AI-assisted Programming Question Generation:

Constructing Semantic Networks of Programming Knowledge

by Local Knowledge Graph and Abstract Syntax Tree

by

Cheng-Yu Chung

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2022 by the
Graduate Supervisory Committee:

Ihan Hsiao, Chair
Kurt VanLehn
Shaghayegh Sahebi
Srividya Bansal

ARIZONA STATE UNIVERSITY

May 2022

ABSTRACT

Persistent self-assessment is the key to proficiency in computer programming. The process involves distributed practice of code tracing and writing skills which encompasses a large amount of training that is tailored for the student's learning condition. It requires the instructor to efficiently manage the learning resource and diligently generate related programming questions for the student. However, programming question generation (PQG) is not an easy job. The instructor has to organize heterogeneous types of resources, i.e., conceptual programming concepts and procedural programming rules. S/he also has to carefully align the learning goals with the design of questions in regard to the topic relevance and complexity. Although numerous educational technologies like learning management systems (LMS) have been adopted across levels of programming learning, PQG is still largely based on the demanding creation task performed by the instructor without advanced technological support. To fill this gap, I propose a knowledge-based PQG model that aims to help the instructor generate new programming questions and expand existing assessment items. The PQG model is designed to transform conceptual and procedural programming knowledge from textbooks into a semantic network model by the Local Knowledge Graph (LKG) and the Abstract Syntax Tree (AST). For a given question, the model can generate a set of new questions by the associated LKG/AST semantic structures. I used the model to compare instructor-made questions from 9 undergraduate programming courses and textbook questions, which showed that the instructor-made questions had much simpler complexity than the textbook ones. The analysis also revealed the difference in topic distributions between the two question sets. A classification analysis further showed that

the complexity of questions was correlated with student performance. To evaluate the performance of PQG, a group of experienced instructors from introductory programming courses was recruited. The result showed that the machine-generated questions were semantically similar to the instructor-generated questions. The questions also received significantly positive feedback regarding the topic relevance and extensibility. Overall, this work demonstrates a feasible PQG model that sheds light on AI-assisted PQG for the future development of intelligent authoring tools for programming learning.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

1.1 Terminology 6

2 RELATED WORK 7

2.1 Automatic Question Generation and Key Information Identification 7

2.2 Semantic Modeling for Knowledge Extraction 9

2.3 The Transformation from the Key Information into Question Forms 11

2.4 Automatic Question Generation for Educational Technologies 12

2.5 Programming Question Generation 13

3 METHODOLOGY 16

3.1 Modeling Conceptual Programming Knowledge 16

3.2 Modeling Procedural Programming Knowledge 18

3.3 Automatic Question Generation Process 19

4 ANALYZING THE CORRELATION BETWEEN QUESTION KNOWLEDGE GRAPHS AND STUDENT PERFORMANCE 22

4.1 Extraction of Semantic Triples and Construction of LKG 23

4.2 Comparing Instructor-made Questions with Textbook Questions 25

4.3 The Relationship Between Performance and Network Characteristics 29

CHAPTER Page

4.4 Summary of the Correlation Between Question Knowledge Graphs and Student Performance 34

5 EVALUATING THE UTILITY OF MACHINE-GENERATED PROGRAMMING QUESTIONS 35

5.1 User Study Design 35

5.2 Data Collection 39

5.3 The Participants' Teaching Experience 39

5.4 High Similarity Between the Instructor-generated Questions and Machine-generated Questions 42

5.5 Significantly Positive Ratings on the Question Utility 44

5.6 The PQG Model Helps Generate General Code-writing Questions and Complex Code-tracing Questions....46

5.7 Either Generated Questions or Generated Code Examples Are Helpful for the Instructors 48

6 DISCUSSION 52

6.1 The Design Focusing on Extensibility of Methodology and Inclusion of Topics 52

6.2 Significantly Positive Ratings from the Experienced Instructors....53

7 CONCLUSIONS 56

7.1 Limitations and Future Work 56

7.2 Contributions from My Dissertation 58

REFERENCES 60

LIST OF TABLES

Table Page

1 Basic Characteristic of the LKG Extracted from the Question Datasets............27

2 Classification Performance of High- and Low-error Labels..............................34

3 The Statistics of One-sample T-tests for the Four Rating Variables 46

4 The One-sample Statistics of the Multilevel Analysis (Preferred Question Types and Ratings)..47

5 The One-sample Statistics of the Multilevel Analysis (Expected PQG Support and Ratings)..51

LIST OF FIGURES

Figure Page

1 A High-level Overview of the PQG Model with Input/Output Examples 21

2 Examples of LKG Composition 24

3 Comparing LKG from the Three Datasets 26

4 The Number of High-error Questions and Low-error Questions in CUCC 31

5. Comparing the Number of High- and Low-error Questions in Different Component Groups 32

6 An Example Template of the Code-aware Model..................................38

7 An Example Template of the Context-aware Model 38

8 Embedding Similarity Between the Input, Machine-generated, and Instructor-generated Questions 43

9 The Distribution of Ratings to the Four Utility Variables, Topic Relevance, Extensibility of Topics, Complexity, and the Instructor's Need 45

10 Multilevel Analysis of Ratings and Preferred Question Types 47

11 Multilevel Analysis of Ratings and PQG Expectations 49

CHAPTER 1

INTRODUCTION

Learning computer programming requires rigorous practices of programming code tracing and writing (Chung et al., 2020; Chung & Hsiao, 2020; Czerkawski & Lyman, 2015). One of the most critical challenges in mastering programming is to integrate and apply conceptual knowledge (*know-what*) and procedural knowledge (*know-how*). The complex nature of computer programming is similar to the adjacent fields, such as Math and Engineering, where the complexity results in a larger number of resources to manage than in the other fields that are only embodied in homogeneous types of knowledge.

Fortunately, in the modern days of programming education, educational technologies have widely reshaped how instructors organize learning resources. For instance, learning management systems (LMS) and adaptive textbooks (Chau et al., 2020) can provide a digitalized hub of resources that allows the instructor to efficiently query and browse the content. An LMS can also help the instructor promptly deliver learning resources such as reading materials, worked code examples, and code-tracing/code-writing assessments to the student. By continuously supplying self-assessments over time (Chung & Hsiao, 2020), the instructor can manage the student's learning condition and then adapt the instructions accordingly, thereby achieving the benefits of personalized learning. Numerous studies have shown how such a technology shapes the student's learning experience in programming learning (Weintrop et al., 2016).

Nevertheless, comprehensive resource management of programming learning should include not only the content organization but also the content generation. In my work, I found that a successful self-assessment learning environment requires a sufficient supply of questions that are tailored for different learning conditions. Therefore, I focus on researching methods to facilitate programming question generation (PQG). This PQG process is never as simple as transforming a descriptive sentence into an interrogative form. The craft of programming questions requires both the mastery of programming knowledge and an efficient question generation routine, i.e., deciding the learning goals, gathering related code examples, adapting the complexity to the level of the students, and transforming questions into appropriate formats. The process can quickly become a tedious and time-consuming task when a large number of questions are needed.

Researchers have developed different tools to support the PQG process. For example, the parameterized question generation (Brusilovsky & Sosnovsky, 2005; Hsiao et al., 2009) provides a straightforward solution that efficiently reduces the cost of authoring code examples. The process "parameterizes" code examples by substituting the variables or values inside, which effectively transforms the examples into templates of questions. In practice, the instructor only needs to prepare a small number of variable cases to create a large collection of questions. Research in related fields of study also sheds light on how to support the PQG process. For example, (L. Zhang & VanLehn, 2016) propose a knowledge-based question generation model that automatically generates questions for introductory biology classes. The model uses an existing, established biology ontology

and templates to generate questions, and the evaluation showed that the machine-generated questions worked well in helping students to learn.

However, these methods are limited in terms of the question types they can address. Although the parameterized question generation can help generate many code examples from a small number of templates, it is unable to generate concept questions that go beyond the predefined templates. The knowledge-based question generation requires an established, ready-to-use ontology as the source of information. Most of such ontologies capture high-level categories of programming concepts but not the procedural knowledge that is necessary for code tracing/writing questions. In addition, since the input ontology is usually static, it is relatively difficult to adapt the knowledge-based generation for different contexts (i.e., the scope of learning content). To facilitate the PQG process that fully involves the integration of conceptual and procedural programming knowledge, I believe that an intelligent PQG model based on a modern AI approach and a flexible knowledge base is needed.

This work proposes and examines a PQG model that captures conceptual and procedural knowledge by a Local Knowledge Graph (LKG) and Abstract Syntax Trees (AST). The knowledge is transformed into a semantic network, by which the model can query related code examples and the embedded concepts in the form of semantic triples. The triples are then used to generate programming questions by matching the semantic structures against predefined question templates. Overall, this work is guided by the following research questions:

1. How does a PQG model capture the conceptual and implementation programming knowledge?
2. Given an input programming question, how does the PQG model query the most relevant programming examples? What similarity index can help rank the outcome?
3. How does the model generate a set of programming questions from the most relevant programming examples?
4. What are programming instructors' opinions about the utility of the PQG model? Are the generated questions aligned with the instructors' needs in teaching introductory programming courses?

I first used the model to analyze instructor-made questions, which were collected from 9 undergraduate programming courses, and textbook questions from selected programming textbooks and question banks. The analysis showed that the two sets of questions had much different topic distributions. The instructor-made questions revealed a more concentrated and simpler semantic network than the textbook questions. A classification analysis further revealed a correlation between the complexity of question content and student performance. Following these results, a group of experienced instructors was recruited to evaluate the quality of machine-generated questions. The result showed that my model was able to generate questions that were semantically similar to instructor-made questions. Furthermore, the questions received significantly positive ratings from the instructors in terms of the topic relevance and extensibility.

This dissertation is organized as follows: Chapter 2 reviews research works about automatic question generation and the essence of PQG; the methodology, including the design of the proposed PQG model and the evaluation task, is described in Chapter 3; an analysis of the relationship between LKG complexity and student performance is reported in Chapter 4, which is followed by the result of the model evaluation in Chapter 5; the answers to the research questions are collectively discussed in Chapter 6; finally, this work, limitations, and potential future work are concluded in Chapter 7.

1.1 Terminology

Following the proposed research question, I accordingly define the following terms: conceptual and procedural knowledge, topic, and difficulty. In this work, I hypothesize that conceptual and procedural knowledge of questions is certain learning content that is (or can be) explicitly described by question texts. I say that a concept is a set of words or phrases that represent certain knowledge related to the learning content. Conceptual knowledge focuses on the remembering of programming concepts, and procedural knowledge focuses on the implementation of programming concepts in program code. For example, the variable declaration by itself is conceptual knowledge, and the declaration statement like "int x = 10;" is procedural knowledge. Concepts may be grouped into the same categories according to their relationship in terms of the order of learning content in textbooks. Such a category is called a "topic" and is defined as an instructor-made label associated with questions on an online self-assessment platform. For example, topics may include "array", "variable", "data types", etc. One question is only associated with one topic label. Also, I define the difficulty of questions as the statistical difficulty measured by the average error rate of first attempts on the platform.

CHAPTER 2

RELATED WORK

2.1 Automatic Question Generation and Key Information Identification

Question generation (QG) refers to the process where a computer model automatically extracts key information from a given context and then produces interrogative sentences that address the information (Kurdi et al., 2020). Research of QG has long evolved as one of the natural language processing (NLP) applications in the field of computational linguistics. The major goal of QG is to answer the two fundamental questions about the context: "what to ask" and "how to ask", where the former is concerned with the identification of key information and the latter is the transformation of question forms.

