Metric Program Synthesis

JOHN FESER, MIT, USA
ISIL DILLIG, UT Austin, USA
ARMANDO SOLAR-LEZAMA, MIT, USA

We present a new domain-agnostic synthesis technique for generating programs from input-output examples. Our method, called *metric program synthesis*, relaxes the well-known *observational equivalence* idea (used widely in bottom-up enumerative synthesis) into a weaker notion of *observational similarity*, with the goal of reducing the search space that the synthesizer needs to explore. Our method clusters programs into equivalence classes based on a *distance metric* and constructs a version space that compactly represents "approximately correct" programs. Then, given a "close enough" program sampled from this version space, our approach uses a distance-guided repair algorithm to find a program that exactly matches the given input-output examples. We have implemented our proposed metric program synthesis technique in a tool called SyMetric and evaluate it in three different domains considered in prior work. Our evaluation shows that SyMetric outperforms other domain-agnostic synthesizers that use observational equivalence and that it achieves results competitive with domain-specific synthesizers that are either designed for or trained on those domains.

1 INTRODUCTION

Programming-by-example (PBE) is a program synthesis task wherein the goal is to learn a program in some domain-specific language (DSL) that is consistent with a given set of input-output examples. Because of its potential to democratize programming for computer end-users, PBE has attracted enormous attention from several research communities and has found a number of useful applications ranging from string and table transformations [Feng et al. 2017; Gulwani 2011] to question answering [Chen et al. 2021] to computer-aided design (CAD) [Du et al. 2018].

While there are many different techniques to solve the PBE problem, existing solutions can be classified as either being *domain-agnostic* or *domain-specific*. Domain-specific techniques (e.g., [Chen et al. 2021, 2020; Du et al. 2018; Feng et al. 2017; Feser et al. 2015; Gulwani 2011; Wang et al. 2017a]) are specialized to a particular DSL and target a pre-defined class of synthesis tasks. Domain-agnostic techniques [Feng et al. 2018; Solar-Lezama et al. 2006; Wang et al. 2018] are parameterized over a DSL and can, in principle, be applied to a wide variety of domains.

A common domain-agnostic solution to the PBE problem is to perform *bottom-up enumeration* over DSL programs [Miltner et al. 2022; Udupa et al. 2013; Wang et al. 2018]. The idea is to start with primitives in the DSL and build up increasingly more complex programs by combining existing terms via DSL constructs. To scale up this approach to non-trivial synthesis tasks, PBE techniques based on bottom-up enumeration leverage the concept of *observational equivalence*: Programs that produce the same output on the given set of input examples are effectively identical (at least for PBE purposes); hence, it suffices to keep only one representative of such observationally equivalent programs. For example, synthesis techniques based on *finite tree automata (FTA)* leverage this observational equivalence idea to build a compact version space representing the set of all programs consistent with the given input-output examples.

Authors' addresses: John Feser, MIT, CSAIL, 32 Vassar St, Cambridge, USA, feser@mit.edu; Isil Dillig, UT Austin, Austin, USA, isil@cs.utexas.edu; Armando Solar-Lezama, MIT, CSAIL, 32 Vassar St, Cambridge, USA, asolar@mit.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

While this observational equivalence idea can significantly reduce the search space in some domains, it is not as effective in scenarios where few programs share the same (relevant) input-output behavior. As an example, consider the *inverse CSG*[1] *problem*, where the goal is to "de-compile" a complex geometric shape into a set of geometric operations that were used to construct it in a computer aided design (CAD) system [Du et al. 2018; Willis et al. 2021]. While this problem can be framed as a PBE task [Du et al. 2018], it is difficult to tackle this problem using domain-agnostic PBE engines because few programs produce *exactly* the same image. However, even in such domains, there are often many programs that have very similar, but not exactly identical, input-output behaviors. This observation motivates the following question: Can we relax the observational equivalence criterion and develop a synthesis algorithm that exploits the *semantic similarity* between different programs?

In this paper, we answer this question affirmatively and present a new PBE algorithm called *metric program synthesis* that can be applied in many domains. Our method relaxes the standard *observational equivalence* criterion to a weaker one called *observational similarity* and groups together programs that produce *similar* outputs on the same input. In particular, given a distance metric δ, our method clusters programs into the same equivalence class if their output is within an ϵ radius with respect to δ. Thus, in domains like inverse CSG where few DSL programs are observationally equivalent but many have *similar* input-output behaviors, such distance-based clustering can reduce the search space much more substantially than existing techniques.

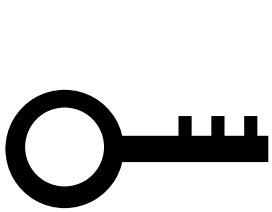
To exploit observational similarity, our metric program synthesis approach proceeds in two phases: First, it performs bottom-up enumerative synthesis to build a *version space* [Lau et al. 2003] that compactly represents all programs up to some fixed AST depth. During bottom-up enumeration, it clusters programs into equivalence classes using the provided distance metric, keeping one representative of each equivalence class. However, due to the use of distance-based clustering, the generated version space is *approximate*: it contains many programs that are incorrect but *close to* being correct. To deal with this difficulty, our method combines version space construction with a second *local search* step: Starting with a program P whose output is close to the goal, it performs hill-climbing search to find a syntactic perturbation P' of P that has the intended input-output behavior. Because it is often possible to find *syntactically* similar representations of *semantically* similar programs in many domains, this combination of approximate version space construction with local program repair makes our approach effective.

We have implemented the proposed approach in a tool called SyMetric and evaluate it on three different domains considered in prior work, namely (1) inverse CSG [Du et al. 2018], (2) regular expression synthesis [Chen et al. 2020; Lee et al. 2016], and (3) tower building [Ellis et al. 2021]. Our evaluation shows that SyMetric is competitive with synthesizers designed/trained for these domains and that it outperforms other domain-agnostic synthesizers based on the observational equivalence idea.

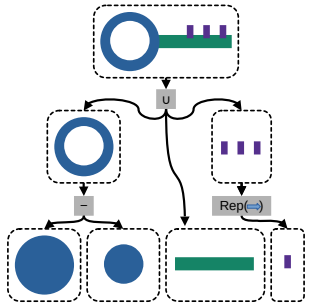
To summarize, this paper makes the following key contributions:

- We introduce *metric program synthesis*, a new synthesis approach that exploits the semantic and syntactic similarity of programs in the search space.
- We show how to use distance metrics to perform effective clustering, ranking, and repair of programs explored during synthesis.
- We implement our technique in a new tool called SyMetric and evaluate it in three different application domains, comparing SyMetric to several relevant baselines.

[1] Constructive Solid Geometry



(a) The input image.



(b) The solution, illustrated.

$$(\texttt{Circle}(4,8,4) - \texttt{Circle}(4,8,3))$$
$$\cup\, \texttt{Rect}(7,7,15,9)$$
$$\cup\, \texttt{Repeat}(\texttt{Rect}(10,9,11,10),2,0,3)$$

(c) Program text of the solution.

Fig. 1. An example CSG problem.

2 OVERVIEW

In this section, we work through an example from the inverse CSG domain that illustrates the key ideas in our algorithm.

Consider the picture of a key shown in Figure 1. To a human, it is clear that this image contains three important component pieces: circles to make up the handle of the key, a rectangle for the shaft, and evenly spaced rectangles for the teeth. We can write a program that generates this key in a simple DSL that includes primitive circles and rectangles as well as union, difference, and repetition operators. In particular, Figure 1c shows a program that can generate this picture. The program composes the three shapes: a hollow circle for the handle of the key, a rectangle for the shaft, and three small, evenly spaced rectangles for the teeth. The hollow circle is constructed by subtracting a small circle from a larger one:

$$\texttt{Circle}(x, y, r) - \texttt{Circle}(x', y', r')$$

The evenly spaced teeth are constructed by replicating a small rectangle three times:

$$\texttt{Repeat}(\texttt{Rect}(x, y, x', y'), dx, dy, 3)$$

Circles are specified by a center point and radius, and rectangles are specified by their lower left and upper right corners.

Suppose that our goal is to synthesize the program in Figure 1c given *just* the picture in Figure 1a. As mentioned in Section 1, one standard approach is to perform bottom-up search over programs in the DSL: that is, create programs by composing together smaller terms, but discarding those that create the same shape as a previously-encountered one. This approach—known as bottom-up enumeration with equivalence reduction [Udupa et al. 2013]—is a simple but powerful domain-agnostic synthesis algorithm that works well in many domains.

However, the problem is that the inverse CSG domain is full of programs that are *similar*, but not identical. To see why, consider the two circles that make up the handle of the key. If the outer circle was slightly larger or slightly shifted, it would still be clear to us that the image is only slightly perturbed. We would be able to fix the program by locally improving it — i.e., shifting the circle back into place by changing its parameters. It should not be necessary to retain both programs in the search space, since it is straightforward to transform one into the other. However, we cannot use equivalence reduction to group these two programs together, even though our intuition tells us that they should be nearly interchangeable.

To synthesize the figure above, our algorithm proceeds in two phases: It first performs coarse-grained search to look for a program P that is *close to* matching the target image. Then, in the second phase, it applies perturbations to P in order to find a repair that *exactly* matches the given image. We now explain these two phases in more detail.

Global coarse-grained search: The first phase of our algorithm is based on bottom-up search and, like prior work [Wang et al. 2018], it builds a data structure that compactly represents a large space of programs. In particular, we represent the space of programs using a variant of a finite tree automaton (FTA) called an *approximate finite tree automaton* (XFTA), which is described in detail in section 3.3. The key idea behind an XFTA is to group together values that are semantically similar: in the CSG context, this means that images that are sufficiently similar to each other are represented using the same state in the automaton.

Our method constructs such an approximate tree automaton in three phases, namely *expansion*, *grouping*, and *ranking*. In the expansion phase, operators are applied to sub-programs to create new candidate programs. For our running example, the first expansion step generates the set of primitive shapes. Later expansions compose shapes together using Boolean operators and the looping operator `Repeat` to create images of increasing complexity, but many of the images generated during the expansion phase are similar to each other.

In the grouping phase, images are put into clusters. Each cluster has a center c and a radius ϵ such that every image in the cluster is within distance ϵ of c. Although every image in the cluster is retained as part of the search space, only the center of the cluster participates in further expansion steps. This clustering phase is essentially a relaxed version of equivalence reduction.

Finally, in the ranking phase, the w clusters that are closest to the goal image are retained. This focuses the search on the programs that are likely to produce the goal. After ranking, the top w clusters are inserted as new states into the XFTA, and the operators that produced each state in the cluster are inserted as edges.

