Synthesizing Conjunctive Queries for Code Search

Chengpeng Wang  
The Hong Kong University of Science and Technology, China

Peisen Yao  
Zhejiang University, Hangzhou, China

Wensheng Tang  
The Hong Kong University of Science and Technology, China

Gang Fan  
Ant Group, Shenzhen, China

Charles Zhang  
The Hong Kong University of Science and Technology, China

Abstract

This paper presents Squid, a new conjunctive query synthesis algorithm for searching code with target patterns. Given positive and negative examples along with a natural language description, Squid analyzes the relations derived from the examples by a Datalog-based program analyzer and synthesizes a conjunctive query expressing the search intent. The synthesized query can be further used to search for desired grammatical constructs in the editor. To achieve high efficiency, we prune the huge search space by removing unnecessary relations and enumerating query candidates via refinement. We also introduce two quantitative metrics for query prioritization to select the queries from multiple candidates, yielding desired queries for code search. We have evaluated Squid on over thirty code search tasks. It is shown that Squid successfully synthesizes the conjunctive queries for all the tasks, taking only 2.56 seconds on average.

2012 ACM Subject Classification  Software and its engineering → Automatic programming; Human-centered computing → User interface programming

Keywords and phrases  Query Synthesis, Multi-modal Program Synthesis, Code Search

Digital Object Identifier  10.4230/LIPIcs.ECOOP.2023.36


Funding  The authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei.

Acknowledgements  We thank the anonymous reviewers, Xiao Xiao, and Xiaoheng Xie for their helpful comments. Peisen Yao is the corresponding author.

1 Introduction

Developers often need to search their code for target patterns in various scenarios, such as API understanding [29], code refactoring [56], and program repair [47]. According to recent studies [34, 30], existing efforts have to compromise between ease of use and capability. Most mainstream IDEs [23] only support string match or structural search of restrictive grammatical constructs although complex user interactions are not required. Besides, static program analyzers, such as Datalog-based program analyzers [44, 35, 3], provide deep program facts for users to explore advanced patterns, while users have to customize the analyzers to meet their needs [12]. For example, if users want to explore code patterns with the Datalog-based program analyzer CodeQL [3], they have to learn the query language to access the derived relational representation. However, there always exists a non-trivial gap

© Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang; licensed under Creative Commons License CC-BY 4.0.

37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 36; pp. 36:1–36:30
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
between a user’s search intent and a customized query. A large number of complex relations make query writing involve strenuous efforts, especially in formalizing search intents and debugging queries, which hinders the usability of CodeQL for code search.

Our Goal. We aim to propose a query synthesizer to unleash the power of a Datalog-based program analyzer for code search. To show the search intent, a user can specify a synthesis specification consisting of positive examples, negative examples, and a natural language description. Specifically, positive and negative examples indicate desired and non-desired grammatical constructs, respectively, while the natural language description shows the search intent by a sentence. Our synthesizer is expected to generate a query separating positive examples from negative ones, which can support code search in the editor. In this work, we focus on conjunctive queries, which have been recognized as queries of an important form to support search tasks [17].

Consider a usage scenario: Find all the methods receiving a parameter with Log4jUtils type and giving a return with CacheConfig type. The user can provide the synthesis specification shown in Figure 1(a). With the relational representation in Figure 1(b) derived from the examples, our synthesizer would synthesize the conjunctive query in Figure 1(c) to express the search intent. In particular, our synthesis specification is easy to provide. The users can often copy desired grammatical constructs from an editor as positive examples [34] and then mutate them to form negative ones. Meanwhile, they can express their need to search code with a brief sentence as the description. Thus, an effective and efficient synthesizer enables the users to express the search intent from a high-level perspective, serving as a user-friendly interface for code search.

---

1 We show five relations as examples, while a Datalog-based analyzer can derive over a hundred relations.
Challenges. Nevertheless, it is far from trivial to synthesize a conjunctive query for code search. First, a Datalog-based program analyzer can generate many relations with multiple attributes as the relational representation. For example, CodeQL exposes over a hundred relations to users for query writing [45]. The various choices of selecting relations and enforcing conditions on attributes induce a dramatically huge search space in the synthesis, which can involve both the comparisons between attributes and string constraints, posing a significant challenge to achieving high efficiency. Second, there often exist multiple query candidates that separate positive examples from negative ones, while several candidates can suffer from the over-fitting problem, failing to express the search intent with no bias [53]. An ineffective query candidate selection would mislead the synthesizer into returning wrong queries and further cause the failure of code search.

Existing Effort. There are three major lines of existing effort. The first line of the studies utilizes input-output examples to synthesize queries in various forms, such as analytic SQL queries [58, 13] and relational queries [42, 46]. Although the queries often have expressive syntax, the synthesizers only take a few relations as input, not facing hundreds of relations as ours. The second line of approaches is the component-based synthesis technique [14, 38], which leverages type signatures to enumerate well-typed programs. However, a significant number of comparable attribute pairs still induce an explosively huge search space even if we adopt the techniques by guiding the search with the schema. The third line of the studies derives program sketches from natural language descriptions via semantic parsing [28] and prioritizes feasible solutions with probability models [54, 4]. Unfortunately, the ambiguity of natural languages and the inadequacy of the training process can make a semantic parser ineffective and eventually miss optimal solutions [40]. It is also worth noting that existing techniques do not attempt to select a feasible solution that maximizes or minimizes a specific metric. Although several inductive logic learning-based techniques adopt heuristic priority functions to accelerate the synthesis [46], they do not guarantee the optimality of the synthesized queries, and thus can not resolve the query candidate selection in our problem.

Our Solution. Our key idea comes from three critical observations. First, only a few relations contribute to separating positive examples from negative ones. For example, the methods in Figure 1(a) have the same modifier, indicating that the relation Modifier is unnecessary. Second, adding an attribute comparison expression or a string constraint to the condition of a conjunctive query yields a stronger restriction on grammatical constructs. If a query misses a positive example, we cannot obtain a query candidate by strengthening the query. Third, a desired query tends to constrain grammatical constructs mentioned in the natural language description sufficiently. For the instance in Figure 1, the query extracting the methods with the return type CacheConfig is a query candidate but not a desired one, as it does not pose any restriction on parameters.

Based on the observations, we realize that it is possible to narrow down necessary relations and avoid their infeasible compositions to prune the search space, and meanwhile, select query candidates with the guidance of the natural language description. According to the insight, we present a multi-modal synthesis algorithm Squid with three stages:

- To narrow down the relations, we introduce the notion of the dummy relations to depict the relations unnecessary for the synthesis and propose the representation reduction to exclude dummy relations, which prunes the search space effectively.
- To avoid infeasible compositions of relations, we perform the bounded refinement to enumerate the queries, skipping the unnecessary search for the queries that exclude a positive example. Particularly, the string constraints are synthesized by computing the longest common substrings, which is achieved efficiently in the refinement.
To select desired queries, we establish the dual quantitative metrics, namely named entity coverage and structural complexity, and select query candidates by optimizing them as the objectives, which creates more opportunities for returning desired queries.

We implement SQUID and evaluate it on 31 code search tasks. It successfully synthesizes desired queries for each task in 2.56 seconds on average. Besides, the representation reduction and the bounded refinement are crucial to its efficiency. Skipping either of them would increase the average time cost to around 8 seconds, and several tasks cannot be finished within one minute. Meanwhile, dual quantitative metrics play critical roles in the selection. Applying only one metric would make 12 or 7 tasks fail due to the synthesized non-desired queries. We also state and prove the soundness, completeness, and optimality of SQUID. If there exist query candidates for a given synthesis specification, SQUID always returns query candidates optimizing two proposed metrics to express the search intent. To summarize, our work makes the following key contributions:

- We propose a multi-modal conjunctive query synthesis problem. An effective and efficient solution can serve as a user-friendly interface of a Datalog-based analyzer for code search.
- We design an efficient algorithm SQUID for an instance of our synthesis problem, which automates the code search tasks in real-world scenarios.
- We implement SQUID as a tool and evaluate it upon 31 code search tasks, showing that SQUID synthesizes the desired queries successfully and efficiently.

2 Overview

This section demonstrates a motivating example and briefly the key idea of our approach.

2.1 Motivating Example

Suppose a developer wants to avoid the security issue caused by log4j library [41]. He or she may examine the methods that receive a Log4jUtils object as a parameter and return a CacheConfig object. One choice is to leverage the built-in search tools of the IDEs to search the code lines containing Log4jUtils or CacheConfig, while the string matching-based search cannot filter grammatical constructs according to their kinds. Although several IDEs enable the structural search [23], their non-extensible templates only support searching for grammatical constructs of restrictive kinds. Another alternative is to write a query depicting the target pattern and evaluate it with a Datalog-based program analyzer, such as CodeQL [3]. However, it not only involves great laborious efforts in query language learning but also creates the burden of query writing and debugging.

To improve the usability and capability of code search, we aim to synthesize a query for a Datalog-based program analyzer. As shown in Figure 1(a), a user can specify the synthesis specification to indicate the search intent. Specifically, the positive and negative examples show the desired and undesired grammatical constructs, respectively, while the natural language description demonstrates the search intent in a sentence. Based on a Datalog-based program analyzer, we can convert the examples to a set of relations as the relational representation along with positive and negative tuples, which are shown in Figure 1(b). For example, the first tuple in the relation Method is the positive tuple indicating the method foo in Figure 1(a), which is a positive example. If we automatically synthesize the conjunctive query in Figure 1(c), the user does not need to delve into the relations and, instead, specifies the synthesis specification from a high-level perspective.
2.2 Synthesizing Conjunctive Queries

The query synthesizer should effectively generate the desired queries that express the search intent correctly. However, it is stunningly challenging to obtain an effective and efficient synthesizer. First, we have to tackle a great number of the relations and their attributes when we choose relevant relations and enforce correct constraints upon them, which can involve both comparisons over attributes and string constraints. Second, the non-uniqueness of query candidates creates the obstacle of selecting proper candidates. Any improper selection would return a non-desired query, leading to code search failure. To address the challenges, we propose a new multi-modal synthesis algorithm Squid. As shown in Figure 2, Squid consists of three phases, which come from the following three ideas.

