Multigraph-based Deep Programming Ability Tracing Method For Students

Yue Wang[1] and Guanwen Zhang[2*]

[1] Faculty of education, Shandong Normal University, Jinan, Shandong, China 250014
[2] School of journalism and communication, Shandong Normal University, Jinan, Shandon, China 250014
[*] Corresponding author. E-mail: sdzhaw@163.com

Received: Oct. 09, 2024; Accepted: Oct. 23, 2024

Programming has become a crucial ability with the development of artificial intelligence, making it increasingly important to track and improve programming proficiency of students. However, most programming ability tracing methods rely primarily on the final outcomes of programming exercises to measure the programming proficiency, and neglect the rich behavioral structure information derived from the programming process, leading to a suboptimal solution in the programming ability tracing. To this end, we propose a multigraph-based deep programming ability tracing method for students (MDPAT), which consists of four components. Specifically, MDPAT devises the multigraph programming modelling via conducting the program knowledge graph and the program iteration graph to generate the submission representations of programming exercises, which effectively captures static structure information and dynamic structure information within the programming process. Then, MDPAT designs the programming knowledge gated update and the programming ability gated update to aggregate information of the programming knowledge and the programming ability that are derived from submission representations, which achieves a balanced and comprehensive tracking of evolving programming proficiency. Meanwhile, MDPAT utilizes the programming answer prediction to generate final programming proficiency measurement of students. Four components of MDPAT collaborate organically to achieve maximum performance in the programming ability tracing. Finally, extensive results on two real world datasets, especially on the Atcoder_C dataset, ACC shows a 5.06% improvement compared to the second best result, verify that MDPAT conducts a new standard baseline in the programming ability tracing.

Keywords: Programming ability tracing; multigraph programming modelling; gated update learning
©The Author('s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are cited.

http://dx.doi.org/10.6180/jase.202508_28(8).0019

1. Introduction

Knowledge tracing (KT) is an important research area focused on analyzing student knowledge states, and has recently gained attention from educators and researchers [1, 2]. KT captures changes in students' knowledge states through their historical answering records, allowing for the prediction of future problem-solving performance. It has been widely applied in various educational contexts, such as predicting career choices for students, forecasting potential errors in second language learning, and providing course recommendations for university students. Traditional knowledge tracing models include item response Theory, bayesian knowledge tracing, and knowledge tracing based on factor analysis [3, 4]. While these models possess good interpretability, they rely on theoretical assumptions and require manual construction of input features. This often leads to limitations and biases, resulting in generally subpar predictive performance. In contrast, deep learning-based knowledge tracing models, such as Deep Knowledge Tracing, have shown significant improvements in predictive accuracy, utilizing advanced techniques such

as Recurrent Neural Networks [5], Memory Networks [6], Self-Attention Mechanisms [7], and Deep Neural Networks [8].

Although knowledge tracing has been widely applied in traditional course learning, programming learning possesses unique characteristics, primarily reflected in the following three aspects: Most programming problems do not have a single correct answer and often assess multiple knowledge points simultaneously, allowing students to provide open-ended solutions based on their learned knowledge[9–11]. The source code submitted by students contains semantic and syntactic information of the programming language, reflecting their mastery of the programming language and the relevant data structures and algorithms. For programming exercises, students typically need to modify and resubmit their code multiple times before successfully completing the problem [12–14].

With the increasing popularity of programming education, research efforts aimed at programming knowledge tracing have gradually increased. Wang et al. were the first to combine deep knowledge tracing models with student programming, using a recurrent neural network to analyze students' multiple attempts on the same exercise [9]. Although this model can capture students' knowledge states, it only considers code features and fails to utilize related features such as problems and knowledge points, overlooking the relationship between code and problems. Swamy et al. improved the code representation method based on Wang et al.'s work, but this approach requires building separate models for each knowledge point to analyze students' mastery of specific topics [10]. Jiang et al. proposed modeling students' multiple attempts on the same exercise by analyzing the abstract syntax tree edit distance sequences between students' submitted answers and the standard answers. However, this approach assumes that each problem has a corresponding standard answer [11].