When the forward search terminates, the XFTA represents a space of programs that are close to the target image, meaning that they produce visually similar images. However, at this point, a new difficulty presents itself: If we only clustered programs that had equivalent behavior, we could simply check whether the target image is present in the XFTA. If it is, then the corresponding program can be extracted from the automaton and the synthesis task is complete. But, because we cluster programs that have similar but not the same behavior, we may complete XFTA construction and find that the target image is not present in the automaton, even though that there are several images that are *close* to the goal. To address this issue, our method performs a second level of *local search*.

Local fine-grained search: The local search proceeds in two phases: first, it extracts a candidate program from the XFTA; then, it attempts to repair the candidate.

Since each node in the XFTA represents a (possibly) exponentially large set of programs, we use a greedy algorithm to select a program from this set rather than attempting to search over it. Starting at an accepting state of the automaton, the program extractor selects the incoming edge that produces the closest image to the goal. Selecting an edge determines the root operator of the candidate program and the program sets from which to select arguments to that operator. Extraction proceeds recursively, always minimizing the distance between the overall candidate program and the goal.

When a candidate program has been extracted, we attempt to *repair* it by applying syntactic rewrites. The sequence of rewrites is chosen using a form of tabu search [Glover and Laguna 1998] and is guided by the distance from the candidate program to the goal. At each step of the repair process, we consider the set of programs that can be obtained by applying a single rewrite rule to

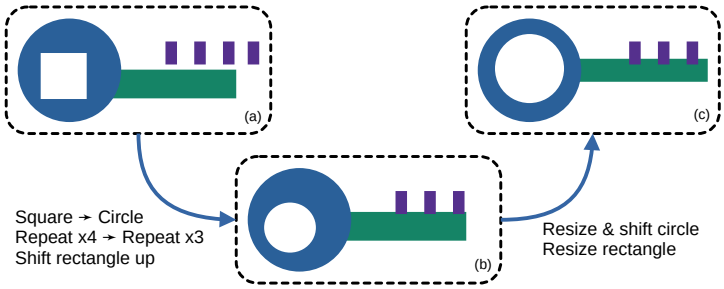


Fig. 2. A sketch of the local search process for the key example.

the current program and choose the one that is closest to the goal. This process continues until the desired program is found or until a maximum number of rewrites have been applied.

Figure 2 gives a high-level view of this repair process. Starting from a candidate program whose output is similar to the input image, the repair process applies rewrites such as changing squares to circles or incrementing/decrementing numeric parameters. Because each rewrite gets us closer to the target image, the local search can often quickly converge to a program that produces exactly the target image. For instance, using our local search algorithm, we can obtain the program that produces the image from Figure 2c starting from the program for generating the image in Figure 2a.

3 METRIC PROGRAM SYNTHESIS ALGORITHM

In this section, we describe our proposed metric program synthesis algorithm. Given a domain-specific language L and a set of of input-output examples of the form $\{(I_1, O_1), \ldots, (I_n, O_n)\}$, the goal of our method is to synthesize a program P in language L such that $\forall i \in [1, n].\ [\![P]\!](I_i) = O_i$, meaning that evaluating P on I_i yields O_i for every input-output example.

The rest of this section is organized as follows: First, because our method builds on bottom-up synthesis using finite tree automata, we start with some preliminary information on FTAs in Section 3.1. Then, in Section 3.2, we give an overview of our top-level synthesis algorithm, followed by discussions of its three sub-procedures in Sections 3.3-3.5.

3.1 Background on Synthesis using FTAs

Our synthesis algorithm builds on prior work on synthesis using finite tree automata (FTA). At a high level, an FTA is a generalization of a DFA from words to trees. In particular, just as a DFA accepts words, an FTA recognizes trees. More formally, FTAs are defined as follows:

Definition 3.1. **(FTA)** A bottom-up finite tree automaton (FTA) over alphabet Σ is a tuple $\mathcal{A} = (Q, Q_f, \Delta)$ where Q is the set of states, $Q_f \subseteq Q$ are the final states, and Δ is a set of transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow q$ where $q, q_1, \ldots, q_n \in Q$ and $\ell \in \Sigma$.

Intuitively, FTAs are useful in synthesis because they can compactly encode a set of programs, represented in terms of their abstract syntax tree [Miltner et al. 2022; Wang et al. 2017b, 2018]. In particular, when used in the context of synthesis, states of the FTA correspond to values (e.g., integers), and the alphabet corresponds to the set of DSL operators (e.g., $+, \times$). Final states are marked based on the specification, and transitions model the semantics of the underlying DSL. For instance, in a language with a negation operator $\neg$, transitions $\neg(0) \rightarrow 1$ and $\neg(1) \rightarrow 0$ express the semantics of negation.

We can view terms over an alphabet Σ as trees of the form $T = (n, V, E)$ where n is the root node, V is a set of labeled vertices, and E is the set of edges. A term T is said to be accepted by an FTA if T to can be rewritten to some state $q \in Q_f$ using transitions Δ. Finally, the language of a tree automaton $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$ and consists of the set of all terms accepted by $\mathcal{A}$.

Given a specification φ, the idea behind FTA-based synthesis is to construct an FTA whose language is the set of all programs satisfying φ. In particular, FTAs can be used to solve the programming-by-example as follows: For each input-output example (I, O), the idea is to start with a state representing I and construct new states and transitions by applying the DSL operators. For example, given FTA states representing integers 1 and 2 and a + operator in the DSL, we generate a new FTA state representing 3 using the transition $+(1, 2) \rightarrow 3$. This process of adding new states and transitions to the FTA continues until either there are no more states to be added or a pre-defined bound on the number of states or transitions has been reached. The final state of the FTA corresponds to the output example O, and the standard intersection operator is used to generate an FTA whose language includes programs that are consistent with *all* input-output examples.

As this discussion makes clear, such an FTA-based approach can compactly represent the version space if there are many DSL programs that share the *same* input-output behavior (because such programs lead to the same FTA state). However, FTA-based synthesis may not scale in application domains that do not have this property. Prior work on FTA-based synthesis has tried to tackle this problem using abstract interpretation and abstraction refinement [Wang et al. 2018]: in that setting, FTA states correspond to *abstract* rather than *concrete* values, and transitions are constructed using the *abstract*, rather than the concrete, semantics of the DSL. Of course, since such an *abstract FTA* over-approximates the set of programs consistent with the specification, one needs to perform abstraction refinement to iteratively rule out spurious programs from the language of the FTA. While this so-called Syngar approach has proven to be effective in some domains like tensor manipulations [Wang et al. 2018], such abstractions are not universally easy to construct. For example, we have found the inverse CSG domain to *not* be particularly friendly for such an abstract interpretation approach. Our metric-based synthesis algorithm is an attempt to solve this problem in a different way using distance metrics rather than abstract domains.

3.2 Overview of Synthesis Algorithm

As mentioned earlier, the key idea behind metric-based synthesis is to relax the *observational equivalence* criterion into *observational similarity* by using a distance metric. More formally, we define observational similarity as follows:

Definition 3.2. **(Similarity)** Two values v and v' are *similar*, denoted $v \simeq_\epsilon v'$ if they are within ϵ of each other, according to a distance metric δ:

$$v \simeq_\epsilon v' \Leftrightarrow \delta(v, v') \leq \epsilon.$$

The main idea behind our synthesis algorithm is to construct an approximate version space by clustering together values that are similar. Just as the Syngar idea of [Wang et al. 2018] groups together values based on an *abstract* notion of observational equivalence, our method groups together values based on this notion of *similarity* and constructs a so-called *approximate FTA (XFTA)* representing programs that produce values close to the desired output.

Our top-level metric program synthesis approach is presented in Algorithm 1 and is parameterized over a (1) distance metric δ, (2) radius ϵ, and (3) domain-specific language L. At a high level, this algorithm consists of three steps:

Algorithm 1 Metric synthesis algorithm.

Require: L is a language, I and O are the input and output examples respectively, c_{max} is the maximum program size to consider when constructing the XFTA, w is the beam width, δ is a distance metric between values, ϵ is the threshold for clustering.

Ensure: On success, returns a program p where $[\![p]\!] = O$. On failure, returns $\perp$.

```
1: procedure MetricSynth(L, I, O, c_max, w, δ, ε)
2:     A ← ConstructXFTA(L, I, O, c_max, w, δ, ε)
3:     for P ∈ Extract(A, I, O, q, δ) do
4:         P ← Repair(I, O, δ, P)
5:         if P ≠ ⊥ then
6:             return P
7:     return ⊥
```

(1) **XFTA construction:** First, MetricSynth constructs an FTA that represents a space of programs that produce values close to the goal (line 2 of Algorithm 1). However, because this FTA is constructed by grouping similar values together, a program excepted by this automaton does not necessarily satisfy the specification.
(2) **Program extraction:** To deal with the approximation introduced by clustering, the algorithm enters a loop in which it repeatedly extracts programs from the FTA using the call to Extract at line 3. The goal of Extract is to find a program in the language of the FTA that produces a value that is sufficiently close to the target.
(3) **Program repair:** Because the extracted program does not satisfy the input-output examples in the general case, the Repair procedure (invoked at line 4) tries to find a syntactic perturbation of P that exactly satisfies the input-output examples. As we discuss in more detail in Section 3.5, the repair procedure is based on rewrite rules and performs a form of tabu search, using the distance metric as a guiding heuristic.

In the following subsections, we discuss the ConstructXFTA, Extract, and Repair procedures in more detail.

3.3 Approximate FTA Construction

Algorithm 2 shows our technique for constructing an approximate FTA for a given set of input-output examples. At a high level, this algorithm builds programs in a bottom-up fashion, clustering together those programs that produce similar values on the same input. In order to ensure that the algorithm terminates, it only builds programs up to some fixed depth controlled by the hyper-parameter c_{max}.

In more detail, ConstructXFTA adds new automaton states and transitions (initialized to I and $\emptyset$ respectively) in each iteration of the while loop. In particular, for each (n-ary) DSL operator ℓ and existing states $q_1, \ldots, q_n$, it obtains a new *frontier* of candidate transitions $\Delta_{frontier}$ by evaluating $\ell(q_1, \ldots, q_n)$. The construction of this frontier corresponds to the *expansion phase* mentioned in Section 2.

In general, the expansion phase can result in a very large number of new states, making XFTA construction prohibitively expensive. Thus, in the next *clustering phase* (line 5 of Algorithm 2), the algorithm groups similar states introduced by expansion into a single state as shown in Algorithm 3. In this context, standard clustering algorithms like k-means are not suitable because they fix the

Algorithm 2 Algorithm for constructing an approximate FTA.

