EXAMPLE GUIDED SYNTHESIS OF RELATIONAL QUERIES

Aalok Thakkar

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2023

Supervisors of Dissertation

Rajeev Alur	Mayur Naik
Zisman Family Professor of Computer and Information Science	Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Val Tannen, Professor of Computer and Information Science
Susan Davidson, Weiss Professor of Computer and Information Science
Oleksandr Polozov, Research Scientist at X, the moonshot factory
Stephan Zdancewic, Schlein Family Professor of Computer and Information Science

ABSTRACT

EXAMPLE GUIDED SYNTHESIS OF RELATIONAL QUERIES

The goal of program synthesis is to automatically generate programs that meet user intention. While a number of methods for expressing user intention has gained traction over the last five decades, programming-by-example has proven to be useful in domains where the user may not be able to articulate the desired program behavior as a logical specification but can describe it through demonstrative input-output examples.

This dissertation studies programming-by-example in the context of relational queries. It is a challenging and foundational problem; ideally, we would like a technique that is simultaneously: (a) scalable enough to be applicable to real-world instances, (b) expressive in terms of the kinds of queries that it can synthesize, and (c) fully automatic, so it requires minimal guidance from non-expert users. Significant progress has been made on this problem in recent years Cropper and Dumančić (2022), and a variety of algorithms have been proposed, including algorithms based on evolutionary search Mendelson et al. (2021), numerical relaxation Si et al. (2019), constraint solving Law et al. (2020a); Cropper and Morel (2021), and counterexample-guided search Raghothaman et al. (2020a).

Each of these approaches require additional supervision in the form of templates to restrict the space of candidate programs and accelerate the search. In this line of work, we propose *example-guided synthesis*, a paradigm of techniques to eliminate the need for such instance-specific supervision by leveraging the underlying structure of the input-output examples. We present an example-guided algorithm for *conjunctive queries*, and then extend it to support expressive features such as disjunction, recursion, and comparison predicates, as well as learning in presence of noise.

We implement this technique and demonstrate that it outperforms the state-of-the-art tools on a diverse set of benchmarks in terms of both, running time and the quality of examples.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

The prevalence and use of structured data for diverse application domains including scientific computing, medicine, and finance require users to produce small but inherently complex queries that demand algorithmic insights and expertise of programming syntax. For end-users of these database systems who may not be experts at programming, designing queries can become ardours.

Over the last ten years, program synthesis technology has matured to become a practical tool that addresses this concern. Program synthesis aims to automatically find programs in a given programming language that satisfy user intent. Unlike compilers that translate a formally specified description (such as a regular language expression) to a low-level machine representation (such as an automaton), synthesis tools perform a search over a space of candidate programs to generate a program. The user intent for these programs can be defined in terms of formal specification, natural language instances, or input-output examples.

Synthesis has been the holy grail of computer science research at least since the late 60s; it was considered by Pnueli to be one of the most central problems in the theory of computation [Pnueli and Rosner (1989)]. It was soon proved that program synthesis, in general, is intractable, and therefore, all efforts at synthesis have had to incorporate human insight into the synthesis process. This is primarily done in two ways:

1. Restricting expressiveness of the programs, and

2. Providing additional supervision for users to direct the search process.

This thesis fundamentally studies the trade-off between these two ways in the context of relational query synthesis.

1.1. Relational Query Synthesis

Relational queries are declarative logic programs and find applications in domains such as knowledge discovery, program analysis, and in querying databases. They operate over relational algebras [Codd (1970)] and form the basis of database query languages such as SQL, Datalog [Abiteboul et al. (1994)], SPARQL [Pérez et al. (2009)], Cypher [Francis et al. (2018)], as well as their variants for querying code, such as PQL [Martin et al. (2005)], LogiQL [Green (2015)], and CodeQL [Avgustinov et al. (2016)].

The problem of synthesizing such logic programs from input-output examples has been studied over the last two decades and a number of tools have been developed with varying restrictions on the expressiveness of the programs and with different needs for additional supervision. In this section, we summarize the advances in enumerative, constraint solving, and hybrid techniques to solve the relational query synthesis problem. Each of the tools uses a form of instance-specific supervision to guide the search.

1.1.1. Syntax-guided Techniques

Syntax-guided Synthesis (SyGuS) is a classical formulation of program synthesis where the user may supplement the input-output examples with syntactic templates to constrains the space of allowed programs. Si et al. (2018) build a syntax-guided tool called ALPS for the synthesis of relational queries (in particular, Datalog queries) where the user provides instance-specific supervision in form of *meta-rules*. According to the Si et al. (2018): *the key challenge is to obtain a set of meta-rules that is general enough to capture useful programs but specific enough to enable efficient synthesis.*

Another paradigm for enumerative synthesis is a two-phased search where the synthesis problem is decomposed into first searching for an *abstract query* and then searching for predicates that can *instantiate* the abstractions. Wang et al. (2017a) develops the tool Scythe and Takenouchi et al. (2021) develops the tool PatSQL that implement such a two-phased approach. Both the tools target the domain of SQL queries and require an exhaustive list of constants that may be used for the comparison operators in the query.

A syntax-guided technique that differs from these two paradigms is GenSynth, an evolutionary search strategy that mutates candidate programs and evaluates their fitness on the input-output examples. GenSynth stands out among the synthesis engine as it requires the least instance-specific supervision. It requires the signatures of *invented predicates*, unless they coincide with that of an input or output relations.

1.1.2. Constraint-solving Techniques

Constraint-solving techniques inherently require syntactic constraints restrict the search space to a finite set of candidate programs. Then, they use SMT solvers to navigate through this finite search space.

Albarghouthi et al. (2017) introduces a constraint-based synthesis technique for Datalog programs that uses uses an SMT solver to search through the space of Datalog programs defined by a set of constraints on the number of clauses in each program as well as the length of each clause.

ProSynth is a provenance-guided technique for Datalog synthesis that generates constraints using the provenance information from a program evaluator, and requires an exhaustive list of candidate rules to restrict the search space.

Similarly, other Inductive Logic Programming tools such as Popper, Metagol, and ILASP use different syntactic constraints (hypothesis constraints, metarules, and mode declarations) to restrict the search space. They all synthesize fragments of Answer Set Programs.

1.2. An Example-guided Approach to Synthesis

All existing techniques for synthesis of relational queries rely on instance-specific supervision. In this thesis, we attempt to restrict the expressiveness of relational queries to develop fully automated push-button techniques.

In recent years, techniques such as FlashFill [Gulwani (2011)] have demonstrated the effectiveness of fully automated techniques when restricted to specific domains. FlashFill synthesizes string transformations by analyzing the structure of input and output examples and searches for common patterns between them. FlashRelate [Barowy et al. (2015)] and Golem [Muggleton and Feng (1990)]

are other examples of fully automated techniques that leverage the structure of the input-output examples, and have shown to be more scalable then their syntax-guided and constraint-solving based counterparts.

While all PBE techniques use the examples in some form, we call a technique example-guided only when it meets the following criteria:

1. The candidate programs enumerated by the search depend on the latent structure of the input-output examples, and not just a grammar of the target language, and
2. The input-output examples cannot be replaced by a black-box verification oracle that checks if a candidate program is consistent with the input-output examples or not.

Additionally, our objective is to develop example-guided techniques that do not require instance-specific supervision and allow for a fully automated push-button synthesis framework for relational queries.

1.3. Contributions of this Dissertation

Concretely, this thesis makes the following contributions:

1. We identify a category of synthesis algorithms for PBE called *Example-guided Synthesis* which exploit the latent structure in the provided examples while generating candidate programs.
2. We study the problem of relational query synthesis and establish its decidability and complexity.
3. We develop an example-guided algorithm for synthesizing conjunctive relational queries by leveraging patterns in the input-output examples.
4. We show that this algorithm can be extended to support relational queries with a variety of features such as disjunction, numerical comparison, and recursion.
5. We show that the algorithm can be extended to support learning in presence of noisy input-output data.

6. We demonstrate that our technique outperforms state-of-the-art tools on a variety of benchmarks across multiple dimensions: running time, quality of programs, and in proving unrealizability.

CHAPTER 2

Problem Formulation

We devote this chapter to formalizing the query synthesis problem and proving its decidability and complexity.

2.1. Problem Setting

We begin by presenting an overview of the example-guided synthesis (EGS) framework. As an example, consider a researcher who has data describing traffic accidents in a city and who wishes to explain this data using information about the road network and traffic conditions.

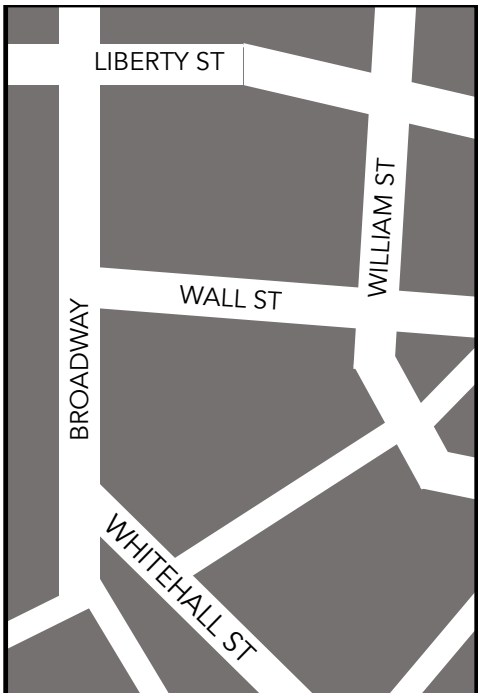
We present this data in Figure 2.1. Suppose that at a given instant, accidents occur on Broadway and Whitehall. The researcher observes that these streets intersect, that they both had traffic, and that the traffic lights on both streets were green. They generalize this observation, and find that the resulting hypothesis, that an accident occurs at every pair of streets with similar conditions, is consistent with the data. One may formally describe their hypothesis as the following Horn clause:

$$\begin{aligned}\texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(x, y), \texttt{HasTraffic}(x), \texttt{HasTraffic}(y),\\ \texttt{GreenSignal}(x), \texttt{GreenSignal}(y),\end{aligned} \tag{2.1}$$

where x and y are universally quantified variables ranging over street names, ":-" denotes implication "$\Leftarrow$", and "," denotes conjunction. Our goal is to automate the discovery of such hypotheses.

This problem can be naturally formalized as a programming-by-examples (PBE) task. Given a set of input facts I encoded as relations, and a set of desirable and undesirable output facts, O^+ and O^- respectively, we seek a program which derives all of the tuples in O^+ and none of the tuples in O^-. In our example, we implicitly assume that the data is completely labelled, so that

$$O^+ = \{\ \texttt{Crashes}(\mathsf{Broadway}), \texttt{Crashes}(\mathsf{Whitehall})\ \},$$



(a)

Intersects	
Broadway	Liberty St
Broadway	Wall St
Broadway	Whitehall
Liberty St	Broadway
Liberty St	William St
Wall St	Broadway
Wall St	William St
Whitehall	Broadway
William St	Liberty St
William St	Wall St

GreenSignal
Broadway
Liberty St
William St
Whitehall

HasTraffic
Broadway
Wall St
William St
Whitehall

Crashes
Broadway
Whitehall

(b)

Figure 2.1: Data describing traffic conditions in a city: (2.1a) Map of the city, (2.1b) and listing of the input and output relations. We would like to explain the accidents occurring on Broadway and Whitehall.

and O^- is the set of all other streets,

$$O^- = \{\ \texttt{Crashes}(\mathsf{Liberty\ St}), \texttt{Crashes}(\mathsf{Wall\ St}), \texttt{Crashes}(\mathsf{William\ St})\ \}.$$

Traditional methods for PBE use syntax-guided enumerative techniques that search the space of candidate programs. In our example, a candidate program would be a Horn clause with the premise consisting of one or more of `HasTraffic`, `GreenSignal`, or `Intersects` literals.

A naive approach is to enumerate all candidate programs in order of increasing size till we find a consistent hypothesis. For the running example, we will have to enumerate more than 12×10^6 candidate programs before discovering the one shown in Equation 2.1. Unsurprisingly, most work on program synthesis has focused on reducing the size of this search space: in our context, tools such as ALPS and ProSynth restrict the search space by only looking for programs composed of rules from a fixed finite set of candidate rules Si et al. (2018); Raghothaman et al. (2020b), while ILASP constrains the space through "mode declarations" that bound the number of joins (in our case conjunctions) and the number of variables used Law et al. (2014, 2020b). On the other hand, Scythe, a synthesis tool for SQL queries, first finds "abstract" queries that over-approximate the desired output, and then searches for concrete instantiations of these abstract queries that are consistent with the data Wang et al. (2017b).

2.2. Syntax and Semantics of Relational Queries

Different fragments of relational queries are defined using a variety of languages such as select-project-join (SPJ) queries, SQL, Datalog, and Prolog. In this thesis, we will use the syntax of Datalog to define relational queries.

2.2.1. Datalog Syntax

A relational query Q is a set of rules. To define the syntax of rules, we first fix a set of *input* predicates, a set of invented predicates, and a set of output predicates. Each predicate R is associated with an *arity* k. A *literal*, $R(v_1, v_2, \ldots, v_k)$, consists of a k-ary predicate R with a list of k variables.

Then, a rule r is of the form:

$$R_h(\vec{u}_h) \text{ :- } R_1(\vec{u}_1), R_2(\vec{u}_2), \ldots, R_n(\vec{u}_n), \tag{2.2}$$

where the single literal on the left, $R_h(\vec{u}_h)$, is the *head* of r and $R_1(\vec{u}_1)$, $R_2(\vec{u}_2)$, ..., $R_n(\vec{u}_n)$, is called the *body* of r. The literals in the body can have input predicates, invented predicates, or output predicates, while the head of the rules must have either invented predicates or output predicates. A variable that occurs in the head must appear at least once in the body in order to bound the variables.

The program in Equation 2.1 is an example of a relational query. The head consists of $\texttt{Crashes}(x)$ and the body has five literals.

2.2.2. Semantics

The semantics of a relational query may be specified in multiple equivalent ways (Abiteboul et al. (1994)). In our work, we will formalize their semantics using rule instantiations and derivation trees. The semantics of a relational query is interpreted over a data domain D whose elements are called *constants*. For simplicity of formalization, we are assuming that there is a single type. The synthesis framework and its theoretical guarantees can be extended to support typed constants and typed relations. We can define rule instantiation as:

Definition 2.2.1 (Rule Instantiation)**.** Given a map v from variables to the data domain D, the rule instantiation of a rule as in Equation 2.2 is:

$$R_h(v(\vec{u}_h)) \impliedby R_1(v(\vec{u}_1)), R_2(v(\vec{u}_2)), \ldots, R_n(v(\vec{u}_n)).$$

That is, one can systematically replace each variable x by $v(x)$. For example, consider the query in Equation 2.1. One can systematically replace its variables according to the map $\{x \mapsto \mathsf{Whitehall}, y \mapsto$

Broadway} to obtain the rule instantiation:

$$\begin{aligned}\texttt{Crashes}(\text{Whitehall}) \impliedby\ & \texttt{Intersects}(\text{Whitehall}, \text{Broadway}),\\ & \texttt{HasTraffic}(\text{Whitehall}), \texttt{HasTraffic}(\text{Broadway}),\\ & \texttt{GreenSignal}(\text{Whitehall}), \texttt{GreenSignal}(\text{Broadway}).\end{aligned} \tag{2.3}$$

We say that a tuple t is derivable from input tuples I if there exists a rule r and a map v such that on instantiating r with v, the head tuple $R_h(v(\vec{u}_h))$ is t, and each of the tuples in the body $R_i(v(\vec{u}_i))$ occur in I. Then, a relational query Q takes input tuples I and returns output tuples $O = [\![Q]\!](I)$ as the set of all tuples that are derivable from I using rules in Q.

2.2.3. Syntax and Semantics of SPJ Queries

We now study select-project-join queries where selection supports categorical and numerical comparisons, and all joins are equi-joins. We will use SQL syntax to denote these SPJ queries.

To define the syntax of these queries, we first fix a set of *input tables* and a set of *output tables*. For simplicity, the columns of each table are of either of the kinds: `categorical`, `numerical`, and `uncomparable`.

The syntax of the select-project-join queries is defined by the grammar in Figure 2.2. The `JOIN` operator featured in this query is an *equi-join*, that is a join parameterized by a set of columns θ. Comparisons of the form $(T.c = k)$ or $(T.c \neq k)$ are supported only for columns of the `categorical` kind, all other comparisons are only supported for the `numerical` kind, and no comparisons are supported for the `uncomparable` kind.

The semantics for these queries are as defined in classical works on relational algebra (Date (2009); Imieliński and Lipski (1984)). We denote the set of tuples produced by query Q on input tables I as $[\![Q]\!](I)$. We consider the *set-semantics* and not *bag-semantics*, that is, a relation is a set of literals with the same predicate (such as `registration`, `instructor`, and `department`).

$$
\begin{aligned}
Q &:\text{-} \quad \texttt{SELECT}\ (T_1.c_1, \ldots T_n.c_n)\ \texttt{FROM}\ J\ \texttt{WHERE}\ \sigma \\
J &:\text{-} \quad T \mid J\ \texttt{JOIN}\ T\ \texttt{ON}\ \theta \\
\sigma &:\text{-} \quad T.c \backsim k \mid \sigma_1\ \texttt{AND}\ \sigma_2 \mid \sigma_1\ \texttt{OR}\ \sigma_2 \\
\theta &:\text{-} \quad T_1.c_1 = T_2.c_2 \mid \theta_1\ \texttt{AND}\ \theta_2 \\
\backsim &:\text{-} \quad = \mid \neq \mid < \mid \leq \mid > \mid \geq
\end{aligned}
$$

Figure 2.2: Grammar for select-project-join queries. T ranges over tables, c ranges over column names, and k ranges over constant values. The grammar does not feature operators for negation, aggregation, or ordering.

2.2.4. Definitions

In this section, we define some terms that are used throughout the proposal.

Definition 2.2.2 (Conjunctive Queries)**.** A query comprising of a single rule as defined in Section 2.2.1 that uses only input predicates in its body is termed a conjunctive query.

Conjunctive queries are also termed *select-project-join* (SPJ) queries because of their representation in relational algebra, and are correspond to queries expressed using the select-from-where idiom in SQL. Adding the disjunction operator to these queries give us *union of conjunctive queries*:

Definition 2.2.3 (Union of Conjunctive Queries (UCQ))**.** A query comprising of rules as defined in Section 2.2.1 that use only input predicates in their bodies is termed a union of conjunctive queries.

Observe that UCQs are inherently non-recursive as the predicates are divided into either output predicates that can occur in the head of a rule or input predicates that can occur in the body of a rule. In order to define recursive queries, we introduce the concept of invented predicates.

Definition 2.2.4 (Invented Predicate)**.** An invented predicate is one that is neither an output predicate, nor an input predicate.

Consider the following program $P_{\texttt{scc}}$ to identify the pair of vertices in a directed graph that are in

the same strongly connected component (given only the `edge` relation as an input) :

$$\begin{aligned} \texttt{scc}(x, y) &\text{ :- } \texttt{path}(x, y), \texttt{path}(y, x). \\ \texttt{path}(x, z) &\text{ :- } \texttt{path}(x, y), \texttt{path}(y, z). \\ \texttt{path}(x, y) &\text{ :- } \texttt{edge}(x, y). \end{aligned} \tag{2.4}$$

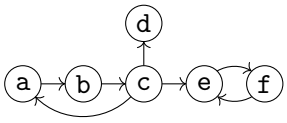
Observe that the predicate `path` is neither an input predicate nor an output predicate. It is the transitive closure of `edge` that is used as an intermediate to define the `scc` relation. Therefore, it is an example of an invented predicate. Additionally, `path` also calls itself and in that sense, it is a recursive predicate.

Definition 2.2.5 (Recursive Predicate)**.** A predicate R is said to be recursive if there exists a finite sequence of rules $r_1, r_2, \ldots, r_k$ such that R occurs in the head of rule r_1 and the body of rule r_k, and the head of each rule r_{i+1} occurs in the body of the rule r_i.

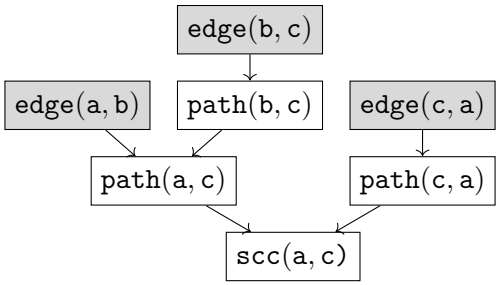
That is, a predicate is said to be recursive if it can call itself during the execution of the program. The semantics of a recursive query are best defined using derivation trees.

Definition 2.2.6 (Derivation Tree)**.** Given a query P and a valuation of the input relations I, a derivation tree of a tuple t is a labelled rooted tree where: (a) each node of the tree is labeled by a tuple, (b) each leaf is labeled by a tuple in I; (c) the root node is labeled by t; and (d) for each internal node labeled α, there exists an instantiation $\alpha \Longleftarrow \beta_1, \ldots, \beta_n$ of a rule in P such that the children of the node are respectively labelled $\beta_1, \ldots, \beta_n$.

We say that a query P derives t using I if there exists a derivation tree for t. Consider, for example the graph in Figure 2.3a. Figure 2.3b shows the derivation tree for $\texttt{scc}(\texttt{a}, \texttt{b})$ in $P_{\texttt{scc}}$. The output $[\![P]\!](I)$ of a query P given an input I is the set of output tuples $R(c_1, c_2, \ldots, c_k)$ which it derives from I. The query $P_{\texttt{scc}}$ on the input in Figure 2.3a generates the output as in Figure 2.3c.



(a) Graph G.



(b) Example derivation tree.

Query semantics ($[\![P_{\texttt{scc}}]\!](I)$):		
scc(a,a),	scc(a,b),	scc(a,c),
scc(b,a),	scc(b,b),	scc(b,c),
scc(c,a),	scc(c,b),	scc(c,c),
scc(e,e),	scc(e,f),	scc(f,e)
scc(f,f)		

(c) Semantics of $P_{\texttt{scc}}$ with respect to the input of directed graph G as in Figure 2.3a. The set I is the set of input tuples and the query semantics are $[\![P_{\texttt{scc}}]\!](I)$.

Figure 2.3: The synthesis task is specified as a search for a relational query P that takes the graph G as an input and returns a set of pairs of vertices O such that O is a superset of O^+ and disjoint from O^-.

2.3. Relational Query Synthesis Problem

Our ultimate goal is to synthesize relational queries which are consistent with a given set of examples. In this context, an example consists of input and output tuples; the user has labeled the output tuples as either positive or negative. The objective then is to synthesize a program which is consistent with the examples, that is, a program which derives all of the positive tuples and none of the negative tuples.

Problem 2.3.1 (Relational Query Synthesis Problem)**.** *Given input relation names $\mathcal{I}$, output relation names $\mathcal{O}$, input tuples I, and output tuples partitioned as O^+ and O^-, return a relational query Q such that $O^+ \subseteq [\![Q]\!](I)$ and $O^- \cap [\![Q]\!](I) = \emptyset$, if such a query exists, and* `unsat` *otherwise.*

We call the triple $M = (I, O^+, O^-)$ an *example*, and a query Q is said to be consistent with it if $O^+ \subseteq [\![Q]\!](I)$ and $O^- \cap [\![Q]\!](I) = \emptyset$.

2.4. Decidability and Complexity

We will now show that checking whether a synthesis problem instance is solvable is co-NP complete. One of the main ingredients of this proof will be the following construction:

Let the data domain $D = \{c_1, c_2, \ldots, c_n\}$, and the input tuples $I = \{R_1(\vec{c}_1), R_2(\vec{c}_2), \ldots, R_n(\vec{c}_n)\}$. Then, for $t = R(\vec{c})$, we then define the rule $r(t)$ as follows:

$$r(t) : R(\vec{v}) \text{:-}\ R_1(\vec{v}_1), R_2(\vec{v}_2), \ldots, R_n(\vec{v}_n).$$

where the head $R(\vec{v})$ and the body literals $R_i(\vec{v}_i)$ are obtained by by consistently replacing the constants in the output tuple $R(\vec{c})$ and input tuples $R_i(\vec{c}_i)$ with fresh variables v_c. The idea is that the body of this rule captures *all* patterns which exist among the input tuples. The rule $r(t)$ is therefore the strongest query in this data which also produces t. This gives us the following lemma:

Lemma 2.4.1. *Given a problem instance $E = (I, O^+, O^-)$, let $Q_{O^+} = \{r(t) : t \in O^+\}$. The problem instance admits a solution if and only if Q_O^+ is consistent with E.*

Proof. One direction of the claim is immediate: if Q_{O+} is consistent with E, then the problem admits a solution.

In the reverse direction, suppose that Q_{O+} is not consistent with E but there exists a query P consistent with E. Observe that for each $t \in O^+$, the rule $r(t)$ can produce it by picking an appropriate instantiation with which it was constructed. Hence, there exists a tuple $t^- \in O^-$ that is produced by Q_{O^+}. We will show that P also produces t^- and establish a contradiction.

Since P is consistent with E, $t \in [\![P]\!](I)$. Let τ be the derivation tree which produces t. Pick the variable valuation $\gamma : X \to D$ which causes $r(t)$ to produce the tuple t^-. Let $v : D \to X$ be the map that was used to construct $r(t)$. Apply the constant renaming map $f = \gamma \circ v : D \to D$ to every node of the derivation tree τ to produce the renamed tree $f(\tau)$. Observe that $f(\tau)$ is still a well-formed derivation tree of the query P, and that $f(t) = t^-$. It follows that the query P also produces t^- as an output tuple, contradicting our assumption that P was consistent with E. □

We now establish our main complexity result, which follows from Claims 2.4.3, 2.4.4, and 2.4.5.

Theorem 2.4.2. *Determining whether an instance of the relational query synthesis problem admits a solution is co-NP complete.*

We devote the rest of this section to the proof of this theorem.

Claim 2.4.3. The problem of determining whether Q_O^+ is consistent with the input-output example $E = (I, O^+, O^-)$ is in co-NP.

