EliRank: A Code Editing History Based Ranking Model for Early Detection of Students in Need

Jungkook Park
School of Computing, KAIST
Daejeon, Republic of Korea
pjknkda@kaist.ac.kr

Alice Oh
School of Computing, KAIST
Daejeon, Republic of Korea
alice.oh@kaist.edu

ABSTRACT

Research on programming education shows that novice programming students benefit significantly from one-to-one tutoring. While many systems propose to replicate the effectiveness of one-to-one tutoring in large-scale classes, it remains a challenge to develop systems with an approach to finding students who need the tutors' help the most. In this paper, we explore the idea of predicting the priority of students in need with a data-driven approach. Among various metrics to calculate the priority of students in need, we adopt time-on-task metric. Previous studies have found that excessively long time-on-task can be used as an indication of students' struggling. Aligned with this, we reduce the problem of finding students with the highest priority to the problem of finding students with the longest time-on-task. To solve the reduced problem, we present EliRank, a ranking model that finds students with the longest estimated time-on-task, using the students' first few minutes of fine-grained code editing history. EliRank recommends students in the descending order of estimated time-on-task, enabling tutors to efficiently monitor and find the students in need at scale in real time. To evaluate the performance of EliRank, we build and publish a new real-world dataset consisting of 15 programming exercises solved by 4000+ students in an introduction to programming class at a university. Unlike the currently available open code editing history datasets, our dataset contains code editing operations at a character-level granularity to minimize the loss of contextual information from students. We also introduce diff-augmented abstract syntax tree (DAST), a novel structured code representation that minimizes the loss of fine-grained code change information during code parsing. The evaluation of EliRank on our dataset shows that EliRank effectively finds students with the longest estimated time-on-task, for early detection of students in need. Also, we illustrate in depth (i) the effectiveness of DAST, (ii) the potential to control the tradeoff between early detection and the prediction accuracy of the model, and (iii) the transferability to unseen programming exercise via zero-shot transfer learning.

CCS CONCEPTS

• **Applied computing → Computer-assisted instruction**.

L@S '23, July 20–22, 2023, Copenhagen, Denmark.
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0025-5/23/07…$15.00
https://doi.org/10.1145/3573051.3593387

KEYWORDS

code editing history, machine learning, programming education, recommendation system

ACM Reference Format:
Jungkook Park and Alice Oh. 2023. EliRank: A Code Editing History Based Ranking Model for Early Detection of Students in Need. In *Proceedings of the Tenth ACM Conference on Learning @ Scale (L@S '23), July 20–22, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3573051.3593387

1 INTRODUCTION

Programming education research found that novice programmers can learn programming more effectively when there is one-to-one help by a tutor [24]. However, this kind of tutoring requires personal interactions between students and tutors and is difficult to scale. For example, in a MOOC environment, tutors can monitor and help only a small fraction of students, as the number of students is much greater than the number of tutors.

A number of tutoring systems have been proposed to replicate the effectiveness of one-to-one tutoring in large-scale classes [2, 7, 14]. One approach to such a system is automating tutors by generating feedback automatically. These systems generate feedback from students' submitted code by executing predefined unit tests, clustering similar codes [22], or comparing the code to the reference solution [27]. Another approach is building a recommendation system that finds students in need among many students and informs tutors. For example, [16] calculates an error quotient score that characterizes how much a student struggles with syntax errors while programming and suggests instructors use this score to sense how the students are doing. Similarly, [14] builds a system that shows students in the descending order of their activity frequency to help tutors monitor the students at scale.

In this paper, we extend the recommendation system approach by combining it with a data-driven methodology used by the automated tutor approach. We design a model for a recommendation system to compute the priority of students in need and inform the tutors of the highest priority students. To improve the recommendation, we apply a data-driven methodology based on the students' code editing history to rank the students in terms of their priority. By combining two approaches, we aim to help tutors monitor and manage students effectively at scale in real time while maintaining the pedagogical effects of the human touch between tutors and students.

Among various metrics to calculate the priority of students, we focus on time-on-task metric. Psychologically, time-on-task metric has two different interpretations; one is that the metric has a positive correlation with the learner's engagement, and the other is that there is a negative correlation with the learner's skill level [12]. We focus on the negative correlation between the metric and learners' skill level, especially the observation that excessively long time-on-task can indicate students' struggling [20]. Aligned with this, we can reduce the problem of finding students with the highest priority to the problem of finding students with the longest time-on-task.

To solve the reduced problem, we present EliRank, a ranking model that finds students with the longest estimated time-on-task, using the students' first few minutes of fine-grained code editing history. EliRank recommends students in the descending order of estimated time-on-task, enabling tutors to efficiently monitor and find the students in need at scale in real time. Especially, EliRank adopts a ranking model rather than a regression because the accurate calculation of time-on-task is not our direct goal, but order matters. EliRank cooperates with graph neural network (GNN), graph pooling, and transformer encoder to extract information about code semantics and students' learning state from code editing histories.

To evaluate the performance of EliRank, we build a new real-world dataset consisting of 15 programming exercises and over 4000 students. Unlike currently available open code editing history dataset with a large scale [11, 15, 31], our dataset contains character-level editing operations that are more granular than commit or submission levels. The fine granularity of the dataset has multiple benefits; for example, a previous study found that the fine-grained code change histories have rich information to understand the intent behind the code better [25]. Also, the fine-grained logs improve the estimation of time-on-task to better correlates with students' performance [20]. We expect that our dataset will provide rich information about code semantics and students' learning status, allowing machine learning models to calculate the priority of students more accurately.

However, adopting our fine-grained dataset to existing machine learning models such as [8, 13] is not trivial. The reason is that code representations such as abstract syntax tree (AST) or data flow used by those models can only parse syntactically correct code for the given programming language. In contrast, the code created from character-level editing operations does not always follow that syntax. Although models can discard intermediate editing operations that make syntactically incorrect code and use only the remains, this results in the loss of fine-grained code change information in the dataset.

To overcome this challenge, we introduce diff-augmented abstract syntax tree (DAST), a variant of AST-based code representation that efficiently embeds code change information obtained from the fine-grained code editing history. DAST augments nodes of AST with an additional property that indicates whether any editing operations in the history affect the nodes. With this property, DAST powers EliRank by minimizing the loss of fine-grained code change information even though some editing operations are discarded to make the code AST parsable.

We evaluate the efficacy of EliRank using our real-world dataset. With four research questions, we investigate (1) the overall performance of EliRank, (2) the effectiveness of DAST, (3) the trade-off between timeliness of early detection and model performance, and (4) the transferability of the model for unseen programming exercise.

The contributions of this paper are:

- Real-world open dataset that captures students' fine-grained code editing history in an introduction to programming class at a university.
- The DAST code representation that efficiently augments AST to minimize the loss of fine-grained code change information.
- The EliRank, a ranking model that takes the students' first few minutes of fine-grained code editing history and finds students with the longest estimated time-on-task, for early detection of students in need.