Idea 1: Removing dummy relations. Although there are many relations potentially used in the synthesis, we can identify a class of relations, named dummy relations, as unnecessary ones and then discard them safely. Specifically, a relation is dummy if it cannot separate a positive tuple from a negative one. As an example, the methods in Figure 1(a) have the same modifier, which is shown by the same values of the foreign keys mdf_id of the relation Method in Figure 1(b). This indicates that the relation Modifier has no impact on excluding negative tuples and thus can be discarded to prune the search space. Based on this insight, we propose the representation reduction to remove the dummy relations, narrowing down the necessary relations for the synthesis.

Idea 2: Enumerating query candidates via refinement. According to the query syntax, the constraints, including attribute comparisons and string constraints, pose restrictions on grammatical constructs. This implies that we cannot obtain a query candidate by refining the query that excludes a positive tuple. In Figure 1, we may obtain a query that enforces both the parameter and the return value of a method have the same type. Obviously, the query excludes the method foo, which is a positive tuple, and thus, we should stop strengthening the restrictions on grammatical constructs. Based on this insight, we introduce the technique of the bounded refinement, which adds conditions for the query enumeration and discards the queries that exclude any positive tuples. Thus, we can avoid enumerating infeasible compositions of relations, which further prunes the search space effectively.

Idea 3: Dual quantitative metrics for selection. Desired queries not only separate positive tuples from negative ones but also tend to cover as many program-related named entities as possible. In Figure 1, we may obtain a query candidate that restricts the return type to be a CacheConfig object. However, it does not pose any restriction on the parameters and leaves the named entity “parameter” uncovered, showing that it does not express the intent sufficiently. Meanwhile, Occam’s razor [5] implies that desired queries should be as simple as possible. Hence, we introduce the named entity coverage and the structural complexity
as the dual quantitative metrics, and perform the candidate selection to identify desired queries by optimizing the metrics. Finally, we blend the selection with the refinement and terminate the enumeration when unexplored candidates cannot be better than the current ones, further avoiding the unnecessary enumerative search.

3 Problem Formulation

This section first presents the program relational representation (§ 3.1) and then introduces the conjunctive queries for code search (§ 3.2). Lastly, we state the multi-modal conjunctive query synthesis problem and brief the roadmap of technical sections (§ 3.3).

3.1 Program Relational Representation

First of all, we formally define the concept of the relation as the preliminary.

Definition 3.1 (Relation). A relation $R(a_1, \ldots, a_n)$ is a set of tuples $(t_1, \ldots, t_n)$, where $n$ is the arity of $R$. For each $1 \leq i \leq n$, $a_i$ is the attribute of the relation.

A relation is structured data that stores the details of different aspects of an entity. Concretely, a Datalog-based program analyzer encodes the program properties with a set of relations in a specific schema [3, 34, 37]. In what follows, we define the relational representation of a program.

Definition 3.2 (Relational Representation). Given a program, its relational representation $\mathcal{R}$ is a set of relations over the following schema $\Gamma$. Specifically, $\Gamma$ maps a relation symbol $R$ to an $n$-tuple of pairs, where each element in the tuple $\Gamma(R)$ has one of the following form:
- $(id, R)$: The attribute $id$ is the primary key of the relation $R$.
- $(a, R')$: The foreign key $a$ in the relation $R$, referencing the primary key of $R' \in \text{dom}(\Gamma)$.
- $(a, \text{STR})$: The attribute $a$ has a string value indicating the textual information.

Particularly, we say $\Gamma$ as the language schema. Without introducing the ambiguity, we use $(a, \cdot) \in \Gamma(R)$ to indicate that $(a, \cdot)$ is an element of the tuple $\Gamma(R)$.

Example 3.1. Figure 1(b) shows five relations as examples, where the values of the primary keys are italic and underlined, and the foreign keys are italic. Based on the first tuple in the relation Method, we can track the identifier, the return type, and the modifier of the method $\text{foo}$ based on the foreign keys. Similarly, we can identify the identifier and the type of each parameter based on the relation Parameter.

Essentially, the relational representation of a program encodes the program properties with relations, which depicts the relationship of grammatical constructs in the program. In reality, various relations can be derived with Datalog-based program analyzers, such as ReferenceType and VarPointsTo provided by DOOP [44], depicting the type information and points-to facts, respectively. Due to the space limit, we only show five relations in Figure 1(b).

3.2 Conjunctive Queries

To simplify the presentation, we formulate the conjunctive queries as the relational algebra expressions [1] in the rest of the paper. In what follows, we first brief several relational algebra operations and then introduce the conjunctive queries for code search.
 Definition 3.3 (Relational Algebra Operations). In a relational algebra, the selection, projection, and rename operations are defined as follows:

- \( \sigma_\Theta(R) := \{(t_1, \ldots, t_n) \in R \mid [t_i \rightarrow t_i | 1 \leq i \leq n] \models \Theta\}\) is the selection of a relation \( R \) with the selection condition \( \Theta \) over its attributes.

- \( \Pi_{a'}(R) := \{(t.a'_1, \ldots, t.a'_k) \mid t \in R\}\) is the projection of a relation \( R \) upon a tuple of attributes \( a' \), denoted by \( \Pi_{a'}(R) \). Here \( a' = (a'_1, \ldots, a'_k) \). \( t.a \) is the value of the attribute \( a \) in the tuple \( t \).

- \( R_1 \times R_2 := \{(t_1^1, \ldots, t_1^k, t_2^1, \ldots, t_2^l) \mid (t_1^1, \ldots, t_1^k) \in R_1, (t_2^1, \ldots, t_2^l) \in R_2\}\) is the Cartesian product of two relations \( R_1 \) and \( R_2 \).

- The rename of a relation \( R \), denoted by \( \rho_A(R) \), yields the same relation named \( A \).

The relational algebra operations enable us to manipulate the relational representation to search desired grammatical constructs. Specifically, we often need to search specific grammatical constructs via string match and enforce them to satisfy several constraints simultaneously. Now we formalize the conjunctive queries to express the code search intent.

 Definition 3.4 (Conjunctive Query). Given the relational representation \( \mathcal{R} \), a conjunctive query \( R_Q \) is a relational algebra expression of the form \( \Pi_{(A_i, \cdots)}(\sigma_\Theta(\rho_{A_1}(R_1) \times \cdots \times \rho_{A_m}(R_m))) \), where \( R_i \in \mathcal{R}, \Theta := \phi_1 \wedge \cdots \wedge \phi_n \), and each \( \phi_i(1 \leq i \leq n) \) occurring in the selection condition \( \Theta \) is an atomic condition in the following two forms:

- An atomic equality formula \( A_j.id = A_k.a \), where \( a \) is the foreign key and \( (a, R_j) \in \Gamma(R_k) \).

- A string constraint \( p(A_k.a, \ell) \) over the string attribute \( A_k.a \), where \( \ell \) is a string literal, and \( p \in \{\text{equal, suffix, prefix, contain}\} \).

In particular, \( \Pi_{(A_i, \cdots)} \) indicates the projection upon all the attributes of the relation \( A_i \).

The form of the conjunctive queries in Definition 3.4 depicts the user intent from two aspects. First, the atomic equality formulas encode the relationship between the grammatical constructs. Second, the four string predicates support the common scenarios of string matching-based code search. We do not focus on synthesizing more expressive string constraints, which is the orthogonal direction of program synthesis [10, 27, 36]. Based on the conjunctive queries, we can simultaneously perform the string matching-based search and filter the constructs with various relations in the program relational representation.

Example 3.2. We can formalize the query in Figure 1(c) as the relational algebra expression:

\[
\Pi_{(A_1, \cdots)}(\sigma_\Theta(\rho_{A_1}(\text{Method}) \times \rho_{A_2}(\text{Type}) \times \rho_{A_3}(\text{Parameter}) \times \rho_{A_4}(\text{Type})))
\]

where the selection condition \( \Theta \) in the selection operation is as follows:

\[
\Theta := (A_1.id = A_3.method_id) \wedge (A_1.ret_type_id = A_2.id) \wedge (A_3.type_id = A_4.id) \wedge \text{equal}(A_2.name, \text{“CacheConfig”}) \wedge \text{equal}(A_4.name, \text{“Log4jUtils”})
\]

The conjunctive queries can be instantiated with various flavors [17, 1, 7]. The select-from-where queries are the instantiations of the conjunctive queries in SQL. Besides, a simple Datalog program can also express the conjunctive query with a single Datalog rule. In our paper, we formulate a conjunctive query as a relational algebra expression. Our implementation actually synthesizes the conjunctive queries as Datalog programs, which are evaluated by a Datalog solver over the program relational representation for code search.
3.3 Multi-modal Conjunctive Query Synthesis Problem

To generate the conjunctive query for code search, the users need to specify their intent as the specification. In our work, we follow the premise of the recent studies on the multi-modal program synthesis [10, 4] that multiple modalities of information can go arm in arm with each other, serving as the informative specification for the synthesis. Specifically, the users provide a natural language sentence to describe the target code pattern and provide several grammatical constructs as positive and negative examples. Notably, the multi-modal synthesis specification is easy to provide. When the users want to explore a specific code pattern, they can describe the pattern briefly in a natural language and provide several examples from their editors instead of delving into the details of the underlying relations and their attributes, enabling users to generate a query for code search in a declarative manner.

