Automating Hint Generation with Solution Space Path Construction

Kelly Rivers and Kenneth R. Koedinger

Carnegie Mellon University, Pittsburgh, PA
{krivers,koedinger}@cs.cmu.edu

Abstract. Developing intelligent tutoring systems from student solution data is a promising approach to facilitating more widespread application of tutors. In principle, tutor feedback can be generated by matching student solution attempts to stored intermediate solution states, and next-step hints can be generated by finding a path from a student's current state to a correct solution state. However, exact matching of states and paths does not work for many domains, like programming, where the number of solution states and paths is too large to cover with data. It has previously been demonstrated that the state space can be substantially reduced using canonicalizing operations that abstract states. In this paper, we show how solution paths can be constructed from these abstract states that go beyond the paths directly observed in the data. We describe a domain-independent algorithm that can automate hint generation through use of these paths. Through path construction, less data is needed for more complete hint generation. We provide examples of hints generated by this algorithm in the domain of programming.

Keywords: automatic hint generation, feedback, learning path construction, solution space, programming tutor

1 Introduction

We have seen a recent boom of interest in educational technology through the emergence of Massive Open Online Courses and the creation of many educational technology start-up companies. Much emphasis there has been on providing students access to high quality lectures, but there is great unmet promise to better scale the kind of learn-by-doing support that intelligent tutoring systems can provide. A key barrier is the difficulty in authoring intelligent tutors and a key opportunity is the use of past student solution data to ease that development process.

The use of historical student data in generating hints has been examined before [1], but in previous work, solution paths were entirely collected from students. This limited the options for future students who would request hints from such systems, requiring them to stay on the paths that their predecessors had travelled. In such a system, a student who tried to go off-path would not

be able to receive hints, even if they were not particularly far from a solution. Furthermore, relying entirely on historical student data makes it difficult to generate hints for new problems that do not have collections of data, which makes building tutors for new problems nearly impossible. Therefore, we examine the problem of whether it is possible to construct new student solution paths automatically, using only the solution states that have already been collected from previous students. We center this problem around programming solutions, as programming problems provide a large solution space to work in that cannot be mapped out by hand.

In this paper, we focus specifically on the problem of **path construction**: how do we find a set of states that can lead a student from her current state to a correct state if no paths have been generated for that state before? With path construction, tutors would not need to rely on previous solution paths, though it could benefit from them; it could theoretically generate hints even for a problem which has only been given a few correct examples. If we can identify a path from the new solution to a correct state, it is possible to construct a hint for the student based on the steps taken within that path. We first examine the relevant work in the field of automatic hint generation, then frame the problem by defining relevant features in the domain that the problem is based in. Finally, we elaborate on the algorithm used to do the path construction, and evaluate it on a dataset of real student solutions.

2 Related Work

As mentioned above, researchers have already examined the problem of generating hints automatically based on historical student data. The Hint Factory [1] builds solution spaces out of data recorded from students' past work and has been applied in the domain of propositional logic proofs. This system constructs solution paths, that is, sequences of solution steps students enter in the interface, by combining all the steps that prior students have taken into a graph. Hints can be provided to new student solution attempts as long as they exactly match a step previously taken and stored in this graph. Because the search space in propositional logic proofs is reasonably constrained, the stored graph provides good coverage of the possible solution space. As long as a student's solution steps stay within the graph, hints can be provided. In practice, the Hint Factory was demonstrated in the logic proof domain to generate hints for students who asked for them about 80% of the time. In other words, the system provides tutoring in the majority of situations without any AI programming. Ideally, we would like an intelligent tutor to provide hints for the other 20% of requests as well. Even more challenging, we would like to see this data-driven approach to developing intelligent tutoring extend to domains with much larger solution spaces.

A different approach to generating feedback for new states is to cluster or abstract the original states into equivalent groups, and provide feedback based on which cluster the new state falls into. There have also been attempts to utilize clusters in larger graphs, so that feedback could be propagated out from [4] or

compared to [3] a most common correct solution in order to generate feedback for many solutions with little work. There is great promise in the use of clustering to provide feedback on students' solution attempts, but clustering does pose a challenge to providing detailed feedback that is personalized to a student's particular solution, especially in domains where there can be great variety in solutions. It is also hard to tell how solutions which do not fit into the space could be paired with a single cluster. Most importantly, while clustering facilitates providing feedback on what's wrong with a solution, it does not, by itself, provide students with hints as to a reasonable next-step they could take (based on their solution so far) when they are stuck; it can only take a student directly to the known solution. Providing next-step hints is an important, powerful feature of intelligent tutoring systems [7].

