Use expert knowledge instead of data – generating hints for Hour of Code assignments –

Milo Buwalda
Utrecht University
The Netherlands
m.a.buwalda@students.uu.nl
ICA-5571839

Johan Jeuring
Utrecht University,
Open University Netherlands
The Netherlands
j.t.jeuring@uu.nl

Nico Naus
Utrecht University
The Netherlands
n.naus@uu.nl

ABSTRACT

Within the field of on-line tutoring systems for learning programming, such as Code.org's Hour of code, there is a trend to use previous student data to give hints. This paper shows that it is better to use expert knowledge to provide hints in environments such as Code.org's Hour of code. We present a heuristic-based approach to generating next-step hints. We use pattern matching algorithms to identify heuristics and apply each identified heuristic to an input program. We generate a next-step hint by selecting the highest scoring heuristic using a scoring function. We compare our results with results of a previous experiment on assignments 4 and 18 of the Hour of code. For assignment 4 we show that a heuristics-based approach to providing hints gives results that are impossible to further improve. For assignment 18, we show that our heuristic-based approach outperforms all models of previous experiments on the Hour of code. These basic heuristics are sufficient to efficiently mimic experts' next-step hints.

ACM Classification Keywords

Applied Computing: Interactive learning environments

Author Keywords

Hints, Student data, Expert knowledge, Learning programming, Interactive learning environments

INTRODUCTION

It is well known that feedback is important for learning [16]. With the rise of online courses such as Massive Open Online Courses (MOOCs), the need for better feedback for learners arises as well. One such online course is the Hour of code by Code.org. The Hour of code introduces computer science to millions of novice learners by providing an hour of learning programming. In the Hour of code assignments, direct feedback aids the learners. However, online courses generally use summative feedback in the form of a quiz that learners take during or at the end of a course. The reason is that the number of learners is far too large in many cases to personally give feedback to. One solution to providing feedback to large amounts of learners is through peer review [4]. This aids a learner somewhat in learning, but in some cases it is far more desirable to provide immediate formative feedback, for instance when learning difficult tasks (e.g. in the domain of math, logic and programming) [16]. One way of providing immediate feedback is through use of hints. Providing immediate hints is the main focus in this paper. In particular, automatically generating hints for novice learners who learn how to program using 2D grid based game assignments, such as the Code.org's Hour of code assignments. There are several approaches to automatically generating hints for learners in programming assignments. Hints can be automatically generated using data-driven techniques. An alternative approach to data-driven hint generation is expert model based hint generation. The two differ in that a data-driven approach relies on historical student data to generate hints and at the same time eliminates the use of expert knowledge.

In this paper we demonstrate a heuristic based approach to automatically generating next step hints, particularly for two Code.org's Hour of code exercises. We generate hints in form of code blocks, through a set of heuristics. We base this set of heuristic on patterns we find in expert hints obtained from Piech et al. [10]. We use expansion techniques from Naus et al. to make sure our model gives hints that lead to the solution program for each possible input program [9]. We evaluate the results of our approach with data obtained from Piech et al. [10]. We believe that our approach is a simple heuristic based hint generation tool that can be used as a stand alone tool or alongside other hint generation tools, including data-driven based tools.

We do not believe that it is necessarily a good idea to use learners' data to give hints of good quality. Price et al. for instance, argue that data-driven hint generation techniques are inherently uncertain and that the quality of the hints varies [13]. This does not necessarily mean, however, that heuristic based hint generation does produce good quality hints. Price et al. mention that hint quality in expert based hint generation can be negatively affected by the expert's "blindspot". It is difficult to compare the quality of hints generated by data-driven models and expert based models. In a later study, Price et al. provide a method for comparing the quality of the hints generated by data-driven models and expert based hint generation models

L@S'18, June 2018, London, UK

ISBN ???...$15.00

DOI: http://dx.doi.org/???

[14]. In their paper, they show it is possible for a data-driven hint generation model to outperform expert based hint generation models in cases where an expert only provides one solution to a problem. However, expert based hint generation models outperform data-driven hint generation in all cases when an expert gives multiple solutions to a problem. This indicates that for simple programming assignments such as in the Hour of code, hint generation can be done equally well, or better using expert based code generation techniques.

The focus in this paper is on how to generate hints, not on how to display obtained hints to a learner.

Related work

In this section, we discuss both data-driven hint generation and expert based hint generation separately.

Data-driven hint generation

One of the earlier data driven tutors that solely rely on learner data to generate hints is the Hint Factory authored by Barnes et al. [2]. They invented a new technique using Markov decision processes to generate one large graph consisting of all the learners' input data when solving logic proofs. Each learner's proof attempt is a graph and each node in the graph represents a step in an attempted proof. All nodes in the graph are weighted and similar nodes joined together form the complete graph. After the Markov decision process generates a complete graph, a learner is guided towards an optimal solution by following the optimal weighted edges. One obvious downside of this approach is the blind spot for unobserved nodes (i.e. unseen learner's proof attempts). These nodes are not contained in the graph and therefore cannot be traversed. As much as 20% of learner input is unobserved. Since this research has been conducted, a number of articles have extended upon the data-driven tutoring concept. Of particular interest is the work done by Rivers et al. [15]. They present a closely related study but they manage to fill the missing hints gap by constructing non-existent paths using an initial expert reference solution and a test method that automatically scores code, in addition to using historical student data. The solution space starts with an initial reference after which historical student data is used to add transitions to the solution space. This ensures that there is always a next-step hint and that hints get better over time. Each time learner data is added, the data is compared using a distance metric to the existing solution space consisting of partial programs, where partial includes incomplete and faulty programs. All possible partial programs together form the total solution space. When a newly added partial program meets certain criteria, the partial program is connected to the closest existing program(s) and a directed edge is added to the solution space. Searching for the closest correct state in the solution space guides a learner along the directed edges to a correct solution state. The Hour of code uses visual programming blocks based on the online Snap programming environment, which in turn is build upon Scratch. Snap however, does not provide any next step feedback by itself. Price et al. further build upon the Snap programming environment with iSnap. iSnap integrates a data-driven hint generation system to enable learners with next step feedback [12]. Price et al. generate hints through the Contextual Tree Decomposition (CTD) algorithm [11]. CTD compares abstracted learner code through AST comparison, but instead of comparing the entire AST, CTD only takes subtrees into consideration. This increases the probability of a match between subtrees and therefore the ability to give hints to learners. This tool builds upon the Hint Factory as well.

We look at the same research questions as Piech et al. in the article on autonomous data driven hint generation [10]. They compare various path tracing algorithms that all use the same learners' input data but differ in the method of selecting the best next-step, which is a program one step closer to the solution program. Piech et al. apply multiple solution space generation algorithms to generate hints for two Hour of code assignments using over one million partial programs per assignment. They manage to obtain accuracies of 95.9% for the first assignment and 84.6% for the second when compared to a gold standard set generated by experts.

Generating data-driven hints presents some problems. Generating hints based on learner data requires a minimum amount of data that is obtained from learners or an expert. For purely data-driven hint generation tools a minimal amount of historical student data is required. The absence of this minimal required amount of information to generate hints at startup is known as the cold start problem. Looking at the case of Piech et al. the best hint generation algorithm would need two thousand learners to obtain the acquired accuracy. But when there are only approximately two hundred fifty learners the average achieved accuracy drops to around 88 percent for the first assignment and 80 percent for the second assignment. When handling small classes with 27 learners, which is the average secondary school class size,[1] accuracies drop to approximately 77 and 67 percent, respectively. In defense of data-driven tutoring, Rivers et al. reduced the cold start problem by using one single reference solution and a test method [15]. Chow et al. generate hints to 90% of the learners using data from 10 successful learners [3].