Require: Σ is a set of operators, all other parameters are the same as in Algorithm 1. k is a hyper-parameter that determines the number of states that the automaton should accept.

Ensure: Returns an XFTA.

1: **procedure** ConstructXFTA($\Sigma, I, O, c_{max}, w, \delta, \epsilon$)
2: $Q \leftarrow I, \Delta \leftarrow \emptyset$
3: **for** $1 \leq c \leq c_{max}$ **do**
4: $\Delta_{frontier} \leftarrow \left\{ \ell(q_1, \ldots, q_n) \rightarrow q \;\middle|\; \ell \in \Sigma, \quad \{q_1, \ldots, q_n\} \subseteq Q, \quad [\![\ell(q_1, \ldots, q_n)]\!] = q \right\}$
5: $(Q_c, \Delta_c) \leftarrow$ Cluster($\Delta_{frontier}, \delta, \epsilon$)
6: $Q' \leftarrow$ TopK($Q_c, \delta(O), w$)
7: $Q \leftarrow Q \cup Q'$
8: $\Delta \leftarrow \Delta \cup \{(\ell(q_1, \ldots, q_n) \rightarrow q) \mid q \in Q', (\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta_c\}$
9: $Q_f \leftarrow$ TopK($Q, \delta(O), k$)
10: **return** (Q, Q_f, Δ)

Algorithm 3 Greedy algorithm for clustering states.

Require: Δ is a set of FTA transitions, all other parameters are the same as in Algorithm 1.

Ensure: Returns a set of FTA transitions.

1: **procedure** Cluster(Δ, δ, ϵ)
2: $Q' \leftarrow \emptyset$ ▷ New (clustered) states
3: $\Delta' \leftarrow \emptyset$ ▷ New (clustered) transitions
4: **for** $(\ell(q_1, \ldots, q_n) \rightarrow q) \in \Delta$ **do**
5: $close \leftarrow \{q_{center} \in Q' \mid q_{center} \simeq_\epsilon q\}$
6: **if** $close = \emptyset$ **then**
7: $Q' \leftarrow Q' \cup \{q\}$
8: $\Delta' \leftarrow \Delta' \cup \{\ell(q_1, \ldots, q_n) \rightarrow q\}$
9: **else**
10: $\Delta_{new} \leftarrow \{\ell(q_1, \ldots, q_n) \rightarrow q' \mid q' \in close\}$
11: $\Delta' \leftarrow \Delta' \cup \Delta_{new}$
12: **return** (Q', Δ')

number of clusters but allow the radius of each cluster to be arbitrarily large. In contrast, we would like to minimize the number of clusters while ensuring that the radius of each cluster is bounded. Hence, we use the Cluster procedure from Algorithm 3 to generate a set of clusters where each state is within some ϵ distance from the center of a cluster. To do so, Algorithm 3 iterates over the new states q in the frontier and starts a new cluster for q if none of the previous frontier states are within ϵ of q (lines 6–8 in Algorithm 3). Otherwise, q is added to an existing cluster (lines 9–11 in Algorithm 3). Furthermore, for each new transition $\ell(q_1, \ldots, q_n) \rightarrow q$ of the frontier, clustering produces new transitions of the form $\ell(q_1, \ldots, q_n) \rightarrow q_c$ where q_c is the center of a cluster that q

Algorithm 4 Algorithm for extracting programs from an XFTA.

Require: $\mathcal{A}$ is an XFTA; all other parameters are the same as in Algorithm 1.

Ensure: Yields program terms that are accepted by $\mathcal{A}$.

1: **procedure** Extract($\mathcal{A}, I, O, \delta$)
2: $Q_f \leftarrow$ FinalStates($\mathcal{A}$)
3: Sort Q_f by $\delta(O)$ increasing.
4: **for** $q_f \in Q_f$ **do**
5: $\Delta \leftarrow$ Transitions($\mathcal{A}$)
6: $\Delta_{root} \leftarrow \{(\ell(q_1, \ldots, q_n) \to q_f) \mid (\ell(q_1, \ldots, q_n) \to q_f) \in \Delta\}$
7: **yield** ExtractTerm($\Delta_{root}, I, \delta(O)$)

Require: Δ is a set of FTA transitions, δ is a distance metric.

Ensure: Returns a program term.

8: **procedure** ExtractTerm(Δ, I, δ)
9: Let $(\ell(q_1, \ldots, q_n) \to q) \in \Delta$ be a transition where q minimizes $\delta(q)$
10: **for** $1 \le i \le n$ **do** ▷ Extract a program for each argument to ℓ.
11: $\delta_i \leftarrow \lambda q.\ \delta([\![\ell(q'_1, \ldots, q'_{i-1}, q, q_{i+1}, \ldots, q_n)]\!])$
12: $p_i \leftarrow$ ExtractTerm(Δ, I, δ_i)
13: $q'_i \leftarrow [\![p_i]\!](I)$
14: **return** $\ell(p_1, \ldots, p_n)$

belongs to. Hence, clustering produces a new set of states Q_c and a new set of transitions Δ_c to add to the automaton, as shown in line 5 of Algorithm 2.

The final component of XFTA construction is the *ranking phase*, which corresponds to lines 6–8 of Algorithm 2. Even after clustering, the automaton might end up with a prohibitively large number of new states, so ConstructXFTA only keeps the top w clusters in terms of their distance to the goal. Thus, in each iteration, Algorithm 2 only ends up adding w new states to the automaton, similar to beam search.

3.4 Extracting Programs from XFTA

We now turn our attention to the Extract procedure for picking a program that is accepted by our approximate FTA. Recall that programs accepted by the XFTA are not necessarily consistent with the input-output examples due to clustering. Furthermore, two programs P, P' that are accepted by the XFTA need not be equally close to the goal state; for example, $[\![P]\!](I)$ might be much closer to O than $[\![P']\!](I)$ with respect to the distance metric δ. Ideally, we would like to find the best program that is accepted by the FTA (in terms of its proximity to the goal); however, this can be prohibitively expensive, as the automaton (potentially) represents an exponential space of programs. Thus, rather than finding the best program accepted by the automaton, our Extract procedure uses a greedy approach to yield a sequence of "good enough" programs in a computationally tractable way.

The high-level idea behind Extract is to recursively construct a program starting from the specified final state q_f via the call to the recursive procedure ExtractTerm. At every step, the algorithm picks a transition $\ell(q_1, \ldots, q_n) \to q$ whose output minimizes the distance from the goal

Algorithm 5 Algorithm for repairing a program.

Require: I, O are the input output examples, δ is a distance metric, P is a program. There are also two hyperparameters: n is the maximum number of rewrites to perform, and R is a set of rewriting rules.

Ensure: Returns a program P' such that $[\![P']\!](I) = O$ or returns $\perp$.

1: **procedure** Repair(I, O, δ, P)
2: $S \leftarrow \emptyset$
3: **while** $i < n$ **do**
4: $neighbors \leftarrow \{P' \mid P \rightarrow_r P', r \in R\} - S$
5: $P \leftarrow \arg\min_{p \in neighbors} \delta(O, [\![p]\!](i))$
6: **if** $[\![P]\!](I) = O$ **then**
7: **return** P
8: $S \leftarrow S \cup P$
9: **return** $\perp$

and then recursively constructs the arguments $p_1, \ldots, p_n$ of ℓ. Note that this algorithm is greedy in the sense that it tries to find a single operator that minimizes the distance from the goal rather than a sequence of operators (i.e., the whole program). Hence, among the programs accepted by $\mathcal{A}$, there is no guarantee that Extract will return the globally optimum one.

3.5 Distance-Guided Program Repair

The final part of our synthesis algorithm (Repair) takes the program that was extracted from the XFTA and attempts to repair it by applying syntactic rewrite rules. In particular, given a program P that is close to the goal, Repair tries to find a program P' that is (1) syntactically close to P and (2) correct with respect to the input-output examples (i.e., $[\![P]\!](I) = O$).

Our Repair procedure is parameterized by a set of rewrite rules R of the form $t \rightarrow s$. We say that a program P can be rewritten into P' if there is a rule $r = (t \rightarrow s) \in R$ and a substitution σ such that $P = \sigma t$ and $P' = \sigma s$. We denote the application of rewrite rule r to P as $P \rightarrow_r P'$.

The Repair procedure is presented in Algorithm 5 and applies goal-directed rewriting to the candidate program, using the distance function δ to guide the search. In particular, it starts with the input program P and iteratively applies a rewrite rule until either a correct program is found or a bound n on the number of rewrite rules is reached. In each iteration of the loop (lines 3–8), it first generates a set of new candidate programs (called *neighbors*) by applying a rewrite rule to P and (greedily) picks the program P' that minimizes the distance $\delta(O, [\![P']\!](I))$. In the next iteration, the new program P' is used as the seed for applying rewrite rules.

Note that our Repair procedure utilizes a (bounded) set S to avoid getting stuck in local minima, as is done in tabu search [Glover and Laguna 1998]. In particular, S contains the most recently explored k programs, and, when applying a rewrite rule, the Repair procedure avoids generating any program in set S.

4 INSTANTIATING METRIC SYNTHESIS IN APPLICATION DOMAINS

The MetricSynth algorithm presented so far is DSL-agnostic, so it can be instantiated in different application domains. In this section, we discuss how we instantiate the metric synthesis algorithm in the (1) inverse CSG, (2) regular expression synthesis, and (3) tower-building domains.

$$E \;:=\; \texttt{Circle}(x, y, r) \mid \texttt{Rect}(x_1, y_1, x_2, y_2) \mid E \cup E \mid E - E \mid \texttt{Repeat}(E, x, y, c)$$

Fig. 3. The syntax of CSG programs.

$$\begin{aligned}
M &= \{(u, v) \mid 0 \le u < x_{max}, 0 \le v < y_{max}\} \\
\textsc{Eval}(\texttt{Circle}(x, y, r)) &= \left\{(u, v) \mid \sqrt{(x-u)^2 + (y-v)^2} < r \mid (u, v) \in M\right\} \\
\textsc{Eval}(\texttt{Rect}(x_1, y_1, x_2, y_2)) &= \{(u, v) \mid x_1 \le u \wedge y_1 \le v \wedge u \le x_2 \wedge v \le y_2 \mid (u, v) \in M\} \\
\textsc{Eval}(e \cup e') &= \{(u, v) \mid \textsc{Eval}(e)[u, v] \vee \textsc{Eval}(e')[u, v] \mid (u, v) \in M\} \\
\textsc{Eval}(e - e') &= \{(u, v) \mid \textsc{Eval}(e)[u, v] \wedge \neg\textsc{Eval}(e')[u, v] \mid (u, v) \in M\} \\
\textsc{Eval}(\texttt{Repeat}(e, x, y, c)) &= \left\{(u, v) \mid \bigvee_{0 \le i < c} \textsc{Eval}(e)[u + ix, v + iy] \mid (u, v) \in M\right\}
\end{aligned}$$

Fig. 4. The semantics of CSG programs, given as an evaluation function.