Based on the multi-modal synthesis specification, the positive and negative examples are converted to the tuples in a specific relation. For example, Figure 1 shows a set of relations as the relational representation of the examples. To formalize our problem better, we define the notion of the relation partition as follows.

**Definition 3.5 (Relation Partition).** The relation partition of $R^* \in \mathcal{R}$ is a pair of two relations $(R^*_p, R^*_n)$ satisfying $R^* = R^*_p \cup R^*_n$ and $R^*_p \cap R^*_n = \emptyset$. The relation partition is non-trivial if and only if $R^*_p \neq \emptyset$ and $R^*_n \neq \emptyset$. We say the tuples in $R^*_p$ and $R^*_n$ are positive and negative tuples, respectively.

**Example 3.3.** As shown in Figure 1, we can construct the relation partition $(R^*_p, R^*_n)$, where $R^*_p = \{(M1, I1, T3, MDF1)\}$ and $R^*_n = \{(M2, I2, T3, MDF1), (M3, I3, T2, MDF1)\}$. Obviously, $R^*_p$ and $R^*_n$ are disjoint, and $R^*_p \cup R^*_n$ is exactly the relation Method.

The positive and negative tuples essentially depict the positive and negative examples, respectively. Based on the program’s relational representation, the examples specified by the users can determine the positive and negative tuples, which can be achieved in various manners. Specifically, users can select a grammatical construct in the IDEs or use a code sample in a specific coding standard [51] as a positive example and remove several sub-patterns from a positive example by mutation to construct negative ones. Such positive and negative examples further constitute a sample code snippet, from which a Datalog-based analyzer derives a set of relations as the relational representation. In this paper, we omit the details of positive/negative tuple generation and formulate a multi-modal conjunctive query synthesis (MMCSQS) problem as follows.

Given a relational representation $\mathcal{R}$, a relation partition $(R^*_p, R^*_n)$ of $R^* \in \mathcal{R}$, and a natural language description $s$, we aim to synthesize a conjunctive query $R_Q$ containing the positive tuples in $R^*_p$ and excluding the negative tuples in $R^*_n$.

**Example 3.4.** Figure 1(a) shows the multi-modal synthesis specification, which consists of a positive example, two negative examples, and a natural language description $s$ as “Find all the methods receiving a Log4jUtils-type parameter and giving a CacheConfig-type return”. Leveraging a Datalog-based analyzer, we obtain the relational representation and the relation partition of Method, which are shown in Figure 1(b). To automate the code search, we expect to synthesize the query in Fig 1(c) or Example 3.2 according to the relational representation, the relation partition, and the natural language sentence.

To promote the code search, we propose an efficient synthesis algorithm SQUID for the MMCSQS problem, which is our main technical contribution. As explained in § 1, it is challenging to solve the MMCSQS problem efficiently, which involves mitigating the huge
search space and selecting queries from multiple candidates. In the following two sections, we formalize the conjunctive query synthesis from a graph perspective (§ 4), and illustrate the technical details of our synthesis algorithm Squid (§ 5), which prunes the search space and selects desired queries effectively.

4 Conjointive Query Synthesis: A Graph Perspective

This section presents a graph perspective of our conjunctive query synthesis problem. Specifically, we introduce two graph representations of the language schema and the conjunctive queries, named the schema graph (§ 4.1) and the query graph (§ 4.2), respectively, which reduces the conjunctive query synthesis to the query graph enumeration. Lastly, we summarize the section and highlight the technical challenges from a graph perspective (§ 4.3).

4.1 Schema Graph

According to Definition 3.2, a relation in the language schema has three kinds of attributes, namely a unique primary key, foreign keys, and string attributes. Obviously, the selection condition $\Theta$ in $R_Q$ should only compare the foreign key of a relation with its referenced primary key or constrain the string attributes with string predicates. To depict the possible ways of constraining the attributes, we define the concept of the schema graph as follows.

- **Definition 4.1 (Schema Graph).** The schema graph $G_\Gamma$ of a language schema $\Gamma$ is $(N_\Gamma, E_\Gamma)$:
  - The set $N_\Gamma$ contains the relation symbols in the schema or the string type STR as the nodes of the schema graph, i.e., $N_\Gamma := \text{dom}(\Gamma) \cup \{\text{STR}\}$.
  - The set $E_\Gamma$ contains an edge $(n_1, n_2, a)$ if and only if either of the conditions holds:
    - $n_1, n_2 \in \text{dom}(\Gamma)$ and $(a, n_2) \in \Gamma(n_1)$: The relation $n_1$ has a foreign key named $a$ referencing the primary key of the relation $n_2$.
    - $n_1 \in \text{dom}(\Gamma)$ and $(a, \text{STR}) \in \Gamma(n_1)$: $a$ is the string attribute of the relation $n_1$.

- **Example 4.1.** Consider the relations in the relational representation shown in Figure 1(b). We can construct the schema graph in Figure 3(a). The edge from Method to Modifier labeled with mdf_id shows that the attribute mdf_id of Method is a foreign key referencing the primary key of Modifier. Similarly, the edge from Type to STR labeled with name shows that the attribute name in Type is a string attribute.

  Noting that there can exist multiple edges with different labels between two nodes in the schema graph, which indicate that a relation take multiple foreign keys referencing the same relation or string attributes as the attributes. Essentially, the schema graph encodes the available relations with its node set and depicts the valid forms of the atomic formulas appearing in the selection condition with its edge set. Although we can compare the attributes of any relations flexibly, a solution to our problem must take the valid form of the atomic formulas as its selection condition, comparing the foreign keys with the referenced primary keys or examining the string attributes of the relations appearing in a Cartesian product.

4.2 Query Graph

As formulated in Definition 3.4, there are two key components in the conjunctive query, namely the Cartesian product and the selection condition. Leveraging the schema graph, we can represent the components with nodes and edges on the graph, which uniquely determines a conjunctive query. Formally, we introduce the notion of the query graph as follows.
4.3 Summary

Leveraging the schema graph, we have reduced the conjunctive query synthesis process to query graph enumeration. To obtain the desired queries, we only need to select the nodes and edges from the schema graph and create the node \((p, \ell)\) with proper string predicates and literals for constructing a query graph \(G_Q\), which should satisfy that the induced query \(\kappa^{-1}(G_Q)\) separates positive tuples from negative ones.

Obtaining the desired queries with high efficiency is a non-trivial problem. The schema graph can be overwhelming, containing over a hundred nodes and edges, which leads to an enormous number of choices for relations occurring in the query graph. Even for a given set of
relations, the flexibility of instantiating equality constraints and string constraints over their attributes can induce a large number of selection choices of edges in a query graph, which exacerbates the search space explosion problem. Additionally, the existence of multiple query candidates necessitates the effective selection of candidates, which is crucial for conducting a code search task. In the next section, we will detail our synthesis algorithm that addresses these challenges, resulting in high efficiency and effectiveness for code search.

5 Synthesis Algorithm

This section presents our synthesis algorithm Squid to solve the MMCQS problem. Squid takes as input the relational representation \( R \) of an example program, a relation partition \( (R^*_p, R^*_n) \) of \( R \), and a natural language description \( s \). It generates the query candidates separating positive tuples \( t_p \in R^*_p \) from negative ones \( t_n \in R^*_n \), which can be further selected and then used for code search. To address the challenges in § 4.3, Squid works with the following three stages:

- To tackle a large number of relations, Squid relies on the notion of dummy relations and conducts the representation reduction based on the positive and negative tuples, which effectively narrow down the relations that can appear in the query graph (§ 5.1).
- To avoid unnecessary enumeration of edges in query graphs, we propose the bounded refinement by enumerating the query graphs based on the schema graph, which essentially appends equality constraints and string constraints inductively (§ 5.2).
- To select query candidates, Squid identifies the named entities in the natural language description \( s \) for the prioritization, and blends the selection with the refinement to collect the desired queries (§ 5.3).

We also formulate the soundness, completeness, and optimality of our algorithm (§ 5.4). For better illustration, we use the synthesis instance shown in Figure 1 throughout this section.

5.1 Representation Reduction

To tackle the large search space, we first propose the representation reduction to narrow down the relations possibly used in the query. Specifically, we introduce the notion of dummy relations to determine the characteristics of unnecessary relations (§ 5.1.1) and propose the algorithm of removing dummy relations for the representation reduction (§ 5.1.2).

5.1.1 Dummy Relations

There exists a class of relations, named dummy relations, which cannot involve in distinguishing positive and negative tuples, such as the relation Modifier in Figure 1. Before defining them, we first introduce the undirected relation path and the activated relation.

Definition 5.1 (Undirected Relation Path). An undirected relation path from \( R_0 \) to \( R_{k+1} \) in the schema graph \( G_T = (N_T, E_T) \) is \( p : R_0 \leftrightarrow (a_0, d_0) \cdots \leftrightarrow (a_i, d_i) R_{k+1} \), where \( R_i \in \text{dom}(\Gamma) \). Here, \( d_i = 1 \) if and only if \( (R_i, R_{i+1}, a_i) \in E_T \), and \( d_i = -1 \) if and only if \( (R_{i+1}, R_i, a_i) \in E_T \).

Definition 5.2 (Activated Relation). Given a tuple \( t_0 \in R_0 \) and an undirected relation path \( p : R_0 \leftrightarrow (a_0, d_0) \cdots \leftrightarrow (a_i, d_i) R_{k+1} \), the activated relation of \( t_0 \) along \( p \) is \( I(t_0, p) = \{ t_{k+1} | t_{i+1} \in R_{i+1}, \text{predicate}(d_i = 1, t_i, a_i = t_{i+1}.id, t_i.id = t_{i+1}.id), 0 \leq i \leq k \} \).

