Selecting Representative Examples for Program Synthesis

Yewen Pu [1] **Zachery Miranda** [1] **Armando Solar-Lezama** [1] **Leslie Pack Kaelbling** [1]

Abstract

Program synthesis is a class of regression problems where one seeks a solution, in the form of a source-code program, mapping the inputs to their corresponding outputs exactly. Due to its precise and combinatorial nature, program synthesis is commonly formulated as a constraint satisfaction problem, where input-output examples are encoded as constraints and solved with a constraint solver. A key challenge of this formulation is scalability: while constraint solvers work well with a few well-chosen examples, a large set of examples can incur significant overhead in both time and memory. We describe a method to discover a subset of examples that is both small and representative: the subset is constructed iteratively, using a neural network to predict the probability of unchosen examples conditioned on the chosen examples in the subset, and greedily adding the least probable example. We empirically evaluate the representativeness of the subsets constructed by our method, and demonstrate such subsets can significantly improve synthesis time and stability.

1. Introduction

Program synthesis (or synthesis for short) is a special class of regression problems where rather than minimizing the error on an example dataset, one seeks an exact fit of the examples in the form of a program. Applications include synthesizing database relations (Singh et al., 2017), inferring excel-formulas (Gulwani et al., 2012), and compilation (Phothilimthana et al., 2016). The synthesized programs are complex, consisting of branches and loops. Recent efforts (Ellis et al., 2015; Singh et al., 2017) show an interest in applying the synthesis technique to large sets of examples, but scalability remains a challenge. We present a method that selects a small *representative subset* of examples from a dataset, such that it is sufficient to specify a correct program, yet small enough to encode efficiently.

There are two key ingredients to a synthesis problem: a domain specific language (DSL for short) and a specification. The DSL defines a space of candidate programs which serve as the model class. The specification is commonly expressed as a set of input-output examples which the candidate program needs to fit exactly. The DSL restricts the structure of the programs in such a way that it is impossible to fit the input-output examples in an ad-hoc fashion: This structure aids generalization to an unseen input despite fitting the training examples exactly.

Given the precise and combinatorial nature of synthesis, gradient-descent based approaches perform poorly and an explicit search over the solution space is required (Gaunt et al., 2016). For this reason, synthesis is commonly casted as a constraint satisfaction problem (CSP) (Solar-Lezama, 2013; Jha et al., 2010). In such a setting, the DSL and its execution can be thought of as a parametrized function F, which is encoded as a logical formula. Its parameters $s \in S$ correspond to different instantiations of programs within the DSL, and the input-output examples D are expressed as constraints which the instantiated program needs to satisfy, namely, producing the correct output on a given input.

$$\exists s \in S. \bigwedge_{(x_i, y_i) \in D} F(x_i; s) = y_i \ .$$

The encoded formula is then given to a constraint solver such as Z3 (de Moura & Bjørner, 2008), which solves the constraint problem, producing a set of valid parameter values for s. These values are then used to instantiate the DSL into a concrete, executable program.

A key challenge of framing a synthesis problem as a CSP is that of scalability. While solvers have powerful heuristics to efficiently prune and search the constrained search space, constructing and maintaining the symbolic formula over a large number of constraints constitutes a serious overhead[1]. Developers of synthesis systems put significant effort into simplifying and rewriting the constraint formula into

[1]Massachusetts Institute of Technology. Correspondence to: Yewen Pu <yewenpu@mit.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

[1]However, if the solver does manage to construct and maintain all the constraints, solving the constraints can be fast as the constraints allow the solver to prune the search space.

a more compact representation (Singh & Solar-Lezama, 2016; Cadar et al., 2008). Nonetheless, to apply program synthesis to a large dataset, one needs to limit the number of examples expressed as constraints.

The standard method to limit the number of examples is CEGIS (counter example guided inductive synthesis) (Solar-Lezama et al., 2006). CEGIS employs two adversarial sub-routines, a synthesizer and a checker: The synthesizer solves the CSP on a subset of examples rather than on the whole set, producing a candidate program; the checker takes the candidate program and produces an adversarial counter example that invalidates the candidate program. This adversarial example is then added to the subset of examples, prompting the synthesizer to improve. CEGIS successfully terminates when the checker fails to produce an adversarial example. By iteratively adding counter examples to the subset as needed, CEGIS can drastically reduce the size of the constraints constructed by the synthesizer, making it scalable to large datasets. However, CEGIS has to repeatedly invoke the constraint solver in the synthesis sub-routine, solving a sequence of challenging CSP problems. Moreover, due to the phase transition (Gent & Walsh, 1994) property of SAT formulas, there may be instances in the sequence of CSPs with enough constraints to make the problem difficult, yet not enough constraints for the solver to prune the search space[2], making the performance of CEGIS volatile.

We describe a method that iteratively construct a *representative subset* of examples, which is both sufficient to specify a correct program and small enough to encode efficiently by a constraint solver. The algorithm is greedy: Starting with a (potentially empty) subset of examples, it uses a pre-trained neural network to compute the probability of other examples not in the subset conditioned on the subset, and extends the subset with the most "surprising" example (one with the smallest probability). The reason being if an example has a low probability conditioned on the given subset, then it is the most constraining example that can maximally prune the search space once added. The algorithm stops when all the input-output examples have a sufficiently high probability. Experiments show that our method does find representative subsets most of the times, and significantly improves synthesis time and stability on the the tasks of automaton induction and inverse rendering against strong baselines.

