Automatic extraction of AST patterns for debugging student programs

Timotej Lazar, Martin Možina, Ivan Bratko

University of Ljubljana, Faculty of Computer and Information Science, Slovenia

Abstract. When implementing a programming tutor it is often difficult to manually consider all possible errors encountered by students. An alternative is to automatically learn a bug library of erroneous patterns from students' programs. We propose abstract-syntax-tree (AST) patterns as features for learning rules to distinguish between correct and incorrect programs. We use these rules to debug student programs: rules for incorrect programs (buggy rules) indicate mistakes, whereas rules for correct programs group programs with the same solution strategy. To generate hints, we first check buggy rules and point out incorrect patterns. If no buggy rule matches, we use rules for correct programs to recognize the student's intent and suggest missing patterns. We evaluate our approach on past student programming data for a number of Prolog problems. For 31 out of 44 problems, the induced rules correctly classify over 85% of programs based only on their structural features. For approximately 73% of incorrect submissions, we are able to generate hints that were implemented by the student in some subsequent submission.

Keywords: Programming tutors · Error diagnosis · Hint generation · Abstract syntax tree · Syntactic features

1 Introduction

Programming education is becoming increasingly accessible with massive online courses. Since thousands of students can attend such courses, it is impossible for teachers to individually evaluate each participant's work. On the other hand, in-time feedback directly addressing students' mistakes can aid the learning process. Providing feedback automatically could thus greatly enhance these courses.

Traditional programming tutors use manually constructed domain models to generate feedback. Model-tracing tutors simulate the problem-solving *process*: how students program. This is challenging because there are no well-defined steps when writing a program. Many tutors instead only analyze individual programs submitted by students, and disregard how a program evolved. They often use models coded in terms of constraints or bug libraries [10].

Developing a domain model typically requires significant knowledge-engineering effort [4]. This is particularly true for programming tutors, where most problems have several alternative solutions with many possible implementations [11]. Data-driven tutors reduce the necessary authoring effort by mining educational data – often from online courses – to learn common errors and generate feedback [17,16,8].

In this paper we address the problem of finding useful features to support data mining in programming tutors. To support hint generation, these features must be robust against irrelevant code variations (such as renaming a variable) and relatable to knowledge components of the target skill (programming).

We describe features with *abstract-syntax-tree patterns* that encode relations between nodes in a program's abstract syntax tree. We use patterns that describe a path between pairs of leaf nodes referring to variables or values. By omitting some nodes on these paths, patterns can match different programs containing the same relation. We then induce rules to predict program correctness from AST patterns, allowing us to generate hints based on missing or incorrect patterns.

We evaluated our approach on existing Prolog programs submitted by students during past lab sessions of a second-year university course. For 73% of incorrect submissions we are able to suggest potentially useful patterns – those that the students had actually implemented in the final, correct programs.

The main contributions presented in this paper are: AST patterns as features for machine learning, a rule-based model for predicting program correctness, and hint generation from incorrect or missing patterns in student programs.

2 Background

Several programming tutors generate hints from differences between the student's program and a predefined set of possible solutions. The possible solution strategies for each problem can be given as a set of programs, or specified in a domain-specific language. Both Johnson's Pascal tutor [9] and Hong's Prolog tutor [7] perform hierarchical goal decomposition based on predefined programming *plans* or *techniques* to determine the student's intent. Gerdes et al. use a small set of annotated model programs to derive solution strategies, which function in a similar way [5].

Rivers and Koedinger compare students' programs directly to a database of previous correct submissions [17]. They reduce program variability using equivalence-preserving transformations, such as inlining functions and reordering binary expressions. Hints are generated by suggesting a minimal correct step leading from the current submission to the closest correct program.

Another option is to compare program behavior. Nguyen et al. classify programming mistakes according to results on a preselected set of test inputs [16]. Li et al. generate test cases to distinguish between programs by selecting inputs that exercise different code paths in the program [14]. Such tutors can point out pertinent failing test cases, which can be very helpful.

Constraints [15] encode domain principles using if-then rules with relevance and satisfaction conditions, e.g. "if a function has a non-void return type, then it must have a return statement" [6]. If a program violates a constraint, the tutor displays a predefined message. Le's Prolog tutor improves constraint-based diagnosis by assigning weights to different types of constraints [12].