Example 5.1. In Figure 1, the path \( p_1 : \text{Method} \leftrightarrow \text{(method_id, -1)} \) \text{Parameter} \leftrightarrow \text{(type_id, 1)} \) Type, and \( t_p = \{ \text{M1}, \text{I1}, \text{T3}, \text{MDF1} \} \in R^*_p \). We have \( t_1 = \{ \text{P1}, \text{I4}, \text{T1}, \text{M1} \} \) in Parameter, and \( t_2 = \{ \text{T1}, \text{Log4jUtils} \} \in \text{Type} \). By inspecting other tuples, we have \( I(t_p, p_1) = \{ (\text{T1}, \text{Log4jUtils}) \} \).
Intuitively, an undirected relation path can depict the restriction upon the relations in the Cartesian product of the query. The activated relation actually contains the tuples enforcing the primary key of \( t_0 \) to appear in a selected tuple. Therefore, it is possible to narrow down the relations for the synthesis by inspecting the activated relations of positive and negative tuples along each undirected relation path. According to the intuition, we formally introduce and define the notion of the dummy relations as follows.

**Definition 5.3 (Dummy Relation).** Given a relation partition \((R_p^*, R_n^*)\) of \( R^* \in \mathcal{R} \), a relation \( R \in \mathcal{R} \) is dummy if for every undirected relation path \( p \) from \( R^* \) to \( R \), either of the conditions is satisfied: (1) There exists \( t_p \in R_p^* \) such that \( I(t_p, p) = \emptyset \); (2) \( I(t_p, p) = I(t_n, p) \) for any \( t_p \in R_p^* \) and \( t_n \in R_n^* \).

Definition 5.3 formalizes two characteristics of relations unnecessary the synthesis. First, the empty activated relation of a positive tuple \( t_p \) indicates the absence of the tuples in the relation \( R \), making the tuple \( t_p \) appear. Second, the relation \( R \) cannot contribute to separating positive tuples from negative ones if several tuples in \( R \) make all the positive and negative tuples appear simultaneously. Thus, such relations can be discarded safely.

**Example 5.2.** Example 5.1 shows that \( I(t_p, p_1) \neq \emptyset \). Similarly, \( I(t_n, p_1) = \{(T2, \text{int})\} \neq I(t_p, p_1) \) when \( t_p = (M2, 12, T4, \text{MDF1}) \in R_p^* \). Hence, Type is not dummy. Besides, any undirected relation path \( p_2 \) from Method to Modifier has the form

\[
\text{Method} \hookrightarrow (\text{mdf_id}, +1) \text{ Modifier} \hookrightarrow (\text{mdf_id}, -1) \text{ Modifier}
\]

where \([\cdot]^{*}\) indicates the repeated subpath. We find that \( I(t_p, p_2) = I(t_n, p_2) = \{\text{MDF1, public}\} \) for any \( t_p \in R_p^* \) and \( t_n \in R_n^* \). Thus, the relation Modifier is a dummy relation.

### 5.1.2 Removing Dummy Relations

Based on Definition 5.3, identifying dummy relations involves two technical parts. First, we should collect all the undirected relation paths from \( R^* \) to each relation. Second, we need to compute \( I(t_p, p) \) and \( I(t_n, p) \) for each undirected relation path \( p \) and positive/negative tuple. However, the schema graph can contain a large and even infinite number of undirected relation paths from \( R^* \). Any cycle induces the infinity of the path number, making it tricky to examine the conditions in Definition 5.3. Fortunately, we realize that the number of cycles in a path does not affect the activated relation, which is stated in the following property.

**Property 5.1.** Given any \( t_0 \in R_0 \), we have \( I(t_0, p) = I(t_0, p') \) for \( p \) and \( p' \) as follows:

\[
p : R_0 \hookrightarrow (a_0, d_0) \cdots R_t \hookrightarrow (a_t, d_t) \cdots R_l \hookrightarrow (a_{l+1}, d_{l+1}) \ R_l^{+} \hookrightarrow (a_l, d_l) \ \cdots \ R_k \hookrightarrow (a_k, d_k) \ R_{k+1}
\]

\[
p' : R_0 \hookrightarrow (a_0, d_0) \cdots R_l \hookrightarrow (a_l, d_l) \cdots R_l \hookrightarrow (a_{l+1}, d_{l+1}) \ R_l \hookrightarrow (a_l, d_l) \ \cdots \ R_k \hookrightarrow (a_{k}, d_{k}) \ R_{k+1}
\]

Here, \([\cdot]^{+}\) indicates the cycle occurring at least one time.

Property 5.1 holds trivially according to Definition 5.2. For each undirected relation path \( p \) containing a cycle, the constraints over the tuples in \( I(t_0, p) \) are the same as the ones over the tuples in \( I(t_0, p') \). Thus, we can just examine the finite number of undirected relation paths in which a cycle appears at most one time.
Algorithm 1 Removing dummy relations for representation reduction.

1 Procedure reduce\((\Gamma, \mathcal{R}, R_p^*, R_n^*)\):
2 \(\mathcal{R}' \leftarrow \emptyset\);
3 \(G_\Gamma \leftarrow \text{SchemaGraph}(\Gamma)\);
4 foreach \(R \in \mathcal{R}\) :
5 \(\mathcal{P} \leftarrow \text{augmentPathWithCycle}(\text{AcyclicPath}(R^*, R, G_\Gamma))\);
6 foreach \(p \in \mathcal{P}, t_p \in R_p^*, t_n \in R_n^*\) :
7 if \(\mathcal{I}(t_p, p) \neq \mathcal{I}(t_n, p)\) :
8 \(\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\};\) break ;
9 foreach \(p \in \mathcal{P}, t_p \in R_p^*\) :
10 if \(\mathcal{I}(t_p, p) = \emptyset\) :
11 \(\mathcal{R}' \leftarrow \mathcal{R}' \setminus \{R\};\) break ;
12 return \(\mathcal{R}'\);

Establish upon the above concepts and property, Algorithm 1 shows the details of the representation reduction by removing dummy relations. Initially, we construct the schema graph \(G_\Gamma\) according to Definition 4.1 (line 3). Then we compute the undirected relation paths from \(R^*\) to \(R\) in \(G_\Gamma\), where any cycle repeats at most once (lines 4–5). Specifically, the function \(\text{AcyclicPath}\) collects all the acyclic undirected relation paths from \(R^*\) to \(R\) in the schema graph \(G_\Gamma\), while the function \(\text{augmentPathWithCycle}\) augments each acyclic path by appending each cycle at most one time. For each undirected relation path \(p\), we compute \(\mathcal{I}(t_p, p)\) and \(\mathcal{I}(t_n, p)\) according to Definition 5.2 for each \(t_p \in R_p^*\) and \(t_n \in R_n^*\), respectively. Therefore, we can identify \(R\) as a non-dummy relation if both the conditions in Definition 5.3 are violated (lines 6–11). Finally, we obtain the reduced relational representation \(\mathcal{R}'\) that excludes all the dummy relations (line 12).

Example 5.3. Consider the undirected relation paths from Method to Modifier in Figure 3(a). Algorithm 1 collects the acyclic path \(p_3: \text{Method} \leftrightarrow_{\{\text{mdf_id} + 1\}} \text{Modifier}\) and augments it to form the path \(p_4: \text{Method} \leftrightarrow_{\{\text{mdf_id} + 1\}} \text{Modifier} \leftrightarrow_{\{\text{mdf_id} - 1\}} \text{Method} \leftrightarrow_{\{\text{mdf_id} + 1\}} \text{Modifier}\). Based on the activated relations \(\mathcal{I}(t, p_3)\) and \(\mathcal{I}(t, p_4)\) for each tuple \(t\) in Method, we can find that \(\mathcal{I}(t_p, p) = \mathcal{I}(t_n, p)\) for every \(t_p \in R_p^*\) and \(t_n \in R_n^*\). Thus, Modifier is a dummy relation.

Essentially, our representation reduction analyzes the example program upon its relational representation. The activated relations provide sufficient clues to identifying unnecessary relations, i.e., the dummy ones. As the first step of the synthesis, the representation reduction narrows down the relations used in the conjunctive query. Furthermore, in the enumeration of the edges of a query graph, i.e., the sets \(E_Q\) and \(\Phi_Q\), we only need to focus on the attributes in the non-dummy relations in the reduced relational representation, which prunes the search space significantly. Lastly, we formulate the soundness of the representation reduction as follows, which can further ensure the completeness of our synthesis algorithm in §5.4. We provide a detailed proof in [49].

Theorem 5.1. (Soundness of Representation Reduction) If an instance of the MMCQS problem has a solution, there must be a conjunctive query \(R_Q\), of which the Cartesian product only consists of non-dummy relations, such that \(R_Q\) is also a solution.

5.2 Bounded Refinement

Based on the reduced relational representation \(\mathcal{R}'\), we can enumerate query candidates by selecting proper nodes corresponding to non-dummy relations in \(\mathcal{R}'\) and edges connecting such nodes. However, the search space is potentially unbounded. The relations can occur in
Synthesizing Conjunctive Queries for Code Search

a query multiple times, i.e., a node in the schema graph can be selected more than one time. Meanwhile, the literal in a string constraint can be instantiated flexibly, which increases the difficulty of enumerating a query graph with proper instantiation of $\Phi_Q$. To achieve high efficiency, we propose the bounded refinement to expand query graphs on demand and strengthen the query with the strongest string constraints. Specifically, we first introduce the notions of the bounded query and the refinable query (§ 5.2.1), and then present the details of enumerating the query graphs (§ 5.2.2).