2. An Example Synthesis Problem

To best illustrate the synthesis problem and explain our approach, consider a diagram drawing DSL (Ellis et al., 2017) that allows a user to draw squares and lines. The DSL defines a $draw(row, col)$ function, which maps a (row, col) pixel-coordinate to a boolean value indicating whether the specified pixel coordinate is contained within one of the shapes. By calling the $draw$ function across a canvas, one obtains a rendering of the image where a pixel coordinate is colored white if it is contained in one of the shapes, and black otherwise. Figure 1 shows an example of a draw function and its generated rendering on a 32 by 32 pixel grid. The DSL contains a set of parameters that allows the $draw$ function to express different diagrams, which are in bold in Figure 1(left). The synthesis task is: Given a diagram rendered in pixels, discover the parameter values in the draw function so that it can reproduce the same rendering.

[2]Imagine a mostly empty Sudoku puzzle, the first few numbers and the last few numbers are easy to fill, whereas the intermediate set of numbers are the most challenging

A Drawing Program and its Rendering

```
def draw(row, col):
  shapes = []
  for i in range(3):
    for j in range(3):
      o_x = 10*i + 0*j + 5
      o_y = 0*i + 10*j + 0
      s = square(0+o_x, 5+o_y)
      l1 = line(0+o_x, 2+o_y, 0+o_x,
                -7+o_y, F, T, T)
      l2 = line(0+o_x, 5+o_y, -7+o_x,
                5+o_y, T, F, T)
      shapes += [s, l1, l2]
  for s in shapes:
    if inside(s, row, col): return T
  return F
```

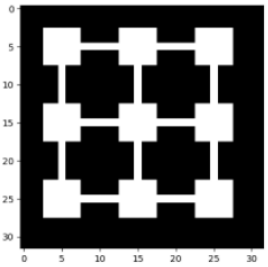

Figure 1. An example draw function (left) and its corresponding rendering (right). The parameters of the draw function are in bold, such as the number of iterations and offsets for the shapes.

The synthesized drawing program is correct when its rendered image matches the target rendering exactly. As the DSL contains control flow structures such as "for" and "if", it is a difficult combinatorial problem that requires the use of a constraint solver. Let $Sdraw$ be the synthesized draw function and $Target$ be the target rendering:

$$\textbf{correct}(Sdraw) \coloneqq \bigwedge_{(row,col)} Sdraw(row, col) = Target[row][col]$$

Here, each of the pixels in the target render is encoded as an input-output pair $((row, col), bool)$ that generates a distinct constraint on all of the parameters. For the 32 by 32 pixel image, a conjunction of 1024 distinct constraints are generated, which impose a significant encoding overhead.

In this paper, we propose an algorithm that approximates a representative subset of input-output examples. This subset is small, which alleviates the encoding overhead, yet remains representative of all the examples so that it sufficiently specifies the correctness condition. Figure 2 (top, left) shows the subset chosen by our algorithm. As we can see, from a total of 1024 examples, only 15% are selected for the representative subset. The representative subset is then given to the constraint solver, recovering the hidden parameter values in Figure 2 (top, right). By comparison,

the CEGIS algorithm chooses a much smaller number of examples that are not representative, Figure 2 (bottom, left): Despite the small size of the subset, since it is not representative, CEGIS ultimately achieves a longer solving time.

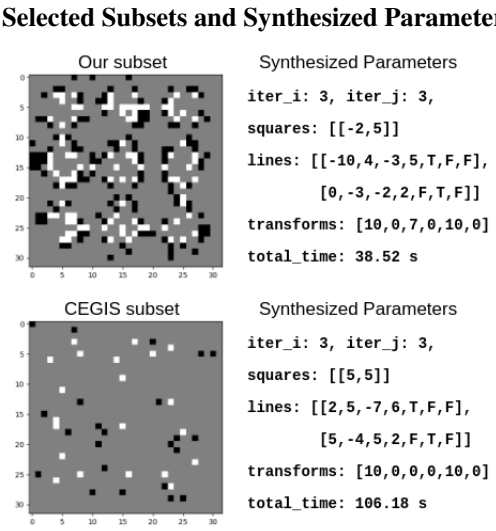

Figure 2. Our algorithm selects more examples than CEGIS, but since the subset is more representative, we achieve better time.

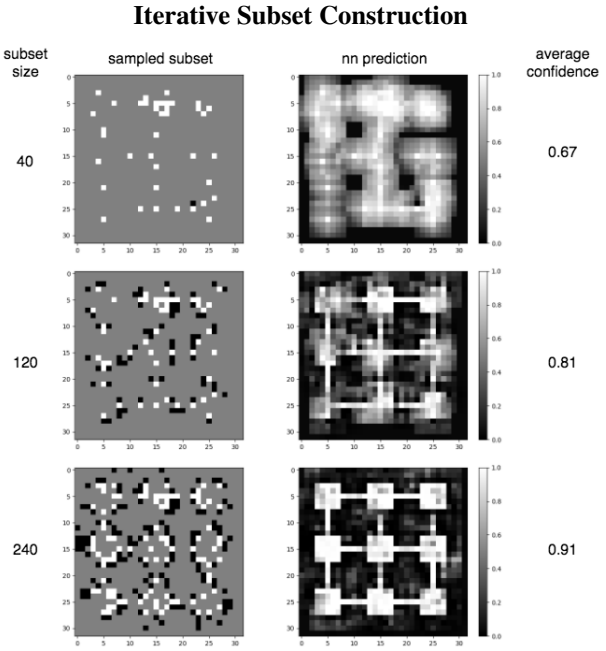

Figure 3. At each step, our algorithm predicts the pixel values of all the pixels conditioned on the sampled pixels, and samples additional pixels with the greatest reconstruction errors.

