The Journal of Systems and Software 216 (2024) 112140

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

SynthoMinds: Bridging human programming intuition with retrieval, analogy, and reasoning in program synthesis☆

Qianwen Gou [a], Yunwei Dong [b,*], Qiao Ke [c]

[a] School of Computer Science, Northwestern Polytechnical University, Xi'an, 710072, Shaanxi, China
[b] School of Software, Northwestern Polytechnical University, Xi'an, 710072, Shaanxi, China
[c] School of Mathematics and Statistics, Northwestern Polytechnical University, Xi'an, 710072, Shaanxi, China

ARTICLE INFO

Keywords:
Deep learning
Program synthesis
Information retrieval

ABSTRACT

Program synthesis revolutionizes software development by automatically generating executable programs based on given specifications. An emerging trend is to augment generative models with external memory before generating programs. Better memory, in general, leads to better results. However, existing models tend to devolve into a copy mechanism, where retrieved memories are copied directly into the generative model, leading to misinformation or confusion. A sharp performance decline is caused when the retrieved memories are irrelevant or incorrect.

Inspired by the human programming process—sketching a solution before programming, we propose SynthoMinds. A novel framework that decomposes program synthesis tasks into retrieval, analogy, and reasoning, enabling the generation of programs by leveraging knowledge learned from previously solved solutions. Specifically, given a natural language (NL) description, SynthoMinds first retrieves similar programs via a retrieval module, and then mines the retrieved memories for some insightful revelations via an analogy module. The revelation acts as a bird's-eye view of a program without delving into implementation details. The reasoning module harnesses the power of insightful revelations and NL to generate programs. Experimental results demonstrate that mining revelations from retrieved memories significantly outperforms existing baselines.

1. Introduction

Program synthesis aims to automatically generate programs in an underlying programming language (PL) that fulfill a set of program specifications (e.g., natural language) (Desai et al., 2016; Gavran et al., 2020; Chen et al., 2022). It simplifies programming for both experts and beginners, enabling them to express high-level intent and write code effortlessly (Yang et al., 2022; Ciniselli et al., 2021). Existing models can be divided into two groups: information retrieval (IR)-based and generative-based models (Parvez et al., 2021). Generative-based models can further be classified into SEQ2SEQ and SEQ2TREE.

The IR-based models aim to retrieve relevant programs from the code corpus based on the NL description, then directly copy the retrieval results (Gu et al., 2018; Xu et al., 2021; Chen and Abedjan, 2021). By retrieving and reusing existing programs, developers can speed up development with ready-made solutions, avoiding reinventing the wheel. Since the code corpus is often obtained from real-world software development scenarios and validated by other developers, the retrieved results are usually grammatical and reusable (Parvez et al., 2021). However, the accuracy of IR-based models heavily relies on the scope and quality of the code corpus, limiting their ability to generate novel and innovative programs, and lacking flexibility (Satter and Sakib, 2017).

Generative models, like SEQ2SEQ (Desai et al., 2016), generate programs token-by-token from scratch, whereas SEQ2TREE models generate programs in a tree-structured manner (Rabinovich et al., 2017; Yin and Neubig, 2018; Jiang et al., 2022; Dahal et al., 2021). SEQ2SEQ models lack explicit structure modeling and may exhibit sub-optimal performance. In contrast, SEQ2TREE models excel at generating code that adheres to syntactic and structural constraints (Jiang et al., 2022; Dahal et al., 2021). Despite their end-to-end approach to generating programs from scratch, sometimes the accuracy is still far from expectations. Specifically, they often focus on learning programs' local context over external knowledge. Additionally, these models assume that all libraries and functions encountered during training are known, ignoring the fact that new libraries and functions are introduced.

☆ Editor: Lingxiao Jiang.

* Corresponding author.
E-mail addresses: qianwen@mail.nwpu.edu.cn (Q. Gou), yunweidong@nwpu.edu.cn (Y. Dong), qiaoke@nwpu.edu.cn (Q. Ke).

https://doi.org/10.1016/j.jss.2024.112140
Received 27 July 2023; Received in revised form 12 March 2024; Accepted 16 June 2024
Available online 21 June 2024

However, simply enlarging the corpus may not fully solve this problem. This limitation hinders their trustworthiness in real-world applications. Hence, existing models face challenges in generating code for unseen libraries, highlighting the need for breakthroughs in program synthesis.

To summarize, IR-based models tend to generate grammatically correct but inappropriate programs, while generative-based methods often do the opposite. Human programmers, in contrast to these existing models, frequently refer to previous solutions when writing code (Nykaza et al., 2002; Lethbridge et al., 2003). This allows programmers to easily use functions and libraries without the burden of learning from scratch. Therefore, it is preferable to combine the above merits. Hayati et al. (2018) pioneered the use of retrieval methods in program synthesis models. The retrieved results, together with the original natural language description, are used to feed the decoder in a standard SEQ2SEQ model. Liu et al. (2021) proposed a retrieval-augmented generation method using a hybrid graph neural network (GNN) with attention-based dynamic graphs to capture structural information in source code.

Despite the value and significance of retrieval-augmented models, the program synthesis problem remains insufficiently addressed. A crucial finding from our preliminary studies was that retrieval-augmented models have a higher tendency to fall into the trap of copying the retrieved memories. In such cases, generative models may inadvertently include incorrect or sub-optimal program snippets into their generated output. This can lead to a sharp performance decline, including bugs, inefficiencies, or code that deviates from the specifications. One possible reason is the mixing of useful and useless information in retrieval memories, making them difficult to interpret and control. Furthermore, program synthesis faces challenges due to the lack of separation between high-level abstraction and low-level implementation. This lack of separation makes it difficult to generate the correct programs.

Suppose a programmer needs to implement a specific requirement. He may try to find a program from online repositories or a code corpus that can be directly copied as a solution. However, if he cannot find something suitable, he may use the retrieved solutions as a reference, and analogize the insightful revelations provided in these solutions to create his own implementation.

Motivated by human programming practices, we introduce SynthoMinds, a novel approach that harnesses the power of Retrieval-Analogy-Reasoning to generate programs by leveraging knowledge learned from previously solved solutions. It can recall similar programs that have been solved when working on a new problem, and use these insightful revelations to synthesize a new program through reasoning. Specifically, it uses a hybrid retrieval mechanism to obtain retrieved memories as references from a large code corpus. Retrieved memories often contain redundant information, necessitating the filtering of unnecessary details and the extraction of essential information prior to use. Next, an analogy module is used to extract insightful revelations from the retrieved memories. These revelations act as a bird's-eye view of a program without delving into implementation details. Finally, a reasoning module is applied to synthesize programs by combining the generation and copy mechanisms. By analogizing the retrieved memories, SynthoMinds can identify crucial revelations, such as significant patterns, that play a vital role in the code's functionality. Optimally leveraging insightful revelations, SynthoMinds efficiently reduces the search space for program synthesis, maximizing the use of available resources.

To investigate the effectiveness of SynthoMinds, we conducted extensive experiments on the two NL to code benchmarks (Django and CONALA). Our experimental results and in-depth analysis demonstrate the superiority of SynthoMinds. To this purpose, the main contributions of this work are summarized as follows:

- Imitating human programming practice, we propose a three-stage enhancement-based generation framework that decomposes program synthesis tasks into retrieval, analogy, and reasoning modules. This enables the generation of programs by leveraging knowledge learned from previously solved solutions. The framework remains agnostic to the underlying neural model.
- Extracting revelations from ASTs, rather than relying solely on the longest common sequence, allows developers to capture the essence of a program's structure. This approach avoids being bogged down by specific implementation details.
- Extensive experiments are conducted on two well-known datasets, and the results demonstrate that SynthoMinds not only achieves competitive performance on program synthesis tasks, but also demonstrates the unique ability of learning by analogy.

2. Related work

In this section, we provide a brief summary of the literature related to our work, covering IR-based models, generative-based models, combined IR and generative models, and sketch-based models.

2.1. IR-based program synthesis

IR-based models are widely used in program synthesis, aiming to retrieve programs from the code corpus that highly match the program specification (Xu et al., 2023; Bui et al., 2019; Gou et al., 2024b). Sourcerer (Bajracharya et al., 2010) employed coarse-grained program information, such as the number of if statements and for loops, to refine the results of program retrieval. Gu et al. (2018) use CNN and LSTM to encode code, generating embeddings based on the syntax and context of code tokens. Xu et al. (2021) use two-stage attention mechanisms to incorporate both textual and structural features of code to improve the model's performance. IR-based models tend to generate high-quality programs that rely on a pre-existing code corpus. However, when a suitable program is unavailable or the code corpus has limited coverage, the quality of the generated program becomes constrained, resulting in repetitive or biased program outputs.

2.2. Generative-based program synthesis

SEQ2SEQ and SEQ2TREE are the most commonly used frameworks for generative models (Ciniselli et al., 2021). SEQ2SEQ models were dominant as the mainstream approach in the early stages. Desai et al. (2016) proposed an RNN-based method for automatically generating programs from NL. Dong et al. (2022) proposed a PDA-based mechanism for SEQ2SEQ-based program synthesis. Soliman et al. (2022) introduced MarianCG, a transformer-based program synthesis model. It utilizes Marian neural machine translation, which is based on a pre-trained language model for NL to Code translation. However, SEQ2SEQ models may not fully capture the program structural hierarchy, as they generate programs sequentially. SEQ2TREE, a modification to SEQ2SEQ that may include program syntax information, was proposed to overcome these issues (Rabinovich et al., 2017; Yin and Neubig, 2018; Jiang et al., 2022; Dahal et al., 2021; Sun et al., 2020, 2019). Compared with SEQ2SEQ models, using AST not only shrinks search space but also inherently captures the recursive structure of programming languages, guiding the synthesis of well-formed programs (Jiang et al., 2021; Shen et al., 2022). Rabinovich et al. (Rabinovich et al., 2017) presented Abstract Syntax Networks (ASN). It extends the standard encoder–decoder framework by incorporating a modular decoder composed of sub-models in a top-down fashion to generate AST. TRANX (Yin and Neubig, 2018) generates tree-structured meaning representations (MRs) using a series of tree-construction actions via pre-order traversal. Jiang et al. (2022) proposed an enhanced decoder for program synthesis in TRANX that incorporates both historical and future information in the AST to improve its performance. Dahal et al. (2021) conducted an empirical analysis to evaluate the importance of parse trees in program synthesis. Their findings suggest that structure-aware encoding is more effective in modeling inputs with

multiple variables and capturing long-range dependencies. Xie et al. (2021) compared program synthesis models using two different decoding methods, namely pre-order traversal and breadth-first traversal, and proposed a mutual learning framework to train them together. Recently, research on program pre-trained models has been thriving (Xu et al., 2022; Wang et al., 2023). Le et al. (2022) introduced CodeRL, a framework addressing program synthesis limitations through pre-trained language models and deep reinforcement learning. Wang et al. (2021)(Wang et al., 2023) proposed CodeT5 and CodeT5+, supporting code understanding and generation with multi-task learning and identifier-aware pre-training. IntelliCode (Svyatkovskiy et al., 2020) is a versatile tool for predicting code tokens and generating syntactically accurate code. In conclusion, all these models adopt an end-to-end approach to generating code from scratch, and sometimes the accuracy is still far from expectations. Our work is motivated similarly, and uses the output of an IR model to provide insightful revelations.