The identification of key information can be viewed as a conventional NLP labeling (or tagging) task (Vedula et al., 2020), where the major goal is to help the machine accurately process the meaning (explicit or implicit) of a given text (context). The labeling task usually focuses on one aspect of the text, therefore having many different implementations. For example, part-of-speech (PoS) is one labeling task that focuses on the syntactic meaning of words. The PoS focuses on word classes and tags each word in the text into a lexical category of the grammar, e.g., "NN" for nominal nouns, and "V" for verbs. For another example, named entity recognition (NER) is a labeling task that focuses on the semantic meaning of words. The NER uses the regularity of information to predict the information carried by the text. Such information is typically predictable across different texts and usually addresses key information from the text. For instance,

the Stanford NER (Finkel et al., 2005) can accurately identify three categories of words, including organization, person, and location. These taggers provide an imperative way for the machine to automatically extract the key information out of the text and transform it into corresponding questions.

The identification of key information can also be treated as a machine learning, classification task. With enough training data, a computer model can automatically learn the relationship between the input context and output key information. For instance, (Chau et al., 2020) demonstrates a supervised, feature-based method for concept extraction from digital textbooks. The researchers first recruited a group of experts to manually annotate key concepts in a selected textbook. Afterward, they examined a list of linguistic and statistical features and external resources that might be related to the key concepts. Finally, they built a classification model to automatically extract key concepts from the textbook. The result showed that their model reached state-of-the-art performance in a benchmark with similar models.

For another example, (Willis et al., 2019) proposes an end-to-end system that generates both questions and their answers from a given context paragraph (which effectively combines the identification of key information and the transformation of question forms). The system was based on an encoder-decoder architecture and trained by a public question-answering dataset. The researchers showed that the system was able to perform better than traditional key phrase predictors, even though it was only trained on part of the dataset. Moreover, they also recruited a group of educational experts to assess the

quality of the generated outcome. The result showed that the system was able to generate educationally meaningful question and answer pairs.

Although machine-learning-based methods may provide better performance than simple rule-based methods in key information extraction, it should be noted that a model trained by one dataset may not generalize for all kinds of learning goals, especially when the learning goal involves certain knowledge that is not explicitly expressed in the text or is expressed in multiple modalities. For example, programming knowledge consists of two modalities: conceptual knowledge and procedural knowledge. Conceptual knowledge is usually expressed in declarative sentences written in a natural language.
But the procedural knowledge may be expressed in both the natural language and a programming language. This characteristic makes the identification of key information in programming knowledge becomes much more challenging, therefore inspiring the development of my work.

2.2 Semantic Modeling for Knowledge Extraction

In addition to the labeling approach, another way to identify key information is using an existing knowledge base (KB) or ontology (Y. Zhang et al., 2006). A knowledge base provides a structured knowledge representation that consists of key concepts and their dependency on them. A KB can be constructed manually or automatically. Automatic KB construction is similar to semantic networks where nodes and edges represent concepts and their relationships, respectively (Gillam et al., 2005). For example, (Veremyev et al.,

2019) demonstrates two approaches to constructing semantic networks, one based on large text corpora and the other based on existing lexical databases. The researchers further analyzed the characteristics of the two networks and revealed that the machine-built networks can provide richer information on the usage and meanings of words. A similar approach from (Fan et al., 2019) builds knowledge graphs by focusing on verb-argument structures in the text. The researchers showed that the network can help compress information and reduce redundancy for indexing multiple documents.

Researchers have tried to use knowledge bases in analyzing programming documents and functionalities (Ancona et al., 2012; Y. Zhang et al., 2006). The relationship between concepts and the concepts themselves may already provide the key information of learning content. For example, in object-oriented programming, the relationship between objects in a class can be described as "The objects are instances of the class". This simple relationship can be used to develop questions like "What is the class of the objects?" or "How many objects are in the class?".

The KB-based approach has been used in many QG applications due to the easy-to-process semantic structure. The structure can be seen as an alternative format of labels that provides an extra layer of information for QG. For example, (L. Zhang & VanLehn, 2016) developed a QG method for introductory biology classes. They obtained an existing knowledge base of biology and defined a set of question schema for different semantic triples (i.e., data entities in the format of subject-predicate-object expressions). Compared to the labeling method, the semantic-based, KB-based method may extract key

information that is more relevant to the domain knowledge when the input knowledge base is accurate. To enhance the accuracy, researchers have tried to encode question answers into the construction process by text spans, context position, or separate encoders (Pan et al., 2019).

2.3 The Transformation from the Key Information into Question Forms

Once the key information is identified from the input context, the next step of QG is to convert it into a question form. The transformation of question forms is a generation task that extends a piece of key information into a full interrogative sentence. There are two common transformation approaches, template-/rule-based methods and statistical methods (Kurdi et al., 2020). A template-/rule-based method fills in key information into predefined question structures by rules or templates. This method is useful especially when the key information is clearly defined or identified. For example, in programming learning, a code-tracing question that asks the student to trace the code logic and find out the output can be converted into a question template by substituting the variables or their values (Guerra et al., 2014). Similarly, a formula in math can be used as a template where the instructor only needs to substitute its content with prepared cases.

Instead of predefined templates or rules, a statistical method automatically learns the transformation schema from some training data. The process can be viewed as the sequence-to-sequence (seq2seq) machine translation that translates a given text into the target language of the question (Bahdanau et al., 2014). A statistical method can even

integrate the identification of key information and the transformation of question forms into one process. For example, (Du et al., 2017) demonstrated a seq2seq model based on the recurrent neural network (RNN) encoder-decoder architecture. They investigated how to encode sentence-level and paragraph-level key information to generate questions. In a related study, (Jia et al., 2020) developed a model to also encode the dependency of questions on the answer in the input context. Their RNN-based model was able to generate exam-style questions including cloze and general questions.

2.4 Automatic Question Generation for Educational Technologies

The delivery of educational content has been revolutionized by remote learning tools. There has been much research that investigates the benefit of remote learning tools in the literature of learning science. For example, previous research has shown that distributed practice is an effective learning method that can benefit learners of different ages and abilities (Dunlosky et al., 2013). Similar to the well-known spacing effect and the testing effect in learning science, the distributed practice effect suggests that retention of information increases when the learner practices retrieving the information in multiple spaced-out practice sessions (Delaney et al., 2010; Dunlosky et al., 2013). Empirical studies have also shown that the time interval between practice sessions is an important factor in the learning outcome (Chung & Hsiao, 2021b; Qiu et al., 2011).

Based on the distributed practice effect, there have been numerous educational tools that support distributed practice. For example, QuizIT is an online practice platform that

focuses on distributed practice and self-assessment in the field of computing education (Alzaid et al., 2017). The platform periodically releases practice questions that help students keep track of their learning progress. Several quantitative studies of the usage of the platform have shown that high-performance students tended to use the platform actively, persistently, and regularly (Chung et al., 2020; Chung & Hsiao, 2020). On the other hand, low-performance students might change their practice patterns over time, and this kind of behavior might be correlated with lower learning outcomes. I believe that when such remote learning tools become part of both online and in-person instructions, the supply of practice content becomes a critical issue for instructors to consider in the instructional design due to several concerns. For example, the question generation has to cover different concepts in the learning content to ensure that students have an adequate amount of practice. The format of questions has to be diverse enough to engage students in the learning content. In addition, questions have to be adaptively supplied to a student according to his/her performance. These concerns have been widely investigated in the literature of ITS that specializes in the recommendation of existing content; nevertheless, I believe there is also a need to research them from the perspective of content generation, by which tools for computer-aided QG can be devised to support the instructional design process.

2.5 Programming Question Generation

QG for educational purposes has been explored in different fields of study, including mathematics, physics, biology, and linguistics. A recent survey in the literature has

shown that automatic QG is especially popular in the field of language learning (Kurdi et al., 2020). The learning content has a close relationship to the grammar, syntax, and semantics of natural languages, which makes the field an obvious subject of QG. In addition, the need for standardized exams also engages more research to focus on the field (Kurdi et al., 2020). In addition to formal learning, much research on QG has also focused on the development of general QA systems for training and instructions, e.g., (Ruan et al., 2019).

Programming learning nowadays has been reshaped by the use of educational technology that emphasizes personalized learning (Czerkawski & Lyman, 2015). For example, (Lu & Hsiao, 2017) uses adaptive learning techniques to personalize the search result for programming learners. Another example, (Alzaid & Hsiao, 2019) focuses on bite-size practices that distribute student practice over time to achieve the distributed learning effect. These educational technologies are all based on the abundant supply of high-quality materials that can not only fulfill the quantity in need but also the quality of personalized learning which requires a sufficient coverage of concepts.

However, despite the need for more questions and the popularity of programming learning, little research has been focused on QG for computer programming learning. An early work from (Brusilovsky & Sosnovsky, 2005) focuses on individualized, parameterized exercises for the C programming language. Similar to mathematical formulas, coding questions in programming learning can be transformed into fixed templates with part of them replaced by variables. The variables can then be replaced by

predefined cases (i.e., values in worked examples) to generate a large number of new questions. Another research from (Zhong et al., 2017) focuses on the SQL programming language which has a structure similar to natural languages. They propose a deep neural network for translating the natural-language content to corresponding SQL queries for questions. It is also worth noting the work from (Denny et al., 2008) where the researchers approached the same issue of question insufficiency by crowdsourcing. They developed a system to require students to participate in the construction of programming questions. Their analysis of the system usage showed that it not only helped the supply of programming questions but also encouraged the development of students' cognitive skills. Overall, I believe there is still a need for further research in the QG for programming learning.

CHAPTER 3

METHODOLOGY

3.1 Modeling Conceptual Programming Knowledge

A programming question usually includes one or multiple sentences that address the concept to learn, and the concept is usually sourced from some descriptions in a textbook. I hypothesize that, for any question, the conceptual programming knowledge to learn, and the action required to approach it, are explicitly addressed by the words in the question content. For example, a common type of programming question is the input/output question like "what is the output of this program?" where the concept of the question can be fully understood as "the output is what". This sentence can be decomposed into the relationship between the two objects, "the output" and "what", and the verb "is". By focusing on verbs and objects that are related to computer programming, I believe this decomposition can help me model conceptual programming knowledge in descriptions in textbooks.

The decomposition is a version of the semantic role labeling (SRL) that identifies the predicate (verbs and their arguments) in a sentence. The SRL has been widely used in natural language processing (NLP) to build a model that can interpret the syntactic structure of natural languages. It also helps index textual information by compact formats like "(subject)-verb-object" triples. For example, the example above can be rewritten into a triple like ("the output"/ARG, "is"/VERB, "what"/ARG). There have been many automatic SRL systems that can automatically identify semantic labels of words or

phrases by rule-based algorithms (e.g., part-of-speech tags, PoS) or supervised learning models that are trained by large-scale NLP corpora like the PropBank corpus (Palmer et al., 2005).

Labeling words by grammar may not be sufficient for the machine to understand natural language. For example, a complex sentence may have an overlapping structure and include more than one predicate. The sentence can be decomposed into multiple semantic triples, each of which reveals different aspects of the meaning. Extending the concept of SRL, researchers have proposed Open Information Extraction (OIE) that considers both SRL and propositions asserted by sentences (Yates et al., 2007). An OIE model can decompose, for example, "computers connected to the Internet can communicate with each other" into two predicates, "(computers connected to the Internet; can; communicate with each other)" and "(computers; connected; to the Internet)". These predicates represent two aspects of the input. The first one focuses on the communication between devices, and the second one focuses on the state of the connection. Compared to the conventional SRL, the OIE can extract more information about the intent of a given sentence. This characteristic may help the machine to better process the meaning of a complex sentence, therefore helping me model different programming concepts described in a textbook. This work uses the OIE model from (Stanovsky et al., 2018) to extract semantic triples from the descriptions around code examples in a textbook, Think Java (Downey & Mayfield, 2019). To aid the query of related programming concepts, this work also builds a semantic network of the triples by following the Local Knowledge Graph (LKG) approach (Fan et al., 2019). The original LKG builds a semantic network

by treating subjects as nodes and their actions (verbs) as edges. In the context of PQG, I believe that verbs are also important messages because they hint at how knowledge is addressed or approached. Therefore, this work builds an LKG by treating both subjects and verbs as nodes and adding an edge if any two nodes are mentioned in one sentence.