Expert model based hint generation

An alternative approach to generating hints using historical student data is generating hints using expert knowledge or heuristics. Many approaches exist today that provide learners with automated formative feedback in learning programming. Keuning et al. have reviewed numerous tools (102 papers on 69 tools) that can provide feedback for programming in an intelligent tutoring environment [6]. One interesting finding is that most tools that handle class 3 assignments, as defined by Le and Pinkwart [7], do not support next-step hints or knowledge on how to proceed feedback. Le and Pinkwart divide programming assignments in three classes: Class 1 assignments consist of a single solution, class 2 assignments have different implementation variants and class 3 assignments have different solution strategies. In case of the more restricted class 2 programming assignments, next-step hints or knowledge on how to proceed feedback is often given. However, many of

[1] https://nces.ed.gov/programs/digest/d14/tables/dt14_209.30.asp?current=yes

those tools are restricted in the support for alternative solution strategies.

In a more recent study, Naus et al. [9] show a generic framework that allows rule-based problem solving processes to provide hints. Naus et al. present several methods to generate feedback from a set of rules. Based on specified rules, the framework generates all possible solutions, which are represented as a tree. By traversing the tree, hints can be offered step by step at any given time, particularly when a learner deviates from the tree. Naus et al. provide various approaches for finding shortest paths to the solution.

One of the earliest expert based tutors is LISP tutor authored by Anderson et al. [1] in 1985. The LISP tutor, called GREATERP, is a programming environment for LISP. The tool is designed to provide immediate feedback to learners and it monitors a learner if and when she actually needs feedback as opposed to just giving feedback when desired. The technique used by the LISP tutor is called model-tracing. Model-tracing means that the tutor works out solutions for problems a learner currently faces. The tutor keeps track of an optimal path at each point when a learner is solving an assignment. Furthermore, the tutor keeps track of what rules the learner used and whether giving feedback is desirable for optimal learning.

Another well investigated tool is the ELM-ART education system, which is based on the LISP ITS. The difference with the LISP tutor is that ELM-ART is web-based[19]. ELM-ART provides a learner with an electronic textbook as the outer loop, as explained by Vanlehn et al. [17]. For generating hints, ELM-ART builds on the ELM part of the system, which gives example-based problem-solving support to a learner. ELM-ART generates hints using: a task description containing programming concepts and rules for problem solving, domain knowledge incorporating known problem solving sequences and a learner model that gradually adds learner specific information to direct what personalized hint better fits the learner [18].

One major disadvantage of manually constructed domain models that use more advanced techniques is the added complications for adjusting the tool and adding new assignments [6]. Especially large scale ITSs heavily rely on expert knowledge and are therefore costly in terms of development costs and time [5].

The Hour of code

The Hour of code on Code.org's Code Studio[2] introduces computer science to millions of novice learners by providing an hour of learning programming. It introduces basic programming concepts to a learner by means of two different kinds of code blocks: basic movement blocks and control flow statement blocks. Using these code blocks, a learner needs to direct a character through a maze.

The Hour of code environment is similar to Scratch [8] and Snap [12]. The interface consists of a game environment and a visual coding environment. The game environment has a 2D grid with game elements such as a playable character that is driven by the code blocks, walkable cells and various different impassable obstacles (e.g. walls, exploding boxes). The visual coding environment consists of a toolbox window and a workspace window. The toolbox window shows various draggable code blocks that a learner can use to build a program by dragging them into the workspace and activating them by chaining new blocks to the start event block. A program is any sequence of activated code blocks in the workspace. A learner is allowed to place any code block in the workspace, in any order. When a learner presses the "Run" button, the game environment runs the code that is activated in the workspace.

In assignment 4, a learner is only provided with simple movement and rotation blocks. In assignment 18, control flow blocks such as repeat-until and if-then-else statements are additionally available. Solutions to these assignments can be found in Figures 1 and 2, respectively.

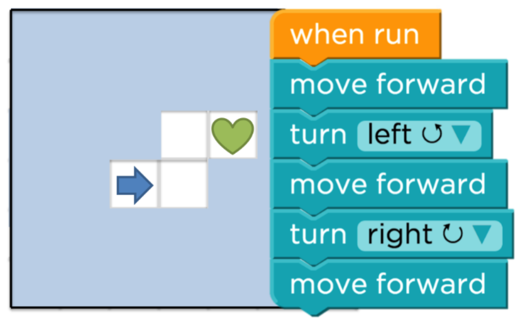

Figure 1: The solution to assignment 4.

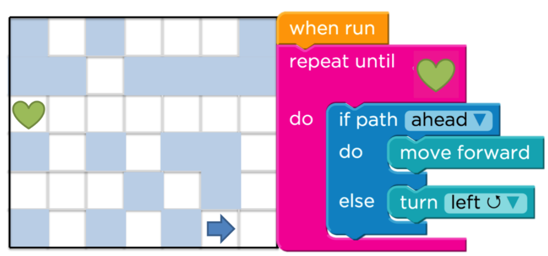

Figure 2: The solution to assignment 18.

Feedback in the Hour of code

A learner can get a small set of hints in the Hour of code environment: a notification that a learner misses a code block, a notification that a learner must fill a control flow block and mentioning that a learner "is not quite there yet." These basic hints however, do not provide a learner with a possible next best step. This leads to the central theme of this paper:

> *How can we provide novice programming learners with step-by-step hints on how to reach the assignment's goal from each possible program?*

Automatically generated hints are particularly useful in situations where direct contact with teachers or experts is not

[2] http://www.code.org/

available, which is often the case when learners are working with on-line assignments as in the Hour of code.

This paper shows how to use a heuristic approach to automatically generate hints for novice learners who learn how to program using 2D grid based game assignments, such as the Hour of code assignments. We evaluate our approach with the same gold standard data set as Piech et al. and obtain an accuracy of 99.1% for assignment 4 and an accuracy of 89.2% for assignment 18.

Piech et al. in their paper, show an accuracy for the static analysis of assignment 4, but do not show an accuracy for the static analysis of assignment 18. They reason that there is a lot of nuance that goes into deciding what a student should do. They furthermore state that the experts seem to suggest hints based on some kind of rules. We can confirm both these statements. Our heuristic based model is a set of rules based on the patterns of the experts. And based on our accuracy, we can indeed say that the experts use rules for correcting errors. This approach of working backwards from gold standard data to a set of heuristics, is less than ideal to find the best rules. The optimal solution for assignment 4 is the shortest path from the assignment's starting location to the goal location. A hint is a suggestion to use an edit to change a program to get closer to the optimal solution. An edit action is an insertion, a deletion or a substitution. For example, when the program of a learner is a prefix of the optimal solution, we can give as hint the next programming block in the optimal solution. When a learner's program results in the character running into a wall, we direct the learner back to the optimal solution by suggesting a single deletion of her last erroneous code block. When a learner almost develops the optimal solution except, for example, one code block, we can offer a suggestion to substitute the erroneous block for the right block.

In assignment 4, a learner can use three type of blocks: move forward, turn right 90° and turn left 90°. Hints consist of one of these blocks with some accompanying text: "add `[Forward]` after position x", "delete `[TurnRight]` from the end", or "substitute `[Forward]` at position x with `[TurnLeft]`". For example, when a learner hands in the following program for assignment 4: `[Forward,TurnLeft,Forward]` we give `add [TurnRight] after position 3` as a hint (insertion-hint). This hint can be translated to match the partial solution Piech et al. use in their paper: `[Forward,TurnLeft,Forward,TurnRight]`.

In assignment 18, a learner is introduced to two additional types of code blocks. Both are control flow statements: repeat-until and if-then-else statement. The repeat-until code block contains an unchangeable condition that checks if the goal is reached. The condition of the if-then-else statement can check one of three things: is there a path ahead, is there a path on the left and is there a path on the right. These three conditions are represented as: `PathAhead`, `PathLeft` and `PathRight` respectively.

The optimal solution for assignment 18 is the shortest program (not path) that navigates the player on a path to the goal location. To pass the hour of code assignment, a learner must use both control flow statements to complete this assignment.

THE GOLD STANDARD

The gold standard data set has been produced by experts participating in the experiment from Piech et al. The data consists of the experts' suggested next step hints for assignments 4 and 18 of the hour of code. The hint is also a program. The hints differ from a learner's program by one single edit, where an edit is either an insertion, a deletion or a substitution of a code block. In this section we describe our analysis of the data for assignments 4 and 18 separately, because the gold standard data for assignment 4 and 18 are too different from each other to create a generic model. For each assignment we analyse the gold standard data by looking for patterns we can distinguish in the expert hints. We extract patterns by classifying hints with similar effects into separate heuristics. We use these heuristics to generate hints that are similar to the experts' hints.