Despite the progress made in programming skill tracing techniques, three major challenges remain: (1) Accurate representation of student-generated code: Existing code representation techniques are primarily designed for industrial-level code, lacking the adaptability required to capture the unique patterns in student submissions. More precise representations are essential to accurately model student programming behaviors and foundational understanding. (2) Comprehensive modeling of the iterative coding process: Students often engage in iterative problem-solving, refining their solutions based on feedback. Effective modeling of this iterative process, from initial solution formulation to subsequent adjustments, is crucial to understanding the evolution of programming proficiency. (3) Granular assessment of programming ability: Programming assessments must go beyond knowledge recall and examine how students apply this knowledge to solve problems. A clear distinction between conceptual understanding and practical application is necessary to provide a more comprehensive and detailed evaluation of programming skills.

To this end, we propose a multigraph-based deep programming ability tracing method for vocational college students (MDPAT), which integrates multigraph programming modeling, gated updates for programming knowledge and ability, and programming answer prediction. MDPAT utilizes multigraph programming modeling to capture both static and dynamic structural information in the programming process, effectively identifying key relationships and patterns in the code, such as structural dependencies between code elements and the evolving changes in students' programming behavior across submissions. By incorporating gated update mechanisms for both programming knowledge and ability, MDPAT selectively aggregates relevant information from submissions, ensuring efficient integration of new knowledge while preserving important prior learning. This allows MDPAT to track programming proficiency comprehensively. Finally, in the programming answer prediction stage, MDPAT unifies programming knowledge and ability into a single predictive architecture, which considers both a student's knowledge mastery and their problem-solving ability, leading to improved prediction accuracy.

MDPAT makes several notable contributions, including:

- MDPAT models dual representation of both static and dynamic code features via multigraph learning, to capture the structural and temporal evolution of code, which enables a richer understanding of student learning patterns and leads to more precise assessments.
- MDPAT designs gated update mechanism to adaptively filter and update the knowledge state, selectively incorporating new submissions while maintaining continuity with prior knowledge. This dynamic retention of salient historical context supports more robust tracking of skill progression.
- Extensive results on two real world datasets, verify that MDPAT conducts a new standard baseline in the programming ability tracing.

2. Multigraph-based deep programming ability tracing method for students

Let the sequence of submissions about programming exercise by a student be denoted as $R = \{r_1, r_2, \ldots, r_n\}$, where

each submission $r_i = (e, c_i, y_i)$ represents a student's attempt c_i on exercise e at the i-th time. If the submission meets the exercise criteria, y_i is set to 1; otherwise, it is set to 0. The goal of the programming ability tracing is to evaluate the programming proficiency of students according to historical programming exercise records. To this end, we propose a multigraph-based deep programming ability tracing method for students in high vocational colleges, which consists of the multigraph programming modelling, programming knowledge gated update, programming ability gated update and programming answer prediction, as shown in Fig. 1.

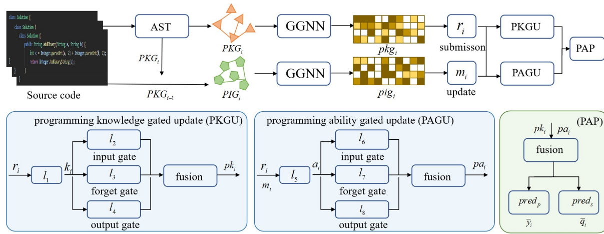


Fig. 1. The illustration of MDPAT. MDPAT consists of the multigraph programming modelling, programming knowledge gated update, programming ability gated update and programming answer prediction

2.1. Multigraph programming modelling

Inspired by prior works [13, 14], we introduce the program knowledge graph and the program iteration graph to capture and represent code features and changes during programming practice. Specifically, PKG is designed to represent the static features of the code submitted by students. It begins with the construction of an Abstract Syntax Tree from the source code c_i, which provides a hierarchical structure that adheres to the syntax of the programming language. Each node in the PKG corresponds to a node in the AST, and edges represent the relationships between these nodes. The graph goes beyond the structure edges of the AST by introducing additional edges that represent data flow and error information, thus highlighting the unique characteristics of the student code. PIG is designed to capture the dynamic evolution of the code as students iterate their codes. It considers the sequence of codes submitted by students over time. The model analyzes the differences between consecutive submissions to identify changes such as additions, deletions, or modifications in the code. These changes are then represented as edges in the PIG, connecting nodes that represent successive submissions. The edges in the PIG are defined based on the presence of changes between consecutive code submissions:

$$E_{\text{PIG}} = \{(v_i, v_j) \mid \Delta_{i,j} \neq \emptyset\} \tag{1}$$

where $\Delta_{i,j}$ represents the differences identified by the function $\text{diff}(r_t, r_{t+1})$, which compares two submissions r_t and r_{t+1}. v_i and v_j denote nodes in PKG.

