Syntax-Guided Synthesis with Counterexample-Guided E-graphs: A Work-in-Progress Report

Guy Frankel[1,*], Rudi Schneider[2], Michel Steuwer[2] and Elizabeth Polgreen[1]

[1]University of Edinburgh, Edinburgh, UK
[2]Technische Universität Berlin, Berlin, Germany

Abstract
Program synthesis algorithms suffer serious scalability issues due to the exponential growth of the space of solutions as programs grow longer. A common way to tame this search space in programming-by-example is to use *observational equivalence*. In this work in progress report we discuss a method for using equivalence graphs (e-graphs) to lift observational equivalence to more general program synthesis problems. We first present a naive synthesis algorithm using an e-graph to perform bottom-up enumeration. We also propose an extension of e-graphs, using approximate equivalence guided by counterexamples. We evaluate a preliminary implementation of both approaches against a state-of-the-art solver, and further outline scope for improvements and further research.

Keywords
Program Synthesis, Equivalence Graphs, CEGIS, Observational Equivalence

1. Introduction

Given a grammar and a logical specification program synthesis methods attempt to find a program within the grammar that satisfies the logical specification. Program synthesis is hard [1] because it requires a search over an *intractable* program space [2].

A common method used to reduce this intractable search space is Observational Equivalence [3]. That is, in bottom-up enumerative search, which iteratively constructs new programs by combining previously generated programs, we can discard programs from the pool of programs used for construction if we observe that they behave equivalently on the specification. This is extremely effective when the specification is input-output examples, i.e., it reasons about a finite number of concrete values. For instance, given the single input-output example $2 \rightarrow 4$; `x+y` and `x*y` are observationally equivalent programs. However, showing that two programs are equivalent for a specification given as an arbitrary first order formula is significantly more challenging, and, as a result, equivalence has rarely been used for pruning for synthesis with logical specifications reasoning about infinite domains.

In this work, we leverage equivalence graphs (e-graphs) to lift equivalence based pruning to synthesis with logical specifications over infinite domains. E-graphs are well suited to compactly storing large sets of terms, and have been used for quantifier instantiation via e-matching [4], learning program optimizations [5] and abstractions [6], and synthesizing rewrite rules [7]. In this work, we present a synthesis algorithm that performs enumeration directly over an e-graph, eliminating the need to enumerate spurious terms by fully leveraging the grouping of symbolically equivalent terms captured by equality saturation. To the best of our knowledge, this is the first exploration of using e-graphs to solve general purpose program synthesis problems.

However, despite the compact representation afforded by e-graphs, the algorithm is still overwhelmed by *combinatorial explosion*. In an attempt to overcome this, we use equivalence relations that only hold under certain assumptions. A common method for synthesis over symbolic variables is Counterexample Guided Inductive Synthesis [8], CEGIS, whereby a synthesis phase enumerates through possible candidate programs, which are passed to a verification oracle which either validates that the candidate

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK
*Corresponding author.
g.frankel-1@sms.ed.ac.uk (G. Frankel)

is a solution to the synthesis problem or returns a counterexample that invalidates the candidate. The counterexamples are used to reduce the number of candidate solutions that need to be passed to the verifier, as each candidate can be quickly tested against the counterexamples first. Inspired by this, we present an algorithm that leverages counterexamples to generate *counterexample e-graphs* (CE-graphs), which capture equivalence relations that hold true under the counterexample. These CE-graphs allow us to aggressively *prune* the search space by discarding sets of programs that fail to satisfy each counterexample, and use the counterexamples to *guide the growth* of new programs constructed using recursive production rules to be only as deep as precisely needed by the counterexamples obtained so far.

1.1. Running Example

In this work, we consider syntax-guided synthesis problems, where a logical specification is given alongside a grammar expressing the space of possible solutions. An example is shown in Fig. 1, which gives a specification for a program that computes the maximum of 3 numbers. In the worst case, a traditional CEGIS loop would need to generate $\sim 50,000,000$ programs before finding one that satisfies the specification, with each of these needing to be checked against the counterexamples and a subset needing to be checked over the whole specification. It is evident that some method to reduce the number of programs generated is required in order to make such synthesis feasible.

2. Preliminaries

2.1. Syntax-Guided Synthesis

A syntax-guided synthesis (SyGuS) [9] problem consists of a context-free grammar, which defines the search space of possible solutions, and a logical specification. The task is to find a function from within that search space that satisfies the specification. First we define context-free grammars:

Definition 2.1 (Context-Free Grammar, CFG). *A context-free grammar is a 4-tuple $\mathcal{G} = (\Omega, \Sigma, R, S)$. Ω is a finite set of variables also known as non-terminal symbols. Σ with $\Sigma \cap \Omega = \emptyset$ is called the set of terminal symbols or alphabet. $R \subseteq \Omega \times (\Omega \cup \Sigma)^*$ is a finite relation describing the production rules of the grammar. $S \in \Omega$ is the start symbol of the grammar G.*

```
(set-logic LIA)
(synth-fun max3 ((x Int) (y Int) (z Int)) Int
  ((Start Int) (StartBool Bool))
  ((Start Int (x y z
            (ite StartBool Start Start)))
   (StartBool Bool ((>= Start Start)))))
(declare-var a Int)
(declare-var b Int)
(declare-var c Int)
(constraint (>= (max3 a b c) a))
(constraint (>= (max3 a b c) b))
(constraint (>= (max3 a b c) c))
(constraint (or
              (= a (max3 a b c))
              (= b (max3 a b c))
              (= c (max3 a b c))))
(check-synth)
```

Figure 1: Running Example: SyGuS program for synthesising a program that computes the maximum from three numbers

Given a context-free grammar $\mathcal{G} = (\Omega, \Sigma, R, S)$ with $x, y \in (\Omega \cup \Sigma)^*$ and $(\alpha, \beta) \in R$ we say that $x\alpha y$ yields $x\beta y$, written $x\alpha y \to x\beta y$. We say that x derives y written $x \to^* y$ if either $x = y$ or $x \to x_1 \to \ldots x_n \to y$ for $n \geq 0$. Finally, we define the *language* of a grammar (which gives the space of possible solutions for our SyGuS problem). $\mathcal{L}^{\mathcal{G}} = \{s \in \Sigma^* \mid S \to^* s\}$.

