Rewrite Rules for Debugging Student Programs in Programming Tutors

Timotej Lazar, Aleksander Sadikov, and Ivan Bratko

Abstract—Data-driven intelligent tutoring systems learn to provide feedback based on past student behavior, reducing the effort required for their development. A major obstacle to applying data-driven methods in the programming domain is the lack of meaningful observable actions for describing the students' problem-solving process. We propose rewrite rules as a language-independent formalization of programming actions in terms of code edits. We describe a method for automatically extracting rewrite rules from students' program-writing traces, and a method for debugging new programs using these rules. We used these methods to automatically provide hints in a web application for learning programming. In-class evaluation showed that students receiving automatic feedback solved problems faster and submitted fewer incorrect programs. We believe that rewrite rules provide a good basis for further research into how humans write and debug programs.

Index Terms—Computer-assisted instruction, intelligent tutoring systems, program synthesis, automatic debugging.

1 Introduction

PROGRAMMING is increasingly recognized as a fundamental skill, with several countries already including it in their national curricula. Many initiatives exist to make programming education available to everyone, such as the European Coding Initiative and Code.org.[1]

Intelligent tutoring systems (ITSs) can help achieve this goal. An ITS is an educational tool that emulates a human teacher in a one-to-one tutoring situation, which is much more effective than teaching in a traditional classroom setting [1], [2]. Like human tutors, ITSs provide immediate, personalized feedback to students [3]. While computer tutors are not as effective as humans in many domains, they can scale much better.

ITSs are typically problem-based: students solve problems while the tutor monitors their progress and provides guidance when necessary or requested [4]. ITS authors must anticipate correct and incorrect steps a student might take, and define appropriate responses. Building a tutor that can provide feedback for a wide variety of problems is thus expensive and time-consuming [5].

Data-driven tutors alleviate authoring efforts by using student data to construct or improve their feedback [6]. One such method – the Hint Factory – was first implemented in a logic tutor [7], where the task is to deduce a conclusion from given premises. The tutor observes which inference rules the students use in each state (a state is described by the set of premises that have already been deduced), and learns a policy for solving each problem. This policy is used to provide hints to a new student who reaches a state the tutor has seen before.

Methods like the Hint Factory work well for domains where the problem-solving steps are chosen from a set of well-defined actions. In the logic tutor, actions are inference rules like *modus ponens*. The user interface has a separate button for applying each rule, so that the tutor can follow students' actions.

In programming, however, the only observable actions are edits to program code. While the action "use *modus ponens* with premises A and A⇒B" tells us something useful about what the student is doing, the action "insert the letter `e` at position 42" does not. Since we can directly observe student behavior only in terms of such actions, it is difficult to break the process of writing a program down into a sequence of meaningful steps.

A catalog of "programming actions" would allow us to represent the process of writing a program as a sequence of generic steps that can be compared between different students. This would help discovering recurring concepts or ideas in programming. The same set of actions could also be used to generate new programs. For example, a tutor could attempt to remove bugs in an incorrect program by applying different possible actions, even if that program has not been encountered before. Creating such a catalog of programming actions is, however, far from trivial.

We propose a data-driven method to learn meaningful actions in programming automatically. The idea is to observe how students write programs, and group related code edits into *rewrite rules* (or simply *rewrites*). For instance, the rewrite rule

```
for (i=1; i<=n; i++)  →  for (i=0; i<n; i++)
```

groups several character-level removals and insertions that together fix a common off-by-one error. This rewrite rule may be used on any program containing the fragment on the left-hand side, by replacing it with the modified version on the right.

Rewrite rules represent generic programming actions. With such actions we can model debugging as search: to fix an incorrect program, search for a sequence of rewrites that transforms it into a correct program. Unlike primitive actions (inserting and removing individual characters),

• *The authors are with the AI Laboratory at University of Ljubljana, Faculty of Computer and Information Science, Ljubljana, Slovenia.*
E-mail: {timotej.lazar,aleksander.sadikov,ivan.bratko}@fri.uni-lj.si.

1. Available at http://allyouneediscode.eu and https://code.org.

rewrite rules provide enough context to significantly constrain the search and make it feasible.

Our method builds catalogs of problem-specific rewrite rules automatically from past student data, while requiring very little language-specific knowledge. We used the discovered rewrite rules to debug student programs in a Prolog tutor. To provide feedback to students, the tutor highlights incorrect parts of the program based on the sequence of rewrites needed to fix it.

We have performed an experiment to show that even this simple, non-verbal feedback can significantly reduce the problem-solving time and the number of incorrect submissions before the student finds a solution. Still, further research is needed to determine how and when to present feedback for optimal learning outcomes. In this paper we focus on describing the technical details of our approach and establishing that it can be useful in the classroom.

The rest of this paper is organized as follows. The next section gives an overview of related work on programming tutors. Section 3 describes how rewrite rules are discovered and used. Section 4 gives a brief overview of the online programming system CodeQ, which was used as a testbed for evaluating our method. Experimental setup and results are presented in section 5. The final two sections discuss results and conclude the paper.

2 Background and Related Work

The Lisp tutor [8], one of the earliest ITSs in general, uses an extensive model of Lisp programming based on the ACT* cognitive theory. The programming process was manually formalized as a large database of productions like

If the goal is to code the body of a function that takes an integer argument
then try integer recursion and set subgoals to plan the base case and the recursive case.

These rules yield a powerful cognitive model that allows the tutor to both track student actions and generate (using a planning algorithm) new programs. Feedback can be produced by analyzing differences between the student's actions and the generated plan.

On the flip side, developing the cognitive model to cover 30-40 hours of instruction took about three person-years [9]. While several successful tutors employ cognitive models for domains such as high-school algebra [10], physics [11], and deductive logic [7], there have been practically no other attempts to create a generative model of programming in terms of goals and plans. The main problems are the complexity of the domain and the large gap between observable user actions and concepts in the cognitive model.

Instead of a generative model, many programming ITSs specify only the features of correct and incorrect solutions. Constraint-based modeling [12] is the most prominent approach, exemplified by the SQL tutor [13]. Given a student-submitted SQL query, the tutor checks it against a set of constraints. Each constraint specifies which submissions it applies to (relevance condition), and the requirements those submissions should meet (satisfaction condition). Like productions in cognitive models, constraints are usually specified as *if-then* rules; for example:

If the `FROM` clause contains the `JOIN` keyword
then it must also contain the `ON` keyword.

Constraints, however, can only be used to describe individual submissions, while the program's evolution from one submission to the next is ignored. This makes constraint-based models easier to develop, and tutors exist for many programming languages including Java [14] and Prolog [15]. For the same reason, constraints are not well-suited for describing the *process* of writing a program.

Regardless of the underlying model, authoring an ITS requires significant knowledge-engineering effort, ranging from tens to hundreds of hours of development time to produce one hour of educational content [16]. Data-driven tutors address this by automatically building and refining their knowledge base.

Several data-driven programming tutors have adapted the Hint Factory approach: they represent the problem domain as a *solution space* of incorrect and correct submissions, and use student data to fill in common transitions between submissions [17], [18], [19]. Like the logic tutor described in the previous section they use the solution-space graph to direct students towards the closest correct submission.

In the programming domain, however, transitions in the solution space usually do not correspond to meaningful problem-solving steps. A transition $s_1 \rightarrow s_2$ between two programs in the solution space means only that a representative number of students first submitted the program s_1 and then s_2. If a program cannot be located in the existing solution space, a tutor cannot analyze it (without, for example, looking for the nearest known submission based on some measure of similarity).

Another problem for data-driven programming tutors is the size of the state space for even the simplest problems. Programming exercises typically have many solutions, stemming from different possible approaches and syntactic and semantic variations [18], [20]. This problem has been addressed using canonicalization [21], for example by renaming variables and rewriting expressions into normal forms. Linkage graphs may be used to find equivalent programs based on dependencies between variables [17]. In both cases, states represent equivalence classes of programs. Another interesting approach finds equivalent code phrases in student submissions based on unit-test results [22].

