PST: Measuring Skill Proficiency in Programming Exercise Process via Programming Skill Tracing

Ruixin Li
Anhui Province Key Laboratory of Big Data Analysis and Application, Institute of Advanced Technology, University of Science and Technology of China
State Key Laboratory of Cognitive Intelligence
rxli98@mail.ustc.edu.cn

Yu Yin
Anhui Province Key Laboratory of Big Data Analysis and Application, School of Computer Science and Technology, University of Science and Technology of China
State Key Laboratory of Cognitive Intelligence
yxonic@mail.ustc.edu.cn

Le Dai, Shuanghong Shen
Anhui Province Key Laboratory of Big Data Analysis and Application, School of Data Science, University of Science and Technology of China
State Key Laboratory of Cognitive Intelligence
{dl123,closer}@mail.ustc.edu.cn

Xin Lin
Anhui Province Key Laboratory of Big Data Analysis and Application, School of Computer Science and Technology, University of Science and Technology of China
State Key Laboratory of Cognitive Intelligence
linx@mail.ustc.edu.cn

Yu Su
School of Computer Science and Technology, Hefei Normal University
Institute of ArtificiaIntelligence, Hefei Comprehensive National Science Center
lyusu@iflytek.com

Enhong Chen*
Anhui Province Key Laboratory of Big Data Analysis and Application, School of Computer Science and Technology, University of Science and Technology of China
State Key Laboratory of Cognitive Intelligence
cheneh@ustc.edu.cn

ABSTRACT

Programming has become an important skill for individuals nowadays. For the demand to improve personal programming skill, tracking programming skill proficiency is getting more and more important. However, few researchers pay attention to measuring the programming skill of learners. Most of existing studies on learner capability portrait only made use of the exercise results, while the rich behavioral information contained in programming exercise process remain unused. Therefore, we propose a model that measures skill proficiency in programming exercise process named Programming Skill Tracing (PST). We designed Code Information Graph (CIG) to represent the feature of learners' solution code, and Code Tracing Graph (CTG) to measure the changes between the adjacent submissions. Furthermore, we divided programming skill into programming knowledge and coding ability to get more fine-grained assessment. Finally, we conducted various experiments to verify the effectiveness and interpretability of our PST model.

CCS CONCEPTS

• **Information systems** → **Data mining**; • **Computing methodologies** → **Artificial intelligence**.

KEYWORDS

Capability assessment; Intelligent education; Programming skill

ACM Reference Format:
Ruixin Li, Yu Yin, Le Dai, Shuanghong Shen, Xin Lin, Yu Su, and Enhong Chen*. 2022. PST: Measuring Skill Proficiency in Programming Exercise Process via Programming Skill Tracing. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '22), July 11–15, 2022, Madrid, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3477495.3531903

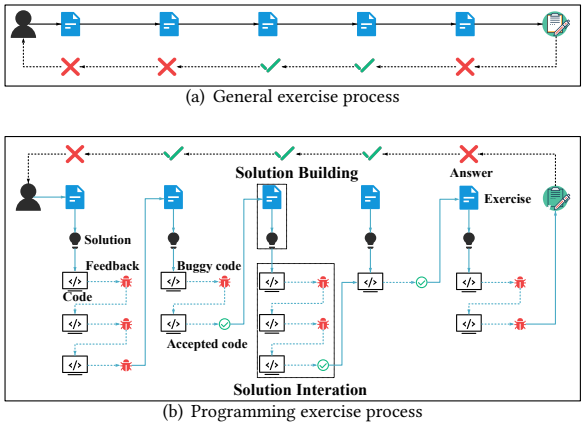
*Corresponding author.

1 INTRODUCTION

Nowadays programming has become an increasingly important skill for individuals, encouraging more and more people to take programming exercises and improve their programming skills. Along the exercise process, it is indeed important for learners to track the proficiency level of their programming skills.

Some related studies such as Cognitive Diagnosis [7, 10, 20, 26] and Knowledge Tracing [1, 17, 23] have put much effort in learner proficiency measurement during general exercise process. However, we may notice that Programming Exercise Process (PEP) is quite different from general exercise process. As demonstrated in Figure 1(a), in general exercise process previously studied, each learner answers exercise questions sequentially, with only one chance to answer every question. In contrast, PEP allows learners to solve the problem with an iteration process. Within each exercise, learners can submit their code, get feedbacks, and modify their solutions *iteratively*, as shown in Figure 1(b). Unlike general exercise (e.g. multiple choice exercises), learner's proficiency is not only reflected in whether they answer the exercise correctly, but also directly in their submitted code (showing their programming knowledge), as well as in the iteration process (showing their coding ability of solving problems). In fact, the whole process produces much richer behavioral data of each learner than general exercise process, which needs more careful consideration when measuring programming skill proficiency.