5.2.1 Bounded Query and Refinable Query

As shown in Example 3.2, a relation can appear multiple times in the Cartesian product of a conjunctive query, inducing an unbounded search space in the synthesis. The unboundedness of the search space poses the great challenge of enumerating the query candidates efficiently. However, we realize that the conjunctive query for a code search task often involves only a few relations, each of which appears quite a few times. Thus, it is feasible to bound the maximal multiplicity of the relation in the query and conduct the bounded enumeration. Formally, we introduce the notion of the $(m, k)$-bounded query as follows.

▶ Definition 5.4 ((m, k)-Bounded Query). An $(m, k)$-bounded query is a conjunctive query with $m$ relations such that (1) each relation appears at most $k$ times; (2) there is a relation appearing exactly $k$ times.

▶ Example 5.4. The conjunctive query $\Pi(A_1, \ast)(\sigma_{true}(\rho_{A_1}(Method)))$ is a $(1, 1)$-bounded query. Similarly, the conjunctive query in Example 3.2 is a $(4, 2)$-bounded query.

Intuitively, we can enumerate the $(m, k)$-bounded queries by selecting non-dummy relations at most $k$ times, forming a query graph with $m$ nodes. When constructing the sets $E_Q$ and $\Phi_Q$ for the query graph enumeration, we only need to concentration on the attributes of the selected relations. However, not all the query graphs are worth enumerating. If $R_Q$ excludes a positive tuple, there is no need to add more nodes and edges to its query graph, as it would induce a stronger selection condition, making the new query still exclude the positive tuple. Formally, we formulate the notion of the refinable query as follows.

▶ Definition 5.5 (Refinable Query). A conjunctive query $R_Q$ is a refinable query if and only if for any $t_p \in R_p^*$ we have $t_p \in R_Q$, i.e., $R_p^* \subseteq R_Q$.

▶ Example 5.5. Consider $R_Q := \Pi(A_1, \ast)(\sigma_{\Theta}(\rho_{A_1}(Method) \times \rho_{A_2}(Type) \times \rho_{A_3}(Parameter)))$, where the selection condition $\Theta$ in the selection operation is as follows:

$$\Theta := (A_1.id = A_3.method_id) \land (A_1.ret\_type\_id = A_2.id) \land \text{equal}(A_2.name, "CacheConfig")$$

It is a refinable query as $R_p^* \subseteq R_Q = \{(M1, I1, T3, MDF1), (M2, I2, T3, MDF1)\}$.

Essentially, a refinable query is the over-approximation of the positive tuples. When it excludes all the negative tuples, the query is exactly a query candidate. Thus, we can collect the query candidates by refining the refinable queries in the bounded enumeration.

5.2.2 Enumerating Query Candidates via Refinement

We denote the sets of $(m, k)$-bounded refinable queries and query candidates by $S_R(m, k)$ and $S_C(m, k)$, respectively, and set a multiplicity bound $K$ to bound the multiplicity of a relation. To conduct a bounded enumeration, we have to compute the set $S_C(m, k)$ for $k \leq K$, 

...
Technically, we achieve the enumerative search via the bounded refinement. Assuming that 
which can be achieved by examining whether the queries in \( S_R \) are subsumed by the ones of a refinable query with fewer relations, which permit 
there exists a query graph 

Property 5.2. For every refinable query \( R_Q \in S_C(m, k) \) and \( \kappa(R_Q) := (N_Q, E_Q, \Phi_Q) \), there exists a query graph \( G_Q^1 \) or \( G_Q^2 \) such that

- \( N_Q = N_Q^1 \cup \{(R, A_i)\}, E_Q^1 \subseteq E_Q, \) and \( \Phi_Q^1 \subseteq \Phi_Q, \) where \( R \) appears in \( G_Q^1 \) exactly \((k - 1)\) times. Here, \( G_Q^1 = (N_Q^1, E_Q^1, \Phi_Q^1) \) and \( \kappa^{-1}(G_Q^1) \in S_R(m - 1, k - 1) \).

- \( N_Q = N_Q^2 \cup \{(R, A_i)\}, E_Q^2 \subseteq E_Q, \) and \( \Phi_Q^2 \subseteq \Phi_Q, \) where \( R \) appears in \( G_Q^2 \) fewer than \( k \) times. Here, \( G_Q^2 = (N_Q^2, E_Q^2, \Phi_Q^2) \) and \( \kappa^{-1}(G_Q^2) \in S_R(m - 1, k) \).

Property 5.2 shows that the sets of the nodes and edges in a query graph of a refinable query are subsumed by the ones of a refinable query with fewer relations, which permit us to enumerate the query candidates by computing \( S_R(m, k) \) and \( S_C(m, k) \) inductively. Technically, we achieve the enumerative search via the bounded refinement. Assuming that we have \( S_R(m', k') \) and \( S_C(m', k') \) for all \( m' < m \) and \( k' \leq k \). Algorithm 2 computes the sets \( S_R(m, k) \) and \( S_C(m, k) \). The technical details of the refinement are as follows:

- For the base case, where \( m = 1 \) and \( k = 1 \), we construct an empty query graph (line 3).
- For a general case, we merge the sets of the query graphs induced by the refinable queries in \( S_R(m - 1, k) \) and \( S_R(m - 1, k - 1) \) (lines 4–7). Hence, we obtain a set of query graphs \( W \) to maintain all the refinable queries with \( m \) relations.
For each query graph $G_Q \in W$, we leverage the function `expand` wraps a specific relation $R$ as a new node and add new edges, producing a set of query graphs containing the relation $R$ (lines 12–16). Such query graphs are maintained in the set $V$.

For each query graph $G_Q \in V$ that induces a refinable query, we add the induced refinable query to the set $S_R(m, k)$ (line 19) and attempt to synthesize a string constraint (lines 20–23). Specifically, the function `synLCS` examines the values of a string attribute $a$ in the positive tuples to compute the longest common substring $\ell$ and the strongest string predicate $p$ (line 21), where all the values of $a$ in the positive tuples satisfy the string constraints induced by $p$ and $\ell$. By updating $\Phi_Q$, we add a new edge to $G_Q$ and produce a new query graph inducing a refinable query (lines 22–23).

Finally, we check whether a $(m, k)$-bounded refinable query is a query candidate or not, which yields the set $S_C(m, k)$ (line 25). The sets $S_R(m, k)$ and $S_C(m, k)$ are exactly the sets of $(m, k)$-bounded refinable queries and query candidates.

Notably, we construct the string constraints on demands to strengthen the selection condition of refinable queries, which is achieved by the function `synLCS` at line 21. The reduction from string constraint synthesis to the LCS computation enables us to leverage existing algorithms, such as general suffix automaton [33], to compute the string literal $\ell$ and select a string predicate $p$ efficiently, which promotes the efficiency of query refinement.

**Example 5.6.** Figure 4 shows the part of the refinement for the instance in Example 3.4. $G_Q^1$ only contains the relation `Method`. After adding the nodes and edges, we construct the query graphs of refinable queries with more relations and atomic constraints, such as $G_Q^2$, $G_Q^3$, and $G_Q^4$. Particularly, we identify the query candidate $\kappa^{-1}(G_Q^1)$, i.e., $R_Q$ in Example 3.2.

### 5.3 Candidate Selection

Based on the bounded refinement, we collect the query candidates in which each relation appears no more than $K$ times, where $K$ is the multiplicity bound. However, not all the query candidates are the desired ones. In this section, we introduce dual quantitative metrics to prioritize queries (§ 5.3.1) and select query candidates during the refinement (§ 5.3.2).

#### 5.3.1 Dual Quantitative Metrics

Desired queries are expected to express the search intent correctly, covering as many grammatical concepts in the natural language as possible. Also, they should be as simple as possible according to Occam’s razor. Based on the intuitions, we formalize the following two metrics and then define the total order to prioritize the query candidates.
Definition 5.6. (Named-Entity Coverage) Given a function $h$ mapping an attribute of a relation to a set of words in natural language, the named entity coverage of a conjunctive query $R_Q$ with respect to a natural language description $s$ is
\[
\alpha_h(R_Q, s) = \frac{1}{|N(s)|} \cdot \left| \bigcup_{(R_i, a_j) \in \mathcal{A}(\Theta)} h(R_i, a_j) \cap N(s) \right|
\]
Here, $\mathcal{A}(\Theta)$ contains the relations and their attributes appearing in the selection condition $\Theta$ while $N(s)$ is the set of the named entities in the description $s$.

Definition 5.7. (Structural Complexity) Let $\delta(\Theta)$ be the number of the atomic formulas in the selection condition $\Theta$. The structural complexity of a conjunctive query $R_Q$ is
\[
\beta(Q) = m + \delta(\Theta)
\]
To compute the named entity coverage, we instantiate the function $h$ manually and obtain the set $N(s)$ from the natural language description $s$ based on the named entity recognition techniques [31]. The computation does not introduce much overhead, as the natural language description exhibits a fairly small length in our scenarios. Meanwhile, measuring the structural complexity is quite straightforward. The quantities $m$ and $\delta(\Theta)$ can be totally determined by the sizes of the sets $N_Q$, $E_Q$ and $\Phi_Q$ in a query graph. Thus, computing the two quantities does not introduce much overhead during the enumeration.

