Automating Hint Generation with Solution Space Path Construction

Kelly Rivers and Kenneth R. Koedinger

Problem

Individualized feedback and next-step hints can significantly improve student learning. However, feedback takes much time to create, and as course sizes scale it may become impossible to create feedback with manpower alone. We can create feedback ahead of time by matching student solution states to hint messages, but this does not scale with the number of possible solutions in domains with complex solution formats, such as programming. Therefore, we'd like to generate hint messages using collected data rather than expert contributions.

Solution Space

A solution space is a graph covering paths that students take while working on a complex problem. Each state in the graph is a partially-formed solution that a student might reach while working. These states can be measured by a test function which determines their correctness.

To create a solution space, we collect the states that students generate while working on a specific problem. Since student work is highly variable at a superficial level, we use canonicalization to reduce the solutions down to a normalized version

Canonicalization

Canonicalization states a student solution and runs various semantics-preserving transformations on it to preserve the function of the solution while removing any unnecessary details. In Figure 1, the student's code is anonymized (so that variable name differences don't matter). It also has an unnecessary equality check removed, and a variable assignment is propagated through the code.

```
def charCount(text):
    s = ""
    for i in range(len(text)):
        if(isLetterOrDigit(text[i]) == True):
            s = s+text[i]
    cC = len(s)
    return cC # You write this!
```

```
def charCount(v0):
    v1 = ''
    for v2 in range(len(v0)):
        if isLetterOrDigit(v0[v2]):
            v1 = (v1 + v0[v2])
    return len(v1)
```

Figure 1: A student program before and after normalization.

Path Construction

The states in the solution space can be constructed easily enough, but we still need the edges that will go between them. These edges will be edits required to get a student from one state to another, and we'll use them to generate hints. To find the best possible hint to give to a student at a particular solution state, we follow the following three steps:

Step 1: Identify Optimal Goal State

First, we need to determine what the student's intended final solution is. We do this by finding the closest correct state that currently exists in the solution space. Then, we find all the changes that exist between the current state and the correct state, and using the test function reduce them to only include those edits which are essential. These edits, when used on our current state, create the optimal goal state.

Step 2: Find Valid Intermediate States

Once we know what the start state and end state are, we can find all edits that exist between the two, and then generate all possible combinations of edits. Each combination of edits, when applied to the start state, creates a possible intermediate state that a student could next go to in their process of solving the problem. We then filter these states to remove any that are not valid, where a valid state must be well-formed, closer to the goal than the start state, and at least as correct as the start state.

Step 3: Create Optimal Edit Path

Once we've narrowed down the possible next states to include only valid ones, we choose a sequence of steps that will lead from the start state to the goal state. Each step can be delivered to the student separately, so that they have a chance to make their own changes at any point in the process. At each stage, the next step is chosen by finding the most desirable next state, where a state is desirable if it has been seen before, if it is near the current state, if it has a high score, and if it's close to the goal.

```
def findPatternAtIndex(dna, pattern, startIndex):
    count = 0
    for i in range(startIndex, len(pattern) + startIndex):
        if len(dna) - startIndex <= len(pattern):
            break
        elif symbolsMatch(dna[i], pattern[i - startIndex]) == True:
            count += 1
    return count == len(pattern)
```

```
Hint: Starting at line 1 in the function, replace the operator <= with the operator <
```

Figure 2: The hint message generated for the given student solution

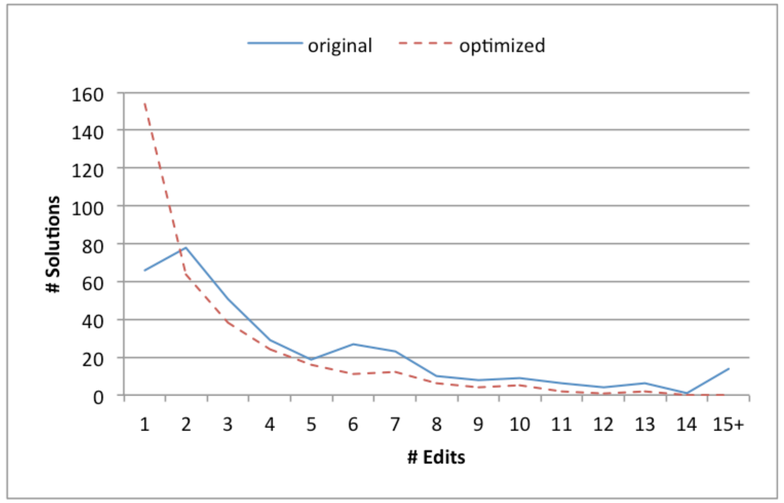

Figure 3: Number of edits required to get from an incorrect solution in the dataset to the state's goal.

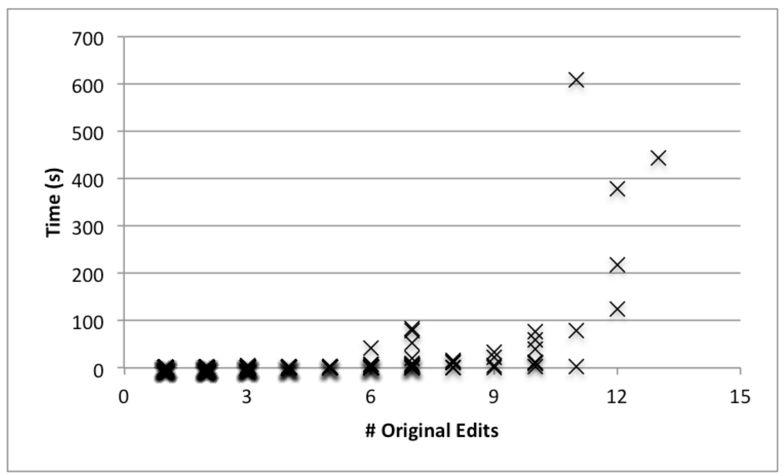

Figure 4: Time taken by the algorithm to generate a next step, based on number of edits between the state and the original goal. The algorithm grows exponentially, which makes it intractable for solutions very far away from the goal, but these far-off solutions are rare.

Carnegie Mellon University

LearnLab
Pittsburgh Science of Learning Center

Acknowledgements

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

pier @ Carnegie Mellon
PROGRAM FOR INTERDISCIPLINARY EDUCATION RESEARCH

HCII