



Synthesizing Graph Queries from Demonstrations

XIAOYU LIU, Simon Fraser University, Canada

QIKANG LIU, Simon Fraser University, Canada

EVAN DYCE, Simon Fraser University, Canada

KEVAL VORA, Simon Fraser University, Canada

YUEPENG WANG, Simon Fraser University, Canada

Writing graph queries is challenging for non-experts due to the complexity of graph data models and the need to identify proper graph patterns. While recent research has advanced query synthesis for relational and document databases, the problem of synthesizing graph queries remains under-explored. We present a novel approach for synthesizing graph queries from computation demonstrations, where users specify the desired output through expressions over properties of input graphs. Our method addresses the challenge of inferring meaningful graph patterns for matching and efficiently constructing the remaining components of the query. Specifically, we combine graph mining, which identifies candidate patterns across input graphs, with deduction-based pruning, which guides an efficient synthesis of the filtering predicate and return clause. We have implemented our approach in a tool called DMINER and evaluated it on 90 benchmarks. Experimental results show that DMINER successfully synthesizes desired queries for 87 benchmarks, with an average synthesis time of 0.6 seconds per query. This outperforms both enumerative search and LLM baselines. We also conducted a user study, which shows that users can provide demonstrations with modest effort and 87.5% of the provided demonstrations are sufficient for DMINER to synthesize the desired query.

CCS Concepts: • **Software and its engineering** → **Programming by example; Automatic programming;**
• **Information systems** → **Database utilities and tools.**

Additional Key Words and Phrases: Program Synthesis, Graph Queries.

ACM Reference Format:

Xiaoyu Liu, Qikang Liu, Evan Dyce, Keval Vora, and Yuepeng Wang. 2026. Synthesizing Graph Queries from Demonstrations. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 392 (October 2026), 28 pages. <https://doi.org/10.1145/3839524>

1 Introduction

Graph databases are increasingly adopted across a wide range of domains, including social network analysis [46], biological data analysis [57], and knowledge graph applications [61]. Extracting information from these databases typically requires writing complex graph queries in languages such as Cypher [19]. A graph query typically consists of a *Match* clause with a *pattern* describing the structural relationship between different types of nodes and edges, as well as *Filter* and *Return* clauses to filter matched subgraphs and project out the results. However, writing a graph query can be challenging for non-experts, because it typically demands a good understanding of both the query mechanism and the underlying graph data model.

Authors' Contact Information: Xiaoyu Liu, Simon Fraser University, Burnaby, Canada, xla411@sfu.ca; Qikang Liu, Simon Fraser University, Burnaby, Canada, qla116@sfu.ca; Evan Dyce, Simon Fraser University, Burnaby, Canada, evan_dyce@sfu.ca; Keval Vora, Simon Fraser University, Burnaby, Canada, keval@cs.sfu.ca; Yuepeng Wang, Simon Fraser University, Burnaby, Canada, yuepeng@sfu.ca.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/10-ART392

<https://doi.org/10.1145/3839524>

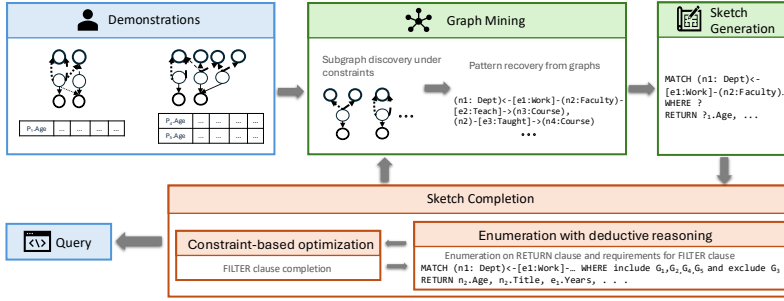


Fig. 1. Schematic workflow.

While recent research explored various synthesis techniques for database queries, including those based on input-output examples [18, 31, 42, 48, 50] and user demonstrations [62], most of the work in this line focused on relational or document databases, leaving the problem of synthesizing graph database queries under-explored. Given the substantial differences in data models and query mechanisms: relational queries rely on joins across tables, document queries extract information from hierarchical structures via pipelines, and graph queries perform pattern matching over graphs, it is difficult, and often infeasible, to re-purpose existing synthesizers through data model or query translation. Thus, we investigate the problem of directly synthesizing graph queries in this paper.

As shown in prior work [62], *computation demonstration* is an intuitive and effective way to express user intent for extracting data from relational databases. Here, the computation demonstration not only specifies what is the output for a given input, but also describes how to compute output from the input. Following this idea, we adapt the concept of computation demonstration as the high-level specification for synthesizing graph queries. Specifically, given a small input graph G , the user provides a demonstration table T that represents the desired output, using expressions over the properties of nodes and edges in G . For example, an expression in the demonstration table can be $P_1.Age$, where P_1 is a node in the input graph, and Age is its property. Then our goal is to synthesize a graph query that is consistent with the input graphs and demonstration tables.

While the demonstrations provide useful information about what data should be extracted, synthesizing graph queries from these demonstrations remains a challenging task. This challenge arises from two main aspects. First, graph queries rely on matching *patterns* to retrieve subgraphs containing relevant data, but the pattern is not explicitly specified in the demonstration. In fact, inferring these patterns is non-trivial, because there are many possible patterns given certain types of nodes and edges, and some of those patterns can be complex. For example, suppose a graph contains two types of nodes A and B and one type of edge E . For this case, the pattern can be $(n_1: A) \rightarrow [e_1: E] \rightarrow (n_2: B)$, meaning that node n_1 with label A connects to node n_2 with label B through edge e with label E ; or instead it can be $(n_1: A) \rightarrow [e_1: E] \rightarrow (n_2: B) \leftarrow [e_2: E] \rightarrow (n_3: A)$, or $(n_1: B) \leftarrow [e_1: E] \rightarrow (n_2: A) \rightarrow [e_2: E] \rightarrow (n_3: B)$, and so on. Second, even after identifying the correct pattern, the synthesizer must find an appropriate filtering predicate and the return list. This step involves exploring a large combinatorial space of possible property predicates and return expressions, as graph elements may have many properties and can be compared in numerous ways.

To address these challenges, we develop a synthesis technique that combines *graph mining* and *deduction-based pruning*. In general, graph mining refers to the process of discovering non-trivial patterns from graphs [6], which is an established technique widely used in fields such as social network analysis [49]. However, graph mining technique has not been utilized to synthesize graph queries before. We use graph mining to generate graph patterns because enumerating arbitrary patterns quickly becomes infeasible: the number of possible patterns grows exponentially with

pattern size, and most such patterns are irrelevant to the user's intended query. Demonstrations provide useful signals that help focus this search. In particular, the desired pattern should occur consistently across the input graphs and should account for the nodes and edges that contribute to the output tables. By mining subgraphs that satisfy these properties and recovering patterns from them, we prioritize patterns that are more likely to express the user's intent while avoiding a large portion of the irrelevant pattern space. Next, to synthesize the filtering predicate and return clause efficiently, we adapt the classic synthesis technique based on deduction-based pruning [17, 18, 52, 53], which has been successful in synthesizing various programs. This technique uses deductive reasoning to eliminate impossible parts of the search space while trying to generate a correct program. For our graph query synthesis, we check whether the evaluation result of the generated candidate query is consistent with user-provided demonstrations. If inconsistent, we prune the candidate query and identify multiple similar incorrect queries to quickly prune the search space.

As shown in Figure 1, our synthesis approach consists of two main steps: sketch generation with graph mining and sketch completion. From user demonstrations, we perform graph mining to identify candidate patterns across all input graphs and generate a query sketch accordingly. Specifically, we define *commonality* and *output-coverage* constraints to extract subgraphs satisfying these conditions as candidate matches, and recover candidate patterns from each candidate match. For each candidate pattern, we construct a query sketch by populating the Match clause with the pattern, while leaving the Filter clause unspecified and the Return clause partially specified.

To complete the query sketch efficiently, we must avoid redundantly enumerating many distinct predicates that are equivalent over the matched subgraphs. We therefore use a two-phase approach. In the first phase, we complete the Return clause and derive high-level filtering requirements, namely which subgraphs should be retained by the Filter clause. We encode possible sketch completions as a logical formula and perform symbolic enumerative search. When an incorrect completion is found, we infer multiple incorrect queries with deductive reasoning and generate a blocking clause to refine the formula. This process repeats until the completed query is consistent with the demonstration tables. In the second phase, we synthesize a filtering predicate that satisfies the high-level requirements. To ensure the predicate is both correct and concise, we frame its synthesis as a constraint optimization problem, where correctness is enforced through hard constraints and conciseness is captured by objective functions. This formulation helps us avoid overfitting to the provided demonstrations.

Based on these techniques, we developed a tool called DMINER for synthesizing graph queries from user demonstrations. We evaluated DMINER on 90 benchmarks collected from various sources, including StackOverflow and the Neo4j forum [37]. The results show that DMINER can successfully synthesize the desired query for 87 out of 90 benchmarks, with an average synthesis time of 0.6 seconds per query. This outperforms both enumerative search and LLM baselines. In addition, to investigate the usability and practicality of synthesizing graph queries by demonstrations, we also conducted a small-scale user study with 8 participants on 5 tasks. The results show that participants were able to provide demonstrations in 196 seconds per task on average, and 87.5% of the provided demonstrations are of high quality and DMINER can synthesize desired queries from them.

Contributions. In summary, we make the following main contributions in this paper.

- We adopt the concept of computation demonstrations as a high-level specification for a new application: synthesizing graph queries from user demonstrations.
- We propose a new synthesis technique that performs graph mining to synthesize patterns for graph queries. To our best knowledge, this is the first technique that uses graph mining for synthesizing graph queries.

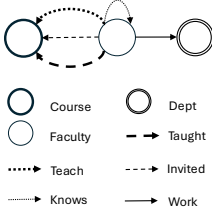


Fig. 2. Schema of the database.

Label	Property	Data Type
Course	Capacity	Integer
Faculty	Age	Integer
	Name	String
	Title	String
Dept	Name	String
Work	Years	Integer

Match clause
`MATCH (n1:Dept) <- [e1:Work] - (n2:Faculty),
 (n2) - [e2:Teach] -> (n3:Course),
 (n2) - [e3:Taught] -> (n4:Course)`

Filter clause
`WHERE n2.Age >= 40`

Return clause **Projections**
`RETURN n2.Age, n2.Title, e1.Years, n1.Name,`

`max(n3.Capacity) / min(n4.Capacity)`
Aggregations

Fig. 3. Desired query.

- We adapt the standard deduction-based pruning technique with new deductive reasoning tailored towards graph queries, which enables effective pruning of the search space.
- We implement our approach in a tool called DMINER and evaluate it on 90 benchmarks from various sources. The results show that DMINER successfully synthesizes desired queries for 87 benchmarks and outperforms both enumerative search and LLM baselines. In addition, a user study of 8 participants over 5 tasks shows that users can provide multiple demonstrations with modest effort, which demonstrates the practicality of our approach.

2 Overview

In this section, we illustrate how to synthesize graph queries from user demonstrations through a running example. Consider a graph database with the schema shown in Figure 2, the nodes correspond to entities, namely Course, Faculty, and Dept, and the edges correspond to their relationships, such as Teach, Taught, Work, and so on.

Suppose a user wants to query the following: for faculties at least 40 years old, compute the ratio of maximum capacity of the courses they teach this semester to the minimum capacity of the courses they taught in the previous semester, grouping by age and title of the faculty, department name, and years of service in the department.¹ The desired query for this task is shown in Figure 3. It includes: (a) a Match clause to find subgraphs with any faculty working in a department who teaches and taught a course; (b) a Filter clause to remove subgraphs where the faculty is aged less than 40; and (c) a Return clause containing the faculty information (which serves as the grouping keys) and the maximum capacity of the courses they teach this semester dividing the minimum capacity of the courses they taught before. Writing such a complex graph query requires significant expertise. However, DMINER can help to synthesize this query automatically.

User demonstrations. To use DMINER, the user can demonstrate their intention as illustrated in Figure 4. Unlike traditional programming-by-example tools, where the synthesis task is specified solely through input-output examples, DMINER allows the user to demonstrate how to calculate the output value through a demonstration table. In particular, the user can provide expressions over the node and edge properties of the input graph in the table. For example, Figure 4a shows an input graph and Figure 4c shows the demonstration table. This notation of user demonstration is inspired by prior work about synthesizing SQL queries from demonstrations [62], and we adapt it to the setting of synthesizing graph queries.

Overall approach. At a high level, our synthesis approach consists of sketch generation and sketch completion. For sketch generation, we perform graph mining to discover candidate patterns across all input graphs. Each pattern is used to construct a query sketch with a complete Match clause corresponding to that pattern. For sketch completion, we focus on synthesizing the Filter

¹In Cypher, the group by operation is not as explicit as in SQL, one can simply include the properties in Return clause alongside aggregate functions for group by.