Assignment 4

In this section we describe our analysis of the data for assignment 4.

Hint analysis

We extract the relevant gold standard data from the "groundtruth.txt" file made available by Piech et al. The gold standard data for assignment 4 is a comma separated file where each line contains a unique student program id and the id of the program that the experts suggest. The gold standard data consists of id's of the 225 most occurring programs. Each id corresponds to a similarly named file that is stored separately. Each file contains a program, which is represented as an Abstract Syntax Tree (AST). We translate all the ASTs to sequences of Forward, TurnLeft and TurnRight motions, which we represent as a string (e.g. "flr").

We use the following notation for describing an edit between two programs. First, the kind of edit is denoted by a symbol: Insertion is +, Deletion is - and Substitution is =. After the kind of edit, we specify the position of the letter in the sequence, where we start counting at 0. In the case of an insertion we write the new character in capitals before the position. In the case of a substitution, we write the substituted letter after the position. For instance, +F2 is an Insertion turning "flr" into "flfr". -1 is a Deletion turning "flr" into "fr", and =1R is a Substitution turning "flr" into "frr".

We make a number of observations based on the gold standard data. First, the experts restrict all hints to one single edit inserting, deleting or substituting a code block. The experts restrict substitutions to rotations only. As a consequence, the number of hints to reach an optimal solution may be larger than necessary. For example, the single edit distance between the input program "fffrf" and the solution program "flfrf" is one (=1L), but the experts suggest two paths to the solution program. The first approach consists of two edits: -1 ("ffrf") and +L1 ("flfrf"). The second approach consists of five edits: -1 ("ffrf") -2 ("frf") =1L ("flf") +R3 ("flfr") and +F4 ("flfrf"). Both examples occur in the gold standard and show that not all experts agree on what hint should be given in what situation.

A second observation is that there are sequences of rotations that can either be left out or replaced by a single rotation. In "rl" and "lr", the second rotation undoes the effect of the first rotation. Such a sequence of code blocks can be removed. Two sequences that are replaceable are "rrr" and "lll", where "rrr" is replaceable with "l" and "lll" with "r". When the experts handle input programs containing one of these sequences, they sometimes ignore these sequences and handle them at the last possible moment. Another observation is that in the case of "frrflf" the experts suggest to apply +R3 to obtain "frrrflf", which is the same as "flflf". The hint containing "rrr" instead of "l" is closer to the solution path in terms of single edit distance than applying =1L to obtain "flrflf" or -1 to obtain "frflf".

Heuristics for hints

We model the patterns we find in the experts' hints into separate hint heuristics.

Experts suggest to delete erroneous forward movements and never suggest to substitute forward movements by rotations. When an input program contains multiple erroneous rotations, the experts sometimes suggest to delete a rotation, and sometimes to substitute an erroneous rotation by another block. We create a set of heuristics, where each heuristic is a procedure that produces a hint when it is applied to an input program. Each heuristic brings a student a step closer to the optimal solution. For instance, for the input program "ff" we can give a hint that deletes the erroneous forward movement through -1 and obtain "f" or give a hint that inserts a left rotation +L1 to obtain the program "flf". Table 1 shows the general heuristics we could distinguish, illustrated with example programs containing substrings, marked in red, on which the heuristic fires. Sometimes multiple heuristics can be applied to an input program.

The heuristic `OnTrack` is identified when a player is on the right path towards the solution. The heuristics `DelErrStart` and `InsStart` are both identified when a student program starts with the wrong code block. For assignment 4 this is the case when a program starts with "l" or "r". If a program starts with a rotation and contains at least one forward movement, `DelErrStart` is identified. `InsStart` is identified when there are only rotations in a student program. If there are any erroneous forward movements, the heuristic `DelErrForward` is identified. For assignment 4, any sequence of multiple forward movements is identified as incorrect because the optimal solution does not contain multiple successive forward blocks. The heuristic `DelErrRot` is identified when a program contains an erroneous rotation that needs to be deleted. For example, in the input program "fllf" the second rotation needs to be removed. `SubstErrRot` is identified when there is an erroneous rotation that can be substituted by a correct rotation. For example, in the input program "frf" the student made a right turn instead of a left. `SingleEdit` is identified when the program gets too lengthy or when the program is a single edit distance away from the solution program.

Experts prefer certain heuristics over others when multiple heuristics can be applied. We need a way to determine which heuristic to choose when there are multiple possible heuristics.

Table 1: Heuristics

Heuristic	**Ratio**	**Value**	**Example programs**
`OnTrack`	5/5	1.00	"flf", "flfr"
`InsStart`	8/8	1.00	"lrl", "rr"
`SingleEdit`	45/49	0.92	"flrf", "ffrfrf"
`DelErrStart`	35/50	0.70	"rf", "lf"
`DelErrForward`	55/110	0.50	"flff", "ffrfrf
`DelErrRot`	62/124	0.50	"fllf", "fflr"
`SubstErrRot`	15/219	0.07	"frf", "fflr"

We count how often experts apply each heuristic on the 225 input programs and how often we identify each heuristic in the input programs based on the conditions mentioned in the previous section. We divide these numbers to obtain a value that determines how likely it is that the experts apply a heuristic to an input program. Table 1 shows the resulting ratios.

For the input program "ffrf", our algorithm recognizes the erroneous substring "ff" and identifies the heuristic `DelErrForward`, which can be addressed by deleting one "f" by -0. Additionally, the heuristic `SingleEdit` is identified, since with +L1 we obtain the optimal solution "flfrf". In this case, suggesting a hint to perform a single edit to obtain the solution is better than suggesting to delete the second erroneous forward step. Here our hint corresponds to the expert hint. However, the experts do not necessarily choose a heuristic in the order of Table 1. For example consider the input program "fflr". We identify the following heuristics: reduce the number of erroneous forward motions "ff": -1 ("flr") (`DelErrForward`), delete the wrong turn "r": -3 ("ffl") (`DelErrRot`), or substitute an erroneous rotation "l": =R2 ("ffrr") (`SubstErrRot`). This list is in order of preference based on the values taken from Table 1. However, `DelErrForward` and `DelErrRot` have the same values. In this case, experts suggest "flr" as a hint, indicating that removing double forward movements has a higher priority than reducing erroneous rotations or suggesting an insertion. We introduce another metric, called a dynamic score, to differentiate between the heuristics.

The dynamic score is based on three measures. The first measure calculates the length of the initial segment of blocks of a program that also appears at the start of the solution. For example, the program "fr" has a score of 1 since the solution starts with "fl". This scoring method favors programs that start correctly. We ignore sequences of rotations that cancel each other out when calculating this score, so "lrflf" becomes "flf" and has a score of 3. The second measure calculates how much longer a program is than the solution. By subtracting this from the first, we favor deletion over substitution or insertion for long programs. The third measure calculates the location of the edit: the closer to the beginning of the program the better. The dynamic score is defined by: $dynScore(i,h) = startSegmentLength(h) - programLength(h) - errorLocation(i,h)$, where i is the input program and h is the hint obtained by applying the heuristic to the input program. The higher the dynamic score, the better.

We use the order of the heuristic in Table 1, to define a static score. We assign 7 to `OnTrack` and one less to each subsequent

```
repeat
└─ if PathAhead
   ├─ Forward
   └─ TurnLeft
```

Figure 3: Tree representation of the solution seen in Figure 2

heuristic, ending with 1 for `SubstErrRot`. We define the total score of a heuristic by: $totalScore(h,i) = staticScore(h) + dynamicScore(h,i)$, where h is the heuristic and i the input program. We select the heuristic with the lowest score. We now measure how well our heuristics-based selection performs against the gold standard. We measure the accuracy of our selection by determining the similarity between our hints and the gold standard. We optimize the static scores by varying the static score values for each heuristic both upward and downward to determine the range in which the static score values maximize the accuracy. It turns out that we only need to adjust `DelErrForward` (to 2.6), `DelErrRot` (to 1.1), and `SubstErrRot` (to 0.0) to obtain the highest accuracy.