Definition 2.2 (Syntax-Guided Synthesis). *A syntax-guided synthesis problem is a 4-tuple $\langle T, \mathcal{G}, \phi, F\rangle$, where $\mathcal{G}$ is a context-free grammar, ϕ is a first-order formula in the background theory T, and F is a function symbol that occurs in ϕ. A valid solution to the SyGuS problem $\langle T, \mathcal{G}, \phi, F\rangle$ is a function f such that the formula $\phi[F/f]$ is T-valid, according to the background theory, and $f \in \mathcal{L}^{\mathcal{G}}$.*

We use $\phi[F/f]$ to indicate the result of substituting the symbol F with a defined function f, and we use $\vec{x} = x_1, x_2, \ldots$ to represent the set of free (nullary) variables in ϕ.

In this work, we assume the background theory is Linear Integer Arithmetic, and the grammar contains one non-terminal symbol of type integer, and one non-terminal symbol of type bool.

2.2. CounterExample Guided Inductive Synthesis

The most common algorithm for solving program synthesis problems is CounterExample Guided Inductive Synthesis, CEGIS, shown in Fig. 2.

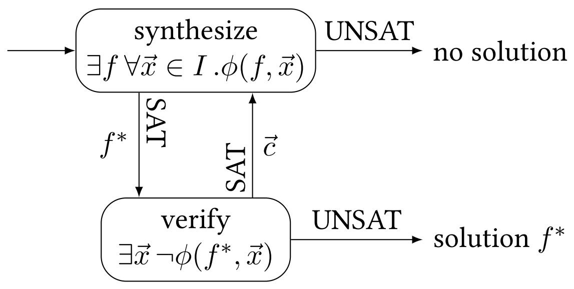
synthesize
$\exists f\,\forall\vec{x} \in I\,.\phi(f,\vec{x})$
UNSAT
no solution
f^*
SAT
SAT
$\vec{c}$
verify
$\exists\vec{x}\,\neg\phi(f^*,\vec{x})$
UNSAT
solution f^*

Figure 2: CounterExample Guided Inductive Synthesis

CEGIS alternates between two phases: We initiate the algorithm with a guessed candidate function f^*. In our running example, this could be x. This is passed to a verification phase that checks the candidate f^* against the specification ϕ over the full set of program inputs. That is, it uses an SMT solver to solve the formula $\exists\vec{x}.\neg\phi[F/f^*]$. If this formula is satisfiable, the solver returns $\vec{c}$, a counterexample assignment to all free variables in ϕ. In our running example, a counterexample might be $\vec{c} = [x \mapsto 0, y \mapsto 1, z \mapsto 0]$. If the verification passes, f^* is returned as the final solution.

If the verification phase fails, the counterexample $\vec{c}$ is returned to the synthesis phase and appended to a list of counterexamples obtained so far, I. The synthesis phase then looks for a new candidate program f^* that satisfies the specification for all $\vec{c} \in I$; that is, $\exists f^*.\forall\vec{c} \in I.\phi[F/f^*, \vec{x}/\vec{c}]$. In our running example, the next candidate solution might be y. This process would repeat, generating new counterexamples and new candidate solutions until we find a solution that passes verification.

The synthesis phase is typically implemented using enumerative techniques [10, 11, 12], which enumerate through programs in the search space until they find one that works for all counterexamples seen so far.

2.3. Equivalence Graphs

The e-graph data structure captures symbolic equivalence between terms through equivalence relations defined as rewrite rules. It also supports equality saturation; a non-destructive compositional method for propagating equivalences [13, 14].

In order to use e-graphs to express the space of possible solutions to a synthesis problem, let us first map the production rules in our grammar into function symbols. Given a production rule $\omega \to rhs$, where $\omega \in \Omega$ and $rhs \in (\Omega \cup \Sigma)^*$, we say the production rule is an n-ary production rule if rhs contains n occurrences of a non-terminal symbol.

Intuitively, an e-graph representing the space of terms within our context-free grammar can be given as:

- A set of e-classes, which consist of groups of e-nodes determined to be equivalent terms in the underlying language.
- An e-node is an n-ary production rule $\omega \to rhs$ from our grammar, with n associated child e-classes. An e-node represents the set of terms obtained by substituting the non-terminal symbols in rhs with any node from within the corresponding child e-classes. An e-node representing applications of $\omega \to rhs$ is annotated with its corresponding non-terminal symbol ω.

Given a synthesis problem $\langle T, \mathcal{G}, \phi, F\rangle$, we can define an e-graph which represents the space of possible solutions as follows:

Definition 2.3 (e-graphs)**.** *An e-graph G is as a tuple $(\mathcal{T}, \cong)$ where $\mathcal{T}$ is a set of terms in a $\mathcal{L}^{\mathcal{G}}$, and $\cong \subset \mathcal{T} \times \mathcal{T}$ is an equivalence relation, such that, for any $t_1, t_2 \in \mathcal{T}$ if $(t_1, t_2) \in \cong$, then $t_1 = t_2$ is T-valid. Each term is annotated with its corresponding non-terminal symbol.*

Rewrite rules An e-graph is equipped with *rewrite rules*, that define the equivalence relation as patterns of the form $\ell \to r$. If a rewrite rule finds a term matching the pattern ℓ, it will create a e-node representing the expression generated by the substitution defined by the rule. If the new e-node already exists in the e-graph, the e-class of this node and the e-node representing the original term are merged, otherwise the new e-node is added to the e-class of the original e-node. We assume that all rewrite rules result in programs that abide by the syntactic restriction of the SyGuS problem, i.e., rewrite rules cannot result in terms that are not contained within $\mathcal{L}^{\mathcal{G}}$.

