

6.4 The Design of Hint Techniques Can Provide Insight into the Nature of Hint Generation

It is clear from the HINTS framework that all automated hint techniques exhibit remarkable similarities in structure and in the processes used to construct them. In particular, they may all be described by two simple operations applied iteratively: narrowing down and transforming hints from previous levels. Considering their diversity and sophistication, this is surprising, and prompts the question of why this is the case.

One potential reason could be that the initial inputs to these systems are already highly complex, so a simple system could still leverage these to produce intelligent feedback. Indeed, hints or tests pre-written by teachers and work produced by other students contain vast amounts of information, and a system that narrowed this down could produce high-quality hints. This could explain why even the simplest automated hint techniques can produce highly sophisticated hints.

Table 8. Links between Steps in Hint Generation and Data-driven Evaluation Techniques

Evaluation Technique	**Related Transformation or Narrow Down Step in a Hint Technique**
check the number of edits between the student's program and the solution (i.e., the distance) (ITAP [45])	narrow down the set of all solutions by checking their distance from the student's program and choosing the closest one (SourceCheck [41])
check how many chains of hints (transformations) actually lead to a solution (ITAP [45])	narrow down sets of transformations to just the ones that lead to a solution [15]
transform student programs into a standardised form and create an state space so teachers can visualise how students complete an exercise [28]	transform student programs into a standardised form and create an state space so the next state can be selected [35]
transform student programs into code-phrases then organise them so teachers can understand the data (e.g., they can "count the number of students who submitted the same or a similar class of solutions") (Codewebs [32])	transform student programs into code-phrases then organise them so automated hints can be given to students based on the code-phrases (Codewebs [32])
use a templating language to express model solutions in many different forms to account for different programming strategies when evaluating student solutions against experts [43]	use a templating language ("strategy language") to express paths to model solutions in many different forms to account for different programming strategies when guiding students to a solutions (AskElle [11])

Another potential reason for why hint techniques share such similar operations could be that these operations reflect the nature of programming hints. Programming hints often involve highlighting mistakes to students, so we would expect the system to have some way of narrowing down the student's program to just the mistakes. In addition, programming hints often involve suggesting new ways to proceed, so we would expect the system to have some knowledge about a goal, then to narrow this down to the parts the student has not yet succeeded in. In addition, programming hints can related to many different aspects of a student's program, such as output, style, structure, syntactic patterns or runtime, so we would also expect the system to need transformations to give feedback on these different aspects. As such, perhaps a reason why hint techniques involve transformation and narrow-down steps could be that these steps reflect the nature of programming hints. Perhaps this insight can help to shape the way we think and communicate about hint techniques, and motivate further work. For example, it would be interesting in future to investigate whether the HINTS framework could be extended to cover techniques from other domains, such as physics, logic or mathematics, where some hints could be of a similar nature.

6.5 The Multitude of Possible Hint Techniques from Components Necessitates Further Work on Evaluation Methods

From the survey of hint techniques presented in Section 5, it is clear that there are many different approaches to generating hints. Specifically, there are many different interchangeable and stackable components that comprise hint techniques, and different combinations of these can result in a vast number of possible hint approaches. This suggests a need to develop scalable and versatile evaluation methods that can cope with such a multitude of potential techniques.

Even now, with a relatively small number of techniques compared to the possible number in future, these techniques are so numerous that it is difficult to decide which are most effective for different scenarios (e.g., different programming languages, learners or courses). This is exacerbated by the fact that evaluation methods are often applied inconsistently [21]. Considering the potential for far more hint techniques in future, it is thus of great importance that we continue to develop and extend evaluation methods to ensure the full potential of automated hint techniques.

7 CONCLUSION

In this article, we have surveyed and developed key theoretical ideas behind recent work from 2014 to 2018 on generating automated hints for programming exercises. Specifically, we have presented a novel framework, the HINTS framework, for describing techniques for hint generation, and surveyed these techniques in the context of this framework. We have shown that, by considering hint techniques as a series of smaller steps, it is possible to draw recent work together into a single coherent picture. We have argued that this perspective on hint techniques has implications for how we design, communicate and evaluate hint systems, and can provide useful insights into the nature of hint generation. Finally, we have identified a potential relationship between hint generation and evaluation techniques that could be utilised to improve both, and have argued that the piecewise nature of hint techniques necessitates the further development of evaluation methods. By bringing more clarity to the area of automated programming hint generation, this work acts as an important step toward realising the full potential of automated programming tutors, with the ultimate goal of maximising educational outcomes.

