Generating Data-driven Hints for Open-ended Programming

Thomas W. Price
North Carolina State Univ.
890 Oval Drive
Raleigh, NC 27695
twprice@ncsu.edu

Yihuan Dong
North Carolina State Univ.
890 Oval Drive
Raleigh, NC 27695
ydong2@ncsu.edu

Tiffany Barnes
North Carolina State Univ.
890 Oval Drive
Raleigh, NC 27695
tmbarnes@ncsu.edu

ABSTRACT

Intelligent Tutoring Systems (ITSs) have shown success in the domain of programming, in part by providing customized hints and feedback to students. However, many popular novice programming environments still lack these intelligent features. This is due in part to their use of open-ended programming assignments, which are difficult to support with existing hint generation techniques. In this paper, we present a new data-driven algorithm, based on the Hint Factory, to generate hints for these open-ended assignments. We evaluate our algorithm on historical student data and show that it can provide hints that successfully lead students to solutions from any state, help students achieve assignment objectives, and align with the student's future solution.

1. INTRODUCTION AND BACKGROUND

Intelligent Tutoring Systems (ITS) have shown much promise in the domain of computer programming [3, 14, 16, 22], with studies arguing that students using an ITS perform as much as two standard deviations higher than those who receive conventional instruction [3]. A key feature of any ITS is the ability to give students context-sensitive feedback during problem solving, often in the form of hints. In the domain of programming, this feedback has been shown to improve students' performance, both inside the tutor and on subsequent assessment [4].

Despite positive empirical evaluations, these specialized ITSs are not generally used in introductory programming classes. In particular, new introductory Computer Science (CS) curricula, such as CS Principles[1] and Exploring CS[2] are turning to programming environments designed specifically for novices, such as Scratch [19], Snap [8] and Alice [5], which engage students in creating open-ended projects, such as games, stories and simulations [25]. These environments have features specifically designed for novices, such as drag-and-drop, block-based interfaces that improve student performance by minimizing the challenges of syntax [18]. They offer improved outcomes over traditional instruction, such as increased retention [11] and improved test scores [5].

Unfortunately, aside from some preliminary research [2], little effort has been made to bring the intelligent features of ITSs to these novice programming environments. This is due in part to the large investment of time required by domain experts to create these systems, which has been estimated as high as 300 hours to create one hour of intelligent content [12]. Further, the use of open-ended programming assignments, which makes these environments so appealing to students and teachers, also serves as a major barrier to providing intelligent, adaptive feedback. These assignments often have multiple, loosely ordered objectives, which cannot be assessed automatically, making it difficult to apply automatic hint generation techniques that rely on test cases (e.g. [15, 22, 23]).

Data-driven tutors have the potential to overcome these barriers. The Hint Factory is an algorithm that has been used to generate data-driven hints from historical student data, originally in the domain of logic proofs [1]. The Hint Factory is like a recommender system that uses student data as a basis for automatic hint generation, making it easy to scale up without additional expert involvement. The Hint Factory has been successfully adapted to the domain of programming in a variety of ways [14, 9, 22]. However, data-driven hints have not been evaluated on open-ended assignments in novice programming environments, and may not be well equipped to handle them [17].

In this paper we present an extension of the Hint Factory specifically designed to provide hints to students working on open-ended programming assignments. The algorithm is fully data-driven, requiring no reference solution or test cases, and presents hints that represent real student actions. It is designed to be programming language and system agnostic, with the intention of making it applicable to a variety of novice programming environments. We evaluate this algorithm on historical student data from an open-ended assignment in a novice programming environment, and show that it is capable of providing hints that successfully lead students to solutions from any state, help students achieve assignment objectives, and align with the student's future solution.

1.1 The Hint Factory

The Hint Factory [24] is an algorithm for generating next-step hints for students working on multi-step problems. It operates on a data-structure called an interaction network [6], which is built from log data of the interactions between students and a learning environment for a given problem. The interaction network is a directed graph, where each vertex represents a state of the problem. In programming a state corresponds to a snapshot of the student's current work (code). States are connected by edges, which represent student actions, such as adding, editing or deleting code, which transform one state into another. Each student attempt is traced from a start state to its final state and is added to the interaction network. If this final state is a correct solution, we label it as a goal state. By combining all students' attempts into a single network and weighting edges

[1] www.csprinciples.org
[2] www.exploringcs.org

with the number of attempts that passed through them, the interaction network forms a compact representation of student problem solving strategies for a given problem.

The Hint Factory uses the interaction network for a given problem to generate hints for new students working on that problem. When a student requests a hint, the algorithm matches that student to an existing state in the network and then calculates the best path from that state to a goal state. The Hint Factory uses a Markov Decision Process (MDP) to calculate this solution path [1], but other techniques can also be used, which are more effective in some contexts [16]. Once a solution path is calculated, it is typically used to provide a next-step hint, which points the student towards the next state in the solution path. The exact method of suggesting this state as a hint is system-dependent.