Jin et al. use *linkage graphs* to describe data dependencies between the program's statements [8]; we use AST patterns in a similar way. Nguyen et al. analyzed submissions in a large machine-learning course to learn a vocabulary of *code phrases*: subtrees of submissions' abstract syntax trees that perform the same function in a given context [16]. By swapping parts between different programs, they built up a search library of functionally equivalent AST subtrees within a given context.

The idea for AST patterns comes from Tregex – *tree regular expressions*, mainly used in the field of natural-language processing [13]. Tregex patterns can encode complex relations between nodes, but can become unwieldy; in this paper we use a simpler s-expression syntax. Another language for describing tree patterns using s-expressions is *trx*, which additionally supports choice, repetition and other operators [1].

3 AST patterns

In this section we describe AST patterns through examples, while Sect. 4.1 explains how patterns are extracted from student programs. Consider the following Prolog program implementing the relation `sister(X,Y)`[1]:

```
sister(X,Y):-       % X is Y's sister when:
  parent(P,X),
  parent(P,Y),      % X and Y share a common parent P,
  female(X),        % X is female, and
  X \= Y.           % X and Y are not the same person.
```

Figure 1 shows the program's AST with two patterns. The pattern drawn with blue dotted arrows encodes the fact that the first argument to the `sister` predicate also appears as the first argument in the call to `female`. In other words, this pattern states that `X` must be female to be a sister. We write this pattern as the s-expression

```
(clause (head (compound (functor 'sister') (args var)))
  (compound (functor 'female') (args var)))
```

Every pattern used in this paper has the same basic structure, and describes paths from a clause node to one or two leaf nodes containing variables or values. All patterns in Figs. 1 and 2 are induced from such pairs of nodes. For each leaf we also include some local context, such as the name of the predicate (e.g. `parent`) and the operators used in unop and binop nodes.

We regard these patterns as the smallest units of meaning in Prolog programs: each pattern encodes some interaction between two objects (variable or value) in the program. Including more than two leaf nodes in a pattern could make it difficult to pinpoint the exact error when generating hints. Each pattern contains

[1] Binary relations like this one should be read as "`X` is a sister/parent/... of `Y`".

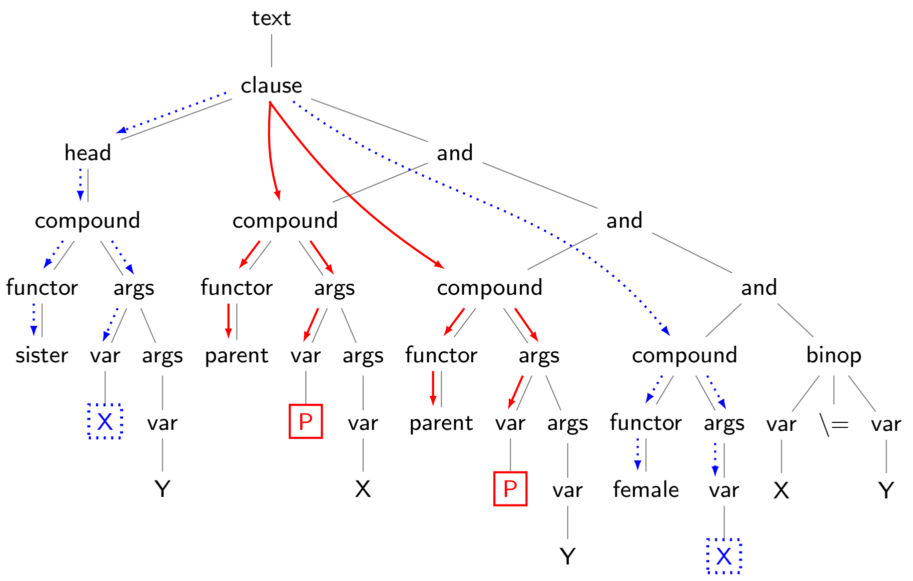

Fig. 1. The AST for the `sister` program, showing two patterns and the leaf nodes inducing them. Solid red arrows equate the first arguments in the two calls to `parent`. Dotted blue arrows encode the necessary condition that `X` must be female to be a sister.

at most two var nodes, so we require they both refer to the same variable – relating two nodes with different variables would not tell us much about the program. We can thus omit actual variable names from patterns.