Our algorithm constructs the representative subset iteratively. Starting with an empty subset, the algorithm uses a neural network to compute the probability of all the examples conditioned on the chosen examples in the subset. It then adds the least probable example to the subset, the intuition being the example with the lowest probability would best prune the search space as a constraint. The algorithm terminates when all the examples in the dataset are given a sufficiently high probability. An example execution of our selection algorithm is shown in Figure 3. The rest of the paper elaborates our approach.

Algorithm 1 greedy selection with count oracle c

Input: data D
Output: data subset D'
Initialize $D' = \{\}$.
repeat
  $(x, y) \leftarrow \text{argmin}_{(x_j,y_j)}\, c(D' \cup \{(x_j, y_j)\})$ # selection
  $D' \leftarrow D' \cup \{(x, y)\}$
until $c(D') = c(D' \cup \{(x, y)\})$
return D'

3. Discovering Representative Examples

The crux of our algorithm is an example selection scheme, which takes in a set of examples and outputs a small subset of representative examples. Let $D' \subseteq D$ be a subset of examples. Abusing notation, let us define the *consistency constraint* $D'(s) \coloneqq \bigwedge_{(x_i,y_i)\in D'} F(x_i; s) = y_i$, that is to say, the parameter[3] s is consistent with all examples in D'. We define the *optimal representative subset* as:

$$D^* = \underset{D'\subseteq D}{\text{argmin}} |D'| \;\; s.t. \;\; \forall s \in S.\;\; D'(s) \Rightarrow D(s).$$

D^* is *representative* of D in a sense any parameter s satisfying D^* must also satisfy D. Finding the exact minimum sized D^* is often intractable, thus we focus on finding a sufficient subset that is as close in size to D^* as possible.

3.1. Examples Selection: a Greedy Strategy

We start with an approximate algorithm with a count oracle c, which counts the number of valid solutions with respect to a subset of examples: $c(D') \coloneqq |\{s \in S | D'(s)\}|$. Algorithm 1 constructs the subset D' greedily, choosing the example that maximally prunes the solution space.

Claim 1: Algorithm 1 produces a subset D' that is representative, i.e. $\forall s \in S.\; D'(s) \Rightarrow D(s)$.

Proof 1: As $D'(s)$ is defined as a conjunction of satisfying each example, c can only be monotonically decreasing with each additional example/constraint: $c(D') \geq$

[3] We'll refer to s as either a "parameter" or a "program" from now on, whichever is most appropriate given the context.

$c(D' \cup \{(x,y)\})$. At termination, the counts remain unchanged $c(D') = c(D' \cup \{(x,y)\}), \forall (x,y) \in D$, meaning no more solutions can be invalidated. Thus we obtain the sufficiency condition $\forall s \in S.\ D'(s) \Rightarrow D(s)$.

Claim 2: Let $prune(D) \coloneqq |\{s \in S | \neg D(s)\}|$ denotes the number of programs invalidated by D, and $k^{opt} = |D^*|$ the size of the optimal subset, then the greedy subset returned by Algorithm 1 satisfies $|D^g| < \frac{\log(prune(D))}{\log(k^{opt}) - \log(k^{opt}-1)}$

Lemma 2.1: It will be helpful to first show that the function $prune(\cdot)$ is both monotonic and sub-modular.

Proof 2.1: To show monotonicity, note that the constraint generated by the examples are conjunctive, thus adding examples strictly increases the number of invalidated programs.

To show sub-modularity, we require for $A \subseteq B \subseteq D$:

$$\begin{aligned}\forall (x,y) \in D.\ & prune(A \cup \{(x,y)\}) - prune(A) \\ \geq\ & prune(B \cup \{(x,y)\}) - prune(B)\end{aligned}$$

Let $A'(s) \coloneqq A(s) \wedge \neg\{(x,y)\}(s)$, the constraint stating that a program s should satisfy A, but fails to satisfy (x,y); Similarly, let $B(s)' \coloneqq B(s) \wedge \neg\{(x,y)\}(s)$. Then, the count $c(A')$ measures how many parameter s becomes invalidated by introducing (x,y) to A, i.e. $c(A') = prune(A \cup \{(x,y)\}) - prune(A)$, similarly, $c(B') = prune(B \cup \{(x,y)\}) - prune(B)$. Note that A' and B' are conjunctive constraints, with B' strictly more constrained than A' due to $A \subseteq B$. Thus $c(A') \geq c(B')$, and we have sub-modularity of $prune(\cdot)$ as claimed.