Graph-based representations are essential for understanding the structure and semantics of codes. Since the Gated Graph Neural Network (GGNN) can capture complex dependencies and relationships within graph-structured data and is suitable for code analysis tasks, we utilize GGNN to encode both local and global contextual information about the entire graph, for effectively extracting features from the program knowledge graph and the program iteration graph. Let PKG_i represent the program knowledge graph for code c_i, and PIG_i represent the program iteration graph. We use GGNN to extract representations from PKG_i and PIG_i, denoted as pkg_i and pig_i, respectively:

$$\begin{aligned} pkg_i &= \text{GGNN}(PKG_i) \\ pig_i &= \text{GGNN}(PIG_i) \end{aligned} \tag{2}$$

Then, the triplet (e, pkg_i, y_i) is viewed as the submission r_i at the i-th time. To represent the updated information for a submission, we consider the previous submission complete representation r_{i-1}, along with the modification representations pig_i for the current submission r_i. The updated information is denoted as m_i:

$$m_i = fusion\,(r_{i-1}, pig_i) \tag{3}$$

where $fusion()$ denotes information aggregation function. The multigraph programming model introduces two key graph representations to capture and analyze code features and iterations. PKG represents the static structure of student code by building upon the Abstract Syntax Tree and adding edges for data flow and error information, while PIG captures the dynamic evolution of code through sequential submissions, with edges representing changes between versions. GGNN are used to encode both PKG and PIG, extracting meaningful representations of static and dynamic features. These representations are then combined to model the evolution of code, allowing the updated submission information to reflect both past and present changes, thus enabling comprehensive analysis of student code development over time.

2.2. Programming knowledge gated update

During the programming process, students construct their submissions to exercises by drawing upon their accumulated programming knowledge. Consequently, we leverage

the submission information to effectively refine and update programming knowledge.

Specifically, the update of the programming knowledge pk_i follows an LSTM-inspired structure, where the programming knowledge is progressively updated based on the current submission r_i and the previous programming knowledge pk_{i-1}. To achieve this, we first derive the new knowledge k_i from the current submission r_i using a nonlinear layer l_1, i.e., $k_i = l_1(r_i)$. Next, we calculate the input gate I_i^{pk}, which determines how much of the new knowledge k_i should be incorporated. This is done by concatenating k_i, the previous programming knowledge pk_{i-1}, and position representations p_i, and then applying a nonlinear layer l_2:

$$I_i^{pk} = l_2([k_i, pk_{i-1}, p_i]) \tag{4}$$

Similarly, the forget gate F_i^{pk} is computed to control how much of the previous programming knowledge pk_{i-1} should be retained:

$$F_i^{pk} = l_3([k_i, pk_{i-1}]) \tag{5}$$

where l_3 is another nonlinear layer. Lastly, the output gate O_i^{pk} is calculated to determine how much information is passed to the next state:

$$O_i^{pk} = l_4([k_i, pk_{i-1}]) \tag{6}$$

Then, the cell state $\widetilde{pk_i}$ is updated by combining the forget gate with the previous knowledge and the input gate with the new knowledge:

$$\widetilde{pk_i} = F_i^{pk} \cdot pk_{i-1} + I_i^{pk} \cdot k_i \tag{7}$$

Finally, the programming knowledge pk_i is updated using the output gate, which integrates the current cell state and regulates the final output:

$$pk_i = O_i^{pk} \cdot \tanh(\widetilde{pk_i}) \tag{8}$$

This structure allows the model to selectively integrate new information from the current code submission while retaining relevant prior programming knowledge, similar to the gating mechanism of LSTM cells.

