Interactive Learning Environments

ISSN: 1049-4820 (Print) 1744-5191 (Online) Journal homepage: https://www.tandfonline.com/loi/nile20

Programming trajectories analytics in block-based programming language learning

Bo Jiang, Wei Zhao, Nuan Zhang & Feiyue Qiu

INTERACTIVE LEARNING ENVIRONMENTS
https://doi.org/10.1080/10494820.2019.1643741

Programming trajectories analytics in block-based programming language learning

Bo Jiang, Wei Zhao, Nuan Zhang and Feiyue Qiu

College of Educational Science and Technology, Zhejiang University of Technology, Hangzhou Shi, People's Republic of China

ABSTRACT

Block-based programing languages (BBPL) provide effective scaffolding for K-12 students to learn computational thinking. However, the output-based assessment in BBPL learning is insufficient as we can not understand how students learn and what mistakes they have had. This study aims to propose a data-driven method that provides insight into students' problem-solving process in a game-based BBPL practice. Based on a large-scale programing dataset generated by 131,770 students in solving a classical maze game with BBPL in *Hour of Code*, we first conducted statistical analysis to extract the most common mistakes and correction trajectories students had. Furthermore, we proposed a novel program representation method based on tree edit distance of abstract syntax tree to represent students' programing trajectories, then applied a hierarchical agglomerative clustering algorithm to find the hidden patterns behind these trajectories. The experimental results revealed four qualitatively different clusters: quitters, approachers, solvers and knowers. The further statistical analysis indicated the significant difference on the overall performance among different clusters. This work provides not only a new method to represent students' programing trajectories but also an efficient approach to interpret students' final performance from the perspective of programing process.

ARTICLE HISTORY
Received 10 September 2018
Accepted 11 July 2019

KEYWORDS
Block-based programing; computational thinking; learning trajectories analytics

1. Introduction

Computational thinking (CT) is the thinking process involved in formulating problems and expressing solutions in a way that computers can perform effectively (Wing, 2014). Programing is a complex cognitive activity that requires students to solve problems using multiple CT skills, i.e. computational concepts, practices and perspectives (Brennan & Resnick, 2012; Pea & Kurland, 1984; Resnick et al., 2009). Recently, block-based programing languages (BBPL) like Scratch (Resnick et al., 2009), Snap! (Garcia, Harvey, & Barnes, 2015) and Blockly (Fraser, 2015) are becoming popular in K-12 CT education. Compared with text-based programing languages, BBPL represent language constructs as visual blocks with different colors and shapes, which greatly reduce cognitive load for novices (David, Jeff, Caitlin, Josh, & Frankly, 2017). Although BBPL learning platforms provide helpful scaffolding and interesting practices for students to learn CT, evaluating novice learning in programing is challenging. Programing is essentially a problem-solving process that finding a correct path through the problem space from the initial state to the goal state (Newell & Simon, 1972). What many novice programers struggle with most is not how to use language constructs, but rather how to compose and coordinate components of a program to find a correct path (Blikstein et al., 2014). The widely used output-based

CONTACT Bo Jiang bjiang@zjut.edu.cn

assessment can not reflect how students solve mistakes and make progress in programing tasks (Helminen, Ihantola, Karavirta, & Malmi, 2012; Hosseini, Vihavainen, & Brusilovsky, 2014; Piech, Sahami, Koller, Cooper, & Blikstein, 2012).

Understanding students' programs evolution is important because it not only improves student modeling in intelligent programing tutoring systems, but also provides better individual feedback to struggling students in classroom (Hosseini et al., 2014). Unlike other subjects, programing learning generally requires computers, more specifically, the development environments, which make students' program be collected more easily. The programs submitted by students reflect students' conception and misconception of related CT skills. Program analysis, also called code analysis, was early used in computer science education field to detect program plagiarism (Chen, Francia, Li, Mckinnon, & Seker, 2004). Recently, the code analysis method has been used for hint generation (Mokbel, Gross, Paassen, Pinkwart, & Hammer, 2013; Paaßen et al., 2018; Piech, Sahami, Huang, & Guibas, 2015; Price, Dong, & Barnes, 2016; Rivers & Koedinger, 2017), teacher support (Diana et al., 2017; McBroom, Yacef, Koprinska, & Curran, 2018), automatic grading (Diana et al., 2018) as well as knowledge tracing (Rivers & Koedinger, 2014; Wang, Sy, Liu, & Piech, 2017b).

Another distinctive feature of programing learning is that it requires students to iteratively write, test and revise their programs to complete a given task (Paaßen et al., 2018; Rivers & Koedinger, 2017). When working on such multi-steps tasks, the intermediate program sequences generated in multiple attempts reflect how students developed their programs over time, which are invisible to teachers and cannot be observed in the final programs (Helminen et al., 2012). In this work, we call all the intermediate programs a student generated in solving task as a programing trajectory. Programing trajectories analytics was used in the software engineering field to track the evolution of software (Fluri, Wuersch, PInzger, & Gall, 2007), and recently in computer science education for students progress modeling (Helminen et al., 2012; Hosseini et al., 2014; Piech et al., 2012).

This research conducts a programing trajectories analytics on a large-scale dataset generated by more than 100,000 valid students in solving an interesting maze problem in *Hour of Code* [1] (Piech et al., 2015). This dataset contains not only the final program submitted by students but also all intermediate programs. Starting from this dataset, we aim to answer the following three research questions (RQs) in this work.

RQ 1 What is the most common misconception students have and how do they correct their errors?
RQ 2 Are there any hidden patterns behind the programing trajectories?
RQ 3 If the hidden patterns exist, are these patterns associated with students' knowledge proficiency?

For the first question, we used statistical analysis to investigate the most common misconception students have and look insight into their error correction strategies. To answer the second one, we first proposed a simple yet efficient indicator based on tree edit distance to represent students' program sequences as real-value sequences. After that, we used the dynamic time warping (DTW) algorithm (Keogh & Pazzani, 2001) to compute the similarities between different programing trajectories. The similarity matrix of programing trajectories is then fed into a hierarchical agglomerative clustering algorithm to find the trajectories pattern. After the clustering process completed, students' future performance in different patterns are investigated via statistical analysis.

