Automatic Generation of Programming Feedback: A Data-Driven Approach

Kelly Rivers and Kenneth R. Koedinger

Carnegie Mellon University

Abstract. Automatically generated feedback could improve the learning gains of novice programmers, especially for students who are in large classes where instructor time is limited. We propose a data-driven approach for automatic feedback generation which utilizes the program solution space to predict where a student is located within the set of many possible learning progressions and what their next steps should be. This paper describes the work we have done in implementing this approach and the challenges which arise when supporting ill-defined domains.

Keywords: automatic feedback generation; solution space; computer science education; intelligent tutoring systems

1 Introduction

In the field of learning science, feedback is known to be important in the process of helping students learn. In some cases, it is enough to tell a student whether they are right or wrong; in others, it is better to give more details on why a solution is incorrect, to guide the student towards fixing it. The latter approach may be especially effective for problems where the solution is complex, as it can be used to target specific problematic portions of the student's solution instead of throwing the entire attempt out. However, it is also more difficult and time-consuming to provide.

In computer science education, we have been able to give students a basic level of feedback on their programming assignments for a long time. At the most basic level, students can see whether their syntax is correct based on feedback from the compiler. Many teachers also provide automated assessment with their assignments, which gives the student more semantic information on whether or not their attempt successfully solved the problem. However, this feedback is limited; compiler messages are notoriously unhelpful, and automated assessment is usually grounded in test cases, which provide a black and white view of whether the student has succeeded. The burden falls on the instructors and teaching assistants (TAs) to explain to students why their program is failing, both in office hours and in grading. Unfortunately, instructor and TA time is limited, and it becomes nearly impossible to provide useful feedback when course sizes become larger and massive open online courses grow more common.

Given this situation, a helpful approach would be to develop a method for automatically generating more content-based and targeted feedback. An automatic approach could scale easily to large class sizes, and would hopefully be able to handle a large portion of the situations in which students get stuck. This would greatly reduce instructor grading time, letting them focus on the students who struggle the most. Such an approach is easier to hypothesize than it is to create, since student solutions are incredibly varied in both style and algorithmic approach and programming problems can become quite complex. An automatic feedback generation system would require knowledge of how far the student had progressed in solving the problem, what precisely was wrong with their current solution, and what constraints were required in the final, correct solutions.

In this paper, we propose a method for creating this automatic feedback by utilizing the information made available by large corpuses of previous student work. This data can tell us what the most common correct solutions are, which misconceptions normally occur, and which paths students most often take when fixing their bugs. As the approach is data-driven, it requires very little problem-specific input from the teacher, which makes it easily scalable and adaptable. We have made significant progress in implementing this approach and plan to soon begin testing it with real students in the field.

2 Solution Space Representation

Our method relies upon the use of *solution spaces*. A solution space is a graph representation of all the possible paths a student could take in order to get from the problem statement to a correct answer, where the nodes are candidate solutions and the edges are the actions used to move from one solution state to another. Solution spaces can be built up from student data by extracting students' learning progressions from their work and inserting them into the graph as a directed chain. Identical solutions can be combined, which will represent places where a student has multiple choices for the next step to take, each of which has a different likelihood of getting them to the next answer.

A solution space can technically become infinitely large (especially when one considers paths which do not lead to a correct solution), but in practice there are common paths which we expect the student to take. These include the learning progression that the problem creator originally intended, other progressions that instructors and teaching assistants favor, and paths that include any common misconceptions which instructors may have recognized in previous classes. If we can recognize when a student is on a common path (or recognize when the student has left the pack entirely) we can give them more targeted feedback on their work.

While considering the students' learning progressions, we need to decide at what level of granularity they should be created. We might consider very small deltas (character or token changes), or very large ones (save/compile points or submissions), depending on our needs. In our work we use larger deltas in order to examine the points at which students deliberately move from one state to the

next; every time a student saves, they are pausing in their stream of work and often checking to see what changes occur in their program's output. Of course, this approach cannot fully represent all of the work that a student does; we cannot see the writing they are doing offline or hear them talking out ideas with their TAs. These interactions will need to be inferred from the changes in the programs that the student writes if we decide to account for them.

It is simple to create a basic solution space, but making the space *usable* is a much more difficult task. Students use different variable names, indentations, and styles, and there are multitudes of ways for them to solve the same problem with the same general approach. In fact, we do not want to see two different students submitting exactly the same code– if they do, we might suspect them of cheating! But the solution space is of no use to us if we cannot locate new students inside of it. Therefore, we need to reduce the size of the solution space by combining all semantically equivalent program states into single nodes.