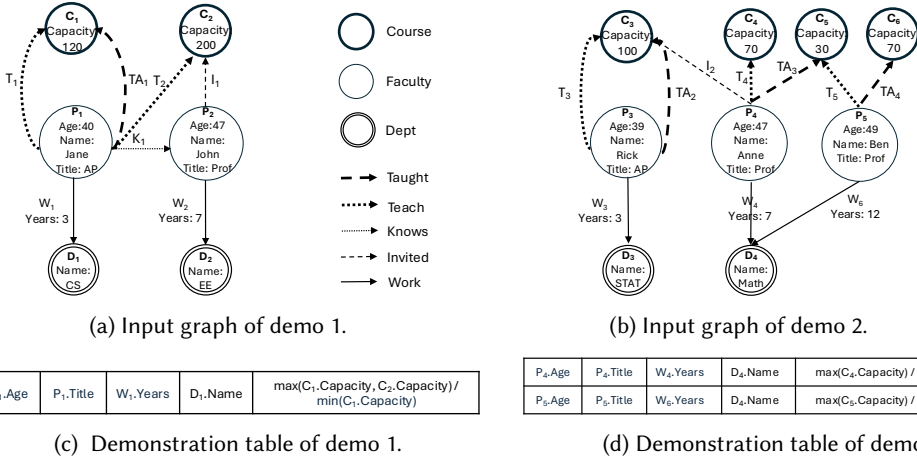


Fig. 4. Sample user demonstrations.

and Return clauses to generate a complete query. Specifically, we adapt the enumerative search technique and perform deduction-based pruning [17, 18, 52, 53] tailored for graph queries. Next, we will explain the synthesis process in more detail.

2.1 Sketch Generation with Graph Mining

Given that the core of a query sketch is the pattern, let us first illustrate how to identify patterns. Although a possible approach is a simple enumerative search, the number of patterns we would need to enumerate increases significantly as the pattern size increases. Moreover, a large portion of those patterns have no matches in the input graphs, which are definitely not the desired patterns. To address this problem, we perform graph mining to identify the *common pattern* across all input graphs such that the *matched subgraph can cover all information in the demonstration table*. Towards this, we first find candidate matches (which are subgraphs) satisfying two constraints and then recover the pattern from candidate matches.

Finding candidate matches. We find subgraphs satisfying the following constraints.

- **Commonality constraint.** Recall overall goal of graph mining is to discover a common pattern that can match all input graphs. Since matched subgraphs can differ across different inputs and pattern matching is computationally expensive, we introduce a necessary condition to prune the space of candidate matches. We refer to this condition as commonality constraint, which requires subgraphs (1) only include nodes and edges whose labels are shared among all input graphs, and (2) are not larger than the largest weakly connected component of any input graph.
- **Output coverage constraint.** Since another goal of graph mining is to discover a pattern covering all information presented in the demonstration table, the output coverage constraint requires that the subgraph should include *at least one* node or edge of each label that occurs in the table. The reader may wonder why we do not require the subgraph to include *all* nodes and edges from the table. It turns out that such a strict requirement often causes candidate matches to overfit the input graphs in the demonstrations, while excluding more general candidate matches that could lead to the desired pattern. For example, consider a graph $A_1 - E_1 -> B_1 < - E_2 - A_2$, where nodes A_1 and A_2 have label A , node B_1 has label B , and edges E_1 and E_2 have label E . Suppose the demonstration table includes all nodes and edges in the graph. If the candidate match is required to cover every node and edge, it can only be the input graph itself, and the resulting pattern is $(:A)-[:E]->(:B)<-[:E]-(:A)$. This pattern is overly specific to the input graph and is the only pattern

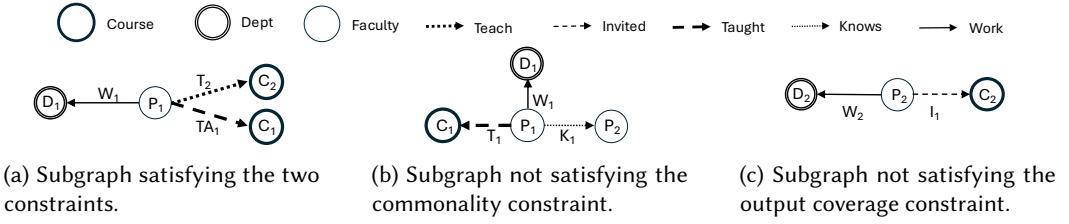


Fig. 5. Example subgraphs satisfying or not satisfying the constraints.

Graph	Pattern
	$(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] (n_2: \text{Faculty})$
	$(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] (n_2: \text{Faculty}) \rightarrow [e_2: \text{Teach}] \rightarrow (n_3: \text{Course})$
	$(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] (n_2: \text{Faculty}) \rightarrow [e_2: \text{Teach}] \rightarrow (n_3: \text{Course}),$ $(n_2) \rightarrow [e_3: \text{Taught}] \rightarrow (n_4: \text{Course})$

Fig. 6. An illustration of pattern recovery from candidate matches (subgraphs).

that can be discovered with the strict requirement. Meanwhile, patterns such as $(:A)-[:E]->(:B)$ cannot be discovered, even though they also match the graph and may be the intended pattern. We therefore relax this requirement to improve generalization and ensure completeness.

For example, Figure 5a shows a subgraph of the input graph in Figure 4a that satisfies both of the constraints. In contrast, Figure 5b shows a subgraph that does not satisfy the commonality constraint, because it includes an edge K_1 with the Knows label, which is not shared by the second input graph in Figure 4b. Figure 5c shows a subgraph violating the output coverage constraint. Observe that the output demonstration table in Figure 4c has one node P_1 with label Faculty, one edge W_1 with label Work, one node D_1 with label Dept, and two nodes C_1, C_2 with label Course. The output coverage constraint requires that the subgraph should include *at least one* node or edge that occurred in the output table for each of these labels. Since the subgraph in Figure 5c does not include P_1, W_1 or D_1 (thus not covering label Faculty, Work, or Dept), it does not satisfy the output coverage constraint.

Recovering patterns from candidate matches. After finding candidate matches satisfying the commonality and output coverage constraints, we recover patterns from each subgraph. The high-level idea is to transform nodes in the subgraph to *node patterns* and transform edges to *edge patterns* following the graph structure. For instance, let us consider the candidate match in Figure 5a. Figure 6 presents a step-by-step illustration of pattern recovery on this subgraph. The node D_1 is transformed to node pattern $(n_1: \text{Dept})$, meaning a node with label Dept. Edge W_1 is transformed to edge pattern $(e_1: \text{Work})$, and so on. Finally, this subgraph is transformed to the pattern $(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] (n_2: \text{Faculty}) \rightarrow [e_2: \text{Teach}] \rightarrow (n_3: \text{Course}), (n_2) \rightarrow [e_3: \text{Taught}] \rightarrow (n_4: \text{Course})$.²

Constructing the sketch. With the identified pattern, we can easily obtain the query sketch:

```
MATCH (n1:Dept) <- [e1:Work] (n2:Faculty) -> [e2:Teach] -> (n3:Course), (n2) -> [e3:Taught] -> (n4:Course)
WHERE ?* RETURN ?1.Age, ?2.Title, ?3.Years, ?4.Name, max(?5.Capacity) / min(?6.Capacity) (1)
```

Here, the Filter clause contains a hole $?*$, representing an unknown filtering predicate. The Return clause corresponds to a row in the demonstration table, where the concrete nodes and edges are replaced with holes such as $\text{max}(\text{?}_5.\text{Capacity})$. Because ?_5 corresponds to a Course node, and it is

²Patterns separated by commas can be viewed as components of a single pattern. Node patterns that share the same variable represent the connection points.

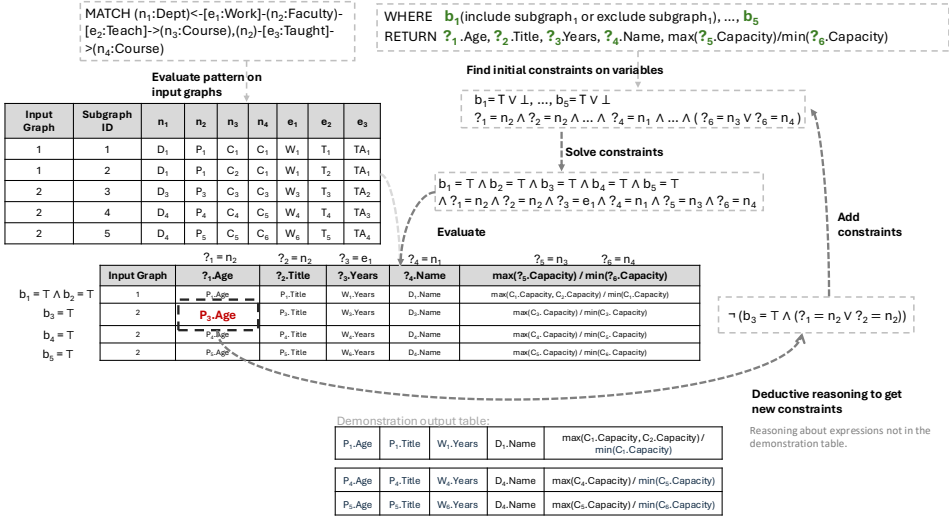


Fig. 7. Illustrative example of enumerative search with deduction-based pruning.

unclear which Course variable (e.g., n_3 , n_4) from the pattern is correct at this stage, we can specify its domain to be $?_5 \in \{n_3, n_4\}$. Similarly, $?_1, ?_2 \in \{n_2\}$, $?_3 \in \{e_1\}$, $?_4 \in \{n_1\}$, and $?_6 \in \{n_3, n_4\}$.

2.2 Sketch Completion

The goal of sketch completion is to fill all the holes in the Filter and Return clauses to produce a complete graph query. To avoid redundant enumeration of incorrect filtering predicates that are equivalent modulo the matched subgraphs, we group predicates into equivalence classes based on their results over the subgraphs and complete the sketch in two phases. First, we perform an enumerative search with deduction-based pruning to find the Return clause and determine “requirements” of the Filter clause (i.e., whether each matched subgraph should be retained). Then, we synthesize a predicate that satisfies these requirements using constraint-based optimization.

Synthesizing return clause and requirements of filter. Inspired by prior work about synthesis with deductive reasoning [17, 18, 53], we adapt the idea of enumerative search with deduction-based pruning to the context of sketch completion for graph queries. Specifically, as illustrated in Figure 7, we first find all subgraphs of the input graphs that match the pattern in the sketch. Then we can encode the search space as a formula and perform symbolic enumerative search to synthesize the return list and requirements of the filtering predicate. Specifically, we use variables $?_i$ for holes in the Return clause and introduce boolean variables b_i representing whether the i -th subgraph should be retained by the filter.