The Error Model Language [23] has been used to define *correction rules* or transformations for synthesizing new programs, and has been used for debugging student programs. Correction rules encode more information than our rewrite rules, but must be specified by the instructor. Rewrites discovered by our method could help with this specification by serving as a starting point for defining correction rules.

Other languages for specifying transformations, such as Maude [24] and Stratego/XT [25], support more advanced rules for rewriting programs. However, transformations defined in those languages cannot be directly mapped to student actions observed in a programming tutor, as they operate on a higher level than text editing.

3 Debugging with Rewrite Rules

Developing a full cognitive model of programming, capable of generating a working program from scratch, is a complex

TABLE 1
Example Rewrite Rules for the Predicate `rev(A,B)`

#	Path	Original fragment		Replacement	Comment
1	*clause* ▹ *head* ▹ *compound* :	`rev([A,B],[B,A])`	→	`rev([],[])`	Overly complex base case.
2	*clause* ▹ *head* ▹ *compound* :	`rev([A|B],C)`	→	`rev([],[]). rev([A|B],C)`	Missing base case.
3	*clause* ▹ *body* ▹ *and* ▹ *compound* :	`conc(A,B,C)`	→	`conc(A,[B],C)`	Incorrect usage of `conc`.
4	*clause* ▹ *body* ▹ *and* ▹ *binop* :	`A = [B|C]`	→	`conc(B,[C],A)`	Incorrectly appending element `C` to list `B`.
5	*clause* ▹ *body* ▹ *and* :	`rev(A,B), C = [B|D]`	→	`rev(A,B), conc(B,[D],C)`	Incorrectly appending element `C` to list `B`.

task even for human experts. Creating the model automatically, and only in terms of observable student actions, is even more difficult.

That said, our in-class experience shows that solving a programming exercise typically proceeds in two stages. Students write the entire program first, test it, then spend most of the time making relatively small modifications to remove bugs; similar behavior has been noted in [21]. Before the first test, there are usually few or no intermediate versions of the program that make syntactic or semantic sense. This stage is relatively short and serves only to "load" the student's initial conception of the program into the editor.

We focus on the second stage of the process: debugging an incorrect program. During this stage, code modifications are easier to follow because they are usually localized to certain parts of the program. Additionally, the unmodified parts of the program serve as context, allowing us to determine where a particular rewrite may be applied. Modeling the debugging phase of the problem-solving process is thus easier than creating a full-fledged model of programming, while potentially just as useful in a tutoring context.

In the following subsections we describe the two main components of our approach: how to extract rewrite rules from student traces, and the algorithm for debugging student programs using these rules. We also explain how the rewrite sequences discovered by our method may be used to provide feedback in a programming tutor.

Before continuing, let us clarify several terms used throughout this paper. An *attempt* corresponds to one student solving one problem. During each attempt a student can submit several programs; these *submissions* are checked using a predefined set of test cases. A *solution* is a correct program with respect to the given tests, i.e. a program that passes all test cases. Note that a correct program in this sense is not guaranteed to be an absolutely correct solution to the given problem. For an attempt to be successful, the student must submit at least one solution. The sequence of observed user actions during an attempt is called a *trace*.

3.1 Example: Reversing a List

We first explain the structure and application of rewrite rules on the list-reversal problem in Prolog. The goal of this problem is to write the predicate `rev(List,Reversed)`, which reverses the order of elements in `List` to create a new list `Reversed`.

Our method discovered about 50 rewrite rules from student traces for this problem. Table 1 gives some examples. Each rule is of the form *path* : *a* → *b*, where *a* is the original fragment and *b* its replacement. The *path* tells us where in a program's abstract syntax tree (AST) a rule may be applied. For each rewrite rule, the last column briefly describes the mistake it fixes. Since our approach is mostly language-independent, we explain Prolog only briefly in the following subsections. Here we point out that `[]` denotes the empty list, while `[X|L]` denotes the list with head `X` and tail `L`.

Note that rewrite rules use normalized variable names, so that the same rule can be used regardless of actual names chosen by the student. The *a* and *b* parts of a rule are stored as sequences of tokens (such as `rev`, `[` or `List`), allowing us to ignore differences in white space.

Several rules may represent the same modification. Rules 4 and 5 in Table 1 are like this, with the latter rule providing more context on the left-hand side and thus establishing a stronger limit on its applicability. We could prune the ruleset and keep only the most generic version of each rule – that is, the rule with the shortest left-hand side. However, specialized rules are more likely to result in a correct program. Consider a rule with the entire program on the left-hand side: we can be sure that applying such a rule will fix the error. On the other hand, it will only be applicable to that program. We want rules to be as specific as possible while remaining applicable to many programs.

A student might submit the following program as the initial version of the list-reversal predicate:

```
rev([Head|Tail],Reversed) :-
    rev(Tail,ReversedTail),
    conc(ReversedTail,Head,Reversed).
```

This is an attempt at the naive recursive solution [26], which reverses the `Tail` of the original list and appends the `Head`

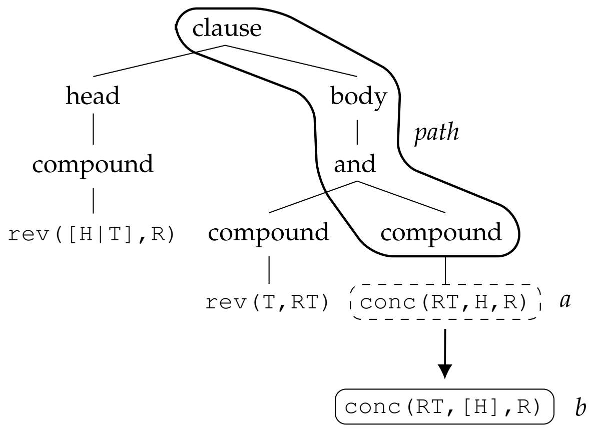


Fig. 1. The AST representing the single Prolog clause with the head `rev([H|T],R)`, and a body consisting of two goals `rev(T,RT)` and `conc(RT,[H],R)`. The original fragment *a* at *path* from AST root is replaced with the new version *b*.

(first) element at the back of `ReversedTail`. The program has two mistakes: (1) the base case is missing (reversing the empty list), and (2) all three arguments to the `conc` predicate should be lists (whereas `Head` is only one element).

Figure 1 shows the AST for this program (simplified by omitting some nodes and shortening variables names). In general, a rewrite rule *path* : *a* → *b* is applicable to any program fragment that matches *a* and can be reached by following *path* from the root of the program's AST.

Only the second and third rewrite rules from Table 1 are applicable to this program. Figure 1 shows the application of the third rule, resulting in the new version (parts different from the previous version are shown in bold)

```
rev([Head|Tail],Reversed) :-
    rev(Tail,ReversedTail),
    conc(ReversedTail,[Head],Reversed).
```

which correctly appends `Head` to the list `ReversedTail`. The missing base-case clause can be added using the second rule, yielding the final, correct program:

```
rev([],[]).
rev([Head|Tail],Reversed) :-
    rev(Tail,ReversedTail),
    conc(ReversedTail,[Head],Reversed).
```

This list-reversal program contains two clauses: the first line states that reversing the empty list `[]` yields again the same list, and the remaining three lines describe how to reverse a nonempty list `[Head|Tail]`.

3.2 Solution Traces

To find rewrite rules, we keep a *trace* for each attempt: the sequence of all observed actions, such as inserting/deleting characters and submitting a program for testing.

Tracking changes at the character level allows us to observe program modifications directly, without having to use a string- or tree-edit distance to extract differences between successive submissions. This way the evolution of each program fragment can be tracked across multiple submissions, independently from other changes to the program.

Fig. 2 shows an example trace for a successful attempt at solving the list-reversal problem. The top (thick) line represents the sequence of actions. Only the four "submit" actions are shown; insertions and deletions that modify the program from one version to the next are omitted.