2.3. Combination of IR and generative models

While IR-based models excel at addressing specific problems by constructing scalable code corpora, their generation capacity is limited as they are unable to generate arbitrary programs. In contrast, generative models can encode knowledge in the model through parameters, which have a certain generalization ability. Nonetheless, it is a challenge to keep all the required knowledge within the parameters of a generative model. To this end, many researchers have attempted to combine IR and generative models in program synthesis tasks (Gou et al., 2024a). Hayati et al. (2018) proposed a sub-tree retrieval method to enhance the performance of generating general-purpose programs. Alokla et al. (2022) proposed a novel retrieval-based transformer model for generating pseudocode. It uses similarity methods to retrieve similar programs according to the input, and then uses a transformer to translate the retrieved programs into the model's output. Parvez et al. (2021) introduced REDCODER, which uses a dense retriever to retrieve relevant comments or code snippets as a supplement for program summarization or program synthesis tasks. SkCoder (Li et al., 2023) retrieved a similar code snippet, extracted relevant portions to form a code sketch, and subsequently refined the sketch to produce the desired code. However, the aforementioned methods rely on the sequence information of the retrieved code, not the structural information, which may lead to syntactically erroneous code. SynthoMinds differs from these models by implementing an extra analogy module to improve the accuracy of the retrieved memories and avoid errors in the final output.

2.4. Sketch-based program synthesis

Sketch-based program synthesis refers to synthesizing a complete program by filling in predefined placeholders, or "sketches", within a partially specified program (Solar-Lezama, 2009). Solar-Lezama (2008) proposed sketch-based program synthesis, where the synthesizer fills in the blanks of a program sketch according to specified requirements. Then it generates a possible program using input–output examples. Murali et al. (2018) a neural generator on program sketches that removes non-generalizable names and operations, and then uses combinatorial techniques to convert them into type-safe programs. Nye et al. (2019) presented an innovative neuro-symbolic program synthesis system that uses a sketch generator and a symbolic synthesizer to learn an appropriate intermediate sketch representation. Although sketch-based program synthesis can generate programs that satisfy syntactic and semantic constraints, it requires a manual process of defining program sketches and mapping them to concrete programs. Furthermore, it may struggle with handling out-of-vocabulary problems or unseen programming language libraries, especially if they are not included in the defined program sketches. This can limit the flexibility and generalizability of the synthesis process. Unlike sketch-based methods, SynthoMinds extracts helpful insights from retrieval memory for the generation process.

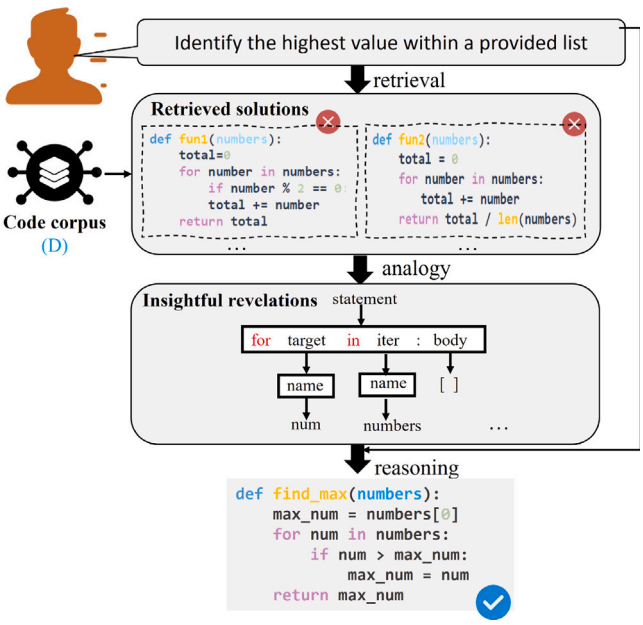


Fig. 1. An Illustration of Developers Programming in a Retrieval-Analogy-Reasoning Manner.

3. Motivated examples

In this section, we provide an illustrative example to demonstrate how developers' prior knowledge influences their development process and explain the motivations behind SynthoMinds.

Software developers frequently encounter the challenge of meeting specific specifications by implementing various functionalities through programming (Cadavid et al., 2023). However, producing a high-quality program from scratch requires significant time and effort. Therefore, developers often resort to retrieving relevant programs from code corpus or online repositories. Sometimes, the retrieved programs may not fully meet specifications. This occurs as these programs are often written for different purposes and may exhibit variations from the specific requirements at hand. Regardless, developers can analyze and analogize retrieved programs to identify insightful revelations that can better solve new problems. For example, they can uncover program structures and design patterns, understand interactions between different program modules, and use the prior knowledge to solve new problems. By leveraging this knowledge, developers can tap into the wisdom and expertise of others, saving time and effort when constructing new solutions from scratch.

Previous studies have mainly focused on either program retrieval or synthesis, with limited research on integrating these two tasks (Desai et al., 2016; Yin and Neubig, 2018; Jiang et al., 2022). As illustrated in Fig. 1, when developers encounter a specification like "*Identify the highest value within a provided list*", they may retrieve relevant programs from the code corpus, like *fun1* and *fun2* in Fig. 1. *fun1* calculates the sum of even numbers in a given list, while *fun2* calculates the average of all numbers in a given list. Despite neither of these programs meeting the specification, the *for-loop structure* used in both programs can provide a valuable clue for writing a program that *searches for the maximum value in a list*. Some attempts have been made to extract sketches from programs, but these approaches are not always practicable as the extracted sketches cannot guarantee grammatical correctness (Solar-Lezama, 2008; Yang et al., 2023). In contrast, ASTs provide a more syntax-aware, structurally faithful, and adaptable approach to programs while maintaining grammatical correctness (Li et al., 2021). In Fig. 1, the retrieved programs have the same revelations (for loop). Without considering specific grammatical implementations,

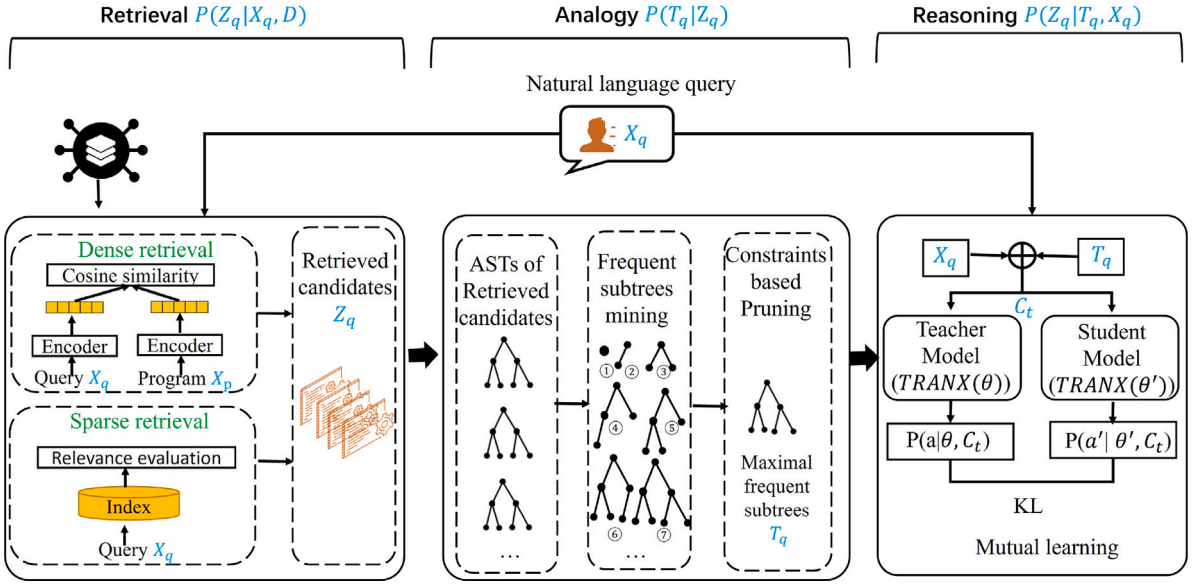


Fig. 2. The Overview Framework of SynthoMinds.

we can express these revelations through frequent sub-trees within these ASTs to capture such insightful revelations.

Imitating programmer programming practices, we propose a Retrieval-Analogy-Reasoning paradigm, SynthoMinds. It can retrieve similar programs that have been solved before when working on a new problem, and use this knowledge to synthesize a new program.

4. Approach

4.1. Overview

The overview of SynthoMinds is illustrated in Fig. 2, which consists of three key components: **(1) Retrieval Module**. It uses a hybrid retrieval method, namely sparse retrieval and dense retrieval, to find the most relevant memories based on NL. **(2) Analogy Module**. Looking closely at the retrieved memories, we observe that they contain valuable revelations that lead to the final solution. Towards this goal, the analogy module aims to identify interesting revelations within the retrieved memories, thereby facilitating the extraction of relevant prior knowledge. **(3) Reasoning Module**. By using the prior knowledge from the analogy module, the reasoning module combines a copy mechanism within a decoder to generate a program that meets the NL. This integration is built upon the SEQ2TREE structure. SynthoMinds is generalizable across programming languages and remains agnostic to the underlying neural model.

4.2. Problem formulation

For a NL specification X_q, our goal is to synthesize a program X_p written in a programming language, like Python. This involves optimizing the model parameters θ to maximize the conditional probability $P_\theta(X_p|X_q)$. SynthoMinds decomposes $P_\theta(X_p|X_q)$ into three steps: retrieval, analogy and reasoning.

Retrieval: Given an input X_q, we first retrieve k relevant programs $Z_q = \{z_q^1, z_q^2, z_q^3, \ldots, z_q^k\}$ from corpus D, where z_p^i represents the ith retrieved memory. D can be constructed flexibly, either as a set of all available program snippets in a programming language, or as a customized subset tailored to the specific domain's scope.

Analogy: Then, we analogize the k retrieved memories and mine their similarities within ASTs. As a result, the maximal frequent subtrees, denoted as $T_q = \{t^1, t^2, \ldots, t^m\}$, are obtained as revelations.

Reasoning: Finally, we condition on both frequent sub-trees T_q and NL description X_q to generate program X_p as Eq. (1).

$$P_\theta\left(X_p \mid X_q, D\right) = \underbrace{P\left(X_p \mid T_q, X_q\right)}_{\text{Reasoning}} \underbrace{P\left(T_q \mid Z_q\right)}_{\text{Analogy}} \underbrace{P\left(Z_q \mid X_q, D\right)}_{\text{Retrieval}}. \tag{1}$$

Our goal is to find $X_p' = \arg\max P_\theta(X_p|X_q, D)$. $|D|$ represents the size of code corpus D. Thus, our problem can be formulated as Eq. (2).

$$\begin{aligned} X_p' &= \operatorname{argmax} P_\theta\left(X_p \mid X_q, D\right) \\ &= \operatorname{argmax} \underbrace{P_\theta\left(X_p \mid T_q, X_q\right)}_{\text{Reasoning}} \underbrace{P\left(T_q \mid Z_q\right)}_{\text{Analogy}} \underbrace{P_\theta\left(Z_q \mid X_q, D\right)}_{\text{Retrieval}}. \end{aligned} \tag{2}$$

Eq. (2) has three parts. First, it estimates the k retrieved memories. Then, it estimates the insightful revelations T_q with $P\left(Z_q \mid X_q, D\right)$. Finally, given X_q and T_q, it estimates the probability of the target program X_p with $P_\theta(X_p|T_q, X_q)$.