Considering the differences between Programming Exercise Process and general exercise process, acquiring accurate proficiency measurement in PEP are mainly faced with three challenges. First, a



(a) General exercise process

(b) Programming exercise process

Figure 1: The difference between general exercise process and programming exercise process.

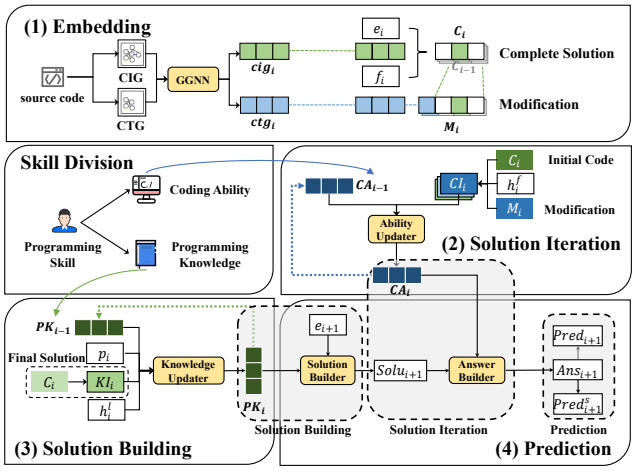


Figure 2: The overview of Programming Skill Tracing Model.

better representation of learner's submitted code is required to form a strong basis of understanding learner's behaviors. Existing code representation methods [2, 4, 8, 11, 13, 28] focus more on industry source code, and have limitations on representing learner's code. Second, learner's solution iteration process needs to be carefully studied and modeled, such as how solutions are initially built, how learners decide where to modify in their code. Third, as mentioned above, programming skill is not only related to learner's knowledge mastery, but also dependent on their ability of problems solving [3]. These two aspects should be more distinguished in programming skill measuring.

To this end, we propose the Programming Skill Tracing (PST) model for measuring programming skill proficiency in PEP. Concretely, we first propose the Code Information Graph (CIG) to represent the feature of learner's solutions of the programming exercise. CIG can meet the requirements on mining code structure and natural semantics, which meanwhile highlights the special feature of learner's solutions by adding the data flow edge and the error information edge. Then, we define the Code Tracing Graph (CTG) based on CIG to measure the changes between the adjacent submissions in a specific programming exercise process. Moreover, we divide programming skill into programming knowledge and coding ability to get more fine-grained assessment of programming skill in PST. Finally, we conduct extensive experiments to evaluate the effectiveness of PST, which indicate that PST has better performance than existing methods. Besides, by modeling students' programming exercise process, PST can measure students' programming skill more explainably.

2 RELATED WORK

Knowledge Tracing. The demand to dynamically trace knowledge mastery drives the development of knowledge tracing. Early probabilistic models[6, 14] like Bayesian Knowledge Tracing [6] assumed the learning process follows the Markov process, where the latent knowledge state of learners can be estimated by their observed learning performance. In recent years, deep learning-based model[17, 18, 22] like Deep Knowledge Tracing [18] achieved great performance with the help of neural network. The latter EKT [17] traces the relation among exercises and gets great performance. However, existing knowledge tracing models only consider the knowledge mastery, which can not completely meet the requirement to measure programming skill as mentioned above.

Programming Ability Measurement. Some works also pay attention to the programming learners. Early works on this field used the exam scores of previous courses or the learner log data to discover the potential struggled learners[5, 9, 21], but these methods paid little attention to the source code itself. To better model the ability of programming learners, Some works[15, 27] used knowledge tracing to measure the programming knowledge on the one-exercise scene. The latter works on programming knowledge tracing such as PDKT[29] began to consider the multi-exercise scene, and implemented a double-sequence modeling process to predict the next submission result. However, to the best of our knowledge, there is no work in this field to measure programming skill according to programming exercise process.

