Metric Program Synthesis for Inverse CSG

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

In this paper, we present a new method for solving the *inverse constructive solid geometry (CSG)* problem, which aims to generate a programmatic representation of a scene from its unstructured representation (e.g., a raster image). Our method, called *metric program synthesis*, hinges on the observation that many programs in the inverse CSG domain produce *similar* (but not identical) images. Based on this observation, we propose clustering programs based on a distance metric and constructing a version space that compactly represents "approximately correct" programs. 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 image.

We have implemented our proposed metric program synthesis technique in a tool called SyMetric and evaluate it in the inverse CSG domain. Our evaluation on 40 benchmarks shows that SyMetric can effectively solve inverse CSG problems that are beyond scope for existing synthesis techniques. In particular, SyMetric solves 78% of these benchmarks whereas the closest competitor can only solve 8%. Our evaluation also shows the benefits of similarity-based clustering and other salient features of our approach through a series of ablation studies.

ACM Reference Format:
John Feser, Isil Dillig, and Armando Solar-Lezama. 2018. Metric Program Synthesis for Inverse CSG. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 21 pages.

1 INTRODUCTION

In recent years, there has been increasing interest in the *CAD reconstruction 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]. Recent work by Tao Du et al. showed that CAD reconstruction could be framed as a program synthesis problem and focused in particular on the case where the target program is represented in a formalism known as constructive solid geometry (CSG) [Du et al. 2018]. CSG describes the construction of a complex shape as a sequence of set operations (union, intersection, difference) on simple primitive shapes. As argued by Du, this *inverse CSG* problem is important because it can enable a number of useful applications, from the creation of new parts for legacy hardware to interoperability among competing CAD systems.

While the work of Du et al. showed the potential for program synthesis to attack the inverse CSG problem, it also showed how difficult the resulting synthesis problems can be. There are a number of factors that make the problems challenging, including (a) the large number of numerical parameters generally present even in a simple model, (b) the size of the required models for any non-trivial shape and (c) the large number of points needed to describe the rendered shape. That paper was able to succeed in its task only by relying on specialized geometric algorithms to identify all the primitives and all the constants in the constructed shape, and by aggressively sub-dividing the shape into smaller more tractable shapes. This simplifies the resulting synthesis problems, but at the expense of generating lower quality solutions, because the synthesizer can no longer capture global patterns such as repeating structures that span the entire shape. There is therefore a need

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/

for more powerful synthesis algorithms that can solve the inverse CSG problem end-to-end and that can capture patterns such as repetition within a shape.

In this paper, we attack this challenge through a novel synthesis technique, dubbed *metric program synthesis*, that can effectively solve a 2D version of the inverse CSG problem end-to-end. Our solution is based on the following key observation:

> *Many programs in the inverse CSG domain produce images that are very similar, though not identical.*

This observation has two key implications:

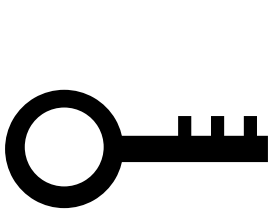
- Synthesis techniques that perform search space reduction using *observational equivalence* [Udupa et al. 2013] do not work well in our setting. This is because such techniques rely on many programs in the DSL to share the *exact same* input-output behavior but this does not hold in our target domain.
- Because many programs produce *similar* images, there is room for reducing this search space by exploiting this similarity.

Our method exploits these observations by relaxing the *observational equivalence* criterion to *observational similarity*. Just as existing synthesis techniques group programs with identical input-output behaviors into an equivalence class [Udupa et al. 2013], our proposed method 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 programs are observationally equivalent but many have *similar* input-output behaviors, such distance-based clustering can result in a much more substantial search space reduction compared to existing techniques.