The source code and the dataset are publicly available on the website.

2 RELATED WORK

Our research is related to two major research areas: an intelligent tutoring system for programming education and machine learning for source codes.

Several studies propose a system supporting programming education on a large scale while preserving the effect of one-to-one tutoring. One approach is to automate the tutor's tasks, such as grading submissions or giving feedback. For example, [22] proposed a system that gives automated feedback to a cluster of codes using predefined unit tests per programming exercises. Likewise, [23] proposed a prototype called robot tutor, which automatically generates answers to the questions from students. In this paper, our approach is in line with making tutors more efficient rather than automation. With this approach, we investigate a way of scaling the system without losing the pedagogical effects of the human touch between tutors and students [3].

Another approach is to provide a system that enables tutors to monitor and manage students efficiently. [14] is one such system that uses a heuristic algorithm for visualizing students in a specific order to make tutors monitor and help a large number of students in real time. [16] designed a score that characterizes how much a student struggles with syntax errors and suggests an interface for tutors to monitor students by the calculated score. Meanwhile, [1, 21] introduced machine learning models that predict students' performance using various logs and find at-risk students based on the predicted performance. These studies have a similar approach to ours that help tutors monitor and find students by a specific metric, where the metric is calculated from datasets. However, the main differences are that our model makes a prediction based on the code semantics, which is not directly visible in the dataset and focuses on finding the priority order of students rather than calculating exact metric values.

Several studies found that graph neural network models are effective on source code-related tasks such as code summarization [18] or clone detection [5]. Recent studies also suggested improving the model performance by expanding the model input from code snapshots to the code history. For example, [8] proposed a system that explicitly modeled the code difference between code commits and achieved state-of-the-art performance on commit message generation tasks. In this paper, we also develop a graph neural network model on structured code representations. However, our model incorporates fine-grained code editing history with character-level granularity, which is more granular than commit or submission levels. Furthermore, we propose a novel code representation specialized in the dataset to minimize the loss of fine-grained code change information during AST parsing.

3 CODE EDITING HISTORY DATASET

To validate our data-driven approach, we build and publish a new real-world dataset gathered from CS1 class at KAIST (Korea Advanced Institute of Science and Technology). This mandatory class is open to all freshman regardless of their major, with an average enrollment of 400 to 500 students per semester. The course consists of a lecture component and a lab session component, and students are required to attend one of each per week.

During the lab session, students are asked to solve five python programming exercises within three hours. Once a student submits their code, it is manually graded by any available tutor. Students are permitted to leave the session after all of their submitted codes have been confirmed as correct by the tutor. Throughout the session, students are free to ask tutors any questions.

To collect dataset, we provide students with a web-based code editor that allows them to edit, execute, and submit codes. While a student is solving the exercises, the editor records character-level editing operations and stores them in a database. The character-level granularity is provided as the best effort and is not guaranteed in the cases where copy and paste are used or the editor's auto-completion feature is invoked. Also, to reduce the server load in heavy traffic situations, editing operations are merged if the interval between consecutive operations is less than 100 milliseconds.

We collected our dataset from 15 python programming exercises from the first three lab sessions of the class. The data spans 11 semesters, from Spring 2016 to Spring 2021, and includes code editing histories from over 4000 students. Note that the lab sessions were conducted offline from Spring 2016 to Fall 2019 and moved online from Spring 2020 to Spring 2021 due to the COVID-19 pandemic.

In the dataset, we define code editing history as the sequence of editing operations, and use Operational Transformation (OT) [28] to efficiently represent editing operations with millisecond time precision. The OT consists of 3 operations:

- skip (*amount*) : move the current cursor forward by *amount* characters.
- insert (*text*) : insert *text* at the current cursor position.
- delete (*text*) : delete *text* from the current cursor position.

For example, given a text "abcde", applying the sequence of OTs $[\text{skip}(2), \text{insert}(\text{“12”}), \text{delete}(\text{“c”})]$ results in a text "ab12de". Table 1 shows the example of code editing history data in the dataset.

To build a dataset, we applied the following preprocessing to raw data. First, if the interval between two consecutive editing operations exceeds 120 seconds, it is considered a different editing session, and the interval is shortened to 120 seconds. This follows the previous study [19] to calculate accurate time-on-task by removing inactive sessions, for example, students taking breaks. Second, to exclude cases where the whole code is copied outside the editor, we exclude code with a shorter history than (1) less than 300 seconds, (2) less than ten editing operations, or (3) the lowest 5% editing operations in the same exercise. Third, to exclude cases where the editor is used for notepad purposes other than solving given exercises, we exclude code with a longer history than the highest 95% editing operations in the same exercise or the AST-parsed code has more than 300 nodes. The second and third steps are primarily designed to exclude cases where the entire keystroke-level history is not captured and is considered incomplete. After the preprocessing, we calculate students' time-on-task by measuring the time between the first and the last editing operations.

Table 2 shows the statistics of the dataset. For each exercise, the Pearson correlation coefficient ρ between the number of editing

Table 1: The example of code editing history data. Each editing operation is recorded as the sequence of Operational Transformations with millisecond time precision.

Class ID	bf95180dd756fd204fb01ce4916ae9323b19a3aa
Doc ID	10d9859fc620db615c8aa74e324f3805b0ae8877
Task ID	P1
Offline?	false
Created	2021-01-01 11:22:33.12345
Skeleton	from cs1robots import *\n# Write your code here!
Edits[0]	1.000, [skip(48), insert("\nx")]
Edits[1]	4.404, [skip(50), insert("=")]
Edits[2]	10.645, [skip(51), insert("10")]
Edits[3]	11.378, [skip(51), delete("1"), insert("2")]
Edits[...]	...

Table 2: The statistics of 15 programming exercises in the dataset after preprocessing. The numbers in the parenthesis represent the standard deviation. ρ represents the Pearson correlation between the time-on-task and the number of edits. ρ_{300} represents the Pearson correlation between the time-on-task and the number of edits before the first 300 seconds.