Assignment 18

In many respects the analysis of assignment 18 is the same as for assignment 4. In this section we describe our analysis of the data for assignment 18 emphasizing the differences with assignment 4.

Hint analysis

The gold standard data for assignment 18 is similar in structure to assignment 4. All programs are represented using the same AST structure as assignment 4 with the addition of the control flow statements: repeat-until and if-then-else. Similar to assignment 4, the gold standard consists of unique student program id's and the id's of corresponding programs that the experts suggest. The gold standard data for assignment 18 consists of 299 expert hints. There are two differences with assignment 4. Repeat-until and if-then-else statements require that there are code blocks inside all their slots in order to work. In some cases the experts suggest that a student deletes a code block, which possibly results in an empty slot in a control flow statement. Because of these requirements, in some cases the experts suggest a program that fails to compile on the Hour of code website. The second difference is that in 18 out of 299 cases the experts suggest a program that is more than one edit distance removed from a student's input program. We currently do not handle these programs.

Instead of representing a program as a string, we represent a program as a tree. We shorten the repeat-until and if-then-else statement for brevity. We sometimes include a statement list node for better readability. The tree shown in Figure 3 represents the same program as shown in Figure 2.

In assignment 4 experts limit substitutions to rotations. In assignment 18 this is still mostly true except for two cases. In one case experts substitute a rotation for a forward movement and in another case a forward movement for a rotation. Similar to assignment 4, different experts sometimes suggest different hints for the same input program.

A general observation is that based on the gold standard data alone, we did not find a sequence of steps from an empty program to a solution program. One possible reason for this missing sequence is that in the Code.org Hour of code, a student must include both control flow statements to complete the assignment. Out of the 299 cases in the gold standard, in only 15 cases the experts suggest to insert a code block as a hint. Out of these 15 cases, the experts suggest to insert repeat-until 14 times and Forward 1 time. The experts never suggest to insert an if-then-else statement nor do they ever delete an if-then-else statement. We therefore do not know the experts' preferred way of dealing with erroneous if-then-else statements. For instance, some student programs contain multiple if-then-else statement where at least one needs to be removed.

The addition of the control flow statements increases the complexity of generating hints for assignment 18. In assignment 4, every program is a list of movement and rotation statements. The addition of an if-then-else statement changes the basic structure of a program from a linear statement list to a tree. Instead of analysing the experts' hints for linear programs, assignment 18 deals with trees.

For all heuristics, the experts take the location of a pattern in the AST into account. There are general rules emerging from the gold standard data. Similar to assignment 4, the experts prefer hints that keep the student on the maze's path. As a result, the experts most often correct errors located deeper in a program. The experts seem to have specific preferences for dealing with if-then-else statements and statement lists, which we will address later.

```
Forward        repeat          if PathAhead
               └─Forward       ├─Forward
                               └─TurnLeft
```

Figure 4: Short partial correct programs

The experts might suggest two additional types of (sub)programs: partially correct (sub)programs and informative (sub)programs. These suggestions may be a program or subprogram located anywhere inside a program. Intuitively, partially correct programs are programs that are on route to the solution. Formally, a partial correct program is a partially ordered set of code blocks taken from the shortest solution as shown in Figure 4, or a prefix of one of the longer solutions that leads to the goal. We show two examples of longer partial correct programs in Figure 5. An informative (sub)program has properties that the experts consider important enough to suggest as a hint even though this (sub)program does not lead to a correct solution. For this assignment, the only informative (sub)program we find in the gold standard is shown in Figure 6 on the left. This statement does not lead to a solution, but the structure has similar properties to the correct variant shown on the right in the same figure. We suppose that the similarity between these two programs is likely the reason some experts suggest this informative (sub)program. An example case is shown in Figure 7, where the experts suggest the program on the right as a hint. An alternative suggestion would be to substitute *PathLeft* into *PathAhead*. A hint with an

```
list
├Forward
├TurnLeft
├Forward
├Forward
├Forward
├TurnLeft
├Forward
├Forward
├Forward
├Forward
├Forward
└Forward
```

```
list
├Forward
├TurnLeft
├Forward
├Forward
├Forward
├TurnLeft
└repeat
  └Forward
```

Figure 5: Long partial correct programs

informative subprogram is only given if it is one single edit distance removed from a student's program. The suggested program seen in Figure 7 brings the character around the first two corners but does not reach the goal.

```
if PathLeft
├TurnLeft
└Forward
```

```
if PathAhead
├Forward
└TurnLeft
```

Figure 6: Informative (sub)program and the corresponding correct variant

Heuristics for hints

In this section we show the heuristics for assignment 18, which are similar to the heuristics for assignment 4. The heuristics for assignment 4 are generally applicable to programs that contain control flow statements. Two heuristics for assignment 4 are too specific for assignment 18. The heuristics `DelErrForward` and `DelErrRot` are replaced by `DelErr`. In assignment 18 the heuristic `OnTrack` does not appear in the gold standard. This means that no case exists where a student program is a prefix of the pre-topologically sorted solution program.

We identify the following heuristics for assignment 18. The heuristic `SingleEdit` is identified when: a partially correct (sub)program or informative (sub)program is one edit distance removed from the current student program. The heuristic `DelErr` is identified when any error occurs. For assignment 18, any movement or rotation outside a control flow statement, is identified as erroneous. The heuristic `InsStart` is identified when a program is the empty program and if a student starts with an incorrect code block that is a subprogram of the optimal solution. Specifically for assignment 18, this heuristic is only identified if it is the empty program and when the (sub)program starts with the second code block of the solution

```
repeat
└list
  ├Forward
  └if PathLeft
    ├TurnRight
    └Forward
```

```
repeat
└list
  ├Forward
  └if PathLeft
    ├TurnLeft
    └Forward
```

Figure 7: Example hint containing an informative subprogram.

program, which is a conditional statement. The heuristic `SubstErr` is identified when an erroneous movement or rotation code block occurs or when the condition in an if-then-else code block is wrong. Specifically for assignment 18, this heuristic is only applied on the condition in an if-then-else code block and rotations inside the then slot and else slot. The reason is that rotations and (substitutable) conditions only occur in the if-then-else statement in the optimal solution of this particular assignment. The experts never suggest to substitute repeat-until and if-then-else code blocks. The heuristic `DelErrStart` is identified when the program starts incorrect. In assignment 18, this heuristic is only applied once on the program that only contains one Forward code block. We show examples of the heuristics in Figure 8.

```
Forward
```

(1) `DelErrStart`

```
repeat
└list
  ├Forward
  └if PathLeft
    ├TurnRight
    └Forward
```

(2) `SingleEdit`

```
list
├Forward
├TurnRight
└TurnLeft
```

```
repeat
└Forward
```

(3) `DelErr`

```
if PathAhead
├TurnRight
└TurnLeft
```

(4) `InsStart`

```
if PathLeft
├Forward
└TurnRight
```

(5) `SubstErr`

Figure 8: Heuristic examples

There are cases where the heuristics `DelErr` and `SubstErr` can be applied in multiple places in a program. For both heuristics, the experts' general rule is to guide the character as long as possible on the maze's path. There are other general rules as well, which determine where to apply the heuristic.

For the heuristic `DelErr`, the experts suggest to delete erroneous movements starting at the end. If a deletion exists that keeps the program on path, the program containing that deletion is suggested. The best deletion orients the character in the right direction on a path towards the goal. For example, the first program in example 3 in Figure 8 of the heuristic `DelErr` has three possible places where we can apply this heuristic. However, only deletion of `TurnRight` results in a program that stays on the path towards the goal and additionally, correctly orients the character. For the heuristic `SubstErr` the same general rule applies but in reversed order. The experts generally suggest to correct errors in the boolean condition first, then-block second and else-block last. This rule has exceptions, that are determined by the context in which this subprogram is in. Figure 8, shows two locations where the heuristic `SubstErr` can be applied. In this case the experts suggest to substitute *`PathLeft`* into *`PathAhead`*.