Example 5.7. Assume that we instantiate the function $h$ as follows:
\[
\begin{align*}
h(\text{Parameter}, \text{id}) &= \{\text{parameter}\}, \\
h(\text{Method}, \text{id}) &= \{\text{method}\}, \\
h(\text{Method}, \text{idf}_\text{id}) &= \{\text{identifier}\} \\
h(\text{Method}, \text{ret_type_id}) &= \{\text{return}, \text{type}\}, \\
h(\text{Method}, \text{mdf}_\text{id}) &= \{\text{modifier}\}
\end{align*}
\]
Consider the natural language description $s$ in Example 3.4 and the conjunctive query $R_Q$ in Example 3.2, of which the query graph is shown in Figure 3(b). We can obtain a set of named entities $N(s) = \{\text{method}, \text{type}, \text{parameter}, \text{return}\}$. Therefore, we have $\alpha_h(R_Q, s) = 1$. According to $m = 4$ and $\delta(\Theta) = 5$, its structural complexity is $\beta(R_Q) = 4 + \delta(\Theta) = 9$.

Intuitively, the selection condition of a query is more likely to conform to the user intent if the query has a higher named entity coverage. Besides, the simpler form query can have better generalization power among the query candidates covering the same number of named entities. Based on Occam’s razor, we should choose the simplest query from the candidates that maximizes the named entity coverage. Thus, we propose the total order of conjunctive queries as follows.

Definition 5.8 (Total Order). Given the function $h$ in Definition 5.6, we have $R_Q^2 \preceq_s R_Q^1$ if and only if they satisfy one of the following conditions:
\[
\begin{align*}
\alpha_h(R_Q^1, s) &\geq \alpha_h(R_Q^2, s) \quad \text{or} \\
\alpha_h(R_Q^1, s) &= \alpha_h(R_Q^2, s), \quad \beta(R_Q^1) \leq \beta(R_Q^2)
\end{align*}
\]

Example 5.8. Consider the following query candidates for the instance in Example 3.4.
\[
R_Q^1 := \Pi_{(A_1, \cdot)}(\sigma_{A_1.\text{name} = \text{“foo”}}(\rho_{A_1}(\text{Method})))
\]
\[
R_Q^2 := \Pi_{(A_1, \cdot)}(\sigma_{\Theta_2}(\rho_{A_1}(\text{Method}) \times \rho_{A_2}(\text{Type}) \times \rho_{A_3}(\text{Parameter}) \times \rho_{A_4}(\text{Type})))
\]
Here, $\Theta_2 := \Theta \land (A_1.\text{name} = \text{“foo”})$ and $\Theta$ is shown in Example 3.2. Given the function $h$ shown in Example 5.7, we can obtain that $\alpha_h(R_Q^2, s) = 1$, $\alpha_h(R_Q^3, s) = 1$, $\beta(R_Q^2) = 2$, and $\beta(R_Q^3) = 10$. According to Example 5.7, we have $R_Q^3 \preceq_s R_Q^2 \preceq_s R_Q$. 

To synthesize conjunctive queries for code search, we avoid the unnecessary enumerative search effectively in real-world code search. Specifically, we explore the bounded search space containing the query graphs of refinable relations. Each round, we fuse the refinement and selection as follows:

- The total order is an adaption of Occam’s razor for our synthesis problem. Without the named entity coverage, we would select the query candidates with the lowest structural complexity, such as $R^3_1$ in Example 5.8, even if they do not constrain the relationship of several grammatical constructs as expected. Based on the total order, we can select the query candidates by solving the dual-objective optimization problem, which finally yields the desired queries for code search.

### 5.3.2 Blending Selection with Refinement

Based on Definition 5.8, we propose Algorithm 3 that blends the candidate selection with the bounded refinement, which is more likely to obtain the desired queries for a code search task. First, we obtain the schema graph and remove the dummy relations via the representation reduction (line 2). We then compute the upper bound of the named entity coverage, which is denoted by $\tilde{\alpha}$ (line 3). After the initialization of $\alpha_{\text{max}}, \beta_{\text{min}},$ and $S_Q$ (line 4), we conduct the bounded refinement and select the query candidates in each round (lines 5–14). Obviously, there are at most $K \cdot |R'|$ relations in a query for a given multiplicity bound $K$ (line 5), and a relation can only appear at most $\min(K, m)$ times in a query with $m$ relations (line 8). In each round, we fuse the refinement and selection as follows:

- Enumerate $(m, k)$-bounded refinable queries and query candidates with Algorithm 2, strengthening the selection conditions of the refinable queries in previous rounds (line 9).
- Compute $\alpha_Q(R_Q, s)$ and $\beta(R_Q)$ for each $(m, k)$-bounded query candidate $R_Q$ and update the selected candidate set $S_Q, \alpha_{\text{max}},$ and $\beta_{\text{min}}$ (lines 10–11). Particularly, $\alpha_{\text{max}}$ and $\beta_{\text{min}}$ are updated to identify the largest candidates with respect to the total order.
- Terminate the iteration in advance and return the set $S_Q$ if $\alpha_{\text{max}}$ reaches the upper bound of the named entity coverage, i.e., $\tilde{\alpha},$ and the queries to be refined in the next round do not have lower structural complexities than $\beta_{\text{min}}$ (lines 13–14).

The refinement strengthens the selection conditions to exclude all the negative tuples. Specifically, we explore the bounded search space containing the query graphs of refinable queries, avoiding the unnecessary enumerative search effectively. In real-world code search...
tasks, the selection condition is often involved different kinds of grammatical constructs, making each relation appear often appear in the conjunction query one or two times. Therefore, we set the multiplicity bound $K$ to 2 for real-world code search tasks in practice, of which the effectiveness will be evidenced by our evaluation.

The natural language description benefits our synthesis process from two aspects. First, the selected queries in $S_Q$ are the largest queries under the total order, and thus they are more likely to conform to the user’s search intent than other query candidates. Second, we terminate the enumerative search if the named entity coverage cannot increase with a smaller structural complexity, avoiding unnecessary enumerative search of bounded query candidates for the efficiency improvement.

Example 5.9. Consider $R_{c1}Q$, $R_{c2}Q$ and $R_{h}Q$ in Example 5.8. We obtain the query candidate $R_{c1}Q$ when $(m,k) = (1,1)$, and discover the candidates $R_{h}Q$ and $R_{c2}Q$ when $(m,k) = (4,2)$. Based on Definition 5.8, we select and maintain the query candidate $R_{h}Q$ in $S_Q$. Also, we find $\alpha_{\text{max}} = \alpha_{h}(R_{h}Q,s) = 1$ reaches $s$, indicating that the candidates in the subsequent rounds cannot yield a larger named entity coverage with lower structural complexity. Algorithm 3 terminates and returns the set $S_Q = \{R_{h}Q\}$.

5.4 Summary

Our synthesis algorithm SQUID is an instantiation of a new synthesis paradigm of the multi-modal synthesis, which reduces the synthesis problem to a multi-target optimization problem. We now formulate and prove the soundness, completeness, and optimality of SQUID with three theorems as follows, which are proved in [49].

- **Theorem 5.2 (Soundness).** For any $R_Q \in S_Q$, where $S_Q$ is returned by Algorithm 3, $R_Q$ must contain all the positive tuples in $R_p^*$ and exclude the negative tuples in $R_n^*$.

- **Theorem 5.3 (Completeness).** If an MMCQS problem instance has an $(m,k)$-bounded query as its solution and $k \leq K$, the set $S_Q$ returned by Algorithm 3 is not empty.

- **Theorem 5.4 (Optimality).** Denote $I = \{(m,k) \mid 1 \leq m \leq K \cdot |R'|, 1 \leq k \leq \min(K,m)\}$ and $\tilde{S} = \bigcup_{(m,k) \in I} S_C(m,k)$. The returned query set $S_Q$ of Algorithm 3 satisfies:
  - $R_Q \preceq_s R_Q$ for every $R_Q \in S_Q$ and $R'_Q \in \tilde{S}$.
  - There do not exist $R_Q \in \tilde{S} \setminus S_Q$ and $R'_Q \in \tilde{S}$ such that $R'_Q \preceq_s R_Q$ and $R_Q \not\preceq_s R'_Q$.

6 Implementation

Established upon the industrial Datalog-based Java program analyzer in Ant Group, SQUID synthesizes conjunctive queries to support code search tasks in Java programs. Noting that our approach is general enough to support the conjunctive query synthesis for any Datalog-based analyzer as long as the generated relations can be formulated by Definition 3.2. In what follows, we provide more implementation details of SQUID.

Synthesis Input Configuration. We design a user interface to convenience the users to specify examples in a code snippet. Specifically, the users can copy a desired grammatical construct from their workspace as a positive example or write a positive example manually. By mutating a positive example, the users can create more positive and negative examples, eventually forming an example program. Then we convert the program to the relational representation, which consists of 173 relations with 1,093 attributes in total, and partitions a
relation into two parts to induce positive and negative tuples. To extract the named entities from the natural language description, we leverage the named entity recognition [31] and construct the dictionary of entities to filter unnecessary named entities in the post-processing. Specifically, the dictionary contains 205 words, which are the keywords describing grammatical constructs in Java programs, such as “method”, “parameter”, and “return”. Furthermore, we also instantiate the function $h$ in Definition 5.6 to bridge the program relational representation with natural language words. We publish all the synthesis specifications and dictionary of entities online [48].

**Synthesis Algorithm Design.** Based on the language schema of Java, we construct the schema graph offline and persist it for synthesizing queries for a given synthesis specification. Instead of invoking the Datalog-based analyzer, we implement a query evaluator for conjunctive queries upon the relational representation to identify the refinable queries and query candidates, which can improve the efficiency of the query evaluation during the synthesis. In the bounded refinement, we set the multiplicity bound $K$ to 2 by default to support code search tasks. To efficiently synthesize string constraints, we leverage the generalized suffix automaton [33] to identify the longest common substrings of a set of string values, which returns the string predicate $p$ and the string literal $\ell$ with low time overhead. Currently, SQUID utilizes four predicates for string match, while we can further extend it to support regex match by adopting existing regex synthesis techniques [27, 10] to Algorithm 2.