2. Related work

Understanding how students solve problem with program language has been in the center of interest of computer science educator for decades. Data-driven method has recently been used to study patterns of behaviors in program construction, compilation, and debugging (Blikstein, 2011; Blikstein et al., 2014; Jadud, 2006; Perkins & Martin, 1986). By observing and interacting with students engaged in programing, study by Perkins and Martin (1986) found three kinds of novice programer, they are *stoppers*, *movers*, and *tinkerers*. *Stoppers* would give up when encounter a difficult problem, *movers* gradually work towards a correct solution, and *tinkerers* iteratively write some code and then make small changes until the problem is solved. Jadud (2006) also found the similar students by analyzing students' compilation behaviors and syntax errors during programing. By analyzing students' key presses, button clicks and code snapshots data, Blikstein (2011) revealed three coding strategies, which are *copy and pasters*, *self-sufficients* and *mix-model*. Study by Blikstein et al. (2014) tried to use the size of the code updates as an indication of programing style and related it to course performance. Although well-defined clusters in students' code update size were found, but the analysis results did not show the strong association with performance. Blikstein further considered not only the update size but also the update frequency, the results showed a stronger but still not significant correlation with performance. It is not surprised that sometimes learning behavior is not closely co-related to the learning performance because learning behavior is influenced by the multiple interactions of how the student thinks, feels and interacts (Powell & Tod, 2004). Recently, some studies proposed to model students' problem-solving path in programing task using fine-grained source code analysis.

A concept extraction tool called *JavaParser* is developed by Hosseini et al. (2014) to extract the programing concepts (for example, method definition and *if-else* statement) used in students source code. With *JavaParser*, the author examined the conceptual change patterns from students' start state to final state and found out the association between conceptual change and correctness of program. The results indicate a clear programing pattern that most of the students start with a small program, and make progress by incrementally adding concepts in each step. The extracted concepts-call-sequence set indicates which concepts students used, but cannot reflect relationships among different concepts.

Study by Piech et al. (2012) combined the concept-call-sequence and abstract syntax tree (AST) to measure the dissimilarity between code snapshots. They further modeled students' programing process using hidden Markov model, where the programing path is regarded as the transition path among different states. A clustering process was first conducted to learn the hidden state, then another clustering algorithm was applied to find the patterns how the students transit through in the space of hidden state. The clustering results reveal several sink states where the students clearly had serious functional problems. Besides, the results also show once a student transited to such a sink state, the student had a high probability of remaining there through several code updates.

A programing path based hint generation method is proposed by Rivers and Koedinger (2014, 2017). Given the current solution of a student, they first identified a nearby optimal goal solution, then generated all feasible intermediate solutions from the current solution to the goal solution. Furthermore, all intermediate solutions in the path were evaluated by a desirability metric that is a weighted sum of their distance to the current solution, solution frequency, solution quality, and finally their distance to the optimal solution. The solution with the highest desirability metric was selected as next-step hint. Here, tree edit distance of AST was used to measure the distance between two solutions.

The solution frequency was also used by Piech et al. (2015) to generate hints. The author assumed the amount of time required for an average student to generate a partial solution is a Poisson process whose rate parameter can be estimated via frequency of this partial solution. The path from a partial solution to a perfect solution is defined as the smallest expected time generated by the average successful student. This method is easy to implement and experimental results show that it can provide more accurate hints than current state-of-the-art methods, such as Stamper's hint factory method (Stamper, Barnes, Lehmann, & Croy, 2008) and River's path construction method (Rivers & Koedinger, 2014, 2017).

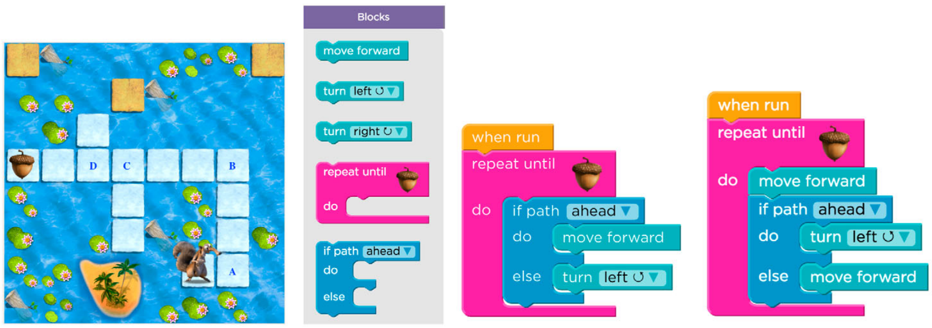
The programing trajectory can also be used for knowledge tracing. Piech et al. (2012) has shown the generated programing trajectory model has high correlation with students' conception on related programing concepts and test scores. Study by Piech et al. (2015) also showed the quality of Poisson path is highly predictive of student's success on the next task that has the same knowledge components. Inspired by these work, some most recent work (Wang, Sy, Liu, & Piech, 2017a; Wang et al., 2017b) proposed a deep knowledge tracing (DKT) model for programing learning (Piech et al., 2015) that only uses programing path data within a single task to predict students' performance on next task. Given the AST of students' all intermediate submissions generated in solving a problem, this method used a recursive neural network to create program embedding of the AST sequences, which afterward feed into a recurrent neural network model to conduct knowledge tracing. The comparison results show the DKT-based method outperformed the Poisson path method.

As seen from the above literature review, existing work in understanding students' programing is gradually moving from programing behavior analysis to more in-depth code analysis. Particularly, investigating the evolution of code, i.e. programing trajectories, makes us more likely to look insight into students' conception and misconception on CT concepts. However, due to the lack of data and the complexity of technology, there are few concerns about the programing trajectories analytics. This work conducted a programing trajectories analytics on a large-scale programing learning dataset, provided by Code.org, [2] which contains more than 100,000 valid students. We investigated students' error corrections on the most common mistake, proposed a simple yet efficient technology to represent programing trajectories and finally conducted a cluster analysis to find the hidden pattern behind these trajectories. The proposed methods can be integrated into teacher dashboard as a component for automatically detecting and visualizing students' progress in programing.

3. Dataset and statistical analysis

3.1. Dataset description

The HOC18 dataset used in this work is generated when students solve a maze problem using block-based programing language Snap! on the *Hour of Code* .[3] The dataset was collected by researchers in Stanford University (Piech et al., 2015). The programing task is depicted in Figure 1(a). In this task, students need to combine the given five blocks, as shown in Figure 1(b), to make the squirrel get the pine cone safely. The five blocks include three motion blocks (*move forward*, *turn left* and *turn right*) and two control blocks (*repeat-do* and *if-else*). As illustrated in Figure 1(a), the key to solve this problem is to make the squirrel keep moving and turn to the correct direction at the four intersections ('A'-'D'). Therefore, the control block *if-else* and the three motion blocks are used to decide when and which direction to turn, and the control block *repeat-do* is used to repeat squirrel's moving. The knowledge components in this maze game are loops, conditionals and their nesting, all of which are core computational thinking concepts (Brennan & Resnick, 2012).



```
when run
repeat until
do
    if path ahead
    do
        move forward
    else
        turn left
```

```
when run
repeat until
do
    move forward
    if path ahead
    do
        turn left
    else
        move forward
```

(a) Problem description (b) Given blocks (c) The best solution S0 (d) The most common error solution S1

Figure 1. The HOC18 programing task. (a) Problem description (b) Given blocks. (c) The best solution S0 and (d) the most common error solution S1.