3 MODEL ARCHITECTURE

3.1 Problem Definition

In the online judge system, we record the learning sequence of a learner as a $LS_n = \{s_1, s_2, , s_3, ..., s_n\}$, where $s_i = (e_i, c_i, f_i)$ represents the solution submitted by the learner in the i step. Generally, when the code c_i meet the requirement of e_i i.e. get the Accepted (AC) result, f_i equals to 1, otherwise f_i equals to 0. To model the programming exercise process, we defined the Problem Exercising Process as the sequence of solutions submitted on the same exercise over a period of time. Then we can define our ***Programming Skill Proficiency Measurement*** problem as follows: given the historical programming exercise process log LS_n of each learner from programming exercise process step 1 to N, our goal is to measure the programming skill proficiency of learners.

3.2 Programming Skill Tracing

The overview of our designed PST model is shown in Figure 2. Specifically, we first transform all solution components and solution modification into embeddings. Then, PST utilizes the solution information to update the programming skill step by step. To focus on the key information of each solution, we utilize different

information in a programming exercise process to update the coding ability or programming knowledge. Then a specially designed Prediction module can utilize the programming knowledge and coding ability to make prediction. We will introduce the above two sub-module in the following subsections.

3.2.1 Embedding. In PST, we represent the exercise set, feedback set and position by three embedding matrices $E \in \mathbb{R}^{I \times d_e}$, $F \in \mathbb{R}^{J \times d_e}$ and $P \in \mathbb{R}^{K \times d_e}$. Then the exercise, the related feedback and position in the solution s_i will be represented as the vector $e_i \in E, f_i \in F, p_i \in P$. Moreover, we extract a graph named Code Information Graph (CIG) to represent the code c_i submitted by the learner in the solution s_i by highlighting the data flow relation and error information. Based on the CIG representation, we propose a Code Tracing Graph (CTG) to capture the changes in learners' solution iterations. Subsequently, we can use GGNN[16] to learn the solution embedding $cig_i \in \mathbb{R}^{d_e}$ for CIG and the solution change embedding ctg_i for CTG in the solution $s_i \in \mathbb{R}^{d_e}$ by a pre-training task to predict whether the source code can get Accepted result. We utilized $h_i^i, h_i^f \in \mathbb{R}^{d_e}$ to represent the position of a solution in PST, where h_i^i is set as the all-one vector if s_i is the initial solution, h_i^f is set as the all-one vector if s_i is the final solution, otherwise h_i^i, h_i^f will be all-zero vector.

Besides, the learners' modifications of their solutions in PEP can directly reflect their knowledge and ability. Therefore, we first utilized the triplet of the solution s_i to represent the complete solution information C_i, and utilized the previous solution s_{i-1} and the modification ctg_i in this solution s_i to represent the solution modification information M_i.

3.2.2 Solution Building. When conducting programming, a programming learner firstly build their solution to the exercise based on their programming knowledge. To measure the programming knowledge, we assume that the learner's solution does not change during a specific programming exercise process, and the submission made by learners gets closer to the solution. Therefore, we utilize the complete final solution information to update the programming knowledge. Specifically, we first fusion the complete solution information C_i to get the knowledge information KI_i. Considering learners with different programming knowledge and different solution iteration length may get different learning gain from same exercise, we design a learning gate Γ^{lk} to track different learning gain with different learners. Furthermore, we believe that learners forget some programming knowledge over time, so we design a forget gate Γ^{fk} to track the forget of programming knowledge. Then we utilize the Γ^{lk}, Γ^{fk}, PK_{i-1} and KI_i to update the programming knowledge PK_{i-1} as follows:

$$\begin{aligned} KI_i &= tanh(W_1 \cdot S_i + b_1) \\ \Gamma_i^{lk} &= \sigma(W_2 \cdot (KI_i \oplus PK_{i-1} \oplus p_i) + b_2) \\ \Gamma_i^{fk} &= \sigma(W_3 \cdot (KI_i \oplus PK_{i-1}) + b_3) \\ LG &= tanh(W_4 \cdot (KI_i \oplus PK_{i-1}) + b_4) \\ PK_i &= (1 - h_i^l) \cdot PK_{i-1} + h_i^l \cdot (\Gamma_i^{fk} \cdot PK_{i-1} + \Gamma_i^{lk} \cdot LG), \end{aligned} \tag{1}$$

where h_i^f is utilized to identify whether the solution is the final one of the exercise. After the learner build a solution to an exercise, they utilize their coding ability to write source code to meet the requirement of the exercise.