For instance, take an e-graph of the term $(+\ x\ x)$ with the rewrite rule $(+\ ?a\ ?a) \to\ (\times\ ?a\ 2)$, where $?a$ is a pattern that captures any term. The equivalence relation defines that $(+\ ?a\ ?a) \cong (\times\ ?a\ 2)$. Applying the rewrite will add the two new e-nodes required to represent $(\times\ x\ 2)$ with the relevant children, namely 2 and $\times$, to the e-graph. Rewrite rules are applied until no further changes occur, referred to as equality saturation.

3. Bottom-up Equivalence Graph Synthesis

The first approach we present is a naive bottom-up enumerative algorithm using e-graphs to represent and efficiently prune the pool of possible programs. This enables us to *lift* the notation of observational equivalence, which has been used successfully to prune the search space of programs in programming-by-example, to more general synthesis specifications.

Bottom-up enumeration has been used as the synthesis phase in many CEGIS implementations [15, 9, 12]. It searches the space of solutions by growing a pool of programs from the grammar, starting with the terminals in the language. At each iteration, new programs are constructed by combining programs from the pool, using production rules from the grammar.

The pool of programs grows exponentially large as the length of the programs increases. For specifications over finite inputs, like programming-by-example, this exponential growth is tamed by *observational equivalence* [3]: if any programs in the pool behave the same on all inputs in the specification as others in the pool, we can discard all but one of those programs. However, for specifications which contain free variables that can take an infinite number of values, this equivalence check is expensive.

3.1. Naive Bottom-up Enumeration with Equivalence Graphs

In our approach, we lift this equivalence-based pruning to general synthesis specifications (even those reasoning about infinite inputs), by using e-graphs to represent the pool of programs and efficiently perform the equivalence check. The synthesis proceeds as follows:

Constructing the initial e-graph Given a syntax-guided synthesis problem $\langle T, \mathcal{G}, \phi, F \rangle$, we initially construct an e-graph $G = (\mathcal{T}, \cong)$, where $\mathcal{T}$ is the set of terms on $\mathcal{L}^{\mathcal{G}}$ that can be obtained from the start symbol S with 1 derivation, and $\cong$ is initially empty.

Equality saturation At the start of each growing iteration, we iteratively apply a set of rewrite rules to saturation, adding equivalence relations to $\cong$, and resulting in an e-graph where each e-class represents groups of equivalent terms in the background theory. That is, for any two terms $t_1, t_2 \in G$, $(t_1, t_2) \in \cong$ implies that $t_1 = t_2$ is T-valid, according to the background theory T.

Checking the pool As before, we check the pool of programs against the counterexamples. However, this time, instead of iteratively checking all programs, we instead iterate through the e-classes and then extract and check the canonical form from each e-class.

Growing the e-graph Once we have checked all the e-classes in the pool, we grow the pool. To do this, we add all terms to the e-graph that can be obtained by combining e-classes from G with a single production rule from the grammar. By using e-classes to replace the non-terminal symbols in the production rules instead of e-nodes, we tame the exponential growth of the search space.

The full algorithm is shown in Appendix A.1 - Algorithm 2, along with an algorithm showing standard bottom-up enumeration, Algorithm 1.

Optimizations: Since rewrite rules in e-graphs are non-destructive, i.e., e-nodes for both the left and right hand side of the rule are retained within the e-graph, we implement two optimizations: First, before adding a new program, during the growing phase, we check whether it already exists in a previously explored e-class in E at a rewrite distance of one. This slows the growth of the program pool. Specifically, we use *compactive* rewrites, whereby we do not add new e-nodes generated by rules and instead only use them to merge e-classes already present in the e-graph, this ensures equality saturation does not add unnecessary e-nodes. Additionally, when checking programs we use structural generalization, described in [15]. This method removes the need to check spurious programs in the following way: given a program p of the form (ite c l r) that has been invalidated by a counterexample that evaluated c as true, structural generalization ensures that no other program of the form (ite c l ?a) needs to be checked, where we know the program will be invalidated by the counterexample, regardless of the sub-program represented by the right-hand branch.

4. Synthesis using CounterExample Guided Equivalences

Bottom-up enumeration with e-graphs significantly reduces the number of programs we have to send to the verifier, but the program space still grows impractically large. For our second synthesis approach, we introduce the notion of *counterexample e-graphs*, or CE-graphs, which capture the sets of programs that behave equivalently under a given counterexample. We exploit CE-graphs for two purposes: firstly, they enable us to *prune* the space of candidates using equivalence rules that are conditioned on the counterexample; second, they allow us to *guide the growth* of the program pool, based on the needs of the counterexamples obtained so far. In this section, we first define CE-graphs, and how to construct them, before presenting a CE-graph based synthesis algorithm.

4.1. Defining CE-graphs

Recall that a counterexample $\vec{c}$ is an assignment to all the free variables $\vec{x}$ which caused a previous candidate solution f^* to violate the specification ϕ.