The initial constraint restricts the search space of the Return and Filter clauses and rules out syntactically invalid queries. The constraint is determined by the database schema and the expressions appearing in the output table. In particular, each hole in the return list can only be instantiated with pattern variables in its corresponding domain. For example, we need to fill $?_6$.Capacity and only nodes n_3, n_4 with label Course have a Capacity property, so the initial constraint includes the clause $?_6 = n_3 \vee ?_6 = n_4$. Therefore, the initial constraint is

$$b_1 = (T \vee \perp) \wedge \dots \wedge b_5 = (T \vee \perp) \wedge ?_1 = n_2 \wedge ?_2 = n_2 \wedge \dots \wedge (?_6 = n_3 \vee ?_6 = n_4)$$

Next, we resort to the constraint solver to get a model of the formula and check the result of the corresponding query. If the query result is consistent with the demonstration table, we can enter the next phase to synthesize a concrete predicate based on the requirements. Otherwise, we analyze

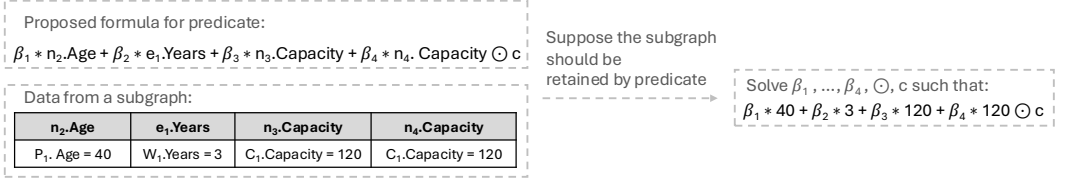


Fig. 8. Example of predicate synthesis from its requirements.

the reason of inconsistency and infer a blocking clause that can prune multiple incorrect queries from the search space. For example, suppose we obtain the following model

$$b_1 = b_2 = b_3 = b_4 = b_5 = \top \wedge ?_1 = n_2 \wedge ?_2 = n_2 \wedge ?_3 = e_1 \wedge ?_4 = n_1 \wedge ?_5 = n_3 \wedge ?_6 = n_4$$

As shown in Figure 7, the query result corresponding to this model is inconsistent with the user demonstration table, because the query result includes $P_3.Age$ but P_3 does not appear in the demonstration table. In fact, when subgraph 3 is retained (i.e., $b_3 = \top$), the execution result always includes node P_3 as long as n_2 is used to instantiate any hole. Given that n_2 can potentially fill in holes $?_1$ and $?_2$, we infer a blocking clause $\neg(b_3 = \top \wedge (?_1 = n_2 \vee ?_2 = n_2))$. Conjoining this clause to the formula can prune multiple incorrect queries in one step. After several iterations, we find the following model whose corresponding query result is consistent with the demonstration table:

$$b_1 = b_2 = b_4 = b_5 = \top \wedge b_3 = \perp \wedge ?_1 = n_2 \wedge ?_2 = n_2 \wedge ?_3 = e_1 \wedge ?_4 = n_1 \wedge ?_5 = n_3 \wedge ?_6 = n_4$$

Concise filter synthesis. Now given the requirements on the filtering predicate $b_1 = b_2 = b_4 = b_5 = \top \wedge b_3 = \perp$, we want to synthesize a concise predicate that filters out the third subgraph and retains the rest. Our solution is to reduce the synthesis problem into an optimization problem in Linear Integer Arithmetic (LIA). Specifically, we encode the requirements as hard constraints and preferences about the predicate format in an objective function. Without loss of generality, we can represent the predicate in disjunctive normal form: $\Phi = \bigvee_{i=1}^m \bigwedge_{j=1}^n \varphi_{i,j}$ where $\varphi_{i,j} = \sum_{k=1}^s \beta_k * p_k \odot c$. Here, each atomic predicate $\varphi_{i,j}$ is represented by a linear equation over properties of nodes and edges in the pattern. In particular, $p_1, \dots, p_s$ are properties. $\beta_1, \dots, \beta_s$ and c are integer variables, and $\odot$ is an operator in $\{=, \neq, \leq, \geq, <, >\}$.

Then our goal is to find the coefficients β_k , constant c , and operator $\odot$ such that the corresponding predicate satisfies the requirements.³ Let us illustrate using the example in Figure 8. From the requirement $b_1 = \top$, we know the filter should retain the first subgraph. Therefore, the predicate must evaluate to true for this subgraph. The values of its relevant properties are provided in Figure 8, so the formula $\beta_1 * 40 + \beta_2 * 3 + \beta_3 * 120 + \beta_4 * 120 \odot c$ must hold. Similarly, we can identify formulas for the other four subgraphs and obtain a system of constraints. Furthermore, to ensure the Filter clause is concise, we design an objective function based on heuristics that minimizes the number of disjunctions and conjunctions in the predicate as well as ensures each atomic predicate to be concise. The intuition is to de-prioritize predicates such as $97 * n_3.Capacity - 17 * n_4.Capacity > -2$, because it correlates two capacities with a complicated inequality.

For our example, a solution to the optimization problem is $\beta_1 = 1, \beta_2 = 0, \beta_3 = 0, \beta_4 = 0, \odot = \geq$, and $c = 40$, which represents the predicate $n_2.Age \geq 40$. Therefore, our synthesized graph query is:

```
MATCH (n1:Dept)->[e1:Work]-(n2:Faculty)-[e2:Teach]->(n3:Course), (n2)-[e3:Taught]->(n4:Course)
WHERE n2.Age ≥ 40 RETURN n2.Age, n2.Title, e1.Years, n1.Name, max(n3.Capacity)/min(n4.Capacity)
```

³For simplicity, assume all properties hold integer values. We will explain how to handle strings in Section 5.4.

Program	Q	$::=$	$\text{Return}(C, L)$
Clause	C	$::=$	$\text{Match}(\pi) \mid \text{Filter}(C, \Phi)$
Pattern	π	$::=$	$(x : l) \mid (x : l) - [x : l] \rightarrow (x : l) \mid \pi, \pi$
Return List	L	$::=$	$r \mid r L$
Return	r	$::=$	$v \mid x.p \mid g(x.p) \mid r \oplus r$
Predicate	Φ	$::=$	$E \odot E \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg \Phi$
Expression	E	$::=$	$v \mid x.p \mid E \oplus E$

$\odot \in \{<, \leq, >, \geq, =, \neq\}$ $\oplus \in \{+, -, *, /\}$ $g \in \{\text{Max, Min, Avg, Count, Sum}\}$
 $x \in \mathbf{Variables}$ $l \in \mathbf{Labels}$ $p \in \mathbf{Property Keys}$ $v \in \mathbf{Constants}$

Fig. 9. Syntax of the graph query language.

3 Preliminaries

In this section, we present preliminaries about graph databases and graph queries.

Graph databases. As is standard [19], we consider a graph database as a *property graph* $G = (N, E, L, K, V)$ where

- N is a set of nodes;
- $E \subseteq N \times N$ is a set of edges;
- $L : (N \cup E) \mapsto \mathbf{Labels}$ is a mapping that maps each node and edge to a label;
- K is a set of property keys⁴;
- $V : (N \cup E) \times K \mapsto \mathbf{Values}$ maps each property key of all nodes and edges to a value.

Patterns. A pattern π in the graph can be defined inductively based on the following grammar

$$\pi ::= (x : l) \mid (x_1 : l_1) - [x_2 : l_2] \rightarrow (x_3 : l_3) \mid \pi, \pi$$

Here, $(x : l)$ is a *node pattern* that can match a node with label l . $[x : l]$ is an *edge pattern* that matches an edge with label l . $(x_1 : l_1) - [x_2 : l_2] \rightarrow (x_3 : l_3)$ is a *path pattern* that matches a simple path from node x_1 with label l_1 to node x_3 with label l_3 through edge x_2 with label l_2 . Two patterns π_1 and π_2 can be composed together to get a more complex pattern π_1, π_2 .

Graph query language. In this paper, we focus on a core fragment of the Cypher language [19] for graph queries. As shown in Figure 9, a graph query mainly consists of three parts: pattern matching with Match, subgraph filtering with Filter, and projection and grouping with Return. At the top level, a query Q projects or aggregates a clause C with a list of return expressions L , where each return expression r can be a constant value v , property of a variable $x.p$, an aggregation over a property $g(x.p)$, or arithmetics over two return expressions. The clause C can be a Match clause with a pattern π or a Filter clause with predicate Φ . We denote the semantics of query Q by $\llbracket Q \rrbracket$ and the execution result of Q on graph G by $\llbracket Q \rrbracket_G$, which is formally described in the Appendix.

4 Problem Formulation

User demonstration. A *user demonstration* D is a pair (G, T) , where G is the input property graph and T is a non-empty demonstration table. Here, the concept of demonstration table is inspired by prior work [62]. Specifically, a demonstration table T is a list of rows, and each row is a list of cells. Each cell is an expression that describes how to compute the corresponding value. For simplicity, we refer to a demonstration table just as a *table* if clear from the context.

Row and table equivalence. Following prior work on the equivalence definition of tables [21], we can define the equivalence of two demonstration tables as follows.

⁴All property keys are assumed to be globally unique. This assumption can be easily fulfilled by prepending the property key with the label of its corresponding node or edge.

Algorithm 1 Top-level synthesis algorithm for graph queries

```

1: procedure SYNTHESIZE( $\mathcal{D}$ )
   Input: A set of user demonstrations  $\mathcal{D}$ 
   Output: A graph query  $Q$  satisfying  $\mathcal{D}$ , or  $\perp$  to indicate failure
2:   for each  $\pi \in \text{GRAPHMINING}(\mathcal{D})$  do
3:      $S \leftarrow \text{GENERATESKETCH}(\pi, \mathcal{D})$ 
4:      $Q \leftarrow \text{COMPLETESKETCH}(S, \mathcal{D})$ 
5:     if  $Q \neq \perp$  then return  $Q$ 
6:   return  $\perp$ 

```

Definition 4.1 (Row equivalence). Given two rows r_1, r_2 of the same length l and a bijective mapping δ between their indices, r_1 and r_2 are said to be *equivalent* modulo δ , denoted by $r_1 \equiv_\delta r_2$, if any pair of expressions at corresponding indices are equivalent, i.e.,

$$r_1 \equiv_\delta r_2 \Leftrightarrow \forall i. 0 \leq i < l \rightarrow r_1[i] \equiv r_2[\delta(i)]$$

Definition 4.2 (Table equivalence). Two demonstration tables T_1 and T_2 are said to be *equivalent*, denoted by $T_1 \equiv T_2$, if there exists a bijective mapping δ between row indices and a bijective mapping σ from rows of T_1 to those of T_2 , such that for any row r_1 in T_1 , there exists a row $r_2 = \sigma(r_1)$ in T_2 and r_1 is equivalent to r_2 modulo δ , i.e.,

$$T_1 \equiv T_2 \Leftrightarrow \exists \sigma. \exists \delta. \forall r_1 \in \text{Rows}(T_1). \exists r_2 \in \text{Rows}(T_2). r_2 = \sigma(r_1) \wedge r_1 \equiv_\delta r_2$$

Problem statement. Given n demonstrations $D_i = (G_i, T_i)$ where $i \in \{1, \dots, n\}$, our goal is to synthesize a graph query Q such that Q satisfies all the demonstrations, i.e.,

$$\forall i. 1 \leq i \leq n \rightarrow \llbracket Q \rrbracket_{G_i} \equiv T_i$$

5 Synthesis Approach

In this section, we explain our approach for synthesizing graph queries from demonstrations.

5.1 Top-Level Algorithm

Our synthesis algorithm aims to generate a graph query Q that is consistent with a set of user demonstrations $\mathcal{D}$, where each demonstration $D = (G, T) \in \mathcal{D}$ specifies how an input graph G becomes the output table T . At a high level, the algorithm integrates *pattern discovery through graph mining* and *enumerative search guided by deduction-based pruning* to efficiently explore the space of candidate graph queries. As shown in Algorithm 1, the SYNTHESIZE procedure first performs graph mining to identify candidate patterns that are common across all input graphs. These mined patterns capture the subgraph structures that are likely relevant to the user's intended query. For each candidate pattern π , the algorithm uses π as the backbone to construct a query sketch S . The sketch essentially includes a Match clause instantiated with π , while leaving some details about Filter and Return clauses unspecified. To navigate the large search space of possible completions of S , the algorithm employs deductive reasoning to generalize observed inconsistencies: when a candidate query is found to be incorrect, it can infer multiple incorrect queries and prune the search space accordingly. This iterative process continues until a correct query Q is synthesized, or if no such query exists for any candidate pattern, the algorithm returns $\perp$ to indicate failure.

5.2 Pattern Discovery by Graph Mining

Since the core of sketch generation is identifying patterns from input graphs, we next describe how to synthesize these patterns before discussing how to generate the query sketch.

Algorithm 2 Mining patterns from input graphs

```

1: procedure GRAPHMINING( $\mathcal{D}$ )
   Input: A set of user demonstrations  $\mathcal{D}$ 
   Output: A set of patterns  $\Pi$ 
2:    $\Pi \leftarrow \emptyset$ 
3:   for each  $G \in \text{FINDCANDIDATEMATCHES}(\mathcal{D})$  do
4:     for each  $\pi \in \text{RECOVERPATTERNS}(G)$  do
5:       if  $\text{IsConsistent}(\pi, \mathcal{D})$  then
6:          $\Pi \leftarrow \Pi \cup \{\pi\}$ 
7:   return  $\Pi$ 

```

A straightforward approach to synthesize patterns is to enumerate all possible graph patterns and check them against the demonstrations. However, this brute-force strategy quickly becomes infeasible because the number of possible patterns grows exponentially with the pattern size. Moreover, most enumerated patterns have no meaningful correspondence with the given demonstrations: they either do not match the input graphs in the demonstrations or do not cover the nodes and edges that appear in the output tables, and thus will not produce the user-specified results. This observation motivates our use of graph mining. Instead of searching over arbitrary patterns, we derive candidate patterns directly from the input graphs: we first mine relevant subgraphs from the demonstrations and then recover patterns from those subgraphs. Our algorithm focuses on patterns that are common across the input graphs and whose matched subgraphs cover the nodes and edges occurred in the demonstration outputs. In this way, graph mining helps the synthesizer focus on meaningful candidate patterns while pruning many irrelevant ones.