3.2.3 Solution Iteration. Usually the initial solution cannot meet the requirement of the exercise well. Therefore, after a learner submitted their source code, they receive feedback about this source code and decide whether to update their solution. So we focused on the modification of the source code made by the programming learner, i.e., the solution iteration.

According to the above descriptions, we update the coding ability by the two-fold information: initial solution and modification in the solution iteration. Specifically, we first fuse the initial solution information and modification in the solution iteration to get the coding information CI_i. We also design a learning gate Γ^{lc} and a forget gateΓ^{fc} to track learning gain and forget of different learners, we hope the coding ability can focus on the iteration process instead of solution position, so we don't utilize position information to build learning gate for coding ability. Then we can utilize the CI_i to update coding ability CA_{i-1} as follows:

$$\begin{aligned} CI_i &= tanh(W_5 \cdot (h_i^i \cdot S_i \oplus (1 - h_i^i) \cdot C_i) + b_5) \\ \Gamma_i^{lc} &= \sigma(W_6 \cdot (CI_i \oplus CA_{i-1}) + b_6) \\ \Gamma_i^{fc} &= \sigma(W_7 \cdot (CI_i \oplus CA_{i-1}) + b_7) \\ LG &= tanh(W_8 \cdot (CI_i \oplus CA_{i-1}) + b_8) \\ CA_i &= \Gamma_i^{fc} \cdot CA_{i-1} + \Gamma_i^{lc} \cdot LG, \end{aligned} \tag{2}$$

where h_i^i is utilized to identify whether the solution is the initial one of the exercise.

3.2.4 Performance Prediction. After the modeling process, we can utilize the programming skill to predict the next programming exercise process performance by modeling a complete programming exercise process. We first get the initial solution by the exercise requirement and programming knowledge, then we model the code writing to form the answer i.e. the source code by the solution and coding ability, finally we make prediction of the next programming exercise process performance and the score of next solution as follows:

$$\begin{aligned} solution_{i+1} &= tanh(W_9 \cdot (PK_i \oplus e_{i+1}) + b_9) \\ answer_{i+1} &= tanh(W_{10} \cdot (CA_i \oplus solution_{i+1}) + b_{10}) \\ pred_{i+1} &= tanh(W_{11} \cdot answer_{i+1} + b_{11}) \\ pred^s_{i+1} &= tanh(W_{12} \cdot answer_{i+1} + b_{12}), \end{aligned} \tag{3}$$

3.3 Objective Function

To learn all parameters in PST, we choose the cross-entropy log loss between the prediction *pred* and actual answer *a* for the binary criteria and the MSE loss between the prediction $pred^s$ and actual score *s* for the continuous criteria as the objective function. Then the loss of our model is defined as follows:

$$L = \alpha \cdot \sum_{i=1}^{M} (a_i \log pred_i + (1 - a_i) \log (1 - pred_i)) + \beta \cdot \sum_{j=1}^{N} |pred_j^s - s_j|, \tag{4}$$

where α, β are the hyper-parameters, and M, N are the number of PEP and solutions respectively.

Table 1: Data statistics of datasets

Statistics	AIZU_Cpp	Atcoder_C
# of learners	5268	6282
# of exercises	2206	1670
# of solutions	271189	425238
Avg. solution of learner	51.48	67.69
Avg. PEP of learner	30.89	36.30
Avg. length of PSP	1.66	2.00
Avg. score of learner	0.77	0.62
AC rate of final solution	74.30%	97.17%

4 EXPERIMENT

In this section, we first introduce the datasets we use, and then introduce the experimental setup and baseline models. Then we show the experimental results of the programming skill proficiency measurement. We provide four downstream tasks to evaluate the models, which will be described in detail in 4.2.1. Subsequently, we conduct visual experiment to verify the interpretability of PST. We consider using programming knowledge, coding ability and programming skill to represent learners in Atcoder_C respectively and implement dimensional reduction.

4.1 Experimental Dataset and Setup

4.1.1 Datasets. We use two real-world datasets to evaluate the effectiveness of our PST. The complete statisics of all the datasets is shown in Table 1. ***AIZU_Cpp*** is collected from the Project_CodeNet [19] dataset supplied by IBM. The benchmark is composed of learners who use CPP to practice exercises in AIZU.org, a popular online programming learning website. ***Atcoder_C*** is collected from online programming competition website Atcoder.org, composed of exercise practice with C languege. For all datasets, we split all the data to training set(60%), validation set(20%) and test set(20%).