Definition 4.1 (CE-graph). *Given a counterexample $\vec{c}$, a counterexample e-graph is defined as $G_{\vec{c}} = (\mathcal{T} \cong_{\vec{c}})$, where $\mathcal{T}$ is a set of terms in the language L, and the congruence relation $\cong_{\vec{c}}$ is defined such that, for any two terms $t_1 \cong_c t_2$ if and only if $t_1[\vec{x}/\vec{c}] = t_2[\vec{x}/\vec{c}]$ is T-valid under the background theory T.*

Given a CE-graph, $G_{\vec{c}}$, we can now group the e-classes into *invalid* e-classes and *valid* e-classes. An *invalid* e-class is an e-class that contains only programs that fail to satisfy the specification on the given counterexample. A *valid* e-class is an e-class that contains only programs that satisfy the specification for the given counterexample.

4.2. Synthesis with CE-graphs

Synthesis with CE-graphs proceeds through two stages: first, building the CE-graph, which allows us to obtain an infinite set of terms which satisfy the specification under the current counterexample; second, pruning the search space by joining the current CE-graph with the previously obtained CE-graphs. This pruning restricts the set of infinite terms to only the depth required by the counterexamples obtained

so far. This implicit e-graph growth, achieved through counterexample guided rewrite rules, allows us to avoid the explicit growth of the e-graph that is necessary in Section 3.
As an overview, the synthesis proceeds as follows:

Initialization We initialise the algorithm by constructing a base e-graph, G_0, which contains all terminal symbols in $\mathcal{G}$, and an application of each production rule in $\mathcal{G}$, applied to all possible terminal symbol combinations. We do this using bottom-up enumeration, as before, to a depth of 2. We then select a term randomly from the e-graph, and pass this to the verifier. We assume this term will fail verification, and a counterexample will be returned.

Build Given a counterexample $\vec{c}'$, we build $G'_{\vec{c}}$ from G_0. If the valid set of $G'_{\vec{c}}$ is empty, we grow G_0, adding one more layer of programs.

Join Join the new CE-graph, $G'_{\vec{c}}$ to the previous $G_{\vec{c}}$, and set this to be the current CE-graph, i.e., $G_{\vec{c}} = G'_{\vec{c}} \sqcup G_{\vec{c}}$. If this is the first CE-graph we have seen, simply set $G_{\vec{c}} = G'_{\vec{c}}$.

Verify Sample a program from the valid e-classes of $G_{\vec{c}}$, and pass this to the verifier. If it passes verification, return this solution to the user. If it fails, return the counterexample to the **Build** step and repeat.

Pseudocode for the full algorithm is included in Appendix A.1, Algorithm 4. We will now discuss the **build** and **join** phases in more detail.

Building the CE-graph Given a counterexample, $\vec{c}$, we encode the counterexample equivalence relation $\cong_{\vec{c}}$ using a set of rewrites derived from the counterexample; for instance, given the counterexample $[x \mapsto 1, y \mapsto 0]$, we generate the rewrites such as $(> x\ y) \to \top$. For example, when considering a grammar involving equivalence relations, we may generate the following rules:

- For each pair of variables in the assignment we have a set of rewrites for the LT, GT and EQ comparators.
- If the specific constants 0 and 1 are within the assignment, we define rewrite sets for assignments that assign free variables to either.

This allows us to construct the e-graph $G_{\vec{c}}$, as defined above, by applying these rewrite rules to the base e-graph G_0. Note that $\cong_{\vec{c}}$ introduces recursive edges to the e-graph, effectively representing infinite sets of terms that behave equivalently under the counterexample, as shown in Fig. 3.

We test one term from each e-class against the current counterexample, and if $\phi[F/t, \vec{x}/\vec{c}] = \top$, we mark the e-class as $valid$. If there are no valid terms, we grow the base e-graph by one iteration. The pseudocode for building the CE-graph is in Appendix A.1 - Algorithm 3.

Pruning via Join We note that any valid solution to the synthesis problem must be in a valid e-class on every counterexample e-graph. Thus, by performing the join of the current counterexample e-graph with the previous one, we can reduce our search space to only terms in the intersection of the valid e-classes. Further optimisations of this join are possible, in which unnecessary e-nodes are removed, however, our current implementation defines join as e-graph intersection, defined below:

Definition 4.2 (Join of CE-graph). *Given two CE-graphs,* $G_{\vec{c}_1} = (\mathcal{T}_{\vec{c}_1}, \cong_{\vec{c}_1})$ *and* $G_{\vec{c}_2} = (\mathcal{T}_{\vec{c}_2}, \cong_{\vec{c}_2})$*, the join of the two e-graphs,* $G_{\vec{c}_1} \sqcup G_{\vec{c}_2}$ *is an e-graph* $G_{\vec{c}_1 \sqcup \vec{c}_2} = (\mathcal{T}_{\vec{c}_1} \cap \mathcal{T}_{\vec{c}_2}, \cong_{\vec{c}_1 \sqcup \vec{c}_2})$*, where* $(t_1, t_2) \in \cong_{\vec{c}_1 \sqcup \vec{c}_2} \iff (t_1, t_2) \in \cong_{\vec{c}_1} \wedge (t_1, t_2) \in \cong_{\vec{c}_2}$.

At each iteration, we join the most recent CE-graph to the CE-graph representing the current search space, G_{curr}. The result of this join is then set to be the new G_{curr}. We then sample a program from the valid e-class of G_{curr} and send this to verification, repeating the loop.

Note that this join discards equivalence relations that are not valid under both counterexamples, effectively pruning the terms represented in the CE-graph by unrolling recursion not supported by all counterexamples obtained so far.