Many techniques have been developed already for reducing the size of the solution spaces of ill-defined problems. Some represent the solution states with sets of constraints [5], some use graph representations to strip away excess material [4], and others use transformations to simplify solution states [9, 8]. We subscribe to the third approach by transforming student programs into *canonical forms* with a set of normalizing program transformations. These transformations simplify, anonymize, and order the program's syntax without changing its semantics, mapping each individual program to a normalized version. All transformations are run over abstract syntax trees (ASTs), which are parse trees for programs that remove extraneous whitespace and comments. If two different programs map to the same canonical form, we know that they must be semantically equivalent, so this approach can safely be used to reduce the size of the solution space.

Example Let us consider a very simple programming problem from one of the first assignments of an introductory programming class. The program takes as input an integer between 0 and 51, where each integer maps to a playing card, and asks the student to return a string representation of that card. The four of diamonds would map to 15, as clubs come before diamonds; therefore, given an input of 15, the student would return "4D". This problem tests students' ability to use mod and div operators, as well as string indexing. One student's incorrect solution to this problem is shown in Figure 1.

```
def intToPlayingCard(value):
    faceValue = value%13
    #use remainder as an index to get the face
    face = "23456789YJQKA"[faceValue]
    suitValue = (value-faceValue)%4
    suit = "CDHS"[suitValue]
    return face+suit
```

Fig. 1. A student's attempt to solve the playing card problem.

To normalize this student's program, we first extract the abstract syntax tree from the student's code, as is partially shown in Figure 2. All of the student's variables are immediately anonymized; in this case, 'value' will become 'v0', 'faceValue' will be 'v1', etc. We then run all of our normalizing transformations over the program, to see which ones will have an effect; in this case, the only transformation used is *copy propagation*. This transformation reduces the list of five statements in the program's body to a single return statement by copying the value assigned to each variable into the place where the variable is used later on. Part of the resulting canonical form is displayed in Figure 2. The new tree is much smaller, but the program will have the same effect.

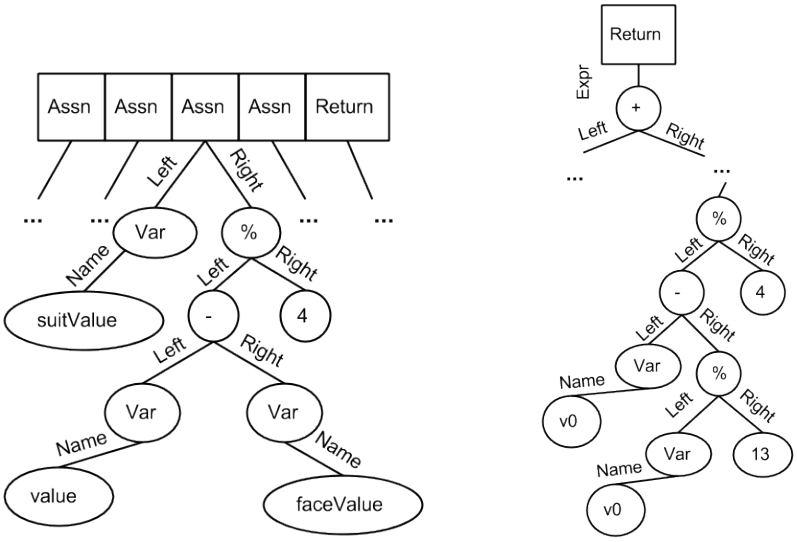

Fig. 2. Subparts of the student's ASTs, before (left) and after (right) normalization.

We have already implemented this method of solution space reduction and tested it with a dataset of final submissions from a collection of introductory programming problems. The method is quite effective, with the solution space size being reduced by slightly over 50% for the average problem [7]. However, we still find a long tail of singleton canonical forms existing in each problem's solution space, usually due to students who found strange, unexpected approaches or made unconventional mistakes. This long tail of unusual solutions adds another layer of complexity to the problem, as it decreases the likelihood that a new student solution will appear in the old solution space.

Our work so far has concentrated only on final student submissions, not on the paths students take while solving their problems. This could be seen as problematic, as we are not considering the different iterations a student might go through while working. However, our very early analysis of student learning progressions from a small dataset has indicated that students are not inclined to use incremental approaches. The students we observed wrote entire programs in single sittings, then debugged until they could get their code to perform correctly.