3.2 Modeling Procedural Programming Knowledge

Programming languages are defined according to well-structured grammar and formal languages. This characteristic ensures that program code can be efficiently parsed into binary machine code by a compiler according to the syntactic structure. The Abstract syntax tree (AST) is an alternative representation of program code that specifically focuses on the syntactic structure. In an AST, nodes represent executable statements or declarations. For example, in an AST of Java code, the node "ClassOrInterfaceDeclaration" represents an entry point of Java class definition, and the node "VariableDeclarator" represents a statement that declares a new variable and its initializer. Although AST does not necessarily link to the runtime nature of computer programs (i.e., references to external libraries or actual flow of data), it provides a convenient way for programmers to parse and represent the semantics of computer programs.

Programming questions are usually accompanied by snippets of program code that can be represented by AST. One benefit of AST is its readability and explainability. AST nodes concisely represent functions of computer programs and are associated with their

arguments. They also provide a way to edit and modify the program. These properties make AST suitable for PQG. First, a PQG model can capture the meaning of program code by AST that to some extent represents the target programming knowledge in a question. Second, AST can help the model edit an existing programming question to produce new versions of the question. In this work, I choose to use an AST parser to extract program code in question texts and map them to certain programming concepts that are related to programming learning. For example, the node "VariableDeclarator" can be an entry point that links to the concept of data types, arrays, or loop statements. Overall, AST provides an interface for my PQG model to search or extend the meaning of an input question, which makes the model able to generate questions that transcend the limit of translation-based QG models.

3.3 Automatic Question Generation Process

This work follows a template-based QG method to transform the key information into different questions. There are two main reasons for this decision. First, the template-based method does not require an extensive amount of training data for a model to learn the pattern of questions. This is an advantage to this work due to the lack of publicly available datasets of programming questions that can be considered the gold standard of PQG. Second, the template-based method provides the flexibility to involve the instructor's knowledge in the design of the model. This makes it easier for the model to capture common types of questions that are used in the instructions.

For a given input question, the PQG algorithm starts with keyword and code extraction by regular expressions. If the code is available, this step transforms them into AST nodes; otherwise, this step matches the question text against a list of textbook keywords. This mechanism ensures that the algorithm can capture as much information as possible from the question text to generate questions. Next, if the AST nodes are available, the PQG algorithm makes a query by the nodes and extracts related code examples from the model by the similarity between the input and the examples. If the nodes are not available, the keywords are used instead. This step produces a set of potentially related code examples that come along with their LKG triples. To find out which examples are the most relevant to the input, the algorithm uses the Tversky index (which is a general version of the Jaccard index) to rank the examples. Afterward, the algorithm selects the top K examples and transforms them into questions according to the templates.

I defined several templates for AST nodes to generate coding questions. For example, a "what is" question is made for nodes about variables, data types, outputs, or "print" statements (e.g., "what is the value and the data type of X?", "what is the output of this program?", "how do you convert the data type from X to Y?"). The templates about the runtime logic of the nodes focus on the use of methods/classes and flow control, e.g., "what is the purpose of the method X?", "how many times the loop will run?", etc. Overall, the templates mainly cover questions about the input/output and the syntax. I also defined a set of templates for LKG triples to address the concepts from the input. These templates rely on the SRL (e.g., "ARG0", "VERB", "ARG1", etc.) from the triples themselves. The labels generally indicate where the triple items should be in a sentence,

and my algorithm plugs in those items in the templates according to the grammar. For example, a triple ("display", "a variable") can be used to make a question "how do you display a variable?" Compared to coding questions, such concept questions theoretically address a broader range of knowledge because the LKG triples are built on top of descriptions/explanations of the code example which may include the implicit semantics beyond the explicit structure of code. To summarize the mechanism of my PQG model, Figure 1 provides a high-level overview of actual input and output.

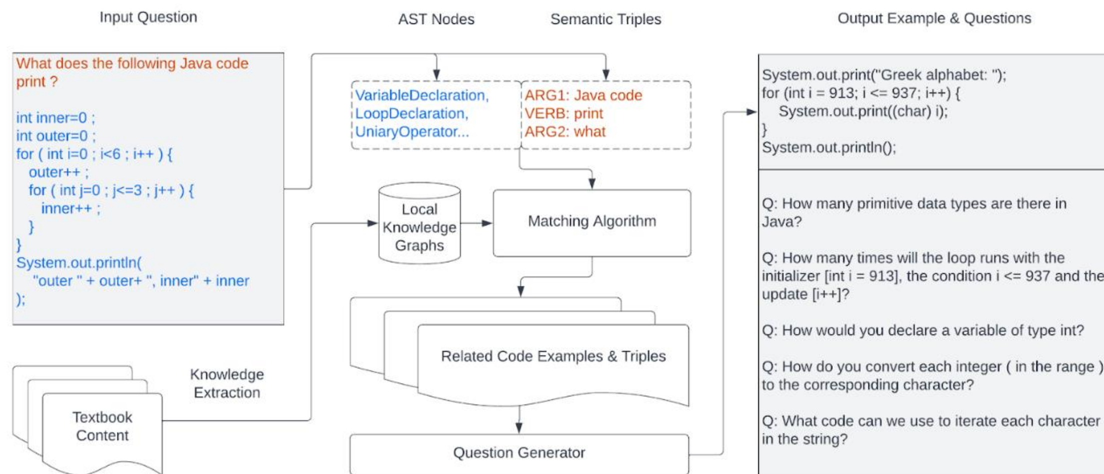

Figure 1. A High-level Overview of the PQG Model with Input/Output Examples.

CHAPTER 4

ANALYZING THE CORRELATION BETWEEN QUESTION KNOWLEDGE GRAPHS AND STUDENT PERFORMANCE

This part of my work consists of two experiments: (1) comparing the difference between textbook questions and instructor-made ones, and (2) finding out the relationship between student performance and characteristics of LKG. The first experiment is aimed to show how instructor-made questions are different from existing questions in textbooks in terms of the complexity of LKG. The second experiment is aimed to provide evidence for the relationship between student performance and the complexity of questions.

I collected three datasets of questions, one of which represent instructor-made questions and the other textbook questions. The first dataset, *QuizIT,* was instructor-made multiple-choice questions (MCQ) collected from QuizIT (Alzaid et al., 2017) that was used in 9 undergraduate courses about entry-level Java programming over 3 years. The instructor might make questions from scratch or select/modify the existing questions in a question bank. Because an MCQ from QuizIT might have the major content of questions in the options (e.g., "Which of the following is correct?"), I chose to concatenate all question texts with their answer options to ensure that sufficient details were included in the dataset. The number of unique questions was 779 (355 when counting unique question text only). The second dataset of practice questions, *Textbook,* was collected from a free online textbook, "*Introduction to Programming Using Java, Eighth Edition*"[1]. I collected

[1] https://math.hws.edu/javanotes/

7 chapters of content that overlapped with the 9 undergraduate courses on QuizIT. In total, there were 163 practice questions, most of which were free-text questions. The third dataset, *QBank*, was collected from a question bank accompanied by the textbook used in the entry-level Java programming course. After processing, I collected 225 practice questions that were also in the format of MCQ.

The student performance was represented by the statistical first-attempt error rate on QuizIT. This dataset consisted of records from 570 students who contributed 14,534 first attempts. More details about this performance data are described in the analysis below.

4.1 Extraction of Semantic Triples and Construction of LKG

This work adopts a supervised OIE model developed by (Stanovsky et al., 2018) to automatically extract propositions from questions. The model is based on an RNN architecture trained to iteratively extract propositions from a given sequence of words. The output of OIE is a set of triples that represent the propositions in a given question. Each triple always consists of one verb (VERB) and the associated arguments (ARG) with index numbers that suggest their semantic roles. The labeling schema follows the PropBank corpus (Palmer et al., 2005) where ARG0 denotes the agent (subject) of the verb, ARG1 denotes the patient (direct object) of the verb, and ARG2 denotes the instrument (or the entity given to ARG1). Higher orders of arguments (e.g., ARG3, ARG4, etc.) and modifiers (ARGM) were omitted in the analysis due to their rarity in the dataset.

After triples were extracted by the OIE model, I similarly constructed an LKG to the work from (Fan et al., 2019) where the researchers used subjects/objects as nodes and verbs as edges. The only difference was that instead of using verbs as edges, I made verbs also nodes of the network and built edges by the subject-verb-object relationship. One hypothetical benefit of this construction is that individual propositions of questions stay strongly connected and can be identified by the existence of closed triplets (triangles) as shown in Figure 2. Although this construction can potentially separate propositions from the same question into multiple components of networks, it makes it easier to analyze closely related propositions by network characteristics like clustering coefficients. All the three question datasets were processed in the same way and transformed into three separate LKG as shown in Figure 3. The processing was implemented mainly by the networkx library and the Gephi visualization tool.

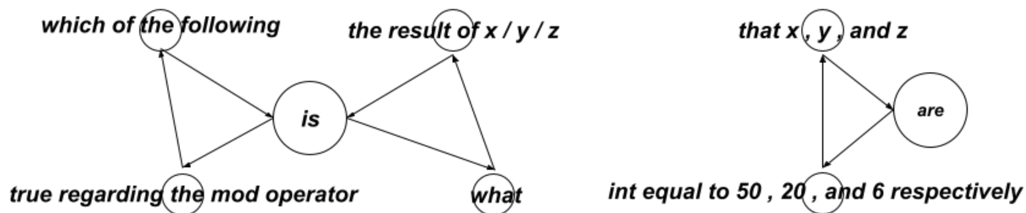

Figure 2. Examples of LKG Composition. The network was constructed using two questions: "Which of the following is true regarding the mod operator, %?" and "Assume that x, y, and z are all int equal to 50, 20, and 6 respectively. What is the result of x/y/z in the following program?". One proposition was extracted from the former question, and two were from the latter question. The propositions that shared the same verb were linked together in one component.

4.2 Comparing Instructor-made Questions with Textbook Questions

Instructor-made Questions Are Less Complex. As shown in Figure 3, the three LKGs had a similar structure (one large component and many small components) but different textures. First, I noticed that the number of questions was not correlated with the size of the networks. The largest network was from QBank, which had 1025 nodes and 1492 edges. The Textbook network was the second largest and had 883 nodes and 1349 edges. The smallest network was from QuizIT, which had only 401 nodes and 573 edges. The average degree for QuizIT was 2.91, 3.06 for Textbook, and 2.91 for QBank. Considering the input data of these networks, the result suggests that the number of questions was not positively correlated with the size of LKG. There are several possible reasons behind this. First, instructor-made questions on QuizIT might be less complex in terms of the content. The instructor might lean toward short and easy questions and use simple languages or words to describe the question because the target audience was mainly novice students in computer programming. However, the questions from Textbook and QBank appeared to be more complex and used more diverse vocabulary because the targeted audience was general readers for all levels of learners (even though the content was generally for beginners of Java programming).

Figure 3. Comparing LKG from the Three Datasets (left: QuizIT; middle: Textbook; right: QBank).