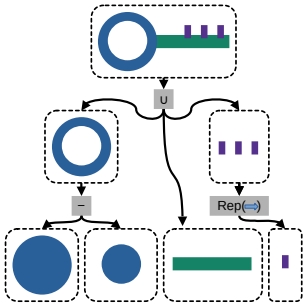
To exploit observational similarity, our proposed method proceeds in two phases: First, it performs bottom-up enumerative synthesis to build a *version space* [Lau et al. 2003] that compactly represents all programs up to some fixed AST depth. During this bottom-up enumeration, it clusters programs based on a provided distance metric and can therefore ensure that the generated version space is compact. However, due to the use of distance-based clustering, the generated version space is approximate in that it contains many programs that are incorrect, albeit perhaps *close to* being correct. In order 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 always possible to find *syntactically* similar representations of *semantically* similar programs in the inverse CSG domain, this combination of approximate version space construction with local program repair allows solving problems that cannot be handled by existing techniques.

We refer to the idea described above as *metric program synthesis* and propose a concrete synthesis algorithm that realizes this idea. In particular, our method constructs a version space in the form of an *approximate finite tree automaton*, then extracts a "good enough" program P from this automaton, and finally repairs P by applying rewrite rules in a distance-guided way. While our target application domain in this paper is inverse CSG, our proposed metric program synthesis algorithm is DSL-agnostic and can be applied to other domains where (1) there are many programs that are semantically similar but not identical, and (2) small semantic differences between programs can be bridged with small syntactic changes.

We have implemented the proposed approach in a tool called SyMetric and evaluate it on 40 inverse CSG benchmarks. Some of these benchmarks are written manually to produce visually interesting images, and others are randomly generated programs in our domain-specific language.



(a) The input image.



(b) The solution, illustrated.

(c) Program text of the solution.

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

Fig. 1. An example CSG problem.

Our evaluation shows that our method significantly outperforms all existing baselines: While SyMetric can solve 78% of the benchmarks, its closest competitor can only solve 8%. We also perform a series of ablation studies and demonstrate that clustering and distance-guided repair are both crucial for making this approach work in practice.

To summarize, this paper makes the following key contributions:

- We introduce a new idea called *metric program synthesis* that exploits the semantic and syntactic similarity of programs in the search space.
- We propose a concrete instantiation of this metric program synthesis idea based on *approximate finite tree automata* and *distance-guided program repair*.
- We implement our technique in a new tool called SyMetric and evaluate it in the inverse CSG domain, showing that our technique significantly outperforms prior solutions.

2 OVERVIEW

In this section, we work through an example 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 several 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 and difference operators. Furthermore, there is some repeating structure in the teeth of the key that we could capture, so adding an operator to repeat a shape would reduce the program's size. Using a repeat operator also makes the drawing more customizable by allowing users to change the number of teeth with a single parameter.

Figure 1c shows a program that can generate this picture. The program composes 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.

Our goal in this work is to synthesize the program in Figure 1c given *just* the picture in Figure 1a. One standard approach is to simply search over programs in the DSL, creating programs by composing together smaller programs. As an optimization, the algorithm can keep track of only a single program for each distinct shape that it creates. This approach—known as bottom-up enumeration with equivalence reduction [Udupa et al. 2013]—is a simple program synthesis algorithm that works well in many domains.

However, 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 scene 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 input 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 4.3. The key idea behind an XFTA is to group together values that are semantically similar: in our 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. When the algorithm starts, the first expansion generates the set of primitive shapes. Later expansions compose scenes together using Boolean operators and the looping operator `Repeat` to create scenes of increasing complexity. Many of the scenes generated during the expansion phase are similar to each other.

In the grouping phase, scenes are put into clusters. Each cluster has a center and a radius ϵ, and every scene in a cluster is within ϵ of the center. Although every scene 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 scene are retained. This focuses the search on the programs that are likely to produce the goal scene. 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 goal scene. At this point, a new difficulty presents itself: If we only clustered programs that had equivalent behavior, we could simply check whether the goal scene is present in the XFTA. If it is, then the corresponding program can be extracted from the automaton and the synthesis task is complete. However, since we cluster programs that have similar but not the same behavior, we may complete XFTA construction and find that the goal scene is not present in the automaton, even though that there are several scenes that are *close* to the goal. To address this issue, our method performs a second level of *local search*.