3 The Domain

In this paper, we describe how path construction would work in the domain of computer science, where each solution state is a program. The technique itself can be extended into other domains, however, assuming that a few constraints are met. Therefore, in this section we detail the features required for our algorithm to generate feedback within a domain.

First, there must be a **collection of solution states**, where each state is represented as a tree structure and has data on how many students have generated it before. The tree should contain enough data about the student solution that it can represent the student's work accurately without requiring every detail. Our states are the abstract syntax trees of the programs submitted by students. In the case that hints need to be generated for a new problem, this collection could be composed of a few correct solutions generated ahead of time as exemplars.

Second, there must be a **method for testing solution states**, *test(s)*, which returns a number between 0 (completely incorrect) and 1 (correct). In our example, we run multiple test functions over a submitted program and average the results of all the tests. Each test function provides a program with specific inputs and checks whether the returned output is the expected value, and together they provide a range of scores that a student can achieve.

Finally, there must be a **method for comparing solution states**, *diff(a,b)*, which returns a number between 0 (identical solutions) and 1 (completely different). If the states have been stored as trees, it should be possible to build a comparison function for them; in fact, how to do this in a domain-general way has already been explored [5].

It is worth noting that there may be several superficial differences between solution states that should not be accounted for when comparing solutions; it can be helpful to use a canonicalization process [6], which removes syntactic differences while ensuring semantic equality, to ensure that any differences between solutions provide actual semantic meaning. Using a normalizing process

has the added benefit of reducing the number of states used in the solution space significantly, while not reducing the range of solution types that the space covers.

The solution states we use as examples in this paper come from final submissions made by students on programming assignments at the introductory programming course at Carnegie Mellon University. Due to this, our data is quite sparse- few of the solutions in our data set provide true intermediate steps, as most are attempted full solutions (though many have small bugs). Therefore, to generate hints we must rely on our path construction algorithm heavily.

4 Path Construction Method

Due to the huge potential size of the solution space, it is impossible to construct full solution paths for students based on what others have done in the past; there are many options for where the student should go, and it is difficult to specify exactly how they should get there. To more efficiently provide feedback and hints online, we construct an initial **solution space** offline based on the collection of solution states that have already been gathered. A solution space can be thought of as a graph containing the paths that a student might take while solving a problem; the solution states form the nodes inside the graph, and they are connected by edges which express the edits required to move from one state to the next. Some paths within this space are more desirable than others; for example, paths that involve fewer steps to get to a final solution are usually preferable. Other factors may be important in the tutoring context, such as whether one solution or another may be more easily understood by a novice student. In this case, we can use the frequency of a state in previous observations as a heuristic for how useful it may be to a new student.

In the following sections, we describe the path construction algorithm that is used on each of the incorrect states to cumulatively create this solution space. The algorithm can also be used for a new student if they request a hint but have a solution which is not currently in the solution space. Thus, our algorithm not only expands the solution space beyond prior paths, but is also capable of providing hints for student states that have never been observed before.

4.1 Identify the Optimal Goal State

First, given a solution state which is not yet correct (that is, a state which has a test score not equal to 1), we need to find a nearby goal state within the solution space. This state will serve as our approximation for the student's intended final product, and can be used to generate hints that will guide the student towards his or her own goal.

To find the optimal goal, we first iterate through all the correct states in the solution space to find the state which is closest to the current state (i.e., has the lowest diff score). While this state is a possible goal for the student, it is equally plausible that there is a better goal available; after all, while some of the

```
def findPattern(dna, pat, start):
    if findAtIndex(dna,pat,start):
        return start
    while(len(dna)>start+len(pat)):
        if findAtIndex(dna,pat,start):
            return start
        else:
            start+=1
```

```
def findPattern(dna, pat, start):
    while start < len(dna):
        check=findAtIndex(dna,pat,start):
        if check ==True:
            return start
        else:
            start += 1
    return -1
```

Fig. 1. In this solution-goal pair, the while loop's test value edit and the addition of the outer return statement are needed, but the removal of the if statement is not necessary.

differences between the current state and the new goal may provide corrections, others may only change superficial features, as in Figure 1.

To determine which of the changes between the two states are actually necessary, we generate all the possible **change vectors** between their solution and the goal. We define a change vector to hold a *tree path*, which is a set of nodes that can be used to find a specific position in a tree, an old subtree that will be removed, and a new subtree that will be added. This change vector can be used to represent the usual edits we wish to perform when modifying trees- additions, which only have a new subtree; deletions, which only have the old subtree; and edits, which use both. Given a solution state and a change vector, we should be able to apply the vector to the state in order to transform it accordingly.

To find all the change vectors between the solution and the goal, we use the diff function to find the nodes where the two trees differ, and return each difference as a vector. In cases where the trees have a set of elements in a single child (such as the body of a function), we can find an optimal matching of the elements according to their types, only deleting and adding lines where it is necessary.

Once all of the change vectors have been found, we begin the process of locating the optimal subset of them which can create a better goal. To do this, we run the test function on the intermediate solution states that result from applying first the changes individually, then all pairs of changes, and so on. If the algorithm can find a state which is correct and closer to the solution state than the current goal, it becomes the new goal state. It is important to note that, in the worst case, this algorithm requires generation of all possible combinations of change vectors (the power set of the original set), which requires exponential time to execute. However, the algorithm can often halt early if it locates a new goal which is closer to the solution than any of the other sets it is investigating.

At this point, an optimal goal for the solution state will have been found. It is worth asking why we can't stop here and simply give a hint to the student based on the difference between their solution and the goal. In some cases, this is a valid solution; for example, showing the student a comparison of their incorrect solution to a close correct version can serve as a valuable example and may indeed improve learning. However, if the algorithm can generate multiple steps for the student by chunking the hints into groups, we can help them focus on individual components of how to improve their solution, which may help them identify similar components when they work on future problems.

4.2 Identify Valid Intermediate States

Once the goal state of the current state has been identified, the set of all change vector combinations between it and the solution is generated. These states represent all possible intermediate states that might be included in the solution path. However, not all of these states are good states; some combinations of edits might produce states which would not seem reasonable to a student. We identify three properties that are required in possible intermediate states:

- A valid state must be well-formed, compatible with the solution language.
- A valid state must be closer to the goal than the original solution state.
- A valid state must do no worse when tested than the original solution state.

The first two properties are easily defended- there is no sense in telling a student to go to a state that is not well-formed, and there is little point in making a change if it does not move the student closer to the solution. In fact, if the diff function is well made, these two properties should always be met. However, the third property can be debated; sometimes, one needs to break a solution to make it better overall. While it is possible that this sort of backtracking may eventually improve a student's solution, it is unlikely that a student will apply a change if they see that it reduces their score, so we retain this property for the initial version of the algorithm.

4.3 Find the Optimal Change Path

At this point, we have found the optimal goal for the solution state and identified all possible intermediate states between the state and the goal. Now we need to create a path out of the intermediate states to lead from the solution to the goal. To do this, we identify several properties that are desirable in stable next states:

- **Seen Before**: a state which has been seen before is a state which we know is fathomable; otherwise, it would not have been submitted by a student in the past. This does not ensure that the state is good, or even reasonable, but it does provide some confidence in the state's stability.
- **Near current state**: it is best if a state is close to the student's original solution; this ensures that the student will not need to make too many changes based on the hint. This also gives the student a chance to make further changes on their own, so they don't need to rely on the hints.
- **Well-performing**: a stable next state should do as well on the test cases as possible, to ensure that the student makes good progress.
- **Close to goal**: the state should be as close to the goal as possible, so as to lead the student directly there.

We combine these four desirable properties to create a **desirability metric** defined by the formula (1). This metric is used to rank possible next states. The weights in the formula can be adjusted to reflect how important each of the properties are within the domain in question. In our data set, we found that

some of the properties were modestly correlated (such as closeness to solution and score), and adjusted the weights to account for this double-counting and give preference to shorter hints.

$$0.3 * frequency(s) + 0.4 * (1 - diff(s, n)) + 0.1 * test(n) + 0.2 * (1 - diff(n, g)) \quad (1)$$

After ranking all of the change states by desirability, we can pick the best state- the one with the desirability score closest to 1- and set it as the first state on the path to the solution. Then we identify each of the next states that would follow the first by locating all states between the chosen state and the goal (e.g., all states which have change vectors containing the vectors in the first state) and iterating on this step. This will generate an entire solution path that extends from the original state to the goal.

At this point, the algorithm can be used to generate the next states for any solution state given to it. With this, the solution space can be fully constructed. Now, when a student needs a hint on a problem, we can locate their solution within this solution space and find their next state. Turning this into a human-readable message is beyond the scope of this paper, but can be done by transforming the change vectors to match the student's original solution and framing them within a few simple templates.

5 Evaluation

Our research question in this project focused on whether we could construct new student solution paths automatically using only a collection of prior student solution states. To determine whether we have met this goal we test our system on three metrics: whether it is possible to generate hints for incorrect states that are stored in our solution corpus, how long hint generation takes, and how well-aligned produced hints are to the students' intentions.

For the purpose of this testing, we utilized the solution sets from five different programming problems assigned in the introductory course at Carnegie Mellon University. These problems are all fairly complex, requiring the use of conditionals and loops, and had on average 34.5% of their normalized solution sets composed of incorrect solutions.

5.1 How often can we generate hints?

Theoretically, we should be able to generate hints for any incoming solution state as long as we have at least one correct solution state in the solution space. After all, in the very worst case we should be able to ask the student to undo all of the work they've done and then take them step by step through the correct solution. While this is not an optimal choice, it does provide a help option for the student, rather than forcing them to work through the problem on their own.

However, the current implementation has an efficiency limitation in the second step of the path construction process where the change vectors are generated.

The algorithm uses the power set of the set of all possible edits to find all possible intermediate states, which means that the algorithm grows at an exponential rate. This is not sustainable when the number of edits grows larger; while, in principle, the algorithm can generate a hint for a student who is far away from all previously seen states, it may not always do so quickly enough to be useful.

To evaluate the extent of this efficiency limitation, we analyzed the incorrect solution states to determine the number of edits between the current solutions and end goals. We compared these to the number of change vectors between the solution and the optimized goal, to see how much the optimization could improve the process. On average, the original goal found in the solution space was about five edits away from the solution. Looking for optimized goals decreased this distance to 2.5; more importantly, however, looking for an optimized goal greatly increased the number of goals that were only one edit away. As is shown in Figure 2, the number of edits required decreases at all levels of edit distance, which means that hints will be better targeted at what the student actually needs to do to fix their solution.

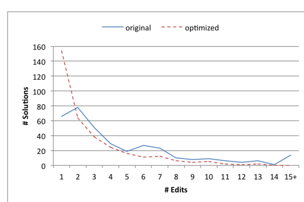

Fig. 2. A comparison of the number of edits between current solution and goal across the dataset. When the goal state is optimized, it is more likely to be only one edit away from the solution.

5.2 How long does it take generate a new feedback message?

As was mentioned before, an automatically generated hint is not particularly useful if the student does not receive it in a timely fashion. This is not a problem if the student's state has been seen before, as the hint will have been stored in the solution space, and can be delivered immediately. However, it is more interesting to look at the cases when the feedback needs to be generated.

In measuring the time taken to generate feedback, we found that the vast majority of solutions are not particularly far from their goals; 60% take less than a second to generate feedback, and 90% take less than a minute. However, we did find a few solution states which took a tremendous amount of time to run, making it infeasible to generate paths online. 12 of the 351 solutions we examined took longer than 20 minutes to run, and all but one of these had 15 or more edits between themselves and the goal. For the rest of the solutions, the time required to run the algorithm was exponentially related to the number of edits between the solution and the goal (see Figure 3). This is related to the power set generated in finding the change vectors, and thus is difficult to address.

Fig. 3. A comparison of number of edits between current state and end goal and the amount of time it takes the algorithm to generate a hint. The time grows exponentially with the number of edits, though the majority of the states are quick to generate.

5.3 Are the Chosen Goals Aligned with the Students' Intentions?

A critical question to address is whether the hints we provide will be well aligned with the students' intentions. We cannot know the student's intention from their submission alone, but we can approximate it by identifying how close the original solution states are to the final goal states. On average, the current and goal states have 64% similarity, suggesting that the solutions are somewhat close.

Since the data set we are using is composed of final submissions, it seems likely that most students should have been close to their final answer when they submitted; therefore, any great distance between a solution and its goal might reflect poor goal choice on our part, rather than a student truly being far away from a correct answer. To determine if this is the case, we examine real hints generated by the algorithm, both in cases where the goal was very close to the solution state and in cases where it was far away.

Our first example, shown to the left in Figure 4, involves a solution that only has one small bug. This solution is two edits away from the closest correct state in the graph, but only one of those edits is necessary; this is represented in the optimal goal that the graph generates. The final hint message generated, "Replace '23456789YJQKA' with '23456789TJQKA' in line 3", pinpoints the bug efficiently. On the other hand, the second example, shown on the right, is very far away from all correct states in the solution space. This would not be a problem if a closer goal could be generated, but the algorithm fails to create one. In examining the code, it is clear why this is the case- the solution has many problems, and hard to map to a goal solution. Gently suggesting that the student start over might be the best we can do.

```
def intToPlayingCard(value):
    faceValue = value%13
    face = "23456789YJQKA"[faceValue]
    suitValue = (value-faceValue)%4
    suit = "CDHS"[suitValue]
    return face+suit
```

```
def intToPlayingCard(value):
    suit = "CDHS"
    facen = "23456789TJQKA"
    suitn = ((value) / 13)
    face = (value - (13 * suitn))+2
    findsuit = suit. find [suitn]
    gaga = findsuit + face
    return gaga
```

Fig. 4. Two student solutions to a programming problem on mapping integers to playing cards. On the left, a solution that is close to its goal; the only mistake is a single typo. On the right, a solution that is far away; it requires almost a full rewrite.

6 Discussion

As we investigate whether the suggested goals are related to student intentions, we should also question whether the solution paths we are building look anything like solution paths students generate while working on their own. Naturally we do not want to make the student's experience with hints identical to their experience without them; after all, hints are supposed to improve their learning. However, we can test whether the hints provided will seem natural to students.

This question has been explored before in the field of learning analytics, through examination of several detailed case studies of student work [2]. Blikstein found that different students had different methods of approaching programming. Those who mostly focused on writing their own code made small changes while iterating on their approach. Therefore, we should also suggest small steps when possible; if we suggest that a student try a large change, they may not be willing to modify so much of their code at once.

In our future work, we aim to determine whether the hints generated by our system are truly beneficial to students. We are currently taking steps to run a study in a classroom, and plan to use the resulting data to continue improving the path construction system. As we gather more data on problems, we should be able to provide hints that are closer and closer to the students' original goals; and hopefully, with a large enough solution space, we will be able to create messages for all students, regardless of how unexpected their solutions may be.

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, T., & Stamper, J. (2008). Toward automatic hint generation for logic proof tutoring using historical student data. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems* (pp. 373-382).
2. Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st international conference on learning analytics and knowledge* (pp. 110-116).
3. Gross, S., Mokbel, B., Hammer, B., & Pinkwart, N. (2012). Feedback Provision Strategies in Intelligent Tutoring Systems Based on Clustered Solution Spaces. In *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik* (pp. 27-38).
4. Huang, J., Piech, C., Nguyen, A., & Guibas, L. (2013). Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *AIED 2013 Workshops Proceedings Volume* (p. 25-32).
5. Mokbel, B., Gross, S., Paassen, B., Pinkwart, N., & Hammer, B. (2013). Domain-Independent Proximity Measures in Intelligent Tutoring Systems. In *Proceedings of the 6th International Conference on Educational Data Mining (EDM)* (pp. 334-335).
6. Rivers, K., & Koedinger, K. (2012). A Canonicalizing Model for Building Programming Tutors. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems* (pp. 591-593).
7. Vanlehn, K. (2006). The behavior of tutoring systems. International journal of artificial intelligence in education, 16(3), 227-265.