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

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

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. By comparing our results with results of a previous experiment on Hour of code we show that a heuristics-based approach to providing hints gives results that are impossible to further improve. 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

THE HOUR OF CODE

The Hour of code on Code.org's Code Studio[1] 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 [2] and Snap [4]. 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 will run 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 loops and if-then-(else) statements are additionally available as well. Solutions to these assignments can be found in Figures 1 and 2, respectively.

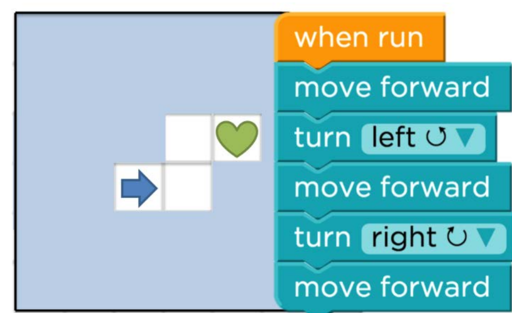

Figure 1: The solution to assignment 4.

Figure 2: The solution to assignment 18.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

L@S 2018, June 26–28, 2018, London, United Kingdom

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-5886-6/18/06. . . $15.00
DOI: https://doi.org/10.1145/3231644.3231690

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, 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 goal from each possible program?

Feedback is important for learning [5]. Feedback can take several forms, and one important such form is feed-forward, or a hint: giving a learner information about how to proceed with a partial or erroneous solution. Automatically generated hints are particularly useful in situations where direct contact with teachers or experts is not available, which is often the case when learners are working with on-line assignments as in the Hour of code.

Automatically generating hints for programming exercises is challenging [1]. Recently, we have seen a trend in using previous student data to generate hints from programming exercises. For example, Piech et al. [3] compare various algorithms that all use learners' input data but differ in the method of selecting the best next-step. A next step is a program one step closer to a correct program. Piech et al. apply multiple algorithms to generate hints for two Hour of code assignments using over one million partial programs per assignment. They obtain an accuracy of 95.9% for the first assignment and 84.6% for the second when comparing steps generated by the 'Problem Solving Policy' they develop to a gold standard set produced by human experts.

We think using data to provide hints in the Hour of code exercises is the wrong approach. The Hour of code exercises are simple enough to describe all necessary feedback upfront, without using any student data. Instead of calculating a Problem Solving Policy based on millions of student submissions, we propose to use a number of simple heuristics to give hints in the Hour of code exercises. Not only can heuristics provide better hints, it also avoids the cold-start problem: generating hints based on learner data requires a minimum amount of data that is obtained from learners or an expert. In the case of Piech et al. the best hint generation algorithm would need two thousand learners to obtain the acquired accuracy.

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. Since we are still working on assignment 18, this is a work-in-progress paper.

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 developed the optimal solution except, for example, one code block, we can offer a suggestion to substitute the erroneous block for the right block.

A learner is allowed to 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]`.

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 two hour of code assignments. 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 assignment 4, and the patterns we can distinguish in the expert hints.

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.

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"

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. 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 an expert applies 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: $dynScore(i,h) = startSegmentLength(h) -$

Table 2: DelErrRot deviating results

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

$programLength(h) - errorLocation(i,h)$, 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: `OnTrack` gets 0, and each subsequent heuristic one more, ending with 6 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 4.6), `DelErrRot` (to 6.1), and `SubstErrRot` (to 7.0) to obtain the highest accuracy.

RESULTS

For 223 out of 225 cases we give the same hint. For each of the 2 remaining cases, we can identify conceptually contradicting expert hints, for example, suggesting an edit either at the beginning or at the end of an erroneous program. All the heuristics we suggest correspond to the expert hints, except for `DelErrRot`, for which we are correct 58 out of 60 times.

Table 2 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".

CONCLUSIONS

We have developed a heuristics-based approach to giving hints for the Hour of code exercises. 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.

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 exercises. 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 exercises.

REFERENCES

1. 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.
2. 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.
3. 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.
4. Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 483–488.
5. Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.