Running Example: To more clearly describe the algorithm let us follow an example of a possible path this method may take, visualised in Fig. 3. Assume that we are searching for a program that returns the maximum of three values, as in Fig. 1. In line with the motivating example, the algorithm provides a guess from the grammar's terminal symbols, x, to which the oracle returns the counterexample, $\vec{c}_1 = [x \mapsto 1, y \mapsto 2, z \mapsto 0]$. Using G_0, a CE-graph $G_{\vec{c}_1}$ is built rewrite rules derived from the counterexample, including, but not limited to:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} (\geq\ x\,y) & \rightarrow & \bot & (\geq\ y\,z) & \rightarrow & \top & (\geq\ x\,z) & \rightarrow & \bot \\ (\times\ \ x\ ?a) & \rightarrow & ?a & (\times\, z\ ?a) & \rightarrow & 0 & x & \rightarrow & 1 \end{array}$$

We use $G_{\vec{c}_1}$ to generate a new candidate which satisfies the specification for $\vec{c}_1$, y. The counterexample $\vec{c}_2 = [x \mapsto 2, y \mapsto 1, z \mapsto 0]$ is returned and used to create $G_{\vec{c}_2}$. We then generate a new CE-graph, $G_{curr} = G_{\vec{c}_1} \sqcup G_{\vec{c}_2}$, representing the subset of $\mathcal{L}^{\mathcal{G}}$ that satisfies both $\vec{c}_1$ and $\vec{c}_2$. The remaining satisfying terms in G_{curr} are made up of those that compare x and y, returning the largest of the two, for instance `(ite (>= x y) x y)`. Providing this as a candidate yields $\vec{c}_3 = [x \mapsto 0, y \mapsto 1, z \mapsto 2]$, used to generate $G_{curr} = G_{curr} \sqcup G_{\vec{c}_3}$. The satisfying terms in G_{curr} now only contains terms that compare the three variables and returns the largest of the three, the set of all satisfying terms.

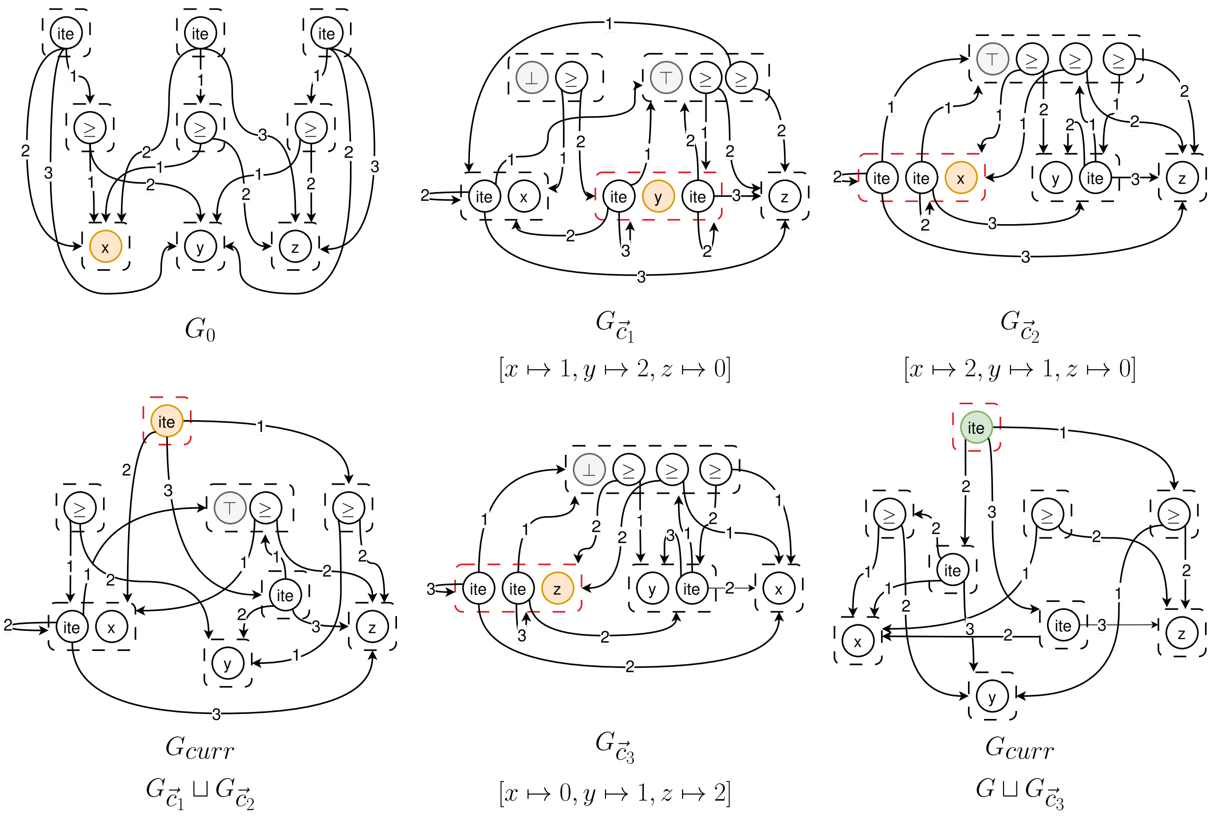
G_0
$G_{\vec{c}_1}$
$[x \mapsto 1, y \mapsto 2, z \mapsto 0]$
$G_{\vec{c}_2}$
$[x \mapsto 2, y \mapsto 1, z \mapsto 0]$
G_{curr}
$G_{\vec{c}_1} \sqcup G_{\vec{c}_2}$
$G_{\vec{c}_3}$
$[x \mapsto 0, y \mapsto 1, z \mapsto 2]$
G_{curr}
$G \sqcup G_{\vec{c}_3}$
ite
x
y
z

Figure 3: An example of synthesising MAX3. E-classes are rectangles with broken edges, with red e-classes representing the class of terms that satisfy the counterexample context. E-nodes are captured by solid circles, with orange e-nodes representing the e-node used to extract a candidate program and green e-nodes representing terms that satisfy the whole specification.