### 7 Evaluation

To quantify the effectiveness and efficiency of SQUID, we conduct a comprehensive empirical evaluation and answer the following four research questions:

- **RQ1:** How effective is SQUID in the conjunctive query synthesis for code search tasks?
- **RQ2:** How big are the benefits of the representation reduction and the bounded refinement in terms of efficiency?
- **RQ3:** Is the query candidate selection effective and necessary for the synthesis?
- **RQ4:** How does SQUID compare to other approaches that could be used in our problem?

**Benchmark.** There are no existing studies targeting our multi-modal synthesis problem, so we construct a new benchmark for evaluation, which consists of 31 code search tasks. As shown in Table 1, the tasks cover five kinds of grammatical constructs, namely variables, expressions, statements, methods, and classes. Specifically, 14 tasks are the variants of C++ search tasks in [34] or the query synthesis tasks in [46]. We also consider more advanced tasks deriving from real demands. For example, Task 2 originates from the coding standard of a technical unit in Ant Group, while Task 21 is often conducted when the developers check the usage of the log4j library to improve reliability. For each task, we specify examples in a program and a sentence as the natural language description. The program is fed to the commercial Datalog-based analyzer in Ant Group to generate the relational representation and the relation partition, while the natural language description is processed via the named entity recognition technique [31]. The columns $L$ and $(P, N)$ in Table 1 indicate the line numbers of the programs and the numbers of positive/negative tuples, respectively.

**Experimental Setup.** We conduct all the experiments on a Macbook Pro with a 2.6 GHz Intel® Core™ i7-9750H CPU and 16 GB physical memory.
7.1 Overall Effectiveness

To evaluate the effectiveness of SQUID, we run it upon the synthesis specification for each code search task, examining whether the synthesized queries express the intent correctly, and meanwhile, measure the time cost of synthesizing queries in each task.

In Table 1, the column $|\mathbf{G}_Q|$ indicates the numbers of the relations, equality constraints, and string constraints. The column $k$ shows the maximal multiplicity of a relation in a synthesized query, while the column $|\mathcal{G}_T|$ indicates the numbers of nodes and edges in the subgraph of the schema graph induced by $\mathcal{R}' \cup \{\text{STR}\}$. The time cost of SQUID is shown in the column $T_0$. According to the statistics, we can obtain two main findings. First, SQUID synthesizes the queries for all the tasks successfully. It manipulates more than three relations in Tasks 22 and 25, which are even non-trivial for a human to achieve. Second, SQUID synthesizes the queries with a quite low time cost. The average time cost is 2.56 seconds, while most of the tasks are finished in three seconds.

As mentioned in § 6, SQUID performs the bounded refinement with the multiplicity bound $K = 2$. In our benchmark, five code search tasks demand several relations appear two times. In practice, the searching condition can hardly relate to more than two grammatical constructs of the same kind, so our setting of the multiplicity bound $K$ enables SQUID to synthesize

| ID | Description | $L$ $(P, N)$ | $|\mathbf{G}_Q|$ | $k$ $|\mathcal{G}_T|$ | $T_0$ (s) |
|----|-------------|-------------|----------------|----------------|----------|
| 1  | Local variables with double type | (2, 2) | (2, 1, 1) | 1 (10, 19) | 2.36 |
| 2  | Float variables of which the identifier contains “cash” | (3, 1) | (3, 2, 2) | 1 (9, 17) | 2.37 |
| 3  | Public field variables of a class | (2, 1) | (2, 1, 1) | 1 (10, 21) | 2.30 |
| 4  | Public field variables whose names use “cash” as suffixes | (3, 2) | (2, 2, 2) | 1 (10, 23) | 2.49 |
| 5  | Arithmetic expressions using double-type operands | (2, 2) | (3, 2, 2) | 1 (9, 29) | 2.62 |
| 6  | Cast expressions from double-type to float-type [46] | (1, 2) | (3, 2, 2) | 2 (9, 23) | 2.31 |
| 7  | Arithmetic expressions only using literals as operands | (2, 3) | (3, 1, 2) | 1 (9, 29) | 2.76 |
| 8  | Expressions comparing a variable and a boolean literal [34] | (2, 1) | (2, 1, 1) | 1 (8, 29) | 2.36 |
| 9  | New expressions of ArrayList | (2, 1) | (1, 2) | 1 (10, 25) | 2.20 |
| 10 | Logical conjunctions with a boolean literal [34] | (3, 1) | (2, 1, 2) | 1 (9, 29) | 2.31 |
| 11 | Float increment expression [46] | (1, 2) | (3, 2, 2) | 1 (9, 23) | 2.55 |
| 12 | Expressions comparing two strings with “==” [34] | (2, 1) | (3, 2, 2) | 1 (11, 46) | 3.01 |
| 13 | Expressions performing downcasting [46] | (2, 2) | (3, 2, 2) | 0 (11, 39) | 2.63 |
| 14 | The import of LocalTime | (2, 1) | (1, 0, 1) | 1 (9, 24) | 2.17 |
| 15 | The import of the classes in log4j | (3, 1) | (1, 0, 1) | 1 (9, 22) | 2.22 |
| 16 | Labeled statements using “err” as the label [34] | (2, 2) | (1, 2, 2) | 1 (10, 21) | 2.18 |
| 17 | If-statements with a boolean literal as a condition [34] | (2, 1) | (2, 1, 0) | 1 (9, 17) | 2.24 |
| 18 | For-statements with a boolean literal as the condition [34] | (2, 1) | (2, 1, 0) | 1 (10, 25) | 2.31 |
| 19 | Invocation of unsafe time function “localtime” [34] | (2, 1) | (2, 1, 1) | 1 (10, 23) | 2.23 |
| 20 | Public methods with void return type [34] | (2, 1) | (3, 2, 2) | 1 (11, 26) | 2.36 |
| 21 | Methods receiving a parameter with Log4jUtils type | (2, 1) | (3, 1) | 1 (9, 20) | 2.45 |
| 22 | Methods using a boolean parameter as a if-condition [46] | (2, 1) | (2, 1, 0) | 1 (11, 26) | 3.23 |
| 23 | Methods creating a File object | (2, 1) | (3, 1, 2) | 1 (12, 30) | 2.38 |
| 24 | Mutually recursive methods [34, 46] | (2, 1) | (2, 2, 2) | 0 (11, 27) | 2.42 |
| 25 | Overriding methods of classes [46] | (2, 2) | (5, 5, 0) | 2 (8, 22) | 5.89 |
| 26 | User classes with “login” methods | (2, 1) | (2, 1, 2) | 1 (11, 28) | 2.53 |
| 27 | Classes containing a field with Log4jUtils type | (2, 1) | (3, 2, 2) | 1 (12, 35) | 2.42 |
| 28 | Classes having a subclass | (2, 1) | (2, 1, 0) | 2 (13, 33) | 2.21 |
| 29 | Classes implementing Comparable interface | (2, 1) | (2, 1, 1) | 1 (13, 35) | 2.58 |
| 30 | Classes containing a static method | (2, 1) | (3, 2, 1) | 1 (11, 28) | 2.46 |
| 31 | Java classes with main functions | (2, 1) | (2, 2, 1) | 1 (10, 28) | 2.35 |
queries for code search tasks in real-world scenarios. Meanwhile, we quantify the time cost of the synthesis in the cases of $K = 3$ and $K = 4$. Averagely, SQUID takes 2.71 seconds and 3.98 seconds under the two settings, respectively. Thus, the overhead increases gracefully when $K$ increases, demonstrating the great potential of SQUID in efficiently synthesizing more sophisticated queries with a larger multiplicity bound.

7.2 Ablation Study on Efficiency

We evaluate two ablations of SQUID, namely SQUIDNRR and SQUIDNBR, to quantify the impact of the representation reduction and the bounded refinement on the efficiency.

- **SQUIDNRR**: This ablation of SQUID does not perform the representation reduction but still leverages Algorithm 2 to conduct the bounded refinement.
- **SQUIDNBR**: The ablation performs the representation reduction as SQUID does, while it enumerates all the query graphs and permits each relation to appear at most $K$ times.

We measure the time cost of two ablations to quantify their efficiency. Specifically, we set the time budget for synthesizing queries for a single task to 30 seconds, as a synthesizer would have little practical value for the real-world code search if it ran out of the time budget.

Figure 5 shows the comparison of the time cost of SQUID and the two ablations. First, the representation reduction can effectively reduce the time cost. Specifically, the average time cost of SQUIDNRR is 8.98 seconds, indicating that the representation reduction introduces a 71.49% reduction over the time cost. Second, the bounded refinement has a critical impact on the efficiency of SQUID. Without the refinement, SQUIDNBR has to explore the huge search space induced by the non-dummy relations, making 14 out of 31 tasks cannot be finished within the time budget, such as Task 4, Task 5, etc. For the failed tasks, we do not show the time cost of SQUIDNBR in Figure 5. SQUIDNBR also takes much more time than SQUID, consuming 7.89 seconds on average, even if it successfully synthesizes the queries.