This suggests that our work using final program states may be close enough to the real, path-based solution space to successfully emulate it.

We note that the solution space is easiest to traverse and create when used on simple problems; as the required programs become longer, the number of individual states in the space drastically increases. We believe it may be possible to address this situation by breaking up larger problems into hierarchies of subproblems, each of which may map to a specific chunk of code. Then each subproblem can have its own solution space that may be examined separately from the other subproblems, and feedback can be assigned for each subproblem separately.

3 Feedback Generation

Once the solution space has been created, we need to consider how to generate feedback with it. The approach we have adopted is based on the Hint Factory [1], a logic tutor which uses prior data to give stuck students feedback on how to proceed. In the Hint Factory, each node of the solution space was the current state of the student's proof, and each edge was the next theorem to apply that would help the student move closer to the complete proof. The program used a Markov Decision Process to decide which next state to direct the student towards, optimizing for the fastest path to the solution.

Our approach borrows heavily from the Hint Factory, but also expands it. This is due to the ill-defined nature of solving programming problems, which specifies that different solutions can solve the same problem; this complicates several of the steps used in the original logic tutor. In this section we highlight three challenges that need to be addressed in applying the Hint Factory methodology to the domain of programming, and describe how to overcome each of them.

Other attempts have been made at automatic generation of feedback, both in the domain of programming and in more domain-general contexts. Some feedback methods rely on domain knowledge to create messages; Paquette et al.'s work on supporting domain-general feedback is an example of this [6]. Other methods rely instead on representative solutions, comparing the student's solution to the expected version. Examples here include Gerdes et al.'s related work on creating functional tutoring systems (which use instructor-submitted representative solutions) [2] and Gross et al.'s studies on using clustering to provide feedback (which, like our work, use correct student solutions) [3]. Though our work certainly draws on many of the elements used in these approaches, we explore the problem from a different angle in attempting to find entire paths to the closest solution (which might involve multiple steps), rather than jumping straight from the student's current state to the final solution. Whether this proves beneficial will remain to be seen in future studies.

3.1 Ordering of Program States

Our first challenge relates to the process of actually mapping out the suggested learning progressions for the student. Even after reducing the size of the solution space, there are still a large number of distinct solutions which are close yet not connected by previously-found learning paths. These close states can be helpful, as they provide more opportunities for students to switch between different paths while trying to reach the solution. Therefore, we need to connect each state to those closest to it, then determine which neighboring state will set the student on the best path to get to a final solution.

One obvious method for determining whether two states are close to each other would involve using tree edit distance, to determine how many changes needed to be made. However, this metric does not seem to work particularly well in practice; the weight of an edit is difficult to define, which makes comparing edits non-trivial. Instead, we propose the use of string edit distance (in this case, Levenshtein distance) to determine whether two programs are close to each other. To normalize the distances between states, we calculate the percentage similarity with $(l - distance)/l$ (where l is the length of the longer program); this ensures that shorter programs do not have an advantage over longer ones and results from different problems can easily be compared to each other. Once the distances between all programs have been calculated, a cut-off point can be determined that will separate close state pairs from far state pairs. Our early experimentation with this method shows that it is efficient on simple programs and produces pairs of close states for which we can generate artificial actions.

Once the solution space has been completely generated and connected, we need to consider how to find the best path from state A to state B. The algorithm for finding this will be naturally recursive in nature– the best path from A to B will be the best element of the set of paths S, where S is composed of paths from each neighbor of A to B. Paths which require fewer intermediate steps will be preferred, as they require the student to make less changes, but we also need to consider the distances between the program states. We can again use string edit distance to find these distances, or we can use the tree edits to look at the total number of individual changes required. Finally, we can use test cases to assign correctness parameters to each program state (as there are certainly some programs which are more incorrect than others); paths which gradually increase the number of test cases that a student passes may be considered more beneficial than paths which jump back and forth, as the latter paths may lead to discouragement and frustration in students.

Example In the previous section, we had found the canonical form for the student's solution; that form was labeled #22 in the set of all forms. As we were using a dataset of final submissions, we had no learning progressions to work with, we computed the normalized Levenshtein distance between each pair of states and connected those which had a percentage similarity of 90% or higher, thus creating a progression graph.

In Figure 3, we see that state #22 was connected to three possible next states: #4, #34, and #37. We know that #34 is incorrect, so it does not seem like a good choice; on the other hand, #4 and #37 are equally close to #22 and are both correct. State #37 had been reached by thirty students, while state #4 had only been reached by four; since #37 is more commonly used, it is probably the better target solution for the student.