4.1 Instantiation for inverse CSG

The *inverse CSG problem* aims to "de-compile" a complex geometric shape into a set of geometric operations that were used to construct it. We now describe how to instantiate our metric program synthesis framework to solve the inverse CSG problem.

4.1.1 Domain-Specific Language. Figure 3 shows the syntax of the domain-specific language that we use for the inverse CSG problem. This DSL includes two primitive shapes, namely circles and rectangles. Circles are represented by a center coordinate and a radius. Rectangles are axis-aligned and are represented by the coordinates of the lower left and upper right corners. These primitive shapes can be combined using union, difference, and repeat operators. In particular, $\texttt{Repeat}(E, x, y, c)$ takes an image E, a translation vector $v = (x, y)$, and a count c, and it produces the union of E repeated c times, translated by v. For example, we have:

$$\texttt{Repeat}(\texttt{Circle}(v, r), v', 2) = \texttt{Circle}(v, r) \cup \texttt{Circle}(v + v', r)$$

`Repeat` allows programs with repeating patterns to be expressed compactly, which also makes these programs easier to synthesize.

Figure 4 presents the semantics of our CSG DSL using an Eval procedure, which takes as input a program and produces a bitmap image. Specifically, we represent a bitmap image as a mapping from each pixel (u, v) to a Boolean indicating whether that pixel is filled or not, or equivalently as the set of all pixels that evaluate to true. Given a program P in our DSL, we use the notation $[\![P]\!]$ to denote the bitmap image produced by P.

4.1.2 Synthesis Problem. Given a bitmap image B, represented as a mapping from pixels (x, y) to Booleans, inverse CSG aims to synthesize a program P such that:

$$\forall x, y \in \mathit{Domain}(B).\ B(x, y) = [\![P]\!](x, y)$$

Viewed in this light, note that inverse CSG is exactly a programming-by-example (PBE) problem: We can think of the bitmap image as a set of I/O examples where each input example is a pixel (x, y) and the output example is a Boolean. However, a key difference from standard PBE is that

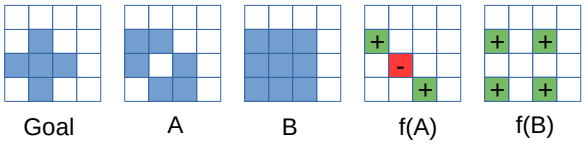


Fig. 5. Illustration of the distance transformation f_O. Note that the Jaccard distance is $\delta_J(A, B) = \frac{1}{3}$, whereas $\delta_{Goal}(A, B) = \frac{3}{5}$. This shows how the distance between images that are very close to the goal is magnified.

the number of examples we need to deal with is quite large: for instance, in our evaluation, we use 32×32 bitmap images, so the number of I/O examples is 1024.

4.1.3 Distance function. A key component of our metric synthesis algorithm is the specific distance metric used. For the inverse CSG domain, we use a distance metric δ that is a slight modification of the standard Jaccard distance that takes into account the goal value. Specifically, we define the distance metric as follows:

$$\delta_O(q, q') = 1 - \frac{|f_O(q) \cap f_O(q')|}{|f_O(q) \cup f_O(q')|} \quad \text{where } f_O(q) = \{(x, y, b) \mid (x, y) : b \in q, b \neq O[x, y]\}.$$

Intuitively, this distance considers only the pixels of the images q and q' that *differ* from the goal image O. This is desirable because images that are very close to the goal are treated as further from each other than images that are far from the goal. Overall, this metric has the effect of making our algorithm more sensitive to differences in images that are close to the goal. We illustrate this effect in Figure 5.

Theorem 4.1. *δ_O is a metric on the set of images.*

Proof. Let O be some goal value. Let $\delta_J(q, q') = 1 - \frac{|q \cap q'|}{|q \cup q'|}$ be the Jaccard distance, which is a metric on finite sets. We have $\delta_O(q, q') = \delta_J(f_O(q), f_O(q'))$ where f_O (defined above) is a function $Image \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{B}$. Because an injective function f from any set S to a metric space (M, δ) gives a metric $\delta(f(x), f(x'))$ on S, we can show that δ_O is a metric by showing that f is injective. To see why this is the case, note that we can define the inverse of f_O as follows:

$$f_O^{-1}(s) = \{(x, y) : b' \mid (x, y) : b \in O, b' = \neg b \text{ if } (x, y, \neg b) \in s \text{ else } b\}.$$

Intuitively, where f_O returns the differences between q and O, f_O^{-1} applies those differences to O to obtain q. □

4.1.4 Rewrite rules for repair. Recall that our Repair procedure is parameterized over a set of rewriting rules R. At a high level, the rewrite rules we use for the inverse CSG problem modify integers (within bounds) and transform squares into circles (and vice-versa). In more detail, our implementation utilizes the following rewrite rules:

$$x \to x + 1 \qquad \text{if } x \text{ is an integer and } x < x_{max}$$

$$x \to x - 1 \qquad \text{if } x \text{ is an integer and } x > 0$$

$$\texttt{Circle}(x, y, r) \to \texttt{Rect}(x - r, y - r, x + r, y + r)$$

$$\texttt{Rect}(x_1, y_1, x_2, y_2) \to \texttt{Circle}(x_1 + r, y_1 + r, r) \qquad \text{where } r = \frac{x_2 - x_1}{2} \text{ if } x_2 - x_1 = y_2 - y_1$$

4.2 Instantiation for Regular Expressions

Our second application domain of metric program synthesis is generating regular expressions from a set of positive and negative examples.

$$E := \emptyset \mid C \mid \texttt{Concat}(E, E) \mid \texttt{Repeat}(E, x) \mid \texttt{RepeatRange}(E, x_1, x_2) \mid \texttt{RepeatAtLeast}(E, x) \mid \texttt{Optional}(E) \mid E \wedge E \mid E \vee E \mid \neg E$$

$$\begin{aligned}
\textsc{Eval}(C, s) &= \{(i, i+1) \mid 0 \le i \le |s|, s[i] \in C\} \\
\textsc{Eval}(\emptyset, s) &= \{(i, i) \mid 0 \le i \le |s|\} \\
\textsc{Eval}(\texttt{Concat}(E, E'), s) &= \{(i, k) \mid (i, j) \in \textsc{Eval}(E, s), (j', k) \in \textsc{Eval}(E', s), j = j'\} \\
\textsc{Eval}(\texttt{Repeat}(E, 1), s) &= \textsc{Eval}(E, s) \\
\textsc{Eval}(\texttt{Repeat}(E, x), s) &= \textsc{Eval}(\texttt{Concat}(E, \texttt{Repeat}(E, x-1)), s) \\
\textsc{Eval}(\texttt{RepeatRange}(E, x_1, x_2), s) &= \bigcup_{x_1 \le i \le x_2} \textsc{Eval}(\texttt{Repeat}(E, i), s) \\
\textsc{Eval}(\texttt{RepeatAtLeast}(E, x), s) &= \textsc{Eval}(\texttt{RepeatRange}(E, x, |s|), s) \\
\textsc{Eval}(\texttt{Optional}(E), s) &= \textsc{Eval}(E \vee \emptyset, s) \\
\textsc{Eval}(E \wedge E', s) &= \textsc{Eval}(E, s) \cap \textsc{Eval}(E', s) \\
\textsc{Eval}(E \vee E', s) &= \textsc{Eval}(E, s) \cup \textsc{Eval}(E', s) \\
\textsc{Eval}(\neg E, s) &= \{(i, j) \mid 0 \le i \le j \le |s|, (i, j) \notin \textsc{Eval}(E, s)\}
\end{aligned}$$

Fig. 6. Syntax and semantics of the regular expression DSL.

4.2.1 Domain-Specific Language. Our regular expression language (Figure 6) is similar to the one used in prior work [Chen et al. 2020]. This DSL includes character classes, concatenation, repetition, optional matches, conjunction, disjunction, and negation. Character classes match a single character from a set; we use a set of single-character classes that includes all of the printable characters as well as multi-character classes for numbers, capital and lowercase letters, symbols, and vowels. $\texttt{Concat}(E, E')$ matches E followed by E'. For example, $\texttt{Concat}(\texttt{<num>}, \texttt{<a>})$ matches the string "0a." $\texttt{Repeat}(E, x)$ matches x repetitions of E. As in the CSG DSL, we include the repetition operators to make repetitive programs more tractable to synthesize.

Programs in the regular expression DSL evaluate to a set of match locations in a string s, as shown in Figure 6. Since the semantics of regular expressions are fairly standard, we do not explain the semantics in detail.

4.2.2 Synthesis Problem. As in prior work [Chen et al. 2020; Lee et al. 2016], we consider the problem of synthesizing regular expressions from a given set of positive and negative examples. Let S^+ be a set of positive examples (strings), and let S^- be a set of negative examples. Then, the synthesis problem is to generate a regular expression E such that:

$$\forall s \in S^+.\ (0, |s|) \in \textsc{Eval}(E, s) \text{ and } \forall s \in S^-.\ (0, |s|) \notin \textsc{Eval}(E, s)$$

However, prior work has shown that synthesizing the *intended* regular expression just from positive and negative examples can be challenging if one only has access to few examples [Chen et al. 2020]. For this reason, [Chen et al. 2020] has advocated using *sketches* obtained from natural language descriptions. Following that work, we consider a modified version of this problem where the regular expression needs to be a completion of the provided sketch in addition to satisfying the positive and negative examples. Specifically, our problem formulation additionally requires a sketch which is given in the form of a program with holes, where the holes may additionally contain constraints on the terms that must be used to fill the hole, as in [Chen et al. 2020].

4.2.3 Distance Function. We use a simple distance function that simply counts the number of examples that the regular expression matches. In particular, we consider a program to match a positive example s if $(0, |s|)$ is in the match set m. A program matches a negative example if $(0, |s|)$ is not in m.