As shown in Algorithm 2, our graph mining process consists of two stages: (1) finding candidate matches from demonstrations $\mathcal{D}$ – identifying subgraphs that likely correspond to meaningful patterns by considering the commonality and output coverage constraints; (2) recovering patterns from each candidate match G – transforming G into patterns that can produce subgraphs including G itself when the patterns are applied to match the input graph. For each recovered pattern π , we need to check whether π is consistent with all demonstrations. Specifically, for each demonstration $(G, T) \in \mathcal{D}$, the IsConsistent procedure checks whether π matches G , and if so, whether the nodes and edges in the match are a superset of the nodes and edges occurred in the demonstration table. If π is not consistent with all demonstrations, then no query containing π can be correct. This is because the subsequent Filter and Return clauses further remove and project out nodes and edges that match the pattern π , resulting in even fewer nodes and edges. Consequently, the query cannot be evaluated to tables equivalent to the demonstration output tables. If π is consistent with all demonstrations, we add it to the return list Π as a candidate pattern.

Necessary conditions on candidate matches. While it is difficult to infer which subgraphs correspond to the desired query, we can identify necessary conditions that such subgraphs must satisfy. Specifically, subgraphs that do not satisfy the following two constraints will not correspond to the desired query. We refer to subgraphs satisfying these constraints as candidate matches.⁵

- *Commonality constraint.* The subgraph should only include nodes and edges whose labels are shared among all input graphs, and its size should not exceed the largest weakly connected component of any input graph.
- *Output coverage constraint.* For each label that presents in the demonstration table, the subgraph should include at least one node or edge with the same label.

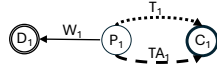
⁵Precise definitions of these constraints are available in Appendix G of the extended version [33]. Proofs of why these constraints are necessary conditions are available in Appendix G.

Algorithm 3 Algorithm to mine candidate matches

```

1: procedure FINDCANDIDATEMATCHES( $\mathcal{D}$ )
   Input: Demonstrations  $\mathcal{D} = \{(G_j, T_j) \mid j = 1, \dots, n\}$ 
   Output: Candidate matches from input graphs of  $\mathcal{D}$ 
2:    $\mathcal{G} \leftarrow \emptyset$ ;  $\mathcal{V} \leftarrow \emptyset$ ;
3:    $\mathcal{M} = \{j \mapsto \text{WCCs}(\text{RemoveUncommonLabel}(G_j, \mathcal{D})) \mid j = 1, \dots, n\}$ 
4:    $\mathcal{W} \leftarrow \mathcal{M}[k^*]$  where  $k^* = \text{argmin}_k \{\text{MaxSize}(\mathcal{M}[k]) \mid k = 1, \dots, n\}$ 
5:   while  $\neg \text{IsEmpty}(\mathcal{W})$  do
6:      $G \leftarrow \text{GetNext}(\mathcal{W})$ 
7:     if  $\text{OutputCoverage}(G, \mathcal{D})$  then
8:        $\mathcal{G} \leftarrow \mathcal{G} \cup \{G\}$ 
9:       for each  $e \in \text{Edges}(G)$  do
10:        for  $G' \in \text{RemoveEdge}(G, e)$  do
11:          if  $G' \notin \mathcal{V}$  then
12:             $\mathcal{W} \leftarrow \mathcal{W} \cup \{G'\}$ 
13:             $\mathcal{V} \leftarrow \mathcal{V} \cup \{G'\}$ 
14:   return  $\mathcal{G}$ 

```



(a) An example graph.

(b) Subgraphs by removing edge W_1 .

Fig. 10. Generating subgraphs from an example graph.

Finding candidate matches. Based on these constraints, Algorithm 3 presents our procedure for identifying candidate matches. The algorithm takes as input the set of user demonstrations $\mathcal{D}$ and outputs a set of candidate matches $\mathcal{G}$. We begin by removing any node or edge whose label is not shared across all input graphs, and then compute the weakly connected components (WCCs) of the obtained graphs. As shown in Line 3, this step produces a map $\mathcal{M}$ that associates the index of each input graph with its corresponding group of WCCs. Intuitively, within each group of WCCs in $\mathcal{M}$, there must exist subgraphs matching the desired pattern; otherwise the output table would be empty. While we can start with any group of WCCs to find candidate matches, we select the group of WCCs that requires the least number of subgraph enumeration for efficiency. In particular, we initialize the worklist $\mathcal{W}$ with the group of WCCs whose largest component size is the smallest among all input graphs (Line 4). After this step, every graph (i.e., weakly connected component) in $\mathcal{W}$ satisfies the *commonality constraint* by definition. For each graph G in $\mathcal{W}$, we check whether it satisfies the *output coverage constraint*, discarding those that violate the constraint (Line 7). If G satisfies the constraint, we add it to the result set $\mathcal{G}$ as a candidate match. We then enumerate subgraphs of G using an incremental edge-deletion process (Line 9-13). Rather than generating all subgraphs of G at once, we remove one edge at a time to obtain a portion of the subgraph of G . This incremental strategy enables early pruning and avoids materializing a large number of candidates upfront. Since removing an edge may disconnect the graph, we decompose the resulting graph into its connected components. Each connected component that has not been enumerated before is added to $\mathcal{W}$ and recorded in $\mathcal{V}$. The algorithm repeats this process until $\mathcal{W}$ becomes empty. Consequently, the procedure explores all candidate matches derived from at least one input graph, including those needed to recover the desired query pattern.

Example 5.1. Consider the graph in Figure 10a, which satisfies the commonality and output-coverage constraints. In one round of subgraph generation, as described in Line 9 of Algorithm 3,

Algorithm 4 Algorithm to recover patterns from a candidate match

```

1: procedure RECOVERPATTERNS( $G$ )
   Input: A candidate match  $G$ 
   Output: A set of patterns that can match  $G$ 
2:   for each  $\tau_i \in \text{SpanningTrees}(G)$  do
3:      $\pi_i \leftarrow \epsilon$ 
4:     for each  $e \in \text{Edges}(\tau_i)$  do
5:        $s, t \leftarrow \text{IncidentNodes}(e)$ 
6:        $\pi_i \leftarrow \pi_i, (\text{NVar}(s) : \text{Label}(s)) - [\text{EVar}(e) : \text{Label}(e)] \rightarrow (\text{NVar}(t) : \text{Label}(t))$ 
7:      $\Pi_i \leftarrow \{\pi_i\}$ 
8:     for each  $e \in \text{Edges}(G) \setminus \text{Edges}(\tau_i)$  do
9:        $\Pi_i \leftarrow \bigcup_{\pi \in \Pi_i} \text{EXTENDPATTERN}(\pi, e)$ 
10:  return  $\bigcup_i \Pi_i$ 
11: procedure EXTENDPATTERN( $\pi, e$ )
   Input: A pattern  $\pi$ , an additional edge  $e$  to add
   Output: All patterns extending  $\pi$  that match  $e$  and its incident nodes
12:   $s, t \leftarrow \text{IncidentNodes}(e); \Pi \leftarrow \emptyset$ 
13:   $S \leftarrow \text{NodePatterns}(s); T \leftarrow \text{NodePatterns}(t)$ 
14:   $N_s \leftarrow (\text{FreshNVar}(s) : \text{Label}(s)); N_t \leftarrow (\text{FreshNVar}(t) : \text{Label}(t))$ 
15:  for each  $(\pi_s, \pi_t) \in (S \times T) \cup (S \times \{N_t\}) \cup (\{N_s\} \times T)$  do
16:     $\Pi \leftarrow \Pi \cup \{\pi, \pi_s - [\text{EVar}(e) : \text{Label}(e)] \rightarrow \pi_t\}$ 
17:  return  $\Pi$ 

```

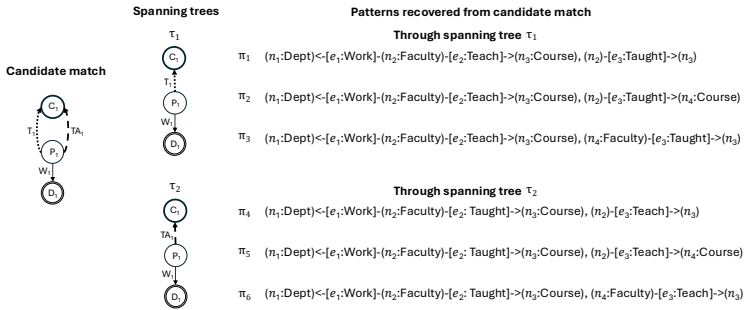


Fig. 11. A candidate match, its spanning trees, and patterns recovered from the candidate match.

we remove one of the edges W_1 , T_1 , or TA_1 . For example, removing W_1 produces an edge-deletion subgraph G' (Figure 10b) containing two disconnected components. We then check whether each component has been generated previously and add any newly discovered component to the worklist.

Recovering patterns from a candidate match. The goal of pattern recovery is to generate all patterns that can match a given graph G . Our key insight is that when G is acyclic,⁶ the pattern's structure mirrors G . However, when G contains cycles, the subtlety arises: while multiple edge-pattern variables with the same label must correspond to distinct edges, multiple node-pattern variables may legitimately match the same node in G . Consequently, a single graph may correspond to multiple valid patterns, because it is ambiguous whether a particular graph node should be represented by one node-pattern variable or by several different ones. For example, there are four

⁶Here, we do not consider directions of edges for cycles.

possible ways to construct a pattern that matches the first graph in Figure 11: (1) A single node variable n_2 matches graph node P_1 , and a single node variable n_3 matches graph node C_1 . This case corresponds to π_1 and π_4 , which represent the same pattern. (2) A single node variable n_2 matches graph node P_1 , while two distinct node variables n_3 and n_4 both match graph node C_1 . This case corresponds to π_2 and π_5 , which represent the same pattern. (3) Two distinct node variables n_2 and n_4 both match graph node P_1 , while a single node variable n_3 matches graph node C_1 . This case corresponds to π_3 and π_6 , which represent two different patterns. These different variable-to-node mappings lead to different patterns that can match the original graph.

Our procedure in Algorithm 4 begins by finding all spanning trees of G . Since a spanning tree contains no cycles, we can extract a pattern π_i from each spanning tree τ_i with a one-to-one correspondence between nodes and edges and pattern variables (Lines 3 - 6). When G contains cycles, some edges appear in G but not in τ_i . We process these complementary edges one by one, and extend all patterns we have obtained so far iteratively (Lines 7 - 9). Specifically, for each pattern $\pi \in \Pi_i$ and each complementary edge e , we invoke the EXTENDPATTERN procedure to extend π with a new edge pattern (and possibly a new node pattern) that can match e . To consider all ways in which the incident nodes s, t of e can be matched, we first find all existing node patterns of s and t (Line 13), denoted by S and T , respectively. We also introduce two new node patterns N_s, N_t with fresh pattern variables (Line 14). Each extension of π must ensure the new edge pattern: (a) connects an existing node pattern from S to one in T , (b) connects an existing node pattern from S to the new N_t , or (c) connects the new N_s to an existing node pattern from T . We consider all such possibilities to get the full set of extended patterns in Π (Lines 15 - 16). Finally, all recovered patterns are obtained by taking the union of Π_i for all spanning trees τ_i (Line 10).

Example 5.2. Consider the first graph G in Figure 11, which has two spanning trees: τ_1 with edges W_1 and T_1 , and τ_2 with edges W_1 and TA_1 . We first take τ_1 as an example. Following RECOVERPATTERNS, we transform each edge in τ_1 together with its incident nodes into corresponding edge and node patterns, yielding $(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] \text{--} (n_2: \text{Faculty}) \text{--} [e_2: \text{Teach}] \rightarrow (n_3: \text{Course})$. Since the complementary edge TA_1 is not included in τ_1 , we extend this pattern to enumerate all patterns that can match TA_1 . This gives us three patterns: π_1 reuses both existing incident node variables by adding $[e_3: \text{Taught}]$ from n_2 to n_3 ; π_2 introduces a new Course node variable n_4 as the end of e_3 ; and π_3 introduces a new Faculty node variable n_4 as the start of e_3 . Similarly, for τ_2 , we first recover the pattern $(n_1: \text{Dept}) \leftarrow [e_1: \text{Work}] \text{--} (n_2: \text{Faculty}) \text{--} [e_2: \text{Taught}] \rightarrow (n_3: \text{Course})$. Because the complementary edge T_1 is not included in τ_2 , we apply the same extension procedure to match T_1 , producing three patterns π_4, π_5 , and π_6 . These correspond respectively to reusing both existing incident nodes, introducing a new Course node as the end of the edge, and introducing a new Faculty node as the start of the edge. Among them, π_4 is equivalent to π_1 , and π_5 is equivalent to π_2 . Thus, we obtain four different patterns matching G shown in Figure 11.

5.3 Sketch Generation

Generating the query sketch from a pattern π is straightforward. Recall that a graph query consists of Match, Filter, and Return clauses. The generated sketch mirrors this structure: the Match clause is populated with the pattern π , the Filter clause is left as a hole to be filled later, and the Return clause contains holes for return expressions following the demonstration table.