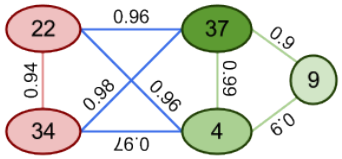

Fig. 3. The program state graph surrounding state #22. Red nodes are incorrect, green correct; a darker node indicates that more students used that approach.

3.2 Generating content deltas between states

Next, we face the challenge of determining how to extract the content of the feedback message from the solution space. The feedback that we give the student comes from the edge between the current and target states, where that edge represents the actions required to get from one state to the other. In well-defined domains, these actions are often simple and concrete, but they become more complex when the problems are less strictly specified.

Before, we used string distance to determine how similar two programs were, in order to find distances quickly and easily. Now that we need to know what the differences actually are, we use tree edits to find the additions, deletions, and changes required to turn one tree into another. It is moderately easy to compute these when comparing ordinary trees, but ASTs add an extra layer of complexity as there are certain nodes that hold lists of children (for example, the bodies of loops and conditionals), where one list can hold more nodes than another. To compare these nodes, we find the maximal ordered subset of children which appear in both lists; the leftover nodes can be considered changes.

After we have computed these edits, we can use them to generate feedback for the student in the traditional way. Cognitive tutors usually provide three levels of hints; we can use the same approach here, first providing the location of an error, then the token which is erroneous, and finally what the token needs to be changed to in order to fix the error. In cases where more than one edit needs to be made the edits can be provided to the student one at a time, so that the student has a chance to discover some of the problems on their own.

It may be possible to map certain edit patterns to higher-level feedback messages, giving students more conceptual feedback. Certain misconceptions and mistakes commonly appear in novice programs; accidental use of integer division and early returns inside of loops are two examples. If we can code the patterns that these errors commonly take (in these cases, division involving two integer values and return statements occurring in the last line of a loop's body), we can

provide higher-level static feedback messages that can be provided to students instead of telling them which values to change. This may help them recognize such common errors on their own in future tasks.

Example To generate the feedback message in our continuing example, we find the tree edits required to get from state #22 to state #37. These come in two parts: one a simple change, the other a more complex edit. Both are displayed in Figure 4. The first change is due to a typo in the string of card face values that the student is indexing (Y instead of T for ten); as the error occurs in a leaf node (a constant value), pointing it out and recommending a change is trivial. Such a feedback message might look like this: *In the return statement, the string "23456789YJQKA" should be "23456789TJQKA".*

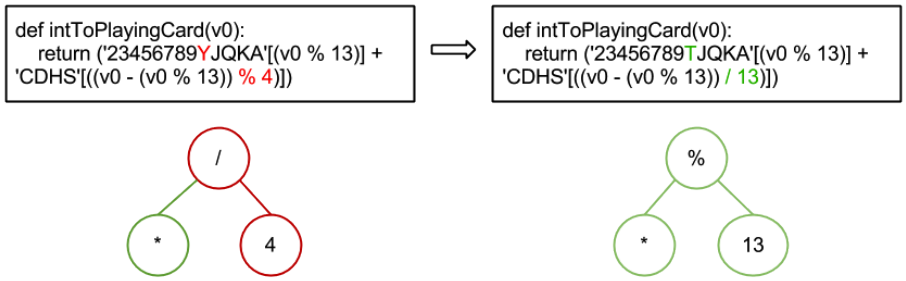

Fig. 4. The change found between the two programs, represented in text and tree format (with * representing a further subexpression).

The second error is due to a misconception about how to find the index of the correct suit value. In the problem statement, the integer card values mapped cards first by face value and then by suit; all integers from 0 to 12 would be clubs, 13 to 23 would be diamonds, etc. This is a step function, so the student should have used integer division to get the correct value. In a terrible twist of fate, this part of the student's code will actually work properly; $v0 - (v0\%13)$ returns the multiple of 13 portion of v0, and the first four multiples of 13 (0, 13, 26, and 39) each return the correct index value when modded by 4 (0, 1, 2, and 3). Still, it seems clear that the student is suffering from a missing piece of knowledge, as it would be much simpler to use the div operator.

In the AST, the two solutions match until they reach the value used by the string index node. At that point, one solution will use mod while the other uses div, and one uses a right operand of 4 while the other uses a right operand of 13. It's worth noting, however, that both use the same subexpression in the left operand; therefore, in creating feedback for the student, we can leave that part out. Here, the feedback message might be this: *In the right side of the addition in the return statement, use div instead of mod.* The further feedback on changing 4 to 13 could be provided if the student needed help again later.