Practice Questions from Different Sources Share Similar Question Structures: what-is and how-would-you-do. Further analysis of network characteristics (Table 1) showed several common and different features of the networks. First, the questions from all three datasets seemed to have similar question compositions. The largest degree nodes were all is/V, you/ARG0, what/ARG1, and what/ARG2. This suggests that many questions were mostly "*what-is*" questions that asked for definitions or "*how-would-you-do*" questions that asked about how to implement some functions in computer programming. Despite this resemblance, it is worth noting that the questions from Textbook and QBank had higher variability than QuizIT. The degree of is/V was 162 for Textbook and 146 for QBank, but it was only 89 for QuizIT. These values again showed that the instructor-made questions in QuizIT might consist of a more selected vocabulary or address limited usage of words.

Table 1. Basic Characteristics of the LKG Extracted from the Question Datasets.

	QuizIT	**Textbook**	**QBank**
Number of Questions	779	163	255
Number of Nodes	462	883	2436
Number of Edges	672	1349	1492
Average Degree	2.91	3.06	2.91
Largest-degree V, ARG0, ARG1, ARG2	(is/V, 89) (you/ARG0, 26) (what/ARG1, 27) (what/ARG2, 15)	(is/V, 162), (you/ARG0, 40) (what/ARG1, 55) (what/ARG2, 54)	(is/V, 146) (you/ARG0, 64) (what/ARG1, 14) (what/ARG2, 10)
The average degree of V, ARG0, ARG1, ARG2	4.83/2.92/2.33/2.26	5.55/3.28/2.24/2.45	5.09/3.17/2.21/2.16
Number of Connected Components	53	74	113
Size of the Largest 5 Connected Components	240, 12, 12, 12, 11	626, 11, 9, 9, 8	601, 27, 15, 10, 10
Global Clustering Coefficient (Transitivity)	0.10	0.05	0.07
Local Clustering Coefficient (Average over Nodes)	0.73	0.72	0.64

Textbook Questions Have a Higher Diversity and Complexity Than Instructor-made Questions. All the networks had many connected components on the circumference that was separate from the major component in the center (as shown in Table 2). In my

construction of the LKG, propositions using the same verbs or arguments would reside in the same connected component. A component that was separate from the others suggested that the underlying propositions used a different set of verbs/arguments. Following this interpretation, the number of connected components became an indication of the diversity of concepts because different concepts might be addressed by different verbs/arguments, therefore resulting in different formats of questions. The largest number of connected components was QBank with 113 components. The Textbook had 74, and QuizIT had 53. This result suggests that questions of QBank might have a higher diversity of concepts than the others. In contrast to this, QuizIT, even though it had the greatest number of questions, resulted in the lowest diversity of concepts. This could be due to the smaller vocabulary used by the instructors when they addressed concepts or that the instructors tended to make questions in similar and simple formats.

Finally, I investigated the clustering coefficient of networks. The clustering coefficient is the ratio of close triplets to open ones. The value of clustering coefficients suggests how closely connected the nodes in a network are. There are two common clustering coefficients: the global clustering coefficient measures the whole network as one (i.e., the total number of closed triplets over the total number of open triplets); the local clustering coefficient measures individual nodes (i.e., for each node N, the number of closed triplets including N over the number of open triplets including N). The clustering coefficient can help me understand how commonly a set of verbs/arguments is reused in the question set. The higher the value is, the more likely the input questions reiterate the same set of verbs/arguments.

For all three networks, I computed the global clustering coefficient and the average of the local clustering coefficient. The result showed that QuizIT had the highest global clustering coefficient and local clustering coefficient. Adhering to the findings from the preceding analysis, this result also suggests that questions in QuizIT were probably simpler than in the other two datasets. The questions might reiterate the same or similar sets of question structures in terms of the verbs/arguments. It should be noted that the finding does not suggest the quality of instructor-made questions in QuizIT or that they could not achieve their intended educational purposes. It only suggests that the questions probably had lower complexity in terms of the forms of questions or the coverage of concepts than the Textbook and QBank questions.

4.3 The Relationship Between Performance and Network Characteristics

Observed Relationship Between the Question Complexity and Student Performance. To understand the relationship between student performance and the characteristics of LKG, I collected the usage data on QuizIT in the courses and computed the first-attempt error rate of the questions. 570 students contributed 14,534 first attempts. The average number of first attempts was 17.27 (SD=19.59), which suggests that some questions were practiced much more than the others. The average error rate of questions was 0.45 (SD=0.31), which suggests that the student performance was moderate and did not lean toward extremely good or poor performance. I grouped questions into "high-error" and

"low-error" based on the average error rate, considering the goal was to find out the difference between difficult and easy questions.

Since the question contents were decomposed into semantic triples and mapped on an LKG, the scope of a mapped area on the network might suggest the concepts that were addressed by the question. To capture the scope of the mapped area, I first summarized a list of unique connected components in the LKG. Then, for each question, I computed all unique connected components it involved (which was a set that I called a "combination of unique connected components", CUCC). Afterward, I summarized how many high-error questions and low-error ones were in each CUCC as shown in Figure 4.

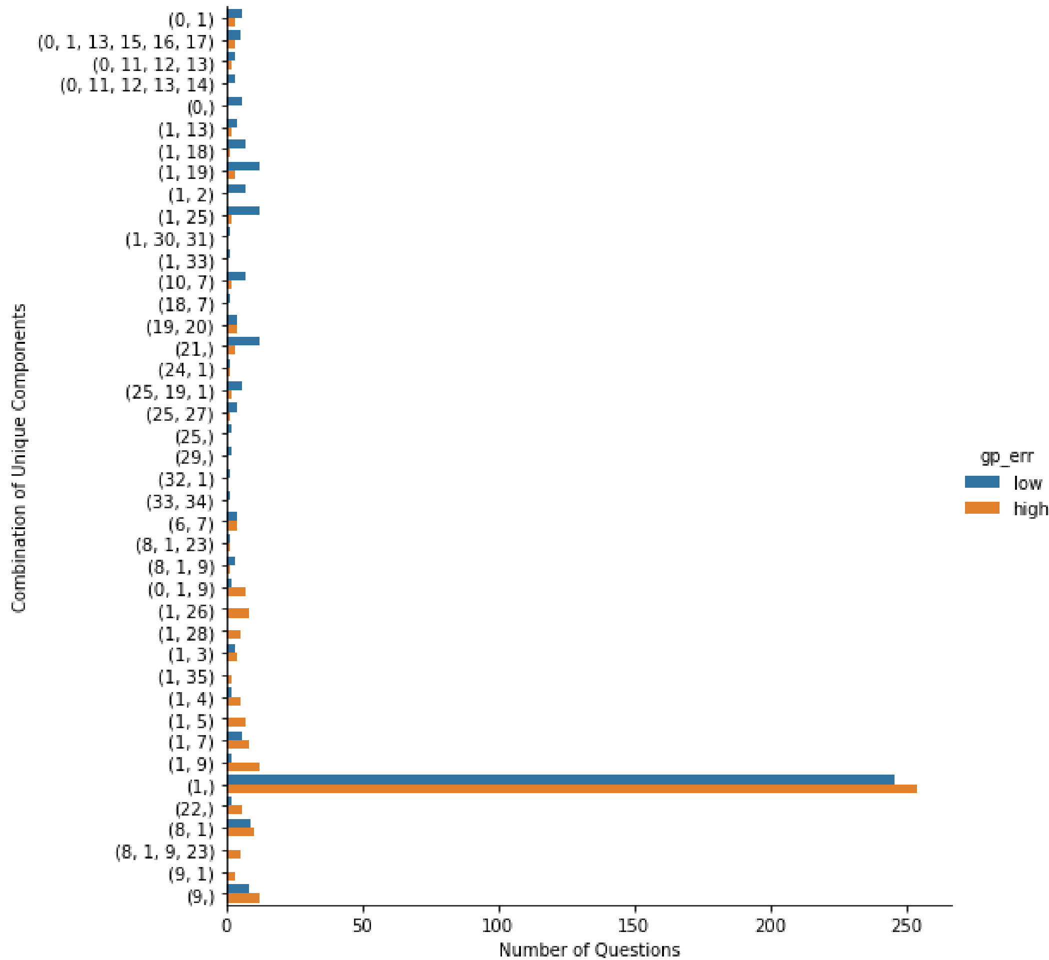

Figure 4. The Number of High-error Questions and Low-error Questions in CUCC. The numbers in a CUCC are unique ID numbers for connected components in the network.

The result showed that there was a clear correlation between the CUCC and the error rate. There was a set of CUCC that had more low-error questions than high-error ones and vice versa. In addition, there was one CUCC that included only component 1 (C1) which had the highest number of questions. This result suggests two aspects of the questions in QuizIT. First, many questions in QuizIT involved only verbs/arguments from C1, which was confirmed as the largest component in the network. This finding adhered to the network characteristics analysis: the instructor-made questions from QuizIT had low

complexity. Second, the CUCC were correlated with the error rate, and some of them only appeared in high- or low-error questions. This finding illustrated that the questions with certain CUCCs were quite extreme, either more difficult or way easier to answer.

I further grouped questions into three categories by the CUCC: cucc_high, cucc_low, and cucc_gen. *cucc_high* includes the CUCC that had more high-error questions than the low-error ones and vice versa for *cucc_low* (Figure 5). The category *cucc_gen* contains C1 only because it had significantly more questions than the other CUCC. A chi-square test showed that both cucc_high ($x^2=28.12$, $p=0.00$) and cucc_low ($x^2=43.45$, $p=0.00$) had significant differences in the number of high- and low-error questions. No difference was found in cucc_gen. This result confirms that the CUCC was somehow correlated with student performance. In other words, this characteristic of LKG can be potentially used to index/categorize the difficulty of questions.

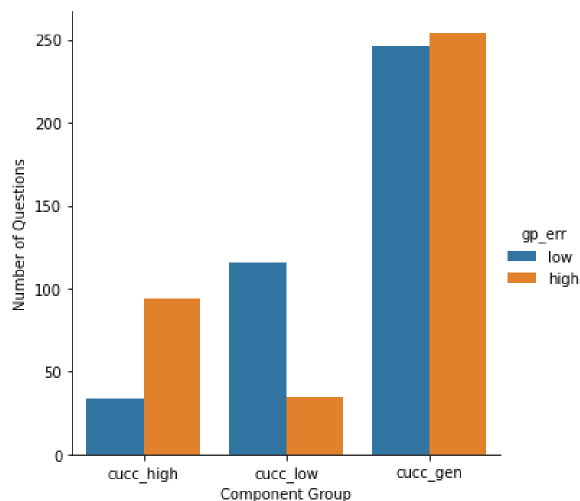

Figure 5. Comparing the Number of High- and Low-error Questions in Different Component Groups. A chi-square test showed that there was a significant difference in cucc_high ($x^2=28.12$, $p=0.00$) and cucc_low ($x^2=43.45$, $p=0.00$).

Question Complexity as Significant Predictors of Student Performance. Finally, I conducted a classification analysis to evaluate the classification performance of the CUCC and other network characteristics. The Random Forest classifier with 10-fold cross-validation was used to benchmark different configurations of features. The result showed that a model using one-hot vectors of the CUCC alone was able to retrieve most high-error questions (f1=0.67, precision=0.54, recall=0.90, accuracy=0.58); however, its precision was mediocre. After several iterations of experiments, I found two network characteristics that helped reach the best performance: the out-degree of verbs (ODV) and the betweenness of argument (BTA). The classification reached f1=0.72, precision=0.75, recall=0.70, and accuracy=0.73 (Table 2). Although the recall was lower than the classification with the CUCC only, these features seemed to find a reasonable balance. This result suggests that the ODV and the BTA might be indirect indicators of the CUCC. Hypothetically, a small component had low ODV and high BTA; a large component had high ODV and low BTA. Although the current state of analysis was not able to examine this hypothesis, the result from the classification analysis can help develop a measure to evaluate the complexity of questions.