When students executed their codes, the logging system captured the abstract syntax trees (ASTs) representation of these codes and stored them as JSON files. If a student attempted multiple times on a task, multiple ASTs were generated over time to produce a programing trajectory. Figure 2 shows an example of the tree structure of the AST data used in this work. The modifications on a source code can be represented by the editing sequences on AST that transform it from one state to another. A student's programing trajectory consists of the submissions sequence generated in all attempts he or she took to solve the problem. For example, the sequence S1→S4→S0 means the student ran submission S1 first, then changed solution to S4 and finally submitted S0. The original dataset contains 79,553 unique code submissions and 83,955 trajectories that made by 263,569 logged-in students of *Hour of Code*. Some submissions and user ID are missed in the original data, so we first conduct data cleaning to identify and drop all invalid trajectories. Finally, we got 51,907 valid trajectories that consists of 47,129 unique code submissions generated by 131,770 students.

3.2. Statistical analysis of programing trajectories

The task requires students to complete this task using as few blocks as possible. The best solution to this challenge is given in Figure 1(c). The best solution S0 just has five lines and all the five blocks are only used once in it. The strategy of solution S0 is that, if there is a maze ahead, move forward, otherwise turn left. This process repeats until the squirrel get the pine cone. Statistical analysis found that, the best solution S0 is found in 37,473 (73.4%) trajectories, which indicates the problem is relatively easy for these students. However, we found only 45.2% students got this with one-shot, which means more than half students had difficult when solving this task. Furthermore, we found that 11,647 (22.4%) trajectories contains the most common error solution S1, which is depicted in Figure 1(d). Compared with the best solution S0, the error solution S1 has an obvious logic error within the *if-else* construct. The condition logic in solution S1 means that if there is a feasible path ahead, the squirrel should turn left, else move ahead. This is obviously contrary to common sense. Also, the first *move forward* block is redundant as it can be included in the condition construct.

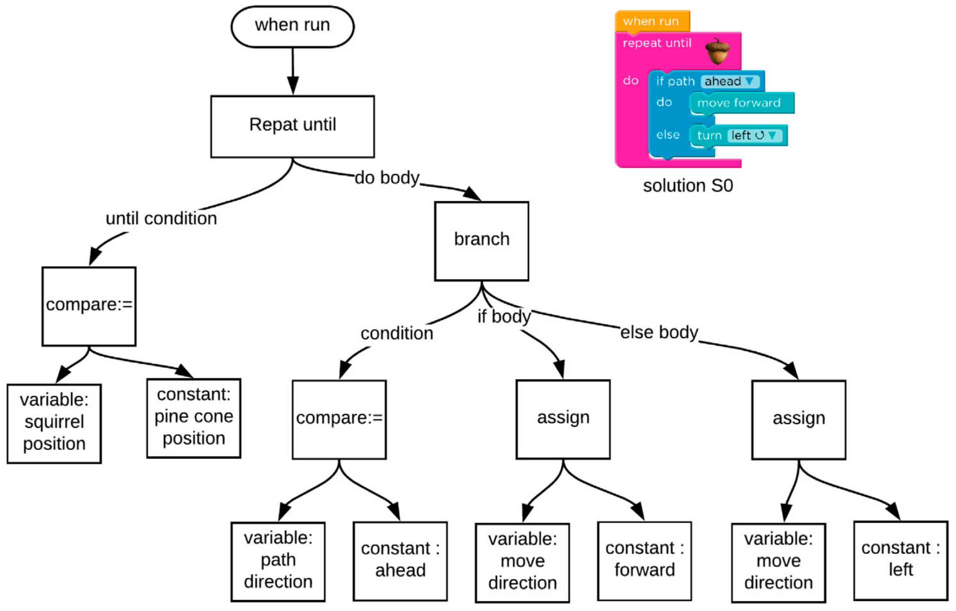


Figure 2. Example: abstract syntax tree for solution S0.

We are also interested in how students escaped from the error solution S1. All students' trajectories from the state S1 is visualized in Figure 3. As can be seen from Figure 3, the most dominant error-correction path is S1→S2, and 3477 (30.0%) students updated their solution to S2. The path S1→S3 is also very popular as there were 1609 (13.8%) students moved toward solution S3. Another two popular solutions that yielded nearly a thousand times are solution S13 and S4. In contrast, only 593 (5.0%) students could get the best solution S0 with one attempt. The four popular trajectories and their real scripts are illustrated in Figure 4. We can see that, except S3, all the other three solutions still contain evident logic errors. Particularly, it seems that there is no logic and direction errors in S2, but putting "*path to the left*" as a condition leads to left-turn at intersection C. This error shows that these students either misunderstood the execution order in *if-else* construct or did not understand the task well. On the contrary, the students who submitted S3 recognized clearly their direction error in S1 and revised it correctly. However, solution S3 is still not global optimal as the first '*move forward*' block is redundant. As shown in Figure 3, only 10.5% students with solution S3 realized this issue and made a perfect moving from S3 to S0. In another popular trajectory S1→S4→S0, students first dropped the redundant "*move forward*" block to generate S4, then fixed the direction error in *if-else* block. Compared with trajectory S1→S3, the frequency of trajectory S1→S4 is lower, but about 35% students with solution S4 moved to the best solution with one