3.3 Reversing deltas to regain content

Finally, we need to take the content of the feedback message which we created in the previous part and map it back to the student's original solution. If the student's solution was equivalent to the program state, this would be easy– however, because we had to normalize the student programs, we will need to map the program state back up to the individual student solution in order to create their personal feedback message.

In some situations, this will be easy. For example, it's possible that a student solution only had whitespace cleaned up and variable names anonymized; if this was the case, the location of the code would remain the same, and variable names could be changed in the feedback message easily. In many other cases, the only transformations applied would be ordering and propagation functions; for these, we can keep track of where each expression occurred in the original program, then map the code segments we care about back to their positions in the original code. Our running example falls into this "easy" category; even though the student's program looks very different from the canonical version, we only need to unroll the copy propagation to get the original positions back.

Other programs will present more difficulties. For example, any student program which has been reduced in size (perhaps through constant folding, or conditional simplification) might have a feedback expression which needs to be broken into individual pieces. One solution for this problem would be to record each transformation that is performed on a program, then backtrack through them when mapping feedback. Each transformation function can be paired with a corresponding "undo" function that will take the normalized program and a description of what was changed, then generate the original program.

Example All of the program transformations applied to the original student program in order to produce state #22 were copy propagations; each variable was copied down into each of its references and deleted, resulting in a single return statement. To undo the transformations, the expressions we want to give feedback on ('23456789YJQKA' and $(v0-(v0\%13))\%4$) must be mapped to the variables that replace them– face and suit (where suit is later mapped again to suitValue). We can then examine the variable assignment lines to find the original location in which the expression was used (see Figure 5), which maps the expressions to lines 2 and 4. The first expression's content is not modified, but the second changes into $(value - faceValue)\%4$. This change lies outside of the feedback that we are targeting, so it does not affect the message.

After the new locations have been found, the feedback messages are correspondingly updated by changing the location that the message refers to. In this case, the first feedback message would change to: In the **second line**, the string '23456789YJQKA' should be '23456789TJQKA'. The second would become: In the **fourth line**, use div instead of mod.

```
def intToPlayingCard(v0):
    return ('23456789YJQKA'[(v0 % 13)] +
'CDHS'[((v0 - (v0 % 13)) % 4)])
```

```
def intToPlayingCard(value):
    faceValue = value%13
    face = "23456789YJQKA"[faceValue]
    #use remainder as an index to get the face
    suitValue = (value-faceValue)%4
    suit = "CDHS"[suitValue]
    return face+suit
```

Fig. 5. A comparison of the canonical (left) and original (right) programs. The code snippets we need to give feedback on are highlighted.

4 Conclusion

The approach we have described utilizes the concept of solution spaces to determine where a new student is in their problem-solving process, then determines what feedback to provide by traversing the space to find the nearest correct solution. Representing the solution space has been implemented and tested, but generating feedback is still in progress; future work will determine how often it is possible to provide a student with truly useful and usable feedback.

Acknowledgements. This work was supported in part by Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education (# R305B090023).

References

1. Barnes, Tiffany, and John Stamper. "Toward automatic hint generation for logic proof tutoring using historical student data." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2008.
2. Gerdes, Alex, Johan Jeuring, and Bastiaan Heeren. "An interactive functional programming tutor." Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. ACM, 2012.
3. Gross, Sebastian, et al. "Cluster based feedback provision strategies in intelligent tutoring systems." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
4. Jin, Wei, et al. "Program representation for automatic hint generation for a data-driven novice programming tutor." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
5. Le, Nguyen-Thinh, and Wolfgang Menzel. "Using constraint-based modelling to describe the solution space of ill-defined problems in logic programming." Advances in Web Based LearningICWL 2007. Springer Berlin Heidelberg, 2008. 367-379.
6. Paquette, Luc, et al. "Automating next-step hints generation using ASTUS." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
7. Rivers, Kelly, and Kenneth R. Koedinger. "A canonicalizing model for building programming tutors." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
8. Weragama, Dinesha, and Jim Reye. "Design of a knowledge base to teach programming." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
9. Xu, Songwen, and Yam San Chee. "Transformation-based diagnosis of student programs for programming tutoring systems." Software Engineering, IEEE Transactions on 29.4 (2003): 360-384.