ID	N	Avg. Edits	Avg. Time (s)	ρ	ρ_{300}
P1	4074	563.6 (242.9)	1217.4 (661.7)	0.702	-0.354
P2	3512	345.2 (120.9)	636.1 (304.5)	0.549	-0.338
P3	3822	434.5 (149.7)	681.2 (319.4)	0.559	-0.370
P4	3636	426.6 (178.5)	742.7 (384.0)	0.602	-0.294
P5	3273	733.2 (336.5)	1463.0 (841.5)	0.703	-0.299
P6	3661	535.6 (303.8)	1200.6 (721.3)	0.736	-0.199
P7	2341	329.1 (215.3)	731.7 (426.6)	0.604	-0.030
P8	4048	434.3 (188.8)	1046.2 (583.2)	0.692	-0.326
P9	2068	307.2 (237.9)	786.7 (509.7)	0.669	-0.011
P10	4133	871.9 (463.3)	2328.9 (1362.8)	0.797	-0.223
P11	3998	411.3 (179.9)	952.0 (495.6)	0.660	-0.352
P12	4039	483.2 (186.9)	936.7 (473.4)	0.635	-0.374
P13	3636	709.0 (413.9)	1579.2 (974.6)	0.748	-0.087
P14	4127	990.7 (467.1)	2436.8 (1296.0)	0.789	-0.165
P15	3672	313.9 (155.4)	867.6 (506.7)	0.696	-0.263

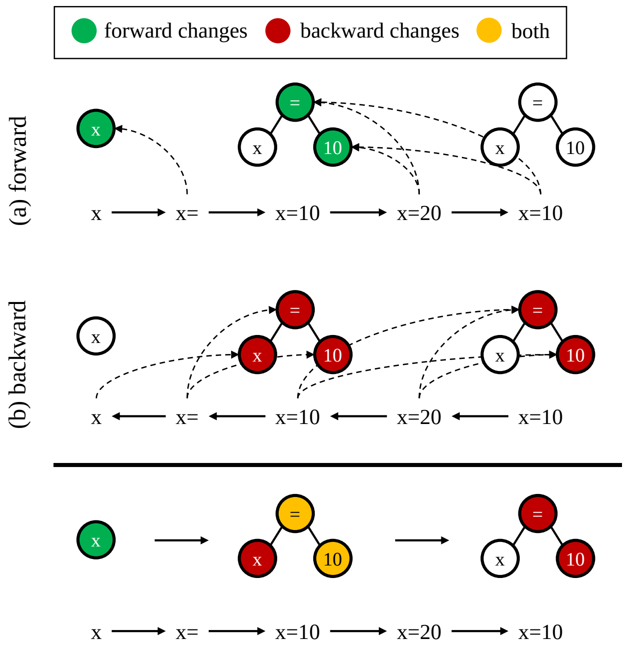

Figure 1: The process of building DAST from a code editing history. The algorithm finds AST nodes that are affected by forward and backward code changes. Then, each nodes are annotated as one of four change states – forward, backward, both, or none of them.

operations and time-on-task is between 0.5 and 0.8. However, in a realistic lab session environment where students participate in real time, the entire editing history is unavailable, and models can only access the first few minutes of history. In that perspective, given the first 300 seconds of the histories, the coefficient ρ_{300} between the two entities becomes between -0.4 and 0, meaning a weak negative or no correlation. This implies that the number of editing operations cannot be used as a direct indicator of time-on-task in the early stages of students' participation and suggests that models estimating time-on-task should consider other information, such as code semantics.

In the dataset, data from offline and online lab sessions are mixed because the data collection period spans the COVID-19 pandemic. However, the measured statistics between the offline and the online sessions show marginal differences. More specifically, the average difference of two statistics – the average number of edits and the average time-on-task – between the two environments show 4.85% and 7.24%, respectively. Therefore, in this paper, we use the dataset as one without distinguishing the two different environments.

4 DIFF-AUGMENTED ABSTRACT SYNTAX TREE

We propose diff-augmented abstract syntax tree (DAST), a novel AST-based code representation that efficiently embeds code change information from a fine-grained code editing history. In addition to the general AST node properties such as source code location and node type, DAST adds two additional Boolean node properties δ_f and δ_b. Each new property represents whether the node is affected by any forward (δ_f) or backward (δ_b) code changes in the editing history.

More specifically, suppose two AST G_1 and G_2 parsed from two different codes on the same code editing history. To simplify the description, the method to choose the two codes from the history will be described in the Section 5.1. Also, we assume that G_1 is located ahead of G_2 in the editing history. Then, if any editing operation in the history from G_1 to G_2 modifies the source code location of AST node $v_1 \in G_1$, δ_f property of v_1 becomes *true* (Figure 1-(a)). Similarly, if any editing operation in the reversed history from G_2 to G_1 modifies the source code location of AST node $v_2 \in G_2$, δ_b property of v_2 becomes *true* (Figure 1-(b)). By tracking editing operations in both forward and backward directions, we reduce the possibility of a situation where operations fail to find affected nodes because the nodes are not created yet. For example, in Figure 1-(a), the second editing operation [skip(2), insert("10")] has no affected nodes because AST nodes for "+" and "10" are not yet created in the previously parsed AST. In that situation, the editing operation's fine-grained information is discarded in the forward path. However, in the backward path (Figure 1-(b)), the same operation now has affected nodes, and the change information from the operation is annotated in the nodes. This approach is language-agnostic and nondestructive to the existing AST structure; therefore, DAST could be applied as an in-place to existing AST-based algorithms.

The new properties of DAST provide additional contextual information beyond the code structure and semantics. This comes from the granular nature of the dataset, where even the two identical codes can produce different DASTs depending on their editing history. For example, suppose two ASTs with the same code and the editing history of writing and deleting a specific code between them. In such a situation, the two ASTs will be the same, whereas the two DASTs will have identifiable nodes where the nodes with δ_f on one side and the nodes with δ_b on the other side. The repetition of this write-and-delete behavior in a short time range can be interpreted as "applying the concept without real understanding" described in the CS-specific educational taxonomy [10] and provides a signal to find out where students are located on the learning path.

The algorithm for constructing DAST is done efficiently by avoiding computationally expensive comparisons between ASTs. The main idea is that the algorithm can track code locations modified by OTs while building the mapping from the code locations to the corresponding AST nodes. Algorithm 1 describes the pseudo-code of building DAST when AST G, code C, and editing history H are given.

(1) The algorithm first calculates a mapping X from code locations to AST node indexes. This process has a time complexity of $O(|C||G|)$.
(2) Let Y be a set of affected nodes.
(3) For each editing operation $e \in H$, we denote the sequence of OTs in the operation as e^{ots}. Then, the set of affected nodes is calculated by applying the following process to $ot \in e^{ots}$ in sequence.

Algorithm 1 An algorithm to identify affected nodes in AST G when the code C and the editing history H are given

```
1:  $minus(n) :=$ a list $[-1, ..., -1]$ with the length $n$
2:  $X \leftarrow minus(|C|)$
3:  for $v \in$ preorder$(G)$ do
4:      $X[v.codeStart : v.codeEnd] \leftarrow v.id$
5:  end for
6:  $Y \leftarrow \{\}$
7:  for $e \in H$ do
8:      $X' \leftarrow []$
9:      $p \leftarrow 0$
10:     for $ot \in e^{ots}$ do
11:         if $ot$ is SKIP then
12:             $X' \leftarrow X' \cdot X[p : p + ot.amount]$
13:             $p \leftarrow p + ot.amount$
14:         else if $ot$ is INSERT then
15:             $X' \leftarrow X' \cdot minus(|ot.text|)$
16:             $Y \leftarrow Y \cup \{X[p-1], X[p]\}$
17:         else if $ot$ is DELETE then
18:             $Y \leftarrow Y \cup \{X[p-1], X[p], ..., X[p + |ot.text|]\}$
19:             $p \leftarrow p + |ot.text|$
20:         end if
21:     end for
22:     $X \leftarrow X' \cdot X[p :]$
23: end for
24: $Y \leftarrow Y \setminus \{-1\}$
25: return $Y$
```

- If ot is *skip*, there are no affected nodes. This process has the time complexity of $O(1)$.
- If ot is *insert*, we mark the code location right before/after the current cursor position, adding corresponding AST nodes to Y. This process has a time complexity of $O(log|G|)$.
- If ot is *delete*, we mark the code location of the deleted text and right before/after the deleted text, adding corresponding AST nodes to Y. This process has a time complexity of $O(|C|log|G|)$.

