Automated Identification of Relevant Worked Examples for Programming Problems

Muntasir Hoq[1], Atharva Patil[1], Kamil Akhuseyinoglu[2], Bita Akram[1] and Peter Brusilovsky[2]

[1]North Carolina State University
[2]University of Pittsburgh

Abstract

Novice programmers can greatly benefit from using worked examples demonstrating the implementation of programming concepts that are challenging to them. Although large repositories of effective worked examples have been generated by CS education experts, one main challenge is identifying the most relevant worked example in accordance with the particular programming problem assigned to a student and their unique challenges in understanding and solving the problem. Previous studies have explored similar example recommendation approaches. Our work takes a novel approach by employing deep learning code representation models to extract code vectors, capturing both syntactic and semantic similarities among programming examples. Motivated by the challenge of offering relevant and personalized examples to programming students, our approach focuses on similarity assessment approaches and clustering techniques to identify similar code problems, examples, and challenges. We aim to provide more accurate and contextually relevant recommendations to students based on their individual learning needs. Providing tailored support to students in real-time facilitates better problem-solving strategies and enhances students' learning experiences, contributing to the advancement of programming education.

Keywords

problem-solving support, program examples, code structure, code similarities

1. Introduction

Example-based problem solving is the cornerstone of intelligent tutoring systems (ITSs) within the programming domain [1]. When students encounter difficulties in problem-solving, such systems aim to provide relevant examples to aid in comprehension and resolution. Traditionally, selecting these examples has relied heavily on domain experts, a time-consuming and resource-intensive process, particularly as the volume of learning content expands. However, alternative approaches have emerged, seeking to link problems and examples dynamically without expert intervention. Content-based methodologies, such as keyword-based approaches, analyze surface-level similarities but often lack the depth necessary to discern truly relevant content [2, 3]. In contrast, knowledge-based approaches investigate the semantic understanding of content, offering higher-quality links by focusing on the underlying concepts [4, 5].

The motivation for exploring innovative example selection methodologies arises from recognizing the significant benefits novice programmers can gain from worked examples that illustrate challenging programming concepts. Hosseini et al. [6] demonstrated the engagement and performance benefits of directly connecting worked examples and similar completion problems into a "bundle" by a tool called Program Construction Examples (PCEX). A more recent study [7] demonstrated that semantic similarity between connected problems and examples is one of the keys to better problem-solving performance and persistence achieved when this connection is provided by the domain expert. In cases where worked examples and problems are not explicitly linked, it is essential to provide clear guidance to students, such as recommending semantically similar examples following a failed problem-solving attempt [8]. Despite the availability of extensive repositories of such examples curated by computer science (CS) education experts, a fundamental challenge persists: How to identify the most relevant worked example tailored to each student's specific learning needs and the nuances of the programming problem at hand with a scalable and reliable approach.

In response to this challenge, we aim to develop an automated recommender system to recommend the most relevant problems and examples to students when they face difficulty solving programming problems. We undertake a vector-based approach where we embed problems and examples into vector representations, preserving their structural and semantic information. To this end, we leverage a deep learning code representation model, Subtree-based Neural Network (SANN) [9], to extract nuanced similarities among programming problems and examples. We applied this model to problems and examples available in PCEX [6].

We aim to provide contextually relevant recommendations that enhance students' problem-solving abilities and enrich their learning experiences in programming education. Using the extracted vectors from SANN, we recommend students with similar worked examples for a problem based on vector similarity. To demonstrate the effectiveness of our recommendation system, we evaluated it using Top-N accuracy metrics (N = 1, 3, and 5). This measures how often the correct example, as labeled by experts, appears within the top N recommendations. Additionally, we used clustering techniques such as DBSCAN and hierarchical clustering to group similar problems and examples, aiming to reduce the manual effort required by experts. Our results suggest that this method effectively identifies similar problems and examples, enabling us to provide guidance and support to students facing similar challenges. Using these advanced techniques, we aim to bridge the gap between the vast repository of programming examples and problems and the lack of manual support for selecting resources according to the specific needs of individual students, thus fostering more effective and personalized learning experiences in programming education [10].