4.3. Retrieval

Intuitively, when asked to write code with a given NL, developers will inevitably rely on program retrieval to find reusable knowledge. To maximize the coverage and richness of the retrieved memories, we propose a hybrid retrieval method based on syntax structure and semantic embedding, namely sparse retrieval and dense retrieval. Furthermore, SynthoMinds allows for easy integration of additional program retrieval methods.

4.3.1. Sparse retrieval

Inspired by previous works (Kamiya et al., 2002; LeClair et al., 2019; Wei et al., 2020), we use the BM25 algorithm. a ranking algorithm that calculates the relevance score between a query and a document based on term frequency and inverse document frequency. The more relevant the two queries are, the higher the BM25 score.

To achieve this, we leverage the open-source search engine Lucene to create and search the index for relevant programs. For X_q, we retrieve the top M most relevant programs based on the BM25 scores. Then we can do the following calculation:

$$\begin{aligned} S_{\text{sparse}} &= \operatorname{Sim}\left(X_q, D\right) = \max_{1<i<|D|} \operatorname{BM25}\left(X_q, X_p^i\right), \\ &\text{if } \operatorname{BM25}\left(X_q, X_p^i\right) > 0. \end{aligned} \tag{3}$$

$R_S = \{X_p^1, X_p^2, \ldots, X_p^M\}$ denotes the top-M candidate programs retrieved by the BM25 algorithm, with M as the maximum number of retrieved programs in sparse retrieval.

4.3.2. Dense retrieval

Sparse retrieval is unable to capture semantic similarities when there are no explicit lexical overlaps between the NL and its relevant programs. Therefore, dense retrieval is necessary. It retrieves the most relevant programs to the given NL in the embedding space. Following previous works (Gu et al., 2018; Xu et al., 2021), we encoded the NLs $Q = \{X_q^1, X_q^2, X_q^3, \ldots, X_q^{|D|}\}$ and the programs $P = \{X_p^1, x_p^2, X_p^3, \ldots, X_p^{|D|}\}$ into dense vector representations, where Q and P denote the set of NLs and programs in code corpus D. The encoder takes the NLs and programs as input and maps them into semantic representations, yielding Equation (4).

$$\begin{aligned} V_q^1, V_q^2, V_q^3, \ldots, V_q^{|D|} &= \text{Encoder}\left(X_q^1, X_q^2, X_q^3, \ldots, X_q^{|D|}\right), \\ V_p^1, V_p^2, V_p^3, \ldots, V_p^{|D|} &= \text{Encoder}\left(X_p^1, X_p^2, X_p^3, \ldots, X_p^{|D|}\right). \end{aligned} \tag{4}$$

Then semantic similarity between X_q and programs in code corpus D can be calculated as the inner product between their semantic representations as Eq. (5).

$$S_{Dense} = \max_{1<n<|D|}\left(V_q^T \cdot V_p^1, V_q^T \cdot V_p^2, \ldots, V_q^T \cdot V_p^{|D|}\right). \tag{5}$$

V_q indicates the semantic representations of X_q. Considering the promising performance of UniXcoder (Huang et al., 2019) in code-related tasks, we use UniXcoder to obtain the semantic vector representations. The retrieved top-N candidate programs, $R_D = X_p^1, X_p^2, X_p^3, \ldots, X_p^N$, are selected based on their similarity scores.

4.3.3. Hybrid retrieval

We explore a hybrid mechanism that combines sparse retrieval and dense retrieval, enhancing the probability of discovering insightful revelations. Sparse retrieval typically focuses on keywords, such as function names, variable names. These keywords can provide useful clues for retrieving relevant programs. Sometimes, NLs and programs may lack matching keywords, leading to sparse retrieval failures. This is when dense retrieval is essential. It allows for more accurate and context-aware retrieval. The retrieved programs produced by these two methods are denoted by $R_S = \{X_p^1, X_p^2, \ldots, X_p^M\}$ and $R_D = \{X_p^1, X_p^2, \ldots, X_p^N\}$, respectively. To discover more insightful revelations, we obtain the final retrieval results for X_q through concatenation.

$$R_h = \left[R_S : R_D\right]. \tag{6}$$

4.4. Analogy

By applying prior knowledge from solved problems, analogy becomes a common practice in software development. For example, when developers need to implement a sorting algorithm, they often retrieve relevant programs from the internet or corpus. Then analyze and analogize these programs to identify insightful revelations. Specifically, they may discover that many sorting algorithms utilize *for loops to iterate an array, compare the elements during iteration, and perform swapping operations*. Equipped with this knowledge, developers can swiftly implement the sorting algorithm by leveraging the prior revelations.

To facilitate analogical learning, we can analogize the ASTs from the retrieved memories to identify interesting revelations. We use ASTs over sequences because ASTs provide a more accurate representation of program syntax, reducing the risk of extracting inaccurate insights. Specifically, sequences only capture the character order of programs, ignoring structure and semantic information, which may lead to grammatically incorrect programs. To mine for interesting, good revelations from the retrieved ASTs, we leverage the frequent tree mining algorithm proposed by Pham et al. (2019), which can identify frequent sub-trees in R_h. Following Pham et al. we define AST as a hierarchical representation of a program's syntactic structure.

Definition 1 (*Abstract Syntax Trees, ASTs*)**.** ASTs are defined as four-tuples $AST = \{V, E, \lambda, \Sigma\}$, representing labeled, ordered, and rooted trees.

- V denotes the set of nodes within an AST.
- $E \subseteq (V \times V)$ denotes the set of edges within an AST.
- Σ represents the set of allowed labels.
- $\lambda : V \rightarrow \Sigma$ is a node label function assigning a label in Σ to a node in V.

When programs share similar functionalities or exhibit comparable patterns, their ASTs tend to contain analogous or overlapping fragments. By analogy in ASTs, it becomes feasible to identify and extract frequent sub-trees that capture revelations of repetitive syntactic structures. As a result, the frequent sub-trees in ASTs can be defined as follows:

Definition 2 (*Frequent Sub-trees*)**.** Given two ASTs $AST_1 = \{V, E, \lambda, \Sigma\}$ and $AST_2 = \{V', E', \lambda', \Sigma'\}$. AST_2 is regarded as a frequent sub-tree of AST_1 if the following conditions are satisfied:

- $\forall v \in V' : v \in V$, which ensures that nodes in AST_2 also exist in AST_1.
- $\forall (u, v) \in E' : (u, v) \in E$, which ensures that edges in AST_2 maintain the same connectivity as in AST_1.
- $\forall v \in V' : \lambda(v) = \lambda'(v)$, which ensures that node labels in AST_2 correspond to the labels in AST_1.
- $\forall u, v \in V' : u < v \Rightarrow \forall u, v \in V : u < v$, which ensures the node order is maintained between AST_2 and AST_1.
- occurrence $(\text{AST}_2) \geq s$, which ensures that the minimum occurrence threshold (s) is met for frequent sub-trees AST_2.

However, in practice, the number of frequent sub-trees can grow exponentially, but the smaller ones may be meaningless, introducing noise to the reasoning module. To address this problem, a common approach is to mine for maximal frequent sub-trees, also known as insightful revelations. The definition of the maximal frequent sub-trees (insightful revelations) is provided in Definition 3.

Definition 3 (*Maximal Frequent Sub-trees*)**.** A frequent sub-tree AST is considered maximal if AST is not dominated by other frequent sub-trees. Therefore, the problem of maximal frequent sub-tree mining is to identify all frequent sub-trees that meet: $\{\text{AST} \in \text{AST}_3 \mid \nexists \text{AST}' \in \text{AST}_3 : \text{AST}' > \text{AST}\}$. $>$ denotes AST is an induced sub-tree of AST′.

Overall, maximal frequent sub-trees are the largest frequent sub-trees that cannot be extended while maintaining their frequency, which can provide insightful revelations into the relationships and dependencies among program elements. By prioritizing maximal frequent sub-trees, we can eliminate the need to analyze smaller frequent sub-trees that are encompassed within these larger structures. This pruning technique effectively reduces redundant computations and enhances the efficiency of the mining process. Compared to enumerating all sub-trees to find frequent sub-trees, mining maximal frequent sub-trees can greatly reduce the computational cost, especially for large ASTs.

However, the number of maximal frequent sub-trees can still be large, consisting of numerous small and uninteresting sub-trees that are challenging to interpret. In response to this, Mens et al. (2021) proposed additional constraints that each sub-tree should satisfy. These constraints focus on node attributes, sub-tree structures, aiming to filter out unimportant sub-trees and retain valuable ones. For a better understanding, below is a brief overview of these constraints.

C0: Minimum and maximum support. It sets upper and lower thresholds for determining the occurrence number of a sub-tree as either frequent or significant.

C1: Allowed root node labels. It ensures that only frequent sub-trees with a certain label as the root node will be considered. This constraint

can reduce the search space by constraining the occurrence of specific root nodes.

C2: Maximum and minimum size constraints. It filters out sub-trees that are either too small or too large, as they are deemed uninteresting or irrelevant.

C3: Blacklisted labels. It includes disallowed labels for frequent sub-tree mining. This constraint plays a vital role in enhancing the quality and relevance of the mined sub-trees.

C4: Obligatory child constraint. It mandates the presence of a specific child node for a given parent node. It helps identify essential structural patterns in the program.

C5: Allowed leaf node labels. It restricts the labels allowed for sub-tree leaves, narrowing the search space by specifying specific permissible labels.

By considering constraints C0–C5, our goal is to maximize the size of frequent sub-trees, as they contain valuable information about ASTs and tend to be more straightforward to interpret. Then the frequent sub-tree mining algorithm works, as depicted in the following steps.

1. Firstly, we apply the FREQT algorithm (Asai et al., 2004) to mine frequent sub-trees under constraints C0–C5. We obtain a collection of frequent sub-trees, denoted as FP.
2. Subsequently, we compute the set of root occurrences (RO) for the frequent sub-trees in FP. The RO of a sub-trees refers to the root node of its occurrences in the retrieved ASTs.

$$\mathrm{RO} = \{\mathrm{occurrence}(\text{sub-trees}) \mid \text{sub-trees} \in \mathrm{FP}\} \tag{7}$$

   To ensure the algorithm can discover all maximal frequent sub-trees, we only retain minimal sets:

$$\mathrm{ROM} = \left\{ r \in \mathrm{RO} \mid \nexists r' \in \mathrm{RO} : r' \subset r \right\} \tag{8}$$

3. Finally, for each sub-tree r in ROM, we identify maximal sub-trees using the mineMaxsub-tree(r) algorithm (Asai et al., 2004).

Theorem 1. *The maximal sub-trees mining process in analogy module is unambiguous, i.e.,* $P(T_q|Z_q) = 1$.