(4) Return the set of affected nodes Y

The overall time complexity becomes $O(|C||G| + N|C|log|G|)$ where $N := \Sigma_{e \in H}|e^{ots}|$ is the total number of OTs in the editing history.

5 CODE EDITING HISTORY BASED RANKING MODEL

We present EliRank, a ranking model that takes the students' first few minutes of fine-grained code editing history and finds students with the longest estimated time-on-task, for early detection of students in need. Using Algorithm 1, EliRank converts code editing history into a sequence of DAST while minimizing the loss of fine-grained code change information from the history. Then, EliRank adopts a graph neural network (GNN) and graph pooling algorithm to extract graph embeddings from DASTs. Finally, with the transformer encoder and feedforward neural network (FNN), the graph embeddings are compacted into a ranking score to calculate the ranking loss to other editing histories. The overall architecture of the model is shown in Figure 2.

Before describing each step of EliRank in detail, we first summarize basic notations for editing history. e_i denotes the i-th editing operation in the editing history H. The operation's created timestamp and OTs are denoted as e_i^{ts} and e_i^{ots}. The timestamp is recorded as the relative time for the first operation in history, starting from 0. c_i denotes a code snapshot at the time e_i is recorded, corresponding to the result of applying $[e_j^{ots}|1 \leq j \leq i]$ to an empty text.

5.1 AST parsing

EliRank parses editing histories on the dataset into the sequences of ASTs. However, due to the character-level granularity of editing operations, the parsing process is nontrivial, and two issues arise during the process.

The first issue is that the granularity of editing operations needs to be more consistent. For example, caused by the optimization logic that merges consecutive editing operations if they are closer than 100 milliseconds, a single editing operation can have OTs that modify multiple characters or even words depending on the keyboard typing speed. Similarly, editing operations can contain arbitrarily long text when copy and paste from an external editor are used. This issue introduces imbalances in data frequency depending on students' editing styles and requires normalization of granularity for better model stability and performance.

The second issue is that not all code snapshots are AST parsable. From the programming language perspective, the code must follow a language-specific syntax to be a valid program. However, in our code editor, there is no restriction on writing arbitrary text, and editing operations are recorded without the awareness of the language-specific syntax. Therefore, the code snapshots for each editing operation result in invalid programs that the AST parser cannot parse.

To mitigate the two issues, EliRank converts each editing history into an intermediate format called representative code snapshots before parsing it to ASTs. We define a *representative code snapshot* as the last AST parsable code snapshot available at each timestamp, dividing the entire editing history by a fixed time interval. More specifically, for the fixed time interval s, i-th representative code snapshot $c'_i := c_{r(i)}$ where $r(i)$ represents the index of the last AST parsable code. Formally, $r(i)$ satisfies the following properties; (i) $e_{r(i)}^{ts} \leq s \cdot i$, (ii) $c_{r(i)}$ is AST parsable, (iii) $\forall j : r(i) < j \wedge e_j^{ts} \leq s \cdot i$, c_j is not AST parsable (Figure 2-(a)). With this definition, any code editing history can be converted into the sequence of representative code snapshots and then parsed into the sequence of ASTs.

5.2 DAST constructing

EliRank converts the sequence of ASTs into the sequence of DAST using fine-grained editing operations in the history. This step aims to compensate for the loss of fine-grained code change information during the first AST parsing step and discover additional context information that AST cannot represent.

To construct DAST g'_i for the i-th AST g_i in the sequence, Algorithm 1 is used with arguments as follows (Figure 2-(b)).

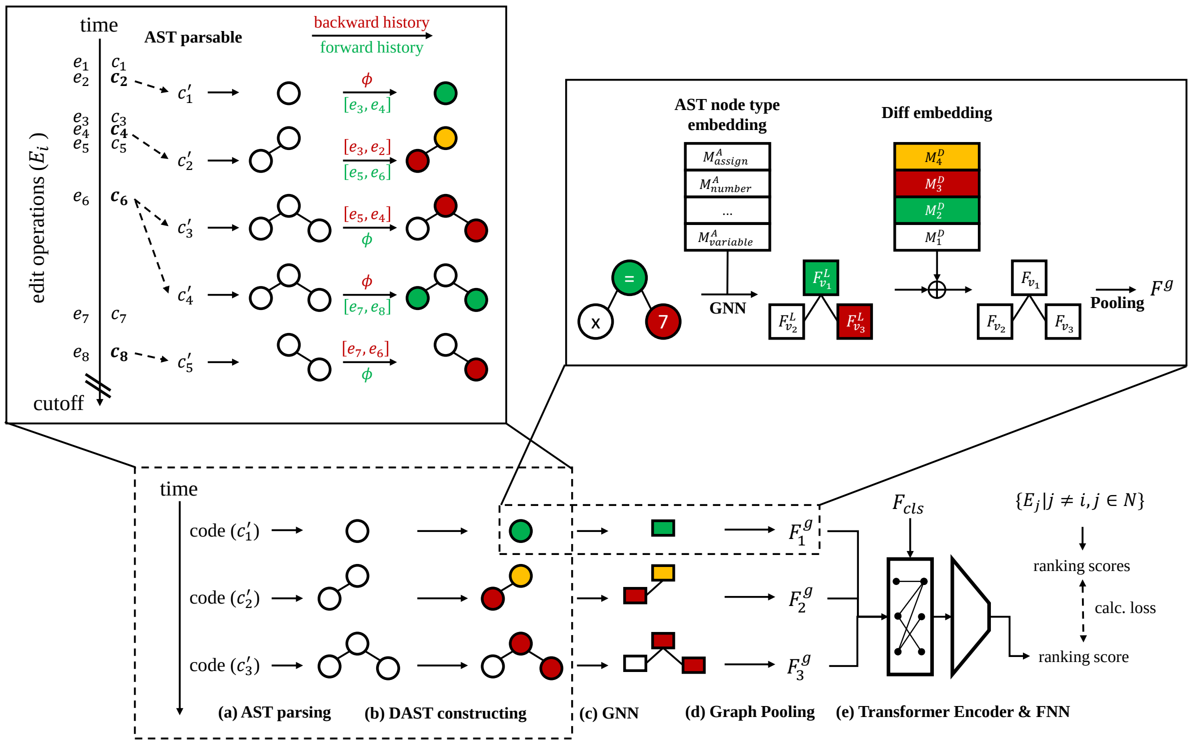

Figure 2: The overall architecture of EliRank. EliRank takes a sequence of fine-grained editing operations. The sequence is pipelined through (a) AST parsing, (b) DAST constructing, (c) GNN, (d) graph pooling, (e) transformer encoder & FNN, then finally calculated as a ranking score.