2.3. Programming ability gated update

In practice, initial submission often fails to meet the exercise requirements. After submitting programming code, students receive feedback to help them decide whether or not to modify their submissions. Therefore, we focus on the iterative refinement of source code submissions, referred to as the programming ability.

The update of the programming ability pa_i is structured similarly to an LSTM mechanism, where the programming ability is iteratively updated based on the current submission r_i and the previous programming ability pa_{i-1}. The calculation of the new knowledge a_i is first performed by concatenating the current submission r_i and updated information m_i, followed by applying a nonlinear layer l_5, i.e., $a_i = l_5(r_i \oplus m_i)$. Next, the input gate I_i^{pa} determines the extent to which information from the new submission should be integrated into the programming ability:

$$I_i^{pa} = l_6([a_i, pa_{i-1}]) \tag{9}$$

The forget gate F_i^{pa} regulates the retention of previous programming ability:

$$F_i^{pa} = l_7([a_i, pa_{i-1}]) \tag{10}$$

and the output gate O_i^{pa} dictates how much of the updated programming ability is carried forward to the next iteration:

$$O_i^{pa} = l_8([a_i, pa_{i-1}]) \tag{11}$$

The input gate controls the integration of new information, the forget gate manages the retention of previous abilities, and the output gate facilitates the passage of updated programming ability to the subsequent iteration. Next, the cell state $\widetilde{pa_i}$ is updated by combining the contributions from the forget gate and the previous programming ability with the input gate:

$$\widetilde{pa_i} = I_i^{pa} \cdot pa_{i-1} + F_i^{pa} \cdot a_i \tag{12}$$

Finally, the programming ability ca_i is computed using the output gate, which integrates the current cell state and determines the final output:

$$pa_i = O_i^{pa} \cdot \tanh(\widetilde{pa_i}) \tag{13}$$

This architecture enables the model to effectively incorporate new information from the current solution while preserving relevant aspects of the prior programming ability, mirroring the gating mechanisms found in LSTM cells.

2.4. Programming answer prediction

After fitting programming knowledge and ability of students, a programming answer prediction network is designed via fusing programming knowledge and programming ability into a unified programming exercise architecture.

Initially, we derive the initial submission for the next exercise using the exercise requirement e_{i+1} and the current programming knowledge pk_i, i.e., $s_{i+1} = l_9([pk_i, e_{i+1}])$. Next, we model programming exercise to form the answer a_{i+1}, which represents the source code, by utilizing the derived submission s_{i+1} and the programming ability pa_i,

Table 1. MDPAT Algorithm Flow

Input: Student sequence of submissions $R = \{r_1, r_2, \ldots, r_n\}$
Output: Programming proficiency measurement $\bar{y}_i$ and $\bar{q}_i$
1. For each submission $r_i = (e, c_i, y_i)$:
2. Construct program knowledge graph and program iteration graph.
3. Extract features using GGNN: $pkg_i \leftarrow GGNN(PKG_i)$, $pigi \leftarrow GGNN(PIG_i)$.
4. Update submission information: $r_i = (e, pkg_i, y_i)$.
5. Update modification information: $m_i \leftarrow (r_{i-1}, pig_i)$.
5. Extract programming knowledge pk_i using gated mechanism.
6. Extract programming ability pa_i using gated mechanism.
7. Predict the answer a_i using current knowledge and ability.
8. Calculate final prediction $\bar{y}_i$ and $\bar{q}_i$.
9. Calculate the loss L and optimize with SGD.
10. End.
11 Return: The model parameters and $\bar{y}_i$ and $\bar{q}_i$, $i = 1, 2, .., n$.

i.e., $a_{i+1} = l_{10}([pa_i, s_{i+1}])$. Finally, we make predictions for both the performance in the next programming exercise process and the score of the next solution using the following equations:

$$\bar{y}_{i+1} = pred_p(a_{i+1}) \\ \bar{q}_{i+1} = pred_s(a_{i+1}) \tag{14}$$

where $pred_p()$ and $pred_s()$ denote prediction networks. The cross entropy loss [15] and the MSE loss [16] are used to optimize MDPAT for achieving programming ability tracing of students within the stochastic gradient descent optimization framework::

$$L = \lambda_1 CE(\bar{y}, y) + \lambda_2 (MSE(\bar{q}, q)) \tag{15}$$

where λ_1 and λ_2 are the hyper-parameters for weighting the cross entropy loss and the MSE loss. y and q denotes programming exercise process performance and the score of programming exercise, respectively. The detailed algorithm process is shown in the Table 1.