The key insight is that the demonstration table already specifies relevant properties and computations for each return expression. The only missing part is which variables to use in the expressions. Based on this observation, we can take any row from the demonstration table and replace the concrete nodes and edges with holes to construct the sketch of the Return clause. The domain of each hole ranges over all variables with the same label as the corresponding concrete node or edge.

Algorithm 5 Sketch completion algorithm

```

1: procedure COMPLETESKETCH( $S, \mathcal{D}$ )
   Input: Sketch  $S$ , demonstrations  $\mathcal{D} = \{(G_j, T_j) \mid j = 1, \dots, n\}$ 
   Output: A complete query  $Q$  or  $\perp$  to indicate failure
2:    $\mathcal{T} \leftarrow [T_j \mid (G_j, T_j) \in \mathcal{D}]$ 
3:    $\mathcal{R} \leftarrow [R_j \mid R_j = \llbracket \text{Match}(\pi) \rrbracket_{G_j}, \pi = \text{Pattern}(S), (G_j, T_j) \in \mathcal{D}]$ 
4:    $\phi \leftarrow \text{INITENCODING}(S, \mathcal{R});$ 
5:   while SAT( $\phi$ ) do
6:      $(\mu, \zeta) \leftarrow \text{GetModel}(\phi)$ 
7:      $\mathcal{T}' \leftarrow \text{Evaluate}(\mathcal{R}, \mu, \zeta)$ 
8:     if  $\forall i. \mathcal{T}'[i] \equiv \mathcal{T}[i]$  then
9:        $\Phi \leftarrow \text{SYNTHESIZEFILTER}(\zeta, \mathcal{R})$ 
10:      if  $\Phi \neq \perp$  then return InstantiateSketch( $S, \mu, \Phi$ )
11:       $\phi \leftarrow \phi \wedge \neg(\mu \wedge \zeta)$ 
12:    else
13:       $\phi \leftarrow \phi \wedge \text{DEDUCTIVEREASONING}(\mu, \zeta, \mathcal{T}, \mathcal{T}')$ 
14:  return  $\perp$ 

```

Example 5.3. Consider again the example in Figure 4 and the pattern obtained by graph mining: $(n_1:\text{Dept}) \leftarrow [e_1:\text{Work}] - (n_2:\text{Faculty}) - [e_2:\text{Teach}] \rightarrow (n_3:\text{Course}), (n_2) - [e_3:\text{Taught}] \rightarrow (n_4:\text{Course})$. The query sketch is shown in Eq (1) of Section 2.

5.4 Sketch Completion

The goal of sketch completion is to fill in the missing parts of the Filter and Return clauses to obtain a complete graph query. A key challenge lies in the fact that many different filtering predicates can retain the same subset of matched subgraphs, leading to redundant exploration during synthesis. To address this, our algorithm organizes possible predicates into equivalence classes based on their behavior over the matched subgraphs and completes the sketch in two alternative phases: (1) complete the Return clause and find the requirements (i.e., which subgraphs should be retained) for the filtering predicate; (2) synthesize a concise predicate from the filtering requirements.

Sketch completion algorithm. At a high level, our approach is inspired by *enumerative search with deductive reasoning* in program synthesis [17, 18, 52, 53], which we adapt to graph query synthesis from demonstrations. As shown in Algorithm 5, given a sketch S and demonstrations $\mathcal{D}$, we first collect the expected output tables into $\mathcal{T}$ (Line 2), evaluate the Match clause of S on each input graph, and store the resulting intermediate tables in $\mathcal{R}$ (Line 3). Each R_i contains the subgraphs of G_i that match the pattern in S . We then perform symbolic enumerative search under the constraint ϕ , which encodes all possible hole instantiations. Whenever ϕ is satisfiable, we obtain a model (μ, ζ) , where μ assigns the holes in the Return clause, and ζ specifies which matched subgraphs should be retained by the Filter clause. Using this model, we compute the actual output tables (Line 7). If the computed tables match the expected tables for all demonstrations, the Return clause is correct and the filtering requirements may be feasible. We then invoke SYNTHESIZEFILTER to synthesize a concise predicate Φ from ζ (Line 9). If successful, instantiating the sketch with μ and Φ yields a correct query. Otherwise, we block the current model by adding $\neg(\mu \wedge \zeta)$ to ϕ and continue the search (Line 11). If the computed tables differ from the expected ones, we analyze the mismatch and use deductive reasoning to rule out multiple incorrect completions at once (Line 13). The process repeats until a correct query is found or ϕ becomes unsatisfiable.

$$\frac{\text{Base}(e_{j,k}) \not\subseteq \bigcup_{\mathcal{F} \in \text{AtomicExprs}(T_i)} \text{Base}(\mathcal{F})}{\mu, \zeta \vdash e_{j,k}, T_i \rightsquigarrow \neg(y_{i,j} \wedge \bigvee_{x_p \in \text{dom}(\mu)} x_p = \mu[x_k])} \text{ (Elem)}$$

Fig. 12. Example deductive reasoning rule for generating blocking clauses. Here, $\text{Base}(e)$ returns all the nodes and edges in the atomic expression e . $\text{AtomicExprs}(T)$ collects all the atomic expressions in table T .

	$x_1 = \Pi_2$	$x_2 = \Pi_2$	$x_3 = \Pi_1$	$x_4 = \Pi_1$	$x_5 = \Pi_3$	$x_6 = \Pi_4$
Input Graph	? ₁ Age	? ₂ Title	? ₃ Years	? ₄ Name	max(75.Cap) / min(75.Cap)	
$y_{2,1} = \top$	2	P ₃ .Age	P ₃ .Title	W ₂ .Years	D ₂ .Name	max(C ₂ .Capacity) / min(C ₂ .Capacity)
$y_{2,2} = \top$	2	P ₄ .Age	P ₄ .Title	W ₄ .Years	D ₄ .Name	max(C ₄ .Capacity) / min(C ₄ .Capacity)
$y_{2,3} = \top$	2	P ₅ .Age	P ₅ .Title	W ₅ .Years	D ₅ .Name	max(C ₅ .Capacity) / min(C ₅ .Capacity)

(a) Actual demonstration table for demo 2.

P_4 .Age	P_4 .Title	W ₄ .Years	D ₄ .Name	max(C ₄ .Capacity) / min(C ₄ .Capacity)
P ₅ .Age	P ₅ .Title	W ₅ .Years	D ₅ .Name	max(C ₅ .Capacity) / min(C ₅ .Capacity)

(b) Expected demonstration table for demo 2.

Fig. 13. Actual and expected demonstration tables.

Initial encoding. Given a sketch S and all matched subgraphs $\mathcal{R}$, we introduce two sets of variables to encode the search space of sketch S .

- For the Filter clause, each requirement is represented by a boolean variable $y_{s,t}$, indicating whether the t -th subgraph in $\mathcal{R}_s$ should be retained.
- For the Return clause, each hole $?_j$ is represented by a variable x_j , where j is the index of the atomic expression in the return list. Since the hole $?_j$ can only be instantiated with expressions in its domain, our initial constraint for $?_j \in \{e_1, \dots, e_n\}$ is $x_j = e_1 \vee \dots \vee x_j = e_n$.

Evaluating models. Recall from Line 7 of Algorithm 5 that the sketch completion procedure needs to evaluate all matched subgraphs using a given model (μ, ζ) to produce the actual demonstration tables. This evaluation process is mostly standard. The only subtlety lies in the fact that the model does not directly correspond to a complete query. Instead, it only specifies the requirements of the Filter clause, not the exact filter predicate. However, this can be easily resolved by referencing ζ in the model, which indicates which subgraphs should be retained by the Filter clause. Furthermore, the Return clause can be derived based on μ and evaluated directly based on the semantics.

Deductive reasoning. When the actual demonstration tables differ from the expected ones, we use deductive reasoning to derive a blocking clause that eliminates many incorrect completions at once. This idea is inspired by prior work on concrete tables for Datalog programs [53], but we adapt it to demonstration tables for graph queries. For each mismatched table T'_i , we identify expressions that appear in T'_i but not in the expected table T_i , since such expressions indicate an error in the current sketch completion. Specifically, we inspect each expression $e_{j,k}$ in T'_i , where $e_{j,k}$ denotes the k -th expression in row j , and apply inference rules to generate a blocking clause. Figure 12 shows an example rule Elem,⁷ where judgments of the form $\mu, \zeta \vdash e_{j,k}, T_i \rightsquigarrow \phi$ denote that, under model (μ, ζ) , the generated blocking clause is ϕ . The Elem rule focuses on the case when nodes or edges in expression $e_{j,k}$ (i.e., $\text{Base}(e_{j,k})$) are not included in the expressions of the expected demonstration table T_i , and no grouping operation was involved. In this case, we enforce that the completion should not choose the same pattern variable $\mu[x_k]$ for any hole of the same type, if the subgraph j is retained by the filter.

Example 5.4. Consider again the sketch (1) in Section 2.1. Suppose the model is

$$\begin{aligned} \mu &= [x_1 \mapsto n_2, x_2 \mapsto n_2, x_3 \mapsto e_1, x_4 \mapsto n_1, x_5 \mapsto n_3, x_6 \mapsto n_4] \\ \zeta &= [y_{1,1} \mapsto \top, y_{1,2} \mapsto \top, y_{2,1} \mapsto \top, y_{2,2} \mapsto \top, y_{2,3} \mapsto \top] \end{aligned}$$

Let T'_2 be the actual demonstration table corresponding to this model in demonstration 2, and let

⁷The complete list of rules is provided in Appendix F.

T_2 be the expected table (as shown in Figure 13). We now explain how to obtain the blocking clause. Here, the expression $P_3.\text{Age}$ in the first row of T'_2 does not occur in the expected table T_2 , which is evidence of why the corresponding sketch completion is incorrect. Furthermore, as long as the first subgraph is retained by the filter (i.e., $\zeta[y_{2,1}] = \top$), P_3 should not occur anywhere in the first row of T'_2 . To enforce it, we figure out the pattern variable n_2 matches the node P_3 in this subgraph. Thus, n_2 cannot be used to fill any hole in the return clause in this case. Since there are two holes $?_1, ?_2$ that include n_2 in their domain, the blocking clause is $\neg((x_1 = n_2 \vee x_2 = n_2) \wedge y_{2,1} = \top)$. It basically states that if the first subgraph is retained by the filter, $?_1$ or $?_2$ cannot be filled with n_2 .

Synthesizing filter from requirements. After obtaining a model corresponding to a candidate query, the final step is to synthesize a Filter clause. While the model indicates the filtering requirements (i.e., which subgraphs should be retained), it does not directly provide the filter predicate. The SYNTHESIZEFILTER procedure takes these requirements and subgraphs as input and produces a predicate satisfying the requirements. The main challenge here is to ensure the synthesized predicate not only adheres to all requirements but also generalizes to unseen cases. We address this by formulating predicate synthesis as a constraint optimization problem, where the constraints ensure satisfying filter requirements, and the objective function improves conciseness and generalizability.

Without loss of generality, we consider the predicate in disjunctive normal form $\Phi = \bigvee_{i=1}^m \bigwedge_{j=1}^n \varphi_{i,j}$, where each atomic predicate $\varphi_{i,j}$ is on graph properties, in one of the following forms:

- $\sum_u \sum_v \beta_{u,v} * p_{u,v} \odot \text{cnum}$. Here, $\odot \in \{=, \neq, >, \geq, <, \leq\}$. $p_{u,v}$ is a numerical (integer) property with index v of pattern variable u . $\beta_{u,v}$ is its coefficient. cnum is an integer constant.
- $p_{u,v} \odot \text{cstr}$. Here, $\odot \in \{=, \neq\}$. $p_{u,v}$ is a string property with index v of pattern variable u . cstr is a string constant.
- $p_{u,v} \odot p_{u',v'}$. Here, $\odot \in \{=, \neq\}$. $p_{u,v}$ is a string property with index v of pattern variable u , and $p_{u',v'}$ is a string property with index v' of pattern variable u' .