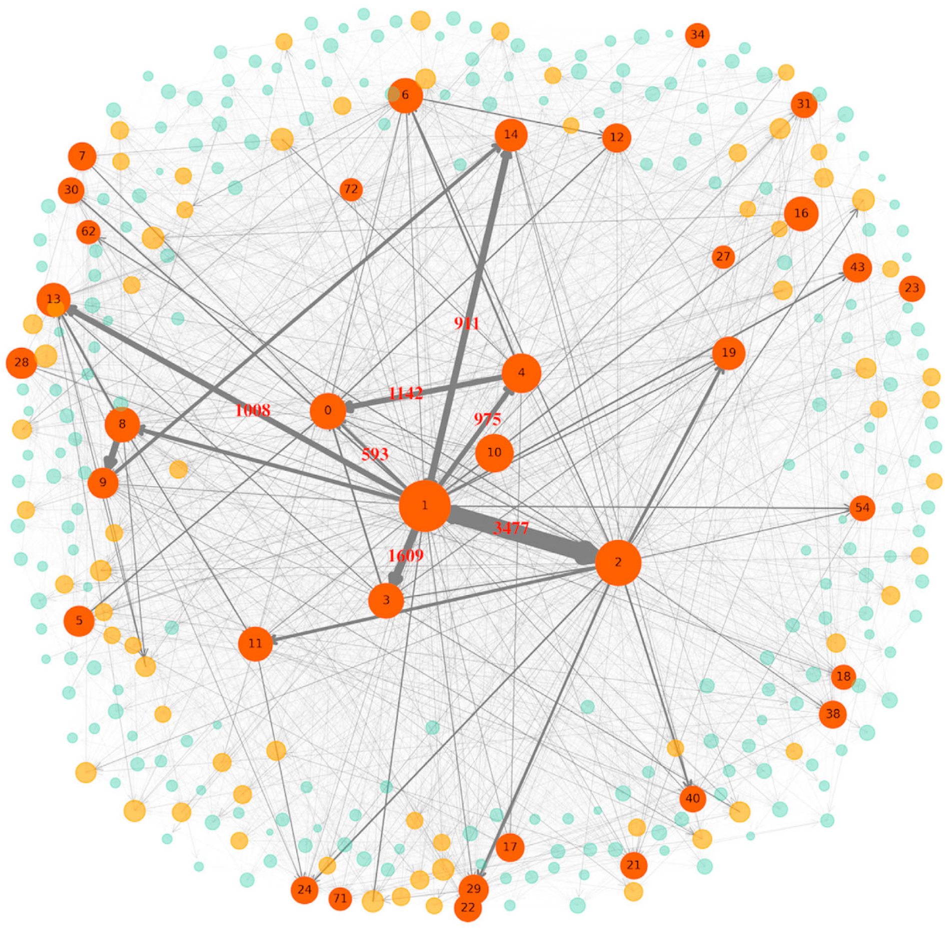


Figure 3. Students' programing trajectories from solution S1 (node with label "1"). Each node represents a solution that labeled by the solution number. students' moving direction is represented by directed edges, on which the numbers denote the frequency. The node size and edge thickness are proportional to their occurrences in dataset.

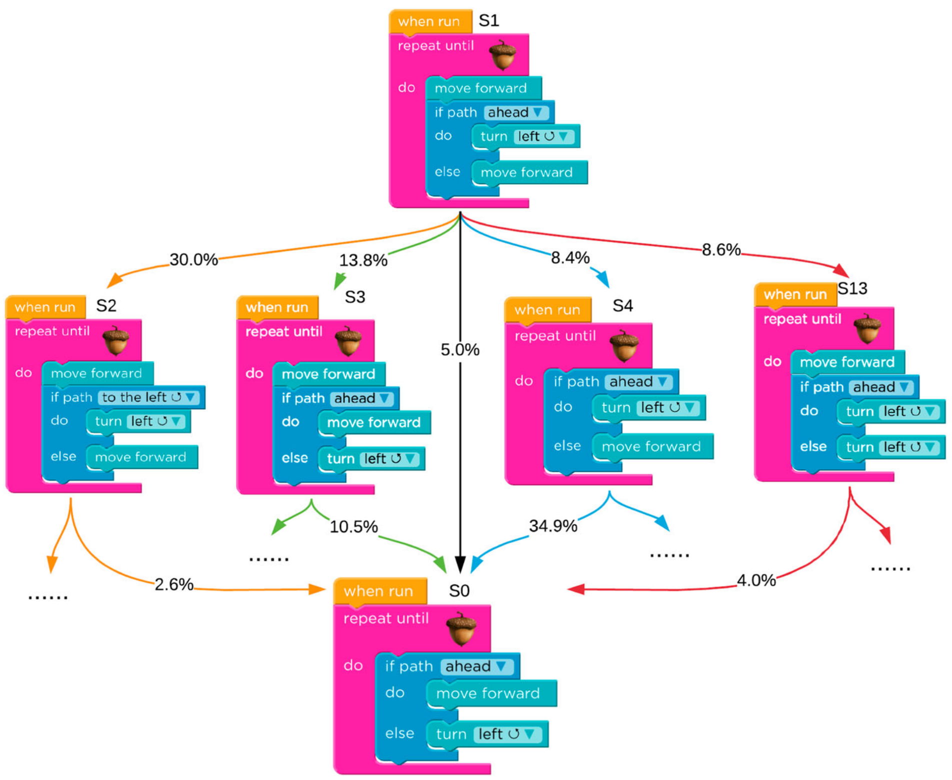


Figure 4. Two most common trajectories generated when students tried to solve the error in S1.

attempt, and the number is only 10.5% in the former. The last frequent trajectory is S1→S13. The solution S13 is even worse than S1 because the blocks inside '*if* ' body and '*else*' body are the same, which makes the squirrel always turn left until it falls into the water.

The above trajectory analysis demonstrates some students' misconceptions on CT concepts and provides insights into how students make progress step by step. From the case study in Figure 3 and Figure 4, we also found that the different error correction strategies lead to different final submissions. For example, the move from S1 to S4 is promising because it make the solution closer to the optimal solution S0. More than 34.9% students are able to move from S4 to S0 with only one attempt. In contrast, the move from S1 to S2 is unfavorable as the solution S2 is farther from the optimal solution. From S2, only 2.6% students can successfully find the optimal solution with one attempt. This finding inspired us to detect the trend of students' programing trajectories in the next section.

4. Clustering analysis of programing trajectories

To find the pattern of students' programing trajectories, we first propose the approaching index to represent students' each intermediate submission. Furthermore, a clustering analysis is conducted on students' programing trajectories using dynamic time warping metric.

4.1. Method

The basic idea here is to present an indicator to identify how *close* an intermediate solution towards the optimal solution. Assume a student updated and ran his/her program *m* times to find a feasible solution. In our work, every time when the student have submitted a program, the program was captured and stored as an abstract syntax tree (AST), which is a finite, labeled and ordered tree. Each node of an AST is labeled to represent a program element and the hierarchical structure represents how these elements organized. In general, a branch node of AST can be labeled to represent function call, control structure or operator, and a leaf node represents variable or constant. Compared with program text and its tokenization, AST could preserve the nested structure of different program elements. In our example, the student's m attempts generated m ASTs $\{T_1, \ldots, T_m\}$, where $(m \geq 1)$. The AST of the optimal solution given by teacher is T_o, and $T_m = T_o$ if the final solution submitted by student matches the optimal solution.