CSEDM'24: 8th Educational Data Mining in Computer Science Education Workshop, July 14, 2024, Atlanta, GA
mhoq@ncsu.edu (M. Hoq); aspatil2@ncsu.edu (A. Patil); kaa108@pitt.edu (K. Akhuseyinoglu); bakram@ncsu.edu (B. Akram); peterb@pitt.edu (P. Brusilovsky)

2. Related Work

The concept of automatically connecting similar content items traces back to the pioneering work of Mayes and Kibby in the realm of educational hypertext [2, 3]. Initially, similarity-based navigation centered on keyword-level similarity, but due to its limited quality, this approach has since been supplanted by more robust semantic linking methodologies, often referred to as intelligent linking. One such approach is metadata-based linking, which computes similarity measures across various facets of metadata to generate higher-quality links [11].

In recent years, the focus has shifted towards ontology-based linking, particularly within the hypermedia research community. Ontology-based linking involves indexing documents with ontology terms and then leveraging ontological structures to identify similar documents [5, 12]. Although early investigations primarily focused on hypermedia applications, the educational domain saw a subsequent adoption of ontology-based linking methodologies [13].

In the programming domain, similarity assessment has mainly relied on content-level information. For example, Gross et al. [14] linked Java programming contents based on the similarity of their Abstract Syntax Trees (ASTs), which encompassed the entire body of examples and problems. However, this approach may overlook finer-grained similarities in smaller code fragments [15]. Recent approaches have explored concept-based similarity methodologies [16], that is, representing examples and problems as vectors of domain concepts and measuring similarity between these vectors. Further work attempted to calculate ontology-based similarity metrics for programming items [8].

With the advent of automated program representation techniques that use deep learning methodologies [9], the extraction of syntactic and semantic structural information from programming code snippets has gained traction. These techniques offer the potential to alleviate the reliance on experts for designing isomorphic problem-example pairs, instead enabling the discovery of relevant examples within learning materials. While such similarity approaches hold promise across a range of code-related programming problems [17, 9], our study focuses on code comprehension problems, wherein students are tasked with predicting the output or final value of variables from a given program code rather than on program composition problems requiring code writing tasks.

3. Dataset

In this study, we used a Python programming dataset sourced from the PCEX system [6]. PCEX offers online access to working code examples and small completion problems referred to as "challenges". To increase learners' motivation and improve overall learning outcomes, problems and examples are further organized into bundles by domain experts, who group together problems and examples that target similar programming constructs and patterns. This combination approach was validated in previous research [6, 7], demonstrating its value across various metrics and stressing the importance of connecting learning and assessment on the level of specific programming patterns in addition to their traditional integration on the level of broader course topics.

The PCEX dataset comprises 123 programming code problems and examples that span 13 topics, including Variables and Operations, If-Else statements, Boolean Expressions, For Loops, and Nested Loops. The problems and examples in the dataset are organized into 52 bundles, with an average of 4 bundles per topic. Bundles start with a single fully worked out example and are followed by 1 to 3 similar problems. On average, each bundle has 1.35 problems. We used the current content organization in PCEX, which represents expert knowledge, as the gold standard for the evaluation of content recommendation approaches [18]. Figures 1 and 2 show two program examples from the same bundle under the Variable and Operations topic. Figure 3 shows another example of a different bundle within the same topic to show the difference in the bundle structures.

```python
#Step 1: Read the integer
text = input("Enter an integer: ")
num = int(text)

#Step 2: Determine whether the integer is even or odd
if num % 2 == 0 :
    print("The integer is even.")
else :
    print("The integer is odd.")
```