Since the filtering requirements specify which subgraphs are retained and which are removed, we can directly substitute the values of these subgraphs into the above predicate template to derive constraints on the coefficients and constants. Specifically, consider the s -th demonstration and the t -th subgraph. Recall from Section 3 that $V_{s,t}$ maps a node or edge and its property key to the corresponding value. If the requirement specifies $\zeta[y_{s,t}] = \top$, i.e., the t -th subgraph of demo s should be retained, the corresponding constraint is $F(s, t) : \bigvee_{i=1}^m \bigwedge_{j=1}^n \varphi_{i,j}[V_{s,t}(u, p_{u,v}) / p_{u,v}]$. Otherwise, if the requirement specifies $\zeta[y_{s,t}] = \perp$, the constraint is $\neg F(s, t)$. Putting all filtering requirements together, the complete constraint is

$$\left(\bigwedge_{(s,t) \in \{(s,t) \mid \zeta[y_{s,t}] = \top\}} F(s, t) \right) \bigwedge \left(\bigwedge_{(s,t) \in \{(s,t) \mid \zeta[y_{s,t}] = \perp\}} \neg F(s, t) \right)$$

To promote concise predicates, our approach is guided by an objective function that provides heuristic measures of conciseness. In practice, we find it is helpful to optimize for three key aspects.

- Minimize the number of disjunctions in predicate Φ , denoted by m . This avoids overfitting to individual subgraphs.
- Minimize the number of coefficients that are not 0, 1, or -1, denoted by C . This helps avoid predicates with counter-intuitive constants that are unlikely to reflect the user's intent.
- Minimize the total number of conjuncts in Φ , denoted by N . This enhances simplicity and conciseness of predicate Φ .

Our objective function for constraint optimization is to minimize $w_1 \cdot m + w_2 \cdot C + w_3 \cdot N$, where w_1, w_2, w_3 are hyperparameters for weights of the above three aspects. In practice, the weights are chosen to be sufficiently separated to enforce the strict priority order among the three aspects, i.e., $w_1 \gg w_2 \gg w_3$.

5.5 Theorems

THEOREM 5.5 (SOUNDNESS). ⁸ Let $\mathcal{D}$ be a set of demonstrations. Suppose $\text{SYNTHESIZE}(\mathcal{D})$ returns a graph query Q , then for each demonstration $(G, T) \in \mathcal{D}$, it holds that $\llbracket Q \rrbracket_G \equiv T$.

THEOREM 5.6 (COMPLETENESS). Let $\mathcal{D}$ be a set of demonstrations. If there exists a graph query Q accepted by grammar in Figure 9 such that $\llbracket Q \rrbracket_G \equiv T$ holds for each demonstration $(G, T) \in \mathcal{D}$, then $\text{SYNTHESIZE}(\mathcal{D})$ does not return $\perp$.

6 Evaluation

We have implemented our proposed techniques in a tool called DMINER and evaluated it with the goal of answering the following research questions:

- RQ1.** Is DMINER effective and efficient to solve Cypher synthesis tasks given user demonstrations?
- RQ2.** How beneficial are the necessary conditions for graph mining and deductive reasoning for sketch completion in DMINER?
- RQ3.** How does DMINER perform compared to existing approaches such as enumerative search and Large Language Models (LLMs)?
- RQ4.** How much effort does the user need to provide demonstrations? What is the quality of the provided demonstrations?

Benchmarks. To evaluate DMINER, we identified 115 candidate tasks from various sources. We filtered out 25 tasks that require features currently outside the supported fragment of DMINER, such as multiple Match clauses, variable-length patterns, aggregation filtering, or subqueries. This leaves us with 90 benchmarks for evaluation, categorized into four sources:

- **StackOverflow (SOF).** We identified 35 benchmarks by searching for posts related to writing Cypher queries on StackOverflow. Each benchmark, including the demonstration and the ground-truth Cypher query, was constructed based on the descriptions and example input graphs provided in the posts.
- **Neo4jForum.** We collected 7 benchmarks from the Neo4j online forum. These benchmarks are constructed with a similar approach as StackOverflow benchmarks.
- **Neo4jTutorial.** We also collected 16 benchmarks from the Neo4j online tutorials, each of which includes at least one input graph along with a ground-truth query from the tutorial. The demonstration tables were constructed based on the ground-truth queries.
- **GraphDatasets.** We curated 32 benchmarks based on practical use scenarios using datasets provided in the Neo4j's official documentation and real-world data about movie production.

Each benchmark contains 1 to 4 demonstrations, depending on query complexity. For complex tasks, we follow workflows observed in the user study (described in Section 6.4) to augment demonstrations from existing to curated input graphs. Specifically, users typically provide a simple input graph to express their intent, then add a few structurally diverse, more complex graphs to distinguish it from alternatives. This process only requires modest effort.

Statistics. Table 1 summarizes the statistics of our benchmark suite. To evaluate DMINER's performance on tasks of different difficulty, we categorize benchmarks as *complex* or *simple* based on the structure of their ground-truth queries. Specifically, we identify complex ground-truth queries based on the complexities of their Match, Filter, and Return clauses. A Match clause is complex if it contains patterns of size 5 or more, with at least 3 nodes and 2 edges. Similarly, a Filter clause is complex if it includes at least 2 logic connectives combining atomic predicates. Finally, a Return clause is complex if it contains at least 4 expressions. Based on these criteria, our benchmark

⁸The proofs of all theorems can be found in Appendix G.

Table 1. The statistics of benchmarks. # denotes the number of benchmarks. #cmplx denotes the number of benchmarks that are complex. #demo denotes the number of demonstrations in each benchmark.

Source	#	Match			Filter			Return			#cmplx	#demo		InputGraph		DemoTable
		avg	max	#c	avg	max	#c	avg	max	#c		avg	max	n	e	
SOF	35	3.23	5	12	1.03	8	4	2.11	4	2	16	1.91	4	8.00	4.47	3.88
Forum	7	3.29	5	2	1.00	2	0	1.29	2	0	2	2.14	3	8.2	4.47	3.13
Tutorial	16	2.13	5	1	1.25	6	1	1.63	5	1	2	2.13	4	5.15	2.50	3.41
Dataset	32	5.81	11	19	1.06	6	6	3.03	9	11	24	2.16	4	8.68	6.01	4.20
Total	90	3.96	11	34	1.08	8	11	2.29	9	14	44	2.06	4	7.75	4.69	3.85

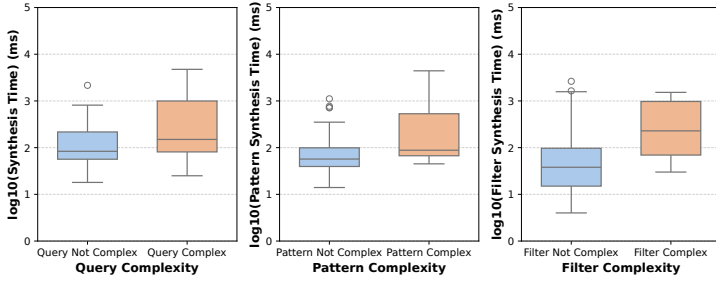


Fig. 14. Main experimental results. X-axis represents complexity of the tasks, y-axis represents the logarithm of synthesis time on base 10 in milliseconds.

suite includes 34 benchmarks with complex Match clauses, 11 benchmarks with complex Filter clauses, and 14 benchmarks with complex Return clauses. A benchmark is complex if it contains at least one complex clause. Overall, there are 44 complex benchmarks in total. These complex queries are distributed across all four sources. We also collected statistics about the complexity of queries in each source. As shown in Table 1, |Match|, |Return|, and |Filter| indicate the size of the pattern in the Match clause, the number of atomic predicates in the Filter clause, and the number of atomic expressions in the Return clause, respectively. In addition, avg denotes the average size, max denotes the maximum size, and #c denotes the number of complex benchmarks.

The average number of demonstrations in each benchmark is 2.06 (max 4). We also summarized the size of user demonstrations. The |InputGraph| and |DemoTable| columns in Table 1 indicate the average number of nodes (denoted by |n|) and edges (denoted by |e|) in the input graphs and the average number of expressions in the demonstration tables. On average, there are about 8 nodes and 5 edges in input graphs across all tasks, and about 4 expressions in the demonstration table.

Experiment setup. We set the timeout of running each benchmark to be 10 minutes. All of the experiments are conducted on a device with an Apple M2 Pro chip and 16GB physical memory running the macOS 13.3 operating system.

6.1 Performance of DMINER

To evaluate DMINER, we ran it on all 90 benchmarks. For each benchmark, we recorded DMINER’s synthesis time and manually checked whether the synthesized query is semantically equivalent to the desired query. The evaluation shows that DMINER is able to synthesize the desired query for 87 out of 90 benchmarks within the timeout of 10 minutes, and the average synthesis time is 0.55 seconds. As shown in the first box plot in Figure 14, 98% of the simple benchmarks can be solved within 1 second. The complex benchmarks take longer time to solve, and around 75% of them can be solved within 1 second. To better understand the difference in synthesis time for

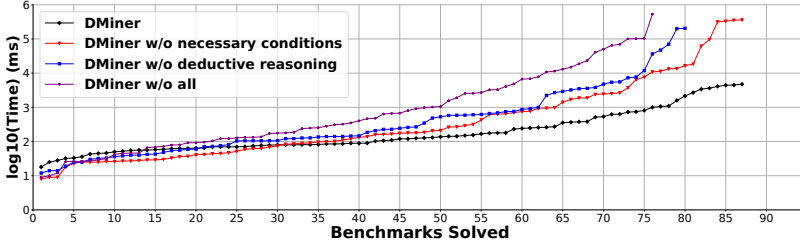


Fig. 15. Ablation study of DMINER. Synthesis time is on a scale of $\log_{10}$ by milliseconds.

complex and simple tasks, we present the second and third box plots in Figure 14, which show the time distribution of pattern synthesis and filter synthesis, respectively. When the pattern is complex, the interquartile range is shifted upward, indicating longer synthesis time compared to the simple pattern case. As for filter synthesis, the time distribution⁹ presents a higher variance, with several very fast cases for simple filters. Overall, the filter synthesis time for complex tasks is also longer than that for simple tasks.

Failure analysis. For the remaining 3 out of 90 benchmarks, DMINER was only able to synthesize plausible queries that are consistent with the demonstrations, but not equivalent to the desired queries. We have manually inspected these benchmarks and analyzed the reason for failure. We found that the ground-truth queries for all these benchmarks contain very complex filtering predicates (with 6, 6, and 8 atomic predicates, respectively). Our synthesized predicates were overfit to the provided demonstrations. To further investigate DMINER’s capability of solving these benchmarks, we added more demonstrations and re-ran the tool. DMINER managed to synthesize the desired queries with 6, 7, and 13 demonstrations in total.

Answer to RQ1. DMINER successfully synthesizes the desired query for 87 of the 90 benchmarks, with an average synthesis time of 0.55 seconds. The 3 failures were all caused by insufficient demonstrations. These results show that DMINER is both effective and efficient at synthesizing graph queries from user demonstrations.

6.2 Effectiveness of Subgraph Pruning and Deductive Reasoning Techniques

Recall from Section 5.2 that we introduce the commonality and output-coverage constraints as necessary conditions for graph mining and from Section 5.4 that we perform deductive reasoning for efficient sketch completion. To understand the effectiveness of these techniques, we conduct an ablation study with the following three variants of DMINER. We evaluated these three variants with a 10-minute time budget.

- Variant 1 (V_1): DMINER without necessary-condition-based pruning for graph mining.
- Variant 2 (V_2): DMINER without deductive reasoning in sketch completion.
- Variant 3 (V_3): DMINER without the above two techniques.

As shown in Figure 15, the gaps between the ablated versions and DMINER are significant, which indicates both necessary conditions for graph mining and deductive reasoning for sketch completion are important to DMINER’s efficiency. Specifically, if pruning with necessary conditions is disabled (red line), DMINER can synthesize 87 desired queries, but with significantly longer time. If deductive reasoning is disabled (blue line), DMINER times out on 9 benchmarks and fails to synthesize the desired query on 10 benchmarks. If both optimizations are disabled (purple line), DMINER times out on 13 benchmarks and fails to synthesize the desired query on 14 benchmarks.

⁹In the box plot, filter synthesis time for benchmarks with no filter is not shown.

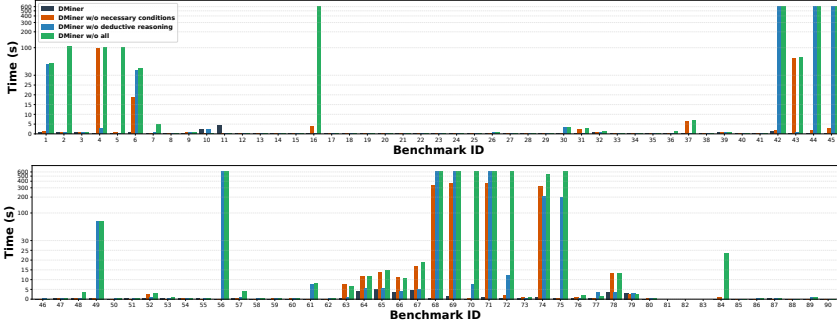


Fig. 16. Per benchmark ablation study result of DMINER. Synthesis time for tasks where DMINER did not synthesize desired queries are not shown (benchmarks 81, 82, and 83).