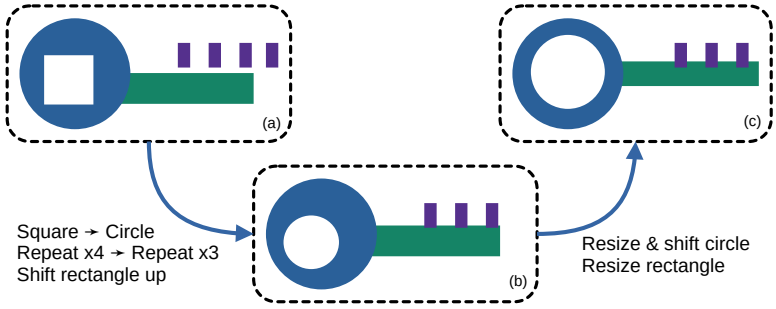


Fig. 2. A sketch of the local search process for the key example.

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

Since each node in the XFTA represents a (potentially) 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 scene 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 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 PROBLEM STATEMENT

We frame the inverse CSG problem as instance of example-based program synthesis. The input to the synthesizer is a bitmap image and the output is a program in the target CAD domain-specific language. In the remainder of this section, we present the DSL for this domain and formalize the synthesis problem.

3.1 Domain-Specific Language for Inverse CSG

Figure 3 shows the syntax of the domain-specific language in which we synthesize programs. 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,

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

Fig. 3. The syntax of CAD programs.

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

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

difference, and repeat operators. In particular, $\texttt{Repeat}(E, x, y, c)$ takes a scene 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 CAD 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.

3.2 Synthesis Problem

Given a bitmap image B, represented as a mapping from pixels (x, y) to booleans, inverse CSG is the problem of synthesizing a program P such that the following condition holds:

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

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. Then, the inverse CSG problem aims to synthesize a program P in our DSL that is consistent with all of these examples. However, a key difference from standard PBE is that 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 METRIC PROGRAM SYNTHESIS ALGORITHM

In this section, we describe our synthesis algorithm for solving the inverse CSG problem defined in the previous section. Because this synthesis technique can have applications beyond the inverse

CSG problem, we describe our algorithm in a more general way, deferring a discussion of how it is instantiated in the CAD domain to the end of this section.

The rest of this section is organized as follows: First, we provide some preliminary information in Section 4.1. Then, in Section 4.2, we give an overview of our top-level synthesis algorithm, followed by discussions of its three sub-procedures in Sections 4.3-4.5. Finally, we show how to instantiate this synthesis framework in the inverse CSG domain (Section 4.6).

4.1 Background on Synthesis using FTAs

Our synthesis algorithm builds on prior work on synthesis using finite tree automata (FTA). Hence, we first give some background on FTAs and how they are used in program synthesis. 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, we define FTAs as follows:

Definition 4.1. **(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. 2017, 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 that satisfy φ. If many programs in the DSL have the same input-output behavior, then this FTA-based approach can compactly represent all programs that satisfy the specification. Once such an FTA is built, then any program accepted by the FTA can be returned as a solution to the synthesis problem.

4.2 Overview of Synthesis Algorithm

As is evident from the previous discussion, FTA-based synthesis works well in settings where many DSL programs are observationally equivalent on inputs of interest. While the inverse CSG domain does contain many observationally equivalent programs, there are also a large number of programs whose behavior is *similar*, but not exactly the same. Motivated by this observation, our approach builds a data structure called *approximate FTA (XFTA)* where states represent *sets* of values that are within some small radius with respect to a distance metric. We formalize this notion using the following notion of *similarity* between values:

Definition 4.2. **(Similarity)** We say that 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:

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

Our synthesis technique crucially relies on this notation of similarity and is presented in Algorithm 1. The procedure MetricSynth is parameterized over a distance metric δ, a radius ϵ, and a domain-specific language L. The goal is to synthesize a program P in the given DSL that