Proof 2: We now derive an upper bound on the size of the representative subset returned by Algorithm 1. As $prune(\cdot)$ is monotonic and sub-modular, Nemhauser et al. (1978) showed that for any optimal subset of size $k = |D^*_k|$, the greedily constructed subset of size $i = |D^g_i|$ satisfies:

$$prune(D^g_i) \geq (1 - (\frac{k-1}{k})^i) prune(D^*_k)\ .$$

Let $rem(D') = prune(D) - prune(D')$ be the remaining number of solutions yet to be pruned by $D' \subseteq D$. After subtracting both sides of the inequality from $prune(D)$:

$$rem(D^g_i) \leq prune(D) - (1 - (\frac{k-1}{k})^i) prune(D^*_k)\ .$$

Set $k = k^{opt}$, the size of the optimal representative subset, we can substitute $prune(D^*_{opt})$ with $prune(D)$:

$$rem(D^g_i) \leq prune(D) - (1 - (\frac{k^{opt}-1}{k^{opt}})^i) prune(D)\ .$$

Algorithm 1 terminates when there are no more programs to prune, i.e. when $rem(D^g_i) < 1$:

$$rem(D^g_i) \leq prune(D) - (1 - (\frac{k^{opt}-1}{k^{opt}})^i) prune(D) < 1\ .$$

Rearranging terms we see Algorithm 1 terminates when:

$$i < \frac{log(prune(D))}{\log(k^{opt}) - \log(k^{opt}-1)}\ .$$

Unfortunately, this is a rather loose upper-bound as the difference between $\log(k)$ and $\log(k-1)$ is quite small. However, in some instances it can still be helpful: If $prune(D) = 1.0e6$ and $k^{opt} = 20$, we have $|D^g| < 270$, which could be significantly smaller than $|D|$. In the experiment section we explicitly measure the size of the subset returned by our algorithm, showing that in practice one could obtain much smaller subsets than this upper-bound.

The issue with Algorithm 1 is that it requires access to a model counting (Gomes et al., 2008) oracle, which is impractical in practice. We now aim to resolve this issue.

3.2. Example Selection: by Anticipating New Examples

We describe an alternative selection criteria that can be approximated efficiently with a neural network. Let's write the selected subset D' as $\{(x^{(1)}, y^{(1)}) \ldots (x^{(r)}, y^{(r)})\}$ where $(x^{(j)}, y^{(j)})$ denotes the j^{th} input-output example to be added to D'. We define the *anticipation probability*:

$$\begin{aligned}Pr((x,y)|D') \coloneqq & Pr(F(x;s) = y | D'(s)) \\ = & Pr(F(x;s) = y | F(x^{(1)};s) = y^{(1)}, \\ & \ldots, F(x^{(r)};s) = y^{(r)})\end{aligned}$$

Note that $Pr((x,y)|D')$ is **not** a joint distribution on x and y, but rather the probability for the event where the parameterized function $F(\cdot\,; s)$ maps the input x to y, conditioned on the event where $F(\cdot\,; s)$ is consistent with all the input-output examples in D'. We claim that one can use $Pr((x,y)|D')$ as an alternative to the count oracle c.

Claim 3: Assuming uniform distribution $s \sim unif(S)$:

$$\operatorname*{argmin}_{(x,y)} c(D' \cup \{(x,y)\}) = \operatorname*{argmin}_{(x,y)} Pr((x,y)|D')\ .$$

Proof 3: The probability $Pr((x,y)|D')$ can be written as a summation over all the possible parameter values for s:

$$\begin{aligned}Pr((x,y)|D') \coloneqq & Pr(F(x;s) = y | D'(s)) \\ = & \sum_{s \in S} Pr(s|D'(s)) Pr(F(x;s) = y | s)\ .\end{aligned}$$

Note that under $s \sim unif(S)$, we have:

$$Pr(s|D'(s)) = \begin{cases} \frac{1}{c(D')} & \text{if } D'(s) \\ 0 & \text{otherwise} \end{cases}\ .$$

And since $F(\cdot\,; s)$ is a function we have:

$$Pr(F(x;s) = y|s) = \begin{cases} 1 & \text{if } F(x;s) = y \\ 0 & \text{otherwise} \end{cases} .$$

Thus the summation over all s results in:

$$\sum_{s \in S} Pr(s|D'(s)) Pr(F(x;s) = y|s) = \frac{c(D' \cup \{(x,y)\})}{c(D')} .$$

As $c(D')$ is constant under $\text{argmin}_{(x,y)}$ given D', we have $\text{argmin}_{(x,y)}\, c(D' \cup \{(x,y)\}) = \text{argmin}_{(x,y)}\, Pr((x,y)|D')$ as claimed.

It is easy to see that one needs to update the termination condition to $\min_{(x,y)} Pr((x,y)|D') = 1$, when all the input-output examples are completely anticipated given D'.

4. Neural Network Model

We now describe the high level neural network architecture that models the anticipation probability, $Pr((x,y)|D')$.

4.1. Factorization: Enabling Scaling with $|D'|$

A challenge to our neural network encoding is the ability of our model to scale with the size of D': as we might collect a large subset of examples, and the probability of each new example (x,y) depends on the entire subset D'.