4.1.2 Experimental Setup. In PST, we set the embedding dim d_e as 128. To set up training process, we initialize all network parameters with Xavier initialization, and set learning rate as 0.0001, mini batches as 128 and dropout[24] with 0.2. Hyper-parameter α and β to train our PST are both set to 0.5. Our code is available at https://github.com/rosen1998/PST.

4.1.3 Baseline Models. There are several methods to solve similar problems, so we choose various models as baselines. Firstly, to demonstrate the efficiency to extract the feature of learner's code, we compare our PST with the code representation model. Then, we compare our PST with the most recent coding ability measuring model to verify the effectiveness of utilizing the PEP. We also compare our PST model with several existing Knowledge Tracing (KT) models to prove the effectiveness of our efforts to split programming skill into knowledge and ability. The details of the above baselines are as follows:

- **codeBERT**[8] is a code representation learning module. To meet the requirement of tracking the programming skill. we utilize codeBERT to represent source code, and utilize the code embedding, exercise embedding and feedback embedding to model the full learning sequence of learners by the structure of LSTM[12].
- **PDKT**[29] utilizes the exercise information and the source code to implement double-sequence modeling process to track programming knowledge. To satisfy the requirement of our dataset, we replace the bipartite problem embedding in PDKT with the exercise embedding in our PST. We don't find publicly available PLcodeBERT, so we utilize original codeBERT instead of PLcodeBERT to get code embedding and fine-tune the code embedding via the pre-training task mentioned in 3.2.1.
- **DKT**[18] leverages RNN to assess knowledge mastery.
- **EERNN**[17] is one of most recent KT models which track knowledge mastery well by capturing the relation among exercises. We choose both EERNNM and EERNNA as baselines.

4.2 Experimental Results

4.2.1 Programming Skill Proficiency Measurement. We verify our PST model with the Programming Skill Proficiency Measurement task. We first train our PST via task 1 and fine-tune the model on other tasks. We describe these tasks as follows:

- **Task 1:** Predict whether the next programming exercise process will get a AC result.
- **Task 2:** Predict the average score of all solutions in next programming exercise process.
- **Task 3:** Predict the score of initial solution in next programming exercise process.
- **Task 4:** Predict the score of final solution in next programming exercise process.

Task 1 is utilized to evaluate the efficiency to measure programming skill of the models. Furthermore, in our hypothesis, the initial solution and the submission times of the PEP can be used to evaluate how the model fit the coding ability, so we choose the task 2 and task 3 to evaluate how the model track the coding ability. Furthermore, we believe that the final solution is consistent with the actual solution of learners, so we propose task 4 to evaluate how the model can fit the real solution of learners. For the first task, we represent the learner has got at least one AC result in the PEP as 1, otherwise as 0. Then the performance is evaluated in terms of *Accuracy* (ACC) and *Area Under Curve* (AUC). For the last three tasks, we select *Root Mean Square Error* (RMSE) to quantify the distance between predicted score and the actual one.

The experiment results are depicted in Table 2. There are several observations. First, compared to the codeBERT, our PST outperforms in Atcoder_C and the last three tasks in AIZU_Cpp, but the advantages of PST over codeBERT in AIZU_Cpp is little. The efficiency of CIG to represent learners' source code has been proved, but we guess the the pre-training process hinder CIG represent learners' source code well, so PST performs similarly in AIZU_Cpp compared to codeBERT. Second, PST gets much better results on last three tasks in AIZU_Cpp over both DKT and EERNN. Interestingly, contrary to the previous observation, knowledge tracing models can get closer results in Atcoder_C to PST. The fine-grained solution information indeed can help models to better represent coding ability of the programming learners especially novice ones, so PST can easily outperform over knowledge tracing models on AIZU_Cpp with easier exercises (with higher AC rate and with higher average

Table 2: Results of comparison methods on Programming Skill Proficiency Measurement.