5. Evaluation

We implement preliminary versions of both approaches. We use the rust EGG crate [13] as our e-graph data structure, and carry out validation using rust bindings to Z3 [16]. The e-graph in the naive enumerator has 49 symbolically equivalent rewrites, the CE-graphs have an additional 44 conditional rewrites. Both these sets of rewrites are incomplete. The current CE-graph method uses a total e-graph intersection as the join, specifically the e-graph intersection defined in EGG. We compare against cvc5 [17], which implements synthesis via CEGIS, using various highly tuned enumerative approaches for searching the grammar [15]. We use a set of benchmarks from the syntax-guided synthesis competition [18]. Specifically, for this preliminary implementation, we consider benchmarks in linear integer arithmetic, with a single invocation of each synthesis function and without using defined functions within the grammar. We use a timeout of 1 minute.

Performance: The results in Table 1 show that both e-graph based synthesis methods reduce the number of candidate programs sent to the verifier substantially, which is to be expected. The naive bottom-up enumerator solves fewer benchmarks than cvc5, but Fig. 4 shows the performance is comparable with cvc5 on the ones it does solve, which is surprising given the highly preliminary nature of our implementation. From Table 1 we see that the CE-graph approach does not outperform the naive bottom-up method, which we hypothesize is due to an inefficient (for our purposes) implementation of join, as well as an incomplete conditional rewrite rule set.

Synthesiser	Success	Timeout	Error	Candidates checked	Proportion of time in join
Naive bottom-up e-graph	78	70	-	5.42	-
ce-graphs	54	85	9	4.99	36%
CVC5	87	61	-	86.35	-

Table 1
Comparison of synthesis results across different 148 benchmarks with a 60 second timeout. Errors in the conditional e-graphs were due to implementation issues and memory-out of bounds errors.

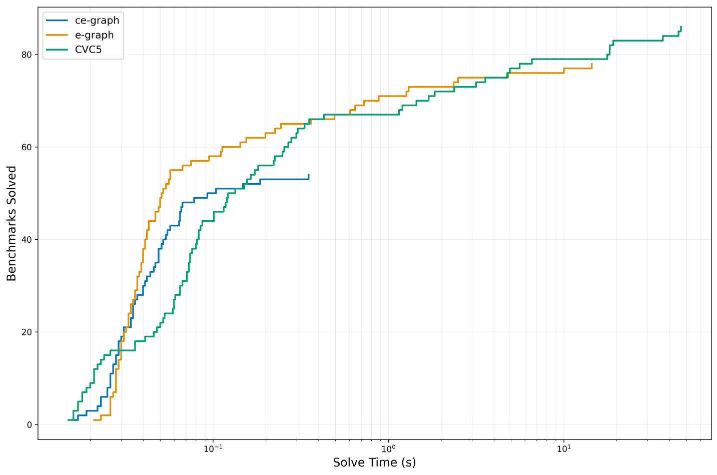

Figure 4: Cumulative number of benchmarks solved by the three synthesisers as a function of time.

5.1. Limitations/Future Work:

Completeness of rewrite rules: The set of rewrite rules we provide to both approaches is incomplete (i.e., $(t_1, t_2) \in \cong_{\vec{c}} \Rightarrow t_1[x/\vec{c}] = t_2[x/\vec{c}]$, but $t_1[x/\vec{c}] = t_2[x/\vec{c}] \not\Rightarrow (t_1, t_2) \in \cong_{\vec{c}}$). For the naive enumeration, this will result in checking more programs than necessary. For the CE-graph synthesis, incomplete rewrite rules will result in more occasions where the valid set does not contain the valid solution and we must explicitly grow the base e-graph. The rewrite rules are also written by hand. One possible direction to explore is executing the terms on each counterexample, and using these results to infer rewrite rules.

Generality: The current CE-graph synthesis formulation is biased towards solving benchmarks with solutions that rely on comparators captured in our conditional rewrites. For other benchmarks, relying on predefined rewrite rules may always be significantly less effective.

Join Efficiency: As mentioned in Section 4.2, there is significant inefficiency in the CE-graph method due to the inefficient implementation of join. Namely, that we calculate the join of all e-classes, including terms that are not within the valid set. A more efficient CE-graph join, inspired by ideas from *colored e-graphs* [19], should improve the performance of the synthesis approach.

Counterexample Ordering: As with many synthesis approaches, the order that we encounter counterexamples in can result in big variations in solving time, as this produces a different sequence of CE-graphs. Determining the best counterexample is an open question in synthesis [20], but this question appears to become even more critical when using CE-graph based synthesis.

Soundness and Completeness Our approach is sound, in that any result that is returned is guaranteed to be correct, provided the verifier is correct. It is not complete, and cannot be complete as the synthesis problem in general is undecidable. However, if the grammar is restricted to one that expresses a finite number of programs, it should be possible to provide guarantees that a program will be found.

6. Related Work

There are many approaches to pruning the search space in program synthesis, including using deductive reasoning [21, 22], using divide and conquer to partition the search space [12], and all varieties of machine-learning or probabilistic guided synthesis [23, 24]. We focus our related work discussion on approaches using observational equivalence.

Observational equivalence has been leveraged for programming by example by using finite tree automata (FTA), in which each node contains all observationally equivalent programs with hyper-edges representing productions rule applied to child nodes [25, 26, 27]. Our naive e-graph algorithm can be viewed as a generalisation of uses of observational equivalence in FTAs.

In SyMetric [28], observational equivalence was relaxed to *observational similarity*, used as an approximate proxy of equivalence based on input-output examples.

Equivalence abstraction in FTAs was introduced in the tool Blaze [29]. This algorithm uses abstract semantics to generate abstract FTAs (AFTAs). For each failing candidate a proof of failure is generated to refine the abstract semantics. The AFTA is then refined by splitting nodes into refined abstract equivalences and a new candidate is generated. The CE-graph synthesis method develops on the ideas in Blaze. Each CE-graphs represents some level of abstract equivalence relation, with the join operation carrying out the refinement.