Figure 1: Example 1 from the same bundle

```python
#Step 1: Read the integer
text = input("Enter an integer: ")
num = int(text)

#Step 2: Determine whether the integer is positive, negative, or zero
if num > 0 :
    print("The integer is positive.")
elif num < 0 :
    print("The integer is negative.")
else :
    print("The integer is zero.")
```

Figure 2: Example 2 from the same bundle

```python
#Step 1: Read the temperature for today and yesterday
text = input("Enter the yesterday's temperature: ")
yesterday = float(text)
text = input("Enter the today's temperature: ")
today = float(text)

#Step 2: Warn the user about the changes in the temperature
if  today < yesterday :
    print("It is getting colder!")
else :
    if  today > yesterday :
        print("It is getting warmer!")
    else :
        print("Temperature is the same as yesterday!")
```

Figure 3: Example 3 from a different bundle

4. Methodology

In this study, we used the Subtree-based Attention Neural Network (SANN) [9] to encode programs into vector representations. We computed the cosine similarity between these vectors to recommend the closest examples for a given problem. To further analyze and group similar problems and examples, we employed clustering techniques such as DBSCAN and Hierarchical clustering.

SANN is our primary model for encoding programs in

vector format. It has demonstrated its efficiency in capturing both syntactic and semantic information from programs in an interpretable manner and understanding the intricate code structure [9, 17, 19]. SANN operates by encoding the source code into vector representations using subtrees extracted from the Abstract Syntax Tree (AST) representation of the code. These subtrees undergo a two-way embedding process in which each subtree and its constituent nodes are individually embedded. The resulting embeddings are then merged into a single embedded vector. Subsequently, the embedded vectors from both approaches are concatenated and passed through a time-distributed, fully connected layer, generating subtree vectors that incorporate both node-level and subtree-level information.

After the generation of subtree vectors, an attention neural network is employed to condense all subtree vectors into a singular source code vector. The attention mechanism assigns scalar weights to each subtree vector, facilitating the aggregation of all subtree vectors into a weighted average. These weights are determined through a normalized inner product between each subtree vector and a global attention vector, followed by applying a softmax function to ensure that the weights sum up to 1. The resulting weighted average of subtree vectors, as determined by the attention mechanism, encapsulates the entire source code snippet. The SANN model leverages attention weights to prioritize the most important subtrees when generating the source code vector. We recursively extract all subtrees from an AST, ensuring comprehensive coverage of the code structure during the encoding process.

Following the extraction of code vectors, we calculated cosine similarity to find the closest example for a given problem for the recommendation. Furthermore, we utilized various clustering techniques, including DBSCAN and Hierarchical clustering, to group similar problems and examples. DBSCAN is adept at identifying clusters of varying shapes and sizes while being robust to noise, while hierarchical clustering provides insights into the clustering structure through dendrogram analysis. Using these techniques, we aim to comprehensively explore the similarity structure within our dataset and facilitate the identification of cohesive groups of programming problems and examples.

5. Experiments and Results

5.1. Code Vector Extraction and Example Recommendation

We employed the Python AST parser[1] to parse Python programming code into ASTs. For SANN training, we partitioned our dataset into 80% training data and 20% testing data. During the splitting process, we ensured that no bundle was excluded from the training set to retain all the diverse structural variations for comprehensive training. The embedding size for both subtree-based and node-based embeddings was set to 64, chosen from {64, 128, and 256}. Consequently, each source code vector was of size 128. Throughout the model training phase, we employed the Adamax optimizer [9] with a default learning rate of 0.001 to learn the weights of the matrices. The batch size was set to 32, and the maximum number of epochs was capped at 200, with an early stopping patience of 20, to prevent overfitting of the model.

