RESEARCH-ARTICLE

Scaling Abstraction Refinement for Program Analyses in Datalog using Graph Neural Networks

ZHENYU YAN, Peking University, Beijing, China

XIN ZHANG, Peking University, Beijing, China

PENG DI, Ant group, Hangzhou, Zhejiang, China

Published: 08 October 2024

Citation in BibTeX format

Scaling Abstraction Refinement for Program Analyses in Datalog using Graph Neural Networks

ZHENYU YAN, Peking University, China
XIN ZHANG[*], Peking University, China
PENG DI, Ant Group, China

Counterexample-guided abstraction refinement (CEGAR) is a popular approach for automatically selecting abstractions with high precision and low time costs. Existing works cast abstraction refinements as constraint-solving problems. Due to the complexity of these problems, they cannot be scaled to large programs or complex analyses. We propose a novel approach that applies graph neural networks to improve the scalability of CEGAR for Datalog-based program analyses. By constructing graphs directly from the Datalog solver's calculations, our method then uses a neural network to score abstraction parameters based on the information in these graphs. Then we reform the constraint problems such that the constraint solver ignores parameters with low scores. This in turn reduces the solution space and the size of the constraint problems. Since our graphs are directly constructed from Datalog computation without human effort, our approach can be applied to a broad range of parametric static analyses implemented in Datalog. We evaluate our approach on a pointer analysis and a typestate analysis and our approach can answer 2.83× and 1.5× as many queries as the baseline approach on large programs for the pointer analysis and the typestate analysis, respectively.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: program analysis, graph neural networks, abstraction refinement

ACM Reference Format:
Zhenyu Yan, Xin Zhang, and Peng Di. 2024. Scaling Abstraction Refinement for Program Analyses in Datalog using Graph Neural Networks. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 325 (October 2024), 29 pages. https://doi.org/10.1145/3689765

1 Introduction

Selecting appropriate abstractions for program analysis is a hard and critical problem. Coarse abstractions tend to produce many false alarms, while fine abstractions may fail to scale to large programs. However, existing works [Li et al. 2020; Smaragdakis et al. 2014; Yao et al. 2021] have shown not all improvements in precision will increase time costs significantly. It is possible to resolve more queries (i.e., assertions) without losing scalability. It is crucial to select abstractions carefully so they balance precision and scalability.

Counterexample-guided abstraction refinement (CEGAR) [Clarke et al. 2000] is a class of query-driven approaches for selecting abstractions wisely. CEGAR usually starts with a coarse abstraction

[*]Corresponding author.

Authors' Contact Information: Zhenyu Yan, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, China, zhenyuyan@stu.pku.edu.cn; Xin Zhang, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China, xin@pku.edu.cn; Peng Di, Ant Group, Hangzhou, Zhejiang, China, dipeng.dp@antgroup.com.

ACM 2475-1421/2024/10-ART325
https://doi.org/10.1145/3689765

to resolve given queries. If it fails to resolve any query, it will refine the abstraction by analyzing counterexamples that hinder resolving failed queries. By refining certain abstraction parameters, CEGAR usually leads to abstractions that are just fine enough to resolve the queries. Problems of analyzing counterexamples and then refining abstractions are typically cast as constraint-solving problems [McMillan 2003; Zhang et al. 2014]. While constraint solvers have made great strides in recent years, it is still hard to scale them to problems that are produced by analyzing large programs due to the complexities of constraint problems. Furthermore, CEGAR-based approach sometimes may refine abstractions unnecessarily, as the constraint problems only encode why the current abstraction does not work but cannot predict accurately how unseen abstractions would work.

To solve this problem, our key idea is that we can scale abstraction refinement using deep learning. Deep learning has demonstrated its power to solve complex problems in many applications [Jumper et al. 2021; Silver et al. 2016], including constraint solving [Lederman 2021; Zhang et al. 2020]. In our case, our goal is not to simply offer a more scalable alternative to constraint solving but to be more effective in finding good abstractions. However, to apply deep learning effectively, we face three challenges. First, how do we apply deep learning to a large range of analyses without requiring heavy engineering from analysis designers? Second, as mentioned above, simply solving constraint problems may refine abstractions unnecessarily. How can we select abstraction refinements that are truly useful for improving analysis precision? Third, in practice, we find that while learning-based approaches are effective in pruning parts of abstraction parameters that do not need to be refined, it is hard for them to identify exactly the parts that need to be refined. This is because multiple parameters can be helpful for resolving a given query while refining just part of them may suffice. However, since all these parameters are helpful, learning-based approaches tend to refine them all, resulting in overly precise abstractions with higher time and space costs.