REFERENCES

[1] Kirsti M. Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Comput. Sci. Educ.* 15, 2 (2005), 83–102.

[2] Aivar Annamaa, Reelika Suviste, and Varmo Vene. 2017. Comparing different styles of automated feedback for programming exercises. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research.* ACM, 183–184. https://doi.org/10.1145/3141880.3141909

[3] Paolo Antonucci, Christian Estler, Durica Nikolić, Marco Piccioni, and Bertrand Meyer. 2015. An incremental hint system for automated programming assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, 320–325.

[4] María Lucía Barrón-Estrada, Ramón Zatarain-Cabada, Francisco González Hernández, Raúl Oramas Bustillos, and Carlos A. Reyes-García. 2015. An affective and cognitive tutoring system for learning programming. In *Proceedings of the Mexican International Conference on Artificial Intelligence.* Springer, 171–182.

[5] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. arXiv:1603.06129. Retrieved from https://arxiv.org/abs/1603.06129.

[6] Zhenghao Chen, Andy Nguyen, Amory Schlender, and Jiquan Ngiam. 2017. Real-time programming exercise feedback in MOOCs. In *Proceedings of the 10th International Conference on Educational Data Mining.* International Educational Data Mining Society.

[7] Sammi Chow, Kalina Yacef, Irena Koprinska, and James Curran. 2017. Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization.* ACM, 5–10.

[8] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: A systematic review. In *Proceedings of the 20th Australasian Computing Education Conference.* ACM, 53–62. https://doi.org/10.1145/3160489.3160492

[9] Bob Edmison and Stephen H. Edwards. 2015. Applying spectrum-based fault localization to generate debugging suggestions for student programmers. In *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'15).* IEEE, 93–99.

[10] Alex Gerdes. 2012. *Ask-Elle: A Haskell Tutor.* Ph.D. Dissertation. Universiteit Utrecht.

[11] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. 2017. Ask-Elle: An adaptable programming tutor for Haskell giving automated feedback. *Int. J. Artif. Intell. Educ.* 27, 1 (2017), 65–100.

[12] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 41–51.

[13] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep reinforcement learning for programming language correction. arXiv:1801.10467. Retrieved from https://arxiv.org/abs/1801.10467.

[14] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 278–283. https://doi.org/10.1145/3159450.3159502

[15] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the 4th ACM Conference on Learning@ Scale*. ACM, 89–98.

[16] Andrew Hicks, Barry Peddycord, and Tiffany Barnes. 2014. Building games to learn from their players: Generating hints in a serious game. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 312–317.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 86–93.

[18] Johan Jeuring, L. Thomas van Binsbergen, Alex Gerdes, and Bastiaan Heeren. 2014. Model solutions and properties for diagnosing student programs in Ask-Elle. In *Proceedings of the Computer Science Education Research Conference*. ACM, 31–40.

[19] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*. ACM, 43–54.

[20] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 41–46.

[21] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.* 19, 1 (2018), 3. https://doi.org/10.1145/3231711

[22] Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic extraction of AST patterns for debugging student programs. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 162–174.

[23] Timotej Lazar, Aleksander Sadikov, and Ivan Bratko. 2017. Rewrite rules for debugging student programs in programming tutors. *IEEE Trans. Learn. Technol.* 11, 4 (2017), 429–440.

[24] Nguyen-Thinh Le. 2016. A classification of adaptive feedback in educational systems for programming. *Systems* 4, 2 (2016), 22.

[25] Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart. 2013. A review of AI-supported tutoring approaches for learning programming. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 267–279.

[26] Victor C. S. Lee, Yuen-Tak Yu, Chung Man Tang, Tak-Lam Wong, and Chung Keung Poon. 2018. ViDA: A virtual debugging advisor for supporting learning in computer programming courses. *J. Comput. Assist. Learn.* 34, 3 (2018), 243–258. https://doi.org/10.1111/jcal.12238

[27] Victor J. Marin, Tobin Pereira, Srinivas Sridharan, and Carlos R. Rivero. 2017. Automated personalized feedback in introductory java programming moocs. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE'17)*. IEEE, 1259–1270.