3. Results and discussion

3.1. Dataset and Setup

The performance of MDPAT is evaluated via two datasets. AIZU_Cpp dataset [3] consists of programming exercises completed by 5,268 unique students on the AIZU.org platform in C++. Collectively, these students attempted 2,206 different exercises, submitting a total of 271,189 solutions. On average, each student made about 51.48 submissions, with an average score of 0.77, closely aligning with the final submission acceptance rate of 74.30%. Atcoder_C dataset [4] contains data from 6,282 students programming in C on Atcoder.org. These students worked on 1,670 exercises, generating a total of 425,238 submissions, with an average of 67.69 submissions per student. The average student score is 0.62, reflecting a 62% success rate on submissions, while the acceptance rate of final submissions is notably higher at 97.17%. Following [17], to evaluate MDPAT performance, three metrics are used: AUC, ACC, and f1.

3.2. Comparison with baselines

Comparison baselines: Six deep programming ability tracing methods are compared on the AIZU_Cpp dataset and Atcoder_C dataset in terms of AUC, ACC, and f1, to demonstrate the performance of MDPAT, containing Deep knowledge tracing (DKT), graph knowledge tracing (GKT), convolutional knowledge tracing (CKT), stable knowledge tracing with diagnostic transformer (SKTDT), enhancing deep knowledge tracing (EDKT), and a pre-trained model for programming and natural languages (Codebert). DKT models interaction information between students and coursework within the RNN architecture to achieve the knowledge tracing. GKT utilizes the bipartite graph to learn programming representations and achieves the knowledge tracing within dual deep architecture. CKT devises individualized learning and common learning for student programming within the convolutional architecture to achieve the knowledge tracing. SKTDT constructs a dual transform architecture for practice and knowledge, capturing student programming knowledge status from multiple dimensions. EDKT utilizes auxiliary tasks to boost programming knowledge tracing within the deep architecture.

Comparison results: As shown in Table 2, MDPAT outperforms six other deep programming ability tracking methods on both the AIZU_Cpp and Atcoder_C datasets. Specifically, MDPAT achieves the best performance in terms of ACC (accuracy), AUC (area under the curve), and F1 (a performance metric combining precision and recall). Compared to the second-best results, ACC, AUC, and F1 scores

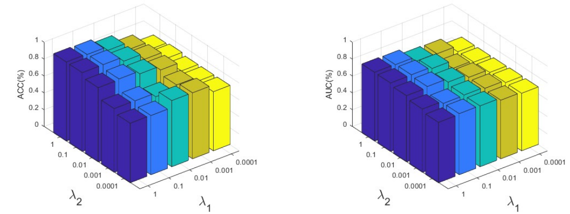
Table 2. Comparison of MDPAT on the AIZU_Cpp and Atcoder_C datasets

Method	AIZU_Cpp			Atcoder_C		
	ACC	AUC	f1	ACC	AUC	f1
DKT [18]	0.8100	0.6785	0.8326	0.8312	0.7019	0.8569
GKT [7]	0.8333	0.6912	0.8562	0.8674	0.7285	0.8766
CKT [8]	0.7946	0.6626	0.8014	0.8155	0.6814	0.8255
SKTDT [2]	0.8946	0.7811	0.8635	0.9095	0.8566	0.9592
EDKT [3]	0.8774	0.7778	0.8551	0.8799	0.8496	0.9367
Codebert [12]	0.8566	0.7000	0.7933	0.8458	0.8012	0.8962
MDPAT	**0.9390**	**0.8116**	**0.8794**	**0.9601**	**0.8871**	**0.9796**