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 ⊥
```

produces outputs O on the given inputs I (represented as a vector). The algorithm starts by calling the ConstructXFTA procedure which produces a finite tree automaton $\mathcal{A}$. Unlike prior work, the language of this FTA is *not* the set of programs that are consistent with the input-output examples; rather, $\mathcal{A}$ is constructed by grouping similar values together to reduce the search space. Hence, we cannot guarantee that a program accepted by $\mathcal{A}$ will produce the correct output.

To deal with this difficulty, our synthesis procedure performs local search via the loop in lines 3–6 of Algorithm 1. In particular, MetricSynth invokes Extract at line 3 to find a sequence of programs P. This Extract procedure performs greedy search over the automaton $\mathcal{A}$ to identify programs that are close to the goal — we discuss this program extraction procedure in more detail in Section 4.4. While P's input-output behavior is semantically similar to the target program, it is not guaranteed to produce exactly the desired output O on input I. Hence, MetricSynth invokes a procedure called Repair at line 4 to find a syntactic perturbation of P that exactly satisfies the input-output examples. As we discuss in more detail in Section 4.5, 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.

4.3 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.

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)

In this context, standard clustering algorithms like k-means are not suitable because they fix the 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 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.

4.4 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 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')

algorithm picks a transition $\ell(q_1, \ldots, q_n) \rightarrow q$ whose output minimizes the distance from the goal 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.

4.5 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.

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 $\mathcal{A}$.

procedure Extract($\mathcal{A}, I, O, \delta$)
  $Q_f \leftarrow$ FinalStates($\mathcal{A}$)
  Sort Q_f by $\delta(O)$ increasing.
  for $q_f \in Q_f$ **do**
    $\Delta \leftarrow$ Transitions($\mathcal{A}$)
    $\Delta_{root} \leftarrow \{(\ell(q_1, \ldots, q_n) \to q_f) \mid (\ell(q_1, \ldots, q_n) \to q_f) \in \Delta\}$
    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.

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

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 $\perp$.

procedure Repair(I, O, δ, P)
  $S \leftarrow \emptyset$
  while $i < n$ **do**
    $neighbors \leftarrow \{P' \mid P \to_r P', r \in R\} - S$
    $P \leftarrow \arg\min_{p \in neighbors} \delta(O, [\![p]\!](i))$
    if $[\![P]\!](I) = O$ **then**
      return P
    $S \leftarrow S \cup P$
  return $\perp$

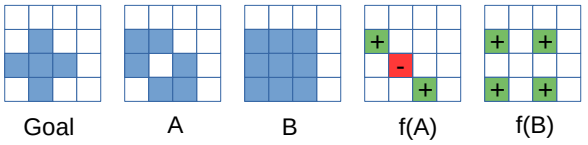


Fig. 5. Illustration of the distance transformation f_O. Note that the Jaccard distance is $\delta_J(A, B) = \frac{1}{3}$, whereas $\delta_{Goal}(A, B) = \frac{3}{5}$. This shows how the distance between scenes that are very close to the goal is magnified.

4.6 Instantiation for Inverse CSG

The MetricSynth algorithm presented so far is DSL-agnostic, but our primary application domain is inverse CSG. In this subsection, we discuss how to instantiate the metric synthesis algorithm in this application domain.

Distance function. A crucial 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 scenes q and q' that *differ* from the goal scene O. This is desirable because scenes that are very close to the goal are treated as further from each other than scenes that are far from the goal. Overall, this metric has the effect of making our algorithm more sensitive to differences in scenes that are close to the goal. We illustrate this effect in Figure 5.

Theorem 4.3. *δ_O is a metric on the set of scenes.*

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 $Scene \to \mathbb{Z} \times \mathbb{Z} \times \mathbb{B}$. 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. □

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:

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

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

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

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

Algorithm 6 Algorithm for checking that a CAD program is well-formed.

$$\begin{aligned}
\mathrm{WF}(\texttt{Circle}(x, y, r)) &= r > 0 \wedge 0 \leq x - r \wedge x + r < x_{max} \wedge 0 \leq y - r \wedge y + r < y_{max} \\