To address this, we make an independence assumption: Let $nb_{k,x}$ be the neighborhood function that computes the top-k neighbors of x from D' and condition (x,y) only on these top-k neighbors:

$$\begin{aligned} Pr((x,y)|D') =& Pr((x,y)|nb_{k,x}(D')) \\ =& Pr((x,y)|(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb})) . \end{aligned}$$

We assume the programmer would be able to come up with an appropriate neighborhood function for each synthesis task. In our experiments, the neighborhood is measured by a distance metric on the input space X. For example, we have used a convolutional neural network (which implicitly uses pixel to pixel distance) and longest matching suffix (sub-string distance). Although we remark that in general, a neighborhood need not depend on a distance metric but can be as arbitrary as needed.

4.2. Anticipation Network: Direct Computation

Figure 4 (top) shows a neural network architecture that models the factorized anticipation probability $Pr((x,y)|(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb}))$ directly.

We train the network on the task of correctly anticipating whether an input-output pair (x,y) should occur based on the top-k neighbors of (x,y) from a subset D'. To do this, we sample a program $s \sim unif(S)$ and a subset of inputs $\{x_1 \ldots x_n | x_i \sim unif(X)\}$. We evaluate the program on the set of inputs to produce a dataset of example pairs $D = \{(x_1, F(x_1;s)) \ldots (x_n, F(x_n;s))\}$. We can then sample a subset $D' \subseteq D$ and a new example $(x,y) \in D \setminus D'$. From this new example, we compute its top-k neighbors $\{(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb})\} = nb_{k,x}(D')$. We also construct a negative sample (x, y_{neg}) by sampling a random $y_{neg} \in Y \setminus \{y\}$. The network is then trained to produce $True$ on the input $\{(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb}), (x,y)\}$ and to produce $False$ on the input $\{(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb}), (x, y_{neg})\}$

Neural Network Architecture

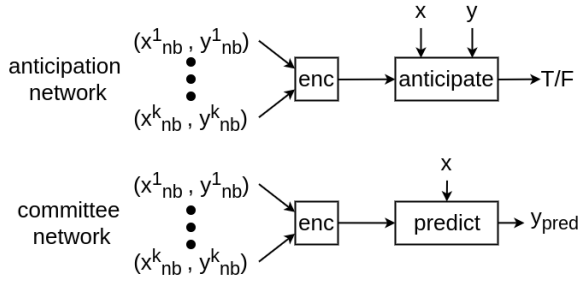

Figure 4. The anticipation network (top) that computes the anticipation probability directly, and the committee network (bot) computes the output on a new given input x

4.3. Committee Network: Peer Consultation

We now present an equivalent neural network architecture that affords a more intuitive understanding. Figure 4 (bot) shows the committee network, which computes an output distribution y_{pred} rather than the anticipation probability. The two architectures are equivalent because:

$$\begin{aligned} & Pr((x,y)|(x^1_{nb}, y^1_{nb}), \ldots, (x^k_{nb}, y^k_{nb})) \\ =& Pr(F(x;s) = y | F(x^1_{nb};s) = y^1_{nb}, \ldots, F(x^k_{nb};s) = y^k_{nb}) \end{aligned}$$

The committee network is trained on the task of producing the correct value y, which has the implicit effect of negative sampling. This network has a very intuitive interpretation: To best predict the a function's output on a new example x, we consult the subset D' for the top-k most relevant input-output pairs to make a prediction on the value of y.

In practice, each synthesis domain would require a different neural-network architecture, as the input/output types of the functions being synthesized and the neighborhood function are domain specific. However, the overall neural-network task remains the same: predicting the function's output on a new example x based on the nearest-k neighbors of x already present in the subset D'. We'll describe the domain specific architecture in detail in the Experiment section.

Algorithm 2 CEGIS

Input: data D, initial subset $D' = \{\}$
Output: satisfying program s
repeat
  $s \leftarrow synthesize(D')$
  $ce \leftarrow check(s, D)$
  $D' \leftarrow D' \cup \{ce\}$
until $check(s, D) =$ None
return s

Algorithm 3 Synthesis with Representative Examples

Require: trained committee model $nn(nb_{x,k}(D'), x)$ approximating $Pr(y|nb_{x,k}(D'), x)$
Input: data D
Output: satisfying program s
Initialize $D' = \{\}$.
repeat
  # Find the least likely input-output example
  $(x, y) \leftarrow \text{argmin}_{(x_j,y_j)}\, nn(nb_{x_j,k}(D'), x_j)(y_j)$
  $D' \leftarrow D' \cup \{(x, y)\}$
until confident(nn, D', D)
return CEGIS(D, D')

5. Synthesis with Representative Examples

The neural network cannot perfectly model the anticipation probability, thus, our example selection algorithm can only approximate a representative subset, causing the synthesized program to be inconsistent with the entire dataset of examples. We remedy this problem by combining example selection and CEGIS, getting the best of both worlds.

5.1. CEGIS: Guarantees with Caveats

CEGIS (Solar-Lezama et al., 2006) is a synthesis algorithm which guarantees total correctness on a set of examples D. It is outlined in Algorithm 2. CEGIS is composed of two adversarial sub-routines, a synthesizer and a checker: The synthesizer produces a candidate program s over the subset D', which is initially empty; The checker takes in this candidate program s and produces an adversarial counter example $ce \in D \setminus D'$ which invalidates s. ce is added to the subset of examples, prompting the synthesizer to improve. CEGIS successfully terminates when the checker fails to produce an adversarial example.