Dataset	Atcoder_C					AIZU_Cpp				
Methods	Task 1		Task 2	Task 3	Task 4	Task 1		Task 2	Task 3	Task 4
	AUC	ACC	RMSE	RMSE	RMSE	AUC	ACC	RMSE	RMSE	RMSE
codeBERT	0.6679	0.8001	0.3017	0.3592	0.3005	0.5054	**0.9600**	0.2309	0.3165	0.1765
PDKT	0.6066	0.7556	0.3494	0.3970	0.3569	0.5078	0.9184	0.3054	0.3773	0.2662
DKT	0.6955	0.8100	0.2962	0.3576	0.2904	0.5119	0.9588	0.2714	0.3689	0.1918
EERNNA	0.7106	0.8040	0.2998	0.3597	0.2954	0.5022	0.9598	0.2535	0.3519	0.1765
EERNNM	0.7138	0.8085	0.2952	0.3556	0.2908	0.5258	0.9592	0.2501	0.3475	0.1746
PST	**0.7159**	**0.8107**	**0.2875**	**0.3453**	**0.2862**	**0.5281**	0.9596	**0.2239**	**0.3073**	**0.1731**

score). The division of programming skill into coding ability and programming knowledge is efficient. Third, compared to PDKT without considering the PEP, our PST outperforms on all tasks in all datasets, which uncover the efficiency of the PEP modeling process. We also note that EERNNM outperforms than EERNNA on both tasks, we guess the similar exercise text hinder the attention mechanism of EERNNA from capturing the relation between two exercises. Furthermore, most methods perform similarly on the task 1 in AIZU_Cpp, we guess the high AC rate as mentioned in Table 1 hinder all the models measure programming skill well.

4.2.2 The Effectiveness of Skill Division. We divide the programming skill to coding ability and programming knowledge, the former one focuses on the initial solution building and the latter one focuses on the solution iteration process. To demonstrate the efficiency of the division, we conduct a learner visualization via t-SNE[25] dimensional reduction. We first divide the learners into three groups with different average scores and with different average solution iteration lengths respectively. As shown in Figure 3, we present the dimensional reduction results of PK_f embedding in the first row, results of CA_f embedding in the second row and results of PS_f (i.e. the concatenation of PK_f and CA_f) in the third row, where PK_f, CA_f is the programming knowledge and coding ability at final time step generated from PST respectively. The figures in the first column show the visualization results of the average score of learners, and the figures in the second column shows the visualization results of the average solution iteration length of learners.

We note some observations in the experimental results. First, programming knowledge embedding well distinguish learners with different average scores, while it cannot distinguishes learners with different solution iteration lengths. Second, coding ability embedding provides clear distinction for learners with different average iteration lengths, while the discrimination of coding ability to learners with different average scores is low. With advantages of both programming knowledge and coding ability, programming skill can distinguish both the two characteristic well. It can demonstrate that the division of programming skill into coding ability and programming knowledge is effective.

5 CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel Programming Skill Tracing (PST) model to measure the programming skill of learners in the programming exercise process, taking consideration of rich behavioral information. Specific-designed Code Information Graph (CIG) was utilized to represent the feature of learners' solution code.

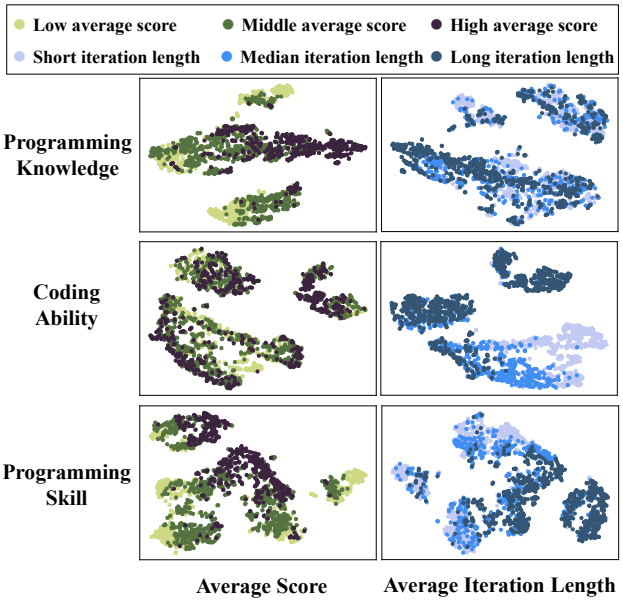


Figure 3: Learner visualization in Atcoder_C.

Moreover, based on CIG, Code Tracing Graph (CTG) was utilized to measure the changes between the adjacent submissions. With CIG and CTG, we managed to measure programming skill by modeling programming knowledge and coding ability respectively and get more fine-grained assessment. Various experiments demonstrated that our PST can measure programming skill proficiency with both high accuracy and high interpretability.