Thus, our problem can be formulated as Eq. (9).

$$X'_p = \operatorname{argmax} \underbrace{P\left(X_p \mid T_q, X_q\right)}_{\textit{Reasoning}} \underbrace{P\left(Z_q \mid X_q, D\right)}_{\textit{Retrieval}}. \tag{9}$$

4.5. Reasoning

The reasoning module utilizes the revelations obtained from the analogy module and applies them to program synthesis. It is worth noting that the insightful revelations $\mathrm{T_q}$ mined in Section 4.4 can bridge the gap between natural language and executable programs by providing valuable insights into the structure and behavior of the target program $\mathrm{X_p}$. While our approach is model-agnostic and has broad applicability, we implement it on top of a syntax-based method (Xie et al., 2021). Following Xie's work (Xie et al., 2021), we employ a knowledge distillation-based framework, involving a teacher model utilizing breadth-first decoding and a student model utilizing pre-order decoding. Both the two models adopt the SEQ2TREE architecture, where the encoder processes both natural language and revelations, and the decoder employs a tree-structured decoder to generate a sequence of actions. These actions can be further transformed into executable programs through a well-defined conversion process.

Given that the output of the reasoning module is action sequences, directly using frequent sub-trees may introduce noise and hinder the module's performance. To address this, an advanced approach involves processing the frequent sub-trees into action sequences, enabling the provision of refined and meaningful prior knowledge to the reasoning module. This enhancement leads to more effective and reliable program synthesis. For each sub-tree t^j in T_q, we have $\mathrm{ta}^j = f(\mathrm{t}^j) = \{\mathrm{ta}^j_1, \mathrm{ta}^j_2, \ldots, \mathrm{ta}^j_n\}$. ta^j denotes the action set corresponding to sub-tree t^j. f is a deterministic function that generates a sequence of actions from a sub-tree.

4.5.1. Encoder

NL Encoder. The NL encoder in both the teacher and student models, is identical and follows the same structure as TRANX. For a NL description X_q, we use a BiLSTM encoder to learn the origin information within it. BiLSTM encodes the sequence of X_q into hidden states h_i. For simplification, we denote the BiLSTM unit as Eq. (10).

$$\begin{aligned} h_i^{\rightarrow} &= \mathrm{LSTM}\left(h_{i-1}^{\rightarrow}, E\left(X_q^i\right)\right), \\ h_i^{\leftarrow} &= \mathrm{LSTM}\left(h_{i-1}^{\leftarrow}, E\left(X_q^i\right)\right), \\ h_i &= \left[h_i^{\rightarrow} : h_i^{\leftarrow}\right], \\ C_i^{NL} &= \sum a_{ij} h_i. \end{aligned} \tag{10}$$

$E(X_q^i)$ denotes the embedding layer of X_q^i. $h_i^{\rightarrow}$ and $h_i^{\leftarrow}$ indicate the hidden states generated in two directions, respectively. h_i represents the hidden states in both forward and backward directions at time step i. [:] denotes the concatenation operation. C_i^{NL} refers to the context vectors.

Revelation Encoder. The revelation encoder aims to encode the action sequences ta^i corresponding to the maximal frequent sub-trees. This encoding process aims to capture the crucial information contained within these sub-trees and facilitate their transfer to the decoder. However, due to the fundamental disparity in traversal methods between the teacher model (breadth-first) and the student model (pre-order), integrating the revelation encoder into the teacher model could introduce conflicts and distort the inherent characteristics of the respective traversal approaches. To ensure the seamless transfer of knowledge and maintain the fidelity of both models' traversal techniques, a judicious decision was made to not consider the inclusion of the revelation encoder in the teacher model.

Therefore, for an action sequence $\mathrm{ta}^j = \{\ \mathrm{ta}^j_1, \mathrm{ta}^j_2, \ldots, \mathrm{ta}^j_n\ \}$, the revelation encoder also encodes it based on BiLSTM, and returns context vectors $C_i^{\mathrm{Revelation}}$. Note that although the NL encoder and the revelation encoder have the same structure, they do not share parameters with each other.

$$\begin{aligned} m_i^{\rightarrow} &= \mathrm{LSTM}\left(s_{i-1}^{\rightarrow}, E\left(\mathrm{ta}_i^j\right)\right), \\ m_i^{\leftarrow} &= \mathrm{LSTM}\left(s_{i-1}^{\leftarrow}, E\left(\mathrm{ta}_i^j\right)\right), \\ s_i &= \left[m^{\rightarrow} : m_i^{\leftarrow}\right], \\ C_i^{\mathrm{Revelation}} &= \sum \beta_{ij} m_i. \end{aligned} \tag{11}$$

Semantic Fusing. The semantic fusion layer offers an efficient mechanism for integrating the two semantic vectors C_i^{NL} and $C_i^{\mathrm{Revelation}}$. Following Yang's work (Yang et al., 2019), we utilize three different fusing operations:

$$\begin{aligned} Y_1 &= \left[C_i^{\mathrm{NL}} : C_i^{\mathrm{Revelation}}\right], \\ Y_2 &= \left[C_i^{\mathrm{NL}} : C_i^{\mathrm{NL}} - C_i^{\mathrm{Revelation}}\right], \\ Y_3 &= \left[C_i^{\mathrm{NL}} : C_i^{\mathrm{NL}} \circ C_i^{\mathrm{Revelation}}\right]. \end{aligned} \tag{12}$$

The concatenation operation of two vectors is denoted by [:]. – denotes the subtraction operation of two vectors, which emphasizes the differences between two semantic vectors. ∘ denotes the dot product operation of two vectors, which emphasizes the similarity between two semantic vectors.

Next, we combine the three vectors Y_1, Y_2, Y_3, to obtain the output C_t, which is then fed into the decoder.

$$C_t = \tanh\left(w\left[Y_1, Y_2, Y_3\right] + b\right) + C_i^{\mathrm{NL}}. \tag{13}$$

4.5.2. Decoder

The decoder is also a BiLSTM, at each time step t, its hidden state is given by Eq. (14).

$$z_t = \mathrm{LSTM}\left(\left[a_{t-1} : z'_{t-1} : p_t\right], z_{t-1}\right). \tag{14}$$

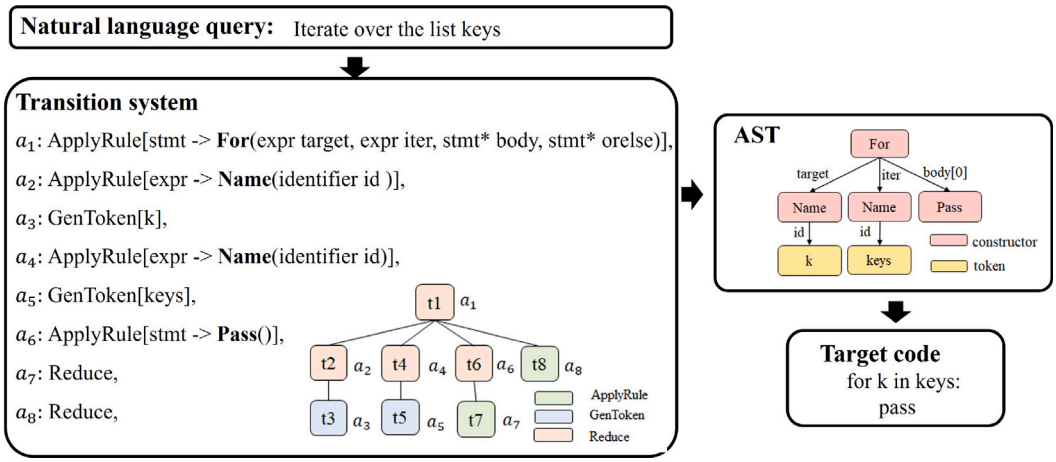


Fig. 3. An Example of Program Synthesis using TRANX.

$a_{t\text{-}1}$ denotes the action at time step t-1. The attentional vector z_t' can be defined as $z_t' = \tanh(w_2[C_t : z_t])$. $z_{t\text{-}1}$ denotes the hidden state at time step t-1. p_t encodes the information of the parent action.

Action Properties. Fig. 3 provides an illustration of TRANX conversion conducted by a SEQ2TREE model. Following TRANX's work (Yin and Neubig, 2018), we choose one of three actions to construct a non-terminal node of the partial AST at each time step. These actions are based on the ASDL grammar, including APPLYCONSTR[c], REDUCE and GETTOKEN[v]. APPLYCONSTR[c] actions apply a constructor c to a corresponding composite frontier field that has the same type as c. REDUCE actions mark the end of generating child values. GETTOKEN[v] actions assign a token v to a primitive frontier field.

At each time step, either an APPLYCONSTR[c] is applied to create a new AST node, or a REDUCE is used to terminate AST node expansion. Formally, we define the probability of using the APPLYCONSTR[c] /REDUCE as:

$$p\left(a_t = \text{APPLYCONSTR}[c]/\text{REDUCE} \mid a_{<t}, X_q\right) = \text{softmax}\left(E(c)^T W_a z_t\right). \tag{15}$$

where $E(c)$ is the embedding of the constructor c.

Then, for GENTOKEN[v], we adopt a hybrid approach that combines both generation and copying. Specifically, the token v can be either generated directly or copied from the input NL description. The GENTOKEN[v] action probability can be decomposed into two parts.

$$P\left(a_t = \text{GENTOKEN}[\text{v}] \mid a_{<t}, X_q\right) = P\left(\text{gen} \mid a_{<t}, X_q\right) P_{\text{gen}}\left(v \mid gen, a_{<t}, X_q\right) + \left(1 - P\left(\text{gen} \mid a_{<t}, X_q\right)\right) P_{\text{copy}}\left(v \mid \text{copy}, a_{<t}, X_q\right). \tag{16}$$

Therefore, the decoder can decide whether to copy the token in the NL according to the context.

4.5.3. Training

To improve the model's performance, a mutual distillation framework is applied to jointly train the teacher and student models. In addition to the traditional maximum likelihood estimation (MLE) loss applied to the training data, we incorporate a Kullback–Leibler (KL) loss to assess the divergence in action prediction distribution between the teacher model and another student model. Thus, the objective function for updating the parameters of both models is formulated as follows Eqs. (17) and (18).

$$J(D,\theta) = \sum_{(\mathbf{X_q},\mathbf{a})\in D} \left\{ J_{\text{MLE}}\left(\mathbf{X_q}, \mathbf{a}; \theta\right) + \frac{\lambda}{T} \cdot \sum_{n\in \mathbf{Z}} \text{KL}\left(p\left(a'_{t'(n)} \mid a'_{<t'(n)}, \mathbf{x_q}; \theta'\right) \parallel p\left(a_{t(n)} \mid a_{<t(n)}, \mathbf{X_q}; \theta\right)\right) \right\}, \tag{17}$$

$$J\left(D,\theta'\right) = \sum_{(\mathbf{x_q},\mathbf{a}')\in D} \left\{ J_{\text{MLE}}\left(\mathbf{x_q}, \mathbf{a}'; \theta'\right) + \frac{\lambda}{T} \cdot \sum_{n\in \mathbf{Z}} \text{KL}\left(p\left(a_{t(n)} \mid a_{<t(n)}, \mathbf{X_q}; \theta\right) \parallel p\left(a'_{t'(n)} \mid a'_{<t'(n)}, \mathbf{X_q}; \theta'\right)\right) \right\}. \tag{18}$$

where θ and θ' denote the parameters of the teacher and student models. $t(n)$ and $t'(n)$ denote the time steps during the traversals of AST z with the teacher and student model. KL(||) represents the KL divergence. λ is the weight coefficient. The training objective $J_{\text{MLE}}(D;\theta)$ can be defined as Eq. (19).

$$J_{\text{MLE}}(\text{X}_\text{q}, \text{a}; \theta) = -\frac{1}{T}\sum_{t=1}^{T} \log p\left(a_t \mid a_{<t}, \text{X}_\text{q}; \theta\right). \tag{19}$$

5. Experiments

In this section, we aim to answer the following four research questions by conducting extensive experiments on two commonly used datasets.