\mathrm{WF}(\texttt{Rect}(x_1, y_1, x_2, y_2)) &= x < x' \wedge y < y' \\
\mathrm{WF}(e - e') &= \mathrm{WF}(e \cup e') = \mathrm{WF}(e) \wedge \mathrm{WF}(e') \\
\mathrm{WF}(\texttt{Repeat}(e, v, c)) &= \|v\| > 0 \wedge c > 1 \wedge \mathrm{WF}(e)
\end{aligned}$$

5 IMPLEMENTATION

We have implemented our proposed synthesis technique in a new tool called SyMetric implemented in OCaml. Our implementation is DSL-agnostic, meaning that SyMetric can be applied to other DSLs beyond the CAD DSL that we focus on in our evaluation. In what follows, we describe some optimizations over the basic synthesis algorithm presented in Section 4.

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 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 CSD domain also performs a number of optimizations. In particular, the CAD language admits a number of programs that are ill-formed, so our implementation prunes these programs from the search space using a lightweight well-formedness check that is presented as rules in Algorithm 6.

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

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:

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

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

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

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

$$[\![\texttt{Repeat}(v[x,y] = true, dx, dy, c)]\!]^\sharp = \bigwedge_{i=0}^{c} v[x + c \times dx, y + c \times dy] = true$$

$$[\![\texttt{Repeat}(\bigwedge_{i=0}^{c} v[x - c \times dx, y - c \times dy] = false, dx, dy, c)]\!]^\sharp = (v[x,y] = false)$$

Fig. 6. Abstract semantics for FTA-Syngar. Predicates are drawn from the universe $\mathcal{U}$.

(1) **RQ1:** How does SyMetric compare against existing synthesis tools in the inverse CSG domain?
(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?

Benchmarks. We evaluate our algorithm on two sets of benchmarks. The first set consists of 25 randomly generated programs in our DSL. These programs were selected to contain between 25 and 35 non-terminals and an AST depth between 5 and 6. In addition, we ensure that the programs have the following *naturalness* properties:

- No subprogram may generate the empty scene.
- Every primitive shape must fit entirely on the canvas.
- No operator may produce one of its arguments as its result.

In addition to these 25 randomly generated programs, we also evaluate SyMetric on a collection of 15 hand-written benchmarks that are of visual interest. These benchmarks have at most 20 non-terminals and use the `Repeat` operator.

Setup. We run these experiments on a machine with two AMD EPYC 7302 processors (64 threads total) and 256GB of RAM. All of our experiments are single-threaded and run with a 1 hour time limit and a 4GB memory limit. SyMetric is a Las Vegas algorithm, so in our benchmarks we run it 5 times and present the *expected runtime*, which we compute as $\mathbb{E}[\text{runtime}] = \mathbb{E}[\text{number of runs until success}] \times \mathbb{E}[\text{time per run}]$. The other algorithms in our comparisons are deterministic, so we run them only once.

Hyperparameters. Our synthesis framework must be configured with a number of hyper-parameters, such as the radius ϵ and beam width w from Algorithm 1. In our evaluation, we use $\epsilon = 0.2$ and $w = 200$. We experimented (separately) with $\epsilon = 0.1$ and $w = 100$ and found that reducing ϵ or w reduced the number of benchmarks that SyMetric could solve. We also experimented with $\epsilon = 0.4$ and $w = 400$, which increased SyMetric's running time but did not allow it to solve additional benchmarks. Finally, recall that our distance-guided repair technique uses a hyper-parameter n, which controls the number of maximum rewriting steps allowed. In our evaluation, we use a value of $n = 500$.

6.1 Comparison with Existing Synthesizers

To evaluate the benefits of our proposed metric synthesis technique, we compare SyMetric against the following baselines:

- Sketch-CAD: This baseline uses the Sketch synthesis system to solve the synthesis problem. It does this by using an encoding similar to the one used by InverseCSG [Du et al. 2018], which is also based on Sketch. However, the encoding is more complex because it needs to support repetition and the discovery of all the parameters in the primitive shapes (see Section 7).
- FTA-Basic: This is an OCaml implementation of basic FTA-based synthesis that essentially performs bottom-up enumeration with observational equivalence reduction. [Wang et al. 2017]