Forward change property δ_f, the algorithm takes graph g_i, code c'_i, and history $[e^{ots}_j | r^{-1}(i) < j \leq r^{-1}(i+1)]$. Here, $r^{-1}(i)$ represents the inverse function of $r(i)$ that satisfies $r^{-1}(r(i)) = i$.

Backward change property δ_b, the algorithm takes the same arguments except for the reversed history reverse$([(e^{ots}_j)^{-1} | r^{-1}(i-1) \leq j < r^{-1}(i)])$. Here, $(e^{ots}_j)^{-1}$ is the inverse operation of e^{ots}_j that satisfies apply(apply$(X, e^{ots}_j), (e^{ots}_j)^{-1}) = T$ for any text X.

This process does not involve direct comparisons between ASTs; therefore, it can be done efficiently with polynomial time complexity.

5.3 Graph Neural Network

For each DAST, EliRank produces the embeddings of DAST nodes using GNN to handle the graph structure and the nodes' features simultaneously. Given a graph $G = (V, E)$, GNN calculates the embedding F^l_v for the node $v \in V$ in the l-th layer by (1). In the equation, $\mathcal{N}(v)$ represents a set of nodes connected to v, Ψ^l and Φ^l are differentiable functions, and **AGG** is a differentiable and permutation invariant aggregation function.

$$F^l_v = \Psi^l(F^{l-1}_v, \mathbf{AGG}\{\Phi^l(F^{l-1}_v, F^{l-1}_u, E_{u,v})\}_{u \in \mathcal{N}(v)}) \qquad (1)$$

EliRank defines node feature as the AST node type index, for example, 1 for *If* node, 2 for *Assign* node, 3 for *Constant* node. For notations, we use m as the total number of AST node types and $T(v)$ as the function for mapping the AST node type of v to a number between 1 and m. EliRank defines two randomly initialized matrices; $M^A \in \mathbb{R}^{m \times d_A}$ defines embeddings for each of AST node type and $M^{A2G} \in \mathbb{R}^{d_A \times d_G}$ resizes AST node type embedding to fit into GNN unit size. Based on these notations, the initial GNN embedding is denoted as $F^0_v := M^A_{T(v)} \times M^{A2G}$ (Figure 2-(c)).

5.4 Graph Pooling

The node embeddings derived from the previous step are aggregated into graph embeddings through GraphMultisetPooling[4], one of the graph pooling algorithms focusing on memory and time efficiency. This step makes the sequence of tree-structured embeddings into the sequence of embedding vectors, allowing various sequence-based machine learning models to be used afterward.

From the observation of the dataset, we found that fine-grained editing operations between two representative code snapshots affect less than 10% of the entire AST on average (Table 3). To reduce the information redundancy from the unaffected nodes and make the pooling algorithm give more attention to the affected nodes, EliRank models the DAST properties δ_f and δ_b in the algorithm. This is done by adding a diff embedding matrix $M^D \in \mathbb{R}^{4 \times d}$ to the node embeddings before the algorithm pools the nodes.

Table 3: The statistics for the number of affected nodes in the dataset when the history is truncated to the first 300 seconds. The affected node is calculated using DAST constructing algorithm. The ratio is calculated for each ASTs.

ID	Avg. # Nodes	Avg. # Affected Nodes	Avg. Ratio
P1	43.97	2.54	9.18%
P2	47.61	1.87	7.92%
P3	69.67	1.77	4.49%
P4	60.61	2.25	5.63%
P5	70.01	3.87	7.03%
P6	61.09	3.20	6.98%
P7	81.22	3.66	5.75%
P8	49.81	2.37	5.78%
P9	77.51	3.07	5.13%
P10	56.97	2.57	5.63%
P11	42.94	2.09	7.36%
P12	59.73	2.50	6.07%
P13	87.80	3.72	5.26%
P14	57.84	2.53	4.26%
P15	83.50	2.04	3.31%

$$F_v = \begin{cases} F_v^L \oplus M_4^D, & \delta_f(v) \wedge \delta_b(v) \\ F_v^L \oplus M_3^D, & \delta_b(v) \\ F_v^L \oplus M_2^D, & \delta_f(v) \\ F_v^L \oplus M_1^D, & \text{otherwise} \end{cases} \tag{2}$$

After the diff embedding is added to the node embeddings, the graph embedding for i-th DAST $G_i = (V_i, E_i)$ is derived as follows (Figure 2-(d)):

$$F_i^g = \text{Pooling}(\{F_v\}_{v \in V_i}, E_i) \tag{3}$$

5.5 Transformer Encoder & Feedforward Neural Network

Transformer encoder [29] and feedforward neural network (FNN) convert the sequence of graph embeddings from the previous step to a ranking score. EliRank uses *cls* vector, a randomly initialized and appended to the input sequence, to aggregate the whole sequence of the embeddings into a single dense vector [17]. Also, to make the encoder assign position-aware attention, we add dynamic positional encoding $P \in \mathbb{R}^{|C'| \times d}$ to the input sequence. As a result, the encoder outputs the embeddings of the sequence as $R^{pre} = \text{Transformer}([F_{cls}, F_1^g \oplus P_1, \ldots, F_{|C'|}^g \oplus P_{|C'|}])$. Among the output embeddings, we only use R_{cls}^{pre}, which is the embedding of *cls* vector. Finally, FNN takes R_{cls}^{pre} and produces a single ranking score $r = FNN(R_{cls}^{pre}) \in \mathbb{R}$ (Figure 2-(e)).

5.6 Loss function

EliRank applies ListMLE[30] as a loss function to optimize EliRank. With ListMLE, losses are minimized when the order of the predicted ranking scores follows the same order as the ground truth. Consequently, the optimization of EliRank becomes easier than solving a regression problem of precisely predicting the ground truth.

To train EliRank, we use truncated code editing histories as input and time-on-task metric as output. The truncation on input simulates a realistic lab session environment where complete code editing histories are yet to be available as students participate in real time. For each training epoch, the optimizer creates a random batch of students' truncated code editing histories and calculates the ranking scores of each using EliRank. Then, ListMLE calculates losses by checking the correctness of the order of the ranking scores.