[28] Jessica McBroom, Kalina Yacef, Irena Koprinska, and James R. Curran. 2018. A data-driven method for helping teachers improve feedback in computer programming automated tutors. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 324–337. https://doi.org/10.1007/978-3-319-93843-1_24

[29] Antonija Mitrovic. 2012. Fifteen years of constraint-based tutors: What we have achieved and where we are going. *User Model. User-adapt. Interact.* 22, 1-2 (2012), 39–72.

[30] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17. https://doi.org/10.1145/3105906

[31] Martin Možina, Timotej Lazar, and Ivan Bratko. 2018. Identifying typical approaches and errors in prolog programming with argument-based machine learning. *Expert Syst. Appl.* 112 (2018), 110–124. https://www.sciencedirect.com/journal/expert-systems-with-applications/issues.

[32] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 491–502.

[33] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The continuous hint factory-providing hints in vast and sparsely populated edit distance spaces. *J. Educ. Data Min.* 10, 1 (2018), 1–35.

[34] Benjamin Paaßen, Joris Jensen, and Barbara Hammer. 2016. Execution traces as a powerful data representation for intelligent tutoring systems for programming. In *Proceedings of the 9th International Conference on Educational Data Mining*. International Educational Data Mining Society.

[35] Barry Peddycord Iii, Andrew Hicks, and Tiffany Barnes. 2014. Generating hints for programming problems using intermediate output. In *Proceedings of the 7th International Conference on Educational Data Mining*. International Educational Data Mining Society.

[36] Daniel Perelman, Sumit Gulwani, and Dan Grossman. 2014. Test-driven synthesis for automated feedback for introductory computer science assignments. In *Proceedings of Data Mining for Educational Assessment and Feedback (ASSESS'14)*.

[37] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *Proceedings of the International Conference on Machine Learning*.

[38] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the 2nd ACM Conference on Learning@ Scale*. ACM, 195–204.

[39] Thomas W. Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating data-driven hints for open-ended programming. In *Proceedings of the 9th International Conference on Educational Data Mining*. International Educational Data Mining Society, 191–198.

[40] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *Int. J. Artif. Intell. Educ.* 29, 3 (2019), 368–395. https://doi.org/10.1007/s40593-019-00177-z

[41] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a data-driven feedback algorithm for open-ended programming. In *Proceedings of the 10th International Conference on Educational Data Mining*. International Educational Data Mining Society.

[42] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 311–322.

[43] Thomas W. Price, Rui Zhi, Yihuan Dong, Nicholas Lytle, and Tiffany Barnes. 2018. The impact of data quantity and source on the quality of data-driven hints for programming. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 476–490. https://doi.org/10.1007/978-3-319-93843-1_35

[44] Kelly Rivers and Kenneth R. Koedinger. 2014. Automating hint generation with solution space path construction. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 329–339.

[45] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *Int. J. Artif. Intell. Educ.* 27, 1 (2017), 37–64.

[46] Saksham Sharma, Pallav Agarwal, Parv Mor, and Amey Karkare. 2018. TipsC: Tips and corrections for programming MOOCs. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 322–326. https://doi.org/10.1007/978-3-319-93846-2_60

[47] Josep Silva. 2011. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.* 42, 11 (2011), 976–991.

[48] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. 2008. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*. 71–78.

[49] Michael Striewe and Michael Goedicke. 2014. A review of static analysis approaches for programming exercises. In *Proceedings of the International Computer Assisted Assessment Conference*. Springer, 100–113.

[50] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the design space of automatically synthesized hints for introductory programming assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 2951–2958.

[51] Thomas James Tiam-Lee and Kaoru Sumi. 2018. Adaptive feedback based on student emotion in a system for programming practice. In *Proceedings of the International Conference on Intelligent Tutoring Systems*. Springer, 243–255. https://doi.org/10.1007/978-3-319-91464-0_24

Received August 2019; revised November 2020; accepted June 2021