4.2.4 Rewrite rules. Some of the constructs used in the regex DSL take integers as arguments. As in the inverse CSG instantiation, we include rewrite rules that transform integers. We also include a rule that transforms `Rewrite` to `RewriteRange`, which allows programs that use `Rewrite` to be generalized to a range of repeat counts.

$$x \longrightarrow x + 1 \qquad \text{if } x \text{ is an integer and } x < x_{max}$$
$$x \longrightarrow x - 1 \qquad \text{if } x \text{ is an integer and } x > 0$$
$$\texttt{Repeat}(E, x) \longrightarrow \texttt{RepeatRange}(E, x, x)$$

4.3 Instantiation for Tower Building

Our third application domain is the tower building task from prior work [Ellis et al. 2021; Nye et al. 2021] that is inspired by AI planning tasks. Given a set of blocks and target "tower" (i.e. configuration of these blocks), this task aims to generate the desired tower by (programmatically) controlling a robot arm.

4.3.1 Domain-specific Language. Figure 7 shows the syntax of the tower building DSL. Programs in this language control a robot arm which can move left and right along a horizontal track and can drop horizontal or vertical blocks. The state of the program includes the x-position of the arm and the list of dropped blocks. In more detail, the DSL includes operators for dropping blocks, moving the robot arm, sequencing, and looping. It also includes the `Embed`(E) operator, which executes E and then restores the position of the arm. This operator gives the DSL a limited way to return to a previous state without needing to specify the movement needed to reset the arm.

The semantics of the DSL are shown in Figure 7. `DropH` and `DropV` both add a new (horizontal or vertical) block to the tower. The block will be placed on the highest block that is below the arm. The move operators both update the position of the arm; `MoveBefore` moves the arm and then executes E, while `MoveAfter` moves the arm after evaluating E. `Embed` gives the language a degree of modularity. It executes E and then resets the arm to wherever it was before. Note that these semantics correspond to an idealized physics model where blocks fall in a perfectly straight line until they land on top of another block.

4.3.2 Synthesis Problem. The input to the synthesizer is a set of blocks B where each block is represented as a tuple (b, x, y). Here, $b \in \{\texttt{H}, \texttt{V}\}$ is denotes the type of the block (either horizontal 1x3 or vertical 3x1) and (x, y) is the block's position. B must be a valid tower, which means that the blocks must not overlap. Given this input, the synthesis problem is to produce a program P such that $\textsc{Eval}(P, s_0) = B$, where $s_0 = (0, [\,])$ is the initial state with no blocks placed and the hand at $x = 0$.

4.3.3 Distance Function. The distance function for the tower building domain is based on the insight that translating a tower along the x-axis is straightforward, so we want our distance function to be *translation-invariant*. Hence, two programs that build the same tower in nearby places should be deemed similar. Based on this intuition, we use the Jaccard distance to compare two towers, and we normalize them before we compare them so that their leftmost block is at $x = 0$. Specifically,

$$E := \texttt{DropH} \mid \texttt{DropV} \mid \texttt{MoveBefore}(E, x) \mid \texttt{MoveAfter}(E, x) \mid E; E \mid \texttt{Loop}(x, E) \mid \texttt{Embed}(E)$$

$$\begin{aligned}
\textsc{Eval}(\texttt{DropH}, (h, bs)) &= (h, (h, \max_{h \le x < h+3} \textsc{Top}(x, bs), \texttt{H}) : bs) \\
\textsc{Eval}(\texttt{DropV}, (h, bs)) &= (h, (h, \textsc{Top}(h, bs), \texttt{V}) : bs) \\
\textsc{Eval}(\texttt{MoveBefore}(E, x), (h, bs)) &= \textsc{Eval}(E, (h + x, bs)) \\
\textsc{Eval}(\texttt{MoveAfter}(E, x), s) &= (h + x, bs) \text{ where } (h, bs) = \textsc{Eval}(E, s) \\
\textsc{Eval}(E; E', s) &= \textsc{Eval}(E', \textsc{Eval}(E, s)) \\
\textsc{Eval}(\texttt{Loop}(1, E), s) &= \textsc{Eval}(E, s) \\
\textsc{Eval}(\texttt{Loop}(x, E), s) &= \textsc{Eval}(\texttt{Loop}(x - 1, E), \textsc{Eval}(E, s)) \\
\textsc{Eval}(\texttt{Embed}(E), (h, bs)) &= (h, bs') \text{ where } (h', bs') = \textsc{Eval}(E, s) \\
\textsc{Top}(bs, x) &= \max\{y \mid (b, x', y) \in bs, x = x'\}
\end{aligned}$$

Fig. 7. Syntax and semantics of the tower building DSL.

the distance metric between states is defined as follows:

$$\delta(s, s') = \delta_J(z(s), z(s')),$$

where δ_J is the Jaccard distance and z is the normalizing function defined as:

$$z((h, bs)) = (h, \{(b, x - x_{min}, y) \mid (b, x, y) \in bs\}.$$

4.3.4 Rewrite Rules. Similar to the previous domains, some of the constructs in the tower building DSL take integer valued arguments. Hence, we use rewrite rules that allow incrementing and decrementing these integers, as in the inverse CSG and regular expression domains. These rules suffice to change loop iteration counts and to modify the movement operators.

5 IMPLEMENTATION

We have implemented our proposed synthesis technique in a new tool called SyMetric written in OCaml. In what follows, we describe some optimizations over the basic synthesis algorithm presented in Section 3.

Randomization. Our implementation of the MetricSynth algorithm is randomized and calls the ExtractTerm and Repair procedure multiple times. In particular, SyMetric samples multiple programs accepted by the XFTA by calling ExtractTerm multiple times. Furthermore, for each extracted program, SyMetric attempts to repair it multiple times if the Repair procedure fails. Hence, in order to sample different programs and different repairs, we introduce randomness in both the extraction and repair procedures. Specifically, we modify the ExtractTerm procedure to consider a randomly selected subset of the automaton transitions when generating a program accepted by the FTA. Similarly, we modify Repair to consider a random subset of the rewrites when generating candidate programs to select from. Such randomization helps compensate for the greedy nature of these algorithms by introducing the possibility of taking a locally suboptimal step that turns out to be globally optimal.

Incremental clustering. Since the clustering technique is a significant cost of approximate FTA construction, our implementation performs a few modifications. In particular, instead of computing all clusters and then sorting them, it first sorts the transitions and uses the first k clusters that it finds. Furthermore, because the number of transitions in $\Delta_{frontier}$ can be very large in the ConstructXFTA

$$\mathcal{U} = \{v[x,y] = b \mid v \in \mathsf{Image}, x, y \in \mathbb{N}, b \in \mathbb{B}\} \cup \{v = c \mid c \in \mathsf{Type}(v)\} \cup \{true, false\}$$

$$[\![f(v_1 = c_1, \ldots, v_n = c_n)]\!]^\sharp = (v = [\![f(c_1, \ldots, c_n)]\!])$$

$$[\![(v[x,y] = true) \cup p]\!]^\sharp = [\![p \cup (v[x,y] = true)]\!]^\sharp = (v[x,y] = true)$$

$$[\![(v[x,y] = false) - p]\!]^\sharp = [\![p - (v[x,y] = true)]\!]^\sharp = (v[x,y] = false)$$

Fig. 8. Abstract semantics for the inverse CSG instantiation of AFTA. Predicates come from the universe $\mathcal{U}$.

algorithm, our implementation incrementally collects the top states in batches. This involves evaluating the frontier multiple times, rather than storing it, but we find that, in practice, we need only a small prefix of the sorted frontier. Finally, our implementation of the Cluster procedure uses an M-tree data structure [Ciaccia et al. 1997] to facilitate efficient insertion and range queries.

Optimizations for inverse CSG. Our instantiation of SyMetric in the inverse CSG domain incorporates three low-level optimizations. First, it represents images as packed bitvectors to reduce their size. Second, our evaluation function for the CSG DSL is memoized. Third, our implementation uses optimized (and, where possible, vectorized) C implementations for bitvector operations, distance functions, and for CSG operators such as `Repeat`.

Optimizations for regular expressions. In our implementation of the regular expression domain we view the match sets as graphs where the positions in the string are the nodes and the matches are the edges. We represent these graphs as adjacency matrices using an efficient packed Boolean representation. This representation is particularly effective for synthesizing regular expressions from examples, because the example strings tend to be short, which mitigates the $O(n^2)$ memory cost of the matrix. We also note that `Repeat`(E, n) matches (i, j) if E matches $(i, k_1), (k_1, k_2), \ldots, (k_{n-1}, j)$. That is, `Repeat`$(E, n)$ matches (i, j) if there is a walk of length n in the match graph for E from i to j. The walks of length n are given by the nth power of the adjacency matrix A_E^n. Therefore, we can build efficient implementations for the `Repeat*` operators and for `Concat` using an efficient Boolean matrix multiplication primitive.

6 EVALUATION

In this section, we describe a series of experiments to empirically evaluate our approach. In particular, our experiments are designed to evaluate the following key research questions:

(1) **RQ1:** How does SyMetric compare against other domain-agnostic and domain-specific synthesis tools in the inverse CSG, regular expression, and tower building domains?
(2) **RQ2:** How much does similarity-based clustering help with search space reduction?
(3) **RQ3:** What is the relative importance of the various ideas comprising our approach?
(4) **RQ4:** How do different components of our synthesis algorithm contribute to running time?

6.1 Inverse CSG

To perform our evaluation in the inverse CSG domain, we compare SyMetric against the following three baselines:

- Sketch-Du: Since prior work on inverse CSG is based on the Sketch synthesis system, we implement a baseline that uses an encoding similar to the one used in InverseCSG [Du et al. 2018]. However, since that work focuses on 3D shapes, we modify the encoding to work with our DSL for 2D geometry and also add support for repetition (see Section 7). In addition, unlike the

$$\mathcal{U} = \{longestmatch(m) \leq k \mid m \in \mathsf{Match}, k \in \mathbb{N}\} \cup \{k \leq firstmatchpos(m) \mid m \in \mathsf{Match}, k \in \mathbb{N}\}$$
$$\cup \{v = c \mid c \in \mathsf{Type}(v)\} \cup \{true, false\}$$

$$[\![f(v_1 = c_1, \ldots, v_n = c_n)]\!]^\sharp = (v = [\![f(c_1, \ldots, c_n)]\!])$$
$$[\![\texttt{Optional}(true)]\!]^\sharp = (0 \leq firstmatchpos(m))$$
$$[\![(longestmatch(m) \leq k) \wedge (longestmatch(m) \leq k')]\!]^\sharp = (longestmatch(m) \leq \min(k, k'))$$
$$[\![(k \leq firstmatchpos(m)) \wedge (k' \leq firstmatchpos(m))]\!]^\sharp = (\max(k, k') \leq firstmatchpos(m))$$
$$[\![\texttt{RepeatRange}((longestmatch(m) \leq k), true, n)]\!]^\sharp = (longestmatch(m) \leq k \times n)$$

Fig. 9. Selected abstract semantics for the regular expression instantiation of AFTA. Predicates are drawn from the universe $\mathcal{U}$.

encoding in [Du et al. 2018], our sketches do not contain hints about the location of primitive shapes, so the synthesizer needs to discover the numeric parameters of these primitive shapes.