Table 2. Classification Performance of High- and Low-error Labels

Features	Precision	Recall	F1-score	Accuracy
One-hot Vectors of Components	0.54	0.90	0.67	0.58
Out-degree of Verbs + Betweenness of Arguments	0.75	0.70	0.72	0.73
Combined	0.73	0.70	0.71	0.72

4.4 Summary of the Correlation Between Question Knowledge Graphs and Student Performance

This chapter is aimed to investigate the characteristics of instructor-made question content by an explainable model from two aspects: its difference from textbook questions and the correlation with student performance. I adopted a semantic network model, local knowledge graph (LKG), which is built by interwoven verb-arguments semantic triples. Comparing one dataset of instructor-made questions from introductory programming courses with two datasets of textbook questions, the analysis showed that instructor-made questions had less diversity of concepts and simpler formats than the textbook questions. I also found a correlation between the coverage of concepts and the average error rate of questions. The direction of this correlation remains unclear and still requires further analysis by manual labeling of text meanings; nevertheless, this work contributes to the understanding of instructors' question generation process by an explainable data model and paves a road to the future development of computer-aided question generation tools.

CHAPTER 5

EVALUATING THE UTILITY OF MACHINE-GENERATED PROGRAMMING QUESTIONS

5.1 User Study Design

To evaluate the performance of the PQG model, I designed a user study that aimed to collect feedback from instructors who have experience in teaching introductory programming courses. The user study consists of two parts: a survey of teaching experience and a task of question evaluation. A participant is expected to spend around 1 hour to finish the whole study.

The survey asks for the participant's teaching experience including the number of years of teaching experience, challenges that s/he encounters in question generation, the type of programming questions that s/he values the most when teaching the course, what kind of support or information s/he expects to obtain from a PQG tool, and how likely s/he would use a PQG tool in production.

In the evaluation task, the participant is given 12 sets of input questions and generated questions. The input questions were selected from a pool of difficult questions according to their statistical difficulty (i.e., the error rate) on a self-assessment platform (Alzaid et al., 2017). I reviewed and selected questions by the following criteria:

- The question must not have any typos.

- The question must address a unique concept but not a duplicate question among the set.
- The answer must be correct and not ambiguous (some high error rates were due to typos or ambiguous answers).
- The question must have an error rate above 0.4, which is considered a high error rate according to the literature (Alsubait et al., 2013; Lowman, 1995).

Considering the cost of the user study, especially the recruitment of experienced instructors, I limited the number of input sets to 12. This decision effectively made one session of the user study only take around an hour and potentially improved the subject's willingness to participate.

For each input question, the participant is asked to generate at least one new programming question. The instructor-generated question demonstrates the characteristics of programming questions that are expected by the instructor for the input question. It serves as a baseline reference, or the "gold standard" in terms of natural language understanding, for my PQG model.

Afterward, the participant is asked to evaluate the quality of the generated questions according to (1) the relevance to the topic of the input ("Topic-Rel"), (2) the extensibility in terms of topics ("Ext-Topics"), (3) the extensibility in terms of complexity ("Ext-Complex"), and (4) the extensibility in terms of the participant's need of question generation ("Ext-Need"). Specifically, the Topic-Rel asks whether "*the generated content*

covers important programming topics in the input"; the Ext-Topics asks "*It is easy for me to modify the content of the generated questions and generate new ones for students to practice topics in the input*"; the Ext-Complex asks "*I can modify/improve the generated content to generate questions that are complex enough to distinguish students' abilities*"; and the Ext-Need asks "*The generated content has sufficient information for me to generate new questions for my students in introductory programming courses.*" All the evaluation questions are 5-item Likert scales with scores from -2 to +2.

As far as I know, there is no existing and publicly available model or benchmark datasets of PQG. To compare the performance of the model with a reference, I devised a reference model by masking part of the proposed PQG model. The reference model, called the "code-aware" model, uses only the AST structures to generate programming questions. The reference model is compared to the other model called the "context-aware" model which uses the LKG structure to generate programming questions. For reference, example templates of the code-aware mode and the context-aware model are shown in Figure 6 and Figure 7, respectively.

Q: How many primitive data types are there in Java?
Q: How would you declare a variable of type int?

```
if name == "variable":
    code_qs_list += [
        f"How many primitive data types are there in Java?",
        f"How would you declare a variable of type {a[1]}?"
    ]
```

Q: How many times will the loop runs with the initializer [int i = 913], the condition i <= 937 and the update [i++]?

```
f"How many times will the loop runs with the initializer {a[1]}, the condition
{a[2]} and the update {a[3]}?"
```

Figure 6. An Example Template of the Code-aware Model. The highlighted part is replaced by arguments of AST nodes according to their functionality. An AST node, (Variable Declaration, int) can be used to generate questions that are related to the implementation of variable declarations. Similarly, an AST node, (Loop Declaration, initializer, condition, update) can be used to generate questions about loop execution.

Q: How do you convert each integer (in the range) to the corresponding character?

```
elif (n_nodes == 4) and (vb_lemma == "convert") and (a1) and (a2) and (aloc):
    qs = f"How do you convert {a1} {aloc} {a2}?"
```

Q: What code can we use to iterate each character in the string?

```
elif (n_nodes == 4) and (vb_lemma == "iterate") and (a0.lower() == "we")
and (a1) and (aloc):
    qs = f"What code can we use to iterate {a1} {aloc}?"
```

Figure 7. An Example Template of the Context-aware Model. The highlighted part is replaced by arguments of LKG triples according to the grammar.

Theoretically, questions generated by the context-aware model cover a broader range of knowledge than the code-aware model because the PQG process involves information from program description that is not specifically limited to the syntax of code. To some extent, the result of this comparison can indicate the performance of the LKG mechanism in this work.

5.2 Data Collection

We recruited 7 participants who had teaching experience in introductory programming courses via communication in professional networks that involved instructors at universities. The study was reviewed and approved by the IRB at Arizona State University (reference ID: STUDY00015306). According to the responses, around 58% of the participants had more than 5 years of teaching experience in introductory programming courses; the other 42% had 2-5 years of teaching experience. For each variable, I collected 84 data points for statistical analysis.

Although the sample size was relatively small from the perspective of statistics, I believe the sample is representative of the population of programming instructors because introductory programming courses in universities usually have similar designs (including variables, operators, control flows, loops, arrays, object-oriented programming). Considering the cost of recruiting experienced instructors in the field, this work may still contribute valuable insights into the experts' opinions about PQG models. Moreover, due to the lack of existing public benchmarks, the performance report of this work can also serve as a baseline reference for related work in the future.

5.3 The Participants' Teaching Experience

The survey focused on three aspects of the participants' teaching experience: the challenges faced during PQG, the type of questions that are valued the most, the expected

support from PQG tools, and the likelihood of using a PQG tool in practice if such a tool is available.

The survey question about the challenges provided four options that included the amount of time to spend on PQG, the difficulty of questions, the lack of reference, and the type of questions. The participant was asked to select all that apply. The result showed that about 58% of the participants indicated that PQG takes them too much time; about 58% indicated that they are not sure about the appropriate difficulty of questions for their students; about 25% indicated that they lack useful reference materials when generating questions; none of the participants indicated that the type of questions is challenging to PQG. This result adheres to the literature on general QG which suggests that the QG process is not only time-consuming but also intelligence-demanding (Kurdi et al., 2020). To make high-quality questions, the instructors are required to have a good understanding of the learning content, the student performance, and an appropriate match of these two. The outcome implies the need for PQG tools that can support instructors to make programming questions.

The question about "the type of questions that you value the most" provided four options including "code-tracing", "code-writing", "remembering", and "debugging". The participants were asked to select one option among themselves. The result showed that about 70% of the participants valued code-writing questions the most, and the other 30% valued code-tracing questions the most. None of the participants voted for remembering and debugging questions. This result is somewhat expected because one learning goal of

introductory programming courses is to train the student to understand the basic syntax of programming languages. To achieve this, the code-writing skill is usually valued the most as it requires an adequate level of code comprehension, which to some extent includes both code tracing and code writing. Interestingly, the survey outcome showed that none of the participants voted for debugging questions, even though the debugging skill is usually considered fundamental to computer programming (Fitzgerald et al., 2008). I believe this could be due to the scarceness of (or the lack of) training of debugging skills in the curriculum of introductory programming courses. Some instructors may also assume that the debugging skill is accompanied by the mastery of both the code-tracing and code-writing skills. Overall, the result of this question provides a direction for the future development of PQG research and tools.

Regarding the expectation of PQG tools, the survey provided three options for the participants to select all that apply. The options included "code examples", "questions", and "the descriptions of concepts to learn". There were about 70% of the participants expected to obtain related code examples; about 70% expected to obtain related questions; and about 28% expected to obtain the descriptions of concepts to learn. This result provides a guideline for the development of PQG tools: both code examples and questions are needed by instructors in introductory programming courses. Although this result seems obvious, it implicitly confirms that the search (or generation) of code examples and questions is a demanding process for instructors. A PQG tool that can provide such information is potentially helpful for instructors in practice. Moreover, the result of this survey question can also serve as a factor in the evaluation of my PQG

model because the instructors' responses to the question may be correlated with their ratings on the machine-generated examples and questions. This multi-level analysis is further explained below.

Finally, the survey question about "how likely the participants would use a PQG tool in practice" provided a scale from 1 to 5 that represents the likelihood of adopting a PQG tool. The outcome showed that about 58% and 25% of the participants gave scores of 3 and 4 respectively to the question. The rest of the participants gave a score of 2. This result shows the need for PQG tools and that most of the participants are willing to try or adopt a PQG tool in practice.

5.4 High Similarity between the Instructor-generated Questions and Machine-generated Questions

To find out the semantic similarity between the collected instructor-generated questions and the corresponding machine-generated questions, I trained a Word2Vec embedding out of public programming textbooks. The embedding model was then used to transform the input questions (*IN*), the instructor-generated questions (*USER*), and machine-generated questions (*CA* for code-aware; *CT* for context-aware) into vectors by which I was able to compute the cosine similarity (*SIM*) between them as shown in Figure 8.

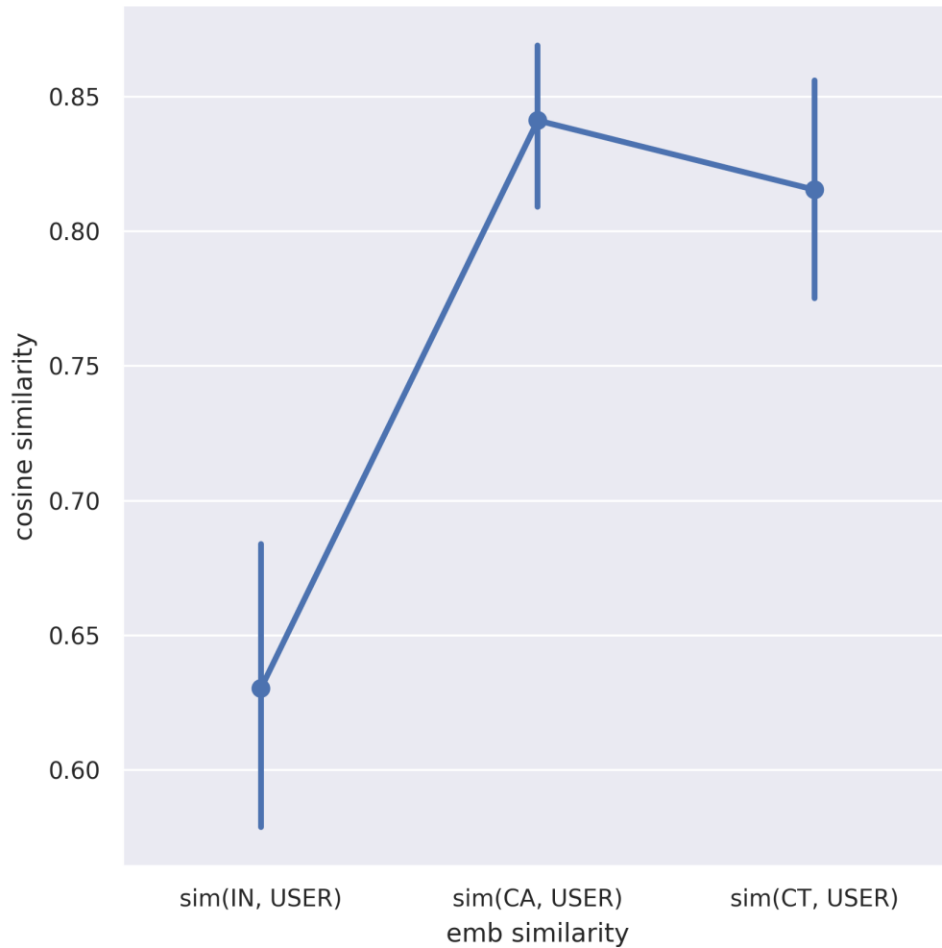

Figure 8. Embedding Similarity between the Input (IN), Machine-generated (CA=code-aware, CT=context-aware), and Instructor-generated (USER) Questions.