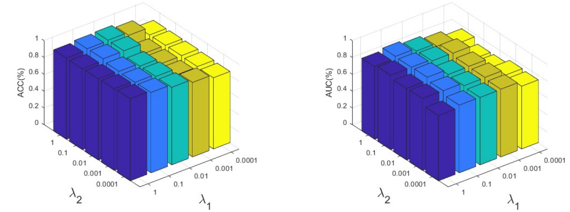
of MDPAT on the AIZU_Cpp dataset are higher by 0.0454, 0.0305, and 0.0169, respectively. On the Atcoder_C dataset, MDPAT surpasses by 0.0359, 0.0225, and 0.0204 in ACC, AUC, and F1, respectively. This performance advantage can be attributed to three critical mechanisms: (1) MDPAT's dual representation of both static and dynamic code features through the Programming Knowledge Graph and Programming Iteration Graph facilitates a holistic view of the underlying programming behaviors. By modeling the structural and temporal evolution of code, MDPAT enables a richer understanding of student learning patterns, leading to more precise assessments. (2) The gated update mechanism, inspired by Long Short-Term Memory (LSTM) networks, enables MDPAT to adaptively filter and update the knowledge state, selectively incorporating new submissions while maintaining continuity with prior knowledge. This dynamic retention of salient historical context supports more robust tracking of skill progression. (3) In the final prediction stage, MDPAT's integrated framework harmonizes the representation of both programming knowledge and ability, creating a unified architecture that simultaneously accounts for conceptual mastery and practical problem-solving skills. This synergistic approach significantly enhances predictive accuracy by capturing both the theoretical understanding and applied proficiency of the learner.

3.3. Parameter Analysis

We analyze the effect of varying λ_1 and λ_2, which are used to balance the cross entropy loss and the MSE loss, on MDPAT performance across the AIZU_Cpp and Atcoder_C datasets, using ACC and AUC as metrics. The results show a clear trend: as λ_1 and λ_2 increase within the range $[0.0001, 1]$, performance improves, peaking when both parameters are in the interval $[0.1, 1]$. In this range, the model achieves near-optimal results for both ACC and AUC. Performance is particularly strong when λ_1 and λ_2 are around 0.5, where a good balance between learning new knowledge and retaining past information is maintained. However, as the values of λ_1 and λ_2 drop below 0.01, performance declines, indicating that small parameter values lead to excessive retention of prior knowledge or difficulty integrating new information. Thus, the optimal range for λ_1 and λ_2 is $[0.1, 1]$, where model stability and accuracy are maximized. Based on this, we set $\lambda_1 = 0.5$ and $\lambda_2 = 0.5$ to maintain an ideal balance between knowledge integration and retention while avoiding overfitting or underfitting.



(a) AIZU_Cpp-ACC(λ_1 and λ_2) **(b)** AIZU_Cpp-AUC(λ_1 and λ_2)



(c) Atcoder_C-ACC(λ_1 and λ_2) **(d)** Atcoder_C-AUC(λ_1 and λ_2)

Fig. 2. Parameter analysis of λ_1 and λ_2 on the AIZU_Cpp and Atcoder_C datasets

3.4. Ablation Analysis

We conduct ablation Analysis to verify component effectiveness in MDPAT. There are three ablation variants. MDPAT_1 utilizes multigraph programming modelling and programming answer prediction to achieve the programming ability tracing. MDPAT_2 utilizes multigraph programming modelling, programming knowledge gated update, and programming answer prediction to achieve the programming ability tracing. MDPAT_2 utilizes multigraph programming modelling, programming ability gated

Table 3. Ablation Analysis of MDPAT on the AIZU_Cpp and Atcoder_C datasets

	AIZU_Cpp			Atcoder_C		
Method	ACC	AUC	f1	ACC	AUC	f1
MDPAT_1	0.8400	0.6799	0.8111	0.8685	0.7991	0.8956
MDPAT_2	0.8833	0.7854	0.8426	0.8847	0.8308	0.9006
MDPAT_3	0.8545	0.7712	0.8033	0.8678	0.8077	0.8888
MDPAT	**0.9390**	**0.8116**	**0.8794**	**0.9601**	**0.8871**	**0.9796**

update, and programming answer prediction to achieve the programming ability tracing.