RQ1: How effective is SynthoMinds compared to state-of-the-art baselines?

RQ2: How does the retrieval module enhance the program synthesis tasks?

RQ3: How does the analogy module help the program synthesis tasks?

RQ4: What are the factors contributing to the performance improvement?

5.1. Datasets

We carry out experiments on the following datasets:

Django (Oda et al., 2015). It consists of 18,805 lines of Python code extracted from the Django Web framework. Each line of code in Django is accompanied by a corresponding NL description. The code in Django encompasses a wide range of practical Python applications, including string manipulation, input/output operations, exception handling, and more. Following Yin and Neubig (2018) and Xie (Xie et al., 2021), we divide Django into 16000 for training, 1000 for validation, and 1805 for testing.

CONALA (Yin et al., 2018). It consists of 2879 NL questions and related Python solutions. It is divided into 2379 training examples and 500 test examples, all of which are manually annotated. Unlike DJANGO, CONALA encompasses a wider range of real-world NL issued by programmers with diverse intents. Consequently, these examples in CONALA present greater challenges due to their extensive coverage and complex target meaning representations.

Table 1
Dataset statistics for Django and CONALA.

Datasets	Django	CONALA
Train	16 000	2179
Validate	1000	200
Test	1805	500
Avg. tokens in NL	14.3	10.2
Avg. characters in code	41.1	39.4
Avg. size of AST (nodes)	17.2	15.4
Avg. size of actions	14	23
Max. size of actions	1067	219
Min. size of actions	1	5

Table 2
Statistics for associated grammars for Django and CONALA.

Datasets	Django	CONALA
Productions	82	96
Types	17	22
Fields	65	68
Vocabulary size	574	483

The data statistics for Django and CONALA are shown in Table 1. Avg/Max/Min. Size of actions refers to the average, maximum, and minimum size or length of actions corresponding to program snippets within a dataset. The grammatical data statistics for Django and CONALA are shown in Table 2. Productions refers to the number of productions in the context-free grammar used to describe the syntactical structures of the programming language.

5.2. Baselines

To demonstrate the effectiveness of SynthoMinds, we have selected the following state-of-the-art baselines from the program synthesis domain:

- SEQ2SEQ (Song et al., 2021). We implemented a SEQ2SEQ model, which incorporates attention mechanisms. The encoder is a BiLSTM, while the decoder is a LSTM.
- Yin17 (Yin and Neubig, 2017). It converts NL into an AST representation, leveraging the inherent syntax structure of the programming language. Subsequently, the AST is transformed into target code.
- TRANX (Yin and Neubig, 2018). It employs an encoder that learns the semantic representations of an input NL, followed by a tree-structured decoder generates a sequence of actions, which corresponds to the pre-order traversal of an AST and can be transformed into the target code.
- RECODE (Hayati et al., 2018). It is a retrieval-based approach that extends to include tree structure generation. It collects n-grams from the retrieved program's output, fostering the model's prediction of code actions.
- Reranker (Yin and Neubig, 2019). It aims to optimize the ranking of the top N predicted program solutions based on their quality or relevance. By applying ranking methods, the reranker can improve the selection of the most suitable program candidates among the initially generated set.
- ML-TRANX (Xie et al., 2021). It facilitates simultaneous improvement of models using different traversal methods (e.g., pre-order and breadth-first) through mutual learning-based model training.
- Tranx-R2L (Jiang et al., 2021). It enhances the SEQ2TREE model by incorporating a context-based branch selector that can dynamically identify the appropriate approach for multi-branch nodes.
- ASED (Jiang et al., 2022). It utilizes AST information and employs multi-task learning to predict both current and future actions. It greatly expands upon the TRANX decoder by incorporating extensive history and future action exploitation.

Table 3
The performance of SynthoMinds in comparison with various baselines.

Model	Django	CONALA	
	EM	EM	BLEU
SEQ2SEQ	13.9	–	–
Yin17	71.6	–	–
RECODE	72.8	1.4	20.6
TRANX	77.2	2.0	24.4
ASED	76.7	1.8	25.3
Reranker	78.2	1.6	20.3
Tranx-R2L	78.5	2.4	25.7
ML-TRANX	78.6	1.8	25.3
REDCODER	38.81	0.0	21.4
SynthoMinds	**79.9**	**2.8**	**29.7**

- REDCODER (Parvez et al., 2021). It is a retrieval-augmented framework that leverages developers' past code to enhance code generation and summarization models.

Out of the mentioned baselines, YIN17, TRANX, ML-TRANX, ASED, and TRANX-R2L all enhance the prediction accuracy of the model by incorporating structural information. RECODE and REDCODER enhance performance by incorporating retrieval information.

5.3. Evaluation metrics

Similar to previous works, we evaluated our results using exact match and BLEU metrics, as suggested by Yin and Neubig (2018).

BLEU (Bilingual Evaluation Understudy). BLEU (Papineni et al., 2002) measures the overlap between the reference sentence and the candidate sentence using n-grams. It is described as the geometric mean of the precision scores for n-gram matching.

$$\begin{aligned} \mathrm{BP} &= \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \le r \end{cases}, \\ \mathrm{BLEU} &= \mathrm{BP} * \exp\left(\sum_{n=1}^{N} w_n \log p_n\right). \end{aligned} \tag{20}$$

where c is the candidate sentence length. r is reference sentence length. p_n signifies the n-gram matching precision scores. N is 4 in our experiments. BP is the brevity penalty.

EM (Exact Match). EM (Yin and Neubig, 2017) measures the percentage of predictions that have the identical token sequence as the ground truth. It offers a precise evaluation of the effectiveness of the generated program, which is calculated as Eq. (21).

$$\mathrm{EM} = \frac{1}{n}\sum_{i=1}^{n} \mathrm{EM}_i. \tag{21}$$

5.4. Experimental results and discussion

5.4.1. SynthoMinds vs. Baselines

In this section, our research objective is to demonstrate that the SynthoMinds outperforms current baselines.

> **RQ1:** How effective is SynthoMinds compared with the state-of-the-art baselines?

For RQ1, our aim is to investigate the effectiveness of SynthoMinds and quantify the performance improvement it can achieve over the nine state-of-the-art baselines.

In Table 3, we show the comparison results between SynthoMinds and the baselines. We leverage the EM metric to measure the differences between the generated code and the reference code in program synthesis. This metric allows us to determine whether the generated

Table 4
Effect of retrieval module on SynthoMinds performance.

Model	Django	CONALA	
	EM	EM	BLEU
$\text{SynthoMinds}_{\text{no\_retrieval}}$	78.6	1.8	25.3
$\text{SynthoMinds}_{\text{sparse}}$	78.9	1.9	25.4
$\text{SynthoMinds}_{\text{CodeBERT}}$	79.3	2.1	26.1
$\text{SynthoMinds}_{\text{CodeT5}}$	79.4	2.4	27.3
$\text{SynthoMinds}_{\text{UniXCoder}}$	79.5	2.4	27.7
$\text{SynthoMinds}_{\text{sparse+CodeBERT}}$	79.5	2.5	26.9
$\text{SynthoMinds}_{\text{sparse+CodeT5}}$	79.6	2.7	29.3
$\text{SynthoMinds}_{\text{sparse+UniXCoder}}$	79.9	2.8	29.7

Table 5
Effect of analogy module on SynthoMinds performance.

Model	Django	CONALA	
	EM	EM	BLEU
$\text{TRANX}_{\text{no\_analogy}}$	72.3	1.6	20.4
$\text{ASED}_{\text{no\_analogy}}$	72.9	1.7	20.9
$\text{SynthoMinds}_{\text{no\_analogy}}$	73.4	1.9	21.6
$\text{SynthoMinds}_{\text{teacher\_analogy}}$	74.5	2.1	24.7
$\text{TRANX}_{\text{analogy}}$	77.9	2.2	25.2
$\text{ASED}_{\text{analogy}}$	77.6	2.0	26.7
$\text{SynthoMinds}_{\text{teacher+student\_analogy}}$	78.7	2.4	25.1
$\text{SynthoMinds}_{\text{student\_analogy}}$	79.9	2.8	29.7

code precisely matches the reference code. Therefore, for models that did not report BLEU-4 values in their studies, we chose not to replicate their results. In Table 3, we have observed that SEQ2SEQ exhibits the weakest performance among all the baselines. This is because SEQ2SEQ directly uses NL description as decoder input, lacking explicit mechanisms for syntax and semantic alignment. This difficulty in capturing both syntactic and semantic aspects of code leads to suboptimal results. Both Yin17 and TRANX decompose AST generation into tree-constructing actions. But Yin17 performs worse. This is because TRANX uses ASTs as the intermediate meaning representation to capture the syntax and semantics of the code. RECODE combines information retrieval with program synthesis, but the performance improvement is limited. REDCODER's performance decline may be attributed to the inclusion of irrelevant code snippets. ASED, Tranx-R2L, and ML-TRANX outperform TRANX, highlighting the significant influence of AST traversal sequence on program synthesis model performance. A key difference between ASED and ML-TRANX is that the latter introduces additional structural information in the AST, but the former is less effective.

Our experimental results demonstrate that SynthoMinds outperforms all baseline models, validating the efficiency of the Retrieval-Analogy-Reasoning framework. This proves that insightful revelations in retrieved memories contain valuable and reusable information compared to noisy program sequences. Therefore, it is reasonable to mine frequent sub-trees within retrieved memories.

5.4.2. Ablation study

RQ2: How does the retrieval module enhance the program synthesis tasks?

For RQ2, our aim is to assess the influence of retrieval module (dense retrieval and sparse retrieval) on experimental results. By combining these two retrievals, we intend to explore the potential advantages and synergistic effects that can be attained. We evaluate the impact of adding the retrieval module in four different scenarios: (1) no retrieval and analogy, (2) sparse retrieval only, (3) dense retrieval only, employing CodeBERT, CodeT5, and UniXcoder, (4) combined dense and sparse retrievals, with dense retrieval using CodeBERT, CodeT5, and UniXcoder.