We design an approaching index (AI) based on AST for measuring the similarity between students' current solution $T_i(i \in [1, m])$ and the optimal solution T_o. Using AST, students' edits on program are mapped to edits on AST, and the progress make by students in problem solving can be represented by AST evolution. There are three kinds of edits can change the state of AST, they are insertion, deletion and replacement. In general, there are $k(k \geq 1)$ different edit sequences $S = \{S_1, S_2, \ldots, S_k\}$ that can transform AST T_1 to T_2, and the shortest edit path from T_1 to T_2 are called tree edit distance (TED). To compute TED, each edit is assigned a *cost function* γ. The TED $d(T_1, T_2)$, between any two ASTs T_1 and T_2, is defined as the cost of the minimum cost sequence of edit operation that transforms T_1 to T_2 (Akutsu, 2010; Bille, 2005), which can be formally represented as

$$d(T_1, T_2) = \min\{\gamma(S)|S = \{S_1, S_2, \ldots, S_k\}\} \tag{1}$$

The above optimization problem is often treated as a dynamic programing problem and *ZhangShasha* algorithm (Zhang & Shasha, 1989) is one of the most popular TED algorithm in literature. The details of TED algorithm is not focus of this work, please refer (Paaßen, 2018) for more details. With TED, we can further define approaching index (AI) as the TED from T_i to the optimal solution T_o, i.e.

$$AI_i = d(T_i, T_o) = \min\{\gamma(S_{i\rightarrow o})\} \tag{2}$$

where $S_{i\rightarrow o}$ denotes all possible edit sequences that can transform T_j to the optimal solution T_o. From Equation 2, we can see that the AI is non-negative and its minimal value is zero, which means the current submission is the optimal solution. A solution with a high AI value indicates it is far away from the optimal solution, otherwise approaches the optimal solution. If a student makes progress step by step, the AI of his/her submissions would decrease gradually to zero. In contrast, if one failed to move forward to the optimal solution, the AI would rise with fluctuation or keep constant.

The AI sequence provides the possibility for us to detect different programing trajectory patterns via clustering algorithm. In this section, we design a clustering algorithm to explore whether there exists different trajectory patterns in AI sequences. Since each student may have different AI sequence length, how to calculate the similarity between AI sequences with unequal length is crucial. A popular sequence similarity measurement dynamic time warping (DTW)(Keogh & Pazzani, 2001) is used here to compute the distance between different AI sequences. The computation process of DTW is also given by Keogh and Pazzani (2001). After the similarity between each AI sequence has been constructed, we use hierarchical agglomerative clustering (HAC) to generate trajectory clusters. HAC has been successfully applied to many disciplines, and it is a conceptually and mathematically simple approach.

In this work, the Python package *zss*[4] was used to compute the *ZhangShasha* tree edit distance and the package *fastdtw*[5] was used to calculate DTW. The clustering algorithm HAC was implemented by *AgglomerativeClustering* model in *scikit* .[6]

4.2. Results

To discover the trend of approaching index changes in the programing process, we conduct our clustering method on the subset of HOC18 dataset, where only 4210 trajectories with counts more than twice are kept. Figure 5 shows the dendrogram generated by HAC algorithm with Ward linkage. Due to the completed dendrogram is so large that cannot be show clearly in limited size, only the last 100 merged clusters are plotted in Figure 5. There is no gold rule to determine the cluster numbers, and the dendrogram can aid us to select a suitable cluster more easily. According to dendrogram in Figure 5, we select the cut-off distance at 2,000 (red dashed line in Figure 5) to generate four clusters.

The four trajectory clusters found is visualized in Figure 6, where the black dashed lines shows the change of average AI values with students' attempts. Note that the higher the AI, the worse the submission. If the AI of a student's solution is 0, it means he/she gets the best solution. We name the four clusters as quitters, approachers, solvers and knowers. As can be seen from Figure 6, the quitters gave up early, the approachers made some progress but failed to reach the best solution, the solvers made great progress and reached the best solution, and the knowers is proficient in solving the problem.

From Figure 6 we can see that the quitters had the worst start state and shortest trajectory length. The high starting AI values indicate these students may have serious misconception on the three knowledge components, or they totally are not familiar with block programing language. Also, the short trajectory length means they gave up early. A typical initial script submitted by quitters is shown in Figure 7(a). It is obviously that this student totally misunderstood the concept of loop and condition as he/she just used the motion block sequence to complete the task.

Also from Figure 6, we can see that the initial solutions submitted by approachers and solvers have very close average AI value, so they have the similar start state. However, most solvers reached the best solution after three attempts, which means they learned something and made great progress in these attempts. A typical example of trajectory of solvers is the previously mentioned $S1 \rightarrow S3 \rightarrow S0$, where the students reached the best solution with two attempts. In contrast, the approachers failed to reach the best solution, while they were very close to the best solution after three attempts. Also, most of the approachers gave up after at most five unsuccessful attempts.

Compared with other clusters, knowers had a perfect start state and achieved the best solution very easy. Specifically, there are 78,452 (66.3%) knowers achieved the best solution S0 with one-shot. Another popular trajectory of knowers is depicted in Figure 7(b), where the initial solution S5 is very close to solution S0, and simply adding a loop block would generate the best solution. We find there are more than 90% knowers solved the task perfectly within two attempts, but the maximal attempts number is 8, which seems unreasonable. To find out the reason behind, we extracted all trajectories with length more than 5 and found an interesting phenomenon, that is these students repeatedly submitted the same source code. Two typical examples of this kind of trajectory are

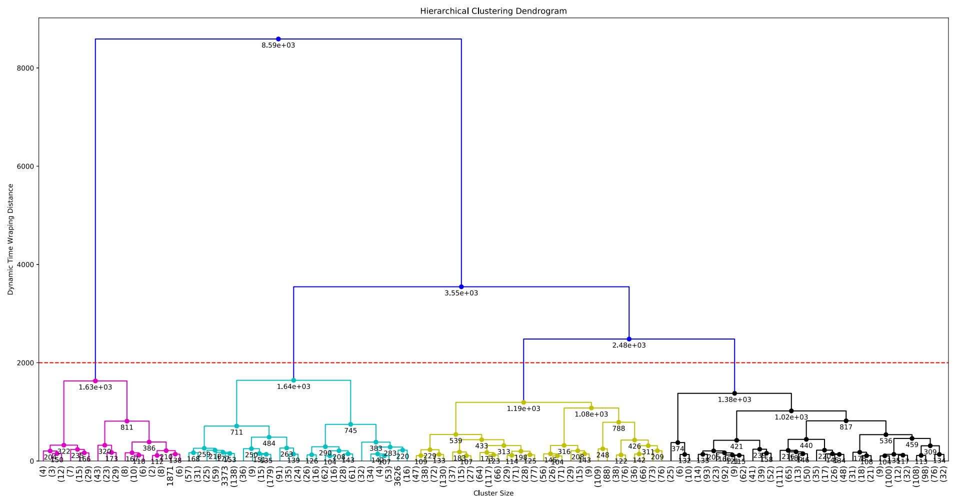


Figure 5. The dendrogram tree generated by Ward agglomerative clustering algorithm.