- FTA: This baseline performs bottom-up synthesis (with equivalence reduction) using FTAs [Wang et al. 2017b]. Like the implementation of SyMetric, this baseline is also implemented in OCaml. Note that this baseline only adds FTA states and transitions until a final state is reached, as our goal is to find *one*, rather than all, programs consistent with the specification.
- AFTA: This baseline performs bottom up synthesis with abstract FTAs (AFTAs) [Wang et al. 2018]. In particular, this method uses abstract values as states of the FTA, constructs FTA transitions using the abstract semantics, and performs abstraction refinement to deal with spurious programs extracted from the AFTA. Our implementation of this baseline is also in OCaml and uses the same matrix abstract domain from [Wang et al. 2018] since bitmap images can be viewed as matrices. However, since the underlying DSLs are different, we implement the abstract transformers shown in Figure 8 for our domain-specific language.

Benchmarks. Since prior work on Inverse CSG [Du et al. 2018; Jones et al. 2021] mostly targets 3D benchmarks, we construct our own benchmark suite for 2D Inverse CSG. Specifically, we consider a total of 40 benchmarks, where 25 correspond to outputs of randomly generated programs (modulo some non-triviality constraints) and 15 are hand-written benchmarks of visual interest.

Setup. We run these experiments on a machine with two AMD EPYC 7302 processors (64 threads total) and 256GB of RAM. We use a time limit of 1 hour and a memory limit of 4GB (so that we can run many benchmarks at the same time). For the hyper-parameters for SyMetric, we use $\epsilon = 0.2$, $w = 200$, and the maximum number of rewriting steps is $n = 500$.

Summary of results. The results of this evaluation are shown in Figure 10(a). SyMetric is able to solve 78% of these benchmarks, but the baselines fail on all of them except at most 3. In what follows, we discuss why the baselines perform poorly and the failure cases for SyMetric.

FTA results. The FTA baseline fails on all but the smallest of the handwritten benchmarks. When it fails, it is universally because it runs out of memory. For this domain, few programs produce exactly the same output image, so equivalence reduction is not sufficient to reduce memory consumption.

AFTA results. We found that the AFTA baseline is unable to solve *any* benchmarks when we include the `Repeat` operator in the DSL, as repetition causes fundamental difficulties with the abstraction refinement phase of the Syngar technique [Wang et al. 2018]. However, the AFTA

baseline can solve 3 of the 40 benchmarks if we omit the `Repeat` operator from the DSL. For many of the remaining benchmarks, the target programs become quite complex without the use of the `Repeat` operator, so the AFTA approach fails either because it reaches the time limit or fails to find a program with the AST depth limit of 40.

Sketch-Du results. Our third baseline Sketch-Du, which is an adaptation of InverseCSG [Du et al. 2018] to our setting, is able to solve three of the handwritten benchmarks but fails on the remaining ones. For the 37 benchmarks it cannot solve, it runs out of memory 75% of the time and runs out of time 18% of the time. We attempted to provide this baseline with parameters that would minimize its memory use and maximize its chances of successfully completing the benchmarks. To that end, we used Sketch's specialized integer solver to reduce memory overhead; we controlled the amount of unrolling in the sketch based on the size of the benchmark program, and we used Sketch's example file feature, which reduces the time required to find counterexamples during the CEGIS loop. However, even with all of these optimizations, we found that the large number of examples in the inverse CSG domain (one per pixel, so 1024 total) causes Sketch to perform many iterations of the CEGIS loop.

SyMetric results. SyMetric performs significantly better than the other baselines, solving all but 9 of the 40 benchmarks. Based on our manual inspection of these 9 benchmarks, we found two dominant failure modes. One of them is that the beam width w may be too narrow in some cases, causing critical subprograms to be dropped from the search space. This effect is more pronounced when the benchmark relies on subprograms that are far from the goal O according to δ. One pattern that we noticed among the failure cases is that they include subprograms where one shape is subtracted from another, producing a complex shape that is distant from its inputs and also distant from the final image. The second way that a benchmark can fail is that there is a program close to the solution that is contained in the XFTA, but the Extract and Repair procedures are unable to find it. While extracting all programs accepted by the XFTA could mitigate the problem, the overhead of doing so is prohibitively expensive in many cases.

6.2 Regular Expression Synthesis

Our second application domain is regular expression synthesis. Given a sketch and a set of positive and negative examples, the task is to find a regular expression that conforms to the given sketch and matches all positive examples, while matching none of the negative ones.

For this application domain, we perform an empirical comparison against the following baselines:

- Regel: This baseline is a state-of-the-art regular expression synthesis tool [Chen et al. 2020]. It performs *top-down* (rather than bottom-up) synthesis and uses a number of SMT-based pruning strategies to reduce the search space.
- FTA: This baseline performs bottom-up enumeration with equivalence reduction using FTAs for our regular expression DSL.
- AFTA: As in the Inverse CSG domain, this baseline is an abstraction-based version of bottom-up enumeration with equivalence reduction [Wang et al. 2018]. We use a predicate abstraction that tracks the length of the longest match and the beginning of the first match (see Figure 9). These predicates effectively allow the tool to avoid examining programs which do not match the whole string or that do not match from the beginning.

We note that the FTA and AFTA baselines use the same interpreter as SyMetric and utilize the optimization discussed in Section 5, which uses matrix multiplication to efficiently evaluate terms. However, Regel uses a different regular expression matching algorithm based on the Brics automaton library, which is not as efficient as our optimized implementation for this DSL.

Benchmarks. For this experiment we use the Stack Overflow dataset taken from [Chen et al. 2020]. This benchmark was collected from user questions and it contains 122 distinct tasks, each of which consists of a natural language description and a set of input-output examples. For each task, Chen et al. automatically generates a set of *sketches* (i.e. partial programs) that capture additional constraints about the target regular expression that are present in the natural language description. This gives us a total of 2173 total task/sketch pairs to use in our evaluation. However, we note that some of these synthesis problems may not be solvable since the generated sketches could be wrong.

Experimental setup. These experiments are run on a machine with two Intel Xeon 8375C processors (with a total of 128 threads) and 256GB of RAM. We use a time limit of 5 minutes and a memory limit of 4GB. We use $\epsilon = 0.3$, $w = 200$, and $n = 100$ for the regular expression domain.

Results summary. The results of this experiment are summarized in Figures 10(c) and 10(d), where the latter figure "zooms in" on the harder benchmarks. Among the four tools, SyMetric achieves the best performance, solving 76% of the regex benchmarks, compared to 61% of Regel. The FTA baseline significantly outperforms the abstraction-based approach, solving 66% (for FTA) compared to 13% (for AFTA). One caveat about these results is that, while the comparison between SyMetric, FTA, and AFTA is apples-to-apples, Regel uses a different (and less efficient) interpreter for their DSL. [2] Another caveat is that, while our predicate abstraction does not yield good results, it *may* be possible to construct abstractions that perform better in the regex domain. Nevertheless, we believe these results substantiate our claim that (a) SyMetric is competitive with state-of-the-art tools for the regex domain, and (b) metric program synthesis yields improvements over basic observational equivalence reduction.

6.3 Solving Tower Building Tasks

As our third application domain, we consider tower building tasks that are inspired by planning in AI and that were used for evaluating program synthesis tools in prior work [Ellis et al. 2021; Nye et al. 2021]. As explained in Section 4, the goal of this task is to generate a program that constructs a given configuration of blocks.

For this domain, we compare SyMetric against two baselines:

- Neural: Our first baseline is a state-of-the-art neural synthesizer [Nye et al. 2021] that combines top-down program synthesis with a neural network that predicts the possible outputs of a partial program. Since this baseline is trained on a set of representative tower building tasks, it can utilize tower motifs encountered during training to solve new tasks.
- FTA: As in the previous two domains, our second baseline performs bottom-up enumerative search with equivalence reduction using FTAs.

For this domain, we do not compare SyMetric against the abstraction-based FTA approach, as the combination of loops and mutable state make this domain difficult grounds for applying prior work [Wang et al. 2018]. In fact, we believe that applying abstraction refinement techniques to this domain/DSL is an open research problem in its own right.

Benchmarks. Our tower building benchmarks are drawn from [Nye et al. 2021], which are constructed by systematically composing tower building programs together (e.g., taking a program that builds a $w \times h$ bridge and building two side-by-side, varying the size and spacing). The original benchmark suite contains 40 tasks, but we found that two of the tasks are duplicates and four are not expressible in the DSL described in [Nye et al. 2021], as they require loops with non-constant

[2] We believe it is due to this implementation difference in the interpreter that FTA slightly outperforms Regel. The pruning heuristics used in Regel allow it to scale without needing a custom interpreter.

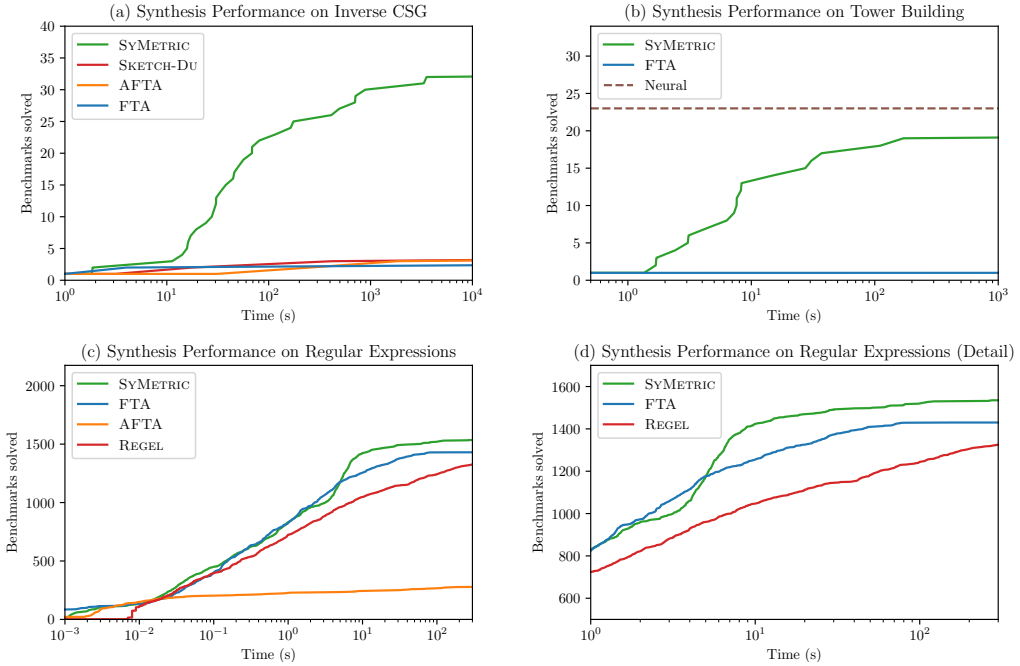


Fig. 10. Synthesis performance on the inverse CSG, tower building, and regular expression domains.

iteration counts. We remove these six tasks, resulting in a total of 34 tower building benchmarks used in our evaluation.