For assignment 18, we still need a method to choose the heuristic that matches the gold standard. For assignment 4, we choose the best suitable heuristic by calculating a static and dynamic score. These scores do not work for assignment 18 unfortunately. For instance in assignment 4, the experts generally prefer to correct erroneous start code block, whereas for assignment 18 the experts prefer to substitute erroneous rotations first.

For assignment 18, to fit our model more closely to the gold standard we order the heuristics. We show the order of precedence of the heuristics in Table 2. When for instance, we identify multiple heuristics, we always select the highest ranking heuristic. If for example, the heuristic `OnTrack` is identified, it does not matter which other heuristic is identified, `OnTrack` is always selected. We also define more specific ordering prerequisites for the heuristics: `InsStart`, `SubstErr` and `DelErr`. These can be found in Table 3. It is essentially an expansion of the heuristic set, with information on surrounding code blocks, the location of the pattern inside the program and the preferred order in which the experts suggest hints. The location and the preferred order are somewhat equivalent to the score in assignment 4.

Table 2: Heuristics placed in rank order

Rank	Heuristics
1	`SingleEdit`
2	`InsStart`
3	`SubstErr`
4	`DelErrStart`
5	`DelErr`

This specific rule set is sufficient to determine which heuristic the experts apply in most cases. We show the rule set in Table 3. Some of the experts' hints do not follow these rules for reasons we cannot always clearly identify. In the upcoming Results section we present the contradicting cases with corresponding examples.

RESULTS

In this section we separately discuss the results for assignment 4 and 18.

Results for assignment 4

For 223 out of 225 cases we give the same hint as the experts. For each of the 2 remaining cases, we can identify conceptually contradicting expert hints, for example, suggesting an edit

Table 3: Context dependent rule set

Heuristic	*Ordering rules*	*Example*
`SingleEdit`	-	
`InsStart`	• The if-then else code block must have the correct condition *`PathAhead`*.	

	• The heuristic `DelErr` takes precedence when applying `DelErr` generates a correct starting program.	`list`
		`└ [Forward]`
		`└ repeat`
		`    └ if` *`PathAhead`*
		`        └ ...`

	• The correct starting code block may not be present *after* the current if-then-else code block. Even if this would lead the student to reach the goal.	`list`
		`└ if` *`PathAhead`*
		`    └ Forward`
		`    └ TurnLeft`
		`└ [repeat]`
		`    └ Forward`

	• There may not be more than one movement or rotation after the current if-then-else code block.	`list`
		`└ if` *`PathAhead`*
		`    └ Forward`
		`    └ TurnLeft`
		`└ Forward`
		`└ [TurnLeft]`

	• The current if-then-else code block may not contain multiple movement or rotation code blocks in the then statement and else statement.	`if` *`PathAhead`*
		`└ [list]`
		`    └ Forward`
		`    └ TurnLeft`
		`└ TurnLeft`

`SubstErr`	• If the heuristic `DelErr` is identified, `DelErr` is selected if it leads the student faster to the optimal solution in terms of edit distance. If the distance to the solution goal is equal to `DelErr`, `SubstErr` is selected.	
`DelErrStart`	-	
`DelErr`	• The previous conditions are not met.	

either at the beginning or at the end of an erroneous program. All the hints we suggest correspond to the expert hints, except for `DelErrRot`, for which we are correct 58 out of 60 times.

Table 4 shows the hints that differ from the gold standard for `DelErrRot`. In the gold standard data, successive rotations that cancel each other out are handled differently than non-cancelling erroneous successive rotations. For the input "flrl" the experts suggest "flrlf", because the neutralizing effect of "lr" turns the input into "fl" and the hint to "flf".

Results for assignment 18

Out of the 299 cases in the gold standard, 18 cases contain hints with an edit distance larger than 1. For the remaining 282 cases, there are 23 student programs where the experts do not agree and present two different hints for the same input. For all these programs we generate one of the two possible hints from the experts. We exclude these hints from the final accuracy calculation, resulting in 259 cases. For 231 out of 259 we generate the same hint as the experts. For the remaining 28 cases we are able to identify contradicting cases, where experts use a different heuristic to give a hint. Table 5 shows all the inconsistencies for the heuristic `DelErr`. Table 6 shows the single inconsistent case for the heuristic `InsStart`. Table 7 shows the 5 remaining cases where we show inconsistencies for the heuristic `SubstErr`. These three tables are located in the appendix at the end of this paper, along with self explanatory comments per case. The gold standard data for assignment 18 is more difficult to analyse than assignment 4 and contradictory cases might have reasonable explanations due to underlying rules that we fail to see.

CONCLUSIONS

We have developed a heuristics-based approach to giving hints for the Hour of code assignments. Analysing the gold data for assignment 4 shows that our approach generates hints with a high precision and is flexible enough to handle different forms of input. We generate the same hint as the gold standard data 223 out of 225 times, leading to an accuracy of 99.1%. This improves upon the best algorithm from the paper of Piech et al., which has an accuracy of 95.9%. Because the expert hints are internally inconsistent, a higher accuracy than ours on this dataset is impossible.

For assignment 18, we successfully generate the same hints as the gold standard data 231 out of 259 times. This leads to an accuracy of 89.2%. The best algorithm from the paper of Piech et al., gives an accuracy of 84.6% for assignment 18. This shows that our heuristic model improves upon that algorithm as well.

We have demonstrated that using basic heuristics we can efficiently mimic experts' next-step hints. Our results shows that you do not need a large quantity of student data to obtain a high accuracy compared with the gold standard. Using expert knowledge, and deriving heuristics from this knowledge, leads to better feedback than using student data. Of course we are fitting our method to expert data, but the resulting heuristics are general, explainable, heuristics, which can also be used for the other Code.org's Hour of code assignments. A disadvantage of our approach compared to an approach based on previous student data is that you need to develop the heuristic for each domain on which you want to provide feedback. However, this extra work comes with the significant advantage that the heuristics can also be used to explain hints. The required investment is negligible compared to the amount of time users spend on the assignments.

DISCUSSION

In some cases, it is impossible to fully model the patterns of the experts. The gold standard does not contain enough information to create a model that will generate next step hints for all possible input programs. We wish to extend our model so that it will work in a real-world application. In the following section we elaborate on possible further improvements to achieve this.

Generalization

As one might expect, we believe that the greatest improvements may come when a set of heuristics is established together with the experts prior to building the gold standard data. As previously mentioned in the gold standard section, the gold standard is not a complete set of hints that can lead a student from start to goal. In this section we introduce further improvements in a generalization of our model.

The gold standard data is insufficient to generate paths starting from any arbitrary program to the solution program. The gold standard does provide a general set of rules from which we can extrapolate heuristics that always lead to the solution program. We always select a hint one single edit closer to the solution. The main focus is then to identify all possible cases.

The heuristics `OnTrack` and `InsStart` are the only insertion heuristics. The gold standard data does not contain the `OnTrack` heuristic. We therefore reintroduce the `OnTrack` heuristic. For the `OnTrack` heuristic, we can either identify all subprograms of the solution program or identify subprograms that are a prefix of the topological sorted (pre-order traversed) solution program. There exists a case where the experts suggest the `Empty` program for the Forward program. Since Forward is a subprogram of the solution program. This implies that the minimal requirement for the identification of the `OnTrack` heuristic is that it has to be a prefix of the pre-order sorted solution program. Figure 9 shows our additions to this heuristic. As previously mentioned, these hints are already given in the assignment on Code.org's Hour of code.

```
repeat
└Empty
```

```
repeat
└if PathAhead
  └Empty
  └Empty
```

```
repeat
└if PathAhead
  └Forward
  └Empty
```

Figure 9: Extended `OnTrack` heuristics

In the gold standard, the experts never suggest to delete an if-then-else statement. Because the optimal solution requires one if-then else statement, this problem only arises when there are multiple if-then-else blocks present in a student's program. Our solution is to extend the heuristic `DelErr` by suggesting to delete the conditional statement that contributes least to reaching the goal.