The initial submission contains several bugs: a missing base case, using the wrong list notation, and incorrectly appending `H` to the end of the reversed list. Due to the missing base case, this program passes no tests.

The second submission fixes the list notation error by replacing `[H,T]` (a list with two elements) with `[H|T]` (a list with the head element `H` and tail `T`). Since the base case is still missing, the modified program also passes no tests.

The next submission adds the base-case clause, allowing the program to pass (only) the simplest tests of reversing the empty list. Finally, the concatenation predicate `conc(A,B,C)` is used to correctly append an element to the list, resulting in the correct implementation of `rev(L,R)`.

3.3 Finding Rewrite Rules

We extract rewrites by tracking how individual program fragments evolve in a student's trace. A *fragment* is any contiguous sequence of tokens in a program. Instead of tracking all possible fragments, we pick only the "interesting" fragments according to the following considerations.

First, a program with n tokens contains $\binom{n+1}{2}$ nonempty fragments. Tracking all possible fragments would result in many "nonsensical" rewrite rules like

```
,A,B),B= → ,[A],B),B=
```

While such rules can be used for debugging, a large catalog of rules means a large branching factor, slowing down the search for a correct program. Second, our goal is to find meaningful transformations that can give us some insight into the programming process. For instance, the rule

```
conc(A,B,C) → conc(A,[B],C)
```

describes the same modification much better.

For these reasons we require the left-hand side of a rewrite represent a complete syntactic unit. Specifically, we consider only fragments corresponding to subtrees of certain non-terminals in the program's AST. In Prolog, we track fragments representing the head of each clause and the goals in its body. For example, the AST in Fig. 1 contains one clause with two goals (subtrees of the "and" node).

We describe the algorithm for extracting rewrites below. In a previously published version we tracked individual lines in the program's code, allowing us to extract rewrites without parsing the program [27]. This paper presents a new version of the algorithm using a more general representation for rewrites: since each line of code corresponds to some fragment, the new approach subsumes the previous version. Additionally, it does not rely on a particular coding style and allows us to use AST paths as context for rewrites. The new algorithm does require syntactically correct programs; we do not consider that a problem, however, since syntactic errors can be handled sufficiently well by the interpreter.

When extracting rewrite rules from a trace, we keep a set F of tracked fragments to follow the evolution of interesting fragments between submissions, and a set R of extracted rules. For every action in the trace we update these sets depending on action type, as follows:

- *Submission*. For each interesting fragment in this submission we add a new item (*a*, *path*, *t*, *start*, *end*) to F, where *a* is the fragment at *path* from AST root spanning character indexes from *start* to *end*, and *t* is the number of tests passed by this submission. We track the fragment's evolution by updating *start* and *end* as characters are inserted and deleted.
  For each item already in F, we check whether the current submission passes more tests than the stored value *t*. If so, we add a new rewrite rule *path* : *a* → *b* to R, where *a* is the stored fragment with AST *path*, and *b* is the modified fragment in the current submission (delimited by the updated indexes *start* and *end* – see the next item).
- *Insertion/deletion*. For every tracked fragment (*a*, *path*, *t*, *start*, *end*) in F we update the indexes *start* and *end* that delimit this fragment. If a character is inserted in front of the fragment, we increment both *start* and *end*; if a character is deleted within the fragment, we only decrement *end*; and so on.

Fig. 2 shows three of the tracked fragments (underlined) from the first two submissions. Several other fragments are

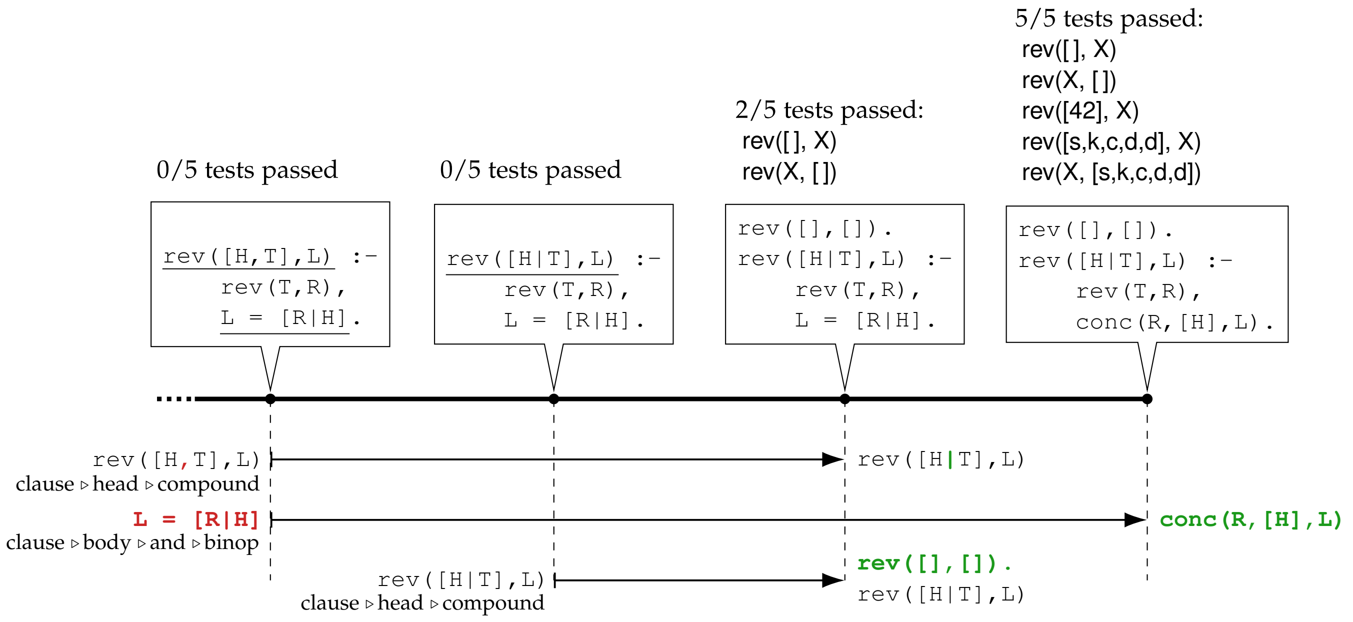


Fig. 2. Sequence of for distinct submissions for an attempt at solving `rev(A,B)`. Each box shows the content of the student's program at one point during the problem-solving process, with the rightmost box showing the final, correct program. Tracking changes to the underlined fragments yields the three shown rewrite rules.

tracked but omitted in the figure for clarity. The arrow for each fragment extends from the submission where it is added to F to the submission where the corresponding rewrite rule is added to R.

For example, in the first submission we add the fragment `rev([H,T],L)` and its AST path to F. While this fragment is fixed by the second submission, no rewrite is added at that point because the program still passes no tests (due to a missing base case).

We keep tracking both marked fragments, and also add (among others) the new fragment `rev([H|T],L)`. The third submission passes two tests, so we add rewrites for the modified fragments to R. Similarly, the final submission modifies the second fragment from the initial version (and passes more tests), so we add the corresponding rewrite.

By directly observing insertions and deletions, we are able to follow modifications to each part of the program independently. This approach also allows for overlapping fragments that modify the same part of a program, as is the case for the first and third fragments in Fig. 2.

We could add rewrites for every submission, regardless of how many tests it passes, or even for program versions between submissions. We found, however, that doing so yields many invalid rules that are more likely to break a program further than fix it. By only considering improved submissions (based on the number of passed test cases), discovered rules are more likely to be useful for debugging. That said, the number of passed test cases is only a rough measure of correctness. Finding appropriate "checkpoints" at which to consider rewrites is an interesting topic that would benefit from further research.

3.4 Rewrite Probabilities

After we have extracted rewrites from all traces for a problem, we associate a probability with each rewrite to guide the debugging algorithm described in the next section. This probability describes how likely a rewrite $path: a \rightarrow b$ is used in a program that contains the fragment a at $path$ from the root of program's AST.