To simplify the PEP modeling process, we assume that the learner's solution does not change during a specific programming exercise process. However, several programming learners will change their solution to better satisfy the exercise requirement, so we will focus the solution changes via source code in the future. We will also consider making tree pruning to better extract learner's code feature. Furthermore, due to dataset limitations, we only consider programming knowledge as one-dimentional knowledge concept. In the future, we will extend our PST to measure multi-dimentional programming knowledge and related coding ability.

ACKNOWLEDGE

This research was partially supported by grants from the National Key Research and Development Program of China (Grant No. 2021YF F0901003), and the National Natural Science Foundation of China (Grant No. U20A20229)

REFERENCES

[1] Ghodai Abdelrahman and Qing Wang. 2019. Knowledge tracing with sequential key-value memory networks. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 175–184.

[2] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336* (2018).

[3] Marc Berges, Andreas Mühling, and Peter Hubwieser. 2012. The gap between knowledge and ability. In *Proceedings of the 12th Koli Calling international conference on computing education research*. 126–134.

[4] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[5] Karo Castro-Wunsch, Alireza Ahadi, and Andrew Petersen. 2017. Evaluating neural networks as a method for identifying students in need of assistance. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 111–116.

[6] Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction* 4, 4 (1994), 253–278.

[7] Susan E Embretson and Steven P Reise. 2013. *Item response theory*. Psychology Press.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Ge Gao, Samiha Marwan, and Thomas W Price. 2021. Early Performance Prediction using Interpretable Patterns in Programming Process Data. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 342–348.

[10] Weibo Gao, Qi Liu, Zhenya Huang, Yu Yin, Haoyang Bi, Mu-Chun Wang, Jianhui Ma, Shijin Wang, and Yu Su. 2021. Rcd: Relation map driven cognitive diagnosis for intelligent education systems. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 501–510.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[13] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[14] Tanja Käser, Severin Klingler, Alexander G Schwing, and Markus Gross. 2017. Dynamic Bayesian networks for student modeling. *IEEE Transactions on Learning Technologies* 10, 4 (2017), 450–462.

[15] Jussi Kasurinen and Uolevi Nikula. 2009. Estimating programming knowledge with Bayesian knowledge tracing. *ACM SIGCSE Bulletin* 41, 3 (2009), 313–317.

[16] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[17] Qi Liu, Zhenya Huang, Yu Yin, Enhong Chen, Hui Xiong, Yu Su, and Guoping Hu. 2019. Ekt: Exercise-aware knowledge tracing for student performance prediction. *IEEE Transactions on Knowledge and Data Engineering* 33, 1 (2019), 100–115.

[18] Chris Piech, Jonathan Spencer, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. *arXiv preprint arXiv:1506.05908* (2015).

[19] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *arXiv preprint arXiv:2105.12655* (2021).

[20] Mark D Reckase. 2009. Multidimensional item response theory models. In *Multidimensional item response theory*. Springer, 79–112.

[21] Cristóbal Romero, Sebastián Ventura, Pedro G Espejo, and César Hervás. 2008. Data mining algorithms to classify students. In *Educational data mining 2008*.

[22] Shuanghong Shen, Qi Liu, Enhong Chen, Zhenya Huang, Wei Huang, Yu Yin, Yu Su, and Shijin Wang. 2021. Learning Process-consistent Knowledge Tracing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1452–1460.

[23] Shuanghong Shen, Qi Liu, Enhong Chen, Han Wu, Zhenya Huang, Weihao Zhao, Yu Su, Haiping Ma, and Shijin Wang. 2020. Convolutional knowledge tracing: Modeling individualization in student learning process. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1857–1860.

[24] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[25] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[26] Fei Wang, Qi Liu, Enhong Chen, Zhenya Huang, Yuying Chen, Yu Yin, Zai Huang, and Shijin Wang. 2020. Neural cognitive diagnosis for intelligent education systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 6153–6161.

[27] Lisa Wang, Angela Sy, Larry Liu, and Chris Piech. 2017. Deep knowledge tracing on programming exercises. In *Proceedings of the fourth (2017) ACM conference on learning@ scale*. 201–204.

[28] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[29] Renyu Zhu, Dongxiang Zhang, Chengcheng Han, Ming Gao, Xuesong Lu, Weining Qian, and Aoying Zhou. 2021. Programming Knowledge Tracing: A Comprehensive Dataset and A New Model. *arXiv preprint arXiv:2112.08273* (2021).