At first glance CEGIS is very similar to our approach, but a deeper look reveals several important differences. First, the subset D' of examples constructed by CEGIS upon termination is *not* representative: It is possible for CEGIS to synthesize the correct program without constructing a representative subset by luck, a fact we demonstrate empirically in the experiments. The danger of synthesis over a non-representative subset is that there might be instances where there are enough constraints to make the synthesis problem challenging, yet not enough constraints for the solver to prune the search space. The result is the hanging of the solver for an extended periods of time, without any guarantee whether the synthesis would ever terminate. Secondly, to build the subset of counter examples, CEGIS must solve $|D'|$ instances of constraint problems, each one with the potential to timeout due to being under-constrained.

5.2. Our Algorithm: Best of Both Worlds

Our algorithm combines representative example discovery and CEGIS by instantiating the subset of counter examples in CEGIS with a representative subset, see Algorithm 3. This algorithm guarantees complete correctness over the input dataset D while alleviating the challenges of CEGIS by presenting CEGIS with a well-constrained subset upfront.

6. Experiments

Our approach is evaluated against two criteria: First, the representativeness of our selected subset is explicitly measured; Then, the time/stability improvement of using such a subset is measured against several strong baselines.

6.1. Explicitly Measuring Representativeness

This experiment explicitly measures the representativeness of the subset selected by our algorithm on the task of ordering synthesis: Given a dataset of pair-wise ordering relations, $D = \{a < b, a < c, b < d, c < d, d > a, c > a\}$, the task is to synthesize any total-ordering that is consistent with D, for instance, (a, b, c, d) or (a, c, b, d). This task is useful because the optimal representative subset can be constructed as a Hasse diagram (Aho et al., 1972) by pruning transitive relations: $D^* = \{a < b, a < c, b < d, c < d\}$. Thus, we can measure both the representativeness and optimality of our selection algorithm. In this experiment, we set $n = 10$ and give our selection algorithm a dataset of size 30% to 100% of all possible pair-wise orderings. Since there are only 100 possible pair-wise relations for $n = 10$, we use a fully-connected neural network without factorization. Refer to the supplementary for specifics of this network and an algorithm to verify representativeness.

We test the representativeness of our approach against the following baselines: **cegis**, **random x**(randomly select x percent of dataset)[4], and **hasse** (the optimal construction). The measurement of average subset size and fraction of representative subsets is shown in Figure 5. As we can see, **our** approach selects about twice as many examples as the optimal subset, and 85% of the times our subsets are repre-

[4] we use 35% because it matches our average subset size

sentative. By contrast, **cegis** and **rand35** fails to discover any representative subset, while **rand80** discovers representative subset only 30% of the times despite sampling 80% of the total data. Figure 6 visualizes the chosen pair-wise orderings on a particular dataset **all**. This dataset specifies a unique total-ordering, which **hasse** was able to concisely represent with the minimal representative subset (bottom). **our** approach also discovers a representative subset, albeit with a few extra redundant relations. By contrast, **cegis** and **rand35** fail to discover a representative subset, as their subsets lack the relationship between elements 7 and 0, which are adjacent in the total ordering.

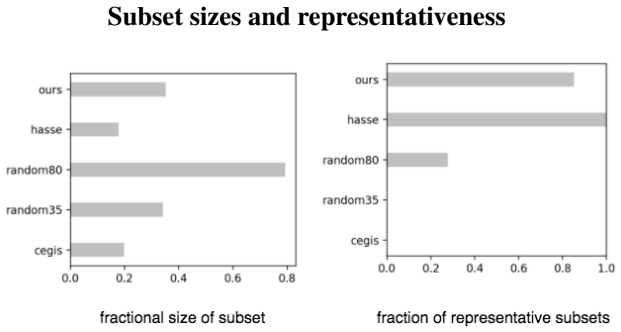

Figure 5. Our approach discovers representative subsets 85% of the times while sampling $2\times$ the optimal subset size. Measured on 500 datasets drawn from randomly sampled total orderings

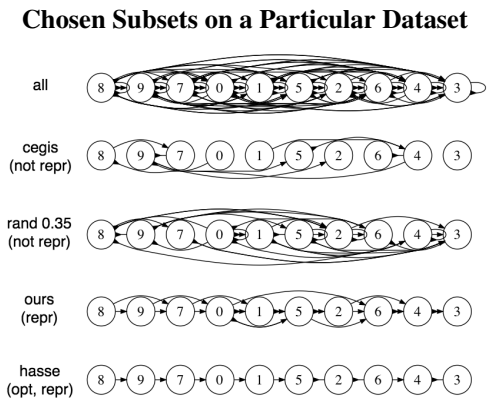

Figure 6. Chosen subsets on a particular dataset **all**. A subset is representative if it contains all adjacent pair-wise ordering

6.2. Measuring Improved Synthesis Times

We now measure the performance in terms of on time and stability by using our approach on two distinct tasks.