Specifically, we calculate the conditional probability of using a rewrite $a \rightarrow b$ when the program contains the fragment a (the AST path must also match, but we omit it here for clarity) as:

$$p(a \rightarrow b|a) = \frac{\text{\# of traces containing } a \rightarrow b}{\sum_x \text{\# of traces containing } a \rightarrow x}.$$

We wish to avoid assigning very high or very low probabilities to rewrite rules. If the probability of a rewrite is too low, it will rarely or never be attempted during debugging; on the other-hand, very high-probability rewrites can prevent less common alternatives from being explored.

For this reason we compress the range of probabilities using the logistic function with steepness $k = 3$ and the average probability $\bar{p} = \text{avg}(p)$ as the midpoint. We calculate the final value p' for each probability p as:

$$p' = \frac{1}{1 + e^{-k(p-\bar{p})}}.$$

This ensures that the values of p' are approximately between 0.2 and 0.8, while leaving probabilities close to the average unchanged. The function and parameters were chosen ad hoc; they performed well in our evaluations, but better options might exist.

3.5 Debugging

We formalize the task of debugging an incorrect program as a search for an appropriate sequence of rewrites. We keep a priority queue of generated programs. In every iteration we test the highest-priority program in the queue; if it is correct, we return it along with the corresponding rewrite sequence.

Otherwise, we use applicable rewrite rules to generate new programs. The algorithm is outlined below.

Input: incorrect program P_0, catalog of rewrite rules R
Output: correct program with associated rewrite sequence

```
let $Q$ be the empty priority queue
let $S_0$ be the empty rewrite sequence
add $(P_0, S_0)$ with priority 1 to priority queue $Q$
while $Q$ not empty do
    pop $(P, S)$ with highest priority $c$ from $Q$
    if $P$ is correct then
        return $(P, S)$
    for all $r \in R$ do
        if rule $r$ is applicable to $P$ then
            apply $r$ to $P$ to get new program $P'$
            append $r$ to $S$ to get new rewrite sequence $S'$
            add $(P', S')$ to $Q$ with priority $c * p(r)$
                /* $p(r)$ is defined in previous section */
```

Essentially, this algorithm performs a best-first search guided by rewrite-rule probabilities. Specifically, we define the "probability" of a sequence of rewrites $r_1 r_2 \ldots r_n$ as the product of the probabilities of individual rewrites:

$$p(r_1 r_2 \ldots r_n) = \prod_{i=1}^{n} p(r_i).$$

The algorithm thus first visits programs resulting from highest-probability rewrite sequences. This heuristic is based on the assumption that rewrites that were used in more traces are more likely to reflect successful problem-solving strategies. By using the product of probabilities, we also implicitly prefer shorter rewrite sequences.

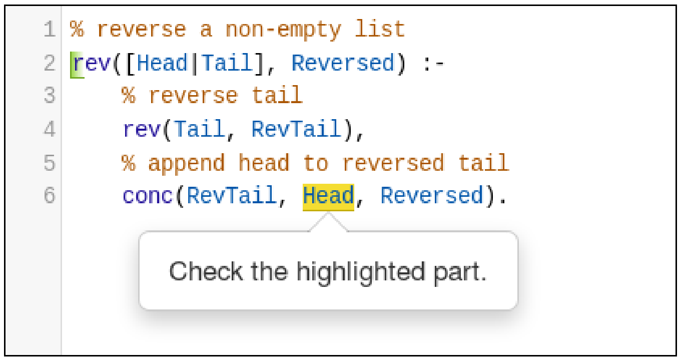
Since we use this algorithm in an interactive application, we terminate the search after some time if no solution is found. For this study we used a timeout of three seconds. An ordinary desktop computer can generate and test between a few ten and a few hundred programs in that time, with testing being (by far) the most expensive operation.

Using a prototype version of the described algorithm we were able to fix approximately 60 percent of incorrect programs [27]. For over 80 percent of the programs we can fix, fewer than a hundred new programs had to be generated before a corrected version was found. Besides improving the representation of rewrites, this paper extends our prior work with an experimental evaluation of rewrite-based hints in the classroom.

3.6 Feedback to Student

Once a sequence of rewrites has been found that fixes an incorrect program, it must be presented to the student in a way that is conducive to learning. Showing the exact rewrites required would constitute a *bottom-out* hint, which would remove an opportunity to practice debugging.

In the present study we wished to determine whether useful feedback can be served without any predefined knowledge (by using feedback templates or similar). To generate hints from a sequence of rewrites we therefore simply mark program fragments that are touched by one or more rewrites. We distinguish three cases and highlight them



```
% reverse a non-empty list
rev([Head|Tail], Reversed) :-
    % reverse tail
    rev(Tail, RevTail),
    % append head to reversed tail
    conc(RevTail, Head, Reversed).
```

Fig. 3. Incorrect program with automatic highlights. The first (green) highlight indicates the position where another clause must be inserted, while the second (yellow) highlight points out an incorrect argument to the `conc` predicate.

with different colors: adding (green), removing (red), and modifying (yellow) fragments. Fig. 3 shows automatically generated highlights for the example in Fig. 1.

Note that we only highlight existing fragments that are actually modified. The green highlight corresponding to the rewrite

```
rev([A|B],C) → rev([],[]). rev([A|B],C)
```

thus only indicates the position where a new clause should be inserted — at the beginning of the program. The second rewrite

```
conc(A,B,C) → conc(A,[B],C)
```

actually corresponds to two insertions (placing a bracket on either side of the variable `Head`). However, we show insertions that are very close together as one "modify" (yellow) highlight instead.

4 CodeQ

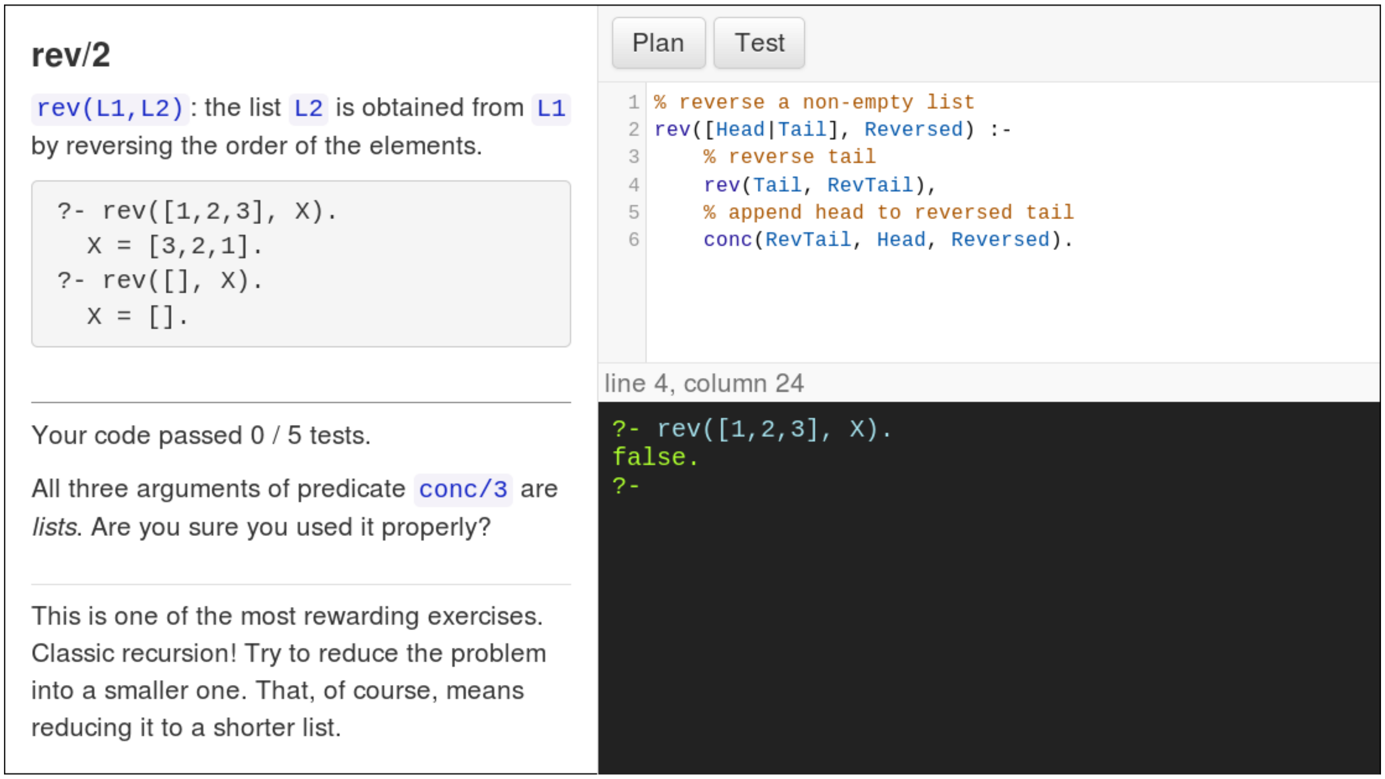
We evaluated the effect of automatically generated hints using CodeQ, a free web application for solving programming exercises.[2] CodeQ provides an integrated online environment for writing and running programs in Prolog and Python.