As shown in Table 3, the complete MDPAT model significantly outperforms all ablation variants across both the AIZU_Cpp and Atcoder_C datasets, particularly in terms of ACC, AUC, and f1 score. This indicates that the inclusion of all key components—multigraph programming modeling, the gated update mechanisms for both programming knowledge and ability, and programming answer prediction—contributes to the overall superior performance of the model. For instance, the full MDPAT model achieves an ACC of 0.9390 and an AUC of 0.8116 on the AIZU_Cpp dataset, surpassing MDPAT_2 and MDPAT_3, which lack the full gated update mechanism, by significant margins. Similarly, on the Atcoder_C dataset, the full MDPAT reaches an ACC of 0.9601 and an AUC of 0.8871, demonstrating its robustness across different datasets. These results highlight the critical role of each component, particularly the gated update mechanisms, in enabling the model to effectively capture and trace students' programming knowledge and ability over time, leading to more accurate and reliable predictions.

3.5. Dimensionality Analysis

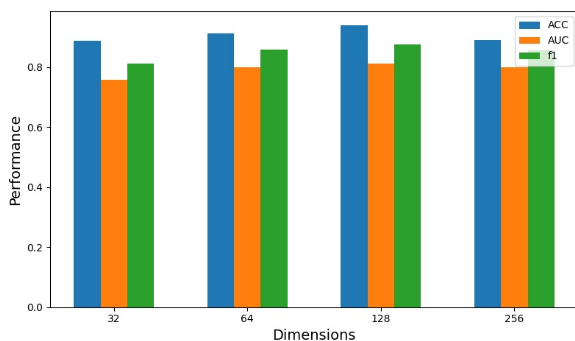


Fig. 3. Dimensionality impact on performance on the AIZU_Cpp

The impact of dimensionality on programming ability metrics is illustrated in Fig. 3. As the dimensionality increases, we observe notable variations in the performance metrics, including accuracy (ACC), area under the curve (AUC), and F1 score. For the dimensions tested, specifically 32, 64, 128, and 256, the accuracy values show a general upward trend, peaking at a dimensionality of 128 with an accuracy of 0.939. However, at 256 dimensions, there is a slight decline to 0.890. This indicates that while a moderate increase in dimensionality can enhance the model's ability to capture relevant features, excessive dimensions may lead to overfitting or increased noise, adversely affecting performance. Similar trends are observed in the AUC and F1 scores. The AUC improves from 0.7569 at 32 dimensions to a maximum of 0.8112 at 128 dimensions, before slightly decreasing to 0.800 at 256. The F1 score reflects a comparable pattern, rising from 0.8129 to 0.8749 and then dropping to 0.8569 at the highest dimensionality. These findings suggest that there exists an optimal range of dimensionality that maximizes programming ability metrics. Specifically, 128 dimensions appear to balance the trade-off between model complexity and generalization ability, leading to superior performance in measuring programming skills.

4. Conclusion

In this paper, we present MDPAT, a novel multigraph-based method for tracing programming ability in vocational college students. By capturing both static and dynamic aspects of programming activities, MDPAT offers a more nuanced assessment of proficiency. Utilizing the Programming Knowledge Graph and the Programming Iteration Graph, MDPAT models the structural and evolutionary features of code. Inspired by LSTM networks, it employs gated update mechanisms for programming knowledge and ability, allowing for selective integration of new information while retaining essential prior knowledge. MDPAT enhances prediction accuracy in programming exercises and outperforms state-of-the-art methods, achieving a 5.06% accuracy improvement on the Atcoder_C dataset. Ablation studies highlight the importance of balancing feature representation and knowledge integration. MDPAT advances programming ability assessment, providing detailed proficiency tracking. In future work, we plan to explore the integration of contextual factors such as problem difficulty and

learning behaviors, as well as extend MDPAT to support additional programming languages and diverse educational environment.

References

[1] G. Abdelrahman, Q. Wang, and B. Nunes, (2023) *"Knowledge tracing: A survey"* **ACM Computing Surveys** ***55***(11): 1–37. DOI: 10.1145/3569576.

[2] Y. Yin, L. Dai, Z. Huang, S. Shen, F. Wang, Q. Liu, E. Chen, and X. Li. "Tracing knowledge instead of patterns: Stable knowledge tracing with diagnostic transformer". In: *Proceedings of the ACM Web Conference 2023*. 2023, 855–864. DOI: 10.1145/3543507.358325.