6 EVALUATION

In this section, we evaluate EliRank with the fine-grained code editing history dataset. We indirectly validate the performance of EliRank by measuring how accurately EliRank finds students with the longest time-on-task, instead of directly calculating the accuracy of finding students in need. The main reason is that the dataset lacks explicit signals of the need for assistance. Furthermore, even if such explicit signals supplement the dataset, EliRank cannot rely on that signals, as students may not "raise a hand" even if they are stuck, and our motivation to find such students proactively. Instead, as finding the students with the highest priority (i.e., the longest estimated time-on-task) can still benefit tutors by reducing the number of candidates to monitor proactively in large-scale classes, we evaluate EliRank from the perspective of its recommendation performance.

We designed the experiments to examine the following four research questions:

- RQ1: How does EliRank perform compared to heuristic algorithms to find students with the highest priority?
- RQ2: How does DAST improve the performance of EliRank?
- RQ3: How much code editing history does EliRank need to make a prediction?
- RQ4: How does EliRank perform under the cold start situation?

6.1 Experiment Setup

For each programming exercise, we split the dataset randomly to prepare a training set, validation set, and testing set, where the proportions of each are 70%, 15%, and 15%. Unless explicitly stated, we train and validate the model with the aggregated set across all 15 programming exercises and test with the set per exercise. We test the model per exercise because the difference in the difficulty of exercises causes the different distribution of ground truth (i.e., time-on-task), resulting in the order of ranking scores being valid only inside each exercise.

We measure the normalized discounted cumulative gain (NDCG) metric, widely used for ranking problems, to test the model's performance. The NDCG becomes 1.0 when the model produces ranking scores that order students in perfect order and get penalties when the order mismatches. The penalty from a mismatched order is calculated with two properties; (i) the amount of penalty is

Table 4: The *NDCG* score for the baselines. Welch's t-test is used to measure the statistical significance of each pair of models. The score in bold shows the significantly ($p < 0.01$) best score compared to all other models.

Model	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
300s cutoff															
Random	0.852	0.865	0.854	0.840	0.831	0.836	0.837	0.852	0.822	0.839	0.847	0.859	0.836	0.844	0.835
EvPerSec	0.837	0.847	0.831	0.829	0.815	0.808	0.844	0.835	0.804	0.821	0.818	0.841	0.825	0.838	0.822
EvPerSecNeg	0.872	0.873	0.878	0.873	0.856	0.843	0.829	0.864	0.793	0.857	**0.876**	0.849	0.846	0.855	0.857
EliRank	**0.893**	**0.918**	**0.903**	**0.891**	**0.872**	**0.881**	**0.906**	**0.898**	**0.858**	**0.866**	0.869	**0.895**	**0.880**	**0.873**	**0.901**
600s cutoff															
Random	0.855	0.866	0.851	0.843	0.839	0.835	0.842	0.855	0.808	0.842	0.843	0.856	0.848	0.846	0.842
EvPerSec	0.871	0.915	0.884	0.889	0.838	0.850	0.886	0.879	0.849	0.837	0.869	0.876	0.850	0.838	0.870
EvPerSecNeg	0.808	0.803	0.791	0.779	0.791	0.775	0.772	0.789	0.730	0.810	0.789	0.795	0.780	0.825	0.778
EliRank	**0.903**	**0.949**	**0.932**	**0.902**	**0.878**	**0.898**	**0.941**	**0.911**	**0.898**	**0.881**	**0.901**	**0.914**	**0.886**	**0.874**	**0.922**
900s cutoff															
Random	0.857	0.866	0.851	0.841	0.828	0.834	0.851	0.849	0.817	0.846	0.840	0.852	0.835	0.847	0.843
EvPerSec	0.889	0.958	0.936	0.919	0.842	0.880	0.941	0.905	0.889	0.849	0.892	0.906	0.872	0.843	0.902
EvPerSecNeg	0.788	0.798	0.785	0.771	0.768	0.758	0.763	0.777	0.720	0.786	0.773	0.781	0.760	0.804	0.765
EliRank	**0.923**	**0.973**	**0.950**	**0.945**	**0.888**	**0.913**	**0.954**	**0.930**	**0.914**	**0.892**	**0.919**	**0.933**	**0.904**	**0.876**	**0.944**

propositional to the ground truth of the misordered item, and (ii) logarithmically increases if the misordered position is close to the first position. These properties of NDCG perfectly fit our situation, as (i) we aim to find the students with the longest time-on-task and (ii) finding the students with high priority is more important than those with low priority. We repeat all experiments with different random seeds five times and report their averaged NDCG scores.

We apply time-based masking to the dataset called "cutoff". The cutoff simulates a real-world lab session where students' complete editing histories are unavailable until they finish the session. We generalize the cutoff by making the model see only a part of editing operations; for example, $x \leftrightarrow y$ cutoff remains editing operations with timestamps between x and y only. For simplicity, we abbreviate the notation $0 \leftrightarrow y$ cutoff to y cutoff. Various cutoff configurations simulate different situations where a trade-off exists between the timeliness of early detection and the amount of available information from the editing histories. In that sense, we try at least two different cutoff configurations for each experiment.

To train EliRank, we apply the following model configurations. The AST parsing process uses the fixed time interval $s = 15$. The AST node type embedding matrix M^A uses $d_A = 8$. For GNN, we use GCN2 [6] with embedding size $d_G = 32$ and layer count $L = 8$. The graph pooling uses the default settings provided by PyG implementation [9], except that the size of hidden units and output units are reduced to 16. The transformer encoder uses 16 for the embedding size, four heads for multi-head attention, and two feedforward layers. FNN uses two ReLU layers with output sizes 16 and 1, respectively.

6.2 RQ1: Overall Performance

To evaluate the overall performance of EliRank, we measured the performance of three heuristic algorithms that find students in need.

Random algorithm computes ranking score randomly. Since the minimum value of the NDCG metric depends on the dataset, we can get a sense of the model's lowest performance through this algorithm.

EvPerSec algorithm [14] computes ranking score as the frequency of editing operations, giving higher weight to the recent operations. In other words, the ranking score is defined as $r := \sum_{x \in E, x^{ts} \leq T^c} e^{-\lambda(T^c - x^{ts})}$ where E is the editing history, T^c is the cutoff value, and λ is a constant to determine how much to penalize the old operation. We use $\lambda = 0.1$ to follow the original paper. We use this algorithm as the baseline because it is designed to achieve the same goal as ours, prioritizing students in need for tutors.

EvPerSecNeg algorithm computes ranking score as the multiply of -1 to the score from **EvPerSec**. This algorithm models situations in which students frequently experience a pause when they are struggling [26], or their engagement drops.

Table 4 summarizes the NDCG score of EliRank and the three heuristics. In all cases except one, EliRank outperforms all the heuristics with statistical significance ($p < 0.01$ from Welch's t-test). Also, the performance of EliRank shows a tendency for the performance to increase as the length of available editing histories increases, which is consistent with our assumption that fine-grained editing history plays a crucial role in understanding code and students. For the result on P11 and 300s cutoff, EvPerSecNeg shows the best score over EliRank, and we conjecture that there was insufficient data to train EliRank completely. Further investigation on this case is in the **Transferability** section.