A web application eliminates the overhead associated with solving programming exercises, such as installing an interpreter and loading source files, or learning to use an advanced IDE. Students log in, select a problem to solve and can immediately start coding; at any time they can log out and resume their attempt later. Many students pointed out these features when asked what they like about the system.

Fig. 4 shows the main screen for the Prolog list-reversal problem. Problem description is given on the left-hand side and includes one or more examples of correct program behavior. Feedback from the tutor is displayed below. The right-hand side contains a code editor and an interactive prompt for submitting queries to Prolog. The current program is loaded into the Prolog engine automatically with every query. Students can request feedback from the tutor using the buttons above the editor.

The *Plan* button provides additional advice when the student is unsure about how to approach a problem. Plan

2. Available online at https://codeq.si/. Source code is available under AGPL3+ at https://codeq.si/code/.



rev/2

`rev(L1,L2)`: the list `L2` is obtained from `L1` by reversing the order of the elements.

```
?- rev([1,2,3], X).
  X = [3,2,1].
?- rev([], X).
  X = [].
```

Your code passed 0 / 5 tests.

All three arguments of predicate `conc/3` are *lists*. Are you sure you used it properly?

This is one of the most rewarding exercises. Classic recursion! Try to reduce the problem into a smaller one. That, of course, means reducing it to a shorter list.

```
% reverse a non-empty list
rev([Head|Tail], Reversed) :-
    % reverse tail
    rev(Tail, RevTail),
    % append head to reversed tail
    conc(RevTail, Head, Reversed).
```

```
?- rev([1,2,3], X).
false.
?-
```

Fig. 4. CodeQ problem-solving screen. The left side gives a description of the problem with examples of correct behavior. Feedback in response to *Plan* and *Test* buttons is output below. On the right there is a code editor and an interpreter for running queries.

messages are written by the instructor and are included in problem definitions. In Fig. 4, a plan has been requested and shown (bottom paragraph in the left column). No limits are imposed on requesting plans, but students are encouraged to do so only as a last resort when stuck on a problem.

A program may be submitted at any time using the *Test* button. As in most programming tutors, testing is done by checking program outputs on a predefined set of inputs. CodeQ responds with the number of test cases the program answered correctly. If additional feedback is available, a *Hint* button appears along with the test results. This allows students to decide whether to ask the tutor for help, or try finding the error on their own. In this example, the hint has already been requested and shown for the program from the previous section.

Feedback is taken from several sources. Any syntax errors reported by the interpreter are relayed to the student directly. For every problem, a hint function may be defined manually to look for typical mistakes. If the hint function is not defined or finds no bugs, CodeQ attempts to debug the program automatically using our method.

Highlights for the program in Fig. 3 have been been generated from automatically discovered rewrites. For the same program, the manually coded hint function returns the following message (also shown in Fig. 4):

> "All three arguments to the `conc` predicate must be *lists*. Are you sure you have used it properly?"

The manual hint function reports a single mistake each time. Once it is fixed, further errors are reported. The order in which errors are detected and reported is determined by the teacher when defining the hint function. After correcting the call to `conc` in this program, the next message would be:

> "The base case appears to be missing. Which list is the easiest to reverse?"

While automatic hints highlight fragments in the code editor, manual hints appear as explanations below the problem description. We discuss the differences and potential uses for the two kinds of hints in Sect. 6.1.

5 Evaluation

Previously we have demonstrated that our method is able to generate hints for many incorrect programs [27]. With this study we wished to determine whether feedback – coded manually or generated automatically – is actually helpful to students. To this end, we evaluated CodeQ in the usual classroom setting. On the one hand, performing the experiment during regular lab sessions limited somewhat our design options. On the other hand, evaluating a tutoring system in a real-world situation should provide the most pertinent results.

In this experiment we considered the time needed to solve each problem, and the number of incorrect submissions made before reaching a solution. We found that the availability of either automatic or manually coded hints significantly reduced both measures, allowing students to solve more problems in the same amount of time. Existing research suggests that test achievement is strongly related to the number of problems a student has solved, with the nature of feedback messages playing a secondary role [28].

We thus establish that our method can serve a useful role in a programming tutor. While hints based on rewrite rules helped the students perform better, these experiments do not tell us in what way hints improved the students' understanding. How to generate and present feedback to optimize learning therefore remains an open question.

5.1 Experimental Design

We performed the experiment during three regular Principles of Programming Languages lab sessions in the spring semester of 2016, at the Faculty of Computer and Information Science, University of Ljubljana. The purpose of lab sessions in this course is to familiarize students with Prolog programming. At the beginning of each session the instructor explained new concepts (Prolog basics with recursion, lists, and arithmetic) and showed a solution to a sample problem on the whiteboard; the same explanation was also available in written form for reference. Students then solved exercises for the remainder of the session.

A total of 119 students were enrolled in the course. Students who have taken the course before (without passing the final exam), exchange students and those who enrolled after the class had started were excluded from the study, leaving 76 participants. They were randomly assigned to three groups: no hints, automatic hints only, and manual hints only. All students received test results, feedback about syntax errors and had the option of using the *Plan* button. To ensure the groups were balanced we controlled for the average grade received on exams in the first-year (for all classes, and only for programming classes). Table 2 shows details about the groups.

TABLE 2
Experimental Groups

Group	N	Average grade (<6 = fail, 10 = best)			
		All exams		Programming exams	
No hints	25	$\mu = 7.94$	$\sigma = 0.79$	$\mu = 7.90$	$\sigma = 0.98$
Automatic	26	$\mu = 7.92$	$\sigma = 0.80$	$\mu = 7.90$	$\sigma = 1.19$
Manual	25	$\mu = 7.92$	$\sigma = 0.84$	$\mu = 8.08$	$\sigma = 1.10$

The lab sessions covered nine problems from the *Family relations* group and 18 problems from the *Lists* group. Four of those problems were either new this year (with no data available for automatic hints) or were solved by the teacher as examples; we exclude these problems from our analysis.

Students solved problems in the CodeQ programming environment. Those who did not wish to participate in the study could use regular SWI-Prolog or create an anonymous account. Because the study was done during regular classes, a teacher was available for help. Students were however encouraged to solve problems on their own for the duration of the study, consulting hints when necessary. We marked cases where a student was unable to finish an exercise without teacher's help and exclude such attempts from the analysis. There were 30 attempts with teacher intervention, out of 1,221 attempts in total. All hints were enabled after the first three lab sessions so as to minimize the potential disadvantage for students in the *no hints* group.