Experimental setup. These experiments are performed on a machine with two Intel Xeon 8375C processors (128 threads total) and 256GB of RAM. We use a time limit of 10 minutes and a memory limit of 4GB. For hyper-parameters, we use $\epsilon = 0.4$, $w = 100$, and $n = 100$.

Results summary. The results for this domain are summarized in Figure 10(b). Note that we do not show running time for Neural, as it is not reported in [Nye et al. 2021] and we do not have access to their model. Overall, we find that SyMetric approaches the performance of the neural approach and that it performs significantly better than FTA. In particular, FTA performs poorly in this domain because the target programs tend to be fairly large and not many of the enumerated programs result in the same block configuration, so equivalence reduction is not as effective in this context. On the other hand, metric program synthesis can deal with the large size of the search space by exploiting observational similarity and by using ranking to select promising subprograms. Finally, we note that, while Neural is able to solve a few more benchmarks compared to SyMetric, it can do so by using motifs learned from training data as building blocks. In contrast, SyMetric can achieve similar performance without requiring access to large amounts of training data.

6.4 Detailed Evaluation of Metric Synthesis

To gain more insights about the effectiveness of metric program synthesis and answer our research questions RQ2-RQ4, we perform a more detailed evaluation of SyMetric in the inverse CSG domain. In particular, we explore distance-based clustering in more depth and present the results of relevant ablation studies.

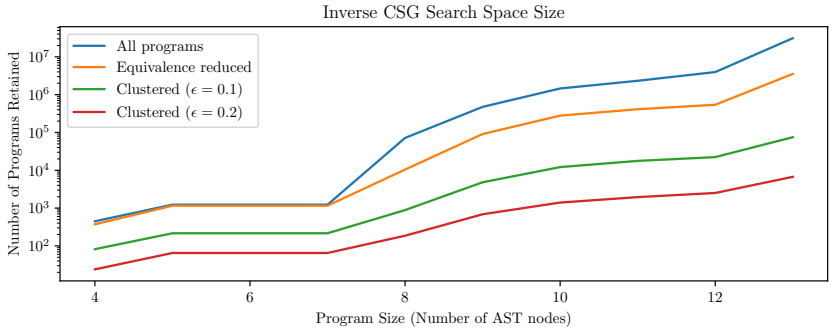


Fig. 11. Size of the CSG program space for programs with up to n AST nodes.

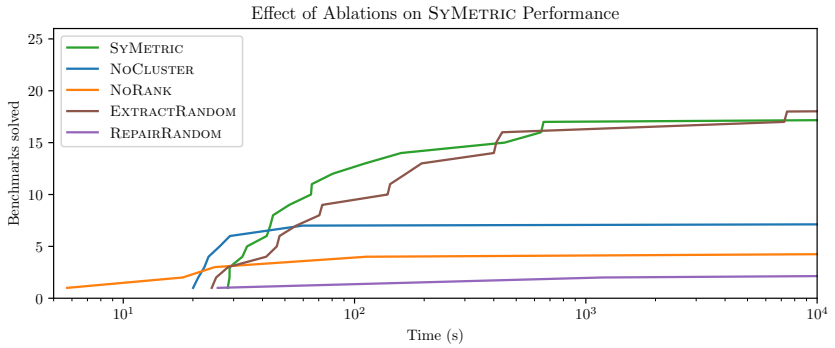


Fig. 12. Effect of our ablations on SyMetric for inverse CSG.

6.4.1 Effectiveness of Clustering. To evaluate the benefits of similarity-based clustering, we perform the following experiment on the inverse CSG domain:

(1) First, we generate a set of programs with n non-terminals.
(2) Then, for each value of n, we apply equivalence reduction to remove equivalent programs.
(3) Finally, we group the set of distinct programs using Algorithm 3.

Figure 11 shows the number of clusters for two different values of ϵ, namely $\epsilon = 0.1$ and $\epsilon = 0.2$. We only consider programs with up to $n = 13$ AST nodes, because enumerating all programs for larger values of n is not computationally feasible.

As is evident from Figure 11, equivalence reduction reduces the number of programs that must be retained by approximately one order of magnitude, and similarity-based clustering reduces the search space even more dramatically. In particular, for $\epsilon = 0.1$, there is an approximately $10\times$ reduction compared to just grouping based on equivalence and an even larger reduction for the coarser ϵ value of 0.2. Hence, this experiment shows that there are many programs that are observationally similar but not equivalent, which partly explains why metric program synthesis is effective in this domain.

6.4.2 Ablation Studies. In this section, we describe a set of ablation studies to evaluate the relative importance of different algorithms used in our approach. We consider the following ablations:

Benchmark	ConstructXFTA		Extract		Repair	
	Median	Max	Median	Max	Median	Max
Generated	35.5	40.9	0.1	12.1	21.2	664.0
Hand-written	15.4	65.1	0.0	21.2	6.0	3481.6
All	29.5	65.1	0.1	21.2	10.2	3481.6

Fig. 13. Runtime breakdown (in seconds) for different sub-procedures of SyMetric on inverse CSG.

- NoCluster: This is a variant of SyMetric that does not perform clustering during FTA construction. However, it still performs repair after extracting a program from the FTA.
- NoRank: This is a variant of SyMetric that does not use distance-based ranking during XFTA construction. Instead, it picks w randomly chosen (clustered) states to add to the automaton in each iteration rather than ranking them according to the distance metric and picking the top w.
- ExtractRandom: This variant does not use our proposed distance-based program extraction technique. Instead, it randomly picks programs that are accepted by the automaton. (However, note that the final states of the XFTA are still determined using the distance metric.)
- RepairRandom: This variant does not use our distance-based program repair technique. Instead, after applying a rewrite rule during the Repair procedure, it randomly picks one of the programs rather than using the distance metric to pick the one closest to the goal.

The results of these ablation studies are presented in Figure 12 (again, for the inverse CSG domain). We find that, for this domain, the most important component of our algorithm is the distance-guided Repair procedure, followed by ranking during XFTA construction, and the use of clustering. The distance-guided Extract procedure seems to have less impact, but there is still a noticeable increase in synthesis runtime for shorter running benchmarks if we randomly choose a program instead of using the distance metric for extraction.

Disabling ranking and clustering both yield noticeable performance improvements for some of the easier benchmarks. This is because both ranking and clustering are relatively expensive parts of the synthesis algorithm (see Section 6.4.3.). While ranking is cheap on its own, if it is disabled, we only need to enumerate new states until we can build w clusters. In contrast, when ranking is enabled, we need to look at the entire frontier at least once so we can sort it. Similarly, disabling clustering is a significant time saver for easier benchmarks. However, both clustering and ranking have a huge positive impact for the harder benchmarks. In fact, without them, the number of benchmarks solved within the 1 hour time limit drops very significantly.

6.4.3 Detailed Evaluation of Running Time. In this section, we explore the impact of different sub-procedures on running time, again on the inverse CSG domain. Specifically, Figure 13 compares the running times of XFTA construction (the ConstructXFTA procedure), program extraction (Extract), and program repair (procedure Repair). Since our synthesis algorithm calls Extract and Repair multiple times, we show the aggregate running time of these procedures across all calls.

In most cases, the running time of the ConstructXFTA procedure dominates total synthesis time. In contrast, program extraction from the XFTA using our greedy approach is quite fast, taking a median of about 0.1 seconds. Finally, while the average running time of Repair is around 10 seconds, it varies widely depending on how many calls to Repair are made and how many rewrite rules we need to apply to find the correct program.

Figure 14 provides a more detailed look at XFTA construction. Recall that ConstructXFTA consists of three phases, namely expansion, clustering, and ranking. In Figure 14, we show the running time of each of these phases during XFTA construction. As we can see in this table, the

Benchmark	Expansion		Clustering		Ranking	
	Median	Max	Median	Max	Median	Max
Generated	21.9	25.4	3.9	8.1	0.0	0.1
Hand-written	7.9	19.9	4.3	36.5	0.0	0.2
All	18.5	25.4	4.0	36.5	0.0	0.2

Fig. 14. Runtime breakdown (in seconds) for different sub-procedures of ConstructXFTA on inverse CSG.

expansion phase dominates XFTA construction time. This is not surprising because expansion requires evaluating DSL programs to construct new states. Ranking is extremely fast and barely takes any time. Clustering takes around four seconds on average, although there are some outlier benchmarks where clustering ends up being more expensive than the expansion phase.

7 RELATED WORK

Our work is related to and builds on several different lines of work that we discuss here.

Bottom-up Synthesis. Our work builds heavily on bottom-up synthesis with equivalence reduction, an idea that was introduced concurrently by Albarghouthi et al. and Udupa et al.. Later work by Wang et al. explored another variation of this idea in the context of version space learning and showed how to use Finite Tree Automata (FTA) to compactly represent the space of programs consistent with a given specification. Our work was particularly inspired by Blaze [Wang et al. 2018] which demonstrated the use of abstraction refinement to speed up bottom up search. Abstractions provide a mechanism for grouping closely related solutions, allowing large sets of solutions to be ruled out by evaluating only one abstract solution. In this regard, Blaze can be viewed as performing equivalence reduction over abstract domains. In this work, we explore another relaxation of observational equivalence based on the notion of observational similarity rather than abstract equivalence. We believe that our metric program synthesis idea is complementary to the abstraction refinement approach and can work well in settings for which abstract domains are hard to design or where abstractions do not effectively reduce the search space.