[1]https://docs.python.org/3/library/ast.html

Table 1
Top-N accuracy for recommending worked examples

Top-N	Accuracy (%)
Top-1	70.97
Top-3	83.10
Top-5	87.32

The dataset has problems/challenges and examples bundled together based on similarity (these are called bundles), and different bundles are combined under different topics by the experts. Therefore, the dataset shows a hierarchy of topics and bundles. Each topic has some bundles, and each bundle has some similar challenges and examples. If a student faces difficulty in a problem, an example from the same bundle will be recommended. We trained the SANN model using only the topic information for challenges and examples, intentionally omitting any bundle information. Although bundles encapsulate more detailed and granular-level information about the program structures, our objective was for SANN to learn this granular insight exclusively from the superficial and abstract topic information. We aim to enable SANN to generalize effectively across diverse program structures by training on topics alone and trying to reconstruct underlying bundles based on their similarity in program pattern and structure to evaluate their effectiveness compared to expert-identified bundles. There are 13 topics and 52 bundles in the dataset. After the training, we tested the trained model on the test data to predict the associated topics. SANN showed a testing accuracy of 88%. Afterward, we extract the source code vectors for the problems and examples from SANN further study. Finally, we investigated the effectiveness of these vectors in forming groups of similar examples and problems that can serve as a recommending tool. We calculated the cosine similarity to find the closest example for a given problem. If the closest example is also from the same original expert-identified bundle as the challenge, the recommended example is correct. We calculated the Top-N accuracy, where N = 1, 3, and 5 as stated in Table 1. The experimental result suggests that our recommendation can effectively find similar worked examples for a given problem when a student is facing difficulty. However, we speculate that this accuracy can be improved with a bigger dataset to train SANN since the current dataset has only 123 challenges and examples, where the average number of examples per problem is only 0.73. We want to investigate the impact of dataset size on improving performance in the future.

We further hypothesize that since the bundles represent very similar challenges and examples, the corresponding vectors should show these similarities by being closer to the programs of the same bundle than others. The same hypothesis is applicable to topics. However, the topics contain slightly less similar challenges and examples. Hence, the vectors of a topic should be close to each other but not as close as those of a bundle. According to our hypothesis, the vectors in these bundles and topics should show patterns in their tightness. Tightness refers to the average distance between points of a bundle or topic. To calculate the tightness, we used the expert labels from the dataset as the gold standard to show the effectiveness of our method and verify the hypothesis. For each topic/bundle, first, we calculated the pairwise distances for all the points within it. Then, we calculated the mean of these pairwise distances, which is

the tightness within the vectors of the topic/bundle. Figure 4 shows the scatter plot of the bundles using PCA = 2.