- FTA-Syngar: This is another FTA-based synthesizer that uses abstraction refinement to improve scalability, as described in [Wang et al. 2018]. We also implement this technique in OCaml and use their abstraction for the matrix domain. Note that bitmap images are essentially matrices, so this abstraction also makes sense in our domain. However, since our DSL is different from the one in [Wang et al. 2018], we implement the abstract transformers shown in Figure 6 for our domain-specific language.

For FTA-Syngar, we found that the `Repeat` operator in our DSL causes challenges for abstraction refinement. In particular, the abstraction refinement phase of this approach needs to introduce new predicates that are sufficient to show that a candidate program p is not correct. In practice, however, the number of new predicates that may be introduced needs to be tightly bounded for scalability reasons. But, because of the semantics of the `Repeat` operator, proving that $\texttt{Repeat}(p', dx, dy, c)$ does not set a pixel (x, y) requires establishing that p' does not set any of the pixels (x, y), $(x - dx, y - dy)$, $\ldots$, $(x - c \times dx, y - c \times dy)$. This, in general, requires the introduction of c new predicates to the abstraction and therefore causes severe scalability issues. In order to deal with this problem, we also consider a variant called FTA-Syngar (NR) which is the same as FTA-Syngar except that it uses a DSL without the `Repeat` operator.

Main results. The results of our evaluation are presented in Table 1. Here, the first column indicates the tool being evaluated, and the second column corresponds to the type of benchmark (Generated, Hand-written, All). The third column shows the running time in seconds for those benchmarks that could be successfully solved. The column labeled "Success" shows the percentage of benchmarks that could be successfully solved, and the last two columns show the percentage of benchmarks that failed due to memory and time limits, respectively.[1] We now discuss the results for each tool in more detail.

SyMetric. Overall, SyMetric is able to solve around 78% of the benchmarks across the two benchmark sets, whereas the other tools fail on all but just 1 or 2 benchmarks. The median synthesis time of SyMetric across all benchmarks is around 45 seconds.

We manually inspected the benchmarks that SyMetric failed to solve to better understand the root causes for failure. Overall, 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 scene. 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

[1] For FTA-Syngar (NR), the numbers in the last three columns do not add up to 100% because some benchmarks are not synthesizable without the `Repeat` operator.

Algorithm	Benchmark	Median Runtime (s)	Success	Memory	Timeout
SyMetric	Generated	56.7	68%	0%	32%
	Hand-written	18.6	93%	0%	7%
	All	44.7	78%	0%	22%
FTA-Syngar	Generated	–	0%	0%	100%
	Hand-written	–	0%	0%	100%
	All	–	0%	0%	100%
FTA-Syngar (NR)	Generated	253.0	4%	20%	0%
	Hand-written	929.5	13%	47%	7%
	All	253.0	8%	30%	2%
FTA-Basic	Generated	–	0%	100%	0%
	Hand-written	2.5	13%	80%	7%
	All	2.5	5%	92%	2%
Sketch	Generated	–	0%	100%	0%
	Hand-written	17.7	20%	33%	47%
	All	17.7	8%	75%	18%

Table 1. Comparison of SyMetric against existing synthesis algorithms. Each benchmark can either succeed or fail due to timeout, memory exhaustion, or in the case of FTA-Syngar (NR) because the benchmark is unsatisfiable without the `Repeat` operator. Runtimes are given in seconds, and are only shown for the benchmarks that succeeded. All benchmarks are run with a 1 hour timeout and 4GB of memory.

by the XFTA could mitigate the problem, the overhead of doing so is prohibitively expensive in many cases.

FTA-Basic. FTA-Basic fails on all but the smallest of the handwritten benchmarks. When it fails, it is universally because it runs out of memory. As we discuss in the next subsection, there are already 10^6 distinct programs with 13 AST nodes, so equivalence reduction is not sufficient to reduce memory consumption. For programs with up to 35 AST nodes (which is the maximum size of our generated benchmarks), FTA-Basic would require an unreasonable amount of memory.

FTA-Syngar. Recall that we run FTA-Syngar in two modes, one with the Repeat operator in the DSL and one without. In the mode containing `Repeat`, FTA-Syngar is unable to synthesize any of the benchmark problems within the 1 hour time limit due to explosion in the number of predicates. In contrast, if we disallow `Repeat` from the DSL, then FTA-Syngar can solve 3 of the 40 benchmarks. However, because many benchmarks do require the use of the `Repeat` operator, this version of FTA-Syngar fails to find the intended program even when it terminates within the given time and memory limits.