DFA Synthesis The task is to synthesize a deterministic finite-state automaton (DFA) from a set of accepted and rejected strings. We use a DSL which contains DFA of 6 states over a binary alphabet of 0 and 1 with a single accept state. The search space of total possible DFAs is of size $6^{12} = 2.18 \times 10^9$. 1000 strings of variable length between 5 and 10 were provided as the dataset for each synthesis task, the experiment consists of 400 tasks. On this domain, given a new example string str, the neighborhood function selects the top 10 closest prefix and suffix matching examples from the subset D'. The neural network architecture is a simple feed-forward neural network that predicts the accept/reject label of str directly (see supplementary for parameters details).

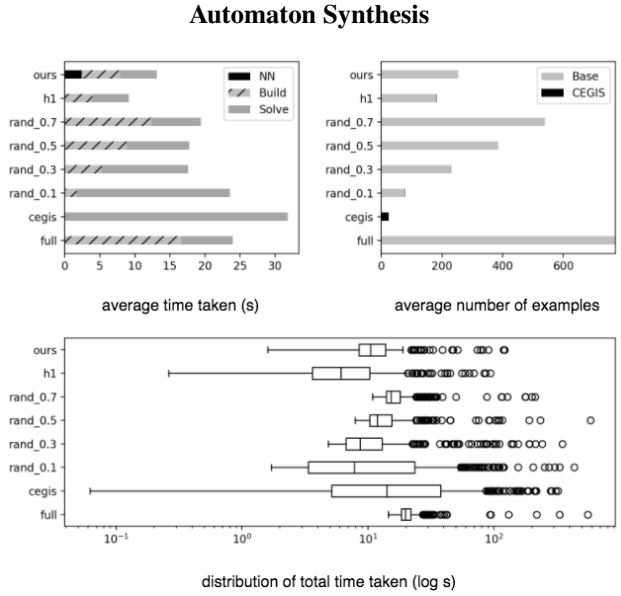

Figure 7. Time performance on DFA synthesis. **our** approach nearly matches the crafted heuristic **h1**, which constructs a suffix-tree over the entire dataset D, and out performs all other baselines.

We measure performance against the following: **full** (all examples are added), **cegis**,**rand_x** (initialize CEGIS with a random x fraction of data), **h1** (a heuristic that construct a suffix-tree over the entire dataset, see supplementary). Figure 7 (top,left) shows the comparison of performances in average time. As we can see, the heuristic subset collection **h1** performs best on average, but **our** approach comes in close (If we disregard the example selection time from NN, the two performs similarly). As we can see, **ours**, **h1**, **full** have similar solve time, which we can infer that **our** approach and **h1** have found a well-constraining representative subset. This is in stark contrast to **cegis** which explodes in solve time with hardly any examples chosen. In terms of stability (Figure 7 (bot)), **our** approach also closely matches that of the heuristic, whereas all other algorithms (except **full**) suffers big variance in total time, likely a result of performing synthesis on under-representative subsets. Figure 7 (top,right) shows the average number of examples in the collected subset, we see that **our** approach outperforms randomly selected subsets of any size.

Programmatic Drawing Synthesis We evaluate our approach on 250 randomly sampled 32×32 pixel renderings created from the drawing DSL in Section 2, the drawing function has a parameter space of size 1.31×10^{23}. The neighborhood function is simply a 7×7 sliding window centered on each pixel, and is implemented as a convolutional neural network (see supplementary for parameter details).

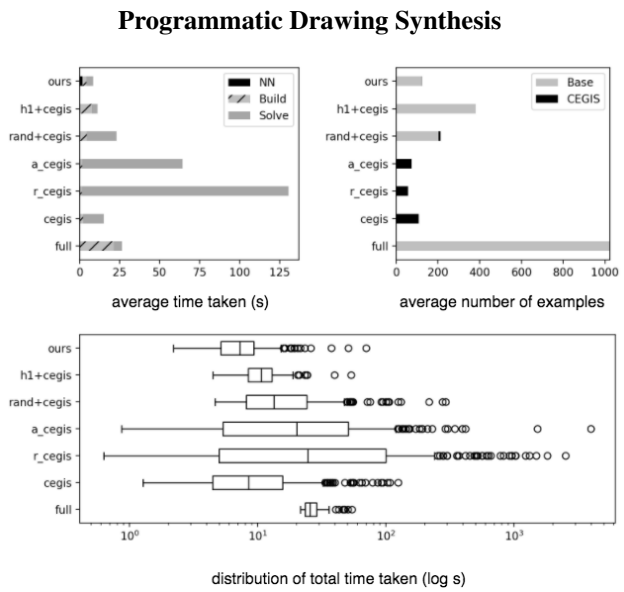

Figure 8. Time performance on programmatic drawing synthesis. **our** approach is best in average time, and achieves similar stability as **full** and **h1+cegis** with much fewer samples.