In Table 4, we present our experimental results, highlighting the critical role of the retrieval module in SynthoMinds. We can observe that neglecting the retrieved information leads to a performance decrease ranging from 79.9 to 78.6 in EM metric on Django dataset. Regarding the combination of sparse and dense retrieval, we find that neglecting either sparse or dense retrieval, or solely relying on sparse retrieval, results in decreased performance. Notably, we observe that the performance of SynthoMinds does not fluctuate much without considering sparse retrieval. However, its inclusion brings about a notable improvement, particularly in terms of EM metric. This underscores the value of leveraging diverse information sources to enhance program synthesis Finally, employing CodeBERT, CodeT5, and UniXcoder as dense retrieval has a relatively minor impact on SynthoMinds' performance. This is due to the similarity in top-20 retrieval results produced by these models.

RQ3: How does the analogy module help the program synthesis tasks?

In addressing RQ3, our aim is to evaluate the effectiveness of the analogy module. This module is designed to leverage the analogy learning between different ASTs to enhance the reasoning module. By examining the impact of the analog module of SynthoMinds, our objective is to gain insights into its effectiveness and potential for enhancing program synthesis models. We mainly examine five different scenarios to assess the effect of the analogy module: (1) a baseline model without the analogy module ($\text{SynthoMinds}_{\text{no\_analogy}}$), (2) the teacher model with the analogy module ($\text{SynthoMinds}_{\text{teacher\_analogy}}$), (3) the student model with the analogy module ($\text{SynthoMinds}_{\text{student\_analogy}}$), and (4) the combined teacher–student model with the analogy module ($\text{SynthoMinds}_{\text{teacher+student}}$), (5) baselines both with the analogy module ($\text{TRANX}_{\text{analogy}}$, $\text{ASED}_{\text{analogy}}$) or without it ($\text{TRANX}_{\text{no\_analogy}}$, $\text{ASED}_{\text{no\_analogy}}$).

We present the experimental results in Table 5. and draw the following key observations: (1) The lack of the analogy module has hampered its performance, as retrieval results often include some irrelevant information, negatively impacting the reasoning module results. This highlights the importance of effective mining of insightful revelations. (2) $\text{SynthoMinds}_{\text{student\_analogy}}$ demonstrates superior performance compared to $\text{SynthoMinds}_{\text{teacher\_analogy}}$, which can be attributed to the use of deep-first traversal in the student model. This traversal strategy typically aligns more closely with human cognition. It first considers the overall structure and progressively delves into finer details, leveraging its recursive nature to yield more consistent and organized ASTs. In contrast, the teacher model employs breadth-first traversal, which may introduce less relevant sub-trees, potentially adding noise to the generated code. Therefore, our experimental results are based on the performance of the student model. (3) The inclusion of the analogy module in both the teacher and student models does not result in a substantial performance improvement. This can be attributed to the potential negative impact of integrating the analog module on the teacher model's performance, which consequently influences the overall performance of the SynthoMinds model. (4) The introduction of retrieval and analogy modules in TRANX and ASED models resulted in performance improvements. This indicates the general applicability of our retrieval-analogy-reasoning framework, suggesting that the analogy module can be substituted with other more advanced models.

5.4.3. Visualization analysis

RQ4: What are the factors contributing to the performance improvement?

To answer RQ4 and explain the effectiveness of SynthoMinds, we conducted the following experiments:

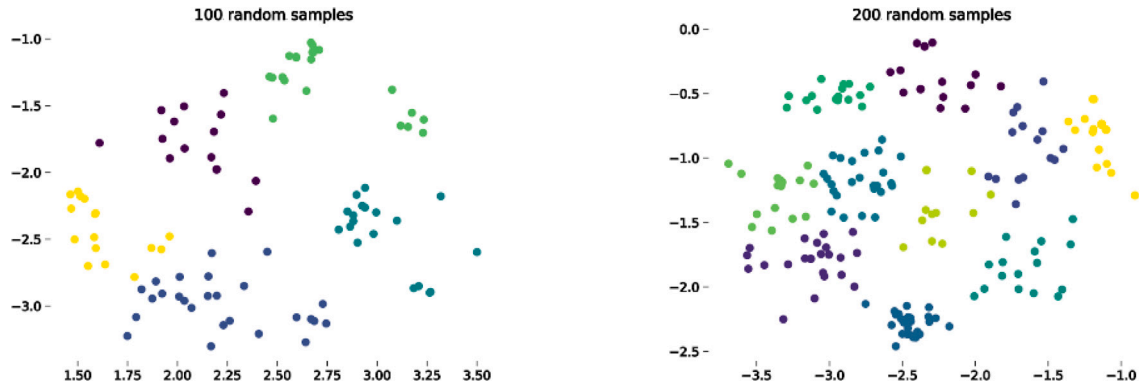


Fig. 4. Distribution plot of 100/200 random samples on Django dataset.

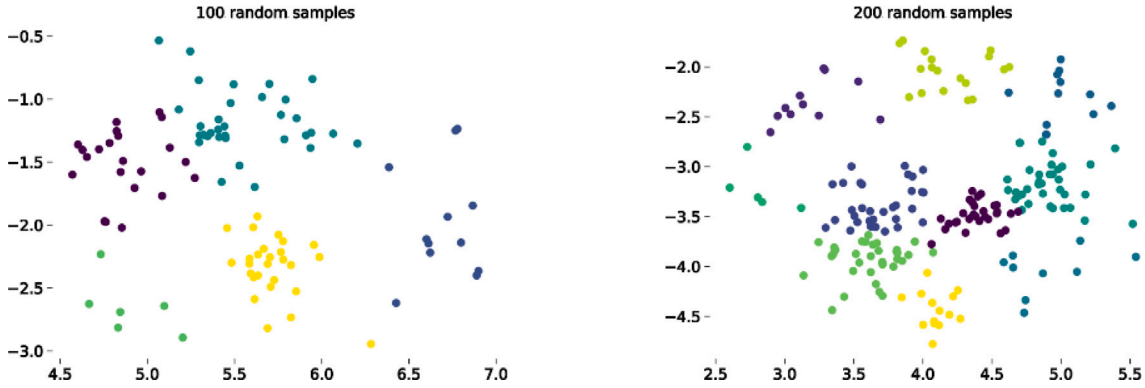


Fig. 5. Distribution plot of 100/200 random samples on CONALA dataset.

- To evaluate the necessity of retrieval module, we randomly sample program examples from the Django and CONALA datasets and apply t-SNE for dimensionality reduction and visualization. The random sampling procedure was employed to ensure the selection process was free from bias. The experimental results are shown in Figs. 4 and 5.
  We randomly selected two subsets of samples from the Django and CONALA datasets, one with 100 samples and another with 200 samples. From Figs. 4 and 5, we can observe that certain points, representing specific programs, exhibited clusters of multiple points surrounding them. This observation indicated the presence of groups of similar programs within the dataset, suggesting shared semantic or contextual characteristics among them. These similarities could be attributed to shared functionality, algorithmic approaches, or coding patterns. By leveraging the retrieval module, users can effectively retrieve and access programs that exhibit similar semantics. This holds valuable insights for program synthesis.
- To further investigate the common characteristics among the clustered points, we randomly select 10 points from Fig. 5 that are clustered together. Subsequently, we parse these 10 programs into ASTs and traverse their nodes. We then extract frequent itemsets of AST nodes, representing patterns of recurring node combinations. To visualize the relationships between these frequent itemsets, we construct a co-occurrence matrix. Finally, we employ this matrix to generate a heat map, providing insights into shared functionality or coding patterns shared by programs within the same cluster. Using different shades of color to represent the varying degrees of co-occurrence.
  From Fig. 6, we observe a darker color in the heat map for the combination *(If, Name, Pass, Compare, Module)*, indicating a higher frequency of occurrence. This combination suggests a recurring pattern where *an if statement (If) is followed by a variable name (Name), a pass statement (Pass), a comparison operation (Compare), and a module (Module)*. Such information provides valuable prior knowledge for program synthesis tasks. This indicates the necessity of the analogy module.
- We visually analyzed SKCoder's (Li et al., 2023) sketcher module on the CONALA dataset. Fig. 7 displays the frequency distribution of sketches and oracle-sketches. "Sketch" refers to those generated by the Sketcher module, while "oracle-sketch" denotes the oracle sketches. Our analysis revealed that SKCoder generates a narrow range of sketches on CONALA dataset. These sketches exhibited a repetitive pattern, dominated by a few specific types, such as ["<pad>", "<pad>", "<pad>", "<pad>", "<pad>"], ["<pad>", "<pad>", "<pad>", "<pad>"], and ["<pad>", "<pad>"]. This narrow range of sketches suggested a potential limitation in the diversity and richness of the generated code representations. Furthermore, our visual inspection indicated that these sketches appear to have limited impact on the generation module and may introduce noise. In contrast, as shown in Fig. 6, SynthoMinds mines insightful revelations, significantly aiding the reasoning module. Hence, the subtle difference in accuracy between SKCoder and SynthoMinds is primarily observed within the generation module rather than in the sketcher module. The superior performance of the generation module can be attributed to its significantly higher parameter scale, exceeding that of SynthoMinds by nearly 60 times. We hypothesize that this discrepancy may stem from differences in the application scope of SKCoder and SynthoMinds. SKCoder excels with larger-scale code datasets, while SynthoMinds shines with smaller datasets, without requiring complex models or extensive computational resources.