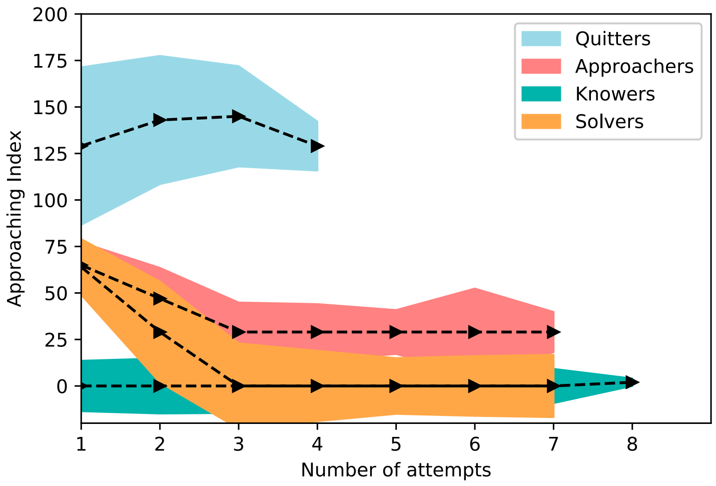


Figure 6. Programing trajectories clusters. The black dashed line shows the change of average approaching indicator in each cluster. The trajectory length in each cluster is determined by the maximal individual trajectory length.

- Example 1: $S7 \rightarrow S7 \rightarrow S7 \rightarrow S7 \rightarrow S7 \rightarrow S7 \rightarrow S7 \rightarrow S0$
- Example 2: $S27 \rightarrow S6 \rightarrow S6 \rightarrow S6 \rightarrow S6 \rightarrow S6 \rightarrow S4 \rightarrow S0$

Figure 7(c) shows the details of the Example 1, where the students repeated solution S7 six times. We can see that, the only difference between solution S7 and S0 is the turning direction within the '*else*' condition, which means the students with solution S7 have mastered the concept of loop and condition but have confusion on moving direction. A possible reason for this unproductive learning behavior may be they thought there are no mistakes or they can not find the mistakes.

5. Students' performance in each cluster

Finally, let's compare the overall performance of students in each clusters. Table 1 shows students average performance on HOC18 and HOC19 task. The HOC19 task is a little more complicated maze problem that has the same knowledge components with HOC18. In this work, when students

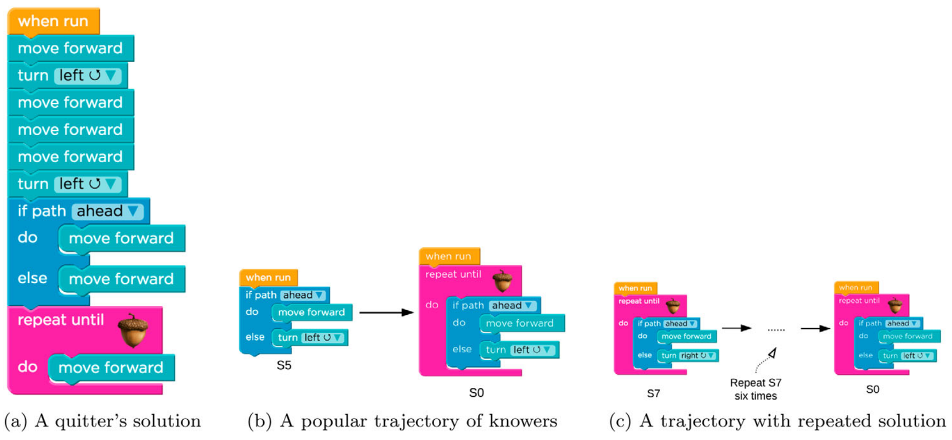

```
when run
move forward
turn left ↺
move forward
move forward
move forward
turn left ↺
if path ahead
do   move forward
else move forward
repeat until
do   move forward
```

Figure 7. Some typical submission and trajectories students generated in HOC18 dataset. (a) A quitter's solution. (b) A popular trajectory of knowers and (c) A trajectory with repeated solution.

Table 1. Cluster size and students' performance in each cluster. The numbers with asterick are the best results.

Cluster	Size	success% in HOC18	success% in HOC19
Quitters	566	0.00(0.00)	14.84(35.58)
Approachers	5,023	0.00(0.00)	75.47(43.03)
Solvers	7,914	99.95(2.25)*	96.99 (17.08)*
Knowers	118,267	93.78(24.15)	97.72(16.46)*

completed HOC18, the system would jump to HOC19 task automatically. Observing students' performance on HOC19 task can help us figure out whether students learned knowledge from HOC18 task, especially for the students failed in HOC18. For example, if a student failed in HOC18 task, but she learned knowledge from her error, we can expect she would perform better in HOC19 task. Here, we call a student failed if she couldn't submit the best solution S0.

As can be seen in Table 1, none quitters and approachers could solve the HOC18 task perfectly. On HOC19 task, the average success rate of quitters and approachers increased to 14.84% and 75.47%, respectively, but the standard deviations are very high. On one hand, the higher average success rate compared with HOC18 indicates some of these students indeed may have learned knowledge from their errors in HOC 18, which helped them to perform better on HOC19 task. On the other hand, the large standard deviation shows most of these students still failed to solve HOC19 task, which means they still have not acquired the knowledge to solve this kind of problem.