To investigate how the efficiency is improved, we further measure the size of the subgraph of the schema graph induced by $R' \cup \{\text{STR}\}$. Initially, the schema graph contains 174 nodes (including the node depicting STR) and 1,093 edges. As shown in the column $|G'_r|$ of Table 1, the induced subgraph only contains around ten nodes and no more than fifty edges. Although SQUIDNRR prunes unnecessary relations by enumerating several bounded queries at the beginning of the refinement, it has to spend more time on the query enumeration than SQUID, which demonstrates the critical role of the bounded refinement in our synthesis. Besides, the running time of SQUIDNBR is similar to SQUID on several benchmarks, such as Task 1, Task 2, Task 3, Task 9, etc., while it takes much longer time than SQUID in other benchmarks.
Although SQUIDNBR does not discard infeasible queries, it still benefits from representation reduction. When a desired query is of small size and the reduced program representation induces a small schema graph $G'_T$, SQUIDNBR can terminate to find an optimal query by enumerating a few candidates. However, if $G_Q$ and $G'_T$ are large, SQUIDNBR enumerates a large number of infeasible queries, which introduces significant overhead.

### 7.3 Impact of Selection

To measure the impact of the query candidate selection, we adapt each metric separately for candidate prioritization. Specifically, we alter Algorithm 3 and select the queries minimizing the structural complexity and maximizing the named entity coverage, respectively. We then count the returned queries and inspect whether they are desired ones or not.

Figure 6(a) shows the numbers of the synthesized queries with the structural complexity as the metric. As we can see, SQUID produces non-desired queries in 12 tasks, while the returned set of synthesized queries in 10 tasks do not contain any desired query. For Task 3 and Task 30, it provides the desired queries along with non-desired ones, which makes the users confused about how to select a proper one. Similar to $R_{Q1}^1$ in Example 5.8, the non-desired queries are caused by the over-fitting of positive and negative examples. Although they have the simplest form of the selection conditions, the relationship of grammatical constructs mentioned in the natural language description is not constrained, making synthesized queries not express the users' intent correctly.

Figure 6(b) shows the numbers of the queries maximizing the named entity coverage. SQUID returns at least one non-desired query for seven tasks. Similar to $R_{Q2}^2$ in Example 5.8, non-desired queries come from over-complicated selection conditions. Although the selected queries have the same named entity coverage, several queries contain more atomic formulas than the desired ones, posing stronger restrictions upon the code. Besides, the synthesized queries in several tasks, e.g., Task 11 and Task 26, have complex selection conditions although they are equivalent under the context of code search. However, such queries exhibit higher structural complexity, posing more difficulty in understanding them.

### 7.4 Comparison with Existing Techniques

To the best of our knowledge, no existing technique or implemented tool targets the same problem as SQUID. To compare SQUID with existing effort, we adapt the state-of-the-art Datalog synthesizer EGS [46] as our baseline. Originally, it synthesizes a conjunctive query to separate a positive tuple from all the negative ones and then group all the conjunctive queries as the final synthesis result, which can be theoretically a disjunctive query. However,
EGS does not synthesize string constraints and only prioritizes feasible solutions based on their sizes, i.e., the structural complexity in our work. Thus, we construct two adaptions, namely EGS-STR and EGS-STRDual, to synthesize the queries under our problem setting.

- EGS-STR computes the longest substring of each string attribute in the positive tuple such that the string values of the attributes in negative ones do not contain it as the substring. We follow the priority function in EGS, which consists of the number of undesirable tuples eliminated per atomic constraint and the size of a query, to accelerate searching a query candidate with a small size. Finally, it obtains a query candidate for each positive tuple and groups the candidates as a result.

- EGS-STRDual further extends EGS-STR by considering the named entity coverage. Specifically, it prioritizes the refinable queries according to the three metrics, including the number of undesirable tuples eliminated per atomic constraint, the named entity coverage, and the size of a query. Other settings are the same as the ones of EGS-STR.

Figure 7 shows the results of the comparison. On average, EGS-STR and EGS-STRDual spend 2.85 and 3.18 seconds on a synthesis instance, respectively, while the average time cost of SQUID is 2.56 seconds. Although EGS-STR and EGS-STRDual accelerate the searching process with priority functions as the heuristic metrics, they have to process the positive tuples in each round, and thus, the number of positive tuples can increase the overhead.

Meanwhile, the two baselines only succeed synthesizing queries for 9 and 14 tasks, respectively. There are two root causes of their failures in synthesizing desired queries. First, they synthesize the query candidates for each positive query separately and, thus, are more prone to over-fitting problems than SQUID. Second, the core algorithms of EGS and the two adaptions, which are the instantiations of inductive logic programming, can not guarantee the obtained solutions are optimal under the given metrics. As reported in [46], the query may not be of the minimal size if EGS leverages the number of undesirable tuples eliminated per atomic constraint to accelerate the searching process of a query candidate. In our problem, our dual quantitative metrics increase the difficulty of achieving the optimal solutions with the two adaptions, which causes the failures of the code search tasks.

7.5 Discussion

In what follows, we demonstrate the discussions on the limitations of SQUID and several future works, which can further improve the practicality of our techniques.
Limitations. Although SQUID is demonstrated to be effective for code search, it has two major limitations. First, SQUID cannot synthesize the query where the multiplicity of a relation is larger than the multiplicity bound \( K \). In other words, Theorem 5.3 actually ensures partial completeness. Although we may achieve completeness for all realizable instances by enumerating queries until obtaining a query candidate in Algorithm 3, SQUID would fail to terminate for unrealizable instances. Second, SQUID does not support synthesizing the queries with logical disjunctions. However, when a code search task involves the matching of multiple patterns, SQUID would not discover the correct queries, which are out of the scope of the conjunctive queries.

Future Works. In the future, we will attempt to propose an efficient decision procedure to identify unrealizable instances. Equipped with the decision procedure, we can only enumerate queries for realizable instances and generalize the query refinement by discarding the multiplicity bound. Besides, it would be promising to generalize SQUID for disjunctive query synthesis. One possible adaption is to divide positive tuples into proper clusters and synthesize a conjunctive part for each cluster separately, following existing studies such as EGS [46] and RHOSYNTH [15]. In addition, we aim to expand SQUID to diverse program domains, such as serverless functions [50] and programs running on networking devices [57]. These use cases have gained significant attention in recent years, which can pose new challenges on code search where new approaches may be needed. Lastly, it would be meaningful to combine SQUID with the techniques in the community of human-computer interaction [8] to unleash its benefit for practice use.

8 Related Work

Multi-modal Program Synthesis. There has been a vast amount of literature on the multi-modal synthesis [11, 4, 9, 10, 40, 16, 34]. For example, the LTL formula synthesizer LTLTalk [16] maximizes the objective function that measures the similarity between the natural language description and the LTL formula, and searches for the optimal solution that distinguishes the positive and negative examples. SQUID bears similarities to LTLTalk in terms of the prioritization, while we use the named entities to avoid the failure of semantic parsing of a sentence. Another closely related work is a query synthesizer named SporQ [34]. Based on code examples and user feedback, SporQ iterates its PBE-based synthesis engine to refine the queries, which demands verbose user interactions and a long time period. In contrast, SQUID automates the code search by solving a new multi-modal synthesis problem, which only requires the users to specify code examples and a natural language description, effectively relieving the user’s burden in the searching process.

Component-based Synthesis. Several recent studies aim to compose several components (e.g., the classes and methods in the libraries) into programs that achieve target functionalities [25, 18, 24, 38, 20, 21]. Typically, SyPet [14] and APIphany [19] both use the Petri net to encode the type signature of each function, and collect the reachable paths to enumerate the well-typed sketches of the programs, which prunes the search space at the start of the synthesis. In our work, SQUID leverages the schema graph to guide the enumerative search, which share the similarity with existing studies. However, our enumerative search space does not consist of the reachable paths in the schema graph, and instead, contains different choices of selecting its nodes and edges. Besides, unlike prior efforts [14, 24, 20, 19], SQUID computes the activated relations and then discards unnecessary components, i.e., dummy relations, which distinguishes SQUID significantly from other component-based synthesizers.
Synthesizing Conjunctive Queries for Code Search

**Datalog Program Synthesis.** There have been many existing efforts of synthesizing Datalog programs [2, 42, 39, 43, 32]. For example, **ZAATAR** [2] encodes the input-output examples and Datalog programs with SMT formulas, and synthesizes the candidate solution via constraint solving. Unlike constraint-based approaches, **ALPS** [42] and **GENSYNTH** [32] synthesize target Datalog programs via the enumerative search, which is similar to our synthesis algorithm. However, existing studies do not tackle a large number of relations in the synthesis [2, 42, 32] or pursue an optimal solution with respect to a natural language description. Meanwhile, they do not support the synthesis of string constraints, making their approaches incapable of string matching-based code search. In contrast, **SQUID** ensures soundness, completeness, and optimality simultaneously and synthesizes string constraints for string matching, showing its potential in assisting real-world code search tasks.

**Datalog-based Program Analysis.** The past few decades have witnessed the increasing popularity of Datalog-based program analysis [22, 52, 55, 3, 44]. For example, **CodeQL** encodes a program with a relational representation and exposes a query language for query writing [3]. Several analyzers target more advanced semantic reasoning. For example, the points-to and alias facts are depicted by two kinds of relations in **DOOP** [6], and meanwhile, pointer analysis algorithms are instantiated as Datalog rules [44]. Other properties, such as def-use relation and type information, can also be analyzed by existing analyzers [26, 37]. Our effort has shown the opportunity of unleashing the power of Datalog-based program analyzers seamlessly to support the code search automatically.

**9 Conclusion**

We propose an efficient synthesis algorithm **SQUID** for a multi-modal conjunctive query synthesis problem, which enables automatic code search using a Datalog-based program analyzer. **SQUID** reduces the search space via the representation reduction and the bounded refinement, and meanwhile, conducts the query candidate selection with dual quantitative metrics. It efficiently synthesizes the queries for 31 code search tasks with the guarantees of soundness, completeness, and optimality. Its theoretical and empirical results offer strong evidence of its practical value in assisting code search in real-world scenarios.

**References**


36:30 Synthesizing Conjunctive Queries for Code Search