Figure 16 shows a bar chart for per-benchmark comparison, where the x-axis represents benchmark IDs, and the four bars for each benchmark show the time required by each variant to synthesize the desired query within the 10-minute timeout. As shown in the figure, there are 19 benchmarks where at least one ablated variant takes at least 30 seconds longer than the full version. In 14 benchmarks, the variant without pruning with necessary conditions is at least 5 seconds slower because graph mining wastes time on incorrect patterns generated from subgraphs not including the desired output information, or even not matching the input graph. In 15 benchmarks, the variant without deductive reasoning is at least 5 seconds slower because it can no longer rule out multiple incorrect completions at once, increasing the number of iterations for sketch completion.

Answer to RQ2. Pruning based on necessary conditions for graph mining and deductive reasoning for sketch completion both improve the efficiency of DMINER.

6.3 Comparison with Existing Techniques

Comparison with enumerative search. Since many existing enumerative synthesizers cannot be directly applied to graph query synthesis without substantial changes, such as developing abstract or inverse semantics for graph operators, we implemented an enumerative search baseline using the EUSOLVER framework [3], a general-purpose program synthesis tool for syntax-guided synthesis [2]. Our implementation supports all operators in the graph query language shown in Figure 9.

Assuming the constants in the Filter clause, aggregation operators, and arithmetic operators in the Return clause are all provided, and pattern sizes in Match clauses are bounded, enumerative search solved 12 benchmarks within the 10-minute time limit. The average synthesis time is 79.53 seconds, and the maximum time is 560 seconds. As shown in Table 2, enumerative search was not able to solve any complex tasks, because it cannot handle the large search space of complex Cypher queries. DMINER, on the other hand, is specifically designed to solve graph query synthesis problem, and is efficient in pruning the search space.

Comparison with LLMs. Having an apple-to-apple comparison with an LLM is inherently challenging because LLM capabilities evolve quickly and their performance depends on the prompting strategy. Our primary goal is not to claim a definitive comparison, but to provide fair and informative baselines using strong LLMs (i.e., GPT-5.4, Opus 4.7) under a carefully designed setup. Specifically, we considered three common strategies for using LLMs: few-shot learning, chain-of-thought prompting, and an agentic loop. For few-shot learning, we provide the graph schema, describe the demonstration format, specify how input graphs and expected output tables are represented, and include rules for generating output tables. We also provide several example synthesis tasks with

Table 2. Comparison with baselines. #T, #P, #D, #I shows the number of benchmarks where the tool times out, yields a plausible but undesired query, a desired query, an incorrect query, respectively. Time represents the average and maximum time to solve the task in milliseconds. #c is the number of solved complex benchmarks.

Baseline	#T	#P	#D	#I	Time (ms)		#c
					avg	max	
Enumerative Search	76	2	12	0	79528	560071	0
Few-shot learning (GPT-5.4)	-	16	41	33	-	-	12
Chain-of-thought (GPT-5.4)	-	14	44	32	-	-	15
Agentic loop (Claude Code + Opus 4.7)	-	21	63	6	-	-	24

desired queries to illustrate the expected behavior. Before the full evaluation, we confirmed that GPT can understand the prompt by checking whether it can explain the output tables and solutions of the example tasks. For chain-of-thought prompting, we add an instruction asking GPT to think step by step. For the agentic loop, we use Claude Code with Opus 4.7 and allow the model to interact with a Cypher evaluation tool, compare generated query results against the demonstration input graphs and output tables, and iteratively revise its outputs when needed.

As shown in Table 2, out of 90 tasks, LLM generated the desired query for 41, 44, and 63 tasks under the three settings, respectively, and generated incorrect queries for 33, 32, and 6 tasks, respectively. Under few-shot and chain-of-thought prompting, many incorrect queries failed to satisfy the demonstrations because these settings provide no execution feedback, preventing the LLM from detecting and correcting errors made during intermediate reasoning. The agentic loop substantially reduces such failures through execution-based validation. However, it can still stop the search incorrectly, either by relying on statically inferred correctness rather than actual execution results or by misinterpreting a partially correct result as fully correct. Overall, chain-of-thought prompting performs comparably to few-shot learning, while the agentic loop performs best among the evaluated LLM baselines. Nevertheless, even the strongest baseline still fails on 27 tasks, producing either incorrect queries or plausible but undesired ones.

In addition, we evaluated whether the LLM baselines can handle the 25 tasks excluded from our benchmark suite because they require features outside DMINER’s supported fragment. The evaluation shows that their success rate remains very low: each setting generated the desired query for only 1 of the 25 tasks. The agentic-loop setting produced the fewest incorrect queries, with 3 incorrect and 21 plausible but undesired queries. Few-shot learning produced 14 incorrect and 10 plausible but undesired queries, while chain-of-thought prompting produced 11 incorrect and 13 plausible but undesired queries. These results suggest that LLMs can occasionally synthesize queries using features outside DMINER’s supported fragment, but current LLM-based approaches still do not reliably solve graph query synthesis problems from demonstrations.

Answer to RQ3. Enumerative search solves only 12 of the 90 benchmarks, while LLM-based approaches solve at most 63. Enumerative search is also significantly slower, requiring 80 seconds on average for each solved task, while LLMs may generate incorrect graph queries. Overall, DMINER outperforms both enumerative search and LLM baselines in synthesizing graph queries from user demonstrations.

6.4 User Study: Effort and Quality of Demonstrations

To explore how much effort users need to provide demonstrations and their quality, we conducted a small-scale user study. In particular, we studied user experience from two perspectives: (1) the effort required from the users, (2) the ability of users to provide high-quality demonstrations that accurately and sufficiently express their intents.

Participants. Our small-scale user study includes 8 participants: 4 software engineers, 2 data analysts, and 2 PhD students in computer science. None of them has prior experience writing graph queries in Cypher. Each participant receives a \$25 gift card as compensation.

Setup. We collected 5 real-world tasks from StackOverflow, where each task consisted of a natural language description and the schema of its corresponding databases. Three tasks are relatively simple, while the other two are more complex: one requires a graph pattern with three nodes and two edges, and the other involves a group-by operation. We ask each participant to provide two or more demonstrations for each task. In each demonstration, participants express the query intent by constructing an input graph that conforms to the given graph database schema and then specifying the corresponding demonstration table as the expected output. To ensure participants understand the demonstration format, we first provide a 15-minute tutorial that introduces computation demonstration with a concrete example. Then each participant is given 30 minutes to complete the demonstrations of all 5 tasks, and we record the time spent on each task. Finally, we have a 15-minute post-study interview with the participant and gather feedback on their experience.

Effort required to provide demonstrations. Based on the user study, participants take an average of 196 seconds to create demonstrations for a task, with the longest completion time being 350 seconds. Full details of the study are provided in Appendix A. Notably, the time required for more complex tasks is not significantly higher than that for simpler ones. In the post-study interview, participants rate the ease of providing demonstrations at an average of 3.7 on a 5-point scale, where 1 indicates very hard and 5 indicates very easy. Most participants reported that it would need some time to understand the task and get familiar with the demonstration format at first, but after that, it was very easy to write the demonstrations.

Quality of provided demonstrations. To measure the quality of demonstrations, we first manually inspect whether they are well-formed. Among the 40 sets of demonstrations collected in total (5 per participant across 8 participants), 3 does not follow the required format. For the remaining 37 well-formed demonstration sets, we run DMINER to determine whether it can synthesize the desired queries. The results show that DMINER can successfully synthesize the desired query for 35 out of the 37 sets, indicating that approximately 95% of the well-formed demonstrations are both accurate and sufficient to convey the intended query semantics. The two failures both arise from demonstrations for complex tasks.

Discussions on providing multiple demonstrations. To understand how users can effectively communicate query intent through demonstrations, we asked participants in the post-study interview whether multiple demonstrations were necessary. The responses are consistently affirmative: all participants indicate that using more than one demonstration is important for clarifying intent, yielding an average rating of 4.5 on a 5-point scale, where 5 denotes very necessary. In general, participants are willing to provide two or more demonstrations and view this effort as both reasonable and useful. As one participant noted, “with more demonstrations, it would be easier to show the task.” Participants also described a consistent strategy for constructing multiple demonstrations. They believe demonstrations should include both straightforward examples that directly express the intended behavior and more complex examples that help exclude similar but undesired intentions. For instance, one participant explained, “The first [demonstration], I think, is a proof of concept, as the simplest way to express my intention. The second or third are used to exclude other cases.”

Answer to RQ4. Our user study with 8 participants shows that users can provide multiple demonstrations for a synthesis task with only modest effort, and that 87.5% of the provided demonstrations are of good quality for DMINER to synthesize the desired queries.

7 Related Work

Synthesizing database queries. A substantial body of research has focused on the synthesis of database queries [18, 31, 35, 38, 48, 50, 60, 62]. For example, SCYTHE [50] synthesizes relational database queries from input-output examples using abstract query semantics to efficiently prune the search space. SICKLE [62] extends this line of work to support more complex analytical queries, including partition-aggregation tasks, based on computation demonstrations. PROSYNTH [42] synthesizes recursive Datalog programs with invented predicates from input-output specifications, leveraging provenance information obtained from a Datalog solver. NOSDAQ [31] synthesizes document database queries by leveraging abstractions over hierarchical data to facilitate pruning of query sketches. Recent work on Text-to-Cypher translation [9, 39, 56] mainly focuses on collecting data or generating synthetic datasets to fine-tune large language models for translating natural language into Cypher queries. However, translating natural language to Cypher is orthogonal to our work, because we focus on synthesizing graph queries from computation demonstrations. The most related along this line is synthesizing SPARQL queries from examples [1, 28]. In particular, prior work leverages techniques such as data provenance analysis [28] and exploratory query refinement [1] to iteratively help users obtain desired SPARQL queries over RDF data. RDF data follows a subject–predicate–object triple model, which differs significantly from the property graph model considered by DMINER. Moreover, DMINER applies graph mining and deduction-based reasoning for efficient synthesis, which is technically distinct from prior work.

Program synthesis by demonstrations. Programming by demonstration has been studied as an effective approach to simplifying and automating programming tasks [7, 8, 11, 26, 27, 40, 41, 62]. Among this line of work, the most related is SICKLE [62], which uses computation demonstrations to capture data provenance for analytical SQL query synthesis. DMINER adopts the notion of computation demonstrations and employs it as the high-level specification to synthesize graph queries. However, due to the substantial differences between graph and relational data models, DMINER takes a distinct approach to synthesize queries: it performs graph mining for pattern synthesis and introduces a new set of rules to prune multiple incorrect queries at once. This is different from the abstract provenance analysis used in SICKLE.

Graph mining. Graph mining is a broad topic about exploring the substructure of graphs on different problems. For example, mining subgraphs with constraints such as mining Maximal Cliques, Quasi-Cliques [10, 24, 29], or frequent subgraphs [22, 25, 36], and the subgraph matching problem [4, 44]. Our focus on graph mining is more closely related to mining patterns with constraints. Unlike the common constraint-based mining techniques such as CONTIGRA [29] where the constraints focus on structures of the graph, or the frequency of the subgraphs [5, 22, 55], our synthesis technique uses commonality and output coverage constraints as necessary conditions to prune subgraphs for graph mining.

Enumerative search with deductive reasoning. *Enumerative search with deductive reasoning* is a widely adopted technique in program synthesis [13–17, 23, 30, 34, 52–54, 59]. Specifically, MRGRATOR [52] computes the minimum failing inputs for sketch completion to achieve fast synthesis of database programs. DYNAMITE [53] extends this idea to the synthesis of Datalog programs. It formulates synthesis as an SMT problem and iteratively refines the encoding by adding blocking clauses derived from minimal distinguishing projections, thereby ruling out spurious candidates. NEO [17] applies conflict-driven learning to generalize from a single infeasible partial program, enabling the pruning of larger parts of the search space. It performs conflict analysis by computing minimal unsatisfiable cores and learning lemmas that guide the search away from similar infeasible decisions. However, NEO’s conflict-driven reasoning relies on domain-specific abstractions. Applying this approach to graph query synthesis would require designing abstractions for particular

graph operators and developing corresponding conflict learning techniques. Therefore, NEO is not directly portable to graph query synthesis in our settings. DMINER also leverages the deductive reasoning idea for sketch completion, but employs a distinct deduction strategy. Our method uses a new set of rules to analyze the difference between actual and expected demonstration tables and derive blocking clauses from the observed difference.