As a safety measure, we can furthermore identify `SingleEdit` when none of the other heuristics are identified. With this addition, we can make sure that in all cases we generate a hint that leads to a partial solution or the optimal solution program.

With these additional heuristics all substitutions, deletions and insertions are covered. The heuristics `OnTrack` and `InsStart` generate insertion hints, the heuristic `SubstErr` generates substitution hints and `DelErr` generates deletion hints. The heuristic `SingleEdit` can generate all three edit types.

Limitations

Our model is currently limited to the particular domain of Code.org's Hour of code assignments. We can generate hints for all 20 assignments if we have the solution program. Without a solution our model is only able to generate shortest path programs. Our model is able to work with a set of options as the conditions in the repeat-until and if- then-else statements. Our model does not support variables.

Acknowledgements

Foremost, I thank God for giving me perseverance. I would like to thank my supervisor Prof. Johan Jeuring for his essential support, patience and guidance. Without his help this thesis would not have been possible. I would also like to thank Nico Naus for his great assistence.

REFERENCES

1. John R Anderson and Brian J Reiser. 1985. The LISP tutor: it approaches the effectiveness of a human tutor. *BYTE* 10, 4 (1985), 159–175.
2. Tiffany Barnes and John Stamper. 2008. Toward automatic hint generation for logic proof tutoring using historical student data. In *International Conference on Intelligent Tutoring Systems*. Springer, 373–382.
3. Sammi Chow, Kalina Yacef, Irena Koprinska, and James Curran. 2017. Automated Data-Driven Hints for Computer Programming Students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation and Personalization (UMAP '17)*. ACM, New York, NY, USA, 5–10. DOI: `http://dx.doi.org/10.1145/3099023.3099065`
4. Peggy A Ertmer, Jennifer C Richardson, Brian Belland, Denise Camin, Patrick Connolly, Glen Coulthard, Kimfong Lei, and Christopher Mong. 2007. Using peer feedback to enhance the quality of student online postings: An exploratory study. *Journal of Computer-Mediated Communication* 12, 2 (2007), 412–433.
5. Jeremiah T Folsom-Kovarik, Sae Schatz, and Denise Nicholson. 2010. Plan ahead: Pricing ITS learner models. In *Proceedings of the 19th Behavior Representation in Modeling & Simulation (BRIMS) Conference*. 47–54.
6. Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 41–46.
7. Nguyen-thinh Le and Niels Pinkwart. 2014. Towards a classification for programming exercises. In *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*.
8. John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4 (Nov. 2010), 16:1–16:15.
9. Nico Naus and Johan Jeuring. 2017. Building a generic feedback system for rule-based problems. *Technical Report Series* UU-CS-2017-013 (2017).
10. Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 195–204.
11. Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating data-driven hints for open-ended programming. In *Proceedings of the 9th International Conference on Educational Data Mining, International Educational Data Mining Society*. 191–198.
12. Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017a. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 483–488.
13. Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017b. Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior. In *International Conference on Artificial Intelligence in Education*. Springer, 311–322.
14. Thomas W Price, Rui Zhi, Yihuan Dong, Nicholas Lytle, and Tiffany Barnes. 2018. The Impact of Data Quantity and Source on the Quality of Data-Driven Hints for Programming. In *International Conference on Artificial Intelligence in Education*. Springer, 476–490.
15. Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.
16. Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.
17. Kurt Vanlehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3 (2006), 227–265.
18. Gerhard Weber. 1996. Episodic learner modeling. *Cognitive Science* 20, 2 (1996), 195–236.
19. Gerhard Weber and Peter Brusilovsky. 2001. ELM-ART: An adaptive versatile system for Web-based instruction. *International Journal of Artificial Intelligence in Education (IJAIED)* 12 (2001), 351–384.

APPENDIX

RESULT OUTPUT

Table 4: Deviating results in assignment 4 in the heuristic `DelErrRot`

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies*
frlr	frl	flr	We give "flr" as a hint because it is faster to the goal solution. The experts remove the last erroneous rotation.	frlrf -> flrf and flrrl -> flrl
frll	fll	frllf	We ignore "rl" and build on "fl", whereas the experts suggest to delete the first part of the erroneous rotation sequence.	flrl -> flrlf and fllr -> fllrf

Table 5: Deviating results in assignment 18 in the heuristic `DelErr`

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies*	
				Input	*GS hint*

list
└ Forward
└ TurnLeft
└ Forward
└ Forward
└ Forward
└ $\boxed{\texttt{TurnLeft}}$
└ repeat
  └ $\boxed{\texttt{Forward}}$
	list
	└ Forward
	└ TurnLeft
	└ Forward
	└ Forward
	└ Forward
	└ TurnLeft
	└ repeat
	  └ $\boxed{\texttt{Empty}}$
		$\boxed{\texttt{list}}$
		└ Forward
		└ TurnLeft
		└ Forward
		└ Forward
		└ Forward
		└ repeat
		  └ Forward
			The experts suggest to remove the erroneous control flow statement. We remove the erroneous movement.	list
				└ Forward
				└ TurnLeft
				└ Forward
				└ Forward
				└ Forward
				└ $\boxed{\texttt{TurnLeft}}$
				└ repeat
				  └ if *PathAhead*
				    └ Forward
				    └ Forward
					$\boxed{\texttt{list}}$
					└ Forward
					└ TurnLeft
					└ Forward
					└ Forward
					└ Forward
					└ repeat
					  └ if *PathAhead*
					    └ Forward
					    └ Forward

list
└ Forward
└ $\boxed{\texttt{TurnLeft}}$
└ repeat
  └ list
    └ $\boxed{\texttt{Forward}}$
    └ if *PathLeft*
      └ TurnLeft
      └ Forward
	$\boxed{\texttt{list}}$
	└ Forward
	└ repeat
	  └ list
	    └ Forward
	    └ if *PathLeft*
	      └ TurnLeft
	      └ Forward
		list
		└ Forward
		└ TurnLeft
		└ $\boxed{\texttt{repeat}}$
		  └ if *PathLeft*
		    └ TurnLeft
		    └ Forward
			The experts delete the erroneous movement in the first statement list, whereas we delete the error in the deepest statement list.	list
				└ Forward
				└ TurnLeft
				└ repeat
				  └ list
				    └ $\boxed{\texttt{Forward}}$
				    └ if *PathAhead*
				      └ Forward
				      └ TurnLeft
					list
					└ Forward
					└ TurnLeft
					└ $\boxed{\texttt{repeat}}$
					  └ if *PathAhead*
					    └ Forward
					    └ TurnLeft

list
└ $\boxed{\texttt{Forward}}$
└ if *PathAhead*
  └ TurnLeft
  └ Forward
	if *PathAhead*
	└ TurnLeft
	└ Forward
		list
		└ Forward
		└ $\boxed{\texttt{repeat}}$
		  └ if *PathAhead*
		    └ TurnLeft
		    └ Forward
			The experts delete the erroneous movement whereas we insert the correct start.	list
				└ Forward
				└ $\boxed{\texttt{if}}$ *PathAhead*
				  └ TurnLeft
				  └ TurnLeft
					list
					└ Forward
					└ $\boxed{\texttt{repeat}}$
					  └ if *PathAhead*
					    └ TurnLeft
					    └ TurnLeft

list
└ Forward
└ if *PathAhead*
  └ TurnLeft
  └ $\boxed{\texttt{Forward}}$
└ repeat
  └ $\boxed{\texttt{Forward}}$
	list
	└ Forward
	└ if *PathAhead*
	  └ TurnLeft
	  └ $\boxed{\texttt{Empty}}$
	└ repeat
	  └ Forward
		list
		└ Forward
		└ if *PathAhead*
		  └ TurnLeft
		  └ Forward
		└ repeat
		  └ $\boxed{\texttt{Empty}}$
			The experts suggest to remove the conditional statement starting at the else block. We remove the last error.	list
				└ if *PathAhead*
				  └ TurnLeft
				  └ TurnRight
				└ repeat
				  └ $\boxed{\texttt{Forward}}$
					list
					└ if *PathAhead*
					  └ TurnLeft
					  └ TurnRight
					└ repeat
					  └ $\boxed{\texttt{Empty}}$