1.2 Hint Generation in Programming

The domain of computer programming presents a serious challenge for automatic hint generation, especially for data-driven systems. Even for simple programming problems, the space of possible solutions is quite large, often infinite, and there may be little overlap among student solutions [17, 20]. Many automated hint generation algorithms search through this space, attempting to transform a student's current program into a solution state using some sort of program generation or synthesis [10, 15, 22, 23, 26]. These techniques require an expert-supplied reference solution and/or set of test cases to ensure that generated programs are correct. To facilitate this transformation, algorithms often represent a student's program using an Abstract Syntax Tree (AST), a directed, rooted tree where each node represents a program element, such as a function call, control structure or variable, and the hierarchy of the tree represents how these elements are nested together.

Zimmerman and Rupakheti [26] use a pq-Gram tree edit distance algorithm to match a student's program to its closest counterpart in a database of target solutions, as well as to identify the set of insertions, deletions and relabelings that will directly transform the student's AST into this solution. Rather relying on a fixed set of solutions, Singh et al. [23] use program synthesis to generate a new solution from the student's current program. They do so using an expert-provided Error Model, which defines a set of potential transformations to a student's code for a given problem. Other techniques are data-driven like the Hint Factory, using previous student solutions to provide hints. Perelman et al. [15] also employ program synthesis to search for a solution program, using a Domain-Specific language (DSL) to define possible program transformations; however, they show that this DSL can be automatically generated from previous student solutions. Our approach also works to transform a student's program into a solution, but rather than using an automated technique like program synthesis, we use edits from actual students. Lazar and Bratko [10] employ a similar approach, applying single-line edits observed in previous student work to transform a student's program into a solution; however, their technique requires a set of test cases to evaluate generated programs, and ours does not.

The Hint Factory has also been adapted to the domain of programming, with modifications to address the large state space and lack of overlap among student solutions. Rivers and Koedinger [22] extend the Hint Factory using a strategy called path construction to generate a path from a student's current state to a previously observed goal state, rather than relying on observed student paths. They compute a change vector of all edits needed to transform the student's current state into the goal state and test to see if any closer solutions are discovered along the way. Peddycord III et al. [14] applied the Hint Factory to a programming game called BOTS, but rather than representing a student's state using an AST (a *codestate*), they used the state of the game world after running the student's program (a *worldstate*). The authors found considerably more overlap among worldstates than codestates, allowing more hints to be generated; however, these hints may be more challenging to apply. Fossati et al. [7] used a similar approach to the Hint Factory to generate both reactive and proactive data-driven feedback in the iList linked list tutor. They found that with this feedback, iList produced equivalent learning gains to a human tutor.

Most methods for hint generation benefit from overlap among student programs. This overlap can be increased through canonicalization, which standardizes the *syntax* of programs, while maintaining their *semantic* meaning. For example, the expression $a > b$ can be rewritten $b < a$ without changing its meaning. Rivers and Koedinger [20] present a comprehensive technique for canonicalization, which standardizes programs in a variety of ways, such as normalizing arithmetic and boolean operators, removing unreachable and unused code and inlining helper functions. Jin et al. [9] take a different approach, representing a student's program as a Linkage Graph, where each vertex is a code statement, and each directed edge represents an ordering dependency. This removes some semantically unimportant ordering information from the program, allowing for more overlap.

2. THE CTD ALGORITHM

In this section we present the Contextual Tree Decomposition (CTD) algorithm for hint generation, our extension of the Hint Factory to the domain of open-ended programming problems. Existing hint generation techniques are effective on traditional programming assignments with single objectives that are easily assessed with test cases. Open-ended assignments, by contrast, may have multiple, loosely ordered objectives that do not lend themselves to automated assessment, as they often deal with user interaction or graphical output. As such, we cannot rely on the program generation techniques discussed in Section 1.2 to create hints. Instead, we take a fully data-driven approach, using student data, rather than automated search, to construct a path to a goal state. Not only does this approach make hint generation feasible for open-ended assignments, it also has the advantage of presenting hints that correspond to real student actions, which should be understandable to other students.

2.1 An Example Assignment

To illustrate the CTD algorithm, we will use an assignment called the "Guessing Game" as a running example throughout this section. In the Guessing Game, students are asked to create a program that stores a random number and then repeatedly asks the player to guess it until they are correct, informing them if they have guessed too high or too low. To begin, the game should welcome the player and greet them by name. The assignment requires the use of loops, conditionals, variables and various arithmetic operators. A common implementation of the Guessing Game is presented in Figure 1.

Note that this is one of many possible solutions to the problem. For example, we could use three `if` statements, rather than an `if/else` block. Now consider a student, Alice, working on

```
GuessingGame:
   Say( "Welcome to the Guessing Game!" )
   answer ← Ask( "What is your name?" )
   Say( Join( "Hello ", answer ) )
   number ← Random( 1, 10 )
   doUntil ( answer == number ):
      answer ← Ask( "Guess a number" )
      if ( answer == number ):
         Say( "Correct!" )
      else:
         if ( answer > number ):
            Say( "Too high!" )
         if ( answer < number ):
            Say( "Too low!" )
```

Figure 1: An example solution to the Guessing Game assignment.

```
GuessingGame:
   number ← 8
   Say( "Welcome!" )
   answer ← Ask( "Who's playing?" )
   Say( Join("Hi ", answer ) )
   doUntil ( answer == Random( 1, 10 ) ):
      answer ← Ask( "Guess a number" )
```

Figure 2: An example of a partial, flawed solution attempt from a student, Alice.

the Guessing Game with code presented in Figure 2. Alice has added the first few lines of code in a different (but correct) order; however, she does not understand how to store and use the random number for the guessing game. A hint could demonstrate the correct behavior for her.