We handle syntactic variations in programs by omitting certain nodes from patterns. For example, by not including and nodes, the above pattern can match a clause regardless of the presence (or order) of other goals in its body (i.e., with any arrangement of and nodes in the AST). Order *is* important for the nodes specified in the pattern; this is explained below.

The second pattern in Fig. 1, drawn with solid red arrows, encodes the fact that the two calls to `parent` share the first argument. In other words, `X` and `Y` must have the same parent `P`.

```
(clause (compound (functor 'parent') (args var))
   (compound (functor 'parent') (args var)))
```

Patterns describe relations between nodes in a program's AST. Specifically, the pattern (a b c) means that the nodes b and c are descended from a, and that b precedes c in a depth-first tree walk. In general, an AST matches the pattern (name p_1 … p_k) if it contains a node n labeled name; the subtree rooted at n must also contain, in depth-first order, distinct nodes n_1 to n_k matching subpatterns p_1 to p_k. The above pattern, for example, matches only the last of the following programs (the first program is missing one call to `parent`, and the second has different variables in positions encoded by the pattern):

```
% nonmatching          % nonmatching          % matching
sister(X,Y):-          sister(X,Y):-          sister(X,Y):-
  female(X),             female(X),             parent(A,X),
  parent(P,X),           parent(A,X),           female(X),
  X \= Y.                parent(B,Y),           parent(A,Y),
                         X \= Y.                X \= Y.
```

A relation between any two objects in a program is insufficient to reason about the program's behavior on the whole. In the tutoring context, however, there are patterns that strongly indicate the presence of certain bugs. Take for example the following incorrect program to sum a list:

```
sum([],0).            % base case: the empty list sums to zero
sum([H|T],Sum):-      % recursive case:
  sum(T,Sum),         %  sum the tail and
  Sum is Sum + H.     %  add first element (bug: reused variable)
```

This error is fairly common with Prolog novices: the variable `Sum` is used to represent both the sum of the whole list (line 2), and the sum of only the tail elements (line 3). The last line fails since Prolog cannot unify `Sum` with a (generally) different value of `Sum + H`. The program's AST is displayed in Fig. 2.

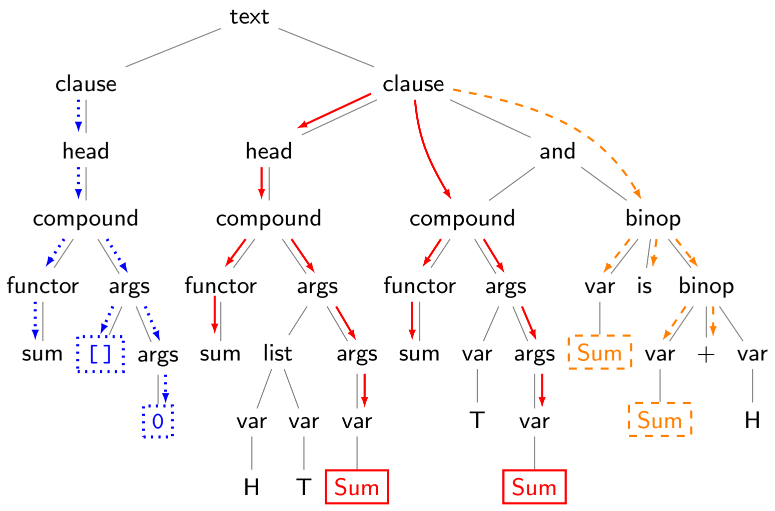

Fig. 2. The AST for the buggy `sum` program. Dotted arrows relate the correct values in the base case. Solid and dashed arrows denote two patterns describing incorrect reuse of the `Sum` variable in the recursive case.

Various patterns capture this mistake. Solid red arrows in Fig. 2 show one example – `Sum` returned by the predicate should not be the same as the `Sum` from the recursive call:

```
(clause (head (compound (functor 'sum') (args (args var))))
    (compound (functor 'sum') (args (args var))))
```

The second pattern, drawn with dashed orange arrows in the figure, indicates the likely error in the arithmetic expression:

```
(clause (binop var 'is' (binop var '+')))
```

The leftmost pattern in Fig. 2, drawn with dotted blue arrows, describes the correct relation between the two constants in the base-case rule:

```
(clause (head (compound (functor 'sum') (args [] (args 0)))))
```

We include such patterns in our feature set to cover the base-case clauses in recursive programs, which often include no variables.