Granularity of equivalence relations Extending the capabilities of e-graphs to represent multiple levels of equivalence relations, as describe in our methods, has been explored. As eluded to in Section 5 colored e-graphs [19] permit a hierarchy of equivalences through hierarchical *colored* e-classes. Colored e-graphs provides an efficient implementation of e-graphs that permit capturing equivalences of e-classes under assumptions. Specifically, it provides a method that shares nodes instead of requiring the generation of individual e-graphs for each assumption. This provides an efficient mechanism to find terms that are equivalent under multiple assumptions. Additionally, work on *contextual equalities* and contextual equality saturation [30] attempts to capture equality that is only true in sub-graphs. These two methods closely align with the CE-graphs; the main exception is the introduction of edges being between nodes and conditioned e-classes, in colored e-graphs, this would be synonymous with having "colored edges" between e-nodes and colored e-classes. It is this feature of CE-graphs that permits their compact representation of languages when the grammar's free variables are substituted for concrete values.

7. Conclusions

We have presented an approach to using e-graphs to efficiently represent and enumerate the space of possible programs for general program synthesis. To the best of our knowledge, this is the first approach to use e-graphs in this way. There are a number of limitations to our approach, yet, despite this, it performs comparably to the state of the art on a small set of benchmarks. Given the intertwined nature of synthesis, SMT solving, and e-graphs, we would welcome discussion with the community on the directions this work should take next.

Declaration on Generative AI

The authors used Grammarly for grammar and spell-checking, and Claude to assist with the implementation of the proposed algorithms. After using these tools, the authors reviewed and edited content as needed and take full responsibility for the publication's content.

References

[1] J. Kim, Program synthesis is σ_3^0-complete, arXiv preprint arXiv:2405.16997 (2024).
[2] S. Gulwani, O. Polozov, R. Singh, et al., Program synthesis, Foundations and Trends® in Programming Languages 4 (2017) 1–119.
[3] A. Albarghouthi, S. Gulwani, Z. Kincaid, Recursive program synthesis, in: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25, Springer, 2013, pp. 934–950.
[4] L. M. de Moura, N. S. Bjørner, Efficient e-matching for SMT solvers, in: CADE, volume 4603 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 183–198.
[5] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, Z. Tatlock, Automatically improving accuracy for floating point expressions, in: PLDI, ACM, 2015, pp. 1–11.
[6] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, N. Polikarpova, babble: Learning better abstractions with e-graphs and anti-unification, Proc. ACM Program. Lang. 7 (2023) 396–424.
[7] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, Z. Tatlock, Rewrite rule inference using equality saturation, Proc. ACM Program. Lang. 5 (2021) 1–28.
[8] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, V. A. Saraswat, Combinatorial sketching for finite programs, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM Press, 2006, pp. 404–415.
[9] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: Dependable Software Systems Engineering, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, IOS Press, 2015, pp. 1–25.
[10] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: Dependable Software Systems Engineering, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, IOS Press, 2015, pp. 1–25.
[11] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, C. Tinelli, cvc4sy: Smart and fast term enumeration for syntax-guided synthesis, in: CAV (2), volume 11562 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 74–83.
[12] R. Alur, A. Radhakrishna, A. Udupa, Scaling enumerative program synthesis via divide and conquer, in: TACAS (1), volume 10205 of *Lecture Notes in Computer Science*, 2017, pp. 319–336.
[13] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, P. Panchekha, Egg: Fast and extensible equality saturation, Proceedings of the ACM on Programming Languages 5 (2021) 1–29.

[14] D. Suciu, Y. R. Wang, Y. Zhang, Semantic foundations of equality saturation, arXiv preprint arXiv:2501.02413 (2025).

[15] A. Reynolds, H. Barbosa, A. Nötzli, C. W. Barrett, C. Tinelli, cvc4sy: Smart and fast term enumeration for syntax-guided synthesis, in: CAV (2), volume 11562 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 74–83.

[16] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS, volume 4963 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 337–340.

[17] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al., cvc5: A versatile and industrial-strength smt solver, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2022, pp. 415–442.

[18] Sygus competition, https://sygus.org/, 2019. Accessed: 2021-05-19.

[19] E. Singher, I. Shachar, Easter egg: Equality reasoning based on e-graphs with multiple assumptions, in: FMCAD, TU Wien Academic Press, 2024, pp. 70–83.

[20] S. Jha, S. A. Seshia, Are there good mistakes? A theoretical analysis of CEGIS, in: SYNT, volume 157 of *EPTCS*, 2014, pp. 84–99.

[21] Y. Feng, R. Martins, O. Bastani, I. Dillig, Program synthesis using conflict-driven learning, in: PLDI, ACM, 2018, pp. 420–435.

[22] A. Abate, C. David, P. Kesseli, D. Kroening, E. Polgreen, Counterexample guided inductive synthesis modulo theories, in: International Conference on Computer Aided Verification, Springer, 2018, pp. 270–288.

[23] S. Barke, H. Peleg, N. Polikarpova, Just-in-time learning for bottom-up enumerative synthesis, Proceedings of the ACM on Programming Languages 4 (2020) 1–29.

[24] W. Lee, K. Heo, R. Alur, M. Naik, Accelerating search-based program synthesis using learned probabilistic models, in: PLDI, ACM, 2018, pp. 436–449.

[25] X. Wang, I. Dillig, R. Singh, Synthesis of data completion scripts using finite tree automata, Proceedings of the ACM on Programming Languages 1 (2017) 1–26.

[26] H. Peleg, R. Gabay, S. Itzhaky, E. Yahav, Programming with a read-eval-synth loop, Proceedings of the ACM on Programming Languages 4 (2020) 1–30.