Constraint-based search. Constraint-based search systematically explores solutions that satisfy a set of constraints, which has been applied to various fields such as program synthesis [12, 17, 18, 42, 45, 47, 51, 58] and program repair [20]. For example, FIREMASON [20] automatically repairs firewalls by employing constraint solvers to synthesize new firewall rules that satisfy new packet routing examples and remain the routing of other packets. Wang et al. [51] apply constraint solvers to verify and filter the synthesized invariants, ensuring the invariants are both inductive and sufficient for verifying the property at hand. More closely related, MORPHEUS [18] and NEO [17] utilize constraint solving to prune infeasible program sketches by formulating constraints grounded in the semantics of individual program components, thereby improving synthesis efficiency. MIGRATOR [52] and DYNAMITE [53] employ symbolic enumeration during sketch completion and eliminate incorrect programs through error analysis. DMINER is inspired by prior work [17, 18, 52, 53] in this line and adapts the deductive reasoning technique into the application of synthesizing graph queries from demonstrations. Different from prior work, DMINER adopts a two-phase approach for filtering predicate synthesis: it first encodes the requirements of the predicate (i.e., which subgraphs should be retained by the filter), then applies constraint optimization to generate a concise filter.

8 Limitations

In this section, we discuss several limitations of DMINER and suggest directions for future work.

First, our synthesis technique currently does not support the full Cypher language. In particular, DMINER does not support features such as multiple Match clauses, variable-length patterns, aggregation filtering, or subqueries. The goal of this work is to establish a feasible approach to graph query synthesis from demonstrations, and we take a first step to support simple common graph queries. We believe DMINER can be extended to support richer Cypher features. For example, queries with multiple Match clauses or subqueries can be handled by a top-down decomposition that splits a complex synthesis task into several smaller sub-tasks. Our current synthesis techniques can then be used to synthesize each subquery individually, and the resulting subqueries can be composed to reconstruct the complete query. Supporting variable-length patterns would require extending the pattern language and the corresponding graph mining procedure. We leave these extensions to future work.

Second, our current technique assumes that demonstrations are correct. This assumption is common in example-based program synthesis, where precise input-output examples are typically required, but it offers limited robustness when user-provided demonstrations are noisy or erroneous. A promising direction is to adapt ideas from prior work on synthesizing programs from noisy inputs [43] to the setting of graph query synthesis, allowing DMINER to tolerate imperfect demonstrations while still generating the intended query.

9 Conclusion

This paper studies synthesizing graph queries from user demonstrations. To tackle the challenges posed by complex graph models and graph pattern matching, we propose a synthesis approach that combines graph mining with deduction-based pruning, tailored for graph queries. We implemented our approach in a tool called DMINER and evaluated it on 90 benchmarks. The results demonstrate that DMINER is effective and efficient in synthesizing graph queries from demonstrations.

Acknowledgments

We would like to thank anonymous reviewers for their insightful feedback on the earlier versions of this paper. This work was conducted with the support of Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants RGPIN-2022-04983, RGPIN-2025-04775, NSERC Discovery Launch Supplement DGECR-2022-00417, NSERC Undergraduate Student Research Award, and Simon Fraser University (SFU) Startup Grant.

Data-Availability Statement

The artifact of this work is available [32], which includes the implementation of DMINER, the benchmark suite used in the evaluation, and the scripts and instructions needed to reproduce the experimental results reported in the paper.

References

- [1] Efrat Abramovitz, Daniel Deutch, and Amir Gilad. 2018. Interactive Inference of SPARQL Queries Using Provenance. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 581–592. doi:10.1109/ICDE.2018.00059
- [2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*. 1–8. <https://ieeexplore.ieee.org/document/6679385/>
- [3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International conference on tools and algorithms for the construction and analysis of systems (TACAS)*, Vol. 10205. 319–336. doi:10.1007/978-3-662-54577-5_18
- [4] Bibek Bhattacharai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. <https://doi.org/10.1145/3299869.3300086>
- [5] Christian Borgelt and Michael R. Berthold. 2002. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*. <https://doi.org/10.1109/ICDM.2002.1183885>
- [6] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.* 38, 1 (2006), 2. doi:10.1145/1132952.1132954
- [7] Sarah Chasins and Rastislav Bodik. 2017. Skip blocks: reusing execution history to accelerate web scripts. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). doi:10.1145/3133875
- [8] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*. 963–975. doi:10.1145/3242587.3242661
- [9] Vashu Chauhan, Shobhit Raj, Shashank Mujumdar, Avirup Saha, and Anannay Jain. 2025. Mind the Query: A Benchmark Dataset towards Text2Cypher Task. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*. 1890–1905. doi:10.18653/v1/2025.emnlp-industry.133
- [10] Joanna Che, Kasra Jamshidi, and Keval Vora. 2024. Contigra: Graph Mining with Containment Constraints. In *European Conference on Computer Systems (EuroSys)*. <https://doi.org/10.1145/3627703.3629589>
- [11] Jiun-Hung Chen and Daniel S. Weld. 2008. Recovering from errors during programming by demonstration. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI)*. 159–168. doi:10.1145/1378773.1378794
- [12] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 487–502. doi:10.1145/3385412.3385988
- [13] Yanju Chen, Ruben Martins, and Yu Feng. 2019. Maximal multi-layer specification synthesis. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 602–612. doi:10.1145/3338906.3338951
- [14] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. 2020. Program Synthesis Using Deduction-Guided Reinforcement Learning. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, Vol. 12225. 587–610. doi:10.1007/978-3-030-53291-8_30
- [15] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. 2022. Synthesis-powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 560–588. doi:10.1145/3563308
- [16] Andrew Cropper and Rolf Morel. 2021. Learning programs by learning from failures. *Machine Learning* 110, 4 (2021), 801–856. doi:10.1007/s10994-020-05934-z

- [17] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 420–435. doi:10.1145/3192366.3192382
- [18] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 422–436. doi:10.1145/3062341.3062351
- [19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1433–1445. doi:10.1145/3183713.3190657
- [20] William T. Hallahan, Ennan Zhai, and Ruzica Piskac. 2020. Automated repair by example for firewalls. *Form. Methods Syst. Des.* (2020), 127–153. doi:10.1007/s10703-020-00346-0
- [21] Yang He, Ruijie Fang, Isil Dillig, and Yuepeng Wang. 2025. Graphiti: Bridging Graph and Relational Database Queries. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1716–1740. doi:10.1145/3729319
- [22] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Principles of Data Mining and Knowledge Discovery (PKDD)*. https://doi.org/10.1007/3-540-45372-5_2
- [23] Marques-Silva Joao, Lynce Ines, and Malik Sharad. 2009. Conflict-Driven Clause Learning SAT Solvers. 185 (2009), 131–153. doi:10.3233/978-1-58603-929-5-131
- [24] Jalal Khalil, Da Yan, Guimu Guo, and Luyheng Yuan. 2022. Parallel mining of large maximal quasi-cliques. *Proc. VLDB Endow.* (2022). <https://doi.org/10.1007/s00778-021-00712-2>
- [25] Michihiro Kuramochi and George Karypis. 2005. Finding Frequent Patterns in a Large Sparse Graph*. *Data Min. Knowl. Discov.* (2005). <https://doi.org/10.1007/s10618-005-0003-9>
- [26] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156. doi:10.1023/A:1025671410623
- [27] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *PLDI*. 542–553. doi:10.1145/2666356.2594333
- [28] Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. 2023. Example-Driven Exploratory Analytics over Knowledge Graphs. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. 105–117. doi:10.48786/EDBT.2023.09
- [29] Guimei Liu and Limsoon Wong. 2008. Effective Pruning Techniques for Mining Quasi-Cliques. In *Machine Learning and Knowledge Discovery in Databases, European Conference (ECML/PKDD)*. https://doi.org/10.1007/978-3-540-87481-2_3
- [30] Junrui Liu, Yanju Chen, Eric Atkinson, Yu Feng, and Rastislav Bodik. 2023. Conflict-Driven Synthesis for Layout Engines. 7, PLDI (2023). doi:10.1145/3591246
- [31] Qikang Liu, Yang He, Yanwen Cai, Byeongguk Kwak, and Yuepeng Wang. 2025. Synthesizing Document Database Queries Using Collection Abstractions. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 476–488. doi:10.1109/ICSE55347.2025.00152
- [32] Xiaoyu Liu, Qikang Liu, Evan Dyce, Keval Vora, and Yuepeng Wang. 2026. Artifact for “Synthesizing Graph Queries from Demonstrations”. <https://doi.org/10.5281/zenodo.21930599>.
- [33] Xiaoyu Liu, Qikang Liu, Evan Dyce, Keval Vora, and Yuepeng Wang. 2026. Extended version of “Synthesizing Graph Queries from Demonstrations”. <https://doi.org/10.5281/zenodo.21913286>.
- [34] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: an extensible synthesis framework for data science. *Proc. VLDB Endow.* 12, 12 (2019), 1914–1917. doi:10.14778/3352063.3352098
- [35] Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. 2021. GENSYNTH: Synthesizing Datalog Programs without Language Bias. *Proceedings of the AAAI Conference on Artificial Intelligence* (2021), 6444–6453. doi:10.1609/aaai.v35i7.16799
- [36] Jinghan Meng and Yi-Cheng Tu. 2017. Flexible and Feasible Support Measures for Mining Frequent Patterns in Large Labeled Graphs. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. <https://doi.org/10.1145/3035918.3035936>
- [37] Neo4J. 2025. Neo4J Community Website. <https://community.neo4j.com/>.
- [38] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco Manquinho. 2020. SQUARES: a SQL synthesizer using query reverse engineering. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2853–2856. doi:10.14778/3415478.3415492
- [39] Makbule Gulcin Ozsoy, Leila Messallem, Jon Besga, and Gianandrea Minneci. 2025. Text2Cypher: Bridging Natural Language and Graph Databases. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025 - Workshops, Abu Dhabi, UAE, January 19-24, 2025*. 100–108. <https://aclanthology.org/2025.genai-1.11/>
- [40] Noah Patton, Kia Rahmani, Meghana Missula, Joydeep Biswas, and Isil Dillig. 2024. Programming-by-Demonstration for Long-Horizon Robot Tasks. *Proc. ACM Program. Lang.* 8, POPL (2024). doi:10.1145/3632860

- [41] Gordon W Paynter. 2000. *Automating iterative tasks with programming by demonstration*. Ph.D. Dissertation. The University of Waikato.
- [42] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2019). doi:10.1145/3371130
- [43] Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 761–774. doi:10.1145/2837614.2837671
- [44] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. Pilos: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *International Conference on Data Engineering*. <https://doi.org/10.1109/ICDE65448.2025.00093>
- [45] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer (STTT)* 15, 5 (2013), 475–495. doi:10.1007/s10009-012-0249-7
- [46] Rania Soussi, Marie-Aude Aufaure, and Hajer Baazaoui Zghal. 2010. Towards Social Network Extraction Using a Graph Database. In *International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 28–34. doi:10.1109/DBKDA.2010.19
- [47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2010. From program verification to program synthesis. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. 313–326. doi:10.1145/1707801.1706337
- [48] Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. 2021. PATSQL: efficient synthesis of SQL queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.* 14, 11 (2021), 1937–1949. doi:10.14778/3476249.3476253
- [49] Lei Tang and Huan Liu. 2010. *Graph Mining Applications to Social Network Analysis*. 487–513. doi:10.1007/978-1-4419-6045-0_16
- [50] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 452–466. doi:10.1145/3062341.3062365
- [51] Jingbo Wang and Chao Wang. 2023. Learning to Synthesize Relational Invariants. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. doi:10.1145/3551349.3556942
- [52] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 286–300. doi:10.1145/3314221.3314588
- [53] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. doi:10.14778/3384345.3384350
- [54] Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. 1001–1015. doi:10.1145/3658644.3690293
- [55] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM)*. <https://doi.org/10.1109/ICDM.2002.1184038>
- [56] Chao Yang, Changyi Li, Xiaodu Hu, Hao Yu, and Jinzhi Lu. 2025. Enhancing knowledge graph interactions: A comprehensive Text-to-Cypher pipeline with large language models. *Inf. Process. Manage.* (2025). doi:10.1016/j.ipm.2025.104280
- [57] Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. 2017. Use of graph database for the integration of heterogeneous biological data. *Genomics & informatics* 15, 1 (2017), 19–27. doi:10.5808/GI.2017.15.1.19
- [58] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI (2023). doi:10.1145/3591288
- [59] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 279–285. doi:10.1109/ICCAD.2001.968634
- [60] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing SQL queries from input-output examples. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 224–234. doi:10.1109/ASE.2013.6693082
- [61] Lingfeng Zhong, Jia Wu, Qian Li, Hao Peng, and Xindong Wu. 2024. A Comprehensive Survey on Automatic Knowledge Graph Construction. *ACM Comput. Surv.* 56, 4 (2024), 94:1–94:62. doi:10.1145/3618295
- [62] Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing analytical SQL queries from computation demonstration. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 168–182. doi:10.1145/3519939.3523712

Received 2026-03-17; accepted 2026-08-06