4 Method

This section explains the three steps in our approach: discovering AST patterns, learning classification rules for correct and incorrect programs, and using those rules to generate hints.

4.1 Extracting patterns

We extract patterns from student submissions. As described above, we are only interested in patterns connecting pairs of leaf nodes in an AST: either two nodes referring to the same variable (like the examples in Fig. 1), or a value (such as the empty list `[]` or the number `0`) and another variable/value occurring within the same `compound` or `binop` (like the blue dotted pattern in Fig. 2).

We induce patterns from such node pairs. Given the clause (the second occurrence of each variable – `A`, `B` and `C` – is marked with ' for disambiguation)

```
a(A, B):-
  b(A', C),
  B' is C' + 1.
```

we select the following pairs of nodes: {`A`, `A'`}, {`B`, `B'`}, {`C`, `C'`}, {`B'`, `1`} and {`C'`, `1`}.

For each selected pair of leaf nodes (a, b) we construct a pattern by walking the AST in depth-first order and recording nodes that lie on the paths to a and b. We omit `and` nodes, as explained in the previous section. We also include certain nodes that lie near the paths to selected leaves. Specifically, we include the functor/operator of all `compound`, `binop` and `unop` nodes containing a or b.

Patterns are extracted automatically given above constraints (each pattern connecting a pair of variables or values). We find that such patterns work well for Prolog. Other languages, however, will likely require different kinds of patterns to achieve good performance.

In order to avoid inducing rules specific to a particular program (covering typos and other idiosyncratic mistakes), we ignore rare patterns. In this study we used patterns that occurred in at least five submissions. These patterns form the feature space for rule learning.

4.2 Learning rules

We represent students' programs in the feature space of AST patterns described above. Each pattern corresponds to one binary feature with value `true` when the pattern is present and `false` when it is absent. We classify each program as correct if it passes a predefined set of test cases, and incorrect otherwise. We use these labels for machine learning.

Since we can already establish program correctness using appropriate tests cases, our goal here is not classifying new submissions. Instead, we wish to discover patterns associated with correct and incorrect programs. This approach to machine learning is called *descriptive induction* – the automatic discovery of patterns describing regularities in data. We use rule learning for this task, because rule conditions can be easily translated to hints.

Before explaining the algorithm, let us discuss the reasons why a program can be incorrect. Our experience indicates that bugs in student programs can often be described by 1) some incorrect or *buggy* pattern, which needs to be removed, or 2) some missing relation (pattern) between objects that should be included before the program can be correct. We shall now explain how both types of errors can be identified with rules.

To discover buggy patterns, the algorithm first learns *negative rules* that describe incorrect programs. We use a variant of the CN2 algorithm [2] implemented within the Orange data-mining toolbox [3]. Since we use rules to generate hints, and since hints should not be presented to students unless they are likely to be correct, we impose additional constraints on the rule learner:

– classification accuracy of each learned rule must exceed a threshold (we selected 90%, as 10% error seems acceptable for our application);
– each conjunct in a condition must be significant according to the likelihood-ratio test (in our experiments we set significance threshold to $p = 0.05$);
– a conjunct can only specify the presence of a pattern (in other words, we only allow feature-value pairs with the value `true`).

The first two constraints ensure good rules with only significant patterns, while the last constraint ensures rules only mention the presence (and not absence) of patterns as reasons for a program to be incorrect. This is important, since conditions in negative rules should contain patterns symptomatic of incorrect programs.

With respect to the second type of error, we could try the same approach and use the above algorithm to learn *positive rules* for the class of correct programs. The conditional part of positive rules should define sufficient combinations of patterns that render a program correct. It turns out that it is difficult to learn accurate positive rules, because there are many programs that are incorrect despite having all important patterns, because they also include incorrect patterns.

A possible way to solve this problem is to remove programs that are covered by some negative rule. This way all known buggy patterns are removed from the data, and will not be included in positive rules. However, removing incorrect patterns also removes the need for specifying relevant patterns in positive rules.

For example, if all incorrect programs were removed, the single rule "true $\Rightarrow$ correct" would suffice, which cannot be used to generate hints. We achieved the best results by learning positive rules from the complete data set, but estimating their accuracy only on programs not covered by some negative rule.