To address the first challenge, we target program analyses expressed in Datalog and propose a general framework that translates abstraction refinement problems into classification problems using neural networks. Datalog is a logic programming language that has been used widely in recent years to express various program analyses [Bravenboer and Smaragdakis 2009; Madsen et al. 2016; Naik 2011]. Its declarative nature and powerful runtime allow analysis designers to focus on analysis specifications without worrying much about implementation details. Furthermore, the result of a Datalog-based analysis automatically forms a hypergraph. We propose an approach to transform the hypergraph into a graph. Our graph neural network (GNN) then takes the graph and filters out unhelpful parameters. The input features of our approach are directly extracted from the derivation of the target analysis. This is different from existing learning-based abstractions selection works [Jeon et al. 2019, 2020]. For example, Jeon et al. [2019] relies on hand-crafted syntactic features and Jeon et al. [2020] relies on graphs designed by experts and a predefined feature description language. Therefore, though their approaches achieve good performances in pointer analysis, it can be hard to apply their approaches to other analyses. To show our generality, we run our approach on a pointer analysis and a typestate analysis and the result shows that our approach generalizes well among those two analyses.

To tackle the second challenge that constraint-solving approaches may refine unhelpful parameters, we design an algorithm to filter out unhelpful parameters in the training data. Given refinement traces of constraint-solving approaches, our algorithm iteratively checks whether each refined parameter is truly useful for resolving queries. Then, our algorithm looks for helpful parameters that are not selected in the given traces. By learning from processed traces, our neural network can ignore most unhelpful parameters and focus more on helpful parameters.

For the third challenge that learning-based approaches cannot accurately select a proper set of parameters, we use our neural network to filter out unhelpful parameters and then apply a constraint solver to exactly identify the refinement. In this way, we combine the strengths of the

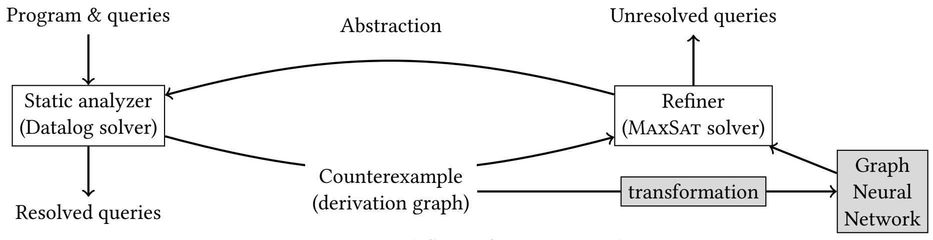


Fig. 1. Workflow of our approach.

neural approach and the traditional constraint-solving approach, overcoming the neural network's tendency to select too many parameters and the constraint-solving approach's potential for selecting unhelpful parameters. Since the neural network only affects the abstraction selection, as long as all abstractions from the family are all sound (which is usually the case), soundness will be preserved.

We have instantiated our approach on a CEGAR framework for Datalog-based analyses proposed by Zhang et al. [2014]. In this framework, the constraint solver is a partial weighted *maximum satisfiability* (MaxSat) solver [Li and Manyà 2021]. To demonstrate the effectiveness of our approach, we have applied it to a context-sensitive, flow-insensitive pointer analysis and a context/flow-sensitive typestate analysis. The experiment result shows that given the same timeout threshold, our approach can answer 2.83× and 1.5× as many queries as the baseline approach on large programs for the pointer analysis and the typestate analysis, respectively.

In summary, our work makes the following contributions:

- We have proposed a general framework to scale abstraction refinements for Datalog-based program analyses by leveraging graph neural networks.
- We have designed an algorithm to improve training data quality by identifying effective refinements selected by existing traces and other unselected alternative effective refinements.
- We have implemented our framework and evaluated it using two analyses. The results show that our approach effectively increases the number of resolved queries on large programs where the baseline approach fails to scale.

2 Overview

Figure 1 presents the high-level workflow of our work, emphasizing the role of our graph neural network. Our approach formulates the calculation (i.e., derivation graph) of a parametric static analyzer as MaxSat constraints, incorporating costs of refining abstraction parameters and the requirement to resolve queries. The graph neural network filters abstraction parameters to prune the MaxSat problem, and the MaxSat solver determines the next abstraction to try for the static analyzer by solving the MaxSat problem. The goal of our approach is to find one of the (approximately) cheapest abstractions among abstractions that can resolve the most queries (see § 3.3).

We illustrate our approach by applying a pointer analysis to a small Java program shown in Figure 2a. This program contains two classes implementing the interface `Dog`, named `Corgi` and `Teddy`. The program creates an object for each class and passes them through identity methods. The query is to determine what objects `corgi3` may point to.

A pointer analysis typically calculates the set of objects that each pointer may point to throughout the program's execution. This is achieved by propagating points-to information through assignments and argument passing. In this case, `dog` in `id1` will be assigned with either `corgi1` or `teddy1`. Since `corgi1` may point to a `Corgi` object and `teddy1` may point to a `Teddy` object, the type of `dog` in

```
1   public interface Dog { void bark(); }
2   public class Corgi implements Dog { /* ... */ }
3   public class Teddy implements Dog { /* ... */ }
4   public Dog id1(Dog dog) { return dog; }
5   public Dog id2(Dog dog) { return dog; }
6   public static void main(String args[]) {
7       Dog corgi1 = new Corgi();
8       Dog teddy1 = new Teddy();
9       Dog corgi