[27] X. Li, X. Zhou, R. Dong, Y. Zhang, X. Wang, Efficient bottom-up synthesis for programs with local variables, Proceedings of the ACM on Programming Languages 8 (2024) 1540–1568.

[28] J. Feser, I. Dillig, A. Solar-Lezama, Metric program synthesis, arXiv preprint arXiv:2206.06164 (2022).

[29] X. Wang, I. Dillig, R. Singh, Program synthesis using abstraction refinement, Proceedings of the ACM on Programming Languages 2 (2017) 1–30.

[30] T. Hou, S. Laddad, J. M. Hellerstein, Towards relational contextual equality saturation, arXiv preprint arXiv:2507.11897 (2024).

A. Appendix

A.1. Algorithms

Algorithm 1 Bottom-Up Enumerator

```
procedure ENUMERATE($\mathcal{G}, \phi, I$)
    $ProgPool \leftarrow \emptyset$
    $NewProgs \leftarrow \mathcal{G}.\Sigma$                    ▷ Add all terminal symbols to the pool of programs
    while 1 do
        $ProgPool \leftarrow ProgPool \cup NewProgs$
        for $p \in NewProgs$ do                                ▷ Check against counterexamples
            if $\forall \vec{c} \in I.\phi(p, \vec{c})$ then
                return $p$
            end if
        end for
        $NewProgs \leftarrow \emptyset$
        for $(r_l, r_r) \in R$ do                              ▷ iterate through production rules
            $operands \leftarrow []$                           ▷ Build list of lists of possible operands
            for $s \in r_r$ do                                 ▷ iterate through symbols in RHS of rule
                if $s \in \Omega$ then                         ▷ If this is a nonterminal symbol
                    $ops \leftarrow$ FILTER($ProgPool, s$)     ▷ Get all of the same type as $s$
                    $operands \leftarrow [operands, ops]$
                end if
                for $args \in$ CARTESIANPRODUCT($operands$) do
                    $NewProgs \leftarrow NewProgs \cup$ BUILDPROGRAM($r_r, args$)
                end for
            end for
        end for
    end while
end procedure
```

Algorithm 2 Naive Bottom-Up Enumeration using E-graph

```
1: procedure ENUMERATE($\mathcal{G}, \phi, I$)
2:     $E \leftarrow$ BUILDINITIALGRAPH($\mathcal{G}$)    ▷ Initial e-graph
3:     while 1 do
4:         for $e \in$ GETECLASSES($G$) do    ▷ Check for each new e-class against counterexamples
5:             $p \leftarrow$ EXTRACT($e$)    ▷ Extract a canonical program from e-class
6:             if $\forall \vec{c} \in I.\phi(p, \vec{c})$ then
7:                 return $p$
8:             end if
9:         end for
10:        for $(r_l, r_r) \in R$ do    ▷ iterate through production rules
11:            $operands \leftarrow []$    ▷ Build list of lists of possible operands
12:            for $s \in r_r$ do    ▷ iterate through symbols in RHS of rule
13:                if $s \in \Omega$ then    ▷ If this is a nonterminal symbol
14:                    $ops \leftarrow$ FILTERECLASSES($E, s$)    ▷ Get all e-classes annotated with $s$
15:                    $operands \leftarrow [operands, ops]$
16:                end if
17:                for $args \in$ CARTESIANPRODUCT($operands$) do
18:                    $p \leftarrow$ BUILDPROGRAM($r_r, args$)
19:                    if !REWRITEONCE($p$) $\in G$ then ▷ Check if equivalent program exists in e-graph by a single rewrite
20:                        $G \leftarrow$ ADD($G, p$)
21:                    end if
22:                end for
23:            end for
24:        end for
25:        $G \leftarrow$ EQUALITYSATURATION($G$)
26:    end while
27: end procedure
```

Algorithm 3 Building a Conditioned E-Graph

```
1: procedure BUILDCEGRAPH($G_0, \phi, \vec{c}$)
2:     $valid \leftarrow \perp$
3:     while not $valid$ do
4:         $R \leftarrow$ GENERATECONREWRITES($\vec{c}$)    ▷ Generate rewrites conditioned on $\vec{c}$
5:         $G_{\vec{c}} \leftarrow$ RUN($G_0, R$)    ▷ Run equality saturation with conditional rewrites
6:         $valid \leftarrow$ GETVALID($G_{\vec{c}}$)
7:         if $valid$ then
8:             return $G_{\vec{c}}$
9:         end if
10:        $G_0 \leftarrow$ ENUMERATE($G_0$)
11:    end while
12: end procedure
```

Algorithm 4 CE-Graph Synthesis

```
1: procedure CE-CEGIS($\mathcal{G}, \phi$)
2:     $I \leftarrow \emptyset$                                         ▷ Set of counterexamples
3:     $G_0 \leftarrow \textsc{InitialEGraph}(\mathcal{G})$           ▷ Initial e-graph built from the grammar
4:     $G_{curr} \leftarrow \emptyset$
5:     Pick $f^* \in \mathcal{G}.\Sigma$                                ▷ Choose an initial terminal symbol
6:     while true do
7:         if $\exists x \,.\, \neg\phi(f^*, x)$ then
8:             $I \leftarrow I \cup \{x\}$
9:             $G_{\vec{c}} \leftarrow \textsc{BuildCEGraph}(G_0, x)$
10:            $G_{curr} \leftarrow G_{curr} \sqcup G_{\vec{c}}$        ▷ Join $G_{curr}$ with new conditioned e-graph
11:            Pick $f^* \in \{t \in G_{curr} \mid \phi(t, I)\}$
12:        else
13:            return $f^*$
14:        end if
15:    end while
16: end procedure
```