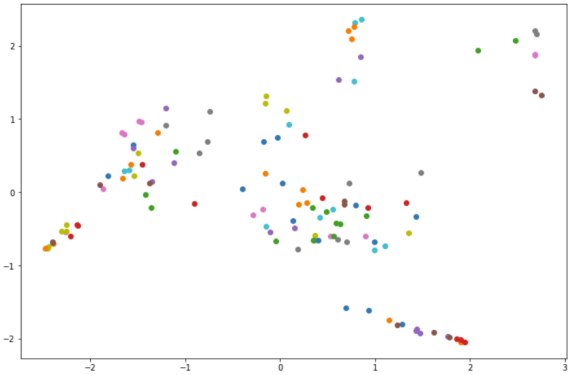

Figure 4: Bundle clusters

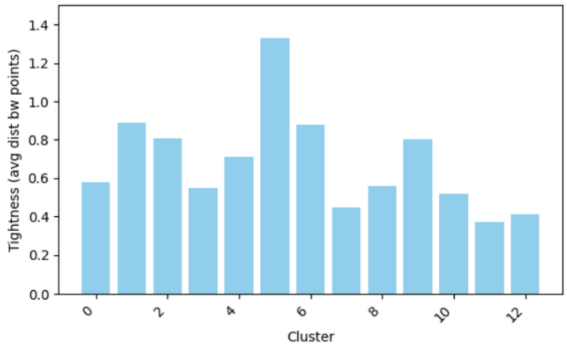

Figure 5: Average tightness of topics

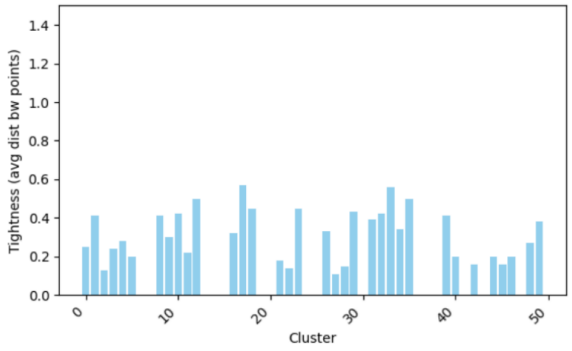

Figure 6: Average tightness of bundles

To verify our hypothesis, we calculated the degree of tightness for (1) vectors of the same bundle, (2) vectors of the same topic, and (3) all vectors in the data set (entire course). Figure 5 shows the topic-level tightness, and Figure 6 shows the bundle-level tightness. Here, we can see that bundles have lower distances, whereas topics have higher distances. We plotted the mean degree of tightness for topics, bundles, and the whole dataset in Figure 7 to get a clearer comparative view. For topics and bundles, the average tightness measures for all individual topics and bundles were calculated. The average tightness of bundles was found to be 0.4 units, and the average tightness of topics was found to be 0.8. This implies that points of a bundle are much closer to each other than those in a topic. Finally, the average distance between all dataset samples was found to be 2.7 units. This result suggests that samples belonging to the same bundle are semantically very similar to each other; samples belonging to the same topic might have more variations than bundles but still more similar than any other sample from other topics in the course.

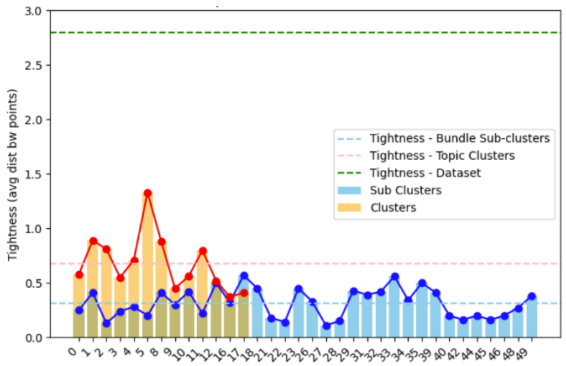

Figure 7: Average tightness of topics and bundles

5.2. Clustering Similar Examples

We investigated the effectiveness of multiple clustering approaches in identifying bundles of similar problems and examples. Firstly, we employed DBSCAN clustering for topics, given its capability to handle irregularly shaped clusters when the number of clusters (topics) is unknown and different structured problems and examples can be set under the same topic. Setting the epsilon value to 0.85 and the minimum points parameter to 2, we successfully identified 13 distinct clusters based on topics, with only 2 points classified as noise. This is because we assume each topic cluster must have at least 2 points, and if some point is not in the vicinity of any other, it is best to consider it as noise rather than part of some cluster. Figure 8 shows the scatter plot visualization that highlights the nonspherical nature of the clusters, indicating their irregular shapes.

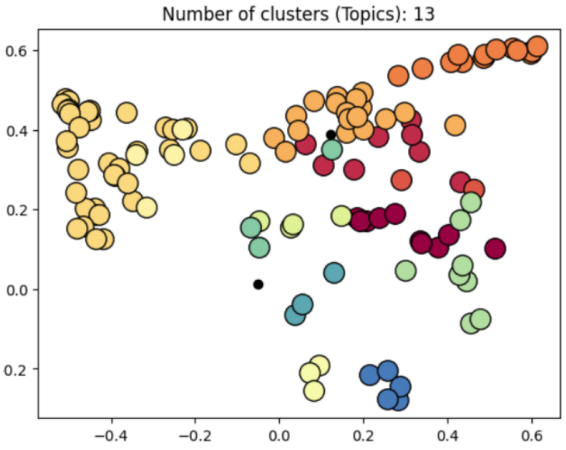

Figure 8: Topic clusters using DBSCAN

We calculated the accuracy of the topic clustering using DBSCAN by determining a clustering error, which was assessed by comparing the assigned clusters to gold standard