Proof. By construction, $O^+ \subseteq [\![Q_{O^+}]\!](I)$ and it only remains to check that $O^- \cap [\![Q_{O^+}]\!](I) = \emptyset$. A rule $r \in Q_{O^+}$ and map v from variables to constants serve as a certificate of $O^- \cap [\![Q_O^+]\!](I) \neq \emptyset$. The certificate can be verified by confirming that the tuple derived by instantiating r with v is in O^-. It follows that checking whether $Q_{I \mapsto O^+}$ is consistent with E is in co-NP. □

To show co-NP hardness, we reduce the problem of checking whether an undirected graph $G = (V, E)$ has a clique of size k to that of determining whether an instance of the synthesis problem is *unsolvable*. Without loss of generality, assume that G does not have self-loops. Consider a set of k constants $V_k = \{v_1, v_2, \ldots, v_k\}$ disjoint from V. Then, consider the instance of the synthesis problem (I, O^+, O^-), where:

$$\begin{aligned} I &= \{\texttt{edge}(u, v) \mid (u, v) \in E\} \\ &\qquad \cup \{\texttt{edge}(v_i, v_j) \mid v_i, v_j \in V_k, v_i \neq v_j\}, \\ O^+ &= \{\texttt{clique}(v) \mid v \in V_k\}, \text{ and} \\ O^- &= \{\texttt{clique}(u) \mid u \in V\}. \end{aligned}$$

Claim 2.4.4. If G does not have a clique of size k, then the given instance is realizable.

Proof. Consider a query Q with only one rule:

$$\texttt{clique}(x_1) \text{ :- } \texttt{edge}(x_1, x_2), \ldots, \texttt{edge}(x_i, x_j), \ldots \texttt{edge}(x_k, x_{k-1}).$$

Where the premise consists of $\texttt{edge}(x_i, x_j)$ for $i \neq j$. If G does not have a clique, then we claim that $[\![P]\!](I) = O^+$. It is clear that $O^+ \subseteq [\![Q]\!](I)$. For sake of contradiction, let $\texttt{clique}(u) \in O^- \cap [\![Q]\!](I)$. Then, there is a map $v : \{x_1, \ldots, x_k\} \to V \cup V_k$ such that instantiating the rule r with v derives $\texttt{clique}(u)$ for some $u \in V$. I must contain a tuple $\texttt{edge}(v(x_i), v(x_j)) \in I$ for each $i \neq j$. By construction of I, if $\texttt{edge}(x, y) \in I$, then $x \neq y$, so each $v(x_i)$ is distinct. Also, we know that $u = v(x_1) \in V$ and $\texttt{edge}(u, v(x_i)) \in I$ for $2 \leq i \leq k$, hence, $v(x_i) \in V$. Let $u_k = v(x_k)$. We have distinct vertices $u_1, \ldots, u_k$ each in V such that there is an edge between them. Then, these vertices form a k-clique, contradicting the assumption. □

Claim 2.4.5. If G has a clique of size k, then the given instance is unrealizable.

Proof. Let the vertices $u_1, \ldots, u_k$ form a clique in G. Consider the map $\pi : V \cup V_k \to V$ where $\pi(u) = u$ for $u \in V$ and $\pi(v_i) = u_i$ for $v_i \in V_k$. For sake of contradiction, let P be a query consistent with the input-output example, and hence, $\texttt{clique}(v_1) \in [\![P]\!](I)$. The derivation tree for $\texttt{clique}(u_1)$ in P can be constructed by replacing each v by $\pi(v)$ in the derivation tree of $\texttt{clique}(v_1)$ in P. Hence, $u_1 \in [\![P]\!](I) \cap O^-$, contradicting the assumption that P is consistent with the input-output example. □

Lemma 2.4.1, and Claims 2.4.3, 2.4.4, 2.4.5 allow us to conclude the Relational Query Synthesis Problem is co-NP complete. Moreover, the Q_{O^+} construction synthesizes a polynomial sized relational query using the input-output examples.

CHAPTER 3

Synthesis of Conjunctive Queries

In this chapter, we describe the core algorithm for example-guided synthesis of relational queries. For the purpose of this chapter, we will be using the running example of traffic crashes.

3.1. Searching Patterns of Co-occurence

Consider the alternative representation of the training data shown in Figure 3.1, summarizing input facts I. We call this the constant co-occurrence graph G_I: every constant is mapped to a vertex, and the edges indicate the presence of a tuple in which the constants occur simultaneously.

In order to synthesize a query, we pick an output tuple, say Whitehall, and focus on the portion of the graph surrounding it. Of the 18 tuples present in the data, only 4 tuples refer to this street: `GreenSignal`(Whitehall), `HasTraffic`(Whitehall), `Intersects`(Whitehall, Broadway), and `Intersects`(Broadway, Whitehall). With these tuples, we can identify the following candidate queries:

$$
\begin{aligned}
q_1: &\quad \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x),\\
q_2: &\quad \texttt{Crashes}(x) \text{ :- } \texttt{HasTraffic}(x),\\
q_3: &\quad \texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(x, y), \text{ and}\\
q_4: &\quad \texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(y, x).
\end{aligned}
$$

Notice that these queries produce the desirable tuples `Crashes`(Whitehall) and `Crashes`(Broadway), but also produce several undesirable tuples: two undesirable tuples by q_1 and q_2, and three undesirable tuples by q_3 and q_4 respectively.

Each of these candidate programs can be made more specific by considering sets of tuples. For example, one can extend the set $C_1 = \{$ `GreenSignal`(Whitehall) $\}$ which produces q_1 with a new

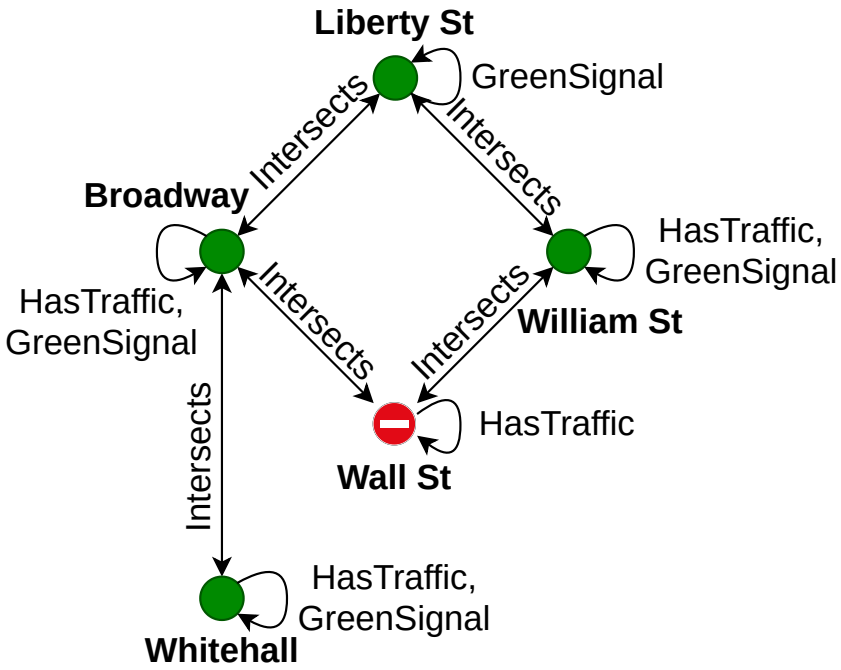


Figure 3.1: The induced *constant co-occurrence graph*, G_I. We would like to explain the accidents occurring on Broadway and Whitehall.

tuple `HasTraffic`(Whitehall) to obtain:

$$q_5: \quad \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x). \tag{3.1}$$

In contrast to q_1, this query only produces one undesirable tuple, namely, `Crashes`(William St).

Instead of directly enumerating candidate programs, our insight is to enumerate the subsets of the constant co-occurrence graph to generate candidates. Our algorithm tracks *enumeration contexts*: each such context is a set of input tuples obtained from a connected sub-graph of the co-occurrence graph G_I, and can be generalized into a candidate program by systematically replacing its constants with fresh variables.

Our main insight is that the only tuples which increase the specificity of an enumeration context are those which are directly adjacent to it in the co-occurrence graph. For example, consider context $C_5 = \{\, \texttt{GreenSignal}(\text{Whitehall}), \texttt{HasTraffic}(\text{Whitehall}) \,\}$ which produces the query q_5 in Equation 3.1.

Figure 3.2: Architecture of the EGS algorithm.

Observe in Figure 3.1 that there are exactly two tuples incident on C_5: $t = \texttt{Intersects}(\mathsf{Whitehall}, \mathsf{Broadway})$ and $t' = \texttt{Intersects}(\mathsf{Broadway}, \mathsf{Whitehall})$. We conclude that there are exactly two contexts which need to be enumerated as successors to C_5, namely: $C_6 = C_5 \cup \{t\}$ and $C_7 = C_5 \cup \{t'\}$. These contexts respectively produce the candidate queries:

$$q_6 : \quad \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x), \texttt{Intersects}(x, y), \text{ and}$$

$$q_7 : \quad \texttt{Crashes}(x) \text{ :- } \texttt{GreenSignal}(x), \texttt{HasTraffic}(x), \texttt{Intersects}(y, x).$$

The EGS algorithm repeatedly strengthens the enumeration context C with new tuples until it finds a solution program. For example, after five rounds of iterative strengthening, the context grows to include the tuples:

$$\begin{aligned} C = \{ \ & \texttt{GreenSignal}(\mathsf{Whitehall}), \texttt{HasTraffic}(\mathsf{Whitehall}), \\ & \texttt{Intersects}(\mathsf{Whitehall}, \mathsf{Broadway}), \\ & \texttt{GreenSignal}(\mathsf{Broadway}), \texttt{HasTraffic}(\mathsf{Broadway}) \ \}, \end{aligned} \tag{3.2}$$

which, when used to explain $\texttt{Crashes}(\mathsf{Whitehall})$, produces the desired solution in Equation 2.1.Figure 3.2 presents the overall architecture of the EGS algorithm. It maintains a set of enumeration contexts, organized as a priority queue, and repeatedly extends each of these contexts with a new tuple, in an example-guided manner. Each enumeration context can be naturally abstracted into a candidate query, as discussed in Section 2.2.2, and the procedure returns as soon as it finds an explanation which is consistent with the data. The priority function depends on both the size of the candidate program, and its accuracy on the training data, and we formally define it in Section 3.2.3.

Additionally, because the training data is finite, the co-occurrence graph is also finite, and therefore the EGS algorithm will eventually exhaust the space of enumeration contexts. At this point, Lemma 3.2.2 guarantees the non-existence of a program which is consistent with the training data,

thus proving the completeness of the synthesis procedure.

While the approach of iteratively strengthening candidate queries is similar to that followed by decision tree learning algorithms Quinlan (1986); Grzymala-Busse (1993), a notable difference is the presence of the queue in EGS, which holds alternative candidate explanations. The difference between the two algorithms is therefore similar to the difference between breadth-first search and greedy algorithms, with EGS being biased towards producing small candidate programs.

In our example, a syntax-guided prioritization would be forced to enumerate all programs with less than five joins, which induces an extremely large search space: SCYTHE takes approximately 16 seconds to find a consistent query and ILASP takes approximately 2 seconds, while the EGS algorithm returns in less than one second.

3.2. Example-Guided Synthesis Algorithm

In this section and the next, we formally describe the EGS algorithm for synthesizing relational queries. For ease of presentation, we first develop our core ideas for the case of a single desirable output tuple with a single column, $t = R(c)$. Given a set of input tuples I, the target tuple t, and a set of undesirable output tuples, the `ExplainCell` algorithm produces a query which is consistent with the example $(I, \{t\}, O^-)$. We extend this synthesis procedure to solve for multi-tuple multi-column output relations in Section 3.3.

The query is constructed by analyzing patterns of co-occurrence of constants in the examples, which we summarize using the *constant co-occurrence graph*. We first formalize this graph, and then introduce *enumeration contexts* as a mechanism to translate these patterns into relational queries. We conclude the section with a description of the `ExplainCell` procedure which searches for appropriate enumeration contexts using the co-occurrence graph.

3.2.1. The Constant Co-occurrence Graph

Recall that the data domain D is the set of all constants which appear in the input tuples $t \in I$. Then, the *constant co-occurrence graph*, $G_I = (D, E)$, is a graph whose vertices consist of constants

in D and with labeled edges E which are defined as:

$$E = \{c_i \xrightarrow{R} c_j \mid \text{input tuple } R(c_1, c_2, \ldots, c_k) \in I\}. \tag{3.3}$$

In other words, there is an edge $c \rightarrow^R c'$ iff there is a tuple t in the input relation R which simultaneously contains both constants c and c'. Observe that this makes each edge bi-directional. If constants c and c' occur in a tuple t, we say that t *witnesses* the edge $c \rightarrow^R c'$. The constant co-occurrence graph induced by the example of Figure 2.1b is shown in Figure 3.1.

The main insight is that patterns in the training data can be inferred by examining the co-occurrence relationships between constants. We express these patterns as subgraphs of the co-occurrence graph: as a consequence, the final `ExplainCell` procedure of Algorithm 1 reduces to the problem of enumerating subgraphs of G_I.

3.2.2. Enumeration Contexts

An *enumeration context* is a non-empty subset of input tuples, $C \subseteq I$. Equation 3.2 shows an example of an enumeration context. As Algorithm 1 explores G_I, it builds these contexts out of the tuples which witness each subsequent edge.

We can naturally translate a context $C = \{R_1(\vec{c}_1), R_2(\vec{c}_2), \ldots, R_n(\vec{c}_n)\}$ and an output tuple $t = R(\vec{c})$ into a conjunctive query $r_{C \mapsto t}$ as follows:

$$r_{C \mapsto t}: \quad R(\vec{v}) \text{ :- } R_1(\vec{v}_1), R_2(\vec{v}_2), \ldots, R_n(\vec{v}_n), \tag{3.4}$$

where the head $R(\vec{v})$ and body literals $R_i(\vec{v}_i)$ are obtained by consistently replacing the constants in the output tuple $t = R(\vec{c})$ and in the contributing input tuples $R(\vec{c}_i)$ with fresh variables v_c. We say that a context C *explains* a tuple t when the rule $r_{C \mapsto t}$ is consistent with $(I, \{t\}, O^-)$.

Recall from Section 2.2.2 that a rule may be instantiated by replacing its variables with constants, analogous to the process of specialization. In contrast, the procedure to obtain $r_{C \to t}$ from the context C and output tuple t may be viewed as a process of generalization. This correspondence between

enumeration contexts and rule instantiations allows us to state the following theorem:

Theorem 3.2.1. *Given an example $M = (I, \{t\}, O^-)$, there exists a context $C \subseteq I$ explaining t if and only if there exists a conjunctive query consistent with the example.*

Proof Sketch. Clearly, if context C explains t, then, by definition, $r_{C \mapsto t}$ is consistent with example M. Conversely, if there is a conjunctive query Q consistent with M, then let v be a valuation map deriving t in query Q. Then, consider the context $C \subseteq I$ to be the set of tuples that occur in the premise of the rule in Q when it is instantiated with v. Observe that $r_{C \mapsto t}$ is the rule in query Q and hence the context $C \subseteq I$ explains t. □

If a context C explains a tuple t and if $C \subseteq C'$, then C' also explains t. We can therefore apply Theorem 3.2.1 with the largest available context, $C = I$, i.e. the set of *all* input tuples, to prove the following lemma, which establishes the decidability of the relational query synthesis problem:

Lemma 3.2.2. *The given instance of the relational query synthesis problem $M = (I, \{t\}, O^-)$ admits a solution if and only if $r_{I \mapsto t}$ is consistent with M.*

3.2.3. Learning Conjunctive Queries

See Algorithm 1 for a description of the `ExplainCell` procedure, which forms the core of the EGS synthesis algorithm. See Figure 3.2 for a graphical description of its architecture.

The algorithm maintains a priority queue L of enumeration contexts and iteratively expands these contexts by drawing on adjacent tuples from the constant co-ocurrence graph G_I. It initializes this priority queue in Step 2, with all input tuples t' that contain the target concept c. In the case of our running example, to explain the tuple `Crashes`(Broadway), we would initialize L to $\{C_1, C_2, C_3, C_4\}$, with $C_1 = \{$`GreenSignal`(Broadway)$\}$, $C_2 = \{$`HasTraffic`(Broadway)$\}$, $C_3 = \{$`Intersects`(Whitehall, Broadway)$\}$, and $C_4 = \{$`Intersects`(Broadway, Whitehall)$\}$. These contexts result in the queries q_1–q_4 shown in Section 3.1. It subsequently iterates over the elements of L, and enqueues new contexts for later processing in Step 3(c)ii. In Step 3b, the algorithm returns the first enumeration context which is found to be consistent with the training

data.

Algorithm 1 `ExplainCell`$(I, R(c), O^-)$, where $t = R(c)$ is an output tuple with a single field. Produces an enumeration context $C \subseteq G_I$ such that $r_{C \mapsto t}$ is consistent with the example $(I, \{t\}, O^-)$.

1. Let $G_I = (D, E)$ be the constant co-occurrence graph.
2. Initialize the priority queue, L:

$$L \coloneqq \{\{t'\} \mid t' \in I \text{ contains the constant } c\}. \tag{3.5}$$

   Each element $C \in L$ is a subset of the input tuples, $C \subseteq I$.
3. While $L \neq \emptyset$:
   (a) Pick the highest priority element $C \in L$, and remove it from the queue: $L \coloneqq L \setminus \{C\}$.
   (b) If $r_{C \to t}$ is consistent with $(I, \{t\}, O^-)$, then return C.
   (c) Otherwise:
      i. Let $N = \{c \in D \mid \exists t' \in C \text{ where } t' \text{ contains } c\}$.
      ii. For each constant $c \in N$, edge $e = c \to^R c'$ in G_I, and for each additional input tuple $t' \in I \setminus C$ which witnesses e, update:

$$L \coloneqq L \cup \{C \cup \{t'\}\}.$$

4. Now, since $L = \emptyset$, return `unsat`.

A critical aspect of the `ExplainCell` algorithm is the priority function which arranges elements of the queue L. The EGS algorithm permits two choices for this priority function: We could consider the enumeration contexts in ascending order of their size, so that:

$$p_1(C) = -|C|.$$

This would guarantee the syntactically smallest solution which is consistent with the data. Alternatively, we could organize the enumeration contexts in lexicographic order of their *scores*, defined as the number of undesirable tuples eliminated per literal,

$$\text{score}(C) = \frac{|O^- \setminus [\![r_{C \mapsto t}]\!](I)|}{|C|},$$

and the size of the context, so that:

$$p_2(C) = (\text{score}(C), -|C|).$$

For example, the score of the contexts C_1 from Section 3.1 is 1.0 tuples/literal, as it eliminates one undesirable tuple, `Crashes`(Wall St), using one literal. On the other hand, the context C_3 does not eliminate any undesirable tuples, so that its score is 0. Similarly, the context C_5 eliminates two undesirable tuples, `Crashes`(Wall St) and `Crashes`(Liberty St) using two literals, therefore resulting in the score 1.0 tuples/literal. Therefore we have $p_2(C_1) > p_2(C_5) > p_2(C_3)$.

In this way, the priority function p_2 simultaneously prioritizes enumeration contexts with high explanatory power and small size, and is inspired by decision tree learning heuristics which greedily choose decision variables to maximize information gain. In practice, we have found this function p_2 to result in faster synthesis times than p_1 without incurring a significant increase in solution size, and we therefore use this function in our experiments in Section 3.4. We remark that the desired solution may not always be the smallest, and searching for small solutions can result in overfitting. We further discuss overfitting in Section 3.4.

After enumerating all possible contexts, if the algorithm has not found any context which explains the training data, Lemma 3.2.2 implies that the problem does not admit a solution. The following theorem formalizes this guarantee:

Theorem 3.2.3 (Completeness)**.** *Given an example $M = (I, \{t\}, O^-)$, where $t = R(c)$, `ExplainCell`(I, t, O^-) returns a context $C \subseteq I$ such that the query $r_{C \mapsto t}$ is consistent with $(I, \{t\}, O^-)$ if such a query exists, and returns `unsat` otherwise.*

Proof Sketch. In the first direction, if `ExplainCell`(I, t, O^-) returns a context C then, by construction, $r_{C \mapsto t}$ is consistent with $(I, \{t\}, O^-)$. To prove the converse, we assume for simplicity that the graph G_I is connected. If `ExplainCell`(I, t, O^-) returns `unsat`, then the last context considered in the loop in Step 3 must have been the set of all input tuples, $C = I$. From Lemma 3.2.2, it follows that the problem is unsolvable. □

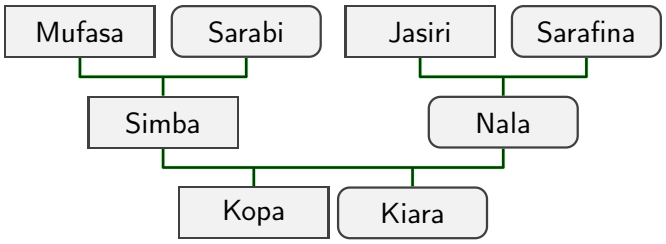


Figure 3.3: Example of a genealogy tree, used as training data to learn the programs $P_{\texttt{grandparent}}$ and $P_{\texttt{sibling}}$. Sarabi, Sarafina, Nala, and Kiara are female while Mufasa, Jasiri, Simba, and Kopa are male.

3.3. Extensions of the Synthesis Algorithm

In this section, we extend the central `ExplainCell` procedure described in Algorithm 1 with the ability to synthesize output relations of any arity and with any number of tuples, and also to synthesize queries which require negation.

As an example, we consider the problem of learning kinship relations from the training data in Figure 3.3. We have two binary (two column) input relations, `father` and `mother`, and we would like to learn queries which describe `grandparents` and `siblings`.

3.3.1. Multi-Column Outputs

In order to support multi-column outputs, we explain the fields of the tuple one at a time. Say the output table has k columns, and we wish to explain a tuple of the form $t = R(c_1, c_2, \ldots, c_k)$. We modify the `ExplainCell` procedure to synthesize explanatory contexts $C_1 \subseteq C_2 \subseteq \ldots \subseteq C_k \subseteq I$ such that each context C_i explains the first i fields of t, that is, they explain $t[1..i] = R_i(c_1, c_2, \ldots, c_i)$. We call this object the *i-slice* of t. We also refer to slices of entire relations such as $O^+[1..i]$ and $O^-[1..i]$ by lifting the slicing operation to sets of tuples in the natural manner.

For example, consider the task of learning the `grandparent` relation from the input data in Figure 3.3. Consider the output labels:

$$O^+ = \{\texttt{grandparent}(\mathsf{Sarabi}, \mathsf{Kiara})\}$$

$$O^- = \{\texttt{grandparent}(\mathsf{Sarabi}, \mathsf{Simba})\}$$

Then, in order to find a query consistent with $M = (I, O^+, O^-)$, we will first search for a context $C_1 \subseteq I$ which explains $t[1] = \texttt{grandparent}_1(\mathsf{Sarabi})$, and then grow it to C_2 which explains $t = t[1..2] = \texttt{grandparent}(\mathsf{Sarabi}, \mathsf{Kiara})$.

Observe that the negative examples also need to be *sliced* appropriately. In this example, the search for a context consistent with $(I, O^+[1], O^-[1])$ would fail since $O^+[1] = O^-[1] = \{\texttt{grandparent}_1(\mathsf{Sarabi})\}$, making this instance unrealizable. We therefore define the *forbidden i-slice*, F_i as the set of tuples $t_f = (c'_1, c'_2, \ldots, c'_i)$ of arity i such that every extension of t_f into a k-ary tuple, $t_e = (c'_1, c'_2, \ldots, c'_i, \ldots, c'_k)$, is destined to appear in O^-: $t_e \in O^-$. We achieve this by formally defining:

$$F_i = O^-[1..i] \setminus (U \setminus O^-)[1..i], \tag{3.6}$$

where $U = D^k$ is the set of all k-ary tuples over the data domain. In the `grandparent` example we have $F_1 = \emptyset$, resulting in the *sliced* example:

$$M_1 = (I, \{t[1]\}, F_1) = (I, \{\texttt{grandparent}_1(\mathsf{Sarabi})\}, \emptyset).$$

Now, we wish to find $C_1 \subseteq C_2 \subseteq I$ such that $r_{C_1 \mapsto t[1]}$ is consistent with M_1 and $r_{C_2 \mapsto t}$ is consistent with M. We can find C_1 by calling $\texttt{ExplainCell}(I, t[1], F_1)$, which will give us the result:

$$C_1 = \{\texttt{mother}(\mathsf{Sarabi}, \mathsf{Simba})\}.$$

To grow it to C_2, we modify the `ExplainCell` procedure to initialize the worklist L in Equation 3.5 as:

$$\begin{aligned} L &= \{C_1 \cup \{t\} \mid \forall t \in I \text{ containing } \mathsf{Kiara}\} \\ &= \{C_1 \cup \{\texttt{father}(\mathsf{Simba}, \mathsf{Kiara})\}, \\ &\qquad C_1 \cup \{\texttt{mother}(\mathsf{Nala}, \mathsf{Kiara})\}\}. \end{aligned}$$

More formally, we define the $\texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$ procedure by modifying the initialization step of Equation 3.5 so that:

$$L = \{C_{i-1} \cup \{t'\} \mid t' \in I \text{ contains } t[i]\}. \tag{3.7}$$

We then follow the same process to expand the subgraph one edge at a time, which in case of our running example produces the context:

$$C_2 = \{\texttt{mother}(\mathsf{Sarabi}, \mathsf{Simba}), \texttt{father}(\mathsf{Simba}, \mathsf{Kiara})\}$$

We formally present the `ExplainTuple` procedure in Algorithm 2. The completeness guarantee of Theorem 3.2.3 carries over as:

Lemma 3.3.1. *Given a context C_{i-1} which explains a sliced example $M_{i-1} = (I, \{t[1...(i-1)]\}, F_{i-1})$, $\texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$ returns a context $C_i \subseteq I$ such that the query $r_{C_i \mapsto t[1..i]}$ is consistent with $M = (I, \{t[1..i]\}, F_i)$ if such a query exists, and returns `unsat` otherwise.*

Algorithm 2 $\texttt{ExplainTuple}(I, t, O^-)$. Given a tuple t with arity $k \geq 1$, synthesizes a context C which is consistent with the example $(I, \{t\}, O^-)$.

1. Let $t = R(c_1, c_2, \ldots, c_k)$.
2. Initialize the context $C_0 = \emptyset$.
3. For $i \in \{1, 2, \ldots, k\}$, in order:
   (a) Construct the forbidden i-slice, F_i as in Equation 3.6.
   (b) Define $C_i = \texttt{ExplainCell}_{C_{i-1}}(I, t[1..i], F_i)$. If the procedure fails, return `unsat`.
4. Return C_k.