As shown in Table 1, average 99.95% solvers and 93.78% knowers succeeded in HOC18 task. On HOC18 task, the independent-samples *t*-test results show that there was a significant performance difference between solvers (M=99.95%, SD=2.25%) and knowers (M=93.78%, SD=24.15%); $t = 22.72, p = 10^{-114}$. For HOC19 task, most solvers and knowers can solve it perfectly, and there was no significant difference between them (t=−1.14,p=0.25). Furthermore, an independent-samples *t*-test was conducted to compare the performance of each cluster in HOC18 and HOC19. For solvers, there was a significant performance drop from (M=99.95%, SD=2.25%) to (M=96.99%, SD=17.08%); $t = -15.27, p = 10^{-52}$. For knowers, there was a significant performance improvement from (M=93.78%, SD=24.15%) to (M=97.72%, SD=16.46%); t=40.38,p=0. These results suggest that the difficulty of tasks really does have an effect on students' performance. Specifically, our results suggest that the increased difficulty reduced solvers' performance. As we know, most knowers have better start state than solvers, they are proficient in loop and conditional construct and just confused by moving direction. So, when the problem becomes more complicated but the knowledge involved in it does not change, they learned from their mistakes and performed very well. In contrast, the solvers had more or less misconception on loop or conditional construct, so they need to make more edits to transform their solutions to the best one. Although solvers learned how to use loop and conditional construct from their mistakes in HOC18, but they were still not proficient in the two knowledge components, so they are more sensitive to the difficulty change. Therefore, the solvers need more practices on these knowledge components than knowers.

6. Discussions and conclusion

This study answered our proposed three research questions by programing trajectories analytics. For the *RQ1*, we found the most common mistake students had and how they corrected it via statistical analysis on programing trajectories. We found some diversified trajectories that generated by different error correction strategies. Inspired by this, we further proposed a method to extract the trend of the programing trajectories to answer the *RQ2: Are there any hidden patterns behind the programming trajectories?* To do this, we presented an approaching index to represent each partial solution as its distance to the best solution based on the tree edit distance of their abstract syntax trees. After featurization, the time warping distance was used to calculate the similarity matrix among trajectories and the hierarchical agglomerative clustering algorithm was applied to conduct clustering.

The experimental results revealed four different trajectory patterns, we call them quitters, approachers, solvers and knowers. Finally, to answer *RQ3*, we investigated the performance of students in the four clusters. The results demonstrated that none of quitters and approachers solved the HOC18 problem, and their average performance somewhat improved on the more complicated HOC19 task. Also, most of the solvers and knowers solved the both HOC18 and HOC19 task perfectly.

The above result is significant since the data we used is students' programing process data within single practice. The process data provide us an unique insight into how students make progress, which is impossible to find out from final performance data. For example, the statistical analysis based on final program indicates neither quitters nor approachers succeeded in HOC18 task, but the trajectory pattern clustering results clearly show us they have different pathways. The quitters generally had a very poor start state and gave up the task only after few attempts. The approachers had a good start state, improved their solutions with few attempts and gave up finally. Take solvers and knowers as another example, most of students in these two clusters solved the task perfectly, but the trajectory pattern clustering results reveal the different pathways. Compared with knowers, the solvers had a relatively poor start state and most of them solved the task perfectly after multiple attempts. In contrast, the knowers had a great start state and they are proficient in the knowledge components related to the task. The quitter and solver have the similar activity feature as the *stopper* and *mover* that are found by Perkins and Martin (1986).

The proposed method can be used not only in BBPL learning environment but also in text-based programing language learning. The proposed learning analytic method may be used in the following two ways. Firstly, the trajectories analysis and clustering method could be used in developing teacher dashboard to dynamically visualize students' programing process. With the proposed method, teachers can easily monitor all students' programing progress and orchestrate lessons to ensure students access to learning resources they need. For example, for the quitters, teacher could push them more learning materials about the basic program construct as they totally have no idea of how to solve problem with *if-else* and *repeat-do* constructs. The approachers and solvers should practise more on similar tasks to further improve their conception, and the former needs more practices than the latter. Instead of completing such easy tasks, the knowers should be assigned more complicated tasks. Secondly, the proposed clustering method could also be combined with *Hint Factory* (Stamper et al., 2008) to provide next-step hints for students in an intelligent tutoring system. One can develop a hint generation system that based on the similarity of both solution and trajectory. For example, when a student is stuck, the system would recommend him the next-step solution using the solution of students who have the most similar trajectory and must solved the problem perfectly. There is no doubt that this approach can reduce the tutoring burden of teachers in classroom.

Before concluding this work, we provide some more discussion regarding our work's limitations. First, the proposed approaching index is based on the abstract syntax tree (AST), which is available for computer programs but may not suitable for other subjects such as mathematics learning. Secondly, the computation of DTW similarity matrix and the TED of AST are time consuming especially when there are more than thousands of trajectories. Thirdly, the proposed approaching index is designed for the situation that there is only one optimal solution for each programing task. Lastly, since lack of students' demographic data and their performance on the related subjects, we can not conduct a deeper analysis to understand students' trajectories and investigate the relationship between the game performance and the exam score.

In our future work, we would attempt to use different similarity computation methods such as *pq-*Gram (Price et al., 2016; Zimmerman & Rupakheti, 2015) and code chunk method (Diana et al., 2018). As seen from our experimental results, there are significant differences in the performance of students in different clusters, so we would attempt to predict students' knowledge proficiency using approaching index. Lastly, it would be interesting to evaluate the proposed method on more programing learning tasks, especially more complicated programing tasks where the effectiveness of the proposed method could be further testified.

Notes

1. https://hourofcode.com
2. https://code.org/research
3. https://studio.code.org/hoc/18
4. https://github.com/timtadh/zhang-shasha
5. https://github.com/slaypni/fastdtw
6. http://scikit-learn.org/stable/index.html

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work is partly supported by the National Natural Science Foundation of China under Grant No. 61503340.

Notes on contributors

Bo Jiang is currently an associate professor with Department of Educational Information Technology, Zhejiang University of Technology, China. He received his PhD in Control Science and Engineering from Zhejiang University, Hangzhou, China, in 2014. His current research interests include educational data mining, learning analytics, and computational thinking education. He is an executive committee member of APSCE and serves as an editor board member of *International Journal of Bio-Inspired Computation* (2016–2018) and *Research and Practice in Technology Enhanced Learning* (2019–2021).

Wei Zhao and ***Nuan Zhang*** are currently master students in Department of Educational Information Technology, Zhejiang University of Technology, China.

Feiyue Qiu is currently a professor with Department of Educational Information Technology, Zhejiang University of Technology, China. His current research interests include game-based learning, VR/AR enhanced learning and adaptive learning.