The result showed that the SIM(CA, USER) (M=0.84, SD=0.14) and the SIM(CT, USER) (M=0.82, SD=0.19) were similar and significantly higher than the SIM(IN, USER) (M=0.63, SD=0.25) (t(166)=6.72, p=0.00; t(166)=5.43, p=0.00). The relatively low SIM(IN, USER) to some extent suggests that the instructor-generated questions addressed concepts that were more extensive than the input questions, which is expected because the instructors were asked to generate new questions that can help their students to practice the concept from the input question. The significantly high and similar

SIM(CA, USER) and SIM(CT, USER) suggest that the machine-generated questions addressed concepts similar to the instructor-generated questions. Also, there was no significant difference between the code-aware and the context-aware models. These outcomes plausibly indicate that the knowledge extraction of my PQG model was aligned with the instructors' opinions. In other words, the model was able to generate questions that were similar to those generated by the instructor.

5.5 Significantly Positive Ratings on the Question Utility

In Figure 9 and Table 3, I summarized the instructors' subjective opinions on the machine-generated questions regarding the topic relevance and extensibility in terms of topics, complexity, and instructional needs. First, I found that both experiment models received a significantly positive (with respect to zero) rating, especially on the extensibility of complexity. There was no significant difference found between the two models. This result suggests that the experienced instructors were generally satisfied with the utility of the machine-generated questions. They found the machine-generated questions relevant to the input questions. The machine-generated questions also provided sufficient details for them to generate new questions that address similar topics and are complex enough to distinguish between high-performing and low-performing students. Overall, my PQG model was able to supply sufficient information for them to generate new questions for students.

Interestingly, the code-aware model did not perform worse than the context-aware model, which rejects my hypothesis about their performance (the code-aware model covers a narrower range of knowledge than the context-aware model, therefore being an inferior option). Plausibly, this result suggests that my PQG model was helpful for the instructors to generate new questions no matter if it was only code-aware or only context-aware, even though the effect of the full model remains unclear and requires further analysis. Due to the similar performance of the two masked models, the following analyses report the ratings with the two combined for readability.

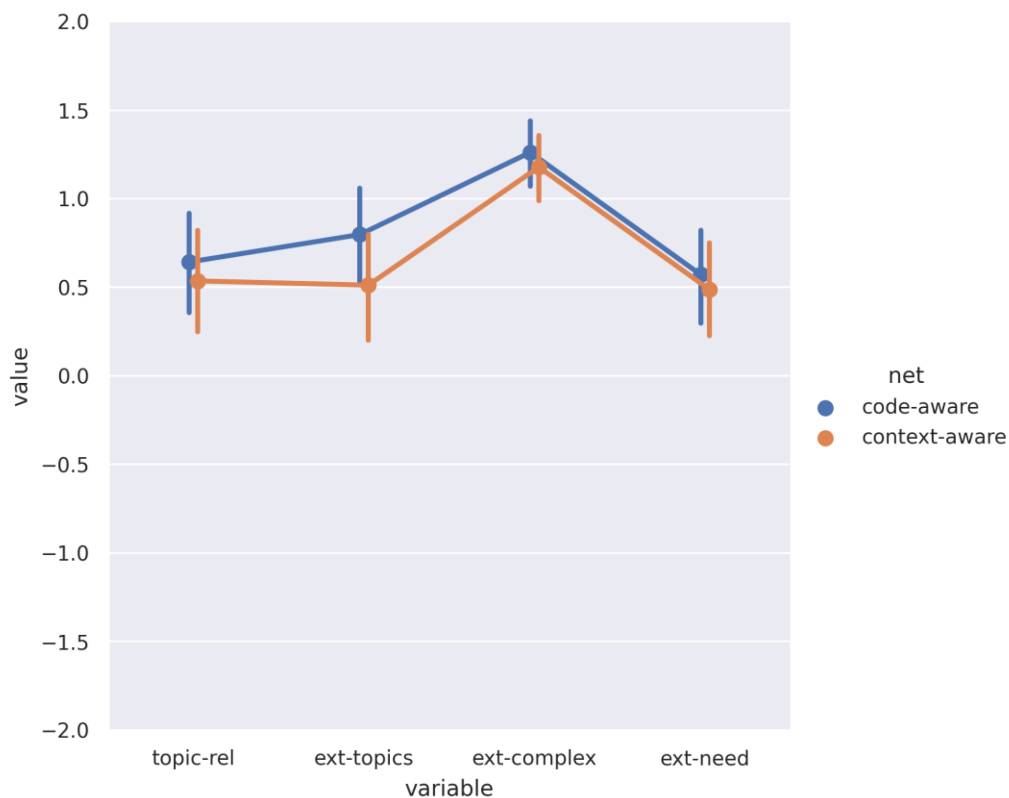

Figure 9. The Distribution of Ratings to the Four Utility Variables, Topic Relevance (Topic-Rel), Extensibility of Topics (Ext-Topics), Complexity (Ext-Complex), and the Instructor's Need (Ext-Need). The instructors consistently gave significantly positive ratings to all the variables.

Table 3. The Statistics of One-sample T-tests for the Four Rating Variables (Reported in the Format "M, SD, test(DoF)=V (pval)"

	Code-aware	**Context-aware**
Topic-Rel	0.64, 1.34, t(83)=4.39 (0.00)	0.64, 1.34, t(83)=3.59 (0.00)
Ext-Topics	0.80, 1.27, t(83)=5.76 (0.00)	0.80, 1.27, t(83)=3.37 (0.00)
Ext-Complex	1.26, 0.81, t(83)=14.31 (0.00)	1.26, 0.81, t(83)=12.67 (0.00)
Ext-Needs	0.57, 1.24, t(83)=4.21 (0.00)	0.57, 1.24, t(83)=3.62 (0.00)

5.6 The PQG Model Helps Generate General Code-writing Questions and Complex Code-tracing Questions

Instructors likely need different kinds of support when generating new programming questions. To find out whether my PQG model can address certain needs of the instructors, I conducted a multilevel analysis by factoring in the preferred question types as shown in Figure 10 and Table 4. The analysis showed that the instructors who valued code-writing questions were the major source of positive ratings in the evaluation. They unanimously expressed significantly positive ratings on all four variables. However, the instructors who valued code-tracing questions the most had mixed ratings: Only the extensibility of complexity received a significantly positive rating from this group.

Figure 10. Multilevel Analysis of Ratings and Preferred Question Types. The instructors who prefer code-writing questions gave consistently positive ratings to all variables; however, those who prefer code-tracing questions gave mixed ratings except for the extensibility of complexity.

Table 4. The One-sample Statistics of the Multilevel Analysis (Preferred Question Types and Ratings).

Preferred Question Type	**Variable**	**Statistics**
Code-tracing	Topic-Rel	-0.56, 1.57, t(47)=-2.46, 0.02
	Ext-Topics	-0.50, 1.49, t(47)=-2.31, 0.03
	Ext-Complex	1.10, 1.07, t(47)=7.11, 0.00
	Ext-Needs	-0.27, 1.54, t(47)=-1.21, 0.23
Code-writing	Topic-Rel	1.05, 0.90, t(119)=12.69, 0.00
	Ext-Topics	1.12, 0.92, t(119)=13.19, 0.00
	Ext-Complex	1.27, 0.70, t(119)=19.63, 0.00
	Ext-Needs	0.85, 0.91, t(119)=10.19, 0.00

This result is interesting as it points out that my PQG model may only address the needs of a certain population. For code-writing questions, the PQG model was able to provide topic-relevant and highly extensible questions for the instructors to use after edits in practice. However, for code-tracing questions, the model was limited and only able to provide questions that were complex enough but not topic-relevant nor extensible in

terms of topics and instructional needs. One plausible explanation is that the instructors might expect code-tracing questions to be more aligned with the content they use in class, e.g., small code snippets or worked examples in the instructions. Nevertheless, the design of my PQG model is based on the content of public textbooks that may not include such an example. Considering the target audience of the instructors is students from introductory programming courses, my PQG model likely generated questions that were too complex for the students to learn, which is partially confirmed by the high rating for the variable Ext-Complex.

5.7 Either Generated Questions or Generated Code Examples are Helpful for the Instructors

The instructor's expectation of PQG tools may implicitly influence her/his ratings on the utility of my PQG model since s/he may put more weight on the thing s/he is looking for. I believe there is a correlation between the expectation and the rating for the machine-generated questions. To better understand whether my PQG model fulfills the instructor's need for PQG, I further analyzed the ratings for my PQG model by the instructor's expectations of PQG tools as shown in Figure 11 and Table 5. The ratings were grouped into four categories of instructors as follows: (1) expecting both questions and code examples (*qs=True, ex=True*), (2) expecting only questions (*qs=True, ex=False*), (3) expecting only code examples (*qs=False, ex=True*), and (4) expecting neither questions nor code examples (*qs=False, ex=False*). I discuss each of these categories in the following sections.