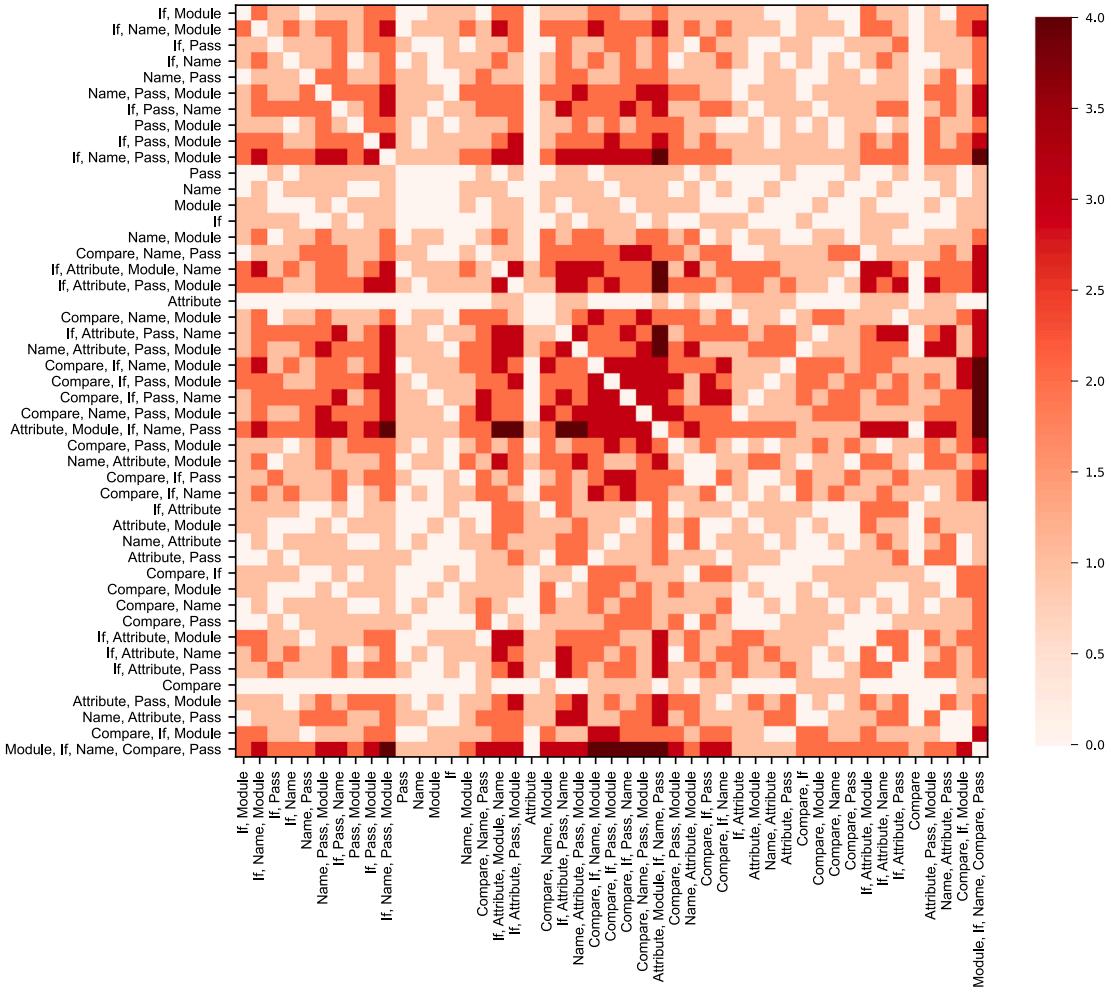
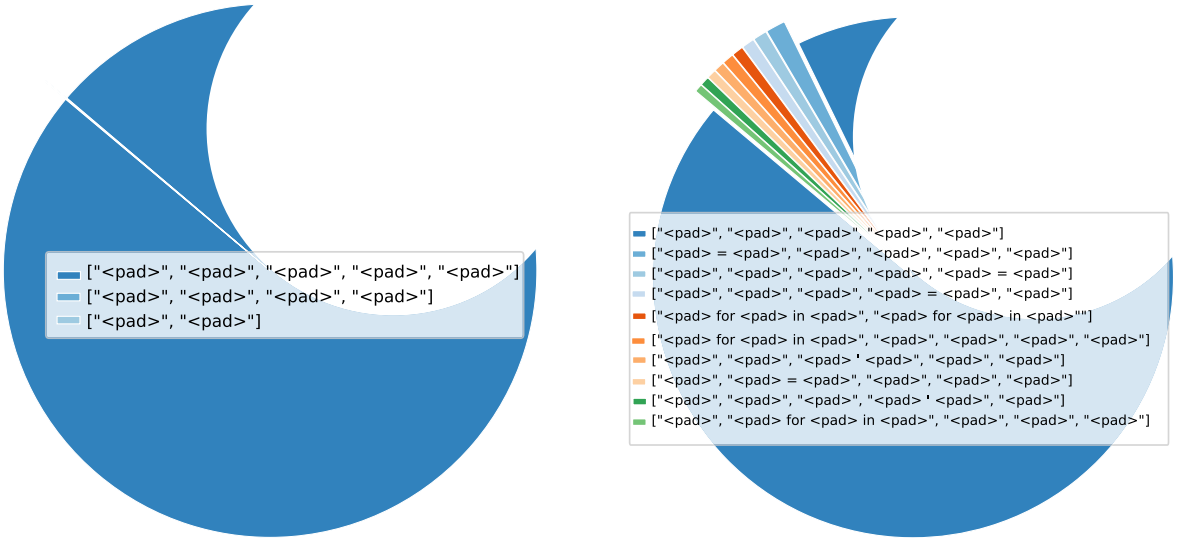


Fig. 6. Heat map of Frequent Itemset in AST Nodes.



(a) Frequency Distribution of Sketch on CONALA Dataset

(b) Frequency Distribution of Oracle-Sketch on CONALA Dataset

Fig. 7. Visual Analysis of SKCoder's Sketch Module on CONALA Dataset.

Table 6
A macro-level case study of SynthoMinds.

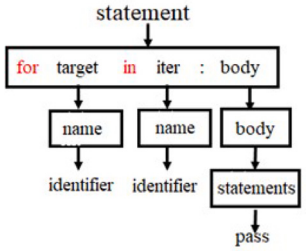
Problem	Iterate the elements of the 'filelist' using the variable 'fname'.
Retrieved memories	For filename in gen_filenames(): pass For ldir in dirs: pass For f in files: pass For f in file_field_list: pass
Insightful revelations	
ML-TRANX results	For filelist in frame:
SynthoMinds results	For filename in filelist: pass

Table 7
The result of human evaluation.

Method	Correctness	Readability	Maintainability
TRANX	3.66 (±1.43)	3.80 (±1.21)	2.12 (±1.38)
ASED	3.82 (±1.27)	3.96 (±1.05)	1.90 (±1.50)
ML-TRANX	3.83 (±1.50)	4.01 (±1.13)	2.63 (±1.23)
SynthoMinds	3.95 (±1.45)	4.03 (±1.25)	2.89 (±1.45)

5.5. Case study

To better understand how SynthoMinds works in the program synthesis task, we exploited macro-level case study. Table 6 provides a simple example of how the insightful revelations help the SynthoMinds generate the correct program. As shown in Table 6, which describes a problem about "*Iterate the elements of the filelist using the variable fname*". ML-TRANX's prediction is incorrect when it tries to solve the problem through an inductive approach. It appears that ML-TRANX lacks the fundamental understanding of for loops and often generates programs with incorrect syntax. Therefore, we attempt to solve the problem using an analogy and reasoning approach, which yields the correct program. By analogizing the retrieved memories, we observe that the SynthoMinds can identify the common patterns among them. We can conclude that, harnessing the retrieval and analogy aspects yields improved interpretation programs, resulting in enhanced generation accuracy.

5.6. Human evaluation

We use human evaluation to measure the quality of generated programs. The randomly selected 30 samples from the test set are given to five master students. Each student was asked to rate generated programs on a scale of 0–5 based on three aspects: (1) how well the generated programs solve the problem (correctness), (2) how readable is the generated program (readability), (3) how easy is it to understand and modify the generated code (maintainability).

As shown in Table 7, SynthoMinds shows statistically higher degrees of correctness, readability, and maintainability than the other models. These results indicate that programs generated from SynthoMinds are more likely to be accurate, easier to read, and easier to maintain.

6. Threat to validity

We have identified potential threats to the validity of SynthoMinds, which include: (1) Data quality. The quality of the data used in our experiments may impact the performance of SynthoMinds. Since the datasets were automatically collected from GitHub and Stack Overflow, there is a possibility that the data may contain quality issues or errors. To alleviate this threat, we carefully select and process two widely used datasets, i.e., Django, COANLA. Both datasets are extracted from the real world. (2) Programming languages. Different programming languages may have a significant impact on the performance of the model due to variations in their syntax and complexity. It should be noted that we only evaluated SynthoMinds in Python. However, it can still be useful for other programming languages. In the future, we plan to extend SynthoMinds to other programming languages. (3) Evaluation metrics. We acknowledge that our primary evaluation metric, the BLEU and EM score, may not always be consistent with human evaluation. However, it is widely used in the literature for evaluating program synthesis models. Moreover, we have conducted a case study and human evaluation to verify the competitiveness of SynthoMinds.

7. Conclusion and future work

In this paper, we introduce SynthoMinds, a novel approach that combines information retrieval and analogy learning to enhance generative models for program synthesis tasks. It mainly consists of three key modules: retrieval, analogy, and reasoning. Given a natural language description, SynthoMinds generated insightful revelations based on retrieved memories. These insightful revelations serve as a pivotal knowledge resource, guiding the program synthesis process with precision. By leveraging these revelations, the generated program is directed towards a more refined and comprehensive solution, enhancing its directionality.

In the future, we plan to extend our work in two directions. Firstly, we intend to use large language models within the reasoning module, followed by the joint training of both the retrieval and reasoning modules within the SynthoMinds framework. Additionally, we plan to add more programming languages, such as C++ and Java, to our approach.

CRediT authorship contribution statement

Qianwen Gou: Methodology, Writing – original draft, Writing – review & editing. **Yunwei Dong:** Conceptualization, Funding acquisition, Writing – review & editing. **Qiao Ke:** Conceptualization, Funding acquisition, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work is supported by the Major Program of the National Natural Science Foundation of China with No. 62192733, No. 62192730 and the National Natural Science Foundation of China with No. 12201498.

References

Alokla, A., Gad, W., Nazih, W., Aref, M., M.Salem, A.-B., 2022. Retrieval-based transformer pseudocode generation. Mathematics 10, 604.

Asai, T., Abe, K., Kawasoe, S., Sakamoto, H., Arimura, H., Arikawa, S., 2004. Efficient substructure discovery from large semi-structured data. IEICE Trans. Inf. Syst. 87-D (12), 2754–2763.

Bajracharya, S.K., Ossher, J., Lopes, C.V., 2010. Leveraging usage similarity for effective retrieval of examples in code repositories. In: Roman, G., van der Hoek, A. (Eds.), Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010. Santa Fe, NM, USA, November 7-11, 2010, ACM, pp. 157–166.

Bui, N.D., Yu, Y., Jiang, L., 2019. SAR: Learning cross-language API mappings with little knowledge. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 796–806.

Cadavid, H., Andrikopoulos, V., Avgeriou, P., 2023. Improving hardware/software interface management in systems of systems through documentation as code. Empir. Softw. Eng. 28 (4), 100.

Chen, B., Abedjan, Z., 2021. RPT: Effective and efficient retrieval of program translations from big code. In: 43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings. ICSE Companion 2021, Madrid, Spain, May 25-28, 2021, IEEE, pp. 252–253.

Chen, Q., Pailoor, S., Barnaby, C., Criswell, A., Wang, C., Durrett, G., Dillig, I., 2022. Type-directed synthesis of visualizations from natural language queries. Proc. ACM Program. Lang. 6 (OOPSLA2), 532–559.

Ciniselli, M., Cooper, N., Pascarella, L., Mastropaolo, A., Aghajani, E., Poshyvanyk, D., Di Penta, M., Bavota, G., 2021. An empirical study on the usage of transformer models for code completion. IEEE Trans. Softw. Eng. 48 (12), 4818–4837.

Dahal, S., Maharana, A., Bansal, M., 2021. Analysis of tree-structured architectures for code generation. In: Zong, C., Xia, F., Li, W., Navigli, R. (Eds.), Findings of the Association for Computational Linguistics. ACL/IJCNLP 2021, Online Event, August 1-6, 2021, In: Findings of ACL, vol. ACL/IJCNLP 2021, Association for Computational Linguistics, pp. 4382–4391.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., Roy, S., 2016. Program synthesis using natural language. In: Dillon, L.K., Visser, W., Williams, L.A. (Eds.), Proceedings of the 38th International Conference on Software Engineering. ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, pp. 345–356. http://dx.doi.org/10.1145/2884781.2884786.

Dong, Y., Jiang, X., Liu, Y., Li, G., Jin, Z., 2022. CodePAD: Sequence-based code generation with pushdown automaton. arXiv preprint arXiv:Arxiv-2211.00818.