ORCID

Bo Jiang http://orcid.org/0000-0002-7914-1978
Wei Zhao http://orcid.org/0000-0001-7709-383X

References

Akutsu, T. (2010). Tree edit distance problems: Algorithms and applications to bioinformatics. *IEICE Transactions on Information and Systems*, *93*(2), 208–218.

Bille, P. (2005). A survey on tree edit distance and related problems. *Theoretical Computer Science*, *337*(1–3), 217–239.

Blikstein, P (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. *Proceedings of the 1st International Conference on Learning Analytics and Knowledge* (pp. 110–116).

Blikstein, P., Worsley, M., Piech, C., Sahami, M., Cooper, S., & Koller, D. (2014). Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, *23*(4), 561–599.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Proceedings of the 2012 annual meeting of the American Educational Research Association (pp. 1–25). Vancouver, Canada.

Chen, X., Francia, B., Li, M., Mckinnon, B., & Seker, A. (2004). Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, *50*(7), 1545–1551.

David, B., Jeff, G., Caitlin, K., Josh, S., & Frankly, T. (2017). Learnable Programming: Blocks and Beyond. *Communications of the ACM*, *60*(6), 72–80.

Diana, N., Eagle, M., Stamper, J., Grover, S., Bienkowski, M., & Basu, S. (2018). Data-driven generation of rubric criteria from an educational programming environment. *Proceedings of the 8th International Conference on Learning Analytics and Knowledge* (pp. 16–20).

Diana, N., Eagle, M., Stamper, J., Org, J., Grover, S., Bienkowski, M., & Basu, S. (2017). An Instructor dashboard for real-time analytics in interactive programming assignments. *Proceedings of the seventh international learning analytics & knowledge*(pp. 272–279).

Fluri, B., Wuersch, M., PInzger, M., & Gall, H. (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, *33*(11).

Fraser, N (2015). Ten things we've learned from Blockly. *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE* (pp. 49–50).

Garcia, D., Harvey, B., & Barnes, T. (2015). The beauty and joy of computing. *ACM Inroads*, *6*(4), 71–79.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems?: An analysis of interaction traces. *Proceedings of the Ninth Annual International Conference on International Computing Education Research*Proceedings of the ninth annual international conference on international computing education research (pp. 119–126). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2361276.2361300

Hosseini, R., Vihavainen, A., & Brusilovsky, P. (2014). Exploring Problem Solving Paths in a Java Programming Course. *Psychology of Programming Interest Group Conference* (pp. 25–76). Pittsburgh.

Jadud, M. C (2006). Methods and tools for exploring novice compilation behaviour. *Proceedings of the second international workshop on Computing education research* (pp. 73–84).

Keogh, E. J., & Pazzani, M. J. (2001). Derivative dynamic time warping. *Proceedings of the 2001 SIAM International Conference on Data Mining* (pp. 1–11).

McBroom, J., Yacef, K., Koprinska, I., & Curran, J. R. (2018). A data-driven method for helping teachers improve feedback in computer programming automated tutors. *International Conference on Artificial Intelligence in Education* (324–337).

Mokbel, B., Gross, S., Paassen, B., Pinkwart, N., & Hammer, B. (2013). Domain-independent proximity measures in intelligent tutoring systems. *Proceedings of 6th International Conference on Educational Data Mining*.

Newell, A., & Simon, H. A. (1972). *Human Problem Solving. Vol. 104, No. 9*. Englewood Cliffs, NJ: Prentice-Hall.

Paaßen, B (2018). Revisiting the tree edit distance and its backtracing: A tutorial. arXiv preprint arXiv:1805.06869.

Paaßen, B., Hammer, B., Price, T. W., Barnes, T., Gross, S., & Pinkwart, N. (2018). The continuous hint factory -providing hints in vast and sparsely populated edit distance spaces. *Journal of Educational Data Mining*, *10*(1), 1–35.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New ideas in psychology*, *2* (2), 137–168.

Perkins, D., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. *First workshop on empirical studies of programmers on Empirical studies of programmers* (pp. 213–229).

Piech, C., Bassen, J., Huang, J., Ganguli, S., Sahami, M., Guibas, L. J., & J. Sohl Dickstein (2015). Deep knowledge tracing. In *Advances in Neural Information Processing Systems* (pp. 505–513).

Piech, C., Sahami, M., Huang, J., & Guibas, L. (2015). Autonomously Generating Hints by Inferring Problem Solving Policies. *Proceedings of the Second (2015) ACM Conference on Learning @ Scale - L@S '15* (pp. 195–204). New York, USA: ACM Press. Retrieved from http://dl.acm.org/citation.cfm?doid=2724660.2724668

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling How Students Learn to Program. *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 153–160). Retrieved from http://dl.acm.org/citation.cfm?doid=2157136.2157182

Powell, S., & Tod, J. (2004). *A systematic review of how theories explain learning behaviour in school contexts*. EPPI-Centre, Social Science Research Unit, Institute of Education, University of London. Retrieved from http://eppi.ioe.ac.uk/cms/Portals/0/PDF%20reviews%20and%20summaries/BM(CCC)_2004review.pdf?ver=2006-03-02-125203-580

Price, T. W., Dong, Y., & Barnes, T. (2016). Generating data-driven hints for open-ended programming. *Proceedings of 9th International Conference on Educational Data Mining* (pp. 191–198).

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. B. (2009). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67.

Rivers, K., & Koedinger, K. R. (2014). Automating hint generation with solution space path construction. *Proceedings of International Conference on Intelligent Tutoring Systems* (pp. 329–339). Retrieved from https://link.springer.com/content/pdf/10.1007%2F978-3-319-07221-0_ 41.pdf

Rivers, K., & Koedinger, K. R. (2017). Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, *27*(1), 37–64.

Stamper, J., Barnes, T., Lehmann, L., & Croy, M. (2008). The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track* (pp. 71–78).

Wang, L., Sy, A., Liu, L., & Piech, C. (2017a). Deep knowledge tracing on programming exercises. *Proceedings of the 4th ACM Conference on Learning@ Scale* (pp. 201–204).

Wang, L., Sy, A., Liu, L., & Piech, C. (2017b). Learning to represent student knowledge on programming exercises using deep learning. *Proceedings of the 10th International Conference on Educational Data Mining* (pp. 324–329).

Wing, J. (2014). Computational thinking benefits society. *40th Anniversary Blog of Social Issues in Computing*.

Zhang, K., & Shasha, D. (1989). Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, *18*(6), 1245–1262.

Zimmerman, K., & Rupakheti, C. R. (2015). An automated framework for recommending program elements to novices (n). *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (pp. 283–288).