We measure performance against the following: **full** (all examples are added), **cegis, rcegis, acegis** (different CEGIS flavours on how the counter examples are selected: canonical top-left most pixel, random, and a fixed but arbitrary order), **rand+cegis** (instantiate CEGIS with a random 20% subset), and **h1+cegis** (a heuristic that adds a pixel if any pixel within a 5×5 window has a different value). The results are shown in Figure 8. As we can see, **our** approach performs best on average, beating all competitors on average time. One unexpected outcome is that **cegis** performs very well on this domain: We postulate that the top-left-most counter-examples chosen by **cegis** happen to be representative as they tend to lay on the boundaries of the shapes, which is well suited for the drawing DSL domain. However, such coincidence is not to be expected in general: By making the counter example be given at random, or given at a fixed but arbitrary ordering, **rcegis** and **acegis** were unable to pick a representative set of examples and suffer in overall time. In terms of variance (Figure 8 (bot)), our approach was able to match the variance of **full** (clear representative) and **h1+cegis** (also representative as a 5×5 sliding window can distinguish squares and lines perfectly). However, **our** approach was able to discover representative subsets with a much smaller number of examples (Figure 8 (top, right)).

Overall, our approach improves synthesis time and stability by providing CEGIS with a representative subset upfront. [5]

7. Related Work

In recent years there have been an increased interest in *program induction*. Graves et al. (2014), Reed & De Freitas (2015), Neelakantan et al. (2015) assume a differentiable programming model and learn the operations of the program end-to-end using gradient descent. In contrast, in our work we assume a non-differentiable programming model, allowing us to use expressive program constructs without having to define their differentiable counter parts. Works such as (Reed & De Freitas, 2015) and (Cai et al., 2017) assume strong supervision in the form of complete execution traces, specifying a sequence of exact instructions to execute, while in our work we only assume labeled input-output pairs to the program, without any trace information.

Parisotto et al. (2016) and Balog et al. (2016) learn relationships between the input-output examples and the structures of the program that generated these examples. When given a set of input-outputs, these approaches use the learned relationships to prune the search space by restricting the syntactic forms of the candidate programs. In contrast, our committee network learns a relationship between the input-output examples, a relationship entirely in the semantic domain. In this sense, these approaches are complimentary.

The predictive task of our neural network is similar to that of (Pu et al., 2017), which learns the inter-relationships between observations for active learning. In contrast, in our domain the labels to the observations are known in advance. The committee neural-network structure is most similar to the meta program-induction network in (Devlin et al., 2017). One key difference being we assume a neighborhood function which both limits and orders neighboring input-output examples to be encoded. As our subset D' can grow arbitrarily large, having a hard cap on the number of neighbors is important for efficiency.

Acknowledgements

We like to thank the reviewer for their helpful insights; Xin Zhang, Osbert Bastani for constructive criticisms; Steve Mussmann for in depth reference on sub-modularity; and Twitch Chat for moral supports.

This work was funded by the MUSE program (Darpa grant FA8750-14-2-0242).

[5] The supplementary material and the code can be found at https://github.com/evanthebouncy/icml2018_selecting_representative_examples

References

Aho, A. V., Garey, M. R., and Ullman, J. D. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. doi: 10.1137/0201008. URL https://doi.org/10.1137/0201008.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Cadar, C., Dunbar, D., and Engler, D. R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pp. 209–224, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

Cai, J., Shin, R., and Song, D. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

de Moura, L. M. and Bjørner, N. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pp. 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

Devlin, J., Bunel, R. R., Singh, R., Hausknecht, M., and Kohli, P. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, pp. 2077–2085, 2017.

Ellis, K., Solar-Lezama, A., and Tenenbaum, J. B. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 973–981, 2015.

Ellis, K., Ritchie, D., Solar-Lezama, A., and Tenenbaum, J. B. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.

Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL http://arxiv.org/abs/1608.04428.

Gent, I. P. and Walsh, T. The sat phase transition. In *ECAI*, volume 94, pp. 105–109. PITMAN, 1994.

Gomes, C. P., Sabharwal, A., and Selman, B. Model counting, 2008.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012. doi: 10.1145/2240236.2240260. URL http://doi.acm.org/10.1145/2240236.2240260.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pp. 215–224, 2010. doi: 10.1145/1806799.1806833. URL http://doi.acm.org/10.1145/1806799.1806833.

Neelakantan, A., Le, Q. V., and Sutskever, I. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294, 1978.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Phothilimthana, P. M., Thakur, A., Bodík, R., and Dhurjati, D. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pp. 297–310, 2016. doi: 10.1145/2872362.2872387. URL http://doi.acm.org/10.1145/2872362.2872387.

Pu, Y., Kaelbling, L. P., and Solar-Lezama, A. Learning to acquire information. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*, 2017. URL http://auai.org/uai2017/proceedings/papers/237.pdf.

Reed, S. and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Singh, R. and Solar-Lezama, A. SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pp. 185–192, 2016. doi: 10.1109/FMCAD.2016.7886678. URL https://doi.org/10.1109/FMCAD.2016.7886678.

Singh, R., Meduri, V., Elmagarmid, A. K., Madden, S., Papotti, P., Quiané-Ruiz, J., Solar-Lezama, A., and Tang, N. Generating concise entity matching rules. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 1635–1638, 2017. doi: 10.1145/3035918.3058739. URL http://doi.acm.org/10.1145/3035918.3058739.

Solar-Lezama, A. Program sketching. *STTT*, 15(5-6): 475–495, 2013. doi: 10.1007/s10009-012-0249-7. URL https://doi.org/10.1007/s10009-012-0249-7.

Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S. A., and Saraswat, V. A. 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*, pp. 404–415, 2006. doi: 10.1145/1168857.1168907. URL http://doi.acm.org/10.1145/1168857.1168907.