6.3 RQ2: Effectiveness of DAST

Table 5: The *NDCG* score for different code representations. The score in bold shows the significantly best score within the same cutoff. $^{*}p < 0.05, ^{**}p < 0.01, ^{***}p < 0.001$ **from Welch's t-test.**

	300s cutoff		600s cutoff		900s cutoff	
ID	AST	DAST	AST	DAST	AST	DAST
P1	0.889	0.893	0.907	0.903	0.915	0.923
P2	0.907	*****0.918**	0.934	****0.949**	0.961	*****0.973**
P3	0.904	0.903	0.925	****0.932**	0.947	0.950
P4	0.885	0.891	0.901	0.902	0.928	****0.945**
P5	0.864	***0.872**	0.877	0.878	0.893	0.888
P6	0.876	0.881	0.888	0.898	0.907	0.913
P7	0.899	*****0.906**	0.943	0.941	0.953	0.954
P8	0.889	***0.898**	0.910	0.911	0.935	0.930
P9	0.842	*****0.858**	0.887	***0.898**	0.902	****0.914**
P10	0.860	****0.866**	0.874	0.881	0.890	0.892
P11	0.868	0.869	0.897	0.901	0.908	****0.919**
P12	0.887	***0.895**	0.914	0.914	0.936	0.933
P13	0.858	*****0.880**	0.880	***0.886**	***0.914**	0.904
P14	0.858	*****0.873**	0.874	0.874	0.879	0.876
P15	0.884	****0.901**	0.916	0.922	0.935	0.944

Table 5 summarizes the results of the ablation study on DAST. The effectiveness of DAST is maximized at the 300s cutoff, and the difference between AST and DAST becomes indistinguishable as the length of available code history increases. This verifies that DAST effectively encodes the fine-grained code change information, especially when there is not enough information from code snapshots. Also, the results emphasize the importance of DAST on our goal, which is the early detection of students in need, since models should keep the required length of editing history as short as possible for early detection.

6.4 RQ3: Cutoff Configuration

Table 6 summarizes the result of different cutoff configurations. We designed two configurations to see two different aspects of the cutoff configuration. The first configuration excludes the cutoff position's effect and examines the cutoff length's effect by comparing the 750 ↔ 900 cutoff with the 0 ↔ 900 cutoff. On the other hand, the second configuration compares the 150 ↔ 300 cutoff with the 750 ↔ 900 cutoff to examine the effect of the cutoff position.

The first results show that the longer the cutoff length, the better the performance of the EliRank. This result also validates our editing history-based approach to finding students in need since the last snapshot of the code alone – the extreme case that the cutoff length becomes zero – shows sub-optimal performance. However, depending on the exercises, the NDCG scores between the two configurations do not show statistical significance. From this, we found the possibility of future improvements in fine-tuning the cutoff length on a per-exercise basis for efficiency while minimizing model performance degradation.

Table 6: The *NDCG* score for different cutoff configurations. The score in bold shows the significantly best score within the same cutoff configuration. $^{*}p < 0.05, ^{**}p < 0.01, ^{***}p < 0.001$ **form Welch's t-test.**

	Fixed Cutoff Position		Fixed Cutoff Length	
ID	750 ↔ 900	0 ↔ 900	150 ↔ 300	750 ↔ 900
P1	0.925	0.923	0.893	*****0.925**
P2	0.966	****0.973**	0.922	*****0.966**
P3	0.944	***0.950**	0.902	*****0.944**
P4	0.940	0.945	0.888	*****0.940**
P5	0.890	0.888	0.874	*****0.890**
P6	0.915	0.913	0.885	*****0.915**
P7	0.941	*****0.954**	0.902	*****0.941**
P8	0.924	0.930	0.894	*****0.924**
P9	0.911	0.914	0.856	*****0.911**
P10	0.883	***0.892**	0.867	****0.883**
P11	0.911	***0.919**	0.865	*****0.911**
P12	0.932	0.933	0.891	*****0.932**
P13	0.899	0.904	0.880	*****0.899**
P14	0.875	0.876	0.872	0.875
P15	0.933	****0.944**	0.901	*****0.933**

The second result shows that EliRank performs better as the cutoff position goes backward. This is natural since the degree of uncertainty decreases as the degree of completion of the code increases. However, for effective early detection of students in need, it is necessary to set the cutoff position close to 0, indicating that the tradeoff between the model's performance and early detection needs to be carefully adjusted.

6.5 RQ4: Transferability

Table 7 shows experimental results for three training strategies to examine the transferability of the model.

Individual strategy trains models per exercise. Each model takes training data and validation data from the selected exercise only. There is no parameter share between the models for different exercises.

Global strategy is a default strategy that trains a single global model with training data from all exercises.

Transfer strategy trains models per exercise. Each model takes training and validation data from all exercises except the selected exercise. This strategy examines zero-shot transfer learning.

In all cases, the global strategy shows the best or no statistical significance to the best. This implies that EliRank, with the global strategy, effectively learns exercise-specific and shared information across all exercises. Also, we found that the effect of the shared information becomes noticeable, especially when the short cutoff

Table 7: The *NDCG* score for the individual (S_{ind}), the global (S_{global}), and the transfer (S_{trans}) strategies. The score in bold shows the best score. The significance is calculated to the strategy of the best score within the same cutoff configuration. $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$ **from Welch's t-test.**

	300s cutoff			600s cutoff		
ID	S_{ind}	S_{global}	S_{trans}	S_{ind}	S_{global}	S_{trans}
P1	**0.882	**0.893**	0.888	0.900	**0.903**	0.901
P2	0.913	**0.918**	*0.914	*0.937	**0.949**	0.948
P3	*0.900	**0.903**	0.901	*0.928	**0.932**	0.930
P4	0.882	**0.891**	0.889	**0.904**	0.902	0.902
P5	**0.862	**0.872**	0.871	**0.864	0.878	**0.879**
P6	*0.868	0.881	**0.884**	*0.885	**0.898**	0.893
P7	0.903	**0.906**	0.901	0.912	0.941	**0.944**
P8	***0.884	**0.898**	0.896	0.908	**0.911**	0.907
P9	*0.843	**0.858**	0.855	*0.884	0.898	**0.899**
P10	0.865	**0.866**	0.866	0.874	**0.881**	0.881
P11	**0.877**	0.869	0.868	**0.903**	0.901	*0.886
P12	0.890	**0.895**	0.891	**0.921**	0.914	0.912
P13	**0.862	**0.880**	0.878	**0.887**	0.886	0.885
P14	**0.858	0.873	**0.874**	0.874	**0.874**	0.872
P15	**0.883	**0.901**	0.898	**0.909	**0.922**	*0.915

configuration is applied, where there is a lack of exercise-specific information.