Sketch. Sketch is able to solve three of the handwritten benchmarks, but it fails on all of the generated benchmarks. It fails by running out of memory 75% of the time and by running out of time 18% of the time. We attempted to provide Sketch with parameters that would minimize its memory use and maximize its chances of successfully completing the benchmarks. 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. We found that the large number of examples in the inverse CSG problems (one per pixel, so 1024 total) meant that Sketch needed to perform many iterations of the CEGIS loop to make progress.

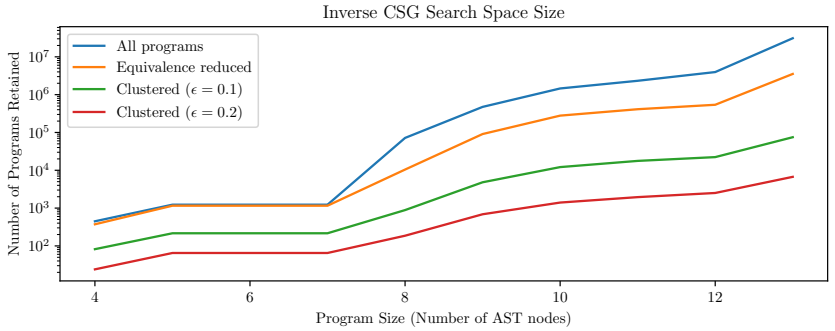


Fig. 7. Size of the CAD program space for programs with up to n AST nodes.

Additionally, the benchmark problems yield large sketches, which require much more than 4GB of memory for the solver. The arithmetic required for the `Circle` and `Repeat` operators is particularly expensive for Sketch.

6.2 Effectiveness of Clustering

To better understand why SyMetric performs much better than the other baselines in the inverse CSG domain, we perform an experiment to evaluate the benefits of similarity-based clustering. To perform this evaluation, we generate a set of programs with n non-terminals. Then, for each value of n, we apply equivalence reduction to remove equivalent programs. Finally, the set of distinct programs is grouped using Algorithm 3. We show the number of clusters for two different values of ϵ: $\epsilon = 0.1$ and $\epsilon = 0.2$ (Figure 7). We only consider programs with up to $n = 13$ AST nodes, because enumerating all programs for larger values of n is not computationally feasible.

The results of this evaluation are shown in Figure 7. As is evident from this graph, 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× 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 the inverse CSG domain is indeed full of programs that are similar, which (partly) explains why our technique performs significantly better compared to the other baselines.

6.3 Ablation Studies

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

- 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 that is close to the goal.
- NoRank: This is a variant of SyMetric that does not use distance-based ranking during XFTA construction. In other words, 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 then picking the top w ones.
- ExtractRandom: This is a variant of SyMetric that does not use our proposed distance-based program extraction technique. Instead, it randomly picks programs that are accepted by the

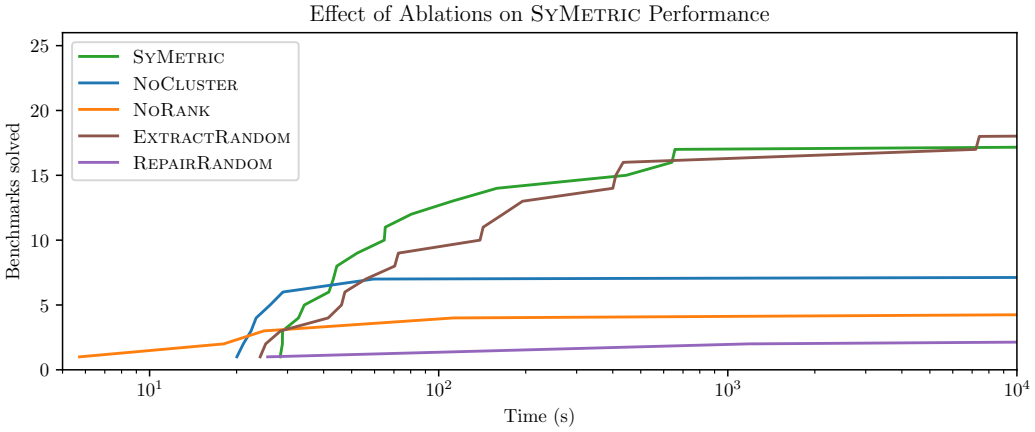


Fig. 8. Effect of our ablations on SyMetric.

automaton. (However, note that the final states of the XFTA are still determined using the distance metric.)

- RepairRandom: This is a variant of SyMetric that 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 8. We find that 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.). 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 Detailed Evaluation of Running Time

In this section, we explore the impact of different sub-procedures on running time. Specifically, Figure 9 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

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. 9. Runtime breakdown for the different sub-procedures of SyMetric. All times are in seconds.

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. 10. Runtime breakdown for the different sub-procedures of ConstructXFTA. All times are in seconds.

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 10 provides a more detailed look at XFTA construction. Recall that ConstructXFTA consists of three phases, namely expansion, clustering, and ranking. In Figure 10, we show the running time of each of these phases during XFTA construction. As we can see in this table, the 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.

Program synthesis for CAD. There has been a lot of interest in the CAD community in using program synthesis techniques to reverse engineer CAD problems. An early work in this space is InverseCSG [Du et al. 2018], where the goal is to translate from a point cloud representing the surface of a 3D shape to a set of operations in constructive solid geometry (CSG).