Quantitative Synthesis. A key part of our algorithm is the use of distance metrics to group similar programs and to rank them. In this regard, our method bears similarities to prior work exploring the use of quantitative goals in program synthesis, both in the reactive synthesis space [Cerný and Henzinger 2011] and in functional synthesis (e.g. [Schkufza et al. 2013, 2014]). However, in many of these cases, the quantitative objectives are used to deal with noisy or probabilistic specifications [Handa and Rinard 2020; Raychev et al. 2016], whereas we use them to perform search more effectively.

Neural-guided Synthesis. More recently, there has been significant interest in neural-guided program synthesis, where a neural network is trained to guide the search for a program that satisfies a specification. In early incarnations of this idea, the neural network was used simply to select components that were likely to be used by the program [Balog et al. 2017], but starting with the work of Devlin et al., the neural network has been heavily involved in directing the search. Especially relevant to our work is the work on execution guided synthesis [Chen et al. 2019; Ellis et al. 2019], which uses the state of the partially constructed program to determine the most promising next step for the synthesizer. The work by Ellis et al. in particular inspired the ranking phase of our current algorithm. That work uses a learned value function which is trained to evaluate the output of many different candidate programs to determine which ones to keep as

part of the beam. A common limitation of all the neural guided synthesis approaches is that they require significant work ahead of time to collect a dataset and train the algorithm on that dataset. In contrast, our approach relies on a domain-specific distance function, and we find that simple distance functions often work fairly well. There is a potential for future work that seeks to apply the insights of this work in a deep learning context.

Diversity. One effect of the grouping performed by our algorithm during search is to increase the diversity of the programs in the beam, allowing it to cover more distinct programs and increasing the chance that a program close to the correct program will be included in the beam. There has been some prior work on the use of diversity measures as part of a search procedure. For example, the genetic programming community has long recognized that diversity in a population of programs is crucial to avoid converging to low-quality local minima and has explored a number of diversity measures to maintain diversity of the population [Burke et al. 2004]. Similarly, in the context of beam search for NLP, there has been recent recognition that the top-K elements of a beam may be too close to each other and fail to capture multiple modes in the underlying distribution, which has led to the proposal of *Diverse Beam Search* to force proposals in a beam to be sufficiently different from each other [Vijayakumar et al. 2018]. Our work shares some intuitions with some of these prior works, but to our knowledge, this paper is the first to apply the idea of grouping based on a similarity function in the context of FTA-based synthesis.

Program Synthesis for Inverse CSG. There has been a lot of interest in the CAD community in using program synthesis techniques to reverse engineer CAD problems. Two early works in this space are InverseCSG [Du et al. 2018] and the work of Nandi et al.. Both aim to reverse engineer 3D CSG programs from meshes, but both rely on specialized algorithms to do a significant part of the work. For example, Nandi et al. rely on a set of domain specific oracles that examine the mesh and generate proposed decompositions (i.e. splitting the mesh into a union of two simpler shapes). These oracles are powerful but highly specialized to the CSG domain. Similarly, InverseCSG relies on a specialized preprocessing phase to identify all constituent primitive shapes and their parameters, so the synthesizer only has to discover the Boolean structure of the shape. In contrast, our synthesis method can solve for all primitives and their parameters without relying on a domain-specific preprocessor. Additionally, InverseCSG relies on a segmentation algorithm to break a large shape into small fragments that are then assembled into the final shape. In contrast, we aim to solve the entire inverse CSG problem as a single synthesis task. Finally, our program space is richer than that of InverseCSG because it includes a looping construct in addition to the primitives and Boolean operations. Our experimental results show that we can do with a single algorithm what prior work required an entire pipeline of complex and very specialized algorithms.

In addition to the program synthesis oriented work on inverse CSG, there is a line of work that focuses on the use of neural networks to recover structured models from unstructured input [Sharma et al. 2018; Tian et al. 2019]. Their performance is generally very good but dependent on training data (e.g., Shape2Prog [Tian et al. 2019]) is specialized to just furniture shapes). One of the goals of our algorithm is to only require limited domain knowledge, which are captured through the distance function and repair rules. However, we believe that our approach can complement the neural methods, either by using a learned distance metric or by using a network to predict a likely space of programs, as in [Lee et al. 2018].

There is also prior work that processes CSG programs, either to extract common structure [Jones et al. 2021] or to capture regularity [Nandi et al. 2020]. ShapeMOD [Jones et al. 2021] is a technique for extracting common macros from a library of CSG programs. These macros form a domain specific language for a particular class of CSG programs (e.g., furniture) and help provide more physically plausible results that are biased towards patterns common in the target domain. Our

method does not rely on a set of training programs but could utilize macros (like the ones learned by ShapeMOD) if they were available. Nandi et al. show that it is possible to post-process the output of an InverseCSG-like system in order to extract loops, which produces programs that are more general and easier to modify. However, such an approach first requires synthesizing the loop free program, which can get quite large for models with a lot of repetition.

Synthesis of Regular Expressions. There is a long history of work on synthesizing regular expressions from examples. Recent work includes Regel [Chen et al. 2020] and AlphaRegex [Lee et al. 2016], both of which are top-down synthesizers that employ upper and lower bounds for pruning. Regel additionally leverages natural language descriptions to improve generalizability. In particular, Regel parses the natural language descriptions into so-called *hierarchical sketches* and uses top-down enumerative search combined with SMT-based pruning to find a sketch completion that satisfies the examples. In our evaluation, we utilize the sketches generated by Regel but solve those sketches using metric program synthesis instead of top-down SMT-guided search.

Repair of regular expressions, which is related to our local search, has also attracted significant attention from the research community [Pan et al. 2019; Rebele et al. 2018]. Because our method starts the local search process with programs that are already fairly close to the goal, our method can use simpler rewrite-based techniques for repair.

Synthesis of Tower Building Programs. The tower building domain first appears in [Ellis et al. 2021] and is used as a benchmark in [Nye et al. 2021]. The appeal of this domain comes from its use of loops and mutable state, as well as its connection to classical AI planning tasks. Prior work has focused on the application of neural guided synthesis to this domain, whereas we show that a non-neural approach can also perform quite well.

8 CONCLUSION

We presented a new synthesis technique, called *metric program synthesis*, that performs search space reduction using a distance metric. The key idea behind our technique is to cluster similar states during bottom-up enumeration and then perform program repair once a program that is "close enough" to the goal is found. In more detail, our approach constructs a so-called *approximate finite tree automaton* that represents a set of programs that "approximately" satisfy the specification. Our method then repeatedly extracts programs from this set and uses distance-guided rewriting to find a repair that exactly satisfies the given input-output examples.

Our proposed synthesis algorithm is intended for domains that have two key properties: (1) the DSL contains many programs that are semantically similar, and (2) programs that are semantically similar also tend to be syntactically close. With this intuition in mind, we have instantiated our synthesis framework in three different domains (inverse CSG, regular expression synthesis, and tower building) by defining suitable distance metrics. Our evaluation shows that our tool, SyMetric, outperforms prior domain-agnostic FTA-based techniques in these domains. Furthermore, we compare our approach against domain-specific synthesizers and show that the performance of SyMetric is competitive with these tools despite not utilizing any training data or domain-specific synthesis algorithms.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=ByldLrqlx

E.K. Burke, S. Gustafson, and G. Kendall. 2004. Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation* 8, 1 (2004), 47–62. https://doi.org/10.1109/TEVC.2003.819263

Pavol Cerný and Thomas A. Henzinger. 2011. From boolean to quantitative synthesis. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 149–154. https://doi.org/10.1145/2038642.2038666

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. 328–343. https://doi.org/10.1145/3453483.3454047

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. 487–502. https://doi.org/10.1145/3385412.3385988

Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gfOiAqYm

Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, Vol. 97. 426–435.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 990–998. http://proceedings.mlr.press/v70/devlin17a.html

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. Inversecsg: Automatic Conversion of 3d Models To CSG Trees. *ACM Trans. Graph.* 37, 6 (2018), 213:1–213:16. https://doi.org/10.1145/3272127.3275006

Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 9165–9174. https://proceedings.neurips.cc/paper/2019/hash/50d2d2262762648589b1943078712aa6-Abstract.html

Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. 835–850. https://doi.org/10.1145/3453483.3454080

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 420–435. https://doi.org/10.1145/3192366.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 229–239. https://doi.org/10.1145/2737924.2737977

Fred Glover and Manuel Laguna. 1998. Tabu Search. In *Handbook of Combinatorial Optimization: Volume1–3*, Ding-Zhu Du and Panos M. Pardalos (Eds.). Springer US, 2093–2229. https://doi.org/10.1007/978-1-4613-0303-9_33

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. https://doi.org/10.1145/1926385.1926423

Shivam Handa and Martin C. Rinard. 2020. Inductive program synthesis over noisy data. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA,*

November 8-13, 2020. 87–98. https://doi.org/10.1145/3368089.3409732

R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2021. Shapemod: Macro Operation Discovery for 3d Shape Programs. *ACM Trans. Graph.* 40, 4 (2021), 153:1–153:16. https://doi.org/10.1145/3450626.3459821

Tessa A. Lau, Steven A. Wolfman, Pedro M. Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1-2 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 70–80. https://doi.org/10.1145/2993236.2993244

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 436–449. https://doi.org/10.1145/3192366.3192410

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498682

Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.* 2, ICFP (2018), 99:1–99:31. https://doi.org/10.1145/3236794

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 31–44. https://doi.org/10.1145/3385412.3386012

Maxwell I. Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2021. Representing Partial Programs with Blended Abstract Semantics. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. https://openreview.net/forum?id=mCtadqIxOJ

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair of Regular Expressions. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 139:1–139:29. https://doi.org/10.1145/3360565

Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 761–774. https://doi.org/10.1145/2837614.2837671

Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding Missing Words to Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II*. 67–79. https://doi.org/10.1007/978-3-319-93037-4_6

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. 305–316. https://doi.org/10.1145/2451116.2451150

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 53–64. https://doi.org/10.1145/2594291.2594302

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 5515–5523. https://doi.org/10.1109/CVPR.2018.00578

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=rylNH20qFQ

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

Ashwin K. Vijayakumar, Michael Cogswell, Ramprasaath R. Selvaraju, Qing Sun, Stefan Lee, David J. Crandall, and Dhruv Batra. 2018. Diverse Beam Search for Improved Description of Complex Scenes. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February*

2-7, 2018. 7371–7379. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17329

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017a. 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, Barcelona, Spain, June 18-23, 2017*. 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Synthesis of data completion scripts using finite tree automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis Using Abstraction Refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2021. Fusion 360 gallery: a dataset and environment for programmatic CAD construction from human design sequences. *ACM Trans. Graph.* 40, 4 (2021), 54:1–54:24. https://doi.org/10.1145/3450626.3459818