5.2 Data Set

We have collected solution traces for the past three iterations of the same course (spring semesters in the years 2013 to 2015) using an online application similar to a simplified version of CodeQ. These data were used to learn rewrite rules. On average our method discovered 42 rewrite rules per problem.

Altogether 355 students have attended the course over the past three years. Of these, 254 students have solved ten or more problems using the online environment. Overall we have collected traces for 4,649 successful solutions to problems considered in this study, for an average of 202 solutions per problem. The distribution of attempts across problems is similar to this year's, shown in Fig. 5 in the next subsection. The number of students attempting to solve each problem drops to a half by the end of the third session, and keeps decreasing for the remainder of the course.

5.3 Results

Fig. 5 graphs the number of students attempting and solving each exercise during the study. Problem difficulty increases overall and within each group; both trends can be clearly seen in the graph. The drop in the number of attempts is especially noticeable after the first session; this is probably due to students who come in only once to check out the class. We observed no difference in attrition rates between the three experimental groups.

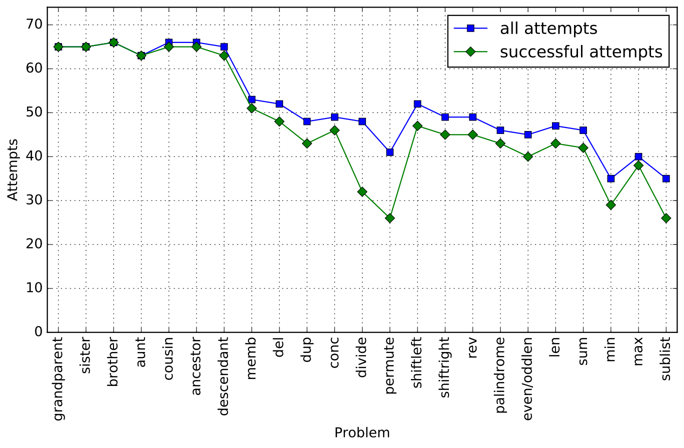


Fig. 5. Number of students attempting and solving each problem.

Other authors have noted the high variability of submissions characteristic for the programming domain [18], [22]. Our experiment confirms this. After normalizing submitted programs by removing superfluous white space and renaming variables, 2,978 distinct correct and incorrect submissions were observed across all attempts (including unsuccessful attempts without a correct submission). Only 228 of those programs have been submitted by more than one student. Fig. 6 shows how many students submitted individual programs. Note that the x-axis uses a logarithmic scale; even so, the long tail of unique submissions remains apparent.

Table 3 breaks down successful solutions by problem. The second column (Time) shows the average solving time for each problem, defined as the sum of time deltas between successive actions. We only consider actions before a correct solution is reached (sometimes students experiment with the code after it passes all tests – we ignore such actions).

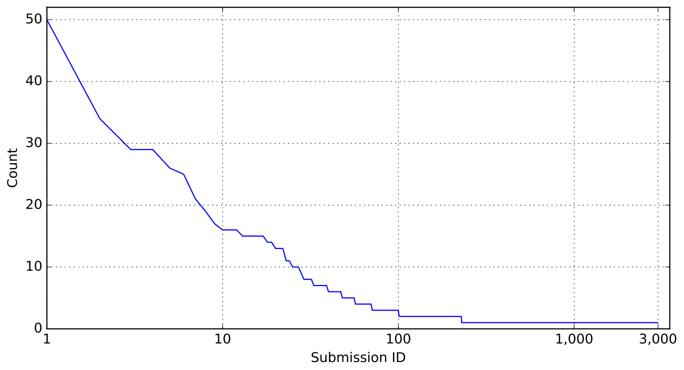


Fig. 6. Number of students submitting each distinct program.

Time deltas are capped at 15 minutes; if a student is idle longer than that, we consider them to have gone off-task. The third column (Subs.) shows the average number of incorrect programs submitted before a submission passed all tests.

The remaining columns show the average number of hints offered (by displaying the *Hint* button after an incorrect program is submitted) during one attempt in the two hint groups, and the percentage of those hints that were actually viewed (the student pressed the *Hint* button). Due to an unfortunate oversight, these data are not available for the first week of the study. Exercises in the first group serve as an introduction to Prolog and are not very difficult, evidenced by the low number of incorrect submissions.

Table 3 shows that many more hints were offered to the manual group. For most problems, one or more generic "catch-all" hints were defined, which would always trigger when more specific feedback was not available. Such hints present general instructions, for example "check that the recursive rule is correctly implemented". This probably contributed to the fact that the manual group consulted a significantly lower percentage of hints – while the *automatic* group viewed 73 percent of all offered hints, the *manual* group only viewed 49 percent of hints.

To see whether hints help students solve problems, we measured the time and number of distinct incorrect submissions before a correct program was submitted. Since problems vary in difficulty, we normalized both values to the average across all attempts for each problem. Table 4 shows the average and standard deviation for problem-solving time and number of incorrect submissions.

Overall, students receiving no hints needed $T_0 = 1.15$ times as long as the average to solve a problem, while students receiving either automatic or manual hints needed $T_A = T_M = 0.91$ as long. Availability of hints also reduced the number of submissions required before reaching a solution. Students in the *no hints*, *automatic* and *manual* groups submitted $S_0 = 1.18$, $S_A = 0.94$ and $S_M = 0.84$ as many distinct incorrect programs as the average.

Solving times and submission counts are not distributed normally, so we used the Kruskal–Wallis H-test to determine the significance of our results. Statistically significant results in Table 4 are marked with * ($p < 0.05$) or ** ($p < 0.01$). We found a significant difference between T_0 versus T_A and T_M, and between S_0 versus S_A and S_M. The difference in solving time between none (T_0) and automatic (T_A) groups is significant at the level of $p < 0.01$. The same holds for the difference in the number of submissions between none (S_0) and manual (S_M) groups.

TABLE 3
Average Time, Incorrect Submissions, and Hints per Attempt

Problem	Time (s)	Subs. (#)	Automatic hints Offered	Automatic hints Viewed	Manual hints Offered	Manual hints Viewed
grandparent	99	0.3	n/a		n/a	
sister	379	3.0	n/a		n/a	
brother	92	0.7	n/a		n/a	
aunt	191	0.8	n/a		n/a	
cousin	433	2.1	n/a		n/a	
ancestor	255	1.0	n/a		n/a	
descendant	223	1.1	n/a		n/a	
memb	604	2.1	1.10	64%	2.38	42%
del	788	3.8	1.62	71%	4.83	79%
dup	987	5.4	0.57	100%	7.60	62%
conc	1174	4.5	2.07	48%	0.91	60%
divide	1188	6.8	2.78	52%	7.00	57%
permute	1168	4.4	1.40	71%	4.67	54%
shiftleft	566	2.4	1.25	60%	2.15	46%
shiftright	595	2.6	0.67	67%	2.44	50%
rev	706	3.4	1.67	73%	3.38	19%
palindrome	435	3.4	2.40	88%	3.11	54%
even/oddlen	357	1.9	1.20	83%	3.00	56%
len	241	1.9	0.71	80%	2.38	32%
sum	124	0.8	1.25	80%	2.67	25%
min	625	2.7	0.75	100%	2.86	45%
max	96	0.8	1.00	67%	2.40	67%
sublist	705	3.7	0.75	67%	3.00	43%

TABLE 4
Relative Time and Number of Submissions Until Solution

Group	Solving time (T)		Submission count (S)	
No hints	$\mu = 1.15$	$\sigma = 1.21$	$\mu = 1.18$	$\sigma = 2.01$
Automatic	$\mu = 0.91$**	$\sigma = 0.73$	$\mu = 0.94$*	$\sigma = 1.48$
Manual	$\mu = 0.91$*	$\sigma = 0.74$	$\mu = 0.84$**	$\sigma = 1.29$

5.4 User Survey

After the experiment we conducted a survey to see how well CodeQ was received by the students. The survey consisted of four scaled questions, and three optional open-ended questions asking for comments and suggestions about the system. Table 5 shows mean responses to the scaled questions for each experimental group. In the last two questions, "feedback" refers to automatic or manual hints, test results, syntax errors, and planning messages. Note that the best answer for the last question is 1.