In some respects, InverseCSG is solving a much harder problem than what is being solved in this paper in that it works on real 3D CAD designs that can reach up to 100 primitives. On the other hand, in this paper we are solving a much harder *synthesis* problem than what was solved in InverseCSG. This is because, in InverseCSG, the actual program synthesizer is just one component in the middle of a larger pipeline. In particular, InverseCSG relies on a specialized preprocessing phase to identify all the primitives in a shape and their parameters, so the synthesizer only has to discover the Boolean structure of the shape, but not its primitives or their parameters. In contrast, we are asking the synthesizer to solve for all primitives and their parameters in addition to the Boolean structure. Additionally, Inverse CSG relies on a segmentation algorithm to break a large shape into small fragments that are then assembled into the final shape, whereas in our setting, we are aiming to solve the whole problem as a single synthesis problem. Finally, our program space is richer than that of InverseCSG because it includes a looping construct in addition to the primitives and Boolean operations. More recent work has shown that it is possible to post-process the output

of an InverseCSG-like system in order to extract loops [Nandi et al. 2020], but this can be expensive because it requires first synthesizing the loop free program, which can get quite large for models with a lot of repetition. 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.

Neural guided synthesis. Neural guided synthesis techniques have been previously applied to the CAD reconstruction problem. An early example of this paradigm exploited the output of partially constructed models to guide the search [Ellis et al. 2019] and inspired the ranking phase of our current algorithm. More recent forms of neural guided synthesis have been shown to support very expressive CAD formalisms that go beyond constructive solid geometry (e.g. [Willis et al. 2021]). A common limitation of all the tools in this space is that they require significant work ahead of time to collect a dataset and train the algorithm on that dataset. In contrast, our approach works out-of-the-box without the need for specialized training. There is a potential for future work that seeks to apply the insights of this work in a deep learning context.

Bottom-up synthesis. Our work builds heavily on bottom up synthesis. Early algorithms for bottom-up synthesis were introduced concurrently by Albarghouthi et al. and Udupa et al.. Wang et al. improved on the basic bottom-up synthesis approach by demonstrating the use of Finite Tree Automata (FTA) to efficiently represent the space of searched programs. Our work was particularly inspired by the work of Wang et al. that 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. As explained earlier, we improve upon this work by using instead a similarity metric to group related solutions and by using a local search to explore promising regions of the search space.

8 CONCLUSION AND FUTURE WORK

We presented a new synthesis technique, called *metric program synthesis*, and used it to solve the inverse CSG problem. The key idea behind our technique is to use a distance metric to cluster similar states together 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.

We have applied our proposed synthesis framework to the inverse CSG domain which satisfies the key assumptions underlying our approach. Our evaluation in this domain shows that our proposed technique is much more effective in this setting compared to existing synthesis algorithms. In particular, our approach can solve 78% of the benchmarks, whereas its closest competitor can only solve around 8%.

There are several interesting directions for future work. First, we are interested in applying our approach to other domains that contain many semantically similar programs that are also syntactically close. Second, we are interested in exploring improvements to our approach for the inverse CSG domain so that we can synthesize even more complex scenes. Finally, we are interested in automated refinement techniques that can be used to adaptively refine the radius of clusters during bottom-up enumeration, with the goal of improving scalability even further.

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

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.

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

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

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

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, 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

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

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. 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