While our main interest is discovering important patterns, induced rules can still be used to classify new programs, for example to evaluate rule quality. Classification proceeds in three steps: 1) if a negative rule covers the program, classify it as incorrect; 2) else if a positive rule covers the program, classify it as correct; 3) otherwise, if no rule covers the program, classify it as incorrect.

We note that Prolog clauses can often be written in various ways. For example, the clause "`sum([],0).`" can also be written as

```
sum(List,Sum):- List = [], Sum = 0.
```

Our method covers such variations by including additional patterns and rules. Another option would be to use rules in conjunction with program canonicalization, by transforming each submission into a semantically equivalent normalized form before extracting patterns [17].

4.3 Generating hints

Once we have induced the rules for a given problem, we can use them to provide hints based on buggy or missing patterns. To generate a hint for an incorrect program, each rule is considered in turn. We consider two types of feedback: *buggy hints* based on negative rules, and *intent hints* based on positive rules.

First, all negative rules are checked to find any known incorrect patterns in the program. To find the most likely incorrect patterns, the rules are considered in the order of decreasing quality. If all patterns in the rule "$p_1 \land \cdots \land p_k \Rightarrow$ incorrect" match, we highlight the corresponding leaf nodes. As a side note, we found that most negative rules are based on the presence of a single pattern. For the incorrect `sum` program from the previous section, our method produces the following highlight

```
sum([],0).           % base case: the empty list sums to zero
sum([H|T],Sum):-     % recursive case:
  sum(T,Sum),        %  sum the tail and
  Sum is Sum + H.    %  add first element (bug: reused variable)
```

based on the rule "$p \Rightarrow$ incorrect", where p is the solid red pattern in Fig. 2. This rule covers 36 incorrect programs, and one correct program using an unusual solution strategy.

If no negative rule matches the program, we use positive rules to determine the student's intent. positive rules group patterns that together indicate a high likelihood that the program is correct. Each positive rule thus defines a particular "solution strategy" in terms of AST patterns. We reason that alerting the student to a missing pattern could help them complete the program without revealing the whole solution.

Table 1. Results on five selected domains and averaged results over 44 domains. Columns 2, 3, and 4 contain classification accuracies of our rule learning method, majority classifier, and random forest, respectively. Columns 5 and 6 report the number of all generated buggy hints and the number of hints that were actually implemented by students. The following three columns contain the number of all generated intent hints (All), the number of implemented hints (Imp) and the number of implemented alternative hints (Alt). The numbers in the last column are student submission where hints could not be generated. The bottom two rows give aggregated results (total and average) over all 44 domains.

Problem	CA			Buggy hints		Intent hints			No hint
	Rules	Maj	RF	All	Imp	All	Imp	Alt	
sister	0.988	0.719	0.983	128	128	127	84	26	34
del	0.948	0.645	0.974	136	136	39	25	10	15
sum	0.945	0.511	0.956	59	53	24	22	1	6
is_sorted	0.765	0.765	0.831	119	119	0	0	0	93
union	0.785	0.783	0.813	106	106	182	66	7	6
...									
Total				3613	3508	2057	1160	244	1045
Average	0.857	0.663	0.908	79.73	77.34	46.75	26.36	5.55	23.75

When generating a hint from positive rules, we consider all *partially matching* rules "$p_1 \wedge \cdots \wedge p_k \Rightarrow \mathsf{correct}$", where the student's program matches some (but not all) patterns p_i. For each such rule we store the number of matching patterns, and the set of missing patterns. We then return the most common missing pattern among the rules with most matching patterns.

For example, if we find the following missing pattern for an incorrect program implementing the `sister` predicate:

(clause (head (compound (functor '`sister`') (args var))) (binop var '`\=`')),

we could display a message to the student saying "comparison between `X` and some other value is missing", or "your program is missing the goal `X \= ?`".

This method can find several missing patterns for a given partial program. In such cases we return the most commonly occurring pattern as the main hint, and other candidate patterns as alternative hints. We use main and alternative intent hints to establish the upper and lower bounds when evaluating hints.