clusters based on predefined topics. Specifically, we calculated how many items were incorrectly assigned to clusters compared to their actual topic labels. The clustering error demonstrated an average of 11.69% (std dev 0.15) over all the problems and examples. The highest clustering error for a topic was 44%. The topic with 44% error was the topic "Strings." This can be considered an outlier because the code for string programs is likely similar to other topics where some string operations are also required. It is important to note that three topics, including "For Loops," "Nested Loops," and "Lists," were assigned to the same cluster. We found that these topics are very similar in structure and have overlapping, i.e., using loops to traverse a list, hierarchical For Loops in Nested Loops.

Hierarchical clustering was utilized for bundles inside topics as it allows for the exploration of hierarchical structures within the data and accommodates scenarios where the number of clusters is uncertain. DBSCAN may not be ideal for this because the plotted points for bundles are unlikely to have irregular shapes, since problems and examples inside a bundle tend to be the most similar. Hierarchical clustering starts by treating each sample as a separate cluster. Then, it repeatedly executes the following two steps: (i) identify the two clusters that are closest together, and (ii) merge the two most similar clusters. This iterative process continues until all the clusters have merged together.

The dendrogram from hierarchical clustering illustrated that samples sharing similar bundles and topics clustered closely together, with their parent clusters predominantly aligning with their respective topics. In addition, we assessed the closest sample for each item, categorizing them based on bundle name and topic similarity. Based on this closest sample data, we evaluated the number of items for which their closest sample had (1) the same bundle name, (2) a different bundle name but the same topic, and (3) a different bundle name and topic. As evident in Figure 9, 43.9% had their closest sample from the same bundle and 30.9% had their closest pair from the same topic. However, there were 25.2% samples whose closest pair was from a different topic and bundle. This result suggests that samples of the same topic are closer and contained within the same local region. In addition, samples belonging to the same bundle are even closer to each other. However, discrepancies between the clustering results and expert labels emerged when problems and examples involved multiple topics or multiple bundles, for example, the use of loops in For Loops, Nested Loops, List, and Strings.

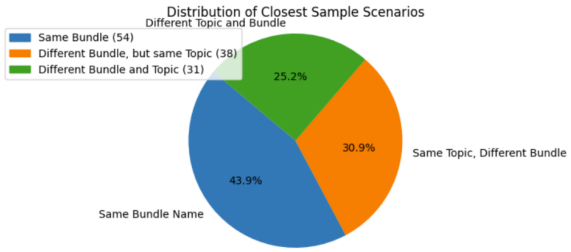

Figure 9: Hierarchical clustering summary

6. Discussion

In this study, we tried to address the long-standing challenge of dynamically recommending relevant programming examples tailored to individual student needs within the context of computer science (CS) education. Our approach centered on leveraging the Subtree-based Neural Network (SANN) model to extract nuanced syntactic and semantic similarities among programming examples, thus facilitating the identification of analogous examples crucial for problem-solving support. In this study, SANN was trained only on the topic information of the examples. However, the dataset used also contains bundle information, where similar problems and examples are bundled together under a topic. We used topic-level information about the problems and examples to get deeper structural insight using SANN, which helps to identify similar worked examples for struggling students. Using the extracted code vectors, we recommend students with worked examples for a given problem based on vector similarity. The experiment suggests that the recommendation has an accuracy of 70.97%, 83.10%, and 87.32% with the Top-1, Top-3, and Top-5 recommendations, respectively.