3.3.2. Unions of Conjunctive Queries

Observe that while the context generated above captures the concept:

$$\texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{father}(z, y),$$

the assumption of a single output tuple does not allow us to express the full grandparent relation (involving both *grandfather* and *grandmother* concepts). We therefore extend the tool to allow for multiple positive output tuples and extend the query language to support disjunctions, that is, we

now synthesize unions of conjunctive queries (UCQ). Suppose we are given:

$$O^+ = \{\texttt{grandparent}(\mathsf{Sarabi}, \mathsf{Kiara}), \texttt{grandparent}(\mathsf{Mufasa}, \mathsf{Kopa}),$$
$$\texttt{grandparent}(\mathsf{Jasiri}, \mathsf{Kopa}), \texttt{grandparent}(\mathsf{Sarafina}, \mathsf{Kiara})\}$$
$$O^- = \{\texttt{grandparent}(\mathsf{Mufasa}, \mathsf{Kiara}), \texttt{grandparent}(\mathsf{Sarafina}, \mathsf{Nala})\}$$

In order to find a UCQ consistent with $M = (I, O^+, O^-)$, we use a divide-and-conquer strategy: We separately synthesize a conjunctive query that explaining each desired tuple, and then construct their union. Because the rules are non-recursive, it follows that their union is consistent with the training data. In the running example, we get the following queries for each of the four tuples in O^+:

$$q_1: \quad \texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{father}(z, y).$$
$$q_2: \quad \texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{mother}(z, y).$$
$$q_3: \quad \texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{father}(z, y).$$
$$q_4: \quad \texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{mother}(z, y).$$

Observe that the UCQ with the rules $\{q_1, q_2, q_3, q_4\}$ is consistent with (I, O^+, O^-). This approach is similar to the technique used by EUSOLVER which first synthesizes small programs that conform to portions of the full specification, and then glues them together using conditional statements and case splitting operators provided by the target language Udupa et al. (2013).

In order to implement this procedure, we maintain a set of unexplained output tuples $O^?$, which is initialized to O^+, and repeatedly generate conjunctive queries explaining tuples $t \in O^?$ until all tuples are explained. We construct these conjunctive queries by invoking the `ExplainTuple` procedure of Section 3.3.1. We formally describe this process in Algorithm 3. Using the completeness guarantee of the `ExplainTuple` procedure, we have:

Lemma 3.3.2. *Given example $M = (I, O^+, O^-)$, `LearnUCQ`(I, O^+, O^-) returns a* union of conjunctive queries *Q consistent with M, if such a query exists, and returns `unsat` otherwise.*

Algorithm 3 EGS(I, O^+, O^-). Given an example $M = (I, O^+, O^-)$, finds a UCQ Q consistent with M if such a query exists, and returns `unsat` otherwise.

1. Initialize Q to be the empty query, $Q \coloneqq \emptyset$.
2. Initialize the set of still-unexplained tuples, $O^? \coloneqq O^+$.
3. While $O^?$ is non-empty:
   (a) Pick an arbitrary tuple $t \in O^?$.
   (b) Synthesize an explanation,

$$C_t = \texttt{ExplainTuple}(I, t, O^-),$$

   and construct $q_t = r_{C_t \to t}$.
   (c) If synthesis fails, return `unsat`.
   (d) Otherwise, update:

$$Q \coloneqq Q \cup \{q_t\}, \text{ and } O^? \coloneqq O^? \setminus [\![q_t]\!](I).$$

4. Return Q.

3.3.3. Negation

Finally, we extend the EGS algorithm to synthesize queries with negation. Similar to propositional formulas, UCQs also admit *negation normal forms*, where the negation operators are pushed down all the way to the individual literals. For example, a rule of the form:

$$r: \quad R(x, y, z) \text{ :- } \neg(R_1(x), R_2(y)), R_3(z).$$

can instead be written as the disjunction of two rules:

$$r_1: \quad R(x, y, z) \text{ :- } \neg R_1(x), R_3(z).$$
$$r_2: \quad R(x, y, z) \text{ :- } \neg R_2(y), R_3(z).$$

We therefore limit ourselves to learning UCQs in negation normal form. In our implementation, the user identifies input relation names that can possibly be negated in the final result. For an input relation name R of arity k, let $I(R)$ denote the set of tuples in I labeled with R. Given the data

domain D, we explicitly construct the negated relation $\neg R$ with the following tuples:

$$I(\neg R) = \{R(\vec{c}) \mid \vec{c} \in D^k \text{ and } R(\vec{c}) \notin I(R)\}.$$

We add $\neg R$ to the set of input relations and find a solution using Algorithm 3, exactly as before.

Consider, for example the task to learn the sibling relation from the training data in Figure 3.3. Suppose we are given:

$$O^+ = \{\texttt{sibling}(\mathsf{Kopa}, \mathsf{Kiara})\}$$
$$O^- = \{\texttt{sibling}(\mathsf{Kopa}, \mathsf{Kopa})\}.$$

We can show that no strictly positive program exists which can distinguish $\texttt{sibling}(\mathsf{Kopa}, \mathsf{Kiara})$ from $\texttt{sibling}(\mathsf{Kopa}, \mathsf{Kopa})$ as our hypothesis space does not support the inequality check, $\mathsf{Kopa} \neq \mathsf{Kiara}$. If we allow negation, a query consistent with (I, O^+, O^-) is:

$$\texttt{sibling}(x, y) \text{ :- } \texttt{mother}(z, x), \texttt{mother}(z, y), \neg(x = y).$$

We can encode the relation $\neg(x = y)$ using a two-column relation table that pairs unequal constants. We call this relation `neq`, and define it as:

$$I(\texttt{neq}) = \{(c, c') \in D^2 \mid c \neq c'\}.$$

With this additional input relation, EGS is able to solve for the desired concept in less than one second.

3.4. Experimental Evaluation

We have implemented the EGS algorithm in Scala comprising 2200 lines of code. In this section, we evaluate it to answer the following questions:

Q1: Performance: How effective is EGS on synthesis tasks from different domains in terms of synthesis time?

Q2: Quality of Programs: How do the programs synthesized by EGS measure qualitatively?

Q3: Unrealizability: How does EGS perform on synthesis tasks that do not admit a solution?

We present our benchmark suite in Section 3.4.1 and three baselines to compare EGS against in Section 3.4.2. We present our empirical findings for **Q1**–**Q3** in Sections 3.4.3–3.4.5.

We performed all experiments on a server running Ubuntu 18.04 LTS over the Linux kernel version 4.15.0. The server was equipped with an 18 core, 36 thread Xeon Gold 6154 CPU running at 3 GHz and with 394 GB of RAM. Note that EGS is single-threaded and is CPU-bound rather than memory-bound on all benchmarks. Therefore, similar results should be obtained on contemporary laptops and desktop workstations with similarly-clocked processors.

3.4.1. Benchmark Suite

We evaluate the EGS algorithm on a suite of 86 synthesis tasks. Of these, 79 admit a solution, meaning there exists a relational query which can perfectly explain their input-output examples. These 79 benchmarks are from three different domains: (*a*) knowledge discovery, (*b*) program analysis, and (*c*) database queries.

Knowledge discovery. These benchmarks comprise 20 tasks that involve synthesizing conjunctive queries and unions of conjunctive queries frequently used in the artificial intelligence and database literature.

Program analysis. These benchmarks comprise 18 tasks that involve synthesizing static analysis algorithms for imperative and object-oriented programs.

Database queries. These benchmarks comprise 41 tasks that involve synthesizing database queries. These tasks, originally from StackOverflow posts and textbook examples, are obtained from Scythe's benchmark suite Wang et al. (2017b).

There are seven additional benchmarks that do not admit a solution. We describe them in Section 3.4.5.

In each of the benchmark is provided with an exhaustive set of positive output tuples. The tuples not in the positive set are labelled negative. This data is provided upfront and not in an interactive fashion.

Table 3.1 presents characteristics of all 86 benchmarks, including the number of input-output relations, number of input-output tuples, and whether the intended programs involve disjunctions ($\vee$) or negations ($\neg$). In total, 17 tasks involve disjunctions while 9 of them involve negations.

3.4.2. Baselines

We compare EGS with three state-of-the-art synthesizers that use different synthesis techniques: Scythe Wang et al. (2017b), which uses enumerative search; ILASP Law et al. (2020b), which is based on constraint solving; and ProSynth Raghothaman et al. (2020b), which uses a hybrid approach by combining search with constraint solving.

ILASP and ProSynth phrase the synthesis problem as a search through a finite space of candidate rules. In order to evaluate them on our benchmark suite, we generated candidate rules for each benchmark using *mode declarations* in ILASP. A mode declaration is a syntactic restriction on the candidate rules such as the length of the rule, number of times a particular relation can occur, and the number of distinct variables used. In our experiments we only focus on the number of times an input relation occurs in a rule, and the number of distinct variables used. Providing a suitable set of mode declarations is a delicate balancing act: generous mode declarations can hurt scalability while insufficient mode declarations can result in insufficient candidate rules to synthesize the desired program. Given a query, one can recover the minimum mode declarations required to generate it. For example, for the program in Equation 2.1 in the running example, we have the mode declarations:

```
#modeb(2, GreenSignal(var(V)), (positive)).
#modeb(2, HasTraffic(var(V)), (positive)).
#modeb(1, Intersects(var(V),var(V)), (positive)).
```

Table 3.1: Benchmark characteristics. For each benchmark, we summarize the number of input-output relations, number of input-output tuples, and whether the intended programs involve disjunctions ($\vee$) or negations ($\neg$).

Name	Input		Output		Features
	#Relations	#Tuples	#Relations	#Tuples	
Knowledge Discovery					
abduce	2	12	1	8	$\vee$
adjacent-to-red	4	18	1	4	
agent	4	106	1	5	$\neg$
animals	9	50	4	17	
cliquer	1	4	1	4	
contains	2	14	1	4	
grandparent	2	8	1	7	$\neg$
graph-coloring	2	19	1	3	
headquarters	2	9	1	4	
inflammation	12	640	1	49	$\vee, \neg$
kinship	2	8	1	5	$\vee$
predecessor	1	9	1	9	
reduce	2	10	1	6	
scheduling	2	8	1	1	$\neg$
sequential	2	9	3	17	$\vee$
ship	3	15	1	5	
son	3	12	1	3	
traffic	3	18	1	2	
trains	12	223	1	5	
undirected-edge	1	3	1	5	$\vee$
Program Analysis					
arithmetic-error	3	11	1	1	
block-succ	3	21	1	1	
callsize	3	21	1	3	
cast-immutable	3	15	1	2	
downcast	5	89	4	175	$\neg$
increment-float	4	16	1	1	
int-field	3	9	1	1	
modifies-global	3	9	1	1	
mutual-recursion	1	13	1	3	
nested-loops	3	39	1	3	
overrides	2	6	1	1	
polysite	3	97	3	27	
pyfunc-mutable	3	19	1	2	
reach	3	17	1	2	
reaching-def	2	6	1	1	
realloc-misuse	3	18	1	1	
rvcheck	4	74	1	2	
shadowed-var	2	12	1	1	
Database Queries					
sql 1 $\sim$ 41	≤ 6	≤ 65	1	≤ 20	$\vee, \neg$
Unsynthesizable Benchmarks					
isomorphism	1	2	1	1	-
sql 42 $\sim$ 44	≤ 2	≤ 8	1	≤ 4	-
traffic-extra-output	3	18	1	3	-
traffic-missing-input	2	8	1	2	-
traffic-partial	3	11	1	1	-

```
#modeh(Crashes(var(V))).
#maxv(2).
```

This specifies for each candidate rule the output relation is `Crashes`, the input relations `GreenSignal` and `HasTraffic` occur at most twice, `Intersects` occurs at most once, and at most two distinct variables are used. This particular choice of modes generates 97 rules. Increasing the mode declarations results in a larger space of candidate rules. For our suite of benchmarks, we observed that a given input relation occurs in a rule at most thrice (such as in `sequential`), and the number of distinct variables in a single rule are at most 10 (as in `increment-float`). This allowed us to generate two set of candidate rules per benchmark:

1. *Task-Agnostic Rule Set*: Candidate rules where any given input relation occurs at most thrice and the number of distinct variables is at most 10, and

2. *Task-Specific Rule Set*: Candidate rules generated using the minimum mode declarations for the desired program.

With a threshold of 300 seconds, the candidate rule enumerator timed out when generating the task-agnostic rule set for 31 of the 79 benchmarks and the task-specific rule set for 2 benchmarks. We summarize the number of candidate rules generated per benchmark in Appendix A.

Similar to EGS, Scythe does not require a set of candidate rules, but the fragment of relational queries targeted by Scythe is SQL (with selection, join, projection, constant comparison, aggregation, and union). In order to compare the four tools fairly, we disable Scythe's support for aggregations. Also, Scythe supports complete labeling, that is every tuple either occurs in O^+ or O^-; therefore, we consider the set of negative examples O^- to be all tuples of appropriate arity that do not occur in O^+.

3.4.3. Q1: Performance

We ran EGS and the three baselines (with ProSynth and ILASP with two sets of candidate rules each) on all 79 benchmarks with a timeout of 300 seconds. We tabulate the results in Appendix A, and present a graphical summary in Figure 3.4.

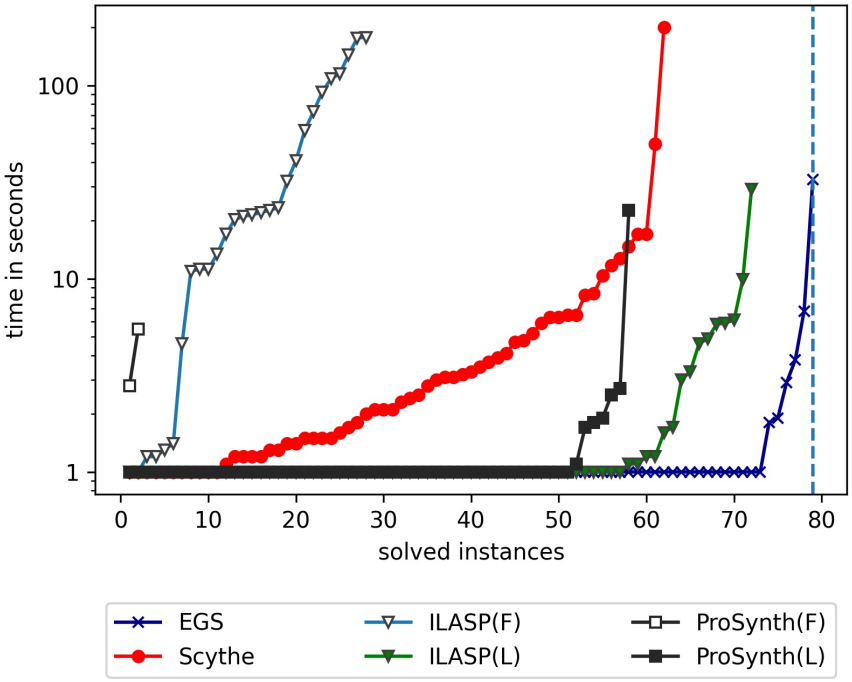


Figure 3.4: Results of our experiments using EGS, Scythe, ILASP, and ProSynth to solve a suite of 79 benchmarks. A datapoint (n, t) for a particular tool indicates that it solved n of the benchmarks in less than t time. Note EGS was the only tool to solve all 79 benchmarks. L and F refer to Task-Specific and Task Agnostic Rule Sets respectively.

EGS runs fastest with an average runtime of under a second and no timeouts. In fact, for all but 6 benchmarks, EGS returns a solution in less than one second, and never takes more than 33 seconds for any benchmark. Scythe takes an average of 7.6 seconds for 62 benchmarks and times out on 17 of the rest.

When provided with a task-specific rule set, both ILASP and ProSynth exhibit competitive performance on a subset of the benchmarks, and return a solution in less than one second for 57 and 51 benchmarks respectively. Still, their performance suffers on the more complicated benchmarks, and they exhibit timeouts on 7 and 21 of the 79 benchmarks, respectively. However, when provided with a task-agnostic rule set, the performance of both tools quickly degrades, and they timeout on 51 and 77 benchmarks, respectively.

All three baselines are disadvantaged by the enumeration required, and this causes EGS to outperform them, especially on benchmarks with larger numbers of input tuples, larger numbers of relations, or complex target queries. ProSynth and ILASP sometimes outperform EGS when provided with a task-specific choice of target rules on particularly simple benchmarks. However, we emphasize that, in all these cases, all three tools solve the problem in less than one second.

Notably, there are four benchmarks where EGS succeeds, but where all other tools time out: `animals`, `sequential`, `downcast`, and `polysite`. Upon examination, these benchmarks reveal the situations which cause the baseline techniques to underperform. For example, the `animals` benchmark involves classifying animals into their taxonomic classes based on their characteristics which are represented through 9 input relations. The larger number of input relations induces a complex search space causing Scythe to timeout. Furthermore, ILASP enumerates over 2000 candidate rules, even in the task-specific setting, causing both ILASP and ProSynth to also timeout.

3.4.4. Q2: Quality of Programs

We investigated the quality of the synthesized programs for each of the 79 benchmarks and observed that the program synthesized by EGS captures the target concept. For all but two cases, the programs generated by EGS also matched a program crafted by a human programmer. The two outliers

are `sequential` and `sql36`. In `sequential`, one of the tasks is to learn the `great-grandparent` relation. The desired program has eight rules (each representing a combination of the `mother` and `father` input relations to form rules of size three); however, we are provided with only two output tuples, and hence we learn a program with 2 rules that correctly explains the data. This can be fixed by adding more training data such that it covers all cases of the target concept. In case of `sql36`, the task involves comparing numbers; however, the input only includes the successor relation. The output of EGS therefore unfolds the greater-than relation using a four-way join of successors. While this is the smallest query that one can generate consistent with the examples, a more succinct query can be learned if we are provided an input table for the greater-than relation. In general, we observe overfitting when either there exists a program consistent with the input-output examples that is smaller than the desired program (as `sequential`) or when the training data does not represent all of the desired features of the target program (as in `sql36`). In general, one can overcome these cases by providing richer input-ouput examples.

One may also observe overfitting when our heuristic generates a consistent but larger program. This is possible as the priority function greedily optimizes over explanatory power and size simultaneously. We have not observed this case in any of our 78 benchmarks.

We also manually inspected the outputs of the baselines. The programs synthesized by ProSynth and ILASP are identical to ours in the cases when the tools terminate (in both, task-agnostic and task-specific rule sets). However, the programs synthesized by Scythe are neither small nor easy to generalize. In many cases, including knowledge discovery benchmarks such as `adjacent-to-red`, `graph-coloring`, and `scheduling`, we find the synthesized queries to be inscrutable. Appendix B compares the output of EGS and Scythe on these three benchmarks.

3.4.5. Q3: Unrealizability

To test the completeness guarantees provided by the EGS algorithm, we evaluated it on 7 unrealizable benchmarks. The results of these experiments are summarized in Table 3.2.

The first benchmark, `isomorphism`, is the simplest benchmark which does not admit a solution. In

Table 3.2: Unrealizable benchmarks. For each benchmark, we summarize runtimes on EGS and the three baselines. Note that Scythe overfits `sql42` and `traffic-partial` using operators like comparisons and negation.

# **Benchmark**	EGS	Scythe	ILASP	ProSynth
`isomorphism`	0.1	-	0.2	12.4
`sql42`	0.2	1.79	0.6	-
`sql43`	0.1	-	-	-
`sql44`	0.1	-	-	-
`traffic-extra-output`	0.2	-	0.2	0.1
`traffic-missing-input`	0.1	-	0.1	0.4
`traffic-partial`	59.5	2.33	0.2	1.5

this benchmark, we have the input $I = \{\texttt{edge}(\texttt{a}, \texttt{b}), \texttt{edge}(\texttt{b}, \texttt{a})\}$, and attempt to distinguish between the two vertices by specifying the outputs, $O^+ = \{\texttt{a}\}$ and $O^- = \{\texttt{b}\}$. From symmetry considerations, it follows that the benchmark does not admit a solution, and our algorithm successfully reports this in less than one second, while Scythe times out on this benchmark, and ILASP and ProSynth claim that there is no solution with respect to the given mode declarations.

We remark that while ILASP and ProSynth do not provide completeness guarantees like we do, Lemma 3.2.2 allows us to also strengthen their claims. Observe that as the input I has only two tuples, and any rule explaining the tuple needs at most one join. This can be used to construct an upper bound on the mode declaration which permits ILASP and ProSynth to also prove the unrealizability of the benchmark. However, as these mode declarations grow with the set I, we observe time outs in other unrealizable benchmarks.

The next three benchmarks `sql42`–`sql44` are sourced from the Scythe's benchmark suite, and involve some form of aggregation, which is unsupported by EGS. The task in `sql42` is to assign row numbers to the tuples, in `sql43` is to get the top two records grouped by a given parameter, and in `sql44` is to sum items using several IDs from another table. EGS proves the unrealizability of each of these tasks in less than a second. For `sql42`, Scythe produces an overfitting solution (using comparison operators) and ILASP proves the absence of a solution in less than a second. The mode declarations for these benchmarks were the same as that for the task-agnostic rule set.

The final three unrealizable benchmarks are modifications of the running example generated by

adding noise. In `traffic-extra-output` we have a constant in the output that does not occur in the input, in `traffic-missing-input` we do not provide the `Intersects` input relation, and in `traffic-partial` we remove certain input and output tuples which are essential to explain the crashes. While SCYTHE overfits a solution to `traffic-partial` using negation, EGS takes about a minute to prove that there cannot exist a solution which does not involve negation or aggregations.

CHAPTER 4

Synthesis of Queries with Comparison Operators

Unlike the queries discussed in Chapter 3, real world queries involve features beyond multi-column outputs, disjunction, and negation. In particular, queries for practical application in domains such as bioinformatics Seo (2018), big-data analytics Shkapsky et al. (2016), robotics Poole (1995), networking Loo et al. (2009), and program analysis Naik et al. (2021b) require comparison predicates.

Consider the example of university records of students taking courses, subjects students are majoring in, and departments in a university, as shown in Figure 4.1. Suppose the user intends to discover a concept that explains students 'Alice' and 'Bob', but excludes 'Charlie' and 'David'. The simplest explanation is that 'Alice' and 'Bob' take an *undergraduate* course (a course with ID less than 500) in the *Engineering* school. This explanation can be expressed as the SQL query shown in Figure 4.1c. The output examples, both positive and negative, are represented by entries from a single column (or a subset of columns) as in Figure 4.1b. We illustrate the two aforementioned challenges using this example.

Learning the join policy corresponds to learning the projections and joins that correspond to the `SELECT` and `FROM` clauses in Figure 4.1c. Search-based query synthesis techniques have made significant strides in learning relational queries over multiple tables by effectively enumerating the possible ways in which tables may be joined, thereby specializing in navigating the network of tables in a relational database. ILP techniques use language bias mechanisms such as mode declarations and meta-rules while program synthesis methods enumerate candidate programs using syntactic constraints or an explicit list of candidate rules to define a hypothesis space and explore the different key-foreign key pairs to join tables. Example-guided techniques rely on the underlying patterns in the data to discover them. All of these techniques struggle to address the challenge of synthesizing comparison predicates like `courseID` < 500 or `school` $=$ Engineering required by the target query.

Most query synthesis techniques require additional supervision from the user in the form of an exhaustive list of constants that may be used in comparisons. On the other hand, decision tree

registration		
studentID	deptCode	courseID
Alice	Comp.	201
Alice	Chem.	310
Alice	Mech.	550
Bob	Mech.	320
Bob	Mech.	550
Charlie	Chem.	310
David	Comp.	500
David	Mech.	502
Erin	Chem.	310

department	
deptCode	school
Chem.	Arts and Science
Comp.	Engineering
Math.	Arts and Science
Mech.	Engineering

major	
studentID	deptCode
Alice	Chem.
Bob	Comp.
Charlie	Math.
David	Chem.
Erin	Mech.

(a) Instances of tables `registration`, `department`, and `major` provided as the input relations I.

Positive Labels (O^+)
Alice
Bob

Negative Labels (O^-)
Charlie
David

(b) Labeled output examples O^+ and O^-.

```
SELECT registration.studentID
FROM registration JOIN department ON
    registration.deptCode = department.deptCode
WHERE registration.courseID < 500 AND
    department.school = "Engineering"
```

(c) The target SQL query Q_{EX}.

Figure 4.1: Example of a task to synthesize a relational query that takes instances of tables `registration`, `department`, and `major` (as in 4.1a) as input relations I, and outputs a set of student constants that contains all elements of O^+ and does not contain any elements in O^- (as in 4.1b). The query in 4.1c is a solution to this task.

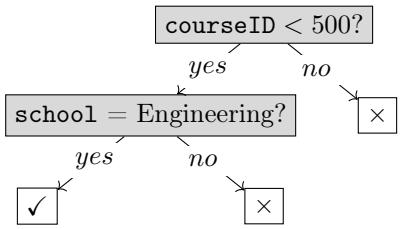
learning techniques Wu et al. (2007); Quinlan (1986) solve this problem in a limited setting in which the data is provided as a *single* table, where each row represents an instance of the input, and each column represents a feature of the instance Mitchell (1997); Kumar et al. (2016). In such a setting, the labeling is given by a partition of the instances into *positive* and *negative* examples. These techniques then construct classifiers using greedy information gain heuristics to search for and combine locally optimal comparison predicates.

However, using these techniques requires additional user supervision to produce the single table they take as input. This is typically done by performing key-foreign key joins to construct such a single table and then applying a feature selection method Guyon et al. (2006). For example, the user would have to provide the table from Figure 4.2a to the learning algorithm, along with the correct labels for each row, to obtain the decision tree in Figure 4.2b that corresponds to the `WHERE` clause in the target query. To obtain this table, the user must manually join the `registration` and `department` tables over the `deptCode` column. This process is tedious and prodigal as it requires a careful analysis of the relational database, and errors in the process can introduce data redundancy and impact the efficiency of the learning algorithm. Manually doing so also becomes intractable for databases with several tables, necessitating the automation of this process. Additionally, the user must provide accurate labels to obtain the correct decision tree, though there is no clear way to do so given only the output examples.

We thus observe a dichotomy of existing techniques — they either support multi-table databases (as with search-based relational query synthesis) or excel at learning comparison predicates (as in the case of decision trees), but not both. We leverage the strengths of the two paradigms to design an end-to-end algorithm for the synthesis of relational queries that feature both comparison predicates and joins across multiple tables. Specifically, we focus on the class of *select-project-join* (SPJ) queries like the one in Figure 4.1c which constitute an important fragment of relational algebra Imieliński and Lipski (1984). These queries feature *equi-joins*, that is joins across tables parameterized by a set of columns (with matching types), as well as categorical and numerical comparisons for selections.

studentID	deptCode	courseID	school	label
Alice	Comp.	201	Engineering	✓
Alice	Chem.	310	Arts and Science	×
Alice	Mech.	550	Engineering	×
Bob	Mech.	320	Engineering	✓
Bob	Mech.	550	Engineering	×
Charlie	Chem.	310	Arts and Science	×
David	Comp.	500	Engineering	×
David	Mech.	502	Engineering	×
Erin	Chem.	310	Arts and Science	?

(a) The result of joining `registration` and `department` over the `deptCode` columns of each table.



(b) A decision tree for classifying rows of the table in Figure 4.2a. It can be flattened to a Boolean formula (`courseID` < 500) $\wedge$ (`school` = Engineering).

Figure 4.2: Each candidate join can be translated to a single table. The table in 4.2a represents the join of `registration` and `department` tables. The `label` column denotes the ideal labels which result in learning the decision tree in Figure 4.2b. The user can annotate the rows of this table as positive (✓) or negative (×) to support decision tree learning. On running a decision tree algorithm on it, we get the tree in Figure 4.2b.

Our key insight is to interpret the query synthesis problem as a search across a two-dimensional space defined by comparison predicates on one side and candidate joins on the other. To efficiently search through this space, we introduce an interleaved approach that allows us to leverage the strengths of both decision tree learning and search-based synthesis.

This interleaved approach seeks to address the challenge of finding the optimal join policy by enumerating different projection and join policies as partial queries, each producing a single intermediate table over which a decision tree could be learned to generate the comparison predicates for the target query. We use the example-guided search strategy Thakkar et al. (2021) to enumerate these queries as it prioritize joins with fewer tables, thus synthesizing queries that contain only sufficient and necessary features from the database.

Once we generate a candidate single intermediate table, the next step is to synthesize comparison predicates. Classical decision tree learning techniques require the rows of the intermediate table to be labeled, while in our setting, only certain constants (or tuples of constants) are labeled. There is no straightforward way to handle this discrepancy without additional user supervision. We fundamentally modify the classical decision tree learning algorithm ID3 Quinlan (1986) by changing the definition of entropy and information gain used by it and present it in Section 4.1.2.

Together, the search for candidate joins and the modified decision tree learning procedure can work in tandem to synthesize the relational queries with categorical and numerical comparisons. In practice, our algorithm synthesizes queries that are general (that is, it does not overfit the data) and of minimal size. We also prove the completeness of the algorithm—it synthesizes a query consistent with the training data *iff* there exists such a query. We implement this interleaved approach as Libra, and evaluate it on a benchmark suite of 1,475 instances of SPJ queries from the Spider Yu et al. (2018) and Geography Finegan-Dollak et al. (2018); Zelle and Mooney (1996); Iyer et al. (2017) datasets over 160 different databases, each with multiple tables. Libra solves 1,361 of these instances with a timeout of 10 minutes per task, and takes 58.9 seconds on average per instance. We also compare with state-of-the-art tools Scythe and PatSQL that can synthesize select-project-join queries. They can solve 195 and 673 instances, and take 139.50 and 23.13 seconds on average per task respectively.

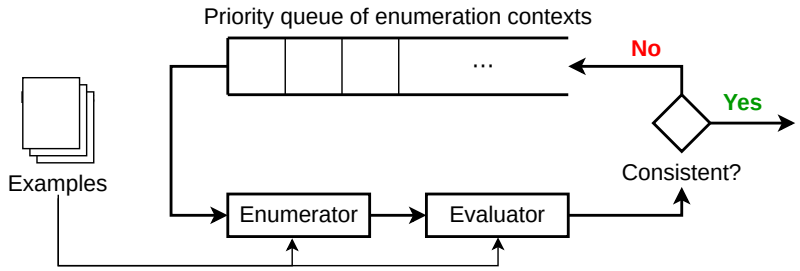


Figure 4.3: Architecture of the LIBRA algorithm. The algorithm interleaves decision tree learning of comparison predicates with example-guided enumeration of candidate joins. Throughout, we maintain the size of the program and check against this size to ensure that the synthesized query is minimal among all consistent queries (subject to optimality of decision tree learning).

All the benchmarks solved by the baselines are also solved by at least one instance of our framework, and our framework additionally solves a significantly larger set of the total benchmarks which the baselines fail to solve.

4.1. Algorithm

In this section we describe the end-to-end LIBRA algorithm, which takes input-output examples $E = (I, O^+, O^-)$ as input and returns a relational query Q consistent with E. Algorithm 4 summarises the procedure and Figure 4.3 presents its architecture.

We start with an example-guided search to construct a partial query with projection and joins (and without the comparisons for the selection operator). This partial query is constructed by analyzing patterns of co-occurrence of constants in the input-output examples Section 4.1.1 formalizes these patterns as *enumeration contexts* that can be translate into partial queries with projection and join operators. These correspond to the step 2 of the algorithm.

To synthesize categorical and numerical comparisons for the selection operator we turn to supervised learning. We maintain the enumeration contexts in a priority queue L ordered by increasing size. For each context C in L, we convert C into a single table T_c through a join of the input relation tables that occur in C. This is followed by the modified decision tree learning procedure (DTL) which completes the query. This corresponds to step 4a-4c.

In step 4d, we expand the context by one tuple. This corresponds to considering a join with an

Algorithm 4 LIBRA(I, O^+, O^-), where I is the set of input tuples, and O^+ and O^- are the sets of positively and negatively labeled output tuples respectively.

1. Set $ans =$ `unsat` and $N = \infty$.
2. For an arbitrary $t \in O^+$, let $\mathcal{C}_t$ be the initial contexts that explain t as defined in Equation 4.1.
3. Initialize the priority queue as $L = \mathcal{C}_t$.
4. While L is non-empty:
   (a) Pick the smallest size element $C \in L$, and remove it from the queue: $L := L \backslash \{C\}$.
   (b) If $|C| > N$, exit the loop and go to Step 5.
   (c) For each table T_C constructed using Equation 4.4:
      i. Let $\top$ be a node of an empty decision tree. Run DTL$(T_C, \top, O^+, O^-)$.
      ii. If DTL$(T_C, \top, O^+, O^-)$ returns a tree Δ such that $|\Delta| + |C| \leq N$ and entropy of $|\Delta|$ is 0,
         A. Set $ans = Q(T_C, \Delta)$ as defined in Equation 4.6.
         B. Set $N = |C| + |\Delta|$.
   (d) For each tuple t' that shares a constant with a tuple in C, update:

$$L = L \cup \{C \cup \{t'\}\}.$$

5. Return *ans*.

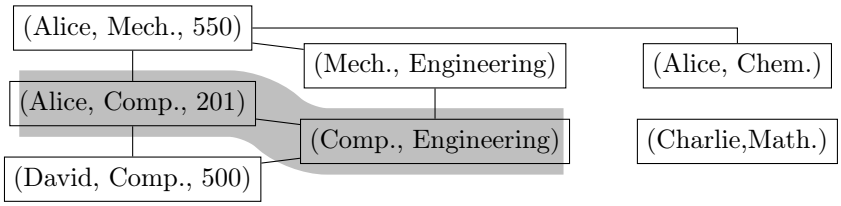


Figure 4.4: A collection of rows of the input table I. Two rows are shown connected with an edge if they share a constant. The shaded part represents a context $C \subseteq I$ which corresponds to the join in Equation 4.2.

additional table. Therefore, the steps 4a-4c explore comparison predicates and step 4d explores joins. Through the two, we search through the two-dimensional search space.

Throughout the algorithm, we maintain the size of the query Q as variable N and guarantee that the query Q has minimal size among all queries consistent with the input-output example (subject to the optimality of the decision tree). Additionally, if no such query exists, the algorithm terminates and returns `unsat`. We prove this completeness result in Theorem 4.1.1.

4.1.1. Example-Guided Enumeration of Projection and Joins

An *enumeration context* is a non-empty subset of the input tuples, $C \subseteq I$. The shaded part of Figure 4.4 corresponds to the context $C = \{(\text{Alice, Comp., 201}), (\text{Comp.,Engineering})\}$. An enumeration context $C \subseteq I$ is said to explain a tuple $t \in O^+$ when for each column c of t, there is a tuple $t_c \in C$ such that for some column c' in t_c, we have $t.c = t_c.c'$.

Given a tuple $t \in O^+$, we construct the initial set of enumeration contexts for step 2 of Algorithm 4 by considering enumeration contexts:

$$\begin{aligned} \mathcal{C}_t = \{\{t_c : \ &\text{for each column } c \text{ of } t, \\ &\text{there exists a column } c' \text{ of } t_c \text{ such that } t.c = t_c.c'\}\} \end{aligned} \tag{4.1}$$

In step 4d we extend a context C by adding a tuple t' such that for some $t \in C$, there is an edge $t \to t' \in E$ with appropriate labels. One can translate a context $C = \{t_1, \ldots, t_n\}$ and an output tuple t into a partial query with projection and joins.

Consider the output 'Alice' that is explained by the context

$$C = \{(\text{Alice, Comp., 201}), (\text{Comp., Engineering})\}$$

. This can be translated to the query:

```
SELECT registration.studentID
    FROM registration JOIN department
        ON registration.deptCode = department.deptCode
```
(4.2)

This is because the constant 'Alice' occurs in the column `studentID` of `registration`, and the tuples from the tables `registration` and `department` share a constant for the columns `department.deptCode` and `registration.deptCode`. In general, given an output tuple t and a context $C \subseteq I$, we first consider the sequence of columns $(T_{\pi_1}.c_{\pi_1}, \ldots T_{\pi_k}.c_{\pi_k})$ from where we get the

constants in t. These will correspond to the columns for the projection operator. For each tuple t_i in C, we consider the table T_i. These will correspond to the tables to be joined. In order to construct the parameters for the join, for each $t_i \in C$, let E_i be the set of predicates of the form $(t_i.c = t'.c')$ where $t' \in \{t_1, \ldots, t_{i-1}\}$. Then, we can construct the queries of the form:

$$\begin{aligned} &\texttt{SELECT } (T_{\pi_1}.c_{\pi_1}, \ldots T_{\pi_k}.c_{\pi_k}) \\ &\quad \texttt{FROM } (\ldots (T_1 \texttt{ JOIN } T_2 \texttt{ ON } \theta_2) \ \ldots \texttt{JOIN } T_n \texttt{ ON } \theta_n) \end{aligned} \tag{4.3}$$

Where each θ_i is a conjunction of comparison predicates that label edges in E_i. We consider all possible non-empty subsets of the labels in E_i, and therefore, for each pair (t, C) of output tuple and context, there may be multiple candidate joins.

4.1.2. Supervised Learning of Comparisons for Selection

We now turn to decision trees to add a selection operator to the query in Equation 4.3. Our approach is motivated by the Iterative Dichotomiser 3 (ID3) algorithm for learning decision trees Quinlan (1986). We first need to convert the tables $T_1, \ldots, T_n$ in context C into one single table T_C. This is achieved by implementing the join in Equation 4.3, that is we consider the output of the SQL query:

$$\texttt{SELECT } * \texttt{ FROM } (\ldots (T_1 \texttt{ JOIN } T_2 \texttt{ ON } \theta_2) \ \ldots \texttt{JOIN } T_n \texttt{ ON } \theta_n) \tag{4.4}$$

This join produces a single table. As before, each context corresponds to multiple joins and hence there are multiple candidates for T_C. We consider all of them in our search.

We start by introducing some notation. Let γ be the schema of the output tuples, that is the types of the columns from which we draw output tuples. Let $\pi_\gamma(T_C)$ represent the projection of T_C to the columns in γ. Then, the entropy of a node N is defined as:

$$\begin{aligned} p &= P(O^+ | \pi_\gamma(T_C), O^+ \cup O^-) = \frac{|\pi_\gamma(T_C) \cap O^+|}{|\pi_\gamma(T_C) \cap (O^+ \cup O^-)|} \\ n &= P(O^- | \pi_\gamma(T_C), O^+ \cup O^-) = \frac{|\pi_\gamma(T_C) \cap O^-|}{|\pi_\gamma(T_C) \cap (O^+ \cup O^-)|} \end{aligned}$$

T_C with (school = Engineering)			
studentID	deptCode	courseID	school
Alice	Comp.	201	Engineering
Bob	Mech.	320	Engineering
Alice	Mech.	550	Engineering
Bob	Mech.	550	Engineering
David	Comp.	500	Engineering
David	Mech.	502	Engineering

(a) Table with rows of T_C that satisfy the predicate (school = Engineering).

T_C with ¬(school = Engineering)			
studentID	deptCode	courseID	school
Alice	Chem.	310	Arts and Science
Charlie	Chem.	310	Arts and Science
Erin	Chem.	310	Arts and Science

(b) Table with rows of T_C that do not satisfy the predicate (school = Engineering).

Figure 4.5: In order to compute the information gain of a comparison predicate at a given node, we split the rows at the node into two parts, those that satisfy the predicate and the others that don't. Here, we have split the joined table T_C (from Figure 4.2a) on the predicate (school = Engineering).

$$S(N) = -\left(p \log_2 p + n \log_2 n\right)$$

Here, we restrict our analysis to output tuples that occur in O^+ or O^- only. Consider the joined table in Figure 4.2a. We can compute the entropy of the node with label (school = Engineering):

$$p = P(\{Alice, Bob\} | \{Alice, Bob, Charlie, David\}) = \frac{1}{2}$$
$$n = P(\{Charlie, David\} | \{Alice, Bob, Charlie, David\}) = \frac{1}{2}$$
$$S(N) = -\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right) = 1$$

Here, we do not consider 'Erin.' This is our first concrete modification to the decision tree learning algorithm.

A comparison predicate a splits the table T_C into two: $\sigma_a(T_C)$ which comprises of rows that satisfy a and $\sigma_{\neg a}(T_C)$ which comprises of rows that do not satisfy a. Let $\sigma_a(T_C)$ correspond to a node L and $\sigma_{\neg a}(T_C)$ correspond to a node R. Then we can compute their entropies $S(L)$ and $S(R)$ just as above. Consider the predicate (school = Engineering) which splits the joined table in Figure 4.2a into two tables as shown in Figure 4.5. Let $\sigma_{(\texttt{school=Engineering})}(T_C)$ form node L and $\sigma_{\neg(\texttt{school=Engineering})}(T_C)$ form node R. The entropies $S(L)$ and $S(R)$ are 0.918 and 1 respectively.

We can now compute the *information gain*. Information gain is defined as the difference between the

entropy of the node and the weighted sum of the entropy of its children. That is, the information gain at node N is of the form:

$$IG(N) = S(N) - (\alpha S(L) + \beta S(R))$$

where $\alpha + \beta = 1$. In a classical setting, the coefficients α and β are the ratio of the number of rows corresponding to the child nodes L and R. In our study, we focus on projection, and only the tuples in O^+ and O^-. For ease of notation, let $|\pi_\gamma(\sigma_a(T_C)) \cap (O^+ \cup O^-)|$, the number of rows in $\sigma_a(T)$, projected to columns γ, that occur in either O^+ or O^- be λ_a (and analogously for $\sigma_{\neg a}(T_C)$ be $\lambda_{\neg a}$). Then information gain at Node N with comparison predicate a is defined as:

$$IG(N, a) = S(N) - \left(\frac{\lambda_a}{\lambda_a + \lambda_{\neg a}} S(L) + \frac{\lambda_{\neg a}}{\lambda_a + \lambda_{\neg a}} S(R) \right) \tag{4.5}$$

In our running example, λ_a is 3 and $\lambda_{\neg a}$ is 2. This gives us an information gain of 0.0328. The change in the weighted sum is our second concrete modification to decision tree learning.

The decision tree learning algorithm as described in Algorithm 5 starts with the table T and node N as an input. We introduce node N so we can call this procedure recursively. If O^+ or O^- is empty, we return the trivial tree with N as the only node. Otherwise, we construct a set of comparison predicates of the form $(T.c \backsim k)$, where $T.c$ is a column of the table T, k is a constant that occurs in the column $T.c$, and $\backsim$ is a comparison operator (in our case either $=$, $<$, or $\leq$. Then, similar to the classical algorithm, we pick a comparison predicate a that maximizes the information gain $IG(N, a)$. If no comparison predicate can maximize the information gain beyond 0, we return the trivial tree with N as the only node, labeled with '?' and terminate the process. This is the case where there is no classifier for the given input data.

Otherwise, we split the table T on predicate a as tables $\sigma_a(T)$ and $\sigma_{\neg a}(T)$, introduce child nodes L and R corresponding to them, and call the DTL process recursively on the children of N. When we call DTL on the L and R nodes, we ensure that the O^+ and O^- are updated to the output tuples that occur in $\sigma_a(T)$ and $\sigma_{\neg a}(T)$.

Algorithm 5 DTL(T, N, O^+, O^-), where T is a table, N is a node, and O^+ and O^- are the sets of positively and negatively labeled tuples respectively.

1. If O^+ is empty, label N with $\times$, return the leaf node N, and terminate.
2. If O^- is empty, label N with $\checkmark$, return the leaf node N, and terminate.
3. Otherwise, let $A = \{\}$.
4. For each column c in T,
   (a) if c is of the `categorical` type, then for each constant k in column c, update:
   $$A = A \cup \{(T.c = k)\}$$
   (b) if c is of the `numerical` type, then for each constant k in column c, update:
   $$A = A \cup \{(T.c < k), (T.c \leq k)\}$$
5. For each $a \in A$, compute $IG(N, a)$ using the formula in Equation 4.5.
6. Find a predicate a for which $IG(N, a)$ is maximum. If the maximum for $IG(N, a)$ is 0, label N as ?, return the leaf node N, and terminate the process.
7. Otherwise, label N with predicate a and create new nodes L and R as left child and right child of N respectively.
8. Recursively compute:
   $$\Delta_L = \text{DTL}(\sigma_a(T), L, O^+ \cap \pi_\gamma(\sigma_a(T)), O^- \cap \pi_\gamma(\sigma_a(T))) \text{ and}$$
   $$\Delta_R = \text{DTL}(\sigma_{\neg a}(T), R, O^+ \cap \pi_\gamma(\sigma_{\neg a}(T)), O^- \cap \pi_\gamma(\sigma_{\neg a}(T))),$$
   where γ is the sequence of projected columns for the output.
9. Return the tree with root node N, left sub-tree Δ_L and right sub-tree Δ_R.

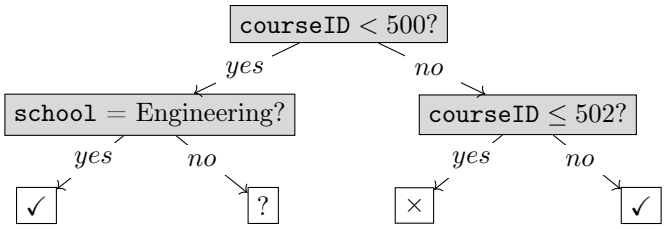


Figure 4.6: The decision tree generated by the process DTL on T_C (from Figure 4.2a) with $O^+ = \{\text{Alice}, \text{Bob}\}$ and $O^- = \{\text{Charlie}, \text{David}\}$.

On executing the DTL procedure on our running algorithm, we get a tree as in Figure 4.6. Observe that it has a redundant right subtree, and one of the leaves is labeled '?'. Instead, the desired tree is the one in Figure 4.2b.

The problem of finding a minimal decision tree, or even approximating it, is NP-complete Sieling (2008). Therefore we opt for a greedy search that is computationally efficient. Instead of considering all possible Boolean combinations of comparison predicates, Algorithm 5 makes *locally optimal* decisions, it enabling it to handle large data-sets efficiently and produces satisfactory results in practice. While it is possible that locally optimal choices may not lead to the smallest decision tree, it most often leads to good enough solution that is succinct and general, as observed in Section 4.2 The soundness check of Algorithm 4 also ensures that while DTL may generate a larger tree, the synthesized query will always be consistent with the given input-output examples. Greedy heuristics based on information gain commonly used in decision tree learning and search algorithms for this reason Quinlan (1986); Su and Zhang (2006); Suthaharan (2016).

By using the greedy heuristic, DTL generates a *perfect separator* between O^+ and O^-, however, we only need a *partial separator*. That is, we seek a relational query Q that captures some derivation for each tuple in O^+, and no derivation for any tuple in O^-. We do not have a stronger requirement that Q should capture all derivations for tuples in O^+. On the other hand, the decision tree attempts to branch till every node is at entropy 0, that is every node either leads to tuples in O^+ or O^- exclusively, instead of stopping when there is at least one leaf node corresponding to every tuple in O^+. As our setting allows for a weaker notion of separation, we can further trim the decision trees.

More concretely, the right branch of the root node in the tree in Figure 4.2b corresponds to rows with `studentID` values in {Alice, Bob David}, as all three of them are taking courses with `courseID` ≥ 500. DTL naturally assumes that one needs to branch further to separate Alice and Bob from David. However, it is not necessary as the node labeled ✓can explain Alice and Bob. Similarly, at the leaf labeled '?,' the projected column has values $\{\text{Alice}, \text{Charlie}\}$, and we do not have any comparison predicate that separates them.

We implement an *lazy* version of DTL to achieve the trimmed decision trees. This is our third modification to classical decision tree learning. In the DTL process, we introduce a set of *unexplained output tuples* $O^?$, initialized to O^+ and a first-in-first-out (FIFO) queue that maintains a list of nodes, initialized to $\{N\}$. Throughout the algorithm, we update $O^?$ by removing the output tuples that are already explained by a particular leaf of the decision tree. While there exist any unexplained tuples, we dequeue a node from the queue and branch it out as described in Algorithm 5. Instead of calling the process recursively, we enqueue the children and then eventually get to them only when there are unexplained tuples. This lazy evaluation allows us to generate smaller trees with fewer redundancies. With this modification, we get the desired tree depicted in Figure 4.2b.

In summary, the three modifications allow us to adapt decision tree learning to our setting. Additionally, these modifications do not compromise any guarantees about termination of the procedure or size of the learned decision tree Mitchell (1997).

4.1.3. Interleaving Decision Tree Learning with Example-Guided Search for Joins

A decision tree Δ can then be converted to a boolean formula σ_Δ in disjunctive normal form. For each leaf of the tree that is marked ✓, we consider a clause that is composed of the conjunction of the predicate at its parent node (if the node is a left child, and the negation of the predicate otherwise). And then, we construct the disjunction of each of these clauses. For example, we can translate the tree in Figure 4.2b to the formula (`courseID` < 500) $\wedge$ (`school` = Engineering). The negations, if any, can be removed by considering the negated comparison operators ($\neq$, $>$, and $\geq$).

Therefore we can convert a joined table T_C and decision tree Δ into a query $Q(T_C, \Delta)$ by using the

boolean formula σ_Δ to complete the query in Equation 4.3. This gives us the query:

$$\begin{aligned} &\texttt{SELECT}\ (T.c_1, \ldots, T.c_k) \\ &\quad \texttt{FROM}\ (\ldots (T_1\ \texttt{JOIN}\ T_2\ \texttt{ON}\ \theta_1)\ \ldots \texttt{JOIN}\ T_n\ \texttt{ON}\ \theta_{n-1}) \\ &\quad\quad \texttt{WHERE}\ \sigma_\Delta \end{aligned} \tag{4.6}$$

Figure 4.3 summarizes Libra. The end-to-end algorithm guarantees completeness:

Theorem 4.1.1 (Completeness)**.** *If there exists a relational query consistent with the input-output example $E = (I, O^+, O^-)$, then* Libra *produces a query Q consistent with E.*

The proof of this theorem relies on the completeness of the example-guided enumeration and the completeness of decision tree learning. We assume the reader is familiar with the analogous guarantees for example-guided synthesis of conjunctive queries Thakkar et al. (2021), and those for classical decision trees. Observe that if a context C explains a tuple t, then all contexts $C' \supseteq C$, also explain t. We can consider the largest context $C = I$, that is, the set of all input tuples, to prove the following lemma:

Lemma 4.1.2. *If there exists a relational query consistent with the input-output example $E = (I, O^+, O^-)$, then there exist a decision tree Δ with predicates of the form $(T.c \backsim k)$ where T is an input table, c is a column of T, and k is a constant in the column c, such that the query $Q(T_I, \Delta)$ is consistent with E.*

It follows from the completeness of example-guided enumeration that a decision tree must exist, however, it remains to show that the predicates for the decision tree must be of the said form. Without loss of generality, suppose the comparison predicate is of the form $(T.c > k_1)$, where k_1 does not occur in c. The arguments for other comparison operators is analogous. Observe that must exist the greatest lower bound k_2 of k_1 in c (that is, $k_2 = \max\{k \in c : k < k_1\}$). Replacing the predicate by $(T.c > k_2)$ does not change the semantics of the query with respect to input I, as there are no constants in between k_1 and k_2. By systematically replacing the predicates in a query consistent with E, we can prove that there must exist a query where the selection operator

corresponds to a decision tree of the said form. As we exhaustively enumerate all possible predicates, we can guarantee:

Lemma 4.1.3. *Given a table T, a node N and output tuples partitioned as O^+ and O^-, if there exists a decision tree that separates O^+ from O^-, then* $\mathrm{DTL}(T, N, O^+, O^-)$ *will return such a tree.*

Together, Lemma 4.1.2 and Lemma 4.1.3 can prove Theorem 4.1.1.

Additionally, observe that at each step of the algorithm, we maintain the constant N that tracks the size of the query. As the contexts are maintained in increasing order of size, the number of joins in the enumerated queries is always increasing.

4.2. Evaluation

We have implemented the Libra algorithm in Scala. In this section, we evaluate it on a large-scale benchmark suite. First, we measure the performance of our algorithm compared to state-of-the-art synthesis tools. We do so by comparing the number of instances solved by each tool and the time taken by each tool to do so. Next, we evaluate the generality of the solutions generated by each tool. To do so at scale, we leverage Occam's razor to use the succinctness of a query as a proxy of how specific a query is to the training data. As such, we propose to answer two main research questions through this evaluation:

Q1. Performance: How effective is Libra on synthesis tasks from different domains in terms of synthesis time?

Q2. Succinctness: How large are the programs synthesized by Libra compared to the reference solution?

We discuss our benchmark suite in Section 4.2.1 and the baselines against which we compare Libra in Section 4.2.2 along with the setup for each. We present our findings for **Q1** and **Q2** in Sections 4.2.3 and 4.2.4. We performed all experiments on a Linux server equipped with an 18-core, 36-thread Xeon Gold 6154 CPU running at 3 GHz and with 394 GB of RAM.

4.2.1. Benchmarks

We evaluate LIBRA on the set of all SPJ instances from the SPIDER Yu et al. (2018) and GEOGRAPHY Finegan-Dollak et al. (2018); Zelle and Mooney (1996); Iyer et al. (2017) datasets. SPIDER is an open-access large-scale manually annotated dataset. There are 1,203 SPJ instances in the SPIDER dataset over 159 databases. On the other hand, GEOGRAPHY is a dataset of SQL queries about US geography. We use version 4 of the modified SQL dataset for GEOGRAPHY from Finegan-Dollak et al. (2018); Zelle and Mooney (1996); Iyer et al. (2017). Upon deduplication of the queries, we extract 272 SPJ instances, all over the same database, giving us a total of 1,475 instances over both datasets. For each benchmark, we consider the tables from its corresponding database as the input tables and the result of running the ground truth query over that database as the output table.

Each benchmark has 2 to 26 input tables (with a median of 8), each with 1 to 352 columns (with a median of 30), containing 8 to around 553k tuples in the input tables (with a median of 937). Additionally, each benchmark is labeled with a *ground truth* query that serves as a reference solution. This reference solution is used to obtain the output examples for the corresponding benchmark. Overall, the reference solutions feature a join of at most 6 tables and the use of at most 3 predicates.

4.2.2. Baselines and Setup

We evaluate LIBRA against the following baselines in Sections 4.2.3 and 4.2.4: SCYTHE Wang et al. (2017a), which synthesizes SQL queries using enumerative search, and PATSQL Takenouchi et al. (2021), which uses relational algebra properties to perform a more scalable enumerative search.

We now discuss the experimental setup for each benchmark. For each benchmark, we provide each tool with the corresponding input and output tables as described in Section 4.2.1. We initialize O^+ as the set of all expected output tuples. Since both SCYTHE and PATSQL require exhaustive labeling, i.e. any tuple not labeled as O^+ is considered to be O^-, we initialize O^- to be all tuples of appropriate arity that do not occur in O^+ for all tools being compared. The benchmarks are labeled with a reference solution which identifies each column of the input tables as either `categorical`, `numerical`, or `uncomparable`.

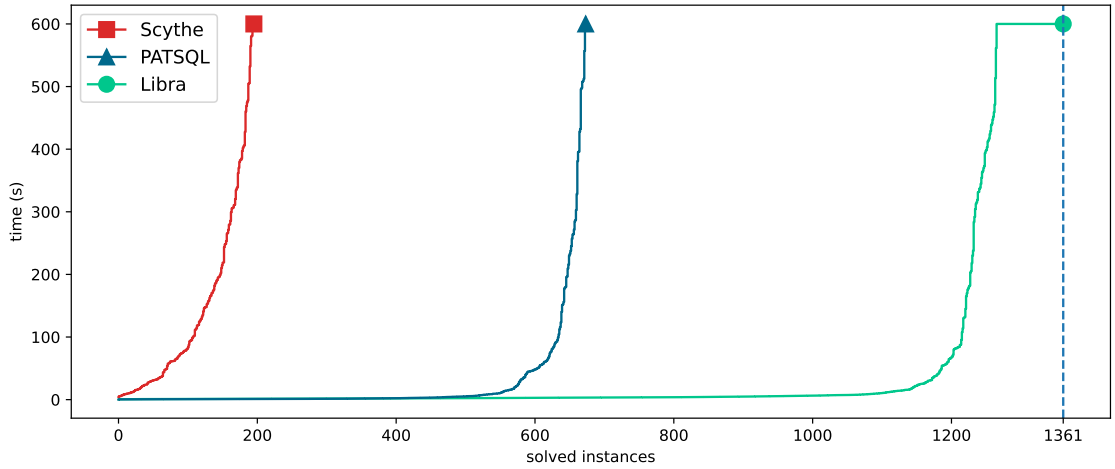


Figure 4.7: Performance of Libra against Scythe and PatSQL on the 1,475 benchmarks from the Spider and Geography datasets. Each data point $(n,\ t)$ for a tool indicates that it solved n benchmarks each within t seconds.

For the baselines Scythe and PatSQL, the user is required to specify constants that may occur in the comparison predicates. We recover the list of constants that occur in the reference solution and provide it to the two baselines as additional supervision which is not provided to Libra.

4.2.3. Performance

We run Libra, Scythe, and PatSQL on all 1,475 benchmarks with a timeout of 10 minutes and summarize the performance of each tool in a cactus plot in Figure 4.7. From this figure, we see that Libra solves the most number of benchmarks, solving 1,361 out of 1,475 in an average of 58.9 seconds, and solves 1,097 of those within 10 seconds. Of the 1,361 solved benchmarks, 1,090 are SPJ instances from the Spider dataset, while 271 are from the Geography dataset.

The plot for Libra plateaus at 600 seconds since it searches for a minimal solution to a benchmark, but returns the best solution found so far when it times out. PatSQL is outperformed by Libra, solving 673 benchmarks in an average of 23.13 seconds, and 548 in 10 seconds. Scythe solves only 195 benchmarks, in an average of 139.50 seconds, and only 15 in 10 seconds. All of the benchmarks solved by both PatSQL and Scythe are SPJ instances from the Spider dataset; neither tool solves a single instance from the Geography dataset. Also, PatSQL solves 2 benchmarks unsolved by Libra, while all benchmarks solved by Scythe are solved by Libra.

Among the benchmarks that Libra uniquely solves, a significant portion of the benchmarks have ground truths involving many joins, but with a few shared constants between tables, leading to a sparse tuple co-occurrence graph while there are syntactically many possible joins. The following generated query which Libra is the only to produce (and which happens to match the reference solution) shows how the example-guided technique allows for learning very large queries and combined with decision tree learning allows for learning complex SPJ queries:

```
SELECT employee.emp_fname, class.class_room
    FROM (((class JOIN employee ON class.prof_num = employee.emp_num)
        JOIN professor ON employee.emp_num = professor.emp_num)
        JOIN department ON department.dept_code = professor.dept_code)
            WHERE department.dept_name = "Accounting"
```

The example-guided strategy used by Libra allows it to explore solutions of a larger size more quickly than syntax-guided strategies since the smaller joins that are syntactically valid but don't explain any output tuple are skipped. This results in Libra solving benchmarks with reference solutions of a larger size where other baselines would require a longer time to search through the hypothesis space even with the additional supervision that was provided to each.

However, it is difficult to scale Libra over larger input databases. For the 114 benchmarks unsolved by Libra, over 70% have more than 5,000 tuples, and all have tables with over 20 columns, with a median of 64 columns. Libra faces two main issues when solving these benchmarks. First, it may struggle to build the tuple co-occurrence graph that it uses to enumerate contexts, and second, frequently occurring constants can result in a large number of contexts to be enumerated. The second case is true for the 2 benchmarks that PatSQL solved which were unsolved by Libra, since they contained 43 and 20 columns, with 103 and 577 rows respectively. However, the ground truth solutions for those benchmarks could be easily explored by syntax-guided processes, with one of the benchmarks consisting only of joins, and so PatSQL was able to synthesize them.

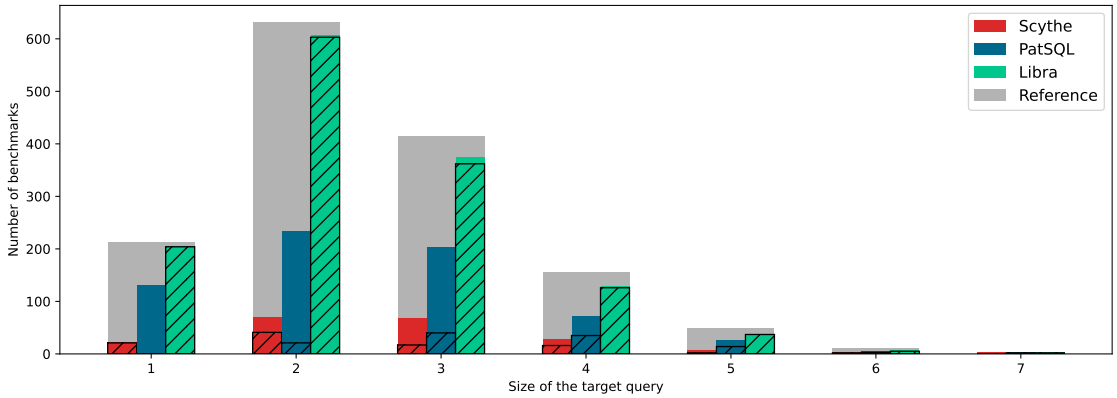


Figure 4.8: Sizes of generated programs for Libra, Scythe, and PatSQL. The bars represent the benchmarks with reference solution of a given size that are solved by each tool, and the hatched bar represents the subset of these queries that are minimal. Since 99% of the queries generated by Libra are minimal, there is very little visible unhatched bar.

4.2.4. Succinctness

We now turn to evaluating the quality of the programs in terms of succinctness. Algorithm 4 is sound by construction, that is the synthesized query is always consistent with the training data. In order to inspect for *generalizability*, we use the size of the query as a measure of its specificity with respect to the training data, where a more succinct query is assumed to be less specific to the particular data, and we rely on Occam's razor to assess over-fitting.

The size of the query is defined as the sum of the number of tables joined and the number of comparison predicates in the selection operator in the disjunctive normal form (DNF). We summarize the size of the programs synthesized by both instances of Libra and the baselines in Figure 4.8.

We observe that 1,339 of the 1,361 programs (around 99%) synthesized by Libra are minimal, that is, the size of the query is equal to or smaller than that of the reference solution. In 271 of the 1,361 programs, Libra generates a smaller query than the reference solution. This is a peculiar case common to programming-by-examples where the input-output examples under-specify the task. That is, the input-output examples do not feature all the cases that the synthesis tool should consider. Here is an example of one of the benchmarks where the input table `campuses` consists of columns for the id, campus, location, county, and year for a set of college campuses, and the input table

`csu_fees` consists of columns for campus, year, and campus fee for a set of campuses. The reference solution is:

```
SELECT campusfee FROM campuses
    JOIN csu_fees ON campuses.id = csu_fees.campus
    WHERE (campuses.campus = "San Francisco State University")
        AND (csu_fees.year = 1996)
```

Instead of this solution, LIBRA generates the query:

```
SELECT campusfee FROM csu_fees
    WHERE (csu_fees.campus = "18")
```

This is because the campus name "San Francisco State University" occurs only once in `campuses` with id "18", and the only row with campus "18" in `csu_fees` has year of 1996. Therefore, the conjunction on both the campus name and year is unnecessary and there is also no longer a need for the join of `campuses` with `csu_fees` since selecting campus "18" directly from `csu_fees` is sufficient.

There are 22 benchmarks where the size of the query generated by LIBRA is larger than the reference solution. On manual inspection of these benchmarks, we observe that the larger size is due to the sub-optimal size of the decision tree generated by DTL. As discussed before, the problem of finding a minimal decision tree is intractable and hence we adopt a greedy heuristic-based search. Therefore, any minimality guarantee will be subject to the performance of the decision tree, but we quantitatively observe that 99% of the synthesized programs are minimal.

For the baselines, we observe that the size of the synthesized programs is usually large. In contrast to LIBRA, only 115 of the 673 (17%) programs synthesized by PATSQL are minimal, and only 99 of the 195 (51%) programs synthesized by SCYTHE are minimal. Figure 4.8 shows the number of benchmarks each tool finds a solution for at each reference benchmark size shown on the x-axis, and the subset of these solutions which are minimal is shown in a bright color. We see LIBRA consistently outputs minimal solutions across program sizes.

CHAPTER 5

Synthesis of Recursive Relational Queries

In the context of programming-by-examples, recursion is crucial to synthesizing queries that generalize to arbitrary data [Cropper and Dumančić (2022)]. Recursive queries also find applications in numerous domains, including knowledge discovery Bohan et al. (2011), program reasoning Sivaraman et al. (2019); Naik et al. (2021a), and database querying Wang et al. (2022). Significant strides have been made in this area by techniques including constraint solving, enumerative search, and their combinations.

As discussed in Chapter 1, it is a greater challenge to specify the language bias mechanisms in presence of invented and recursive predicates. Determining a suitable set of templates is a delicate balancing act: overly general templates hurt scalability whereas overly constrained templates fail to synthesize the desired program.

On the other hand, there exist fully automated techniques to synthesize *non-recursive* relational queries, as discussed in the previous chapters.

In this chapter, our goal is to address the gap between template-free techniques to synthesize non-recursive queries and template-dependent techniques to synthesize recursive queries. We observe that previous synthesis techniques for non-recursive queries are successful at inferring patterns in data of finite size but are limited in generalizing those patterns to perform computation on data of arbitrary size. Additionally, the recursive techniques are successful in generalizing the patterns once templates summarize some patterns in the data and constrain the space of candidate programs. We therefore seek to leverage the strengths of the two paradigms to construct an end-to-end template-free algorithm for synthesizing recursive relational queries.

We materialize this insight as a two-phase synthesis engine called Mobius as depicted in Figure 5.1. In the first phase, we synthesize a non-recursive query Q using example-guided synthesis of conjunctive queries. In the second phase, we use this non-recursive query to constrain the hypothesis space

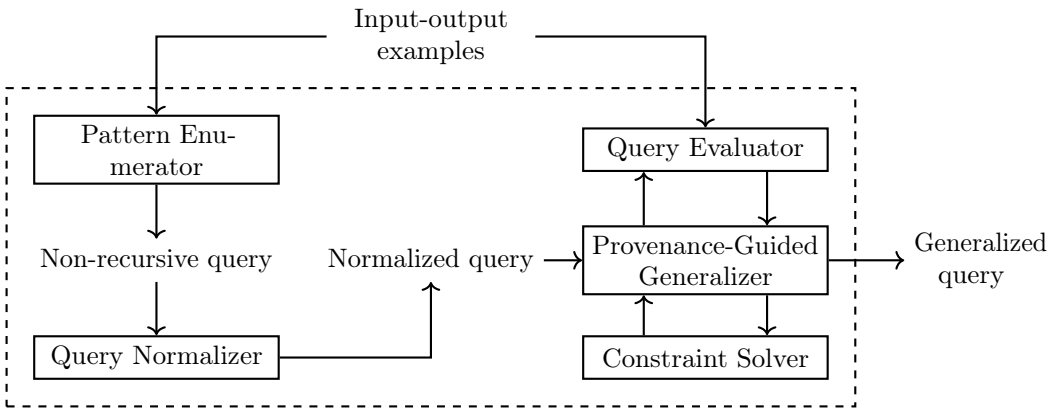
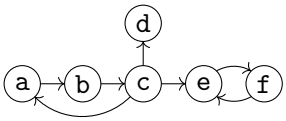


Figure 5.1: The architecture of the MOBIUS synthesis engine. We start by using a pattern enumerator (such as EGS) to generate a non-recursive query that is consistent with the input-output examples, and then generalize it into a recursive query using a provenance-guided generalization algorithm. This procedure, GENERALIZE, repeatedly uses a constraint solver to generate candidate solutions whose consistency it determines using SOUFFLE query evaluator Zhao et al. (2020). Analysis of failed candidate solutions result in additional constraints that are fed back to the constraint solver thereby pruning the search space in subsequent iterations.

to queries that *generalize* it. Our key technical contribution is a procedure GENERALIZE that realizes this generalization through provenance-guided *unification* of invented predicates. While these predicates may not feature in Q itself, we propose a normalization procedure that exposes them by rewriting Q to a semantically equivalent query $\overline{Q}$. Then, generalization proceeds in an iterative fashion that involves synergistic interaction between a constraint solver (Z3) and a query evaluator (SOUFFLE). In each iteration, the former selects a candidate unification μ, and the latter checks whether the resulting query $\mu(\overline{Q})$ is consistent with the given input-output data. If so, the process terminates; otherwise, the constraints are updated to avoid the ill-fated unification choice and the process is repeated.

A naive constraint formulation suffers from prohibitively slow convergence in practice due to an exponential number of unification choices. To accelerate the process, we develop a novel provenance-guided technique that leverages *data provenance* a derivation tree that serves as a witness of a given spurious tuple to identify a minimal incorrect core of the ill-fated unification choice (Cheney et al. (2009); Zhao et al. (2020)). We thereby eliminate from future consideration all other unification choices that are similarly destined to derive the spurious tuple.



(a) Graph G.

Input I	
edge(a,b),	edge(b,c),
edge(c,a),	edge(c,d),
edge(c,e),	edge(e,f),
edge(f,a)	

(b) Input `edge` relation.

Positive labels O^+:		
scc(a,a),	scc(a,b),	scc(a,c),
scc(c,b),	scc(e,f),	scc(f,e),
Negative labels O^-:		
scc(a,d),	scc(c,d),	scc(c,e),
scc(d,e),	scc(c,f),	scc(e,c)

(c) Positive and negative labels for `scc`.

Figure 5.2: The synthesis task is specified as a search for a relational query P that takes the graph G as an input and returns a set of pairs of vertices O such that O is a superset of O^+ and disjoint from O^-. We call such a query consistent with the input-output examples.

5.1. Demonstrative Example

We begin with a high-level overview of our end-to-end synthesis framework. As a running example, we consider the task of synthesizing a query that computes the relation induced by the strongly connected components (SCCs) in a directed graph.

5.1.1. Problem Setting

Figure 5.2a shows a directed graph and Figure 5.2b describes its adjacency relation `edge`. A user can provide this relation as an input I to a synthesis engine with the intent to synthesize a query that computes a relation `scc` representing SCCs in the graph. In order to express this intent, they label some pairs of vertices as positive tuples O^+ and some as negative tuples O^- such that the tuples in O^+ must be present in relation `scc` while those in O^- must be absent. Figure 5.2c shows an example of the positive and negative labelled output tuples. The synthesis task is to find a query P *consistent* with (I, O^+, O^-), that is, a query that takes I, the `edge` relation, as an input and generates all tuples in O^+ but none in O^-.

The following relational query $P^{\leq 3}_{\texttt{scc}}$ is consistent with (I, O^+, O^-):

$$
\begin{aligned}
r_1 &: \texttt{scc}(x, x) \text{ :- } \texttt{edge}(x, y), \texttt{edge}(y, x). \\
r_2 &: \texttt{scc}(x, y) \text{ :- } \texttt{edge}(x, y), \texttt{edge}(y, x). \\
r_3 &: \texttt{scc}(x, x) \text{ :- } \texttt{edge}(x, y), \texttt{edge}(y, z), \texttt{edge}(z, x). \\
r_4 &: \texttt{scc}(x, y) \text{ :- } \texttt{edge}(x, y), \texttt{edge}(y, z), \texttt{edge}(z, x). \\
r_5 &: \texttt{scc}(x, z) \text{ :- } \texttt{edge}(x, y), \texttt{edge}(y, z), \texttt{edge}(z, x).
\end{aligned}
\tag{5.1}
$$

$P^{\leq 3}_{\texttt{scc}}$ is a collection of rules $\{r_1, \ldots, r_5\}$. We can interpret each rule in $P^{\leq 3}_{\texttt{scc}}$ as a Horn clause. For instance, the second rule means that if both tuples (x, y) and (y, x) are in the `edge` relation, then vertices x and y are in the same SCC.

As we can observe, $P^{\leq 3}_{\texttt{scc}}$ correctly captures all SCCs in the graph of Figure 5.2a, and in general, in all directed graphs with SCCs of size 2 or 3. A program synthesis technique such as example-guided synthesis (EGS) can efficiently synthesize such queries. However, the goal of the synthesis task is to find a query that is not only consistent with the user input, but also generalizes to match the intent of the user.

5.1.2. Synthesis of Recursive Queries

We next illustrate a query for computing SCCs that matches user intent.

The following relational query $P_{\texttt{scc}}$ computes SCCs in a given graph:

$$
\begin{aligned}
r'_1 &: \texttt{scc}(x, y) \text{ :- } \texttt{path}(x, y), \texttt{path}(y, x). \\
r'_2 &: \texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{path}(y, z). \\
r'_3 &: \texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).
\end{aligned}
\tag{5.2}
$$

Observe that $P_{\texttt{scc}}$ uses a predicate `path` which is not pre-defined (that is, it does not occur as an input to the synthesis task) and also calls itself in rule r'_2. A predicate that does not appear in the synthesis task as an input or an output predicate is called an *invented* predicate. A predicate that

can *call* itself by applying a series of rules is called a *recursive* predicate. Our goal is the discovery of succinct and general queries such as $P_{\texttt{scc}}$ that potentially use invented and recursive predicates.

Further, observe that a non-recursive query synthesis engine such as EGS or Scythe cannot generate $P_{\texttt{scc}}$, nor can we modify them to directly enumerate such a query as they do not support recursive or invented predicates. Two principal challenges when synthesizing such queries: *First*, the outputs of intermediate relations are under-constrained and are not explicitly specified in the input-output examples. This significantly inhibits the ability of the synthesizer to prune candidate queries during search. *Second*, synthesis engines which attempt to enumerate candidate programs also need constraints on the number and schema of these intermediate predicates. Tools such as ProSynth and ILASP that support recursion would require additional supervision in form of the correct set of mode declarations that specify the invented and recursive predicates with their schema. Although GenSynth is able to discover invented predicates, it implicitly assumes that they must share schema with one of the input or output predicates already provided as part of the problem description. In our experiments in Section 5.5, we will present benchmarks that require both *predicate invention* as well as *schema invention*, and observe that state-of-the-art tools fail to correctly synthesize these queries.

We leverage a non-recursive query $P^{\leq 3}_{\texttt{scc}}$ that can be generated without templates (by using EGS) as a starting point for the search for $P_{\texttt{scc}}$. Observe that $P_{\texttt{scc}}$ *generalizes* $P^{\leq 3}_{\texttt{scc}}$. That is, on any graph $P_{\texttt{scc}}$ will also report all pairs of vertices generated by $P^{\leq 3}_{\texttt{scc}}$. In addition, for graphs with SCCs of size 4 or more, $P_{\texttt{scc}}$ can report pairs of vertices $\texttt{scc}(x, y)$ that $P^{\leq 3}_{\texttt{scc}}$ would miss.

In order to generalize it, we first *normalize* the given query, that is, convert it into a semantically equivalent query where a premise comprises of at most one input predicate or two invented predicates.

For ease of notation, let $Q = P^{\leq 3}_{\texttt{scc}}$. The normal form $\overline{Q}$ for the query Q would look like:

$$\begin{aligned}
\rho_1 &: \texttt{scc}(x, x) \text{ :- } R_1(x, y), R_1(y, x).\\
\rho_2 &: \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_1(y, x).\\
\rho_3 &: \texttt{scc}(x, x) \text{ :- } R_1(x, y), R_2(y, x).\\
\rho_4 &: \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_2(y, x).\\
\rho_5 &: \texttt{scc}(x, y) \text{ :- } R_2(x, y), R_1(y, x).\\
\rho_6 &: R_2(x, z) \text{ :- } R_1(x, y), R_1(y, z).\\
\rho_7 &: R_1(x, y) \text{ :- } \texttt{edge}(x, y).
\end{aligned} \tag{5.3}$$

Observe that rules $\rho_1, \ldots, \rho_5$ in $\overline{Q}$ correspond exactly to the rules $r_1, \ldots, r_5$ in Q, and uses two invented predicates R_1 and R_2. The rules for these invented predicates are ρ_6 and ρ_7.

At this point, we can highlight our key insight. There is a correspondence between the rules ρ_6 and ρ_7 in $\overline{Q}$ and the rules r_2 and r_3 in $P_{\texttt{scc}}$. These rules are identical up to renaming of the predicates. That is, if we could map $R_1(x, y)$ and $R_2(x, y)$ to $\texttt{path}(x, y)$, we would have synthesized the rules r_2 and r_3 in $P_{\texttt{scc}}$. Applying this mapping to the rest of the rules gives us a query similar to $P_{\texttt{scc}}$.

Our generalization technique builds on this insight and identifies an efficient way to search for such a map that *unifies* invented predicates. Also observe that normalization automatically discovers schema of the intermediate relations, thus eliminating the need for them to be explicitly provided as a part of the input. In this context, the normalized program $\overline{Q}$ effectively serves as a template and constraints the space of candidate programs to those that can be generated by unification.

5.1.3. Provenance-Guided Generalization

In order to search for a query P that generalizes $\overline{Q}$, we seek ways to unify the invented predicates R_1 and R_2. Section 5.4 details out a way to encode this as a constraint satisfaction problem. We start with a bound on the number of invented predicates. For the sake of example, let it be 1. That is, we wish to map R_1 and R_2 to the same predicate, say S_1. Clearly, there are four ways to permute the

variables for R_1 and R_2, and each of them gives us a map:

$$\mu_1 : R_1(x, y) \mapsto S_1(x, y), \quad R_2(x, y) \mapsto S_1(x, y)$$
$$\mu_2 : R_1(x, y) \mapsto S_1(x, y), \quad R_2(x, y) \mapsto S_1(y, x)$$
$$\mu_3 : R_1(x, y) \mapsto S_1(y, x), \quad R_2(x, y) \mapsto S_1(x, y)$$
$$\mu_4 : R_1(x, y) \mapsto S_1(y, x), \quad R_2(x, y) \mapsto S_1(y, x)$$

In order to apply a unification μ to $\overline{Q}$, we replace each occurrence of $R_1(x, y)$ and $R_2(x, y)$ with $\mu(R_1(x, y))$ and $\mu(R_2(x, y))$ respectively in each rule. For example, on applying μ_2 to $\overline{Q}$ we get the query $T(\overline{Q}, \mu_2)$:

$$\mu_2(\rho_1) : \texttt{scc}(x, x) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_2(\rho_2) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_2(\rho_3) : \texttt{scc}(x, x) \text{ :- } S_1(x, y), S_1(x, y).$$
$$\mu_2(\rho_4) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(x, y).$$
$$\mu_2(\rho_5) : \texttt{scc}(x, y) \text{ :- } S_1(y, x), S_1(y, x).$$
$$\mu_2(\rho_6) : S_1(x, z) \text{ :- } S_1(y, x), S_1(z, y).$$
$$\mu_2(\rho_7) : S_1(y, x) \text{ :- } \texttt{edge}(x, y).$$

Observe that if a tuple is produced by $\overline{Q}$, then for any unification map μ, the same tuple can be generated by $T(\overline{Q}, \mu)$ by applying the corresponding set of rules. We formally prove this in Theorem 5.4.3. However, it is possible that $T(\overline{Q}, \mu)$ has an output larger than $\overline{Q}$. In this sense, unification leads to generalization. We call $T(\overline{Q}, \mu)$ a candidate query.

We then check if the candidate query $T(\overline{Q}, \mu)$ is consistent with the input-output example (I, O^+, O^-). If it is, then we can return it as a synthesized result. On the other hand, it is also possible that such a generalization is *too general*, that is, it also generates some of the tuples in O^-.

In the example above, the tuple $\texttt{scc}(\texttt{c}, \texttt{d})$ can be generated by $T(\overline{Q}, \mu_2)$ while `c` and `d` are not in the same SCC. We can analyze the derivation tree for a tuple like $\texttt{scc}(\texttt{c}, \texttt{d})$ to assign *blame* to a part of the unification map. This blame can be converted into a constraint to rule out future unification maps where similar patterns may occur, and thereby guide the search towards correct generalizations.

In order to implement this, we set up an interactive process involving a constraint solver that proposes candidate queries and a query evaluator that verifies whether the candidate is consistent with the input-output example. In the case where the candidate is not consistent, the query evaluator provides a derivation tree of every tuple in $t \in O^-$ that can be generated by the candidate. We use these derivation trees to craft the constraints.

The key insight of the provenance-guided technique is to leverage the derivation tree of an unexpected tuple. The derivation tree allows us to identify a number of unifications that lead to such a tuple and they can be avoided in future iterations.

Eventually, the constraint solver proposes the unification map μ_1. Not only is the query $T(\overline{Q}, \mu_1)$ consistent with (I, O^+, O^-), it is also *similar* to the intended query $P_{\texttt{scc}}$. It has the rules:

$$\mu_1(\rho_1) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_1(\rho_2) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_1(\rho_3) : \texttt{scc}(x, x) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_1(\rho_4) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_1(\rho_5) : \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(y, x).$$
$$\mu_1(\rho_6) : S_1(x, z) \text{ :- } S_1(x, y), S_1(y, z).$$
$$\mu_1(\rho_7) : S_1(x, y) \text{ :- } \texttt{edge}(x, y).$$

The rules $\mu_1(\rho_2)$, $\mu_1(\rho_5)$, and $\mu_1(\rho_6)$ correspond exactly to the rules r'_1, r'_2, and r'_3 in $P_{\texttt{scc}}$ (Equation 5.2). The rules $\mu_1(\rho_4)$, and $\mu_1(\rho_5)$ are identical to $\mu_1(\rho_1)$ or can be derived using it. Using the

rules $\mu_1(\rho_1)$ and $\mu_1(\rho_5)$, one can *derive* the rule $\mu_1(\rho_1)$ and $\mu_1(\rho_3)$. Once simplified, this gives a correct and interpretable solution to the problem originally posed in Figure 5.2.

5.2. Minimal Generalization Problem

Our ultimate goal is to synthesize a recursive relational query which is consistent with given input-output examples. Given a set of input tuples, I, and a set of output tuples partitioned as O^+ and O^-, tools such as EGS and Scythe can effectively synthesize queries P such that P generates tuples in O^+ and does not generate any tuple in O^-. However, these are non-recursive query. As discussed in the overview, we are interested in the generalization problem where given a query Q that is consistent with the input-output examples, we wish to find a query P that generalizes it. For this purpose, we first define subsumption:

Definition 5.2.1 (Subsumption)**.** A relational query P subsumes a relational query Q if for any set of input tuples I, $[\![Q]\!](I) \subseteq [\![P]\!](I)$.

That is, for any input I, if Q generates a tuple t, then P also generates t. For example, the query P_{scc} subsumes $P^{\leq 3}_{\mathsf{scc}}$ We can now define the size of a relational query. The size of a query is the sum of size of the rules in the query. For example, the size of the query P_{scc} is 5 and of $P^{\leq 3}_{\mathsf{scc}}$ is 11. We can now define the minimal generalization problem:

Problem 5.2.2 (Minimal Generalization)**.** *Given an input-output example $E = (I, O^+, O^-)$, and a relational query Q consistent with E, find a relational query P that subsumes Q, is consistent with E, and is of minimal size among such queries.*

For example, the user may specify the input and output tuples and seek a query that *explains* the relation between input and output tuples. They may use the query $P^{\leq 3}_{\mathsf{scc}}$ generated by EGS as a seed in order to search for the query P_{scc} that uses invented and recursive predicates so it can match user intention.

5.3. The Synthesis Algorithm

In this section we describe the end-to-end MOBIUS algorithm, which takes an input-output example $E = (I, O^+, O^-)$ as input and returns a relational query P (which potentially has invented and recursive predicates). Algorithm 6 summarises the procedure.

Algorithm 6 MOBIUS(I, O^+, O^-), where (I, O^+, O^-) is an instance of the synthesis task.

1. Let $Q_0 = \text{EGS}(I, O^+, O^-)$. If EGS fails to return a relational query, end the procedure and return `unsat`.
2. Initialize $Q \coloneqq \emptyset$.
3. While there is a tuple $t \in O^+ \setminus [\![Q]\!](I)$:
   (a) Let $r \in Q_0$ derive t. Update $Q \coloneqq Q \cup \{r\}$.
   (b) Let $\overline{Q} = \text{NORMALIZE}(Q)$.
   (c) Compute $P = \text{GENERALIZE}(\overline{Q}, I, O^-)$.
   (d) If $O^+ \subseteq [\![P]\!](I)$, end the procedure and output t.

We start with using a non-recursive query synthesizer EGS. The output of EGS, Q_0, is a non-recursive query. We construct a query $Q \subseteq Q_0$ on demand, initialized to the empty set, that grows till the synthesized query is not consistent with (I, O^+, O^-).

In order to generalize the query Q, we first normalize it to $\overline{Q}$. We discuss the normalization procedure in Section 5.3.2. The normalized query is then provided as an input to the provenance-guided generalization procedure which we discuss in Section 5.4.

5.3.1. Example-guided Synthesis

Example-guided Synthesis (EGS) is a template-free algorithm to synthesize non-recursive queries from input-output examples. While EGS supports features such as multi-way joins and unions, it does not allow for invented or recursive predicates. Therefore, on inherently recursive tasks, EGS cannot synthesize the *intended* query. $P^{\leq 3}_{\texttt{scc}}$ is an example of a query that EGS may generate. Additionally, EGS cannot be modified to generate recursive programs as it not a syntax-guided tool.

However, EGS does provide a completeness guarantee that if there exists a non-recursive query consistent with the input-output example (I, O^+, O^-), then EGS will find a consistent query Q_0. We can prove that there exists a recursive relational query consistent with a given input-output example $E = (I, O^+, O^-)$ only if there is a non-recursive query that is consistent with E. Therefore, when

EGS returns `unsat`, we can conclude that there does not exist a query (recursive or non-recursive) that is consistent with the input-output example. Using this, we can prove:

Theorem 5.3.1 (Completeness)**.** *If there exists a query consistent with the input-output example* $E = (I, O^+, O^-)$*, then* Mobius *produces a relational query* P *consistent with* E*.*

This is because we use EGS as a first step in the process, which allows Mobius to ensure that if there is no consistent query, then we do not proceed with a futile search. On the other hand, if EGS produces a query, Mobius only further generalizes it and in the worst case, it may output the same query (after normalization). This allows us to conclude a completeness guarantee for the end-to-end synthesis procedure:

5.3.2. Normalization

Once we have the query Q_0, we construct Q on demand. Then, in Step 3b, we normalize Q to $\overline{Q}$. Normalization introduces invented predicates in the query which we further use for generalization through unification. The following definition of a normal query is motivated by the Chomsky Normal Form for context-free languages Sipser (2012).

Definition 5.3.2 (Normal Form)**.** A relational query is said to be in the normal form if every rule is of one of the two forms:

$$R(\vec{x}) \quad \text{:-}\ R_1(\vec{x}_1), R_2(\vec{x}_2)$$
$$R(\vec{x}) \quad \text{:-}\ R_{in}(\vec{x}_{in})$$

where R, R_1, and R_2 are invented predicates and R_{in} is an input predicate. That is, the body of a rule either has two invented predicates or one input predicate.

For example, $P^{\leq 3}_{\texttt{scc}}$ in Equation 5.1 is not in normal form while $P_{\texttt{scc}}$ in Equation 5.2 is. Analogous to context-free languages, the normalization of relational queries can be carried out by rewriting the rules into semantically equivalent rules and introducing invented predicates, and we can show that every query can be *normalized*. We employ a greedy heuristic to normalize queries that allows us to

minimize the size of the number of invented predicates as well as their arity.

Let a given rule be of the form:

$$R(\vec{x}) \text{ :- } R_1(\vec{x}_1), R_2(\vec{x}_2), \ldots, R_n(\vec{x}_n).$$

We partition the literals in the body into two disjoint sets S_l and S_r such that the number of variables shared by literals in S_l and literals in S_r are minimal. Let $\vec{x}_l$ be a vector of variables that occur in the literals in S_l and either in $\vec{x}$ or a literal in S_r. Similarly, let $\vec{x}_r$ be a vector of variables that occur in the literals in S_r and either in $\vec{x}$ or a literal in S_l. Then, we can rewrite r as:

$$\begin{aligned} R(\vec{x}) &\text{ :- } R_l(\vec{x}_l), R_r(\vec{x}_r) \\ R_l(\vec{x}_l) &\text{ :- } R_{i_1}(\vec{x}_{i_1}), \ldots, R_{i_n}(\vec{x}_{i_n}) \\ R_r(\vec{x}_j) &\text{ :- } R_{j_1}(\vec{x}_{j_1}), \ldots, R_{j_m}(\vec{x}_{j_m}), \end{aligned}$$

where we have $S_l = \{R_{i_1}(\vec{x}_{i_1}), \ldots, R_{i_n}(\vec{x}_{i_n})\}$ and $S_r = \{R_{i_1}(\vec{x}_{i_1}), \ldots, R_{i_n}(\vec{x}_{i_n})\}$. We can iteratively apply this rewriting rule to normalize the query. Observe that this is a greedy process, and hence minimality of the normal form is not guaranteed.

Secondly, instead of recreating R_l and R_r at every step of the normalization procedure, we *reuse* the invented predicates. That is, if there are two predicates which are described by syntactically identical rule bodies (up to permutation of variables) that exists in the query, we require only one of them. This allows us to reuse predicates and shrink the search space. If one chooses not to *reuse* the predicates, they will be eventually unified during the generalization step. However, this heuristic allows us to reduce the size of the search space and hence accelerate the synthesis process. Note that this optimization does not compromise the end-to-end guarantee of Theorem 5.3.1. For the running example, the normalization of $P^{\leq 3}_{\mathsf{scc}}$ generates the query $\overline{Q}$ in Equation 5.3.

5.4. Provenance-guided Generalization

We can use the normalized relational query as a template to constrain the space of candidate queries to only those which can be constructed by unification of predicates. As discussed in the overview, the user may provide a query like $P_{\mathtt{scc}}^{\leq 3}$ (Equation 5.1) and intend to generalize it to $P_{\mathtt{scc}}$ (Equation 5.2). In the rest of this section, we develop a provenance-guided technique to solve the minimal generalization problem (Problem 5.2.2) for queries in the normal form. We have named this provenance-guided generalization procedure GENERALIZE and outline it in Algorithm 7.

Algorithm 7 GENERALIZE$(\overline{Q}, I, O^-)$, where $\overline{Q}$ is a normalized query, I is the set of input tuples and O^- is the set of negatively labeled output tuples.

---

1. Initialize $\phi \coloneqq \phi_0(\overline{Q})$.
2. Let $\overline{Q}$ have K invented predicates. Then, for $k = 1$ to $k = K$:
   (a) Let $\mu =$ GENERALIZE$(\overline{Q}, k, \phi)$.
   (b) If the GENERALIZE procedure fails to find a unification map μ, then break the loop.
   (c) Otherwise, let $P = T(\overline{Q}, \mu)$.
      i. If $[\![P]\!](I) \cap O^- = \emptyset$, end the procedure and return P.
      ii. Otherwise, for each t in $[\![P]\!](I) \cap O^-$, update:

$$\phi \coloneqq \phi \wedge \textsc{Constraint}(\overline{Q}, \mu, t).$$

We can show that GENERALIZE solves the generalization problem (Problem 5.2.2) for normal queries with the guarantee that the output of Algorithm 7 will have the least number of invented predicates.

Theorem 5.4.1. *Given a normalized query $\overline{Q}$, input tuples I and negatively labeled output tuples O^-, the query P generated by* GENERALIZE$(\overline{Q}, I, O^-)$ *is a normalized query that subsumes $\overline{Q}$, does not generate tuples in O^-, and has the fewest invented predicates among all such queries.*

The proof of this theorem relies on Theorem 5.4.3 which ensures that P subsumes $\overline{Q}$, the soundness check in step 2(c)i of Algorithm 7 that ensures no tuple in O^- are generated, and the fact that step 2 of Algorithm 7 searches for the least number of invented predicates incrementally. Therefore, Algorithm 6 solves Problem 5.2.2 when the input is a normalized query $\overline{Q}$. In cases where the input query is not normalized, we can guarantee subsumption but not minimality. The rest of this section discusses the details of the algorithm.

5.4.1. Generalization Algorithm

This algorithm approaches generalization as a unification procedure. That is, as explained in the overview, we rewrite the literals in the query. In order to carry out this process, we seek a map μ from the literals using the invented predicates in $\overline{Q}$ to literals using *fresh* invented predicates. For this section, we consider the following query Q_0 that uses three invented predicates R_1, R_2, and R_3:

$$\begin{aligned}
\rho_1 &: \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_2(x, y). \\
\rho_2 &: R_1(x, y) \text{ :- } \texttt{edge}(x, y). \\
\rho_3 &: R_2(x, z) \text{ :- } R_1(z, y), R_1(y, x). \\
\rho_4 &: R_3(x, z) \text{ :- } R_1(x, y), R_2(z, y).
\end{aligned} \tag{5.4}$$

Here, the predicate $R_1(x, y)$ represents that there is an edge between x and y, $R_2(x, z)$ represents there is a path of length two from z to x and $R_3(x, z)$ represents there is a path of length three from x to z. Consider a unification map μ that maps all of $R_1(x, y)$, $R_2(x, y)$, and $R_3(x, y)$ to an invented predicate $S_1(x, y)$. Formally, we define:

Definition 5.4.2 (Unification Map)**.** Given query Q with invented predicates in $\mathcal{R}$ and variables in X, and a set of invented predicates $\mathcal{S}$ that do not occur in Q, a unification map $\mu : \mathcal{R} \cdot X^* \to \mathcal{S} \cdot X^*$ is a function from literal $R(\vec{x})$ in Q that use predicate $R \in \mathcal{R}$ to a literal $S(\vec{x}')$ that uses predicate $S \in \mathcal{S}$ and where $\vec{x}'$ is a permutation of variables $\vec{x}$.

In order to apply a unification map μ to a query $\overline{Q}$, we replace each occurrence of the literal $R(\vec{x})$ in $\overline{Q}$ with $S(\vec{x}')$. We denote such a query with $T(\overline{Q}, \mu)$ where T is a transformation that applies μ to $\overline{Q}$. In the running example, we have $T(Q_0, \mu)$:

$$\begin{aligned}
\mu(\rho_1) &: \texttt{scc}(x, y) \text{ :- } S_1(x, y), S_1(x, y). \\
\mu(\rho_2) &: S_1(x, y) \text{ :- } \texttt{edge}(x, y). \\
\mu(\rho_3) &: S_1(x, z) \text{ :- } S_1(z, y), S_1(y, x). \\
\mu(\rho_4) &: S_1(x, z) \text{ :- } S_1(x, y), S_1(z, y).
\end{aligned} \tag{5.5}$$

This method of unification provides a subsumption guarantee that the query $T(Q_0, \mu)$ generates all the tuples generated by Q_0:

Theorem 5.4.3 (Subsumption)**.** *For every relational query Q and a unification map μ, the query $T(Q, \mu)$ subsumes Q, that is, on every input I, $[\![Q]\!](I) \subseteq [\![T(Q, \mu)]\!](I)$.*

Proof. Consider a tuple t that can be derived by Q and has a derivation tree τ. Then, to prove that t can be derived by $T(Q, \mu)$, we construct a derivation tree for t in $T(Q, \mu)$ by replacing each rule ρ in t by $\mu(\rho)$. It is immediate that the constructed tree uses rules in $T(Q, \mu)$ to derive t. □

However, $T(Q, \mu)$ may derive undesirable tuples, for instance, consider the tuple $\texttt{scc}(\texttt{c}, \texttt{d})$ derived by $T(Q_0, \mu)$ (as shown in Figure 5.3a). Hence, $T(Q_0, \mu)$ is an incorrect generalization and we would like to prune it out in the next iteration of the generalization procedure. Observe that for a given query $\overline{Q}$, the space of unification maps is finite. One can enumerate all unification maps, construct the corresponding queries and check if they are consistent with the input-output example. If there are K predicates in $\overline{Q}$ and k predicates after unification, then the number of possible maps are given by:

$$\sum_{k=1}^{K} \left\{ {K \atop k} \right\} (K-k)n! \geq n! \sum_{k=1}^{N} \binom{K}{k-1} (K-k) \backsim n! 2^K \geq 2^{n+K},$$

where $\left\{ {K \atop k} \right\}$ is the Stirling number of the second kind and we assume that each predicate has arity n and the signatures are untyped. This implies that number of candidate queries that can be generated by unification grow exponentially in both the number of invented predicates in the normalized query as well as the arity of the predicates, making an exhaustive search infeasible. Therefore we reduce this search problem to a constraint satisfaction problem by encoding the possible unification maps as variables, and prune it using provenance.

5.4.2. Encoding Generalization as Constraint Satisfaction

Let the bound on the number of invented predicates in the candidate query be k. Then, for every invented predicate R of arity n in $\overline{Q}$, we introduce:

1. an integer variable $c(R)$ such that $1 \leq c(R) \leq k$, and

2. for each integer i such that $1 \leq i \leq n$, an integer variable $p(R, i)$ such that $1 \leq p(R, i) \leq n$.

Additionally, we have the constraint that each $p(R, i)$ should be unique for i. That is, for all relations R and indices i and j, $p(R, i) = p(R, j) \implies i = j$.

The conjunction of the above constraints form the initial constraint $\phi_0(\overline{Q})$. In order to interpret an assignment to this encoding as a unification, we say that the literal $R(x_1, \ldots, x_n)$ is mapped to $S_{c(R)}(x_{p(R,1)}, \ldots, x_{p(R,n)})$.

5.4.3. Provenance-Guided Constraint Generation

Observe that μ is an incorrect generalization and we would like to eliminate the assignment that leads to it. For this purpose, we carefully analyze the program $T(Q_0, \mu)$. Intuitively, it is clear that unifying R_1 and R_2 can lead to an incorrect program as the former represents paths and the latter represents reverse paths. Therefore, one can assign the *blame* of incorrect generalization to the unification of $R_1(x, y)$ with $R_2(x, y)$, and this is independent of how $R_3(x, y)$ is unified with either of the two.

In general, the goal is to identify a minimal set of predicates whose unification leads to incorrect generalization, and use this to prune out all unification maps that contain them. For this purpose, we construct a program that is equivalent to $T(Q, \mu)$ by introducing *tunneling clauses* to Q. In the unification map if some predicate $R(\vec{x})$ is unified with $R'(\vec{x'})$, then we add the rules $R(\vec{x})$:- $R'(\vec{x'})$ and $R'(\vec{x'})$:- $R(\vec{x})$. For a query Q and unification map μ, the program constructed using the tunneling clauses is represented as $T'(Q, \mu)$. We wish to show that $T(Q, \mu)$, the program generated by unifying predicates in Q is semantically equivalent to $T'(Q, \mu)$, the program generated by adding the tunneling clauses.

Theorem 5.4.4 (Tunneling)**.** *The programs $T(Q, \mu)$ and $T'(Q, \mu)$ are semantically equivalent.*

Proof. Consider an input I. We will show that $T(Q, \mu)$ can derive a tuple t using input I, if and only if, $T'(Q, \mu)$ can derive t. The proof in both directions proceed by structural induction on the derivation tree of t.

In the forward direction, consider a derivation of t in $T(Q, \mu)$. If $R(\vec{x}')$ is unified to $S(\vec{x})$, for every rule of the form $S(\vec{x})$:- $R_{in}(\vec{x}_{in})$ we introduce the rule $R(\vec{x}')$:- $R_{in}(\vec{x}_{in})$. For every rule $\mu(\rho)$ of the form $S(\vec{x})$:- $S_1(\vec{x}_1), S_2(\vec{x}_2)$, consider the rules $\mu(\rho_1)$ and $\mu(\rho_2)$ whose heads derive the predicates $S_1(\vec{x}_1)$ and $S_2(\vec{x}_2)$. Let ρ be $R(\vec{x})$:- $R_1(\vec{x}_1), R_2(\vec{x}_2)$ and the heads of ρ_1 and ρ_2 be $R_1'(\vec{x}_1')$ and $R_2'(\vec{x}_2)'$ respectively. As we have $\mu(\rho)$ using the heads of $\mu(\rho_1)$ and $\mu(\rho_2)$, we have that $R_1(\vec{x}_1)$ is unified with $R_1'(\vec{x}_1')$ to form $S_1(\vec{x}_1)$ (and similarly for $S_1(\vec{x}_1)$). Hence, we can introduce the rules ρ and the tunneling clauses $R_1(\vec{x}_1)$:- $R_1'(\vec{x}_1')$ and $R_1'(\vec{x}_1')$:- $R_1(\vec{x}_1)$. The corresponding derivation tree in $T'(Q, \mu)$ can derive t.

Now consider a derivation tree in $T'(Q, \mu)$. If it uses a rule $\rho \in Q$, we introduce the rule $\mu(Q)$. If it uses a tunneling clause $R_1(\vec{x}_1)$:- $R_2(\vec{x}_2)$, then it must be the case that μ unifies $R_1(\vec{x}_1)$ and $R_2(\vec{x}_2)$ to some $S(\vec{x})$. Then, we introduce the rule $S(\vec{x})$:- $S(\vec{x})$. The corresponding tree derives t using rules in $\mu(Q)$ along with tautological rules of the form $S(\vec{x})$:- $S(\vec{x})$. Observe that tautological rules can be eliminated in the derivation tree as their head is the same as the predicate in the premise. This gives us a derivation tree for t using rules $\mu(Q)$.

Therefore, any tuple that can be derived in $\mu(Q)$ can be derived in $T'(Q, \mu)$, and they are semantically equivalent queries. □

Below is the query with tunneling clauses for the running example Q_0 with unification μ (that is, $T(Q_0, \mu)$ as in Equation 5.5) .

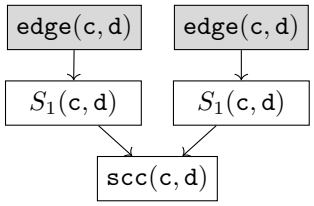
$$\rho_1 : \texttt{scc}(x, y) \text{ :- } R_1(x, y), R_2(x, y). \qquad \rho_6 : R_1(x, y) \text{ :- } R_3(x, y).$$

$$\rho_2 : R_1(x, y) \text{ :- } \texttt{edge}(x, y). \qquad \rho_7 : R_2(x, y) \text{ :- } R_1(x, y).$$

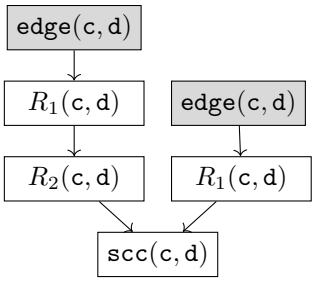
$$\rho_3 : R_2(x, z) \text{ :- } R_1(z, y), R_1(y, x). \qquad \rho_8 : R_2(x, y) \text{ :- } R_3(x, y).$$

$$\rho_4 : R_3(x, z) \text{ :- } R_1(x, y), R_2(z, y). \qquad \rho_9 : R_3(x, y) \text{ :- } R_1(x, y).$$

$$\rho_5 : R_1(x, y) \text{ :- } R_2(x, y). \qquad \rho_{10} : R_3(x, y) \text{ :- } R_2(x, y).$$



(a) Example derivation tree of the output tuple scc(c, d) for the query $T(Q_0, \mu)$.



(b) Example derivation tree of the output tuple scc(c, d) for the query $T'(Q_0, \mu)$.

Figure 5.3: The derivation tree of the tuple scc(c, d) for the queries $T(Q_0, \mu)$ and $T'(Q_0, \mu)$. The input to the query is the graph of Figure 2.3a.

On evaluating the program $T(Q_0, \mu)$ on the input graph of Figure 5.2b, we observe that it derives the tuple scc(c, d). By Lemma 5.4.4, $T'(Q_0, \mu)$ also derives scc(c, d). Figure 5.3 shows the derivation tree for the two programs.

We will use the derivation tree of scc(c, d) in $T'(Q, \mu)$ to assign the blame of generating a tuple in O^-. That is, in the running example, we analyze the derivation tree in Figure 5.3b and seek all tunneling rules used in the derivation. Observe that the tree uses only the rule ρ_7 which corresponds to unifying $R_1(x, y)$ and $R_2(x, y)$. Any unification map that unifies these two predicates (with the same rearrangement of variables) will generate scc(c, d) and we can eliminate them in the future iterations. However, the derivation tree does not use a rule with R_3, and hence we can conclude that a unification of R_3 is irrelevant to the derivation of the undesirable tuple. In this sense, the analysis of the derivation tree gives us a part of the unification to assign blame for generating a tuple in O^-.

In general, if the derivation tree includes a tunneling clause of the form $R(\vec{x}_1)$:- $R'(\pi(\vec{x}))$ for some permutation of variables π, we add the constraint:

$$\neg \left(c(R) = c(R') \wedge \bigwedge_{i=1}^{k} p(R, i) = p(R', \pi(i)) \right),$$

where k is the arity of R. This constraint prunes out all unifications where $R(\vec{x}_1)$ is unified with $R'(\pi(\vec{x}))$. If the derivation tree uses more than one tunneling clause, we take the conjunction of all

of them.

The process of constructing the derivation tree of a tuple t in $T'(Q, \mu)$ is implemented as a subroutine CONSTRAINT(Q, μ, t), and is used in Algorithm 7.

5.5. Experimental Evaluation

Our implementation of MOBIUS consists of approximately 1,300 lines of Python code. We use SOUFFLE Zhao et al. (2020) to evaluate candidate queries and compute data provenance, and we use Z3 to solve the constraints generated by the GENERALIZE procedure. Our evaluation in this section attempts to answer the following questions:

Q1. *Effectiveness:* How effective is MOBIUS in synthesizing queries with a variety of recursion schemes compared to state-of-the-art tools?

Q2. *Generalizability:* Does predicate unification improve accuracy when the learned query is tested on unseen data?

Q3. *Expressibility:* How does the expressive power of MOBIUS compare against the baselines?

Q4. *Convergence:* Does accounting for data provenance improve convergence time?

We describe our benchmark suite in Section 5.5.1 and the three baselines against which we compare MOBIUS in Section 5.5.2. We present our findings for **Q1**, **Q2**, **Q3**, and **Q4** in Sections 5.5.3–5.5.6 respectively.

5.5.1. Benchmarks

We evaluate MOBIUS on a suite of 21 synthesis tasks obtained from the domains of knowledge discovery and program analysis. The intended solutions for all of these tasks involve the use of recursion. We present a summary of these benchmarks in Table 5.1. The benchmarks are divided into seven categories:

1. *Transitive Closure*: This is the simplest example of a recursive query that constructs the transitive closure of the input predicate. We use the example of reachability in directed graphs

for this category.

2. *Boolean Transitive Closure*: This category comprises of queries that involve transitive closure and some Boolean operation such as conjunction or disjunction. It includes five benchmarks that draw from the domains of knowledge discovery and program reasoning.

3. *Linear Queries*: A linear query is one where the invented (or output) predicate occurs at most once in each rule Abiteboul et al. (1994). While the previous two benchmark categories also include only linear queries, this category includes three benchmarks from knowledge discovery that are not covered by Boolean transitive closure.

4. *Intersection*: These queries correspond to intersection of linear queries (such as `scc` is an intersection of `path` and its reverse). This category consists of two benchmarks.

5. *Schema Invention*: The `monochromatic` query corresponds finding monochromatic paths in a vertex colored graph. We discuss it in detail in Section 5.5.4.

6. *Non-linear Queries*: These are three other queries from knowledge discovery and program analysis that cannot be expressed as a linear query.

7. *Mutual Recursion*: This category consists of six linear and non-linear queries involving mutual recursion, that is, they have two or more recursive predicates that call each other.

These benchmarks are collected from previous literature on relational query synthesis and express a diverse range of challenges from across different application domains.

5.5.2. Baselines

We compare Mobius with three state-of-the-art synthesizers that use different synthesis techniques: GenSynth Mendelson et al. (2021), which uses an evolutionary search algorithm, and ILASP Law et al. (2020a) and Popper Cropper and Morel (2021), which are based on constraint solving techniques.

ILASP and Popper model the synthesis problem as a search through a finite space of candidate

Table 5.1: Table summarizing benchmark characteristics. We evaluate MOBIUS on a suite of 21 benchmarks featuring diverse recursion schemes. For each benchmark, we summarize the number of input-output relations and the number of input-output tuples. Ten of these benchmarks use invented predicates.

Name	Brief description	Input		Output	
		Preds	Tuples	Preds	Tuples
transitive closure					
path	graph reachability Raghothaman et al. (2020a)	1	7	1	31
boolean transitive closure					
ancestor	find ancestor in a family tree Muggleton et al. (2015)	2	8	1	19
connected	unidirectional graph reachability Mendelson et al. (2021)	1	20	1	104
escape	escape analysis for Java Si et al. (2018)	4	13	1	6
union-find	equivalence of elements in same set Si et al. (2018)	3	21	1	36
wikiedits	extract edit history in Wikipedia	4	16	1	7
linear queries					
rsg	reverse-same-generation in family tree Abiteboul et al. (1994)	3	17	1	11
sgen	same generation in family tree Abiteboul et al. (1994)	1	7	1	21
zero	checking equality of numbers	2	12	1	38
intersection					
blue-and-green	graph reachability with two colored paths	2	9	1	5
scc	compute SCCs in graph Raghothaman et al. (2020a)	1	10	1	25
schema invention					
monochromatic	monochromatic paths in a vertex colored graph	2	134	1	56
non-linear queries					
andersen	inclusion-based pointer analysis for C Andersen (1994)	4	7	1	7
dyck	well balanced parentheses	2	10	1	8
modref	mod-ref analysis for Java Si et al. (2018)	7	18	5	34
mutual recursion					
1-call-site	1-call-site pointer analysis for Java Whaley and Lam (2004)	7	28	1	4
1-object	1-object-sensitive pointer analysis Milanova et al. (2002)	9	40	1	4
1-object-1-type	1-type-1-object sensitive analysis Smaragdakis et al. (2011)	10	48	1	6
1-type	1-type-sensitive pointer analysis Smaragdakis et al. (2011)	10	42	1	5
2-call-site	2-call-site pointer analysis for Java Whaley and Lam (2004)	7	30	1	4
buildwall	learn a stable wall strategy Muggleton et al. (2015)	4	30	1	4

queries. In order to evaluate them in our setting, we generated candidate rules for each of the 20 benchmarks using instance-specific *mode declarations*. A mode declaration is a syntactic constraint on the candidate queries such as the length of the rule or the number of times a particular relation can occur in its body. In particular, we provide ILASP with the names and signatures of all predicates, including invented predicates, whether they can appear as the head of a clause, and the maximum number of times each predicate can appear in a clause body. In addition, we also provide the maximum number of variables in each rule. Similarly, we provide POPPER with bounds on the number of learned rules, their lengths, and the number of variables which can occur in each rule. We ensure uniformity by running all baselines in single-threaded mode.

For each benchmark query, we recovered the minimum mode declarations required from its reference solution. For example, for the query:

$$\texttt{path}(x, z) \text{ :- } \texttt{path}(x, y), \texttt{path}(y, z).$$

$$\texttt{path}(x, y) \text{ :- } \texttt{edge}(x, y).$$

We have the mode declarations:

```
#modeb(1, edge(var(V), var(V)), (positive)).
#modeb(2, path(var(V), var(V)), (positive)).
#modeh(path(var(V), var(V))).
#maxv(3).
```

These mode declarations specify that `edge` and `path` predicates may appear in rule bodies, and also specify the maximum number of times they may be used. Additionally, the head of a rule can only have `path` or `scc` predicate, and no rule should use more than 3 variables. That is, the mode declaration implicitly specify that there is only one recursive predicate `path`. In case of invented predicates, the user must explicitly provide the invented predicate along with its schema.

Lemma 4.2 of 3.2.2 alternatively provides an instance-agnostic technique to derive mode declarations. However, as we will see in our evaluation in Section 3.4.3, the baseline tools often run out of time with even the more constrained instance-specific settings, thereby rendering this instance-agnostic approach infeasible.

In summary, we make the most favorable case for these baselines by choosing the tightest set of mode declarations that contains the reference solution for the corresponding synthesis task.

5.5.3. Effectiveness

We compared the performance of Mobius against the baselines by running each of them on the benchmarks. We set a uniform timeout of 15 minutes for all tools, and ran the experiments on a desktop workstation with a Ryzen 9 5950X CPU and 128 GB of memory running Linux. We measured the running time of each of these tools on each benchmark. We present the running times in Table 5.2 and present an alternative visualization in Figure 5.4a.

Overall, Mobius consistently produces solutions in the least time, despite requiring lesser guidance

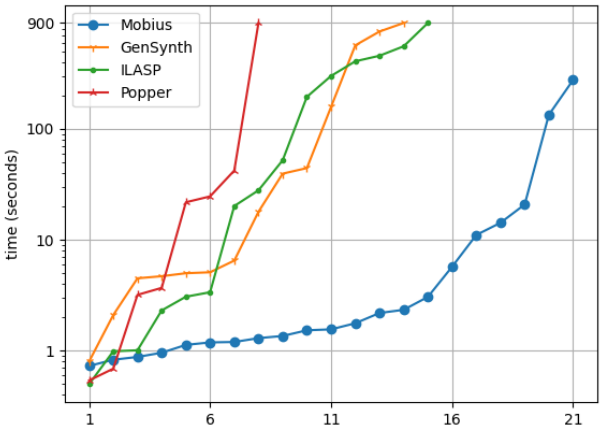
Table 5.2: Table summarizing effectiveness of synthesis. We evaluate Mobius and the three baselines on a suite of 21 benchmarks. All tools are run in single-threaded mode. Mobius successfully synthesizes all benchmarks with an average run-time of 23.1 seconds, while GenSynth, ILASP, and Popper time out on 7 benchmarks each. Note that GenSynth and Popper fail to find a solution for 1 and 7 benchmarks respectively.

Name	**Runtime**			
	Mobius	GenSynth	ILASP	Popper
path	<1	<1	<1	<1
ancestor	**<1**	5.0	timeout	21.8
connected	**1.2**	-	timeout	-
escape	**<1**	4.5	1.0	-
union-find	**1.0**	2.1	20.2	timeout
wikiedits	1.5	157.8	**1.0**	42.1
rsg	**1.8**	39.5	27.9	3.7
sgen	1.3	6.5	2.3	**<1**
zero	11.1	timeout	**3.4**	-
blue-and-green	**1.6**	17.9	3.1	3.2
scc	**2.2**	4.7	timeout	-
monochromatic	14.2	**5.1**	timeout	timeout
andersen	**5.7**	timeout	554.8	timeout
dyck	133.9	565.9	**52.1**	-
modref	**275.8**	timeout	timeout	-
1-call-site	**1.1**	timeout	timeout	timeout
1-object	**1.4**	753.7	299.9	timeout
1-object-1-type	**20.8**	timeout	455.9	24.6
1-type	**3.1**	timeout	timeout	timeout
2-call-site	**1.2**	timeout	406.5	timeout
buildwall	**2.3**	44.3	194.3	-

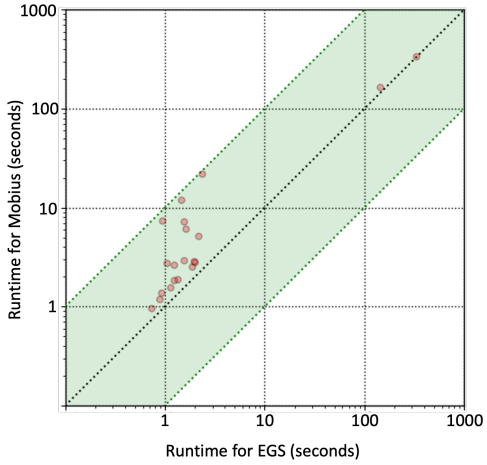
than all three baseline tools. Across the 21 benchmarks, on average, Mobius requires 41%, 76%, and 60% of the time needed by GenSynth, ILASP and Popper respectively. Observe also that it is the only tool which does not timeout on any benchmark, and in several instances is the only system which successfully synthesizes a solution.

We note that Mobius solves all but two synthesis tasks in less than 30 seconds. In the two most expensive benchmarks, modref (a program analysis task) and dyck (matching well-parenthesized strings), more than 85% of the final synthesis time is needed by EGS to produce the seed query in Step 1 of Algorithm 6 (Figure 5.4b shows a breakdown of the time needed for the initial synthesis of non-recursive queries).

Additionally, ILASP and Popper fail to solve 7 and 14 problem instances respectively. For Popper, these failures arise both from running out of time and because of its inability to synthesize invented predicates. In particular, it reports infeasibility for five synthesis tasks, including the SCC query (which requires the invented predicate `path`).



(a) Comparing the performance of MOBIUS, GENSYNTH, ILASP, and POPPER on the suite of 21 benchmarks.



(b) Comparing the total runtime of the end-to-end MOBIUS tool against the time spent in the non-recursive phase that uses EGS for the suite of 21 benchmarks.

Figure 5.4: Summary of the runtime of MOBIUS on the benchmark suite for the effectiveness study. MOBIUS outperforms the state-of-the-art baselines GENSYNTH, ILASP, and POPPER. The synthesis time of MOBIUS is split between the non-recursive phase where we use EGS and the generalization phase where we use a provenance-guided search.

5.5.4. Generalizability

Next, we asked whether the generalization algorithm improves the accuracy of learned queries when they are applied to previously unseen datasets. In order to determine the empirical accuracy of Mobius and to compare it to that of EGS and GenSynth, we focused on three synthesis tasks involving graph properties: `path`, `connected`, and `scc`. The two vertices x and y are related by `connected`(x, y) if there is either a path from x to y or a path from y to x. It is therefore similar to SCC, with the top-level conjunction instead replaced with a disjunction.

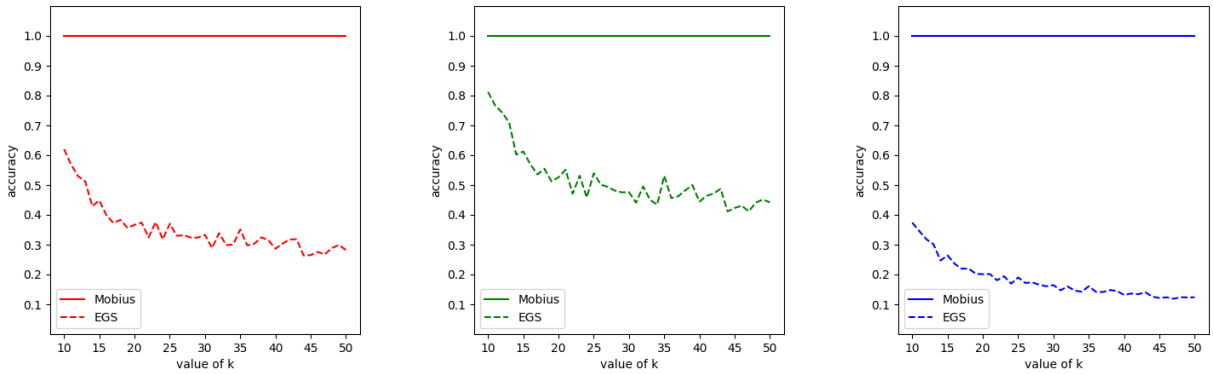
We used the same training data as in Section 5.5.3. The test data is generated by sampling graphs of increasing size (ranging from 10 to 50 vertices) that contain Hamiltonian cycles. The Hamiltonian cycle ensures that all sampled graphs are non-trivial, i.e., $\emptyset \subsetneq$ `path`, `connected`, `scc` $\subsetneq V \times V$. The experiment is repeated 10 times and the mean values are reported in Figure 5.5.

The queries learned by EGS only achieve at most 62%, 81%, and 37% test accuracy in each of the three queries. This is unsurprising: because the test graph is larger than any of the training graphs, it is unlikely for non-recursive queries learned by EGS to achieve perfect accuracy in test.

Two expected trends are also evident from this: First, the test accuracy of EGS decreases as the size of the input graph increases. On average, the accuracy falls by over half for graphs with 50 vertices compared to those with 10 vertices. Second, complex benchmarks such as `scc` that involve invented predicates show significant drops in accuracy compared to relatively simpler benchmarks such as `path` and `connected`.

On the other hand, observe that Mobius consistently achieves perfect test accuracy consistently. In other words, Algorithm 7 effectively transforms the non-recursive query to produce a query with recursive and invented predicate that generalizes to arbitrarily large unseen data.

Another effect of predicate unification is that the learned queries are smaller than the seed non-recursive queries. Over sufficiently large training datasets, the compressed query stops growing once the Generalize procedure has identified the target concept, while the non-recursive query generated by EGS continues to grow as larger instances of these patterns occur in the training data.



(a) Generalization study for the `path` benchmark.

(b) Generalization study for the `connected` benchmark.

(c) Generalization study for the `scc` benchmark.

Figure 5.5: Summary of the accuracy studies on three graph benchmarks: `path`, `connected`, and `scc`. Observe that MOBIUS achieves perfect accuracy on unseen data as recursion is required to express the target concepts.

5.5.5. Expressibility

As a next example, consider the case of `monochromatic` which captures monochromatic paths in a vertex colored graph. One can interpret this as finding a connected component of friends with a common characteristic such as having visited the same place or having a shared interest. This benchmark requires *schema invention*, that is, it uses an invented predicate with a schema different from that of the input or output predicates. The following query represents the desired solution:

$$\texttt{path}(x, y, c) \text{ :- } \texttt{edge}(x, y), \texttt{color}(x, c), \texttt{color}(y, c).$$

$$\texttt{path}(x, z, c) \text{ :- } \texttt{path}(x, y, c), \texttt{path}(y, z, c).$$

$$\texttt{monochromatic}(x, y) \text{ :- } \texttt{path}(x, y, c).$$

It uses the invented predicate `path` with arity 3. Notably, `path` does not share a schema with any of the relations described as part of the input-output data. As discussed before, tools such as ILASP and POPPER are limited by requiring the user to specify explicitly specify the invented predicate and its schema. While GENSYNTH can automatically invent predicates, it assumes that the invented predicate shares its schema with either the input or the output predicates, and therefore fails on

benchmarks such as `monochromatic` which require schema invention.

On the other hand, MOBIUS can effectively invent the predicate `path` with arity 3 and synthesize the desired program. This example highlights our two key contributions:

1. The two-phased approach of first synthesizing a non-recursive query and then generalizing it allows for synthesis of queries with recursive and invented predicates (including those with new schemas), and

2. The end-to-end synthesis technique is complete (as proved in Theorem 5.3.1) and does not require additional supervision from the user in terms of mode declarations or schemas for invented predicates.

5.5.6. Convergence

Finally, we ask whether accounting for data provenance in the generalization process improves convergence time. We construct a variation of Algorithm 7 where the assignment in Step 2(c)ii is instead replaced by:

$$\phi := \phi \wedge \bigvee_{v} (v \neq \mu(v)),$$

where v ranges over all variables currently in context. In other words, we prohibit the constraint solver from producing the same unification map μ in future, but do not perform failure analysis or generalization of any kind.

We ran this modified algorithm on all 20 benchmarks with the same 15 minute time limit as before. In this setting, the algorithm only succeeds on 4 of the 20 synthesis tasks: path, ancestor, union-find, and escape. Observe that all of these are variations of transitive closure. The non-recursive seed queries produced by EGS were correspondingly small and had fewer invented predicates. This greatly reduced the size of the search space, making an exhaustive search feasible. On the other hand, for most other benchmarks, we conclude that provenance-guided generalization is crucial for successful termination.

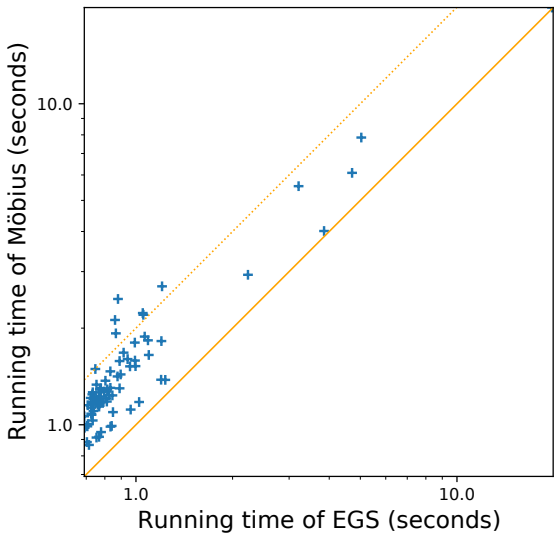


Figure 5.6: Comparison of the running time of EGS and Mobius on a suite of 79 non-recursive benchmarks. Because Mobius begins with seed queries produced by EGS, all points are naturally to the top-left of the $y = x$ diagonal. The dotted line corresponds to Mobius taking twice the time needed by EGS. Only 6 benchmarks take longer to complete.

5.5.7. Performance on Non-Recursive Benchmarks

The user may occasionally be unaware of whether the intended solution for a problem instance requires the use of recursion. In these cases, they may directly call Mobius in order to synthesize a program, instead of beginning with an exploratory run of a non-recursive query synthesizer. Therefore, in our final experiment, we analyze the performance of Mobius on a suite of 79 non-recursive benchmarks, drawn from the evaluation of EGS in 3.4.1.

We summarize our observations of running time in Figure 5.6. On average, recursive synthesis imposes only a 57% time overhead, and in all but 6 of the benchmarks, end-to-end synthesis using Mobius requires less than 2× the time needed for synthesis using EGS.

Additionally, it is possible that Mobius generalizes the non-recursive program to a more succinct non-recursive program using an invented predicate. We see this in the case of generating the *grandparent* relation. EGS generates:

```
grandparent(x, z) :- mother(x, y), mother(y, z).
```

```
grandparent(x, z) :- mother(x, y), father(y, z).
grandparent(x, z) :- father(x, y), mother(y, z).
grandparent(x, z) :- father(x, y), father(y, z).
```

While this is a correct solution, MOBIUS generalizes it by inventing a predicate corresponding to the *parent* relation (denoted below with S), and returns the following solution with the size of the program reduced from 8 to 4:

```
          S(x, y) :- mother(x, y).
          S(x, y) :- father(x, y).
grandparent(x, z) :- S(x, y), S(y, z).
```

CHAPTER 6

Synthesis in Presence of Noise

In many real-world scenarios, data is rarely perfect and often contains some degree of noise. Therefore, it is important to consider synthesis in the presence of noise to ensure that the resulting programs are robust and can perform well even when the input data is not ideal.

When it comes to synthesizing programs in the presence of noisy data, there are several techniques that can be leveraged to mitigate the negative effects of this noise. For example, the input-output data can be pre-processed through data cleaning. By removing or correcting incorrect or irrelevant data, we can improve the quality of the data set and reduce the impact of noise on the final output. On the other hand techniques such as ensemble learning can improve the overall performance of the system and reduce the impact of individual noisy data points.

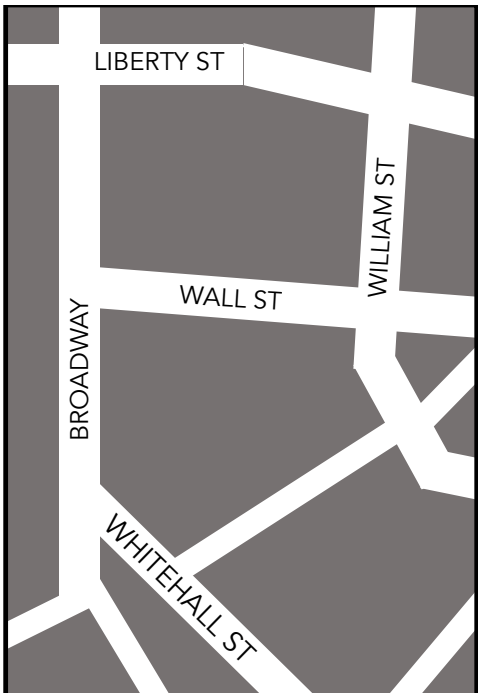
In the context of relational queries, constraint solving tools such as Zaatar, Popper, and ILASP and evolutionary algorithms such as GenSynth can support learning in presence of noise. Motivated by the parallelization in GenSynth, we adapt Algorithm 1 of Chapter 3 to handle noisy input-output examples during the learning process.

6.1. Problem Formulation

For our setting, we assume that noise can either feature as the occurrence of:

1. Undesirable tuple in positive output examples O^+,
2. Desirable tuples in negative output examples O^-,
3. Missing input example in I, or
4. Additional input example in I

Due to the definitions of positive and negative output examples, we do not need to consider the case when we are missing any tuples from them.



(a)

Intersects	
Broadway	Liberty St
Broadway	Wall St
Broadway	Whitehall
Liberty St	Broadway
Liberty St	William St
Wall St	Broadway
Wall St	William St
Whitehall	Broadway
William St	Liberty St
William St	Wall St

GreenSignal
Broadway
Liberty St
William St
Whitehall

HasTraffic
Broadway
Wall St
William St
Whitehall

Crashes
Broadway
Whitehall
Liberty St

(b)

Figure 6.1: Data describing traffic conditions in a city: (6.1a) Map of the city, (6.1b) and listing of the input and output relations. The output relation `Crashes` has an additional undesirable tuple Liberty St

Consider the example from Section 2.1 with traffic and car crashes in a city as depicted in Figure 6.1. However, observe that compared to Section 2.1, the output table `Crashes` contains an additional undesirable tuple Liberty St, highlighted in italics. In this case, we would like to synthesize the desired query:

$$\texttt{Crashes}(x) \text{ :- } \texttt{Intersects}(x, y), \texttt{HasTraffic}(x), \texttt{HasTraffic}(y),$$
$$\texttt{GreenSignal}(x), \texttt{GreenSignal}(y),$$

However, observe that the query produces the output {Broadway, Whitehall} instead of the desired output {Broadway, Whitehall, Liberty St}, and therefore it is not consistent with the input-output examples. Therefore, in order to support robust learning, we need to first define ϵ-consistency.

Definition 6.1.1 (ϵ-consistency)**.** Given $\epsilon \in [0, 1]$ and input-output examples $E = (I, O^+, O^-)$, a query Q is said to be ϵ-consistent with E if $|O^+ \cap [\![Q]\!](I)| \geq (1 - \epsilon)|O^+|$ and $|O^- \cap [\![Q]\!](I)| \leq \epsilon|O^-|$.

Observe that under this definition, with $\epsilon = \frac{1}{3}$, the above query is partially consistent with the input-output examples in Figure 6.1. With this definition, we can reformulate Problem 2.3.1:

Problem 6.1.2 (Robust Relational Query Synthesis Problem)**.** *Given input tuples I, output tuples partitioned as O^+ and O^-, and an error threshold $\epsilon \in [0, 1]$, return a relational query Q such that Q is ϵ-consistent with (I, O^+, O^-) if such a query exists, and* `unsat` *otherwise.*

6.2. Synthesis Algorithm

In order to support robust relational query synthesis, we modify Algorithm 1 and 3 from Chapter 3 by using the definition of ϵ-consistency as well as parallelization. This section extensively uses notation introduced in Chapter 3.

Algorithm 8 `ExplainCell-R`$(I, R(c), O^-, \epsilon)$, where $t = R(c)$ is an output tuple with a single field. Produces an enumeration context $C \subseteq I$ such that $r_{C \mapsto t}$ is consistent with the example $(I, \{t\}, O^-)$.

---

1. Let $G_I = (D, E)$ be the constant co-occurrence graph as defined in Section 3.1
2. Initialize the priority queue, L:

$$L \coloneqq \{\{t'\} \mid t' \in I \text{ contains the constant } c\}.$$

   Each element $C \in L$ is a subset of the input tuples, $C \subseteq I$.
3. While $L \neq \emptyset$:
   (a) Pick the highest priority element $C \in L$, and remove it from the queue: $L \coloneqq L \setminus \{C\}$.
   (b) If $r_{C \to t}$ is ϵ-consistent with $(I, \{t\}, O^-)$, then return C.
   (c) Otherwise:
      i. Let $N = \{c \in D \mid \exists t' \in C \text{ where } t' \text{ contains } c\}$.
      ii. For each constant $c \in N$, edge $e = c \to^R c'$ in G_I, and for each additional input tuple $t' \in I \setminus C$ which witnesses e, update:

$$L \coloneqq L \cup \{C \cup \{t'\}\}.$$

4. Now, since $L = \emptyset$, return `unsat`.

We first modify Algorithm 1 by introducing the notion of ϵ-consistency, to get Algorithm 8 as described above. Concretely, Algorithm 8 differs from Algorithm 1 only at line 3b. Then, the `ExplainTuple` procedure of Algorithm 2 can be canonically extended to support ϵ-consistency in a

similar way to get the procedure `ExplainTuple-R`.

`ExplainTuple-R`. allows us to tackle three of the four scenarios in which noise may be introduced in the input-output examples, that is, the cases when either there is noise in the input examples, or there is a desirable tuple in the negatively labelled outputs. However, it does not address the case when there may be an undesirable tuple in the positive examples, and in particular, when we make an unfortunate choice for the selected t for `ExplainCell-R`.

For this purpose, we introduce parallelization in Algorithm 3. That is, for each tuple $t \in O^+$ we search for conjunctive queries q_t that may explain t (up to ϵ-consistency), and maintain a set Q that is their union. The entire search can be paused when Q is ϵ-consistent with the given input-output examples. The `EGS-R` procedure of Algorithm 9 realizes the same. The correctness and completeness of Algorithm 9 follows from that of Algorithm 3 in Chapter 3.

Algorithm 9 `EGS-R`(I, O^+, O^-, ϵ). Given an example $M = (I, O^+, O^-)$ and $\epsilon \in [0, 1]$, finds a UCQ Q ϵ-consistent with M if such a query exists, and returns `unsat` otherwise.

1. Initialize Q to be the empty query, $Q \coloneqq \emptyset$.
2. For each tuple $t \in O^+$, in parallel:
   (a) Synthesize an explanation,
   $$C_t = \texttt{ExplainTuple-R}(I, t, O^-, \epsilon),$$
   and construct $q_t = r_{C_t \to t}$. If no solution is found terminate the parallel search for the given t.
   (b) Otherwise, update: $Q \coloneqq Q \cup \{q_t\}$
   (c) If Q is ϵ-consistent with M, terminate all parallel threads and return Q.
3. If Q is not Q is not ϵ-consistent with M, return `unsat`.

6.3. Experimental Evaluation

We have implemented the `EGS-R` algorithm in Python which makes calls to the EGS algorithm from Chapter 3. In this section, we evaluate it to determine the effectiveness of `EGS-R` on synthesis tasks in presence of noise.

We have selected 12 knowledge discovery tasks from Section 3.4.1 that involve 5 or more output tuples. This is because the threshold of error depends on the fraction of noisy tuples to the desired tuples. That is, for benchmarks such as `scheduling` with only one output tuple, adding noise has a

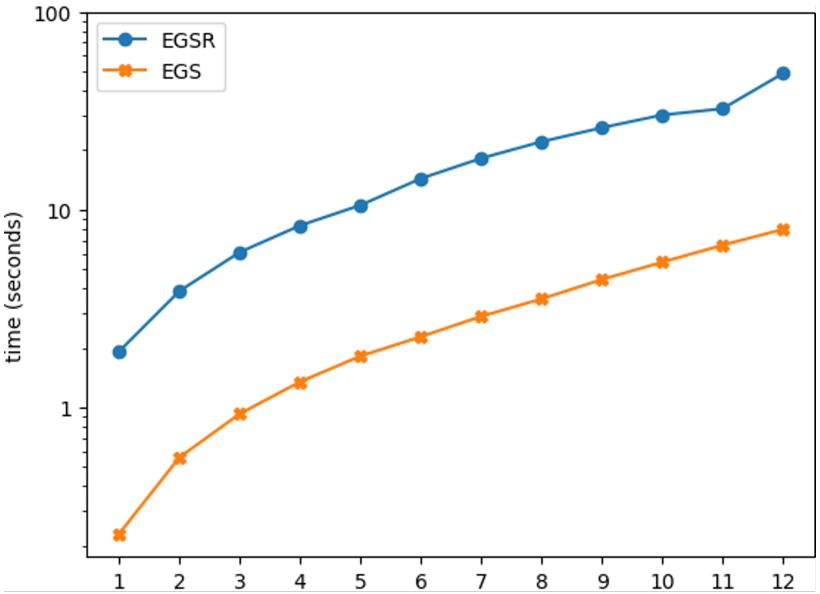


Figure 6.2: Results of our experiments using EGS and `EGS-R` on a suite of 12 knowledge discovery benchmarks. A datapoint (n, t) indicates that the corresponding tool solved n of the benchmarks in less than t time.

grave impact compared to a benchmark such as `inflammation` that has 49 output tuples.

We performed all experiments on a server running Ubuntu 18.04 LTS over the Linux kernel version 4.15.0. The server was equipped with an 18 core, 36 thread Xeon Gold 6154 CPU running at 3 GHz and with 394 GB of RAM. Unlike EGS, `EGS-R` is multi-threaded and hence the results may vary based on the degree of parallelization.

In order to set up the benchmarks, we added to or removed tuples from the input-output examples at random. At any point, the ratio of edits was bounded by 20% across all benchmarks. Then, we ran `EGS-R` on them with a timeout of 600 seconds. In order to compare `EGS-R` with EGS, we also ran EGS on the selected benchmarks (without adding noise).

We observe that the `EGS-R` terminates on all benchmarks and takes an average time of 3.23 seconds even with up to 20% noise. This is a significant blow-up compared to EGS which takes an average of 0.66 seconds on these 18 tasks. This blow-up can be attributed to the tool choosing an erroneous tuple to begin the search as well as the time for the Python implementation of `EGS-R` to call on the Scala implementation of `ExplainCell-R`. In general, we observe a smooth trend across the 12

benchmarks with no outliers.

On qualitatively examining the synthesized queries, we observe that while they are ϵ-consistent with the given input-output examples, they do not necessarily generalize to the desired output. Consider `abduce`, a benchmark that synthesizes the `grandparent` relation using a family tree. Due to the nature of the noise, we end up eliminating a corner case which yields a solution of the form:

$$\texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{father}(z, y).$$
$$\texttt{grandparent}(x, y) \text{ :- } \texttt{father}(x, z), \texttt{mother}(z, y).$$
$$\texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{father}(z, y).$$

And fails to include:

$$\texttt{grandparent}(x, y) \text{ :- } \texttt{mother}(x, z), \texttt{mother}(z, y).$$

This is an example of `EGS-R` over-fitting due to under-generalization. Identifying rules such as the one above when the problem instance is under-specified (here, due to noise) remains a core challenge for programming-by-example Halbert (1984).

CHAPTER 7

Conclusion and Future Work

The work in this thesis proposes methods for example-guided synthesis of relational queries with varying levels of expressiveness and addresses the challenge of learning these queries from a small set of demonstrative input-output examples. The evaluation of the approaches on an expansive set of benchmarks demonstrates its effectiveness in generating efficient and accurate queries. In addition, the experiments have shown that the method is able to handle noisy and incomplete data and can outperform state-of-the-art systems.

The contribution of this work is not only limited to the proposed method, but opens up several possibilities in the domain of programming-by-example, and proposes program synthesis as a promising direction for improving the usability and accessibility of database systems. Additionally, it can be used in the context of knowledge discovery to learn intepretable hypotheses from data.

Example-guided synthesis can find application in designing developer tools that allow non-expert end-users to generate challenging and complex programs by providing only a small set of input-output examples. Another compelling use case is writing program analyses. Information from the analyzed programs can be extracted and represented as relational data. The user can provide a set of output examples (for instance by highlighting the section of code in an IDE), and we can synthesize a hypothesis explaining the highlighted outputs.

Limitations and Future Work

We now discuss a few limitations of the Example-Guided Approach and outline opportunities for future work:

1. *Expressiveness:* While the presented approach targets rich fragments of relational queries with features such as conjunction, negation, recursion, predicate invention, and numerical and categorical comparisons, it currently does not support aggregation.

   However, tools such as Scythe and PatSQL can support more expressive features such as

aggregation, group-by operators, and ordering. This makes them a better fit for the SQL domain. Two ways in which example-guided synthesis can be extended to support aggregation are adding support for user-provided templates and interactive synthesis.

As discussed before, this thesis explores the trade-off between instance-specific supervision and expressibility of synthesis tools. In that sense, seeking additional supervision from the user beyond the input-output examples can help guide the synthesis process. This is especially important for aggregators that lead to *constant invention*, that is in the case of operators like `sum`, `count`, or `average`, when the output can have a constant that does not occur in the input. In such scenarios, an approach that is based on co-occurence of constants is bound to fail.

The second approach relies on the user's ability to provide the outputs of sub-queries of the target query, for example the sub-query to which the aggregator is applied. A co-occurence based approach is useful in such a scenario and effectively bypasses the need to synthesize aggregation.

2. *Scalability:* Another challenge with a data-driven approach like example-guided synthesis is that is heavily depends on the size of the data. As proved in Section 2.4, the query synthesis problem is co-NP complete and hence we do not have any efficient algorithms to solve it. While this is not a major challenge for applications in end-user programming where there are only a small number of input-output examples, the approach is limited for applications such as knowledge discovery where the provided data-set might be expansive.

   A part of that can be mitigated by employing alternative representations of co-occurrence of constants (such as the one in Section 4.1.1), adopting heuristics that expedite the search, as well as identify ways in which the end-user can provide additional supervision to prune the search.

3. *Robustness:* In Chapter 6, we adapt the example-guided technique to support synthesis in presence of noise. However, this is an ad-hoc modification of the synthesis algorithm and requires the user to additionally specify a threshold of noise. As discussed in Section 6.3,

the the introduction of noise may lead to under-specification of the problem instance, in particular, for end-user programming where the size of input-output examples is already small. Secondly, it remains to quantitatively study how the noise in the input examples translate to the requirements for the threshold ϵ that is necessary for the `EGS-R` procedure as well as evaluate the efficacy of `EGS-R` with change in the degree of noise.

4. *Usability:* The above three points add up to the user-experience and usability for the tool. The implementations developed for this thesis are yet to be integrated into one end-to-end tool that supports the diverse set of features detailed here.

   Additionally, the problem formulation requires the user to explicitly specify positive and negative examples. The underlying algorithms can be seconded with a more user-friendly set-up where the problem specification can be provided intuitively and implicitly, for example, by graphical or logical representations.

   Additionally, there are a few instances where the tool may generate an *incorrect* (that is, different from the target) query. In such a case, neither does the tool provide any feedback to the user on how to better specify the problem instance and nor can the user provide any feedback to the synthesizer to revise the solution. Adding support for feedback can significantly improve the usability of the tool.

   In general, while the dissertation outlines the core algorithmic ideas for example-guided synthesis of relational queries, it also opens up opportunities to develop usable and user-oriented applications.

5. *Under-generalization:* As discussed in Section 6.3, and in general, complex benchmarks with small training data suffer from under-generalization. This is because the variety of scenarios covered by the training data may be insufficient to completely convey user intent, resulting into over-fitting queries which are not general enough. As such, this is a well-known limitation of synthesis tools operating in the programming-by-example (PBE) paradigm Halbert (1984). One can overcome this limitation by using a richer set of input-output examples that cover all

features of the target query.

6. *Over-generalization:* Conversely, the synthesis algorithm might discover spurious patterns in the training data which allow it to further compress the learned queries. Analogous to the previous limitation, this occurs when the user fails to provide sufficiently many negative examples to preclude overfitting. Providing a representative set of examples can become burdensome in the case of complex concepts. Interactive program synthesis Le et al. (2017) has the potential to overcome this limitation. Exploring the space of tradeoffs between fully automatic synthesis and the inclusion of a user who actively guides and refines the synthesis process is an exciting direction of future research.

7. *Greedy Hueristics:* Throughout the algorithms presented in this dissertation, we adopt greedy hueristics and optimizations. For example, the implementation of the priority queue in Algorithm 1 and the normalization strategies in Mobius. While these heuristics improve the performance of the tool, they can compromise guarantees such as minimality of the solution. Based on the choice of local decisions made by the tools, the synthesized query may also lead to an over-fitting query, a non-minimal query, or a query that, while equivalent to user intent, may be challenging to interpret due to gratuitous predicates. A potential future direction is to study the impact of these heuristics and develop robust alternatives to them.

BIBLIOGRAPHY

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Pearson, 1st edition, 1994.

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of Datalog programs. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, 2017.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2016.

Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. *SIGPLAN Not.*, 50(6):218–228, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737952. URL https://doi.org/10.1145/2813885.2737952.

David Bohan, Geoffrey Caron-Lormier, Stephen Muggleton, Alan Raybould, and Alireza Tamaddoni-Nezhad. Automated discovery of food webs from ecological data using logic-based machine learning. *PLOS One*, 6(12), 2011.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009. ISSN 1931-7883.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL https://doi.org/10.1145/362384.362685.

Andrew Cropper and Sebastian Dumančić. Inductive Logic Programming at 30: A new introduction. *Journal of Artificial Intelligence Research*, 74, 2022. URL https://www.jair.org/index.php/jair/article/view/13507.

Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110:801–856, 2021.

C Date. *SQL and Relational Theory: How to Write Accurate SQL Code*. O'Reilly Media, Inc., 2009. ISBN 0596523068.

Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-sql evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, July 2018. URL http://aclweb.org/anthology/P18-1033.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018.

Todd J. Green. LogiQL: A declarative language for enterprise applications. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2015.

J. Grzymala-Busse. Selected algorithms of machine learning from examples. *Fundam. Informaticae*, 18, 1993.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926423. URL https://doi.org/10.1145/1925844.1926423.

Isabelle Guyon, Masoud Nikravesh, Steve Gunn, and Lotfi A. Zadeh, editors. *Feature Extraction*. Springer Berlin Heidelberg, 2006. doi: 10.1007/978-3-540-35488-8. URL https://doi.org/10.1007/978-3-540-35488-8.

Daniel Conrad Halbert. *Programming by Example*. PhD thesis, 1984. AAI8512843.

Tomasz Imieliński and Witold Lipski. The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, 28(1):80–102, 1984. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(84)90077-1. URL https://www.sciencedirect.com/science/article/pii/0022000084900771.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, 2017. URL http://www.aclweb.org/anthology/P17-1089.

Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. To join or not to join? thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 19–34, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2882952. URL https://doi.org/10.1145/2882903.2882952.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, 2014.

Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for inductive learning of answer set programs. *CoRR*, abs/2005.00904, 2020a.

Mark Law, Alessandra Russo, and Krysia Broda. The ILASP system for inductive learning of answer set programs. *CoRR*, abs/2005.00904, 2020b.

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. Interactive program synthesis. *ArXiv*, abs/1703.03539, 2017.

Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11), November 2009.

M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. GenSynth: synthesizing datalog programs without language bias. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2021.

Ana Milanova, Atanas Rountev, and Barbara Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2002, pages 1–11. ACM, 2002. ISBN 1-58113-562-9.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *New Generation Computing*. Academic Press, 1990.

Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. *Machine Learning*, 100(1), 2015.

Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)*, 2021a.

Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 84–99, 2021b.

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. 34(3), 2009.

A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 179–190, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75293. URL https://doi.org/10.1145/75277.75293.

David Poole. Logic programming for robot control. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, 1995.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1), 1986.

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2020a.

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2020b.

Jiwon Seo. Datalog extensions for bioinformatic data analysis. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1303–1306. IEEE, 2018.

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD. ACM*, 2016.

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of Datalog programs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing Datalog programs using numerical relaxation. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI 2019, pages 6117–6124. AAAI Press, 2019. ISBN 978-0-9992411-4-1.

Detlef Sieling. Minimization of decision trees is hard to approximate. *Journal of Computer and System Sciences*, 74(3):394–403, May 2008. doi: 10.1016/j.jcss.2007.06.014. URL https://doi.org/10.1016/j.jcss.2007.06.014.

Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE, pages 292–303. IEEE, 2019. doi: 10.1109/ICSE.2019.00044. URL https://doi.org/10.1109/ICSE.2019.00044.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2011, pages 17–30. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926390. URL http://doi.acm.org/10.1145/1926385.1926390.

Jiang Su and Harry Zhang. A fast decision tree learning algorithm. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, page 500–505. AAAI Press, 2006. ISBN 9781577352815.

Shan Suthaharan. *Decision Tree Learning*, pages 237–269. Springer US, Boston, MA, 2016. ISBN 978-1-4899-7641-3. doi: 10.1007/978-1-4899-7641-3_10. URL https://doi.org/10.1007/978-1-4899-7641-3_10.

Keita Takenouchi, Takashi Ishio, Joji Okada, and Yuji Sakata. Patsql: Efficient synthesis of sql queries from example tables with quick inference of projected columns. *Proc. VLDB Endow.*, 14(11):1937–1949, jul 2021. ISSN 2150-8097. doi: 10.14778/3476249.3476253. URL https://doi.org/10.14778/3476249.3476253.

Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-guided synthesis of relational queries. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1110–1125, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454098. URL https://doi.org/10.1145/3453483.3454098.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2017a.

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466. ACM, 2017b. ISBN 978-1-4503-4988-8.

Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, and Dan Suciu. Optimizing recursive queries with progam synthesis. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 79–93, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3517827. URL https://doi.org/10.1145/3514221.3517827.

John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2004, pages 131–144. ACM, 2004. ISBN 1-58113-807-5. doi: 10.1145/996841.996859. URL http://doi.acm.org/10.1145/996841.996859.

Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand,

and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, December 2007. doi: 10.1007/s10115-007-0114-2. URL https://doi.org/10.1007/s10115-007-0114-2.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*, 2018.

John M. Zelle and Raymond J. Mooney. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, pages 1050–1055, 1996. URL http://dl.acm.org/citation.cfm?id=1864519.1864543.

David Zhao, Pavle Subotic, and Bernhard Scholz. Debugging large-scale datalog: A scalable provenance evaluation strategy. *ACM Trans. Program. Lang. Syst.*, 42(2), 2020.

APPENDIX A

Run-time Comparisons

Table A.1: Performance of EGS, Scythe, ILASP, and ProSynth on 20 knowledge discovery benchmarks.

Benchmark	**EGS**	**Scythe**	**ILASP** *Task-Agnostic Rule Set*	**ILASP** *Task-Specific Rule Set*	**ProSynth** *Task-Agnostic Rule Set*	**ProSynth** *Task-Specific Rule Set*	**#Rules** *Task-Agnostic*	**#Rules** *Task-Specific*
Knowledge Discovery								
abduce	0.4	–	–	6.1	–	–	–	4917
adjacent-to-red	0.4	1.5	365.7	0.3	–	0.8	209799	101
agent	0.8	–	–	0.3	–	1.8	–	142
animals	0.4	–	–	–	–	–	1242184	2000
cliquer	0.3	0.7	1.0	0.2	–	0.7	1484	79
contains	0.3	0.8	176.1	0.2	–	0.1	7557	1
grandparent	0.5	–	–	5.9	–	–	–	4917
graph-coloring	0.4	5.2	177.2	0.1	–	0.3	96079	23
headquarters	0.3	0.7	11.2	0.2	–	0.1	4057	1
inflammation	0.6	–	–	3.0	–	–	–	847
kinship	0.5	–	–	5.8	–	–	–	4917
predecessor	0.2	1.7	1.2	0.2	–	0.1	1484	5
reduce	0.3	0.7	114.8	0.1	–	0.1	7557	1
scheduling	0.4	1.5	336.7	0.1	–	0.2	160016	16
sequential	0.8	–	–	–	–	–	–	–
ship	0.3	1.3	–	1.2	–	–	–	1426
son	0.3	1.1	–	1.0	–	–	–	1199
traffic	0.5	6.5	143.9	0.3	–	0.7	93326	97
trains	0.4	–	–	3.3	–	–	–	601
undirected-edge	0.3	1.0	1.3	0.2	–	0.3	1484	79

Table A.2: Performance of EGS, Scythe, ILASP, and ProSynth on 18 program analysis benchmarks.

Benchmark	**EGS**	**Scythe**	**ILASP** *Task-Agnostic Rule Set*	**ILASP** *Task-Specific Rule Set*	**Prosynth** *Task-Agnostic Rule Set*	**Prosynth** *Task-Specific Rule Set*	**#Rules** *Full*	**#Rules** *Task-Specific*
Program Analysis								
arithmetic-error	0.2	1.0	–	0.1	–	0.1	263853	13
block-succ	0.4	–	–	9.9	–	–	–	9758
callsize	0.3	1.2	20.2	0.2	–	0.4	14446	11
cast-immutable	0.3	1.2	420.0	0.1	–	0.1	225108	18
downcast	1.8	–	–	–	–	–	–	3392
increment-float	0.3	1.6	58.5	0.1	–	0.1	19594	10
int-field	0.3	0.5	–	0.3	–	0.2	–	109
modifies-global	0.3	0.7	23.3	0.1	–	0.1	17679	6
mutual-recursion	0.3	1.3	1.2	0.2	–	0.1	1484	25
nested-loops	2.9	–	–	1.1	–	–	–	1053
overrides	0.3	1.2	–	1.6	–	–	–	1804
polysite	3.8	–	–	–	–	–	–	1025
pyfunc-mutable	0.4	2.0	17.0	0.2	–	0.1	12185	6
reach	0.3	1.0	545.3	0.3	–	0.2	256549	15
reaching-def	0.2	0.8	–	0.3	–	0.1	–	8
realloc-misuse	0.4	–	–	0.1	–	0.2	669744	22
rvcheck	0.6	–	–	29.0	–	–	–	20186
shadowed-var	0.3	1.8	–	0.3	–	0.2	13291	38

Table A.3: Performance of EGS, Scythe, ILASP, and ProSynth on 41 database querying tasks.

Benchmark	**EGS**	**Scythe**	**ILASP** *Task-Agnostic Rule Set*	**ILASP** *Task-Specific Rule Set*	**Prosynth** *Task-Agnostic Rule Set*	**Prosynth** *Task-Specific Rule Set*	**#Rules** *Task-Agnostic*	**#Rules** *Task-Specific*
Relational Queries								
sql01	0.4	1.4	108.3	0.3	–	2.7	82475	200
sql02	0.2	1.5	21.0	0.3	–	1.9	22073	212
sql03	0.4	4.7	–	1.2	–	22.6	381295	752
sql04	0.4	3.3	–	0.2	–	0.1	763408	2
sql05	0.2	2.5	4.6	0.2	–	0.1	1571	1
sql06	0.3	1.2	–	0.5	–	0.1	–	21
sql07	0.6	3.7	–	0.2	–	0.3	258271	36
sql08	0.3	–	21.4	0.1	–	0.1	14415	11
sql09	0.4	2.1	–	0.4	–	0.1	–	8
sql10	0.3	8.4	1.4	0.1	2.8	0.1	331	1
sql11	0.2	49.7	40.8	0.2	–	0.1	14415	11
sql12	0.3	4.1	–	0.2	–	0.1	–	2
sql13	0.3	3.0	–	0.1	–	0.1	86032	3
sql14	0.4	2.3	–	0.2	–	0.1	182739	8
sql15	0.6	2.4	–	1.1	–	–	–	1461
sql16	0.4	10.4	–	0.6	–	0.1	–	3
sql17	0.3	4.8	–	0.2	–	0.1	187020	2
sql18	0.2	2.1	31.9	0.1	–	0.1	14403	1
sql19	0.5	3.1	–	1.7	–	–	–	1832
sql20	0.2	1.5	0.8	0.2	5.5	0.1	344	1
sql21	0.3	3.5	325.0	0.1	–	0.1	86032	3
sql22	1.9	6.3	92.5	0.2	–	1.1	54821	51
sql23	0.3	6.3	–	0.1	–	0.1	2037	2
sql24	0.2	1.4	10.9	0.1	–	0.1	1958	2
sql25	0.4	17.0	13.4	0.1	–	0.3	8946	9
sql26	0.3	14.7	11.2	0.1	–	0.1	4445	4
sql27	0.5	5.9	22.6	0.1	–	0.2	13810	18
sql28	0.3	8.2	403.5	0.2	–	0.1	181232	22
sql29	0.3	1.0	–	0.3	–	0.2	–	76
sql30	0.3	3.1	–	0.3	–	0.1	763408	2
sql31	0.4	2.1	73.1	0.2	–	2.5	53813	166
sql32	0.3	17.0	418.3	0.1	–	0.1	225108	18
sql33	0.4	2.8	–	4.6	–	–	–	6632
sql34	0.2	3.2	540.3	0.1	–	0.1	225108	18
sql35	0.7	–	–	0.5	–	0.1	–	4
sql36	32.6	199.8	–	–	–	–	–	247986
sql37	0.7	12.7	–	–	–	–	–	–
sql38	0.5	–	22.0	0.3	–	0.2	13810	18
sql39	6.8	11.7	–	4.9	–	1.7	–	325
sql40	0.3	6.5	–	–	–	–	–	559577
sql41	0.2	3.9	–	0.1	–	0.1	–	44

APPENDIX B

Quality of Synthesized Programs

Here we describe the output of EGS and Scythe on three knowledge-discovery benchmarks: `adjacent-to-red`, `graph-coloring`, and `scheduling`.

adjacent-to-red

The target program identifies vertices neighboring red colored vertices. It can be expressed as:

$$\texttt{target}(v) \text{ :- } \texttt{edge}(v, v'), \texttt{color}(v', c), \texttt{red}(c).$$

Scythe synthesizes the following query:

```
Select t4.Col_02 From (Select t2.Col_0, t2.Col_01, t2.Col_1,
  t2.Col_02, t2.Col_11, t6.Col_0 As Col_03 From (Select *From
  (Select t5.Col_0, t5.Col_01, t5.Col_1, t1.Col_0 As Col_02,
  t1.Col_1 As Col_11 From (Select * From (Select Red.Col_0,
  t3.Col_0 As Col_01, t3.Col_1 From Red Join Color As t3) As
  t5 Where t5.Col_0 = t5.Col_1) Join Edge As t1) As t2
  Where t2.Col_01 = t2.Col_11) Join Green As t6) As t4;
```

graph-coloring

The target program identifies mistakes in vertex coloring, that is, it identifies vertices v which have neighbors v' with the same color. It can be expressed as:

$$\texttt{target}(v) \text{ :- } \texttt{edge}(v, v'), \texttt{color}(v, c), \texttt{color}(v', c).$$

Scythe synthesizes the following query:

```
Select t3.Col_01 From (Select t1.Col_0, t1.Col_1, t1.Col_01,
  t1.Col_11, t2.Col_0 As Col_02, t2.Col_1 As Col_12 From
```

```
(Select * From (Select Color.Col_0, Color.Col_1, t4.Col_0
As Col_01, t4.Col_1 As Col_11 From Color Join Color As t4)
As t1 Where t1.Col_1 = t1.Col_11) Join Edge As t2) As t3
Where t3.Col_0 = t3.Col_12 And t3.Col_01 = t3.Col_02;
```

scheduling

The target program identifies scheduling conflicts, that is, it identifies days d on which for some time slot s, two distinct classes c and c' are scheduled. It can be expressed as:

$$\texttt{conflict}(d) \text{ :- } \texttt{assigned}(d, s, c), \texttt{assigned}(d, s, c'), \neg(c = c').$$

SCYTHE synthesizes the following query:

```
Select t3.Col_0 From (Select t1.Col_0, t1.Col_1, t1.Col_2,
  t1.Col_01, t1.Col_11, t1.Col_21, t4.Col_0 As Col_02,
  t4.Col_1 As Col_12, t4.Col_2 As Col_22 From (Select *
  From (Select Assigned.Col_0, Assigned.Col_1, Assigned.Col_2,
  t2.Col_0 As Col_01, t2.Col_1 As Col_11, t2.Col_2 As Col_21
  From Assigned Join Assigned As t2) As t1 Where
  t1.Col_0 = t1.Col_01 And t1.Col_1 < t1.Col_11) Join
  Assigned As t4) As t3 Where t3.Col_11 < t3.Col_12 And
  t3.Col_2 = t3.Col_22;
```