5 Evaluation

We evaluated our approach on 44 programming assignments. We preselected 70% of students whose submissions were used as learning data for rule learning. The submissions from the remaining 30% of students were used as testing data to evaluate classification accuracy of learned rules, and to retrospectively evaluate quality of given hints. Problems analyzed in this paper constitute a complete introductory course in Prolog, covering the basics of the language.

Table 1 contains results on five selected problems (each representing a group of problems from one lab session), and averaged results over all 44 problems.[2] The second, third, and fourth columns provide classification accuracies (CA) of the rule-based, majority, and random-forest classifiers on testing data. The majority classifier and the random forests method, which had the best overall performance, serve as references for bad and good CA on particular data sets.

For example, our rules correctly classified 99% of testing instances for the `sister` problem, the accuracy of the majority classifier was 66%, and random forests achieved 98%. CA of rules is also high for problems `del` and `sum`. It is lower, however, for `is_sorted` and `union`, suggesting that the proposed AST patterns are insufficient for certain problems. Indeed, after analyzing the problem `is_sorted`, we observed that our patterns do not cover predicates with a single empty-list (`[]`) argument, which occurs as the base case in this problem. For this reason, the rule learning algorithm failed to learn any positive rules and therefore all programs were classified as incorrect. In the case of `union`, many solutions use the cut (`!`) operator, which is also ignored by our pattern generation algorithm.

We evaluated the quality of hints on incorrect submissions from those student traces that resulted in a correct program. In the case of the `sister` data set, there were 289 such incorrect submission out of 403 submissions in total.

The columns captioned "Buggy hints" in Table 1 contain evaluation of buggy hints generated from negative rules. For each generated buggy hint we checked whether it was implemented by the student in the final submission. The column "All" is the number of all generated buggy hints, while the column "Imp" is the number of implemented hints. The results show high relevance of generated buggy hints, as 97% (3508 out of 3613) of them were implemented in the final solution; in other words, the buggy pattern was removed.

The intent hints are generated when the algorithm fails to find any buggy hints. The column "All" contains the number of generated intent hints, "Imp" the number of implemented main intent hints, and "Alt" is the number of implemented alternative hints. Notice that the percentage of implemented intent hints is significantly lower when compared to buggy hints: in the case of problem `sister` 84 out of 127 (66%) hints were implemented, whereas in the case of problem `union` only 66 out of 182 (36%) hints were implemented. On average, 56% of main intent hints were implemented.

The last column shows the number of submissions where no hints could be generated. This value is relatively high for the `is_sorted` problem, because the algorithm could not learn any positive rules and thus no intent hints were generated.

To sum up, buggy hints seem to be good and reliable, since they are always implemented when presented, even when we tested them on past data – the decisions of students were not influenced by these hints. The percentage of implemented intent hints is, on average, lower (56%), which is still not a bad result, providing that it is difficult to determine the programmer's intent. In 12% (244

[2] We report only a subset of results due to space restrictions. Full results and source code can be found at `https://ailab.si/aied2017`.

out of 2057) of generated intent hints, students implemented an alternative hint that was identified by our algorithm. Overall we were able to generate hints for 84.5% of incorrect submissions. Of those hints, 86% were implemented (73% of all incorrect submissions).

High classification accuracies for many problems imply that it is possible to determine program correctness simply by checking for the presence of a small number of patterns. Our hypothesis is that for each program certain crucial patterns exist that students have difficulties with. When they figure out these patterns, implementing the rest of the program is usually straightforward.

6 Conclusion

We have used AST patterns as features to describe program structure. By encoding only relations between particular nodes, each pattern can match many programs. AST patterns thus function as a sort of “regular expressions” for trees.

We presented a method for automatically extracting AST patterns from student programs. Our patterns encode relations between data objects in a program, with each pattern connecting either two instances of the same variable, a variable and a value, or two values. We consider such patterns as the atomic syntactic relations in a program, and use them as machine-learning features.

We explained how to induce rules for classifying correct and incorrect programs based on AST patterns. Since the goal of our research is to generate hints, we adapted the CN2 algorithm to produce rules useful for that purpose. We induce rules in two passes: we first learn the rules for incorrect programs, and then use programs not covered by any such rule to learn the rules for correct programs.

Evaluation shows that our patterns are useful for classifying Prolog programs. Other programming languages will likely require different patterns. For example, in commonly taught imperative languages such as Python or Java, each variable can take on different values and appear in many places. Further research is needed to determine the kinds of patterns useful in such situations.