There is some variation in student responses across the three groups. However, none of the differences approaches statistical significance. Nearly all students answered 4 or 5 to the first two questions. Responses to questions 3 and 4 show that verbal feedback is more useful and easier to understand than simple highlights. This is expected, as manually written hints explain the problem and point to a solution, whereas automatic hints only highlight the

TABLE 5
Mean responses to the post-experiment survey (1 = no, 5 = yes)

Question	Group		
	None	Automatic	Manual
1. Did you find CodeQ easy to use?	4.69	4.70	5.00
2. Did CodeQ help you learn Prolog?	4.69	4.70	4.88
3. Did you find the feedback useful?	4.08	3.90	4.43
4. Was the feedback ever unclear?	2.46	3.20	2.43

problematic areas. The student is left the non-trivial task of understanding the error.

The open-ended questions asked about which aspects of CodeQ the students found most useful, and what could be improved. Positive comments mainly related to ease of use afforded by an integrated online application – no installation is required, programs are automatically loaded into Prolog interpreter, and per-problem test cases allowing students to easily determine whether a solution is correct.

Suggested improvements mainly concerned usability problems in the current version of the application. Most commonly raised issues include: cumbersome access to solutions to completed problems, no indication which test case(s) have failed, and the limited functionality of the Prolog engine compared to a locally installed interpreter.

6 Discussion

Our results indicate that highlighting erroneous code fragments can indeed serve as useful feedback that helps students find and correct errors. This finding agrees with our in-class experience, where we often observed students having difficulty locating problematic parts in a misbehaving program. Highlights direct their debugging efforts and reduce the time needed to discover the mistake. Furthermore, highlighting incorrect fragments also indicates which parts of the program are already correct, providing a degree of assurance that the student is on the right path.

Somewhat surprisingly, hand-written explanations received by the *manual* group did not reduce the average problem-solving time any more than automatic hints. A possible reason is that these explanations require some time to read and understand (and the student still needs to actually locate the error) which would increase the total solving time.

On the other hand, students in the *manual* group did submit fewer incorrect programs before finding a solution, though the difference was not statistically significant. This is likely because manual hints actually explained errors and directed the student towards a solution, while automatic hints only highlighted suspect fragments. Students still had to figure out what the error was and how to fix it, and often needed more than one attempt to do so.

We observed that some students resort to tinkering when faced with a buggy program – making small modifications to the program in hope of stumbling onto a solution. Automatic testing and hints might have either positive or negative effects in such cases: they can serve as a starting point to motivate a more systematic approach to debugging, or they can encourage random tinkering by limiting the range and number of variations a student has to try. Such undesired behavior may be discouraged by appropriate prompts from the tutor [29]. For example, a tutor could advise students to run their programs in the interpreter and suggest relevant inputs to try. These problems are however out of scope for the present paper.

Like previous studies, our experiment has also demonstrated the huge variability of student programs. For instance, the canonical solution for the problem `palindrome` uses the list-reversal predicate `rev` and fits in a single line:

```
palindrome(L) :- rev(L,L).
```

Almost 50 students attempted this problem, and over 40 submitted a working program. In total we received – after normalizing white space and variable names – 166 distinct submissions. Only five of those programs were submitted by more than one student, and the remaining 161 programs were unique (i.e. appeared in a single attempt). These results are particularly surprising because the problem `rev` directly precedes `palindrome`.

Most data-driven programming tutors generate feedback for an incorrect program based on what previous students did after submitting the same program. This approach fails when a program is encountered that has not been observed before. This problem may be alleviated by using canonicalization, or simply by collecting many solution traces.

Using rewrite rules, however, we can debug novel programs without relying on language-specific transformations. In this sense, rewrites represent generic programming actions that are not tied to specific submissions. Since rewrites are considered locally, we were able to extract from only a few hundred attempts rewrite rules that allow us to fix many common programming errors.

We have shown that even simple non-verbal feedback can be effective. Such feedback can be generated by a conceptually simple method, requiring very little language-specific knowledge. No canonicalization is done beyond renaming identifiers, which can be done in a generic fashion by adding scope information to the parser.

Regardless of hints, we observed that students tend to persist longer and solve more problems when working in the online environment compared to the traditional setup (using an ordinary text editor and Prolog interpreter). Based on responses to the open-ended survey questions this can mostly be attributed to the testing functionality. Test cases provide a clear goal for the student, and the system is able to confirm when their program is correct.

We have shown that both manual and automatic hints affect students' problem-solving. However, further experiments are required in order to determine the extent of these effects and to better understand how different kinds of hints influence learning. To measure learning gains directly, students' skills should be tested before and after a tutoring session in a controlled environment, with similar problems used both for tutoring and testing.

6.1 Future Directions

While our implementation of automatic debugging in a programming tutor demonstrates the usefulness of rewrite rules, many questions remain open for further research. We outline some of the main ideas here.

When debugging a program, no heuristic is used to estimate the distance to a solution. A crude approach would be to estimate the correctness of a program from the number of passed tests, or the syntactic distance to the nearest solution. While many features could be defined to better approximate distance to solution (e.g. number of clauses and goals, or the use of recursion), finding a language-independent heuristic for guiding the debugging process is an interesting challenge.

Most examples of incorrect programs in this paper contain bugs that can be localized to a single fragment. More involved bugs require two or more rewrites to fix. A common example in Prolog is placing a negated goal too early in the program. Fixing this bug requires two rewrites: removing the goal from the current location and reinserting it at a later point. While our debugging method applies each rewrite independently, it could be extended to consider common *sequences* of rewrites.

Rewrite rules are extracted separately for each problem. By comparing rules between problems we could discover common misconceptions, allowing us to categorize programming problems according to typical errors. This would enable a tutor to recommend problems based on errors made by a particular student, implementing the *outer-loop* functionality of an ITS [4].

Manually coding hints for CodeQ was a challenging task – despite many years of teaching experience, it is far from trivial to recall all the mistakes students make. By grouping incorrect programs according to sequences of rewrites required to fix them, we could automatically generate a list of common mistakes for each problem, together with actual examples of corresponding student submissions. This information would be very helpful when manually authoring feedback. Another option would be for instructors to annotate each rewrite (or combination of rewrites) with explanatory messages for students when that rewrite is used to fix their program.

Generated rewrite sequences were only used to highlight problematic code fragments. In order to provide explanations in natural language, some domain knowledge is required. Templates could for example be used to generate feedback based on the tokens touched by the sequence of rewrites – for example, "variable `Head` should not appear in the recursive call to `rev`".

To compare the efficacy of automatic and manual hints we have limited feedback in each experimental group to one or the other type of hints. The two types differ greatly in content and presentation, however: automatic hints help with locating bugs, while manual hints focus on explaining the error and leave it up to the student to find it. Now that we have established that hints can be useful, we will try to determine in which situations each type can provide the most benefit.

For example, some students might prefer explanations, which provide some clues but still allow the students to discover the concrete errors themselves. Other, less motivated students would perhaps be better served by highlights, which provide an immediate starting point for debugging and might keep them from getting frustrated. Finally, both types of hints could be provided in tandem, such as by only showing the explanation at first, and then displaying the highlights on demand.

We have implemented our approach first for Prolog in order to take advantage of available problem-solving data. Our method can easily be ported to other programming languages. Further testing is needed, however, to determine how well it would perform for other commonly taught languages such as Python or Java, where programs tend to be longer and more variable.

7 Conclusion

We have described a novel formalization of programming based on rewrite rules. Our approach represents the process of writing a program as a sequence of atomic operations or rewrites. Based on this model we developed a method for automatic program debugging and incorporated this method into an online programming environment.

Through a limited in-class study we have shown that useful feedback can be generated in a programming tutor without using language-specific techniques like canonicalization. Even the rudimentary highlights presented to the *automatic* group had a significant effect on the time needed to reach a solution. How to use rewrite rules to generate and present feedback in an optimal way remains an open question for future research.

We believe the most interesting contribution of this paper is the method for automatically extracting useful rewrite rules from programmers' traces. This can be viewed as a way of automatically modeling human thinking when writing programs.

Rewrite rules allow us to follow student actions directly, while still carrying more meaning than character-level actions. Rewrite rules are thus the generic terms that can be used to compare actions from different students' traces. Despite the rudimentary nature of these rules – they operate on the level of text, with no awareness of language semantics – we therefore believe they have the potential to shed some light on the cognitive processes in programming.

Acknowledgments

This work was partially funded by Slovenian Research Agency (ARRS) under a Young Researcher grant and the project J2-4222 (C), and by EU Structural Funds and Slovenian Ministry for Education, Science and Sport under the project CodeQ.

References

[1] B. S. Bloom, "The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring," *Educational researcher*, vol. 13, no. 6, pp. 4–16, 1984.

[2] K. VanLehn, "The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems," *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011.

[3] J. Self, "The defining characteristics of intelligent tutoring systems research: ITSs care, precisely," *International Journal of Artificial Intelligence in Education*, vol. 10, pp. 350–364, 1998.

[4] K. VanLehn, "The behavior of tutoring systems," *International Journal of Artificial Intelligence in Education*, vol. 16, no. 3, pp. 227–265, 2006.

[5] T. Murray, "An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art," in *Authoring tools for advanced technology learning environments*, T. Murray, S. Blessing, and S. Ainsworth, Eds. Springer Netherlands, 2003, ch. 17, pp. 491–544.

[6] K. R. Koedinger, E. Brunskill, R. S. J. d. Baker, E. A. McLaughlin, and J. Stamper, "New potentials for data-driven intelligent tutoring system development and optimization," *AI Magazine*, vol. 34, no. 3, pp. 27–41, 2013.

[7] T. Barnes and J. Stamper, "Toward automatic hint generation for logic proof tutoring using historical student data," in *Proc. 9th Int'l Conf. Intelligent Tutoring Systems (ITS 08)*, 2008, pp. 373–382.

[8] J. R. Anderson, F. G. Conrad, and A. T. Corbett, "Skill acquisition and the LISP tutor," *Cognitive Science*, vol. 13, no. 4, pp. 467–505, 1989.

[9] J. R. Anderson and E. Skwarecki, "The automated tutoring of introductory computer programming," *Communications of the ACM*, vol. 29, no. 9, pp. 842–849, 1986.

[10] K. R. Koedinger, J. R. Anderson, W. H. Hadley, and M. A. Mark, "Intelligent tutoring goes to school in the big city," *International Journal of Artificial Intelligence in Education*, vol. 8, no. 1, pp. 30–43, 1997.

[11] K. VanLehn, C. Lynch, K. Schulze, J. A. Shapiro, R. Shelby, L. Taylor, D. Treacy, A. Weinstein, and M. Wintersgill, "The Andes physics tutoring system: Lessons learned," *International Journal of Artificial Intelligence in Education*, vol. 15, no. 3, pp. 147–204, 2005.

[12] A. Mitrovic, "Fifteen years of constraint-based tutors: what we have achieved and where we are going," *User Modeling and User-Adapted Interaction*, vol. 22, no. 1-2, pp. 39–72, 2012.

[13] ——, "An intelligent SQL tutor on the web," *International Journal of Artificial Intelligence in Education*, vol. 13, no. 2, pp. 173–197, 2003.

[14] J. Holland, A. Mitrovic, and B. Martin, "J-LATTE: a constraint-based tutor for Java," in *Proc. 17th Int'l Conf. Computers in Education (ICCE 2009)*, 2009, pp. 142–146.

[15] N.-T. Le and W. Menzel, "Using weighted constraints to diagnose errors in logic programming – the case of an ill-defined domain," *International Journal of Artificial Intelligence in Education*, vol. 19, no. 4, pp. 381–400, 2009.

[16] J. T. Folsom-Kovarik, S. Schatz, and D. Nicholson, "Plan ahead: Pricing ITS learner models," in *Proc. 19th Behavior Representation in Modeling & Simulation Conference*, 2010, pp. 47–54.

[17] W. Jin, T. Barnes, J. Stamper, M. J. Eagle, M. W. Johnson, and L. Lehmann, "Program representation for automatic hint generation for a data-driven novice programming tutor," in *Proc. 11th Int'l Conf. Intelligent Tutoring Systems (ITS 12)*, 2012, pp. 304–309.

[18] C. Piech, M. Sahami, J. Huang, and L. Guibas, "Autonomously generating hints by inferring problem solving policies," in *Proc. 2nd ACM Conference on Learning @ Scale (L@S 2015)*, 2015, pp. 195–204.

[19] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor," *International Journal of Artificial Intelligence in Education*, pp. 1–28, 2015. [Online]. Available: http://dx.doi.org/10.1007/s40593-015-0070-z

[20] J. Huang, C. Piech, A. Nguyen, and L. Guibas, "Syntactic and functional variability of a million code submissions in a machine learning MOOC," in *Proc. Workshops 16th Int'l Conf. Artificial Intelligence in Education (AIED 13)*, 2013, pp. 25–32.

[21] K. Rivers and K. R. Koedinger, "Automatic generation of programming feedback: A data-driven approach," in *Proc. Workshops 16th Int'l Conf. Artificial Intelligence in Education (AIED 13)*, 2013, pp. 50–59.

[22] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: scalable homework search for massive open online programming courses," in *Proc. 23rd Int'l World Wide Web Conf. (WWW 14)*, 2014, pp. 491–502.

[23] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[24] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of maude," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65–89, 1996.

[25] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of computer programming*, vol. 72, no. 1, pp. 52–70, 2008.

[26] J. Hong, "Guided programming and automated error analysis in an intelligent Prolog tutor," *International Journal of Human-Computer Studies*, vol. 61, no. 4, pp. 505–534, 2004.

[27] T. Lazar and I. Bratko, "Data-driven program synthesis for hint generation in programming tutors," in *Proc. 12th Int'l Conf. Intelligent Tutoring Systems (ITS 14)*, 2014, pp. 306–311.

[28] A. T. Corbett and J. R. Anderson, "Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2001, pp. 245–252.

[29] R. Baker, K. R. Koedinger, A. T. Corbett, A. Z. Wagner, S. Evenson, I. Roll, M. Naim, J. Raspat, and J. E. Beck, "Adapting to when students game an intelligent tutoring system," in *Proc. 8th Int'l Conf. Intelligent Tutoring Systems (ITS 06)*, 2006, pp. 392–401.

Timotej Lazar received the MSc degree in computer science and mathematics from the University of Ljubljana in 2012. He is currently pursuing a PhD degree in computer science as a junior researcher at the AI Laboratory, Faculty of Computer and Information Science, University of Ljubljana. His research interests include applied artificial intelligence and computer science education.

Aleksander Sadikov received the PhD degree in computer science from the University of Ljubljana. He is a lecturer at the Faculty of Computer and Information Science, University of Ljubljana, where he is a member of the AI Laboratory. His research interests include applications of artificial intelligence in medicine, decision support systems, recommender systems, heuristic search, and games.

Ivan Bratko is a professor of computer science at the Faculty of Computer and Information Science, Ljubljana University, Slovenia, where he heads the AI Laboratory. He has conducted research in several areas of Artificial Intelligence: machine learning and data mining, knowledge-based systems, qualitative modelling, intelligent robotics, intelligent tutoring systems, heuristic search, and computer chess. He has published over 200 scientific papers and books, including Prolog Programming for Artificial Intelligence (4th edition: Pearson Education 2012). He is Fellow of EurAI, and member of SASA (Slovene Academy of Sciences and Arts) and Academia Europaea.