One remarkable result is that the transfer strategy is comparable to the global strategy in most cases. This result suggests that EliRank can perform even under the cold start situations where unseen programming exercises or programming exercises with few participants are given.

We found one more interesting result in the performance of the individual strategy on P11. Unlike other cases, the individual strategy on P11 shows the best score among other strategies. This result shows that if an exercise has a unique characteristic that does not share with other exercises, the transferability of the model would be limited. Nevertheless, this limitation will be mitigated if the amount of data becomes enough based on the observation that the difference between the individual and the global strategies is statistically non-significant and decreases as the cutoff length increases.

6.6 Limitations

One limitation of our study is that the efficacy of EliRank is indirectly evaluated rather than directly calculating the accuracy of how correctly the model finds students in need. The main reason is that, as mentioned at the beginning of this section, our dataset lacks explicit signals of the need for assistance. Moreover, such explicit signals are not enough to evaluate EliRank because the dataset does not reflect the students who do not "raise their hands" even if they are struggling, and our primary goal is to find them proactively with a data-driven approach. Some of the previous studies [1, 21] define struggling students with predicted exam scores or grades. However, their evaluation objective differs from ours because EliRank aims to find struggling students per exercise, not to find students to be failed at the end of the course.

One possible direction to supplement the evaluation is a qualitative analysis of tutor and student experiences in the tutoring interface using EliRank. However, as investigated in [14], designing such a tutoring interface for monitoring students requires consideration of various design factors and goals that affect the experience of tutors and students. We considered this to be part of future work beyond our focus on the model.

7 CONCLUSION

We present EliRank, a ranking model that takes the first few minutes of fine-grained code editing history and predicts the priority of the students, for early detection of students in need. EliRank adopts GNN, graph pooling, and transformer encoder for its prediction and cooperates with DAST to minimize the loss of keystroke-level code change information. We evaluate EliRank with the real-world dataset consisting of fine-grained code editing histories from 15 programming exercises and 4000+ students in an introduction to programming class at a university. The performance of EliRank is assessed indirectly by measuring the NDCG score of recommended students prioritized by their estimated time-on-task. The result shows that EliRank effectively finds students with the highest priority, showing the possible benefits on tutors by reducing the number of candidates to monitor proactively in large-scale classes. Also, the evaluation showed (i) the effectiveness of DAST, (ii) the potential to control the tradeoff between early detection and the performance of the model, and (iii) the transferability to unseen programming exercises via zero-shot transfer learning.

EliRank currently applies simple models to mainly focus on validating our dataset and the approach instead of maximizing performance. Therefore, as a future direction, we can improve the performance of EliRank by combining more sophisticated machine-learning models. For example, EliRank can incorporate pre-training models such as [13], different GNN and graph pooling methods, or different sequence-based models for better prediction.

Also, our contributions to the dataset, DAST, and EliRank are not limited to a specific task, i.e., predicting the priority of students in need. Instead, these contributions can be easily extended to the broad spectrum of software engineering topics.

8 ACKNOWLEDGMENTS

The authors want to thank Elice for supporting us develop and operate a web platform for dataset collection.

REFERENCES

[1] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. 2015. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the eleventh annual international conference on international computing education research*. 121–130.

[2] Umair Z Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*. 139–150.

[3] Susan A Ambrose, Michael W Bridges, Michele DiPietro, Marsha C Lovett, and Marie K Norman. 2010. *How learning works: Seven research-based principles for*

smart teaching. John Wiley & Sons.

[4] Jinheon Baek, Minki Kang, and Sung Ju Hwang. 2020. Accurate Learning of Graph Representations with Graph Multiset Pooling. In *International Conference on Learning Representations*.

[5] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. 384–392.

[6] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*. PMLR, 1725–1735.

[7] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Wang, Sang Won Lee, Walter S Lasecki, and Steve Oney. 2020. EdCode: Towards Personalized Support at Scale for Remote Assistance in CS Education. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[8] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 970–981.

[9] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[10] Ursula Fuller, Colin G Johnson, Tuukka Ahoniemi, Diana Cukierman, Isidoro Hernán-Losada, Jana Jackova, Essi Lahtinen, Tracy L Lewis, Donna McGee Thompson, Charles Riedesel, et al. 2007. Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin* 39, 4 (2007), 152–170.

[11] Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. 2021. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology* 135 (2021), 106566.

[12] Frank Goldhammer, Johannes Naumann, Annette Stelter, Krisztina Tóth, Heiko Rölke, and Eckhard Klieme. 2014. The time on task effect in reading and problem solving is moderated by task difficulty and skill: Insights from a computer-based large-scale assessment. *Journal of Educational Psychology* 106, 3 (2014), 608.

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*.

[14] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.

[15] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[16] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*. 73–84.

[17] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*. 4171–4186.

[18] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[19] Juho Leinonen, Francisco Enrique Vicente Castro, and Arto Hellas. 2021. Fine-grained versus coarse-grained data for estimating time-on-task in learning programming. In *Proceedings of The 14th International Conference on Educational Data Mining (EDM 2021)*. The International Educational Data Mining Society.

[20] Juho Leinonen, Francisco Enrique Vicente Castro, and Arto Hellas. 2022. Time-on-task metrics for predicting performance. *ACM Inroads* 13, 2 (2022), 42–49.

[21] Soohyun Nam Liao, Daniel Zingaro, Kevin Thai, Christine Alvarado, William G Griswold, and Leo Porter. 2019. A robust machine learning technique to predict low-performing students. *ACM transactions on computing education (TOCE)* 19, 3 (2019), 1–19.

[22] Jessica McBroom, Kalina Yacef, Irena Koprinska, and James R Curran. 2018. A data-driven method for helping teachers improve feedback in computer programming automated tutors. In *International Conference on Artificial Intelligence in Education*. Springer, 324–337.

[23] Sarah Müller, Bianca Bergande, and Philipp Brune. 2018. Robot tutoring: On the feasibility of using cognitive systems as tutors in introductory programming education: A teaching experiment. In *Proceedings of the 3rd European Conference of Software Engineering Education*. 45–49.

[24] John C Nesbit, Olusola O Adesope, Qing Liu, and Wenting Ma. 2014. How effective are intelligent tutoring systems in computer science education?. In *2014 IEEE 14th international conference on advanced learning technologies*. IEEE, 99–103.

[25] Jungkook Park, Yeong Hoon Park, Suin Kim, and Alice Oh. 2017. Eliph: Effective visualization of code history for peer assessment in programming education. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. 458–467.

[26] Raj Shrestha, Juho Leinonen, Albina Zavgorodniaia, Arto Hellas, and John Edwards. 2022. Pausing While Programming: Insights From Keystroke Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 187–198.

[27] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. Context-aware and data-driven feedback generation for programming assignments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 328–340.

[28] Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 59–68.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[30] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. 2008. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*. 1192–1199.

[31] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *IJCAI*.