list
└ $\boxed{\texttt{Forward}}$
└ if $\boxed{\textit{PathLeft}}$
  └ TurnLeft
  └ Forward
└ Forward
└ Forward
└ Forward
└ TurnLeft
└ repeat
  └ Forward
	$\boxed{\texttt{list}}$
	└ if *PathLeft*
	  └ TurnLeft
	  └ Forward
	└ Forward
	└ Forward
	└ Forward
	└ TurnLeft
	└ repeat
	  └ Forward
		list
		└ Forward
		└ if *PathLeft*
		  └ TurnLeft
		  └ Forward
		└ Forward
		└ Forward
		└ Forward
		└ TurnLeft
		└ repeat
		  └ $\boxed{\texttt{Empty}}$
			The experts remove the individual erroneous movement first, we remove the erroneous movement inside the last control flow statement.	repeat
				└ list
				  └ Forward
				  └ if *PathLeft*
				    └ TurnLeft
				    └ Forward
				  └ if $\boxed{\textit{PathRight}}$
				    └ TurnRight
				    └ Forward
					repeat
					└ list
					  └ Forward
					  └ if *PathLeft*
					    └ TurnLeft
					    └ Forward
					  └ if $\boxed{\textit{PathAhead}}$
					    └ TurnRight
					    └ Forward

list
└ $\boxed{\texttt{Forward}}$
└ repeat
  └ list
    └ Forward
    └ $\boxed{\texttt{TurnLeft}}$
	$\boxed{\texttt{repeat}}$
	└ list
	  └ Forward
	  └ TurnLeft
		list
		└ Forward
		└ $\boxed{\texttt{repeat}}$
		  └ Forward
			The experts suggest to remove the first forward movement so that the student makes the corner. We remove the erroneous rotation.	list
				└ $\boxed{\texttt{Forward}}$
				└ repeat
				  └ list
				    └ TurnLeft
				    └ Forward
					list
					└ Forward
					└ repeat
					  └ TurnLeft

Continued on next page

Table 5 – continued from previous page

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies*	
				Input	*GS hint*

list
└Forward
└repeat
 └if *PathAhead*
  └TurnLeft
  └Forward
	list
	└Forward
	└repeat
	 └if *PathAhead*
	  └TurnLeft
	  └Empty
		list
		└Forward
		└repeat
		 └if *PathAhead*
		  └Empty
		  └Forward
			The experts suggest to remove the errors in the conditional statement in last to first order. Whereas we suggest a deletion in the true block (first to last order).	list
				└Forward
				└repeat
				 └if *PathAhead*
				  └TurnLeft
				  └TurnRight
					list
					└Forward
					└repeat
					 └if *PathAhead*
					  └Empty
					  └TurnRight

list
└Forward
└repeat
 └if *PathAhead*
  └TurnLeft
  └TurnLeft
	repeat
	└if *PathAhead*
	 └TurnLeft
	 └TurnLeft
		list
		└Forward
		└repeat
		 └if *PathAhead*
		  └Empty
		  └TurnLeft
			The experts suggest to delete the first erroneous movement to make the student start correct. We suggest a deletion that makes the student turn the corner.	list
				└Forward
				└repeat
				 └if *PathAhead*
				  └TurnRight
				  └TurnLeft
					list
					└Forward
					└repeat
					 └if *PathAhead*
					  └Empty
					  └TurnLeft

if *PathAhead* └TurnRight └Forward	if *PathAhead* └Empty └Forward	repeat └if *PathAhead* └TurnRight └Forward	The experts suggest to delete the error in the conditional statement whereas we choose to insert the correct solution.	if *PathAhead* └TurnLeft └Forward	repeat └if *PathAhead* └TurnLeft └Forward

if *PathAhead*
└repeat
 └list
  └Forward
  └TurnLeft
└Forward
	if *PathAhead*
	└list
	 └Forward
	 └TurnLeft
	└Forward
		if *PathAhead*
		└repeat
		 └Forward
		└Forward
			Experts remove the repeat-until block while we remove the erroneous rotation inside the repeat-until block. We would remove the repeat-until block in two steps.	list
				└if *PathAhead*
				 └Forward
				 └TurnLeft
				└repeat
				 └list
				  └Forward
				  └TurnLeft
					list
					└if *PathAhead*
					 └Forward
					 └TurnLeft
					└repeat
					 └Forward

repeat
└list
 └Forward
 └TurnLeft
 └if *PathAhead*
  └Forward
  └Forward
 └Forward
	repeat
	└list
	 └Forward
	 └if *PathAhead*
	  └Forward
	  └Forward
	 └Forward
		repeat
		└list
		 └Forward
		 └TurnLeft
		 └if *PathAhead*
		  └Forward
		  └Forward
			The experts suggest to delete an inner erroneous rotation, whereas we delete the last erroneous rotation that keeps the student longer on path.	repeat
				└list
				 └Forward
				 └TurnLeft
				 └if *PathAhead*
				  └Forward
				  └TurnLeft
				 └Forward
					repeat
					└list
					 └Forward
					 └TurnLeft
					 └if *PathAhead*
					  └Forward
					  └TurnLeft

repeat
└list
 └Forward
 └TurnLeft
 └if *PathAhead*
  └TurnRight
  └Forward
	repeat
	└list
	 └Forward
	 └TurnLeft
	 └if *PathAhead*
	  └Empty
	  └Forward
		repeat
		└list
		 └Forward
		 └if *PathAhead*
		  └TurnRight
		  └Forward
			The experts suggest to handle the errors in the conditional statement first, whereas we remove the erroneous rotation inside the statement list.	repeat
				└list
				 └Forward
				 └TurnLeft
				 └if *PathAhead*
				  └TurnLeft
				  └TurnRight
					repeat
					└list
					 └Forward
					 └if *PathAhead*
					  └TurnLeft
					  └TurnRight

repeat
└list
 └Forward
 └TurnLeft
 └if *PathLeft*
  └Forward
  └Forward
	repeat
	└list
	 └Forward
	 └TurnLeft
	 └if *PathAhead*
	  └Forward
	  └Forward
		repeat
		└list
		 └Forward
		 └if *PathLeft*
		  └Forward
		  └Forward
			The experts suggest to handle the errors in the conditional statement first, whereas we remove the erroneous rotation inside the statement list.	repeat
				└list
				 └Forward
				 └TurnLeft
				 └if *PathAhead*
				  └TurnLeft
				  └Forward
					repeat
					└list
					 └Forward
					 └if *PathAhead*
					  └TurnLeft
					  └Forward

repeat
└list
 └Forward
 └if *PathAhead*
  └list
   └TurnLeft
   └Forward
  └list
   └TurnRight
   └Forward
	repeat
	└list
	 └Forward
	 └if *PathAhead*
	  └list
	   └TurnLeft
	   └Forward
	  └list
	   └TurnLeft
	   └Forward
		repeat
		└list
		 └Forward
		 └if *PathAhead*
		  └Forward
		  └list
		   └TurnRight
		   └Forward
			The expers' suggest a substitution in the else-block of the conditional statement, which leads the student around the corner. We suggest to handle the error in the true-block	repeat
				└list
				 └Forward
				 └if *PathAhead*
				  └list
				   └TurnLeft
				   └Forward
				  └list
				   └TurnLeft
				   └Forward
					repeat
					└list
					 └Forward
					 └if *PathAhead*
					  └Forward
					  └list
					   └TurnLeft
					   └Forward

Continued on next page

Table 5 – continued from previous page

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies* *Input*	*GS hint*

`repeat`
`└list`
` └Forward`
` └if` *PathLeft*
`  └Forward`
`  └Forward`
	`repeat`
	`└if` *PathLeft*
	` └Forward`
	` └Forward`
		`repeat`
		`└list`
		` └Forward`
		` └if` *PathAhead*
		`  └Forward`
		`  └Forward`
			The experts suggest to remove the erroneous forward movement, we substitute the error in the condition of the conditional statement.	`repeat`
				`└list`
				` └Forward`
				` └if` *PathLeft*
				`  └Forward`
				`  └TurnLeft`
					`repeat`
					`└list`
					` └Forward`
					` └if` *PathAhead*
					`  └Forward`
					`  └TurnLeft`