[3] Z. Liu, Q. Liu, J. Chen, S. Huang, B. Gao, W. Luo, and J. Weng. "Enhancing deep knowledge tracing with auxiliary tasks". In: *Proceedings of the ACM Web Conference 2023*. 2023, 4178–4187. DOI: 10.1145/3543507.358386.

[4] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, et al. "CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks". In: *Annual Conference on Neural Information Processing Systems*. 2021. DOI: 10.48550/arXiv.2105.12655.

[5] J. Gao, M. Liu, P. Li, J. Zhang, and Z. Chen, (2023) *"Deep Multiview Adaptive Clustering With Semantic Invariance"* **IEEE Transactions on Neural Networks and Learning Systems**: DOI: 10.1109/TNNLS.2023.3265699.

[6] P. Li, J. Gao, J. Zhang, S. Jin, and Z. Chen, (2022) *"Deep Reinforcement Clustering"* **IEEE Transactions on Multimedia**: DOI: 10.1109/TMM.2022.3233249.

[7] R. Zhu, D. Zhang, C. Han, M. Gaol, X. Lu, W. Qian, and A. Zhou. "Programming knowledge tracing: A comprehensive dataset and a new model". In: *2022 IEEE International Conference on Data Mining Workshops (ICDMW)*. 2022, 298–307. DOI: 10.1109/ICDMW58026.2022.00048.

[8] S. Shen, Q. Liu, E. Chen, H. Wu, Z. Huang, W. Zhao, Y. Su, H. Ma, and S. Wang. "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*. 2020, 1857–1860. DOI: 10.1145/3397271.3401288.

[9] L. Wang, A. Sy, L. Liu, and C. Piech, (2017) *"Learning to Represent Student Knowledge on Programming Exercises Using Deep Learning."* **International Educational Data Mining Society**:

[10] V. Swamy, A. Guo, S. Lau, W. Wu, M. Wu, Z. Pardos, and D. Culler. "Deep knowledge tracing for free-form student code progression". In: *Artificial Intelligence in Education: 19th International Conference, AIED 2018, London, UK, June 27–30, 2018, Proceedings, Part II 19*. 2018, 348–352.

[11] B. Jiang, S. Wu, C. Yin, and H. Zhang, (2020) *"Knowledge tracing within single programming practice using problem-solving process data"* **IEEE Transactions on Learning Technologies** ***13***(4): 822–832. DOI: 10.1109/TLT.2020.3032980.

[12] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., (2020) *"Codebert: A pre-trained model for programming and natural languages"* **arXiv preprint arXiv:2002.08155**: DOI: 10.48550/arXiv.2002.08155.

[13] Y. Watanobe, M. M. Rahman, T. Matsumoto, U. K. Rage, and P. Ravikumar, (2022) *"Online judge system: Requirements, architecture, and experiences"* **International Journal of Software Engineering and Knowledge Engineering** ***32***(06): 917–946. DOI: 10.1142/S0218194022500346.

[14] M. Lu, Y. Wang, D. Tan, and L. Zhao, (2021) *"Student program classification using gated graph attention neural network"* **IEEE Access** ***9***: 87857–87868. DOI: 10.1109/ACCESS.2021.3063475.

[15] Y. Watanobe, M. M. Rahman, M. F. I. Amin, and R. Kabir, (2023) *"Identifying algorithm in program code based on structural features using CNN classification model"* **Applied Intelligence** ***53***(10): 12210–12236. DOI: 10.1007/510489-022-04078-y.

[16] Y. Ouyang, Y. Wang, R. Gao, Y. Zeng, J. Liu, and Z. Ye, (2024) *"Multi-level contrastive graph learning for academic abnormality prediction"* **Neural Computing and Applications** ***36***(7): 3681–3698. DOI: s00521-023-09268-4.

[17] J. Gao, P. Li, A. A. Laghari, G. Srivastava, T. R. Gadekallu, S. Abbas, and J. Zhang, (2024) *"Incomplete multiview clustering via semidiscrete optimal transport for multimedia data mining in IoT"* **ACM Transactions on Multimedia Computing, Communications and Applications** ***20***(6): 1–20. DOI: 10.1145/3625548.

[**18**] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. J. Guibas, and J. Sohl-Dickstein, (2015) *"Deep knowledge tracing"* **Advances in neural information processing systems** *28*: