GENSYNTH: Synthesizing Datalog Programs without Language Bias

Jonathan Mendelson\textsuperscript{1*}, Aaditya Naik\textsuperscript{1*}, Mukund Raghothaman\textsuperscript{2}, Mayur Naik\textsuperscript{1}

\textsuperscript{1} University of Pennsylvania
\textsuperscript{2} University of Southern California
\{jonom,asnaik\}@seas.upenn.edu, raghotha@usc.edu, mhnnaik@seas.upenn.edu

Abstract

Existing techniques for learning logic programs from data typically rely on language bias mechanisms to restrict the hypothesis space. These methods are therefore limited by the user’s ability to tune them such that the hypothesis space is simultaneously large enough to include the target program but small enough to admit a tractable search. We propose a technique to learn Datalog programs from input-output examples without requiring the user to specify any language bias. It employs an evolutionary search strategy that mutates candidate programs and evaluates their fitness on the examples using an off-the-shelf Datalog interpreter. We have implemented our approach in a tool called GENSYNTH and evaluate it on diverse tasks from knowledge discovery, program analysis, and relational queries. Our experiments show that GENSYNTH can learn correct programs from few examples, including for tasks that require recursion and invented predicates, and is robust to noise.

1 Introduction

The problem of learning logic programs from input-output data has been widely studied in artificial intelligence, formal methods, and machine learning. Such programs offer a variety of benefits by virtue of being explainable, interpretable, generalizable, verifiable, and composable.

Datalog (Abiteboul, Hull, and Vianu 1994), a logic programming language, is commonly targeted due to its rich expressivity, declarative rule-based semantics, and efficient implementations. In this setting, the input-output data are specified in the form of tuples over finite relations; the goal is to synthesize a Datalog program that, when executed on the given input tuples, produces the given output tuples.

Figure 1 shows an example task in which the input data is a binary relation \textit{edge} encoding edges in a directed graph, and the output data is a binary relation \textit{scc} representing pairs of nodes in the input graph that belong to the same strongly connected component (SCC). A correct and concise solution is the following recursive program:

\begin{align*}
\text{path}(x,y) &::= \text{edge}(x,y). \\
\text{path}(x,y) &::= \text{edge}(x,z), \text{path}(z,y). \\
\text{scc}(x,y) &::= \text{path}(x,y), \text{path}(y,x).
\end{align*}

The first rule defines the base case for the predicate \textit{path}, stating that any \textit{edge} from \(x\) to \(y\) implies a \textit{path} from \(x\) to \(y\). The second rule defines the inductive step for \textit{path}: an edge from \(x\) to \(z\) and a path from \(z\) to \(y\) implies a path from \(x\) to \(y\). Finally, the third rule states that a path from \(x\) to \(y\) and a path from \(y\) to \(x\) implies that \(x\) and \(y\) are in the same \textit{scc}.

Note that this solution is non-trivial, as it is a recursive program requiring complex joins and an \textit{invented predicate} \textit{path}. An invented predicate is a hidden intermediate concept often necessary for synthesis but not specified in the schema or input-output example.

Existing approaches to this problem are broadly classified into Inductive Logic Programming (ILP), e.g. Metagol (Muggleton 1991); Answer Set Programming (ASP), e.g. ILASP3 (Law 2018); program synthesis, e.g. ProSynth (Raghothaman et al. 2020); and neural learning, e.g. NTP (Rocktaschel and Riedel 2017). Despite using notably different search techniques, however, all of these approaches rely on various language bias mechanisms or restrictions on expressiveness to limit the hypothesis space. We draw comparisons between various contemporary tools in Table 1.

Approaches with significant language bias mechanisms, such as metarules (Metagol), candidate rules (ProSynth), or templates (NTP) run quickly only under a carefully crafted small set of these entities, hereafter called templates. This limits the considerable user burden of authoring the templates which then fundamentally biases the tool toward a specific subset of programs that the author has in mind. Since the runtime of these template-based tools becomes impractical far before they can consider a large sample space, these approaches are critically limited by the user’s ability to strike a balance when providing templates: too many and the tool times out, too few and it fails to synthesize a solution.
Within ILASP, explicitly specify the arity and typing of the programs. Without substantial background information about the problem or knowledge of a potential solution, it would be extremely difficult to craft a small set of templates that contains these three rules. In practice, even the ordering of the metarules significantly impacts Metagol’s performance; oftentimes an unlucky ordering will result in Metagol timing out. When using rule enumeration techniques, tuning the hyperparameters of an enumerator for a specific benchmark to guide the search space is still time consuming and difficult. ProSynth times out on the SCC task when the enumerator is not provided with benchmark-specific hyperparameters obtained either through knowledge of the solution or through tedious trial and error. Finally, template-based neural approaches, such as NTP, fare no better. NTP requires the user to specify not only templates, but how often a template rule should be instantiated and suffers from the same bias and runtime issues as the other approaches.

ILASP3 does not require metarules, but still has language bias in the form of modes, which restrict how often predicates may appear in a clause. Furthermore, ILASP3 is biased due to its support of prescriptive invented predicates; it can only synthesize invented predicates if their schema has been specified in advance. Given that it is not always obvious if an invented predicate is even needed, this is a non-trivial task (Cropper, Dumancic, and Muggleton 2020). Under the hood, ILASP3 first generates candidate rules before running the search algorithm, so it is burdened by the same issues that afflict Metagol, ProSynth, or NTP. In practice, ILASP3 takes about 4 times longer on SCC than GENSYNTH, even after we, within ILASP, explicitly specify the arity and typing of the path invented predicate and enable the anti-reflexive and positive settings.

Popper has no language bias but it cannot handle predicate invention. Thus its search space is limited not by its language bias but by its severe restrictions on expressiveness. Popper is not publicly available for comparison but would not be able to synthesize at least 12 of the 42 noise-free benchmarks in our evaluation which need invented predicates.

**Our Approach.** We introduce GENSYNTH\(^1\), a template-free end-to-end Datalog synthesis tool. By end-to-end, we mean that only an input-output example and input-output schema are provided; in our case, there are no templates, meta-rules, meta-programs, or modes to specify. Furthermore, GENSYNTH automatically synthesizes invented predicates and is therefore free from the bias introduced by approaches that use prescriptive invented predicates. Despite such an unconstrained search space, GENSYNTH is able to generate small and interpretable solutions to Datalog problems, including non-trivial ones like `scc`.

We frame the synthesis task as a search problem through the space of Datalog programs—a very complex surface with a sparse fitness function and riddled with local minima. We depict the algorithm by following one program throughout the run using the SCC example in Figure 2. The algorithm consists of an *accretion phase*, where accretions mutate a program until it has the desired fitness, and a *reduction phase*, where reductions mutate the program decreasing its size but maintaining its fitness. The accretion phase is inspired, in part, by how humans write programs: they start with a simple piece of code and make small changes each time they desire a new feature or encounter a bug.

Our first insight is to combine the search with the rule generation, which occurs implicitly as a result of the mutations. This allows us to prune most of the search space dynamically as the algorithm progresses rather than pruning the search space manually up front, as in template-based approaches. For example, the set of programs explored at generation 4 is very highly constrained; all offspring of this program are similar to the parent. This is a good way to constrain the search space, since we have learned by A4 that this space of programs is likely to have a high fitness. This constrained search space then makes it much more likely that we reach the program with fitness 1.0 found at generation A5.

Another insight is to use mutations to traverse a very complex and coarse surface in an effective way. Differentiable approaches such as Difflog (Si et al. 2019) or neural approaches using SGD struggle as they become trapped in local minima. Using mutations, which only slightly modify the program, makes it likely that a parent and offspring have similar fitness scores, but mutations strung together allow for leaps over local minima.

We also aim to maximize throughput; bottom-up tools like ProSynth that run thousands of candidate rules simultaneously suffer from poor scalability as they overwhelm the Datalog interpreter. GENSYNTH, on the other hand, runs mostly very small programs. It further maximizes throughput by considering only the space of valid programs and taking advantage of its parallelism.

\(^{1}\)Available at https://jonomendelson.github.io/gensynth/

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Definite</th>
<th>Datalog</th>
<th>Datalog</th>
<th>ASP</th>
<th>Definite</th>
<th>Datalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language bias</td>
<td>Metarules</td>
<td>Candidate rules</td>
<td>Templates</td>
<td>Modes</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Predicate invention</td>
<td>✓</td>
<td>○</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Recursion</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Noise handling</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1: Comparison of state-of-the-art tools. Metagol and Popper can also synthesize Datalog programs since Datalog programs are definite. GenSynth and Metagol support automatic predicate invention, whereas ProSynth, NTP, and ILASP3 only support prescriptive predicate invention, in which the schema of all invented predicates are specified.
Finally, the reduction phase is crucial for interpretability, as illustrated in Figure 2. While in generation A5 we have derived a correct program, it is difficult to understand, has vestigial code, and may even overfit the example. The reduction phase, by syntactically and possibly semantically modifying the program, aims to reduce the size of the learned program with the desired fitness value on the training data. We note that the synthesized program may reference invented some argument of the head.

In summary, these insights together make GENSYNTH very effective at quickly synthesizing highly expressive, interpretable programs without language bias.

2 Algorithm

Formally, GENSYNTH takes as input a set of relation schemas $R$ which is divided into the input relations $R_{in} \subseteq R$, and an output relation $r_{out} \in R$. The schema of each relation $r(T_1, T_2, \ldots, T_k)$ describes its arity $k$ and the types of each column $T_i$. The training data consists of a set of input tuples $I$ which populate the input relations, a set of desirable output tuples $O^+$, and a set of undesired output tuples $O^-$ such that $O^+ \cap O^- = \emptyset$. Finally, it also takes as input the fitness threshold $0 \leq f_T \leq 1$. If successful, it returns a Datalog program with the desired fitness value on the training data. We note that the synthesized program may reference invented relations $r_{in} \not\in R$.

As discussed in Section 1, GENSYNTH consists of an accretion phase where it discovers an initial target program with the desired fitness score, followed by a reduction phase in which it attempts to reduce the size of the learned program. We describe these algorithms in Algorithms 2 and 3 respectively. Informally, both procedures are instances of evolutionary algorithms (Fogel 2006) which repeatedly apply mutations to a population of candidate programs, until they discover a program with the desired properties—with adequate fitness score, and with minimal size, respectively.

Throughout the algorithm, we ensure that all programs in the population remain valid programs. Each clause in a valid program must be well-typed, such that the arguments respect the schema of the relations, grounded, such that all arguments that appear in the head must appear at least once in the body, and connected, where all literals must have at least one argument which can be chained, directly or indirectly to some argument of the head.

...
Algorithms 1, 2, and 3 are designed to maintain these invariants throughout execution.

2.1 Clause Generation, Accretion, and Reduction
Recall that the accretion algorithm repeatedly mutates the programs in the population until it achieves the desired fitness score. It starts with a list of \( c \) seed programs, each of which is a small valid program generated by an invocation of the CreateClause method, formally defined in Algorithm 1. In each generation, it selects the best-performing \( s \cdot c \) programs, where \( 0 < s < 1 \), and repeatedly mutates each selected program to restore the population size. Each mutation is itself a sequence of \( n \) element mutations, described in Section 2.3.

The CreateClause procedure is a three-phase process: It starts with the schema of the output relation, and fixes the list of relations appearing in the body of the clause by greedily picking input relations which can bind as many output variables as possible. It then walks through the list of literals, and randomly assigns variable names while ensuring that all conditions for a valid program are met. This algorithm is used to generate the seed programs, in Step 2 of Algorithm 2, as well as in the Append Clause mutation as described in Section 2.3. While it is possible that the seeds generated by the CreateClause procedure may be isomorphic, subsequent mutations in Algorithm 2 will cause them to diverge.

Next, observe that the reduction algorithm closely follows the structure of the accretion phase, with a few notable differences. First, while the initial programs of Algorithm 2 are randomly generated by calls to the CreateClause method, the reduction algorithm initializes its population with \( \beta \) copies of \( \bar{p} \). It follows that the population of programs is therefore a list with possible duplicates rather than a classical set. Second, it sorts the programs by size rather than by fitness score in Equation 2, with the condition that all programs have fitness scores surpassing the threshold. The purpose of the reductive mutations is therefore to reduce the size of the learned program, rather than necessarily improve fitness. Third, at each step, it applies the reductive mutations to generate an offspring program that has never been included in the population before. Finally, in contrast to the accretive mutations, which maintain or increase the size of the program, the reductive mutations reduce or maintain the size of the program to which they are applied. We catalog these mutations in Section 2.3. As a result, it is a self-limiting process guaranteed to terminate in Step 2c, when the 1-step reductions are unable to discover any as-yet-unseen offspring.

2.2 Population-Specific Fitness Functions
One curious aspect of Algorithm 2 is that different populations use different values of \( \beta \) to track the programs under consideration. More precisely, for each population, we sample \( \beta' \sim (0, 1) \) uniformly at random between 0 and 1. Then with probability 0.5, we choose \( \beta = \beta' \), and with probability 0.5, we choose \( \beta = 1/\beta' \).

Recall that the training data consisted of the input tuples \( I \), desired output tuples \( O^+ \), and undesired output tuples \( O^- \). Consider a program \( p \) which, when applied to the input tuples \( I \), produces the output tuples \( p(I) = O \), with \( \text{TP} = |O^+ \cap O^+| \) true positives. In this case, its fitness score \( F_\beta(p) \) is defined as the \( \beta \)-weighted harmonic mean of its precision \( \text{TP}/|O| \) and recall \( \text{TP}/|O^+| \):

\[
F_\beta(p) = \frac{(1 + \beta^2) \cdot \text{TP}}{|O \cap (O^+ \cup O^-)| + \beta^2 \cdot |O^+|}
\]

(3)

As different populations are using slightly different fitness functions, it reduces the chance of all populations simultaneously getting stuck in local maxima. Regardless of this, Step 3c of Algorithm 2 allows the procedure to terminate only if the \( F_1 \) score of the program exceeds the cutoff.

2.3 Mutations
In this section, we describe the six accretive mutations and six reductive mutations used by Algorithms 2 and 3, respectively. Crucially, the mutations are designed such that the offspring produced by any mutation is necessarily a valid program as defined in the introduction to Section 2.

A candidate program is a collection of Datalog clauses of
Algorithm 2 Accrete($R_I, O^+, O^-, f_T$). Given a set of relation names $R$ and training data $(I, O^+, O^-)$, returns a program $\hat{p}$ with fitness score $F_1(\hat{p}) \geq f_T$.

Run $b$ independent populations in parallel. Within each, do:
1. Choose the fitness function $F_2$ as described in Section 2.2.
2. Initialize the list of seed programs $P$ with $c$ calls to the randomized CreateClause($R_{in}, r_{out}$) method.
3. Repeat forever:
   (a) Selection event: Sort the programs $p \in P$ in descending order of their fitness scores $F_2(p)$, and update:
   $$P := [P_1, P_2, \ldots, P_{|s\cdot c|}],$$
   where $s$ is the fraction of programs which survive each selection event, and $c$ is a user-provided limit on the population size.
   (b) Proliferation sub-phase: Produce $(1 - s)/s$ offspring for each program $p \in P$:
      i. Select the number of mutations $n$ to be applied to $p$ by sampling from a distribution, $n \sim B$ (defined in the supplementary material).
      ii. Initialize $p_0 = p$, and for each $i \in \mathbb{N}$, let $p_{i+1} := Mutate(p_i)$.
      iii. Update: $P := P \cup \{p_n\}$.
   (c) Termination: If there is a program $\hat{p} \in P$ such that $F_1(\hat{p}) \geq f_T$, terminate all populations and return $\hat{p}$.

the form $r_0 : - r_1, \ldots, r_n$, where $r_i$ is a relation with arguments $(a_1, \ldots, a_k)$.

**Accretive Mutations.** Step 3b of Algorithm 2 applies a sequence of accretive mutations to produce offspring in each generation. These mutations aim to increase the size of the candidate program.

1. **Append Clause:** Create a new clause using the CreateClause method and append this clause to the candidate program.
2. **Append Literal:** Randomly pick a clause. Append a random input, invented, or output literal to the body of this clause. When assigning arguments to the new literal, choose fresh and existing arguments with equal probability such that the clause is valid.
3. **Extend:** Randomly pick a clause $r_0 : - r_1, \ldots, r_i, \ldots, r_n$, and within this clause, randomly pick a literal $r_l$ with arguments $(a_{i0}, \ldots, a_{ik}, \ldots, a_{im})$. Introduce a fresh argument $a^{l_i}_k$. Append a random input, invented, or output literal $r_j$ to the body of this clause with arguments $(a_{j0}, \ldots, a_{jk}, \ldots, a_{jp})$. Replace argument $a_{ik}$ with $a^{l_i}_k$ such that $r_l$ has arguments $(a_{i0}, \ldots, a^{l_i}_k, \ldots, a_{im})$. The resulting clause is $r_0 : - r_1, \ldots, r_i, \ldots, r_j, r_n$.
4. **Swap:** Randomly pick a clause. Swap the location of two randomly chosen arguments in the body of the clause.
5. **Invent:** Randomly pick a clause and randomly select a literal from the body of the clause, where the clause is

Algorithm 3 Reduce($R, I, O^+, O^-, f_T, \hat{p}$). Returns a reduced program $p^*$ such that $F_1(p^*) \geq F_1(\hat{p})$.

Run $b$ independent populations in parallel. Within each, do:
1. Initialize the list of seed programs $P$ with $c$ copies of $\hat{p}$.
2. Repeat forever:
   (a) Selection event: Let $P' = \{p \in P \mid f(p) \geq f_T\}$ be the list of programs with acceptable fitness scores. Sort the programs in increasing order of their size, and update:
   $$P := [P'_1, P'_2, \ldots, P'_k],$$
   where $k = \min(|s \cdot c|, |P'|)$, and as before, $s$ is the fraction of programs which survive each selection event.
   (b) Proliferation sub-phase: Repeat $(1 - s)/s$ times for each program $p \in P$:
      i. Choose a random mutation type $m$ and let $M_{p,m}$ be the set of 1-step mutants obtained by applying $m$ to $p$.
      ii. Let the set of ancestors, $C$ be the set of all programs which inhabited $P$ at any time.
      iii. If $M_{p,m} \not\subseteq C$, pick a mutant $p' \in M_{p,m} \setminus C$ uniformly at random, and append $p'$ to $P$.
   (c) Termination: If no new programs were added during the proliferation sub-phase, then terminate this population, and return the smallest program $p \in P$.

Let $p_i$ be the program returned by the $i$-th population. Let $p^*$ be the smallest program in $\{p_1, p_2, \ldots, p_b\}$. Return $p^*$.

$r_0 : - r_1, \ldots, r_i, \ldots, r_n$ and $r_i$ is a randomly selected literal. Replace $r_i$ with a new invented predicate $r_{inv}$ that has the same schema and arguments as $r_i$ such that the original clause becomes $r_0 : - r_1, \ldots, r_{inv}, \ldots, r_n$. Add a new clause to the program $r_{inv} : - r_i$ with head $r_{inv}$ and body $r_i$.

6. **Recurse:** Randomly pick a clause and randomly select a literal from the body of the clause, where the clause is $r_0 : - r_1, \ldots, r_{inv}, \ldots, r_n$ and $r_i$ is the randomly selected literal. Replace $r_i$ with a new invented predicate $r_{inv}$ that has the same schema as $r_i$ such that the original clause becomes $r_0 : - r_1, \ldots, r_{inv}, \ldots, r_n$. Then add two new clauses:
   (a) One “base case” clause where $r_{inv} : - r_i$.
   (b) One “recursive step” clause where $r_{inv} : - r_i, r_{inv}$ such that $r_i$ contains at least one argument appearing in the head and $r_{inv}$ and $r_i$ share a newly introduced argument that does not appear in the head.

Observe that the Invent mutation can only synthesize invented predicates whose schemas coincide with that of an input or output relation. While this is indeed a limitation, we have never encountered a task in practice that has required an invented predicate with a distinct schema.

Note also that the accretive mutations are designed such that the algorithm will eventually reach a consistent solution with probability 1 if such a solution exists, given a large enough choice of the peak mutation length. This follows from the observations that: (a) any clause which is currently producing undesired output tuples can be deactivated by appending sufficiently many literals, and (b) any target clause
can be created by a suitable combination of the Append Clause and Append Literal mutations.

**Reductive Mutations.** Step 2b of Algorithm 3 applies a sequence of reductive mutations to produce offspring each generation. These mutations aim to simplify the candidate program by decreasing its size.

1. **Remove Repeating Clauses:** If the candidate program contains two identical clauses, remove one of the clauses.
2. **Remove Repeating Literals:** If there exists some clause with two identical literals with identical arguments, remove one of the literals.
3. **Reduce Invented Predicate:** Randomly pick a clause $r_0 : - r_1, ... , r_n$ such that $r_{inv}$ is a non-recursive, one-clause invented predicate where $r_{inv} : - ir_1, ... , ir_k$. Replace $r_{inv}$ in the body of the clause with the body of $r_{inv}$ so that the original clause becomes $r_0 : - r_1, ... , ir_1, ... , ir_k, ... , r_n$. Update arguments in the new clause appropriately so that the original and new programs are semantically equivalent.
4. **Minimize Clauses:** Randomly pick a clause and remove it from the candidate program.
5. **Minimize Literals:** Randomly pick a clause. Within the body of the clause, randomly pick a literal and remove it from the clause.
6. **Minimize Arguments:** Randomly pick a clause. Randomly choose two arguments that appear in the clause, $a_i$ and $a_j$, where $a_i \neq a_j$. Replace all instances of $a_j$ with $a_i$ throughout the clause.

The first three mutations above are equivalence-preserving and therefore do not alter the overall fitness score of the candidate programs. The remaining mutations, which remove clauses, literals, and arguments, may alter the fitness score of the candidate programs. However, Algorithm 3 ultimately only applies mutations such that the fitness score is either preserved or improved.

### 3 Evaluation

In this section, we experimentally evaluate GENSYNTH with respect to the following criteria:

1. **Effectiveness:** How does GENSYNTH compare to existing approaches that use different kinds of language bias?
2. **Generality:** How does GENSYNTH perform on diverse tasks compared to a state-of-the-art approach?
3. **Robustness:** How does GENSYNTH perform on noisy data compared to a state-of-the-art approach?
4. **Scalability:** How does GENSYNTH scale with the size of the data and the amount of available parallelism?

All experiments were run on a Ubuntu 18.04 server with an 18 core Intel Xeon 3 GHz processor and 394 GB memory. We use the following hyperparameters in our experiments:

1. Number of populations: $b = 32$.
2. Population size: $t = 150$.
3. Selection ratio: $s = 0.2$.

4. Number of mutations in each step: $n \sim B$, where $B = \text{Bin}(n, p)$ is a binomial distribution with $n = \lfloor 15s \rfloor$ and $p = 0.3$. Both $c_1$ and $c_2$ are sampled uniformly at random between 0 and 1.

#### 3.1 Effectiveness

We study the effectiveness of GENSYNTH by considering non-trivial Datalog programs compared to three contemporary approaches: Metagol, a meta-interpretive learning system using a top-down Prolog interpreter; ProSynth, a program synthesizer using a bottom-up Datalog interpreter; and ILASP3, an inductive Answer System Program learning system.

**Setup.** We compare these tools on Andersen, a popular program analysis for statically reasoning about pointer aliasing in programs written in languages like C and Java. The task consists of 4 input relations and 19 input-output tuples. Since Andersen is noise-free, we require a solution with an F1 score of 1.0. We choose Andersen as a representative from the 42 tasks used in Section 3.2 as its solution consists of multiple recursive clauses, making it a challenging task:

\[
\begin{align*}
pt(x, y) & : - \text{addr}(x, y), \\
pt(x, y) & : - \text{addr}(x, z), \text{pt}(z, y), \\
pt(x, y) & : - \text{load}(x, z), \text{pt}(z, w), \text{pt}(w, y), \\
pt(x, y) & : - \text{store}(z, w), \text{pt}(z, x), \text{pt}(w, y).
\end{align*}
\]

As previously stated, Metagol and ILASP3 require a choice of metarules and are further influenced by either the choice of orderings of rule bodies or mode declarations. So we use a single representative benchmark for this comparison as it would be difficult to fairly set up these tools across 42 tasks.

**Methodology.** We compare the time taken by GENSYNTH, Metagol, ProSynth, and ILASP3 on Andersen. We describe each tool’s instantiation using as $(n, r, t)$ where $n$ is the number of templates, $r$ is the number of times the tool was run, and $t$ is the number of runs that timed out in 1 hour.

Metagol’s runtime depends on the ordering of the templates. We thus provide it with the exact four templates needed to learn the solution but order the predicates of each template differently. We run ProSynth with the 64 templates used in (Raghothaman et al. 2020). We also run it with a more natural set of templates generated using the algorithm from (Si et al. 2018), which constructs them by mutating a small number of chain rule templates. Lastly, ILASP3 requires a search space of templates induced via mode declarations that specify how many times each predicate can occur in a rule’s body. We run it with 28,237 templates—the minimum number possible via mode declarations. The templates used by ProSynth and ILASP3 were grounded, whereas the metarules used by Metagol were not.

**Results.** The results are shown in Figure 3. GENSYNTH does not time out on any of its 8 runs and synthesizes the solution within 5 minutes on average. On the other hand, Metagol times out on 54 out of the 72 runs, despite providing it with the minimum set of templates. ProSynth is able to synthesize the solution quickly when using 64 templates, but its performance suffers with the larger choice of templates. ILASP3 solves the benchmark in 48 minutes on average.
We require solutions with an F1 score of 1.0. We run both GenSynth and ProSynth on 42 tasks. Most notably, GenSynth, which never times out, has a clear advantage once the tasks become more difficult, as ProSynth times out on 11 out of the 42 tasks.

Interpretability is a major advantage of program synthesis approaches; producing small and easily readable programs is a large part of their usefulness. We observe from Figure 4b that GenSynth always produces a program with fewer than 10 predicates, and always returns a smaller solution than ProSynth. Note that we define program size as total number of atoms that appear in the body of each rule in the program. This shows that ProSynth often overfits on the data given to it and produces uninterpretable programs. GenSynth’s reduction phase is a large part of the reason why it produces such interpretable programs, and, on average, it accounts for a 43% decrease in the size of programs.

For instance, compare GenSynth’s program for SCC:

\[
\begin{align*}
\text{inv}(x,y) & : - \text{edge}(x,y). \\
\text{inv}(x,y) & : - \text{inv}(x,z), \text{edge}(z,y). \\
\text{sc}(x,y) & : - \text{inv}(x,y), \text{inv}(y,x).
\end{align*}
\]

an easily interpretable solution, with that of ProSynth’s:

\[
\begin{align*}
\text{sc}(x,z) & : - \text{sc}(x,y), \text{inv}(x,z). \\
\text{sc}(x,z) & : - \text{sc}(x,y), \text{inv}(z,y). \\
\text{sc}(y,z) & : - \text{edge}(x,y), \text{inv}(y,z). \\
\text{sc}(y,z) & : - \text{edge}(x,y), \text{edge}(z,x). \\
\text{sc}(y,z) & : - \text{sc}(x,y), \text{inv}(x,z).
\end{align*}
\]

which is nearly triple in size and highly overfits the task.

Thus we see that GenSynth not only produces programs more efficiently, but produces higher quality programs. Despite a lack of templates, it is able to solve a diverse range of tasks. On the other hand, template-based approaches often timeout when too many templates are provided, and even when a program is synthesized, it is potentially of poor quality due to high syntactic bias.

### 3.3 Robustness

We investigate whether GenSynth is resilient to noise and how its resilience compares to existing approaches. Neural approaches naturally handle noise, so we compare it to a state-of-the-art neural approach, the Neural Theorem Prover (NTP).

NTP is a differentiable learning system based on dense vector representations of symbols. We do not compare to Metagol or ProSynth as they are both unable to handle noise.

#### Setup.

We use the Countries benchmark as it is the most difficult of the NTP benchmarks (Rocktäschel and Riedel 2017). It consists of 244 countries, 23 subregions, 5 regions, and 1,158 geographical facts \(\text{locatedIn}(x,y)\) and \(\text{neighborOf}(x,y)\). The countries are split into 198 training, 24 validation, and 24 testing countries. This data was obtained from NTP’s GitHub repository, and we use the same sets of present and missing data for both tools. However, since GenSynth is type-conscious, we further partition the \text{locatedIn} relation into \text{locatedInCR}(c,r), \text{locatedInSR}(s,r), and \text{locatedInCS}(c,s) where \(c\) is a country, \(r\) is a region, and \(s\) is a subregion. Note that we
Results of the scalability experiment. We compare the F1 scores of the problems S2 and S3 while taking less time. A closer look reveals that the difference in F1 scores, especially in S3, is mainly due to the fact that NTP is restricted by templates provided to it:

\begin{verbatim}
3 #1(X,Y):= #1(Y,X).
3 #1(X,Y):= #2(X,Z), #2(Z,Y).
3 #1(X,Y):= #2(X,Z), #3(Z,Y).
3 #1(X,Y):= #2(Z,X), #3(Z,Y), #4(W, Y).
\end{verbatim}

These templates, which specify variable bindings and even how often each will be instantiated, were crafted to perfectly match the following expected solution:

\begin{verbatim}
neighborOf(x,y) :- neighborOf(y,x).
locatedIn(x,y) :- locatedIn(x,z), locatedIn(z,y).
locatedIn(x,y) :- neighborOf(x,z), locatedIn(z,y).
locatedIn(x,y) :- neighborOf(x,z),
neighborOf(z,w), locatedIn(w,y).
\end{verbatim}

However, the expected solution is not, in fact, globally optimal. Since GENSYNTH is not dependent on templates, it finds the following more optimal solution, which also scores a higher F1 score on the test dataset:

\begin{verbatim}
locatedInCR_out(x,y) :- neighborOf(z,x),
locatedInCS(z,w), locatedInSR(w,y).
locatedInCR_out(x,y) :- locatedInCR(x,y).
\end{verbatim}

### 3.4 Scalability

We next investigate how GENSYNTH’s running time is affected by the size of the input-output data and the number of threads. Again, since we test on a noise-free benchmark, we require a solution with an F1 score of 1.0.

**Setup.** We consider the SCC benchmark from the set of 42 tasks described in Section 3.1. This benchmark contains one relation $\text{Edge}$ with 10 tuples. We use SCC since it is easier to control its size while still remaining a complex benchmark requiring recursion and invented predicates.

**Methodology.** We create three variants of the SCC benchmark: 1x, 10x, and 100x, containing 10, 100 and 1,000 tuples respectively, the latter two variants representing unions of multiple disjoint graphs. We then run each of these variants using different numbers of threads, each 8 times, and take the median of these 8 runs. We require that all runs simulate the same number of populations so that the quality of result is not affected. Note then that runs with fewer threads must simulate some populations in sequence.

**Results.** Figure 4c shows the result of this experiment. We observe that GENSYNTH scales very well over size of input-
output data, with only about a 5x slowdown in synthesis time for an 100x increase in input-output data size. Almost all of this slowdown occurs in the Datalog interpreter, Souffle, which generally accounts for over 90% of the running time of GENSYNTH. Since GENSYNTH only interacts with the output of the interpreter, it is possible to use faster interpreters than Souffle to better handle large input-output data.

We also observe that GENSYNTH benefits from its parallelism immensely; since populations can be run independently of one another, we are able to consider many more candidate programs than non-parallelizable approaches.

3.5 Limitations

We discuss some limitations of GENSYNTH in aspects of expressiveness, termination, and efficiency. First, GENSYNTH does not support aggregation operations (e.g., sum and count) nor negation, although such operations could be incorporated as carefully crafted mutations with some effort. Second, as Algorithm 2 indicates, it runs until it finds a solution with an F1-Score above the desired threshold. This means that GENSYNTH is potentially non-terminating, especially in cases where a solution above the threshold does not exist. However, we have not experienced such non-termination in practice. Third, GENSYNTH incurs heavy resource consumption due to its significant use of parallelization. Moreover, since GENSYNTH interacts with a Datalog solver, a significant amount of time is spent in I/O processing.

4 Related Work

Cropper, Dumancic, and Muggleton (2020) provide a comprehensive survey of the relevant literature over the last three decades. We briefly discuss and compare representative works in ILP, ASP, program synthesis, and neural learning.

Genetic Approaches. The idea of using genetic approaches for program synthesis has been explored in previous works. For example, (Wong and Leung 1997) alters the derivation tree of the current candidate program at each step with cross-over being the primary genetic operation. (Tamaddoni-Nezhad and Muggleton 2002) starts with a seed clause and considers mutations which merge variables. This can be seen as a more sophisticated version of our minimize arguments mutation. On the other hand, both these approaches require some form of background knowledge and lack a reduction phase, which GENSYNTH uses as a regularization mechanism to prevent the synthesized program from overfitting. Finally, (Wu 2019) describes several directions for future research, such as learning of existential rules, rules with negation and aggregation, and selection of the fitness function.

ILP and ASP. ILP techniques take besides input-output examples the background knowledge in the form of a logic program. They target Prolog which is more expressive than Datalog. Older ILP systems such as FOIL and Progol work by bottom clause construction (Muggleton 1995) and struggle to synthesize recursive programs, especially from few examples. Modern ILP systems such as Metagol overcome this limitation by using meta-interpretive learning (Muggleton, Lin, and Tamaddoni-Nezhad 2015) but require the user to provide templates. Lastly, compared to GENSYNTH, Metagol supports higher-order programs, but cannot handle noise.

ASP programs are declarative akin to Datalog programs but more expressive. Modern ASP systems such as ILASP (Law, Russo, and Broda 2020) and FastLAS (Law et al. 2020) can handle noise, but still require language bias in the form of mode declarations, which also specify a recall—the maximum number of times that declaration can be used in each rule. Popper (Cropper and Morel 2020), a more recent system which combines ILP with ASP, requires predicate declarations for invented predicates and cannot handle noise.

Program Synthesis. These techniques are based on enumerative search, such as ALPS (Si et al. 2018), or constraint solving, such as Zaatar (Albarghouthi et al. 2017), or hybrid, such as ProSynth (Raghothaman et al. 2020). ALPS and ProSynth search for the target program as a subset of templates whereas Zaatar encodes the templates as an SMT formula whose solution yields the target program. Besides the language bias, they cannot handle noise, and cannot exploit parallelism as easily as GENSYNTH. GENSYNTH also produces smaller and more interpretable programs. On the other hand, these techniques are more efficient at learning from failures and pruning the search space than GENSYNTH.

Neural Learning. Recent works cast logic program synthesis as a neural learning problem in order to handle tasks that involve noise or require subsymbolic reasoning. These works differ from GENSYNTH in a few key aspects.

NeuralLP (Yang, Yang, and Cohen 2017), NLM (Dong et al. 2019), and δILP (Evans and Grefenstette 2018) model relation joins as a form of matrix multiplication, which limits them to binary relations. NTP (Rottke and Riedel 2017) constructs a neural network as a learnable proof (or derivation) for each output tuple up to a predefined depth (e.g. $\leq 2$) with a few (e.g. $\leq 4$) templates, where the network could be exponentially large when the depth or number of templates grows. The predefined depth and a small number of templates could significantly limit the class of learned programs. In contrast, GENSYNTH can synthesize programs with relations of arbitrary arity, and supports rich features like recursion and predicate invention. Lastly, neural approaches face challenges of generalizability and data efficiency.

Difflog (Si et al. 2019) overcomes the above hurdles but scales poorly by reasoning about all candidate programs simultaneously, which not only overwhelms the Datalog solver but also requires MCMC-based random sampling to avoid being stuck in local minima in the complex search surface.

5 Conclusion

We proposed a technique and tool, called GENSYNTH, to learn Datalog programs from input-output examples. GENSYNTH overcomes the need for the user to tune language bias mechanisms or restrict expressiveness. It employs an evolutionary search strategy that effectively navigates the unconstrained space of programs by mutating them and evaluating their fitness on the examples using an off-the-shelf Datalog interpreter. We demonstrated the ability of GENSYNTH to learn correct programs in diverse domains from few examples, including for tasks that require recursion and invented predicates, and in the presence of noise.
Acknowledgements

We thank the anonymous reviewers for providing insightful feedback. This research was supported by grants from NSF (#1836822), DARPA (#FA8750-19-2-0201), ONR (#N00014-18-1-2021), AFRL (#FA8750-20-2-0501), and Facebook.

References