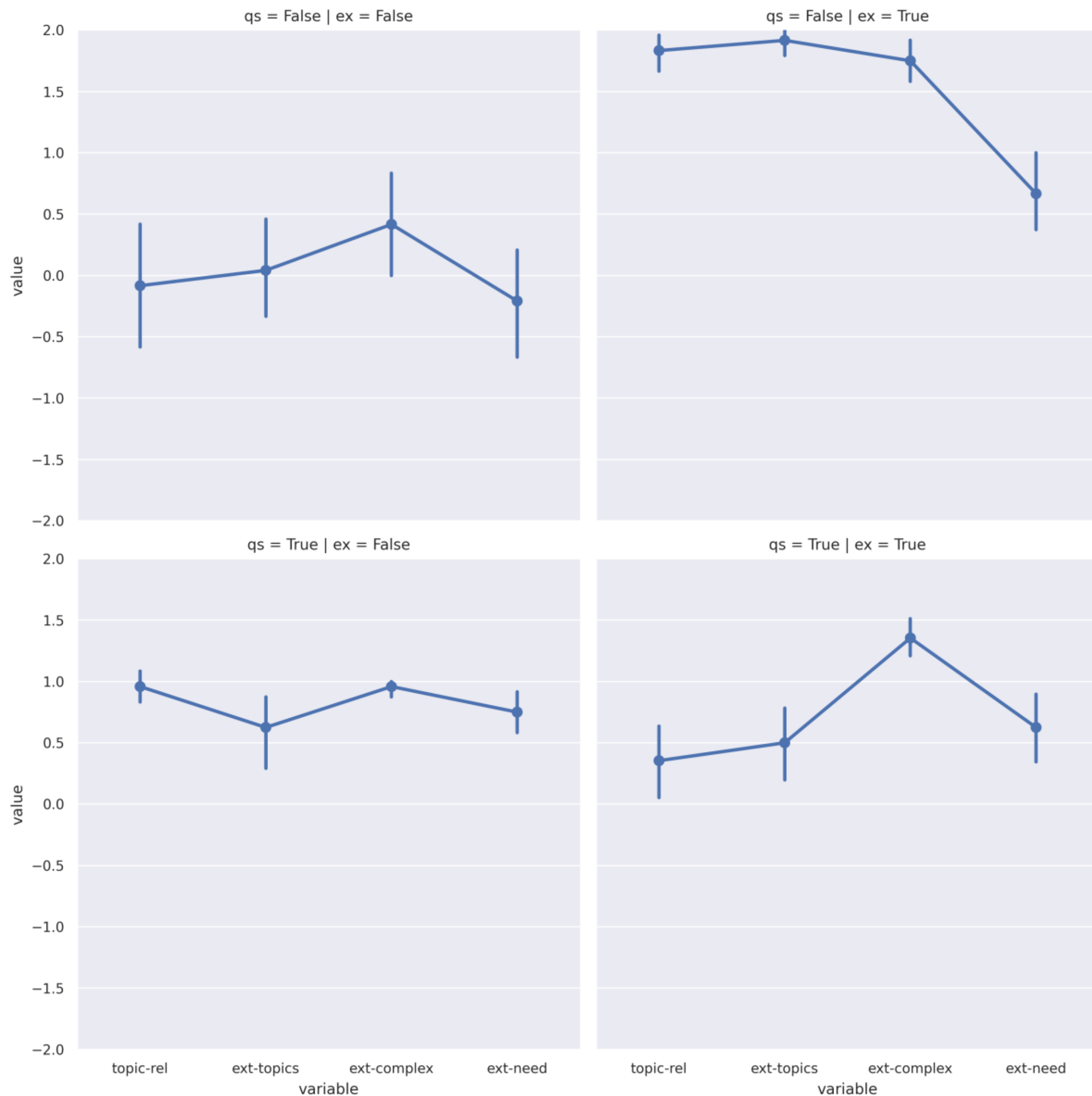

Figure 11. Multilevel Analysis of Ratings and PQG Expectations. The instructors who looked for questions or code examples consistently gave positive ratings to all variables; those who looked for neither questions nor code examples gave positive ratings only to the extensibility of complexity.

***Expecting Only Questions: Significantly Positive Ratings on All Variables*.** The instructors who expect to obtain only questions gave significantly positive ratings for all the variables. This result suggests that the quality of the machine-generated questions was

aligned with those instructors' expectations of PQG tools. The questions were potentially helpful for the instructors to make new programming questions for their students in introductory programming courses.

Expecting Only Code Examples: Significantly Positive Ratings on All Variables. Similar to the instructors who expect to obtain only questions, those who expect to obtain only code examples also gave significantly positive ratings for all the variables. Although my PQG model does not specifically focus on generating code examples, the examples are part of the generated questions according to the algorithm design where the model first retrieves related code examples and then generates a set of related questions. The result suggests that the code examples from my PQG model were aligned with the instructors' expectations of PQG tools.

Expecting Both Questions and Code Examples: A Significantly Positive Rating Mainly on the Ext-Complex. The instructors who expect to obtain both programming questions and code examples gave significantly positive ratings mainly on the Ext-Complex. This result is similar to the rating given by those who value code-tracing questions the most. It suggests that the machine-generated questions and code examples were not completely aligned with the instructors' expectations. There are many possible explanations for this. For example, the affinity between the machine-generated questions and the code examples might not match the instructors' expectations. They might be looking for other programming questions (or code examples) that better match the difficulty of introductory programming courses. It is also likely that the generated questions did not

address necessary concepts for their students to learn. Due to the limitation of the evaluation tasks, future studies are required to find out exact explanations.

Expecting Neither Questions nor Code Examples: Mixed Ratings on All Variables. The instructors who expect neither questions nor code examples gave mixed ratings for all variables in the evaluation task. Due to the limitation of the task design, it remains unclear what kinds of support are expected by those instructors. The mixed ratings only suggest that the output of my PQG model (questions and the accompanying code examples) is not aligned with the information sought by such instructors. Further studies are required to clearly define the needs and requirements of PQG.

Table 5. The One-sample Statistics of the Multilevel Analysis (Expected PQG Support and Ratings).

Expected PQG Support	Variable	Statistics
Only Questions	Topic-Rel	0.96, 0.35, t(23)=13.09, p=0.00
	Ext-Topics	0.62, 0.70, t(23)=4.31, p=0.00
	Ext-Complex	0.96, 0.20, t(23)=23.00, p=0.00
	Ext-Needs	0.75, 0.43, t(23)=8.31, p=0.00
Only Code Examples	Topic-Rel	1.83, 0.37, t(23)=23.59, p=0.00
	Ext-Topics	1.92, 0.28, t(23)=33.26, p=0.00
	Ext-Complex	1.75, 0.43, t(23)=19.38, p=0.00
	Ext-Needs	0.67, 0.85, t(23)=3.76, p=0.00
Both Code Examples and Questions	Topic-Rel	0.35, 1.45, t(95)=2.38, p=0.02
	Ext-Topics	0.50, 1.48, t(95)=3.30, p=0.00
	Ext-Complex	1.35, 0.79, t(95)=16.70, p=0.00
	Ext-Needs	0.62, 1.40, t(95)=4.35, p=0.00
Neither Code Examples nor Questions	Topic-Rel	-0.08, 1.29, t(23)=-0.31, p=0.76
	Ext-Topics	0.04, 1.02, t(23)=0.20, p=0.85
	Ext-Complex	0.42, 1.00, t(23)=2.01, p=0.06
	Ext-Needs	-0.21, 1.12, t(23)=-0.89, p=0.38

CHAPTER 6

DISCUSSION

6.1 The Design Focusing on Extensibility of Methodology and Inclusion of Topics

The RQ1 asks about how to capture the conceptual and implementation programming knowledge. The model is designed to use the LKG to extract the conceptual knowledge and the AST to capture the implementation knowledge. This semantic-network-based approach provides an intuitive model, i.e., verb-argument semantic triples, for the model to process abstract programming knowledge and generate new questions by fitting templates. Such a semantic network is also a widely used approach in the literature on natural language processing, which allows my PQG model to be easily extended for its future development.

The RQ2 asks about how the PQG model queries related code examples from the semantic network. First, for a given question, the model will parse and extract the code (in the form of AST nodes) and the content (in the form of semantic triples). These nodes and triples are used to match against the built semantic network to retrieve related code examples. To rank the query candidates, I chose to use a generalized version of the Jaccard index, the Tversky index, to rank the examples according to the inclusion of topics (i.e., the overlap of AST nodes and semantic triples). One benefit of this index is the ability to control the ratio between the topics of input and the topics of query candidates. In the current work, I chose to maximize the inclusion of topics, which means

the queried result contains not only the input topics but also other potentially related topics. Nevertheless, the effect of this ratio still requires future investigation.

The RQ3 asks about how the PQG model generates a set of new programming questions. This work uses a template-based generation approach that includes a set of templates for semantic triples and AST nodes. Although the template-based method is relatively limited when compared with a statistical method, the template-based method allows the instructor to easily engineer his/her expert knowledge into the PQG model. I believe this characteristic is favored when the instructor needs to have control over what kinds of questions to generate.

6.2 Significantly Positive Ratings from the Experienced Instructors

The RQ4 focuses on the evaluation of the PQG model in terms of instructors' opinions. To answer this research question, I recruited a group of experienced instructors from introductory programming courses. A survey of teaching experience was used to understand the challenges and needs of PQG from the instructors' perspectives. The result showed that, unsurprisingly, the demand of time and the balance of student performance and question difficulty are the top challenges faced by the instructors. The survey also showed that most of the instructors expect a PQG tool to provide related programming questions and code examples for the PQG process. Although these outcomes are aligned with my expectations, considering the lack of empirical studies about PQG in the

literature, the outcomes still provide imperative and first-hand evidence about the nature of PQG from experienced instructors in practice.

Moreover, the RQ4 was further answered by the ratings given by the instructors across the four variables (topic-relevance, extensibility-topic, extensibility-complexity, extensibility-need). The comparison showed that the instructor-generated questions were highly similar to the machine-generated questions. Moreover, the rating responses showed that all the variables received significantly positive ratings, which means the instructors found the machine-generated questions were topic-relevant and extensible in terms of question complexity, topics, and their instructional needs. Although the current study is limited due to the small-scope evaluation task, the result suggested that my PQG model captured important programming knowledge in the input. The generated questions and code examples were aligned with the instructors' opinions. Overall, these outcomes verify the utility of my PQG model.

Finally, considering that the instructors had different needs in PQG, I also explored the correlation between the instructors' ratings and their expectations of PQG tools. Multi-level analysis was conducted by factoring in the ratings, the preferred question types, and the expected support. The result revealed that the instructors who value code-writing questions the most were the major source of significantly positive ratings. Also, those who prefer code-tracing questions gave significantly positive ratings to only the extensibility-complexity. Furthermore, the result also showed that my PQG model was able to generate questions and code examples aligned with the instructors' opinions,

especially from those who seek either the questions or the code examples alone. The implication of these outcomes is twofold. First, they demonstrate that my PQG model was able to provide content that met the instructors' needs in PQG. Second, the outcomes also point out a potential limitation of my PQG model in generating code tracing questions, which probably requires more information on the in-class student performance and focuses on specific class content. To sum up, these results provide valuable insights into future improvements and studies for my PQG model.

CHAPTER 7

CONCLUSIONS

This work focuses on a programming question generation (PQG) model that aims to help instructors make new questions for their students in introductory programming courses. The model uses the Local Knowledge Graph to extract the conceptual programming knowledge and uses the Abstract Syntax Tree to extract the procedural knowledge from textbooks. For a given question, the model can generate a set of related code examples and questions by which the instructor can easily expand the existing question set for different purposes. A group of experienced programming instructors was recruited to help evaluate the utility of the machine-generated questions. The result showed that the instructions had significantly positive feedback on the topic relevance and the extensibility of the generated questions. Furthermore, the machine-generated questions addressed topics that were like the instructor-generated questions. Moreover, a multi-level analysis suggested that my PQG model was able to generate questions and code examples that met the instructors' needs in the PQG process. In conclusion, this work provides empirical evidence for a feasible design of PQG models which have not been explored much in the literature on automatic question generation and programming learning technologies.

7.1 Limitations and Future Work

There are limitations to this work. First, the evaluation task covered a small scope and a limited number of topics in programming learning. Programming topics may range from

basic syntax to advanced knowledge. The evaluation also included a small set of machine-generated questions. Further research is still required to validate the model's capability in different topics. Second, the influence on the rating from part of the machine-generated content remains unclear. The current PQG model generates both code examples and questions about the examples. However, the evaluation task did not ask the instructor to evaluate them individually or separately. The instructors might give more weight to either the examples or the questions in their ratings. Third, the sample size in this work was small due to the cost of recruiting experienced instructors. Although the feedback from such a population is valuable, it still requires a larger sample to generalize the findings of this work. Finally, due to the lack of widely available datasets for the performance analysis of PQG models, this work used two masked models and compared their performance to each other and a set of instructor-generated questions. This made the analysis subjective with respect to the recruited instructors' preferences.

Overall, future studies may consider using the result of this work as one baseline reference or developing a more rigorous design for PQG performance analysis. Several design decisions can be further investigated from this work. For example, the current query mechanism of code examples is based on the Tversky index with the maximum topic inclusion. A future study can investigate the quality of questions that are generated by different degrees of topic inclusions. Moreover, the query mechanism can also be improved by alternative models, e.g., code execution paths, AST tree structures, and the alignment between program code and its context.