2.2 Generating Hints

In the CTD algorithm, as in previous work, we represent a student's state using an AST. Borrowing from Rivers and Koedinger's work [20], we also use basic canonicalization to increase overlap among ASTs. In our ASTs, we use a single label for all variables (`var`) and for all literals (`literal`). The arguments of commutative operators (e.g. `==`, `+`, `*`) are given a fixed ordering, and we rewrite any greater than expression $x > y$ as a less than expression $y < x$. A canonicalized AST for the code presented in Figure 1 is shown in Figure 3.

Most data-driven hint generation algorithms attempt to answer the question, "Given a student's current state, what should their next state be?" Rather than trying to answer this question for a student's entire program, we try to answer it for the children of each node of a student's AST. For example, if Alice were to request a hint, we might tell her to assign a different value to `number`, compare different values using `==` or add code to the body of `doUntil`. By breaking the student's program down into a set of smaller pieces, we can more easily match it to the programs of previous students, as suggested in previous work [10, 21].

To generate hints from student data, we build a *set* of *contextual* interaction networks (CINs), which each model how students edit a subsection of the program over time. We build one CIN for

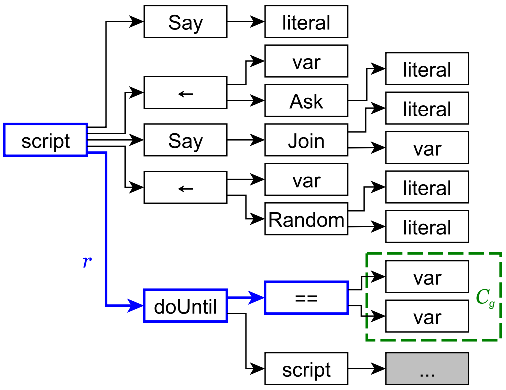

Figure 3: A partial AST for the code shown in Figure 1. A root path r is outlined in bold blue, with its current state (C_g) in dashed green.

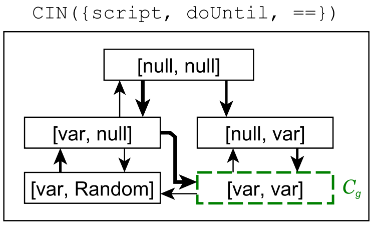

Figure 4: The contextual interaction network CIN(`{script, doUntil, ==}`) with goal state C_g. Edge thickness represents transition frequencies.

each unique *root path* observed in all students' ASTs (including ASTs from intermediate code snapshots). A root path (RP) for a node n in an AST is the path from the root node to n. Figure 3 highlights an example RP for the `(==)` node: `{script, doUntil, ==}`. Some nodes have the same root path, such as the two `(Say)` nodes, which have the RP `{script, Say}`. Each RP r corresponds to a unique CIN, denoted CIN(r), which functions just as the interaction networks described in Section 1.1. However, CIN(r) only models changes to the *immediate children* of the last node in r. For example, CIN(`{script, doUntil, ==}`), shown in Figure 4, models changes to the children (operands) of the `(==)` node. Each state in CIN(r) is a list of the children of the last node in r, and each edge represents an edit to those children. Figure 3 highlights C_g, the list of children of the `(==)` node, which corresponds to a state in the CIN shown in Figure 4. Because the AST shown in Figure 3 is a correct solution, C_g is a goal state in CIN(`{script, doUntil, ==}`). Given that Alice's current state in this CIN is `[var, Random]`, to get to the goal state C_g we would recommend that she delete her `(Random)` node and then replace it with a `(var)` node.

The procedure for building the CINs from previous data is shown in Algorithm 1. We represent a student's work as a sequence of ASTs, T, where each tree t_i in the sequence is a snapshot of the student's work at time i, and the last tree represents the submitted solution attempt. For each sequential pair of trees, t_i and t_{i+1}, we find all pairs of AST nodes (n_i,n_{i+1}) that represent the same code element in both trees, and therefore have the same RP r. We examine the lists of child nodes C_i of n_i and C_{i+1} of n_{i+1} in their respective ASTs. If C_i and C_{i+1} are different, we add the states C_i and C_{i+1} to CIN(r) (if they do not already exist) and add an edge from C_i to C_{i+1}. This edge represents how the student has edited the code in this part of the AST from time i to time $i+1$. Algorithm 1 runs in $O(|T||t_m|^2)$ time for a given student, where $|T|$ is the number of ASTs recorded for that student and $|t_m|$ is size of the largest recorded AST.

Algorithm 1 Add a Student to the CINs

```
Require: A sequence of student ASTs $T$
Ensure: Student data has been added to relevant CINs
  for all $t_i, t_{i+1} \in T$ do
      for all $(n_i, n_{i+1}) \in$ MatchingNodes($t_i$, $t_{i+1}$) do
          $r \leftarrow$ RootPath($n_i$)
          $C_i \leftarrow$ Children($n_i$)
          $C_{i+1} \leftarrow$ Children($n_{i+1}$)
          if $C_i \neq C_{i+1}$ then
              AddEdge(CIN($r$), $C_i$, $C_{i+1}$)
          end if
      end for
  end for
```

Once we have added student data to the CINs, we can generate hints for new students, as shown in Algorithm 2. Because we now have many CINs, rather than a single interaction network, we also generate a *set* of hints. For each node n in a student's current AST, we calculate its root path r and find CIN(r). The student's current state in CIN(r) is C_0, the list of children of n. We then use the Hint Factory algorithm [1] to generate a hint using the interaction network CIN(r) and the student's current state in the network C_0. This hint will recommend a new set of children C_1 for n, which we can then display as a suggestion to the student. Note that if C_0 is already a goal state the Hint Factory will recommend that the student stay in that state, in which case $C_0 = C_1$ and we present no hint for n. Algorithm 2 runs in $O(|t|^2 + |t||S_m|^2)$ time [3], where $|t|$ is the size of the student's AST and $|S_m|$ is the number of states in the largest CIN(r). In practice, $|S_m|$ remains small, as a given CIN models changes to only a small part of a student's code.

Algorithm 2 Get Hints

```
Require: The student's current AST $t$
Ensure: $H$ is a set of node-hint pairs
  $H \leftarrow \{\}$
  for all $n \in$ Nodes($t$) do
      $r \leftarrow$ RootPath($n$)
      $C_0 \leftarrow$ Children($n$)
      $C_1 \leftarrow$ HintFactoryHint(CIN($r$), $C_0$)
      $H \leftarrow H \cup \{(n, C_1)\}$
  end for
```

A classic challenge for the Hint Factory is how to provide hints to states with no exact matches in the interaction network. CINs break a program down into smaller parts to provide more opportunities for matches, but this does not guarantee a match. If no exact match is found for a state C_0, we find the closest state to C_0 in the CIN and use it as a next-step hint. Because CIN states are lists of children, we can use a simple edit distance to determine the closest state. If the distance between the current state and its closest pair in the CIN is beyond a certain threshold (e.g. 3 edits), we assume the student is doing something unknown, and we do not provide a hint for that state.

2.3 Goal States

In order to run on an interaction network, the Hint Factory requires a reward function $R(s)$, which is used by the MDP to assign a reward to each state in the network [1]. Traditionally, this value has been some large number (e.g. 100) for goal states and 0 otherwise. However, in many open-ended programming problems, we cannot automatically determine whether or not a given program state satisfies the goal of the assignment. A simple solution is to assign a reward value to each state proportional to the number of students who submitted a program in that state. We accomplish a similar effect with CINs by finding each node n and corresponding RP r in each student's submitted AST and marking the list of children of n as a goal in CIN(r).

One challenge with CINs is that two different parts of a program may correspond to the same CIN. For example, recall that the two `Say` statements in Figure 3 have the same RP, and thus the same CIN, but ideally these two nodes should end up in two different goal states. The first should end up with children `[literal]`, while the second should have children `[Join]`. Both of these states will be marked as goals in the shared CIN, so how can the algorithm determine when one goal should be chosen over the other?

To address this, each time a node's children are marked as a goal state in a CIN, we also store that node's *context*. This context helps identify when a particular goal state might be applicable. We define a node's context using two lists, consisting of its left and right siblings in the AST. For example, in Figure 3, the first `Say` node has a context `{[], [←, Say, ←, doUntil]}`, while the second node has a context `{[Say, ←], [←, doUntil]}`. Rather than giving goal states a fixed reward value, we determine this value individually for each hint request. For each previous student attempt that finished in a given goal state, we increase the reward for that state by a value inversely proportional to the distance between the previous attempt's context and the current attempt's context. Again, because the contexts consist of lists, a simple edit distance can serve as a distance metric.

2.4 Smoothing Hints

The Hint Factory is typically used to generate a next-step hint, which suggests the next state a student should achieve. The advantage of the Hint Factory is that this action has been done by a previous student, and is therefore likely to seem reasonable to the current student. However, sometimes the path that a real student takes to a solution can be circuitous. Students often add code that they later delete, or add code in one place and later move it to another. In these cases we use the entire *solution path* generated by the MDP, rather than a single state, to make suggestions that will not be contradicted by future hints. We call this process "smoothing", since it will make hints appear more consistent.

We use Algorithm 3 to generate hints which follow real students' paths, while avoiding unnecessary or contradictory edits. We first calculate a full solution path from the student's current state to a goal state using the Hint Factory on the CIN, as described

[3] This assumes we use a constant bound on the number of iterations allowed during the Hint Factory's value iteration.

earlier. Recall that each state in this path is a list of child nodes in the AST. We first reorder nodes in the student's current state to match the goal state ordering. We then insert any new nodes from the next state in the solution path (like set union) and reorder the nodes again to match the goal state. Finally, we remove any nodes that are not in the goal state (like set intersection). If the resulting state is not different than the student's current state, we repeat the process with the next state in the solution path. Using this "smoothing" process helps us avoid giving hints that add code that will later need to be moved or deleted.

Algorithm 3 Get Smoothed Hint

```
Require: The MDP of a CIN and student's $state$
Ensure: $hint$ is a smoothed hint for the student
  $path \leftarrow$ GetSolutionPath($state$, MDP)
  $goal \leftarrow$ Last($path$)
  $hint \leftarrow state$
  $hint \leftarrow$ Reorder($hint$, $goal$)
  for all $s_i \in path$ do
      $hint \leftarrow hint \cup s_i$
      $hint \leftarrow$ Reorder($hint$, $goal$)
      $hint \leftarrow hint \cap goal$
      if $hint \neq state$ then
          return
      end if
  end for
```

3. METHODS

We evaluated the efficacy of the CTD algorithm using data from real students working on the Guessing Game assignment described in 2.1. Data was collected from an introductory undergraduate computing course for non-CS majors during the Fall of 2015, which had approximately 80 students. The first half of the course focused on learning the Snap programming language through a curriculum based on the Beauty and Joy of Computing (BJC) [8]. Snap is a visual programming environment that allows users to create media-rich, interactive programs by dragging blocks of code together to form scripts. Students worked on the Guessing Game assignment during class for approximately one hour, with a teaching assistant (TA) available to assist them and the ability to discuss the assignment with nearby students. We collected trace log data of all student interactions with the programming environment. After each edit to a student's program, the complete program state (a snapshot) was recorded. For the "Guessing Game" assignment, we collected 51 attempts, consisting of 8666 total code snapshots.

Each of the final submissions was graded by two independent graders. The graders used a rubric consisting of nine assignment objectives, such as welcoming the player by name, storing a secret number, and repeatedly asking the player for guesses. The graders had an initial agreement of 94.5%, with Cohen's $\kappa = 0.544$, and after clarifying objective criteria and independently re-grading this rose to 98.1%, with Cohen's $\kappa = 0.856$. Any remaining disagreements were discussed to create final grades for each assignment. The students achieved on average 92.8% of objectives, with all students getting at least 4 out of 9. The high grades can be attributed in part to the presence of TAs, who helped struggling students to complete the assignment. Using the same criteria, an automatic grading program was created, which manually checked code structure for objective completion. The automatic grader was tested on the manually graded data, achieving 100% accuracy on 7 of 9 objectives. On each of the remaining two objectives, it incorrectly marked two submissions as failing since they used atypical approaches. Note that this grader was used in our evaluation but not for hint generation.

We generated and evaluated hints for each code snapshot of each student in our dataset (n=8666), giving us a clear view of hint performance across students and time. We evaluated the hints using a number of criteria, detailed in Section 4. Because Snap lends itself to a "tinkering" approach, code snapshots often contain many extra scripts that students keep in their workspace for later use. Since the Guessing Game uses only one script, these extra scripts do not reflect the student's primary work, and it would not make sense to evaluate hints for them. Therefore, in our analyses we considered only the largest script in a snapshot.

3.1 Hint Policies

To better evaluate the CTD algorithm, we generated hints using four hint policies:

1. **CTD All** (CA): Hints are generated using CTD on all student data (n=51).
2. **CTD Exemplar** (CE): Hints are generated using CTD on data from only exemplar students, whose final submissions achieved all assignment objectives (n=32).
3. **Direct Expert** (DE): Hints modify a student's program directly towards an expert solution using a single node insertion, deletion or relabeling.
4. **Direct Student** (DS): Hints modify a student's program directly towards their own submitted solution, using a single node insertion, deletion or relabeling.

The CA and CE policies both use the CTD algorithm, and comparing them allows us to explore the effect of including students with incorrect final solutions on the algorithm's output. The DE and DS policies both generate hints using a technique outlined by Zimmerman et al. [26], which identifies the node insertions, deletions and relabelings required to transform one AST into another. Each of these modifications is treated as a hint. The DE policy targets a single expert solution, while the DS policy targets the student's own future final solution, and could not actually be implemented on real-time data. In many ways, the DS policy represents an ideal hint policy for students who achieve a correct final solution (which the majority of our sample did), as it perfectly anticipates their solution strategies.

All policies generate a set of hints, where each hint represents a small modification to a student's program. When generating hints for a given student using CTD, we did not include that student in the dataset used to build the CINs, similar to leave-one-out cross validation, since that student's future data could not be used in a real-time setting.

4. EVALUATION AND DISCUSSION

Our evaluation of CTD focused on the following research questions:

RQ1 Can CTD successfully lead students to a solution regardless of their current program?

RQ2 Can CTD hints help students complete objectives?

RQ3 How consistent are CTD hints with student actions?

RQ1 asks whether CTD solves the challenge of generating hints for an open-ended problem where there is little exact overlap among student solution paths. RQ2 investigates whether these hints are good in that they leads student to complete assignment objectives. Lastly, RQ3 asks whether the hints that CTD provides point students in what might be perceived as a reasonable direction, so students will be inclined to use them. Our evaluations for RQ2 and RQ3 compared the CA and CE policies with the baseline DE and DS policies discussed in Section 3.1.

4.1 Providing Hints

RQ1 asks whether CTD can successfully generate hints for solution attempts regardless of how much overlap they have with other attempts in our dataset. Therefore, we first examined how much overlap there was in our dataset. We recorded 8666 snapshots from 51 students; however, many students produced duplicate snapshots, for example by adding and then removing an element of code. If we do not count duplicate snapshots from the same student, we are left with 5103 snapshots. If we also ignore all but the largest script from these snapshots (as is done in our analyses), there are 3181 non-duplicate snapshots. Of these, 2714 (85%) were unique after canonicalization, meaning they showed up in only one student's data. In addition, 47 of 51 students had unique final solution ASTs. We conclude that the state space is quite sparse, with little overlap among student solution paths.

We evaluated hints from the CA and CE policies to determine if they could get students to a solution despite this sparsity. To align student attempts over time, and to balance our sample evenly across students, we took 50 snapshots from each student, spaced evenly throughout their progression, and called these "slices." For each student, we generated a *hint chain* from each of these 50 snapshots to a final solution. A hint chain is the sequence of program states that would result if the students followed sequential "top-level" CTD hints from a given snapshot to program completion. The top-level hint is that which comes from the CIN(r) with the shortest RP r.

Both CA and CE policies were able to generate successful hint chains for every slice, meaning the hint policies always had a hint to provide and there were no hint cycles. Figure 5 shows the average hint chain length for each slice. Both policies showed a steady, near-linear decline in hint chain length over time. This supports the notion that CTD makes good use of the student's existing work. On average, students took 175.8 steps to complete the assignment, so both policies are more efficient than the student until slice 46/50. As students converge on their own solutions, however, the hints chains become less efficient, as they often lead students to alternative solutions.

To understand the quality of solutions created using hint chains, we evaluated the final solutions generated by the hint chains at each slice using the automatic grader discussed in Section 3. The CA policy solutions received grades averaging from 89.5-93.0% across slices, while the CE policy averaged 98.5-100% across slices. Upon closer inspection, we found that all imperfect CA solutions were identical and satisfied 8 of 9 objectives (88.9%), and all correct CA solutions were identical as well. The one objective missed by the imperfect CA solution was also missed by 12 of 51 students (23.5%), indicating that a frequent enough mistake in student data will be reflected in CTD hints. The CE policy produced 3 unique, correct solutions and 2 unique, incorrect solutions, which both satisfied the same 8 out of 9 objectives. These results suggest that both CTD policies can lead students to high-quality (though sometimes imperfect) final solutions, but exemplar data may be required to generate consistently correct solutions. It is important to note that CTD operates without test cases, and therefore cannot guarantee correctness 100% of the time.

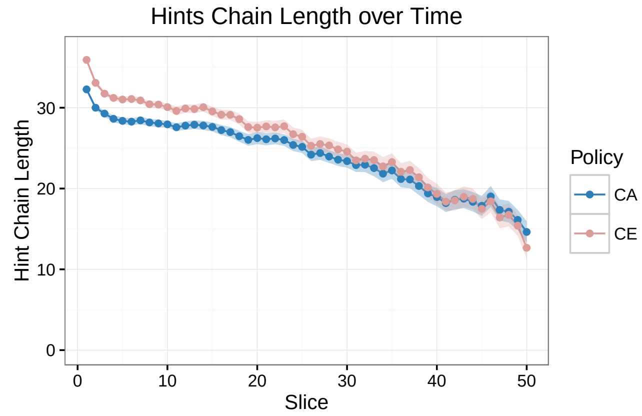

Figure 5: The average CA and CE hint chain length across all snapshot slices. The shading indicates standard error.

4.2 Objective Satisfaction

To address RQ2, we tested how frequently an available hint would complete an assignment objective before the student did. Figure 6 shows, for each policy in Section 3.1, the percentage of students who had an objective completing hint available for each objective. All hint policies perform fairly well, with at least 45% of students having a completion hint available for objectives 3-9. The CTD policies perform much worse on Objective 2, but otherwise they generally keep pace with the Direct policies. Since these Direct policies offer *all* edits towards their target solution as hints, they should discover most of the possible completing hints. However, it is important to remember that it is not always possible to complete an objective before a student because hints cannot add more than one node to the AST at a time, while a student's edit might change many nodes at once by dragging and dropping code.

It is not sufficient for a hint policy to generate good hints; it is equally important that it *not* generate *bad* hints. To evaluate this second facet of RQ2, we tested how frequently hints from each policy undid an objective, meaning the objective was satisfied before applying the hint, but it became unsatisfied afterward. Figure 7 compares each policy, showing the percentage of students who received a hint that would undo each objective. Predictably, the DS policy, which anticipates a student's final solution, performs well across the board. However, the difference between the DE and CE policies is clear. The CE policy stays below 40% on all objectives, and performs as well or better than the DE policy on all but one objective, often by a factor of 2 or more. The CA policy performs slightly worse than the CE policy on most objectives, most notably Objective 1. This can be attributed to the fact that many students did not in fact complete Objective 1, leading the CA policy to suggest removing the code that did so.

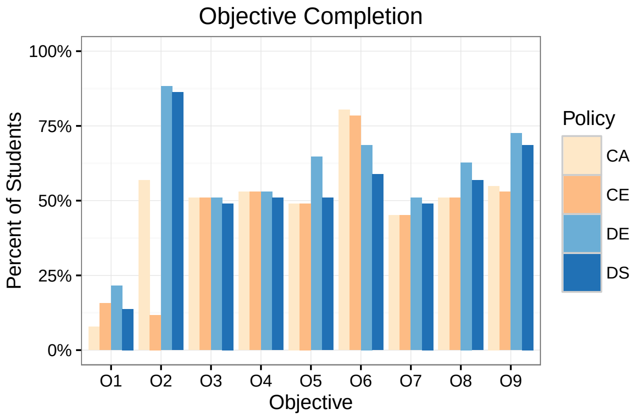

Figure 6: The percent of students who received a hint that completed an objective before the student did under each policy.

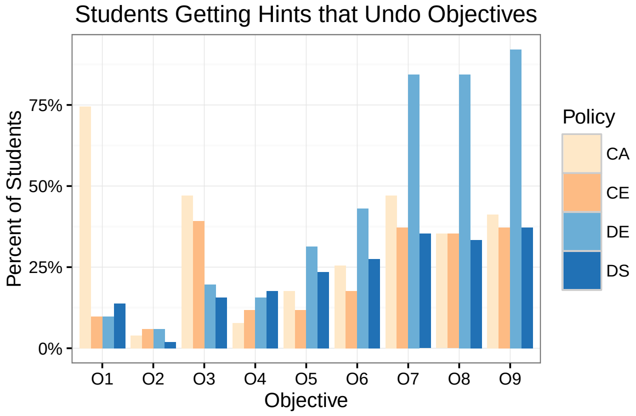

Figure 7: The percent of students who received a hint that undid an objective under each policy.

We do not make the claim that a good hint always completes an assignment objective, nor that undoing an objective always constitutes a bad hint. Still, these criteria serve as good baseline standards for a hint policy. While all policies are fairly successful at suggesting hints that move students toward completing objectives, the CTD and DS policies avoid undoing objectives much better than the DE policy.

4.3 Alignment with Student Actions

RQ3 asks whether or not CTD produces hints which are consistent with a student's solution path. Ideally, a hint policy should not only provide hints which lead to a good solution; as much as possible, these hints should also make sense to the student receiving them. While the comprehensibility of a hint is impossible to measure without user data, we can approximate this by asking whether or not a hint gets the student closer to their future final solution. Presumably, such hints will seem reasonable to the student, as the student eventually went in that direction on their own.

To answer this question, we examined each hint generated with each policy across all code snapshots and calculated whether or not each hint would get the student closer to their final solution than their original state. We used the Robust Tree Edit Distance algorithm [13] to measure the distance between snapshot ASTs. This metric counts the number of insertions, deletions and relabelings required to transform one AST into another. As a baseline, we also calculated this measure for the student's own next state, to determine how frequently a student's actions got them closer to their own final solution state. The results for each policy, averaged over students, are presented in Table 1.

Policy	Closer	SD
CTD All	35.47%	17.50%
CTD Exemplar	32.52%	15.88%
Direct Expert	21.49%	10.02%
Direct Student	39.37%	13.60%
Student Next	60.97%	8.42%

Table 1: The percent of hints under each policy that would bring the student closer to their final solution, averaged over students. Student Next refers to the student's actual next action.

As a baseline, we see that the student's own next step got closer to their final solution 60.95% of the time. The DS policy, which attempts to directly transform the student's state into their solution state, achieves only 39.37%, in part because its hints will often delete useful code and later add it again in a better location. However, the DS policy's performance might be seen as a high target, as it requires future knowledge of the student's actions. In comparison, we see that the CTD policies both approach the DS policy and far outperform the DE policy. The CE policy gets students closer to their final objective 53.4% as frequently as the student's own actions and 82.6% as frequently as the DS policy, and the CA policy performs even better. Post hoc paired t-tests showed that the difference between the CA and DS policies was not significant ($t(50) = -1.63$; $p = 0.109$), while the difference between the CA and DE policies was significant ($t(50) = 6.96$; $p < 0.001$). Interestingly, the difference between the CA and CE policies was also significant ($t(50) = 2.67$; $p = 0.010$), suggesting that restricting data to exemplar students makes CTD hints less reflective of real student behavior. While all policies present some hints that move the student farther away from their final solution, the CTD and DS policies seem to minimize this behavior.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel algorithm called CTD for generating next-step hints for students working on open-ended programming assignments. Using data from 51 students working on one such assignment, we have shown that the hints generated by the CTD hint policies can get a student to a high-quality solution from all observed states. We have also shown that the hints are capable of helping students accomplish most assignment objectives before they would otherwise do so, without presenting many hints which undo these objectives. Further, CTD produces hints which get students closer to their final solutions relatively frequently. We have also compared the CA policy, which uses all student data, to the CE policy, which uses exemplar data only. While both policies perform well, the CA policy aligns closer with real student actions, while the CE policy produces higher quality final solutions and is less likely to suggest undoing assignment objectives.

Despite these positive initial results, much work remains to be done to improve CTD. A major limitation of this work is the reliance on a single assignment for evaluation. Future work will explore the efficacy of CTD with a variety of assignments. One challenge that will be presented by larger assignments is ensuring that the contextual goal matching features discussed in Section 2.3 work for programs with multiple scripts. Additionally, while CTD incorporates some of the strategies of the other hint generation algorithms discussed in Section 1.2, such as canonicalization, there are others, such as Rivers and Koedinger's path construction [22], which could also be incorporated. Because the CINs are simply small interaction networks, any advances to the Hint Factory can also be applied to them. Lastly, we have already incorporated our hints into the Snap environment, and future work will investigate how they impact real students. We will explore the effect of CTD hints on students' performance on assignments, as well as their learning gains.

6. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1432156.

7. REFERENCES

[1] T. Barnes and J. Stamper. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Proc. of the 9th Int. Conf. on Intelligent Tutoring Systems*, pages 373–382, 2008.

[2] S. Cooper, Y. J. Nam, and L. Si. Initial Results of Using an Intelligent Tutoring System with Alice. In *Proc. of the 17th Annual ACM ITiCSE Conf.*, pages 138–143. ACM Press, 2012.

[3] A. Corbett. Cognitive Computer Tutors: Solving the Two-Sigma Problem. In *Proc. of the 8th Int. Conf. on User Modeling*, pages 137–147, 2001.

[4] A. Corbett and J. Anderson. Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *Proc. of the SIGCHI Conference on Human Computer Interaction*, pages 245–252, 2001.

[5] W. Dann, D. Cosgrove, and D. Slater. Mediated transfer: Alice 3 to Java. In *Proc. of the 43rd Annual ACM SIGCSE Conf.*, pages 141–146, 2012.

[6] M. Eagle, M. Johnson, and T. Barnes. Interaction Networks: Generating High Level Hints Based on Network Community Clustering. In *Proc. of the 5th Int. Conf. on Educational Data Mining*, pages 164–167, 2012.

[7] D. Fossati, B. Di Eugenio, S. Ohlsson, C. Brown, and L. Chen. Data Driven Automatic Feedback Generation in the iList Intelligent Tutoring System. *Technology, Instruction, Cognition and Learning*, 10(1):5–26, 2015.

[8] D. Garcia, B. Harvey, and T. Barnes. The Beauty and Joy of Computing. *ACM Inroads*, 6(4):71–79, 2015.

[9] W. Jin, T. Barnes, and J. Stamper. Program Representation for Automatic Hint Generation for a Data-driven Novice Programming Tutor. In *Proc. of the 11th Int. Conf. on Intelligent Tutoring Systems*, pages 1–6, 2012.

[10] T. Lazar and I. Bratko. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Proc. of the 12th Int. Conf. on Intelligent Tutoring Systems*, pages 306–311. Springer, 2014.

[11] B. Moskal, D. Lurie, and S. Cooper. Evaluating the Effectiveness of a New Instructional Approach. *ACM SIGCSE Bulletin*, 36(1):75–79, 2004.

[12] T. Murray. Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art. *Int. Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

[13] M. Pawlik and N. Augsten. RTED: A Robust Algorithm for the Tree Edit Distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[14] B. Peddycord III, A. Hicks, and T. Barnes. Generating Hints for Programming Problems Using Intermediate Output. In *Proc. of the 7th Int. Conf. on Educational Data Mining*, pages 92–98, 2014.

[15] D. Perelman, S. Gulwani, and D. Grossman. Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments. In *Proc. of the Workshop on Data Mining for Educational Assessment and Feedback*, 2014.

[16] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proc. of the 2nd ACM Conf. on Learning @ Scale*, pages 1–10, 2015.

[17] T. W. Price and T. Barnes. An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem. In *Proc. of the Workshop on Graph-Based Data Mining held at EDM'15*, 2015.

[18] T. W. Price and T. Barnes. Comparing Textual and Block Interfaces in a Novice Programming Environment. In *Proc. of the Int. Computing Education Research Conference*, 2015.

[19] M. Resnick, J. Maloney, H. Andrés, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[20] K. Rivers and K. Koedinger. A Canonicalizing Model for Building Programming Tutors. In *Proc. of the 11th Int. Conf. on Intelligent Tutoring Systems*, pages 591–593, 2012.

[21] K. Rivers and K. Koedinger. Automatic Generation of Programming Feedback: A Data-driven Approach. In *Proc. of the First Workshop on AI-supported Education for Computer Science*, pages 50–59, 2013.

[22] K. Rivers and K. R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *Int. Journal of Artificial Intelligence in Education*, 2015.

[23] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. *ACM SIGPLAN Notices*, 48(6), 2013.

[24] J. Stamper, M. Eagle, T. Barnes, and M. Croy. Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. *Int. Journal of Artificial Intelligence in Education*, 22(1):3–17, 2013.

[25] I. Utting, S. Cooper, and M. Kölling. Alice, Greenfoot, and Scratch – A Discussion. *ACM Transactions on Computing Education*, 10(4), 2010.

[26] K. Zimmerman and C. R. Rupakheti. An Automated Framework for Recommending Program Elements to Novices. In *Proc. of the 30th Int. Conf. on Automated Software Engineering*, 2015.