In addition, we show the effectiveness of these vectors by showing the tightness of each topic and bundle in the course. The results suggest that the bundles represent very similar problems and examples, as reflected by the proximity of their corresponding vectors. In contrast, the topics contain multiple bundles with slightly less similar problems and examples. Consequently, the vectors within a topic are close to each other but not as tightly clustered as those within a bundle. We further employed advanced clustering techniques, including DBSCAN and hierarchical clustering, to effectively group similar programming problems and examples and alleviate the expert effort in bundling similar problems and examples. This outcome highlights the initial effectiveness of our approach in organizing and understanding the structural and semantic relationships inherent in programming education datasets. However, with the limited training data (the current dataset has only 123 problems and examples, where the average number of examples per problem is only 0.73), our clustering and performance did not fully align with the expert labels. We hypothesize that minor structural changes and overlapping topics in smaller problems and examples could be captured more accurately with a larger dataset. Exploring this possibility is an interesting direction for future research.

The significance of our study lies in addressing a key challenge in CS education: identifying relevant and contextually appropriate programming examples [1]. By offering a methodological framework for dynamically recommending personalized examples, our study provides a scalable solution to the resource-intensive process of example selection traditionally reliant on domain experts. Our approach effectively connects the extensive collection of programming examples with the unique needs of individual students, improving programming education by promoting more efficient and personalized learning experiences [6, 20].

7. Limitations and Future Work

There are a few limitations that need to be addressed in this study. Firstly, SANN was trained on the topics associated with each problem and examples labeled by experts. This current setup limits our ability to use a vast corpus of

worked examples and programming problems that are not labeled with topics. In the future, we want to eliminate this limitation by training the SANN model in a topic-agnostic way. We propose training the model not on explicit topic information but instead on the underlying code structure using an encoder-decoder architecture. In this approach, the encoder would process the source code to generate a latent representation that captures the structural and semantic nuances of the code. The decoder would then reconstruct the code from this latent representation. This unsupervised learning method aims to enable the model to understand and encode the intricate structure of the code more effectively, leading to better generalization and more accurate recommendations based on structural similarities rather than predefined topic labels.

Additionally, when we explored clustering techniques, we observed that some worked examples are similar, though they are from different topics. It happens because some topics overlap with previously learned topics. For example, when dealing with List problems, they might require knowledge of loops. In such cases, in the future, we might consider sub-categories of these bundles to recommend previous topics when necessary based on the difficulty progression of a student. For example, if a student struggles with traversing a list due to difficulties using loops, they would benefit from revisiting similar examples that focus on loops from previously covered topics.

Another future direction of this work is to make the recommendations more personalized based on student knowledge. We want to track students' learning at various stages of the course and incorporate that information in recommending examples for the current problems they face. The tracing of student learning can also be on a topic level. If a student faces difficulty in a particular topic, this can be important information along with the problem code structure for the recommender system. In addition, struggling with the same topic can also act as an alarm for instructors, indicating that a student needs personalized intervention and support. We also intend to add some baselines from the literature to do a study to show the comparative effectiveness of our framework in the future.

8. Conclusion

In this study, we used the Subtree-based Neural Network (SANN) model to recommend relevant programming examples tailored to individual student needs in computer science (CS) education. Through clustering techniques, including DBSCAN and hierarchical clustering, we effectively organized the structural and semantic relationships of problems and examples to guide the recommendation of similar practices to programming students. Our approach offers a scalable solution to the resource-intensive process of example selection, providing contextually appropriate learning resources tailored to individual student needs.

References

[1] P. Brusilovsky, C. Peylo, Adaptive and intelligent web-based educational systems, International Journal of Artificial Intelligence in Education 13 (2003) 156–169.

[2] M. Kibby, J. Mayes, Towards intelligent hypertext, Hypertext: Theory into Practice (1989) 164–172.

[3] J. T. Mayes, M. R. Kibby, H. Watson, Strathtutor©: The development and evaluation of a learning-by-browsing system on the macintosh, Computers & Education 12 (1988) 221–229.