`list`
`└Forward`
`└TurnLeft`
`└if` *PathAhead*
` └TurnLeft`
` └Forward`
	`list`
	`└Forward`
	`└if` *PathAhead*
	` └TurnLeft`
	` └Forward`
		`list`
		`└Forward`
		`└TurnLeft`
		`└if` *PathAhead*
		` └Empty`
		` └Forward`
			The experts delete the erroneous rotation in the first level, whereas we delete the erroneous rotation in the deepest level.	`list`
				`└Forward`
				`└TurnLeft`
				`└if` *PathAhead*
				` └Forward`
				` └Forward`
					`list`
					`└Forward`
					`└TurnLeft`
					`└if` *PathAhead*
					` └Forward`
					` └Empty`

`list`
`└Forward`
`└TurnLeft`
`└repeat`
` └list`
`  └Forward`
`  └if` *PathLeft*
`   └TurnLeft`
`   └Forward`
	`list`
	`└Forward`
	`└repeat`
	` └list`
	`  └Forward`
	`  └if` *PathLeft*
	`   └TurnLeft`
	`   └Forward`
		`list`
		`└Forward`
		`└TurnLeft`
		`└repeat`
		` └if` *PathLeft*
		`  └TurnLeft`
		`  └Forward`
			The experts delete the erroneous rotation in the first level, whereas we delete the erroneous rotation in the deepest level	`list`
				`└Forward`
				`└TurnLeft`
				`└repeat`
				` └list`
				`  └Forward`
				`  └if` *PathAhead*
				`   └TurnLeft`
				`   └Forward`
					`list`
					`└Forward`
					`└TurnLeft`
					`└repeat`
					` └if` *PathAhead*
					`  └TurnLeft`
					`  └Forward`

`repeat`
`└list`
` └Forward`
` └if` *PathAhead*
`  └Forward`
`  └list`
`   └TurnLeft`
`   └Forward`
	`repeat`
	`└list`
	` └Forward`
	` └if` *PathAhead*
	`  └Forward`
	`  └TurnLeft`
		`repeat`
		`└if` *PathAhead*
		` └Forward`
		` └list`
		`  └TurnLeft`
		`  └Forward`
			The experts delete an error deeper in the AST. We delete such that program starts correctly. Both reach the goal.	`list`
				`└if` *PathAhead*
				` └Forward`
				` └TurnLeft`
				`└Forward`
					`repeat`
					`└list`
					` └if` *PathAhead*
					`  └Forward`
					`  └TurnLeft`
					` └Forward`

`repeat`
`└list`
` └Forward`
` └if` *PathAhead*
`  └TurnRight`
`  └TurnLeft`
` └Forward`
` └Forward`
	`repeat`
	`└list`
	` └Forward`
	` └if` *PathAhead*
	`  └TurnRight`
	`  └TurnLeft`
	` └Forward`
		`repeat`
		`└list`
		` └Forward`
		` └if` *PathAhead*
		`  └Empty`
		`  └TurnLeft`
		` └Forward`
		` └Forward`
			The experts remove the last erroneous movement. We delete the rotation inside the conditional statement so the character reaches the goal by applying SingleEdit.		

Table 6: Internal inconsistencies in assignment 18 in the heuristic `InsStart`

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies* *Input*	*GS hint*

`list`
`└if` *PathAhead*
` └Forward`
` └TurnLeft`
`└Forward`
	`repeat`
	`└list`
	` └if` *PathAhead*
	`  └Forward`
	`  └TurnLeft`
	` └Forward`
		`if` *PathAhead*
		`└Forward`
		`└TurnLeft`
			The experts suggest InsStart which inserts the correct start. We delete the erroneous rotation.	`list`
				`└if` *PathAhead*
				` └Forward`
				` └TurnLeft`
				`└TurnLeft`
					`if` *PathAhead*
					`└Forward`
					`└TurnLeft`

Table 7: Internal inconsistencies in assignment 18 in the heuristic `SubstErr`

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies* *Input*	*GS hint*

`list`
`└Forward`
`└if` *PathAhead*
` └TurnLeft`
` └TurnRight`
	`list`
	`└Forward`
	`└if` *PathAhead*
	` └TurnLeft`
	` └TurnLeft`
		`list`
		`└Forward`
		`└repeat`
		` └if` *PathAhead*
		`  └TurnLeft`
		`  └TurnRight`
			The experts substitute an error in the conditional statement, whereas we insert the correct start.	`list`
				`└Forward`
				`└if` *PathAhead*
				` └TurnRight`
				` └Forward`
					`list`
					`└Forward`
					`└repeat`
					` └if` *PathAhead*
					`  └TurnRight`
					`  └Forward`

Continued on next page

Table 7 – continued from previous page

Input	*GS hint*	*Our hint*	*Comment*	*Internal inconsistencies* *Input*	*GS hint*

repeat
└list
  └Forward
  └if *PathAhead*
    └TurnRight
    └TurnRight
	repeat
	└list
	  └Forward
	  └if *PathAhead*
	    └TurnRight
	    └TurnLeft
		repeat
		└list
		  └Forward
		  └if *PathAhead*
		    └Empty
		    └TurnRight
			The experts suggest to substitute the error in the else-block, whereas we correct the error in the true-block.	repeat
				└list
				  └Forward
				  └if *PathAhead*
				    └TurnLeft
				    └Forward
					repeat
					└list
					  └Forward
					  └if *PathAhead*
					    └Empty
					    └Forward

repeat
└list
  └Forward
  └if *PathLeft*
    └TurnLeft
    └list
      └Forward
      └Forward
	repeat
	└list
	  └Forward
	  └if *PathAhead*
	    └TurnLeft
	    └list
	      └Forward
	      └Forward
		repeat
		└list
		  └Forward
		  └if *PathLeft*
		    └TurnLeft
		    └Forward
			The experts substitute the erroneous condition, whereas we suggest a deletion that leads to a partial	repeat
				└list
				  └Forward
				  └if *PathLeft*
				    └TurnLeft
				    └list
				      └Forward
				      └TurnRight
					repeat
					└list
					  └Forward
					  └if *PathLeft*
					    └TurnLeft
					    └Forward

repeat
└list
  └Forward
  └if *PathLeft*
    └TurnRight
    └TurnRight
	repeat
	└list
	  └Forward
	  └if *PathLeft*
	    └TurnLeft
	    └TurnRight
		repeat
		└list
		  └Forward
		  └if *PathAhead*
		    └TurnRight
		    └TurnRight
			The experts suggest to correct the true-statement and lead the student around the corner. We suggest to correct the condition in the if-then-else statement.	repeat
				└list
				  └Forward
				  └if *PathLeft*
				    └TurnRight
				    └TurnLeft
					repeat
					└list
					  └Forward
					  └if *PathAhead*
					    └TurnRight
					    └TurnLeft

repeat
└list
  └Forward
  └if *PathLeft*
    └TurnLeft
    └Forward
  └if *PathRight*
    └TurnRight
    └Forward
	repeat
	└list
	  └Forward
	  └if *PathLeft*
	    └TurnLeft
	    └Forward
	  └if *PathAhead*
	    └TurnRight
	    └Forward
		repeat
		└list
		  └Forward
		  └if *PathLeft*
		    └TurnLeft
		    └Forward
		  └if *PathRight*
		    └TurnRight
		    └Empty
			The experts substitute an erroneous condition in the last control-flow statment. We delete the last forward movement inside the last conditional statement so the character reaches the goal by applying SingleEdit.	list
				└if *PathAhead*
				  └Forward
				  └TurnLeft
				└Forward
					repeat
					└list
					  └if *PathAhead*
					    └Forward
					    └TurnLeft
					  └Forward