Gavran, I., Darulova, E., Majumdar, R., 2020. Interactive synthesis of temporal specifications from examples and natural language. Proc. ACM Program. Lang. 4 (OOPSLA), 201:1–201:26.

Gou, Q., Dong, Y., Wu, Y., Ke, Q., 2024a. RRGcode: Deep hierarchical search-based code generation. J. Syst. Softw. 211, 111982.

Gou, Q., Dong, Y., Wu, Y., Ke, Q., 2024b. Semantic similarity-based program retrieval: A multi-relational graph perspective. Front. Comput. Sci. 18 (3), 183209.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (Eds.), Proceedings of the 40th International Conference on Software Engineering. ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp. 933–944.

Hayati, S.A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., Neubig, G., 2018. Retrieval-based neural code generation. In: Riloff, E., Chiang, D., Hockenmaier, J., Tsujii, J. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Brussels, Belgium, October 31 - November 4, 2018, Association for Computational Linguistics, pp. 925–930.

Huang, H., Liang, Y., Duan, N., Gong, M., Shou, L., Jiang, D., Zhou, M., 2019. Unicoder: A universal language encoder by pre-training with multiple cross-lingual tasks. In: Inui, K., Jiang, J., Ng, V., Wan, X. (Eds.), Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019, Association for Computational Linguistics, pp. 2485–2494.

Jiang, H., Song, L., Ge, Y., Meng, F., Yao, J., Su, J., 2022. An AST structure enhanced decoder for code generation. IEEE ACM Trans. Audio Speech Lang. Process. 30, 468–476.

Jiang, H., Zhou, C., Meng, F., Zhang, B., Zhou, J., Huang, D., Wu, Q., Su, J., 2021. Exploring dynamic selection of branch expansion orders for code generation. In: Zong, C., Xia, F., Li, W., Navigli, R. (Eds.), Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing. ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021, Association for Computational Linguistics, pp. 5076–5085.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Le, H., Wang, Y., Gotmare, A.D., Savarese, S., Hoi, S.C., 2022. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In: NeurIPS.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering. ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp. 795–806.

Lethbridge, T., Singer, J., Forward, A., 2003. How software engineers use documentation: The state of the practice. IEEE Softw. 20 (6), 35–39.

Li, J., Li, Y., Li, G., Jin, Z., Hao, Y., Hu, X., 2023. SkCoder: A sketch-based approach for automatic code generation. In: 45th IEEE/ACM International Conference on Software Engineering. ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, pp. 2124–2135.

Li, Z., Wu, Y., Peng, B., Chen, X., Sun, Z., Liu, Y., Yu, D., 2021. SeCNN: A semantic CNN parser for code comment generation. J. Syst. Softw. 181, 111036.

Liu, S., Chen, Y., Xie, X., Siow, J.K., Liu, Y., 2021. Retrieval-augmented generation for code summarization via hybrid GNN. In: 9th International Conference on Learning Representations. ICLR 2021, Virtual Event, Austria, May 3-7, 2021, OpenReview.net.

Mens, K., Nijssen, S., Pham, H., 2021. The good, the bad, and the ugly: Mining for patterns in student source code. In: Vescan, A., Serban, C., Henry, J., Praphamontripong, U. (Eds.), EASEAI 2021: Proceedings of the 3rd International Workshop on Education Through Advanced Software Engineering and Artificial Intelligence. Athens, Greece, 23 August 2021, ACM, pp. 1–8.

Murali, V., Qi, L., Chaudhuri, S., Jermaine, C., 2018. Neural sketch learning for conditional program generation. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net.

Nye, M.I., Hewitt, L.B., Tenenbaum, J.B., Solar-Lezama, A., 2019. Learning to infer program sketches. In: Chaudhuri, K., Salakhutdinov, R. (Eds.), Proceedings of the 36th International Conference on Machine Learning. ICML 2019, 9-15 June 2019, Long Beach, California, USA, In: Proceedings of Machine Learning Research, vol. 97, PMLR, pp. 4861–4870.

Nykaza, J., Messinger, R., Boehme, F., Norman, C.L., Mace, M., Gordon, M., 2002. What programmers really want: Results of a needs assessment for SDK documentation. In: Haramundanis, K., Priestley, M. (Eds.), Proceedings of the 20st Annual International Conference on Documentation. SIGDOC 2002, Toronto, Ontario, Canada, October 20-23, 2002, ACM, pp. 133–141.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., Nakamura, S., 2015. Learning to generate pseudo-code from source code using statistical machine translation (T). In: Cohen, M.B., Grunske, L., Whalen, M. (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering. ASE 2015, Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society, pp. 574–584.

Papineni, K., Roukos, S., Ward, T., Zhu, W., 2002. Bleu: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. July 6-12, 2002, Philadelphia, PA, USA, ACL, pp. 311–318.

Parvez, M.R., Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K., 2021. Retrieval augmented code generation and summarization. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Findings of the Association for Computational Linguistics. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021, Association for Computational Linguistics, pp. 2719–2734.

Pham, H., Nijssen, S., Mens, K., Nucci, D.D., Molderez, T., Roover, C.D., Fabry, J., Zaytsev, V., 2019. Mining patterns in source code using tree mining algorithms. In: Novak, P.K., Smuc, T., Dzeroski, S. (Eds.), Discovery Science - 22nd International Conference. DS 2019, Split, Croatia, October 28-30, 2019, Proceedings, In: Lecture Notes in Computer Science, vol. 11828, Springer, pp. 471–480.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, Association for Computational Linguistics, pp. 1139–1149.

Satter, A., Sakib, K., 2017. A similarity-based method retrieval technique to improve effectiveness in code search. In: Sartor, J.B., D'Hondt, T., Meuter, W.D. (Eds.), Companion To the First International Conference on the Art, Science and Engineering of Programming, Programming 2017. Brussels, Belgium, April 3-6, 2017, ACM, pp. 39:1–39:3.

Shen, S., Zhu, X., Dong, Y., Guo, Q., Zhen, Y., Li, G., 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In: Roychoudhury, A., Cadar, C., Kim, M. (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM, pp. 1533–1543.

Solar-Lezama, A., 2008. Program Synthesis by Sketching. University of California, Berkeley.

Solar-Lezama, A., 2009. The sketching approach to program synthesis. In: Hu, Z. (Ed.), Programming Languages and Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 4–13.

Soliman, A., Hadhoud, M., Shaheen, S., 2022. MarianCG: A code generation transformer model inspired by machine translation. J. Eng. Appl. Sci. 69.

Song, J., Kim, S., Yoon, S., 2021. AligNART: Non-autoregressive neural machine translation by jointly learning to estimate alignment and translate. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics, pp. 1–14.

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019. A grammar-based structural CNN decoder for code generation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, the Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, the Ninth AAAI Symposium on Educational Advances in Artificial Intelligence. EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, AAAI Press, pp. 7055–7062.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L., 2020. TreeGen: A tree-based transformer architecture for code generation. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, the Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, the Tenth AAAI Symposium on Educational Advances in Artificial Intelligence. EAAI 2020, New York, NY, USA, February 7-12, 2020, AAAI Press, pp. 8984–8991.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. IntelliCode compose: Code generation using transformer. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (Eds.), ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Virtual Event, USA, November 8-13, 2020, ACM, pp. 1433–1443.

Wang, Y., Le, H., Gotmare, A.D., Bui, N.D.Q., Li, J., Hoi, S.C.H., 2023. CodeT5+: Open code large language models for code understanding and generation. CoRR abs/2305.07922, arXiv:2305.07922.

Wang, Y., Wang, W., Joty, S.R., Hoi, S.C.H., 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Moens, M., Huang, X., Specia, L., Yih, S.W. (Eds.), Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021, Association for Computational Linguistics, pp. 8696–8708.

Wei, B., Li, Y., Li, G., Xia, X., Jin, Z., 2020. Retrieve and refine: Exemplar-based neural comment generation. In: 35th IEEE/ACM International Conference on Automated Software Engineering. ASE 2020, Melbourne, Australia, September 21-25, 2020, IEEE, pp. 349–360.

Xie, B., Su, J., Ge, Y., Li, X., Cui, J., Yao, J., Wang, B., 2021. Improving tree-structured decoder training for code generation via mutual learning. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, the Eleventh Symposium on Educational Advances in Artificial Intelligence. EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, pp. 14121–14128.

Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J., 2022. A systematic evaluation of large language models of code. In: Chaudhuri, S., Sutton, C. (Eds.), MAPS@PLDI 2022: 6th ACM SIGPLAN International Symposium on Machine Programming. San Diego, CA, USA, 13 June 2022, ACM, pp. 1–10.

Xu, L., Yang, H., Liu, C., Shuai, J., Yan, M., Lei, Y., Xu, Z., 2021. Two-stage attention-based model for code search with textual and structural features. In: 28th IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER 2021, Honolulu, HI, USA, March 9-12, 2021, IEEE, pp. 342–353.

Xu, E., Yu, Z., Li, N., Cui, H., Yao, L., Guo, B., 2023. Quantifying predictability of sequential recommendation via logical constraints. Front. Comput. Sci. 17 (5), 175612.

Yang, G., Liu, K., Chen, X., Zhou, Y., Yu, C., Lin, H., 2022. CCGIR: Information retrieval-based code comment generation method for smart contracts. Knowl.-Based Syst. 237, 107858.

Yang, R., Zhang, J., Gao, X., Ji, F., Chen, H., 2019. Simple and effective text matching with Richer alignment features. In: Korhonen, A., Traum, D.R., Màrquez, L. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics. ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, Association for Computational Linguistics, pp. 4699–4709.

Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., Chen, T., 2023. ExploitGen: Template-augmented exploit code generation based on CodeBERT. J. Syst. Softw. 197, 111577.

Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G., 2018. Learning to mine aligned code and natural language pairs from stack overflow. In: Zaidman, A., Kamei, Y., Hill, E. (Eds.), Proceedings of the 15th International Conference on Mining Software Repositories. MSR 2018, Gothenburg, Sweden, May 28-29, 2018, ACM, pp. 476–486.

Yin, P., Neubig, G., 2017. A syntactic neural model for general-purpose code generation. In: Barzilay, R., Kan, M. (Eds.), Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, Association for Computational Linguistics, pp. 440–450.

Yin, P., Neubig, G., 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Blanco, E., Lu, W. (Eds.), Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018, Association for Computational Linguistics, pp. 7–12.

Yin, P., Neubig, G., 2019. Reranking for neural semantic parsing. In: Korhonen, A., Traum, D.R., Màrquez, L. (Eds.), Proceedings of the 57th Conference of the Association for Computational Linguistics. ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, Association for Computational Linguistics, pp. 4553–4559.

Qianwen Gou received the bachelor's degree from Northwestern University, China. She is currently a Ph.D. student with the School of Computer Science, Northwestern Polytechnical University. Her research interests include program synthesis, program recommendation.

Yunwei Dong received the Ph.D. degree from Northwestern University, China. His research interests include embedded systems, information-physical convergence systems, trusted software design and verification, and intelligent software engineering.

Qiao Ke received her Ph.D. degree in Applied Mathematics from Xi'an Jiaotong University, Xi'an, China in 2019. Her research interests are in the areas of deep learning, statistics learning, Bayesian learning, image processing, and the internet of things.