[4] K. R. Koedinger, A. T. Corbett, C. Perfetti, The knowledge-learning-instruction framework: Bridging the science-practice chasm to enhance robust student learning, Cognitive Science 36 (2012) 757–798.

[5] L. Carr, W. Hall, S. Bechhofer, C. Goble, Conceptual linking: ontology-based open hypermedia, in: Proceedings of the 10th International Conference on World Wide Web, 2001, pp. 334–342.

[6] R. Hosseini, K. Akhuseyinoglu, P. Brusilovsky, L. Malmi, K. Pollari-Malmi, C. Schunn, T. Sirkiä, Improving engagement in program construction examples for learning python programming, International Journal of Artificial Intelligence in Education 30 (2020) 299–336.

[7] K. Akhuseyinoglu, A. Klašnja-Milićević, P. Brusilovsky, The impact of connecting worked examples and completion problems for introductory programming practice, in: European Conference on Technology Enhanced Learning (EC-TEL 2024), Lecture Notes in Computer Science, Springer International Publishing, 2024.

[8] R. Hosseini, P. Brusilovsky, A study of concept-based similarity approaches for recommending program examples, New Review of Hypermedia and Multimedia 23 (2017) 161–188.

[9] M. Hoq, S. R. Chilla, M. Ahmadi Ranjbar, P. Brusilovsky, B. Akram, SANN: Programming code representation using attention neural network with optimized subtree extraction, in: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, 2023, pp. 783–792.

[10] K. Muldner, J. Jennings, V. Chiarelli, A review of worked examples in programming activities, ACM Transactions on Computing Education 23 (2022) 1–35.

[11] D. Tudhope, C. Taylor, Navigation via similarity: automatic linking based on semantic closeness, Information Processing & Management 33 (1997) 233–242.

[12] M. Crampes, S. Ranwez, Ontology-supported and ontology-driven conceptual navigation on the world wide web, in: Proceedings of the 11th ACM on Hypertext and Hypermedia, 2000, pp. 191–199.

[13] P. Dolog, N. Henze, W. Nejdl, Logic-based open hypermedia for the semantic web, in: Proceedings of the International Workshop on Hypermedia and the Semantic Web, Hypertext 2003 Conference, 2003.

[14] S. Gross, B. Mokbel, B. Hammer, N. Pinkwart, How to select an example? a comparison of selection strategies in example-based learning, in: Proceedings of the Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Springer, 2014, pp. 340–347.

[15] G. Weber, A. Mollenberg, Elm-pe: A knowledge-based programming environment for learning lisp. (1994).

[16] R. Hosseini, P. Brusilovsky, Example-based problem solving support using concept analysis of programming content, in: Proceedings of the Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Springer, 2014, pp. 683–685.

[17] M. Hoq, Y. Shi, J. Leinonen, D. Babalola, C. Lynch, T. Price, B. Akram, Detecting chatgpt-generated code submissions in a cs1 course using machine learning models, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, 2024,

pp. 526–532.
[18] A. J. Sabet, I. Alpizar-Chacon, J. Barria-Pineda, P. Brusilovsky, S. Sosnovsky, S. Sosnovsky, P. Brusilovsky, A. Lan, et al., Enriching intelligent textbooks with interactivity: When smart content allocation goes wrong, in: Proceedings of the 4th International Workshop on Intelligent Textbooks, volume 3192, 2022.
[19] M. Hoq, J. Vandenberg, B. Mott, J. Lester, N. Norouzi, B. Akram, Towards attention-based automatic misconception identification in introductory programming courses, in: Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2, 2024, pp. 1680–1681.
[20] M. Hoq, P. Brusilovsky, B. Akram, Analysis of an explainable student performance prediction model in an introductory programming course, in: Proceedings of the 16th International Conference on Educational Data Mining, 2023, pp. 79–90.