We showed how to generate hints based on rules by highlighting buggy patterns or pointing out what patterns are missing. Evaluation on past student data shows that useful hints can be provided for many incorrect submissions this way. The quality of feedback could be improved by annotating rules with explanations in natural language. Since patterns and rules are easily interpretable, they can also help when manually authoring tutoring systems, by indicating the common errors and the typical solution strategies for each problem.

In the future we will attempt to improve rule accuracy for certain problems, such as `union`. This will likely necessitate new kinds of patterns, for example to handle the cut operator. Adapting our methods to handle Python programs will give us some insight into what kinds of patterns could be useful in different situations. Finally, we will implement hint generation in an online programming tutor CodeQ, and evaluate the effect of automatic feedback on students’ problem-solving.

References

1. Bagrak, I., Shivers, O.: trx: Regular-tree expressions, now in scheme. In: Proc. Fifth Workshop on Scheme and Functional Programming. pp. 21–32 (2004)
2. Clark, P., Boswell, R.: Rule induction with CN2: Some recent improvements. In: Proc. Fifth European Conference on Machine Learning. pp. 151–163 (1991)
3. Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., Zupan, B.: Orange: Data mining toolbox in Python. Journal of Machine Learning Research 14, 2349–2353 (2013), `http://jmlr.org/papers/v14/demsar13a.html`
4. Folsom-Kovarik, J.T., Schatz, S., Nicholson, D.: Plan ahead: Pricing ITS learner models. In: Proc. 19th Behavior Representation in Modeling & Simulation Conference. pp. 47–54 (2010)
5. Gerdes, A., Heeren, B., Jeuring, J., van Binsbergen, L.T.: Ask-elle: an adaptable programming tutor for haskell giving automated feedback. International Journal of Artificial Intelligence in Education pp. 1–36 (2016)
6. Holland, J., Mitrovic, A., Martin, B.: J-LATTE: a constraint-based tutor for Java. In: Proc. 17th Int'l Conf. Computers in Education (ICCE 2009). pp. 142–146 (2009)
7. Hong, J.: Guided programming and automated error analysis in an intelligent Prolog tutor. International Journal of Human-Computer Studies 61(4), 505–534 (2004)
8. Jin, W., Barnes, T., Stamper, J., Eagle, M.J., Johnson, M.W., Lehmann, L.: Program representation for automatic hint generation for a data-driven novice programming tutor. In: Proc. 11th Int'l Conf. Intelligent Tutoring Systems. pp. 304–309 (2012)
9. Johnson, W.L.: Understanding and debugging novice programs. Artificial Intelligence 42(1), 51–97 (1990)
10. Keuning, H., Jeuring, J., Heeren, B.: Towards a systematic review of automated feedback generation for programming exercises. In: Proc. 2016 ACM Conf. on Innovation and Technology in Computer Science Education. pp. 41–46. ACM (2016)
11. Le, N.T., Loll, F., Pinkwart, N.: Operationalizing the continuum between well-defined and ill-defined problems for educational technology. IEEE Transactions on Learning Technologies 6(3), 258–270 (2013)
12. Le, N.T., Menzel, W.: Using weighted constraints to diagnose errors in logic programming – the case of an ill-defined domain. International Journal of Artificial Intelligence in Education 19(4), 381–400 (2009)
13. Levy, R., Andrew, G.: Tregex and tsurgeon: tools for querying and manipulating tree data structures. In: 5th Int'l Conf. Language Resources and Evaluation (2006)
14. Li, S., Xiao, X., Bassett, B., Xie, T., Tillmann, N.: Measuring code behavioral similarity for programming and software engineering education. In: Proc. 38th Int'l Conf. on Software Engineering Companion. pp. 501–510. ACM (2016)
15. Mitrovic, A.: Fifteen years of constraint-based tutors: what we have achieved and where we are going. User Modeling and User-Adapted Interaction 22(1-2), 39–72 (2012)
16. Nguyen, A., Piech, C., Huang, J., Guibas, L.: Codewebs: scalable homework search for massive open online programming courses. In: Proc. 23rd Int'l Conf. World Wide Web. pp. 491–502 (2014)
17. Rivers, K., Koedinger, K.R.: 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)