In addition to the model design, a future study may also focus on the construction of LKG. The current model only relies on one textbook as the source of knowledge, which has only a limited number of code examples. This may be improved by augmenting the knowledge from multiple textbooks, or by artificially generating code examples out of existing ones. Furthermore, a future study can also allow the user (students or instructors) to provide user-generated content in a crowdsourcing way.

Finally, a future study may focus on the augmentation of the code-aware and context-aware models. Although my results show that the instructors had positive feedback on the two models, the performance of the full model remains unclear. In addition to a comparison between a full model and a masked model, a future study may consider a more intelligent way to pool questions from the two models and devise a query function to retrieve/group the questions according to, for example, their high-level concepts, topics, target skills, or other variables of instructional designs.

7.2 Contributions from My Dissertation

This dissertation encompasses my research of self-assessment for programming learning over three fields of study, learning analytics, educational data mining, and intelligent authoring tools. First of all, I explored the key behavior to the success of learning on a self-assessment platform. I found that practice persistence and regularity were highly correlated with student performance (Chung & Hsiao, 2019, 2020). I also explored the self-explanation effect in programming self-assessment (Chung & Hsiao, 2021a). On top

of the finding of practice persistence and regularity, I further explored various computational models for analyzing self-assessment behavior, including a signal model for the spacing effect (Chung & Hsiao, 2021b) and a knowledge-graph-based approach for question complexity (Chapter 4). To support the supply of self-assessment questions, I also developed an innovative intelligent tool that can automatically generate programming questions for instructors to improve the quality and quantity of existing question sets (Chapter 5). Overall, my dissertation contributes to not only the understanding of effective self-assessment behavior but also various learning analytics and tools that can help implement the instructional design efficiently in classrooms.

REFERENCES

Alsubait, T., Parsia, B., & Sattler, U. (2013). A similarity-based theory of controlling MCQ difficulty. *2013 Second International Conference on E-Learning and E-Technologies in Education (ICEEE)*, 283–288. https://doi.org/10.1109/ICeLeTE.2013.6644389

Alzaid, M., & Hsiao, I.-H. (2019). Behavioral Analytics for Distributed Practices in Programming Problem-Solving. *Proceedings of 2019 IEEE Frontiers in Education Conference (FIE)*, *2019-Octob*, 1–8. https://doi.org/10.1109/FIE43999.2019.9028583

Alzaid, M., Trivedi, D., & Hsiao, I.-H. (2017). The effects of bite-size distributed practices for programming novices. *Proceedings of 2017 IEEE Frontiers in Education Conference (FIE)*, 1–9. https://doi.org/10.1109/FIE.2017.8190593

Ancona, D., Mascardi, V., & Pavarino, O. (2012). Ontology-based documentation extraction for semi-automatic migration of Java code. *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, *2*, 1137. https://doi.org/10.1145/2245276.2231955

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural Machine Translation by Jointly Learning to Align and Translate. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 1–15. http://arxiv.org/abs/1409.0473

Brusilovsky, P., & Sosnovsky, S. (2005). Invidualized Exercises for Self-Assessment of Programming Knowledge: An Evaluation of QuizPACK. *Journal on Educational Resources in Computing*, *5*(3), 6. https://doi.org/10.1145/1163405.1163411

Chau, H., Labutov, I., Thaker, K., He, D., & Brusilovsky, P. (2020). Automatic Concept Extraction for Domain and Student Modeling in Adaptive Textbooks. *International Journal of Artificial Intelligence in Education*. https://doi.org/10.1007/s40593-020-00207-1

Chung, C.-Y., & Hsiao, I.-H. (2019). Quantitative Analytics in Exploring Self-Regulated Learning Behaviors: Effects of Study Persistence and Regularity. *Proceedings of 2019 IEEE Frontiers in Education Conference (FIE)*, 1–9. https://doi.org/10.1109/FIE43999.2019.9028665

Chung, C.-Y., & Hsiao, I.-H. (2020). Investigating Patterns of Study Persistence on Self-Assessment Platform of Programming Problem-Solving. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 162–168. https://doi.org/10.1145/3328778.3366827

Chung, C.-Y., & Hsiao, I.-H. (2021a). Examining the Effect of Self-explanations in Distributed Self-assessment. *Proceedings of the 16th European Conference on Technology Enhanced Learning, EC-TEL 2021*, 149–162. https://doi.org/10.1007/978-3-030-86436-1_12

Chung, C.-Y., & Hsiao, I.-H. (2021b). From Detail to Context: Modeling Distributed Practice Intensity and Timing by Multiresolution Signal Analysis. *Proceedings of the 14th International Conference on Educational Data Mining (EDM21)*, 1–7. https://educationaldatamining.org/edm2021/

Chung, C.-Y., Paredes, Y. V., Alzaid, M., Papakannu, K. R., & Hsiao, I.-H. (2020). A Longitudinal Study on Student Persistence in Programming Self-assessments. *Proceedings of the 4th Educational Data Mining in Computer Science Education (CSEDM) Workshop*.

Czerkawski, B. C., & Lyman, E. W. (2015). Exploring Issues About Computational Thinking in Higher Education. *TechTrends*, *59*(2), 57–65. https://doi.org/10.1007/s11528-015-0840-3

Delaney, P. F., Verkoeijen, P. P. J. L., & Spirgel, A. (2010). Spacing and Testing Effects. In *Psychology of Learning and Motivation - Advances in Research and Theory* (1st ed., Vol. 53, pp. 63–147). Elsevier Inc. https://doi.org/10.1016/S0079-7421(10)53003-2

Denny, P., Hamer, J., Luxton-Reilly, A., & Purchase, H. (2008). PeerWise: Students Sharing their Multiple Choice Questions. *Proceedings of the 8th International Conference on Computing Education Research - Koli '08*, 109. https://doi.org/10.1145/1595356.1595378

Downey, A. B., & Mayfield, C. (2019). *Think Java: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.

Du, X., Shao, J., & Cardie, C. (2017). Learning to Ask: Neural Question Generation for Reading Comprehension. *ACL 2017 - 55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, *1*, 1342–1352. https://doi.org/10.18653/v1/P17-1123

Dunlosky, J., Rawson, K. A., Marsh, E. J., Nathan, M. J., & Willingham, D. T. (2013). Improving Students' Learning With Effective Learning Techniques. *Psychological Science in the Public Interest*, *14*(1), 4–58. https://doi.org/10.1177/1529100612453266

Fan, A., Gardent, C., Braud, C., & Bordes, A. (2019). Using local knowledge graph construction to scale Seq2seq models to multi-document inputs. *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, 4186–4196. https://doi.org/10.18653/v1/d19-1428

Finkel, J. R., Grenager, T., & Manning, C. (2005). Incorporating non-local information into information extraction systems by Gibbs sampling. *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics - ACL '05*, *1995*, 363–370. https://doi.org/10.3115/1219840.1219885

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Gillam, L., Tariq, M., & Ahmad, K. (2005). Terminology and the construction of ontology. *Terminology. International Journal of Theoretical and Applied Issues in Specialized Communication*, *11*(1), 55–81. https://doi.org/10.1075/term.11.1.04gil

Guerra, J., Sahebi, S., Brusilovsky, P., & Lin, Y. (2014). The Problem Solving Genome: Analyzing Sequential Patterns of Student Work with Parameterized Exercises. *Proceedings of the 7th International Conference on Educational Data Mining*, *Edm*, 153–160. http://people.cs.pitt.edu/~sahebi/genome_edm2014.pdf

Hsiao, I.-H., Sosnovsky, S., & Brusilovsky, P. (2009). Adaptive Navigation Support for Parameterized Questions in Object-Oriented Programming. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Vol. 5794 LNCS* (pp. 88–98). https://doi.org/10.1007/978-3-642-04636-0_10

Jia, X., Zhou, W., Sun, X., & Wu, Y. (2020). EQG-RACE: Examination-Type Question Generation. *ArXiv*. http://arxiv.org/abs/2012.06106

Kurdi, G., Leo, J., Parsia, B., Sattler, U., & Al-Emari, S. (2020). A Systematic Review of Automatic Question Generation for Educational Purposes. *International Journal of Artificial Intelligence in Education*, *30*(1), 121–204. https://doi.org/10.1007/s40593-019-00186-y

Lowman, J. (1995). *Mastering the Techniques of Teaching*.

Lu, Y., & Hsiao, I.-H. (2017). Personalized Information Seeking Assistant (PiSA): from programming information seeking to learning. *Information Retrieval Journal*, *20*(5), 433–455. https://doi.org/10.1007/s10791-017-9305-y

Palmer, M., Gildea, D., & Kingsbury, P. (2005). The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, *31*(1), 71–106. https://doi.org/10.1162/0891201053630264

Pan, L., Lei, W., Chua, T.-S., & Kan, M.-Y. (2019). Recent Advances in Neural Question Generation. *ArXiv*, *3*. http://arxiv.org/abs/1905.08949

Qiu, Y., Ql, Y., Lu, H., Pardos, Z. A., & Heffernan, N. T. (2011). Does time matter? Modeling the effect of time in bayesian knowledge tracing. *EDM 2011 - Proceedings of the 4th International Conference on Educational Data Mining*, 139–148.

Ruan, S., Jiang, L., Xu, J., Tham, B. J.-K., Qiu, Z., Zhu, Y., Murnane, E. L., Brunskill, E., & Landay, J. A. (2019). QuizBot: A Dialogue-based Adaptive Learning System for Factual Knowledge. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, *Chi*, 1–13. https://doi.org/10.1145/3290605.3300587

Stanovsky, G., Michael, J., Zettlemoyer, L., & Dagan, I. (2018). Supervised open information extraction. *NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, *1*(Section 4), 885–895. https://doi.org/10.18653/v1/n18-1081

Vedula, N., Lipka, N., Maneriker, P., & Parthasarathy, S. (2020). Open Intent Extraction from Natural Language Interactions. *Proceedings of The Web Conference 2020*, *2*, 2009–2020. https://doi.org/10.1145/3366423.3380268

Veremyev, A., Semenov, A., Pasiliao, E. L., & Boginski, V. (2019). Graph-based exploration and clustering analysis of semantic spaces. *Applied Network Science*, *4*(1), 104. https://doi.org/10.1007/s41109-019-0228-y

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2016). Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology*, *25*(1), 127–147. https://doi.org/10.1007/s10956-015-9581-5

Willis, A., Davis, G., Ruan, S., Manoharan, L., Landay, J., & Brunskill, E. (2019). Key Phrase Extraction for Generating Educational Question-Answer Pairs. *Proceedings of the Sixth (2019) ACM Conference on Learning @ Scale*, 1–10. https://doi.org/10.1145/3330430.3333636

Yates, A., Cafarella, M., Banko, M., Etzioni, O., Broadhead, M., & Soderland, S. (2007). TextRunner: open information extraction on the web. *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations on XX - NAACL '07*, 25–26. https://doi.org/10.3115/1614164.1614177

Zhang, L., & VanLehn, K. (2016). How do machine-generated questions compare to human-generated questions? *Research and Practice in Technology Enhanced Learning*, *11*(1). https://doi.org/10.1186/s41039-016-0031-7

Zhang, Y., Rilling, J., & Haarslev, V. (2006). An Ontology-Based Approach to Software Comprehension - Reasoning about Security Concerns. *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, *1*, 333–342. https://doi.org/10.1109/COMPSAC.2006.27

Zhong, V., Xiong, C., & Socher, R. (2017). Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *ArXiv*, 1–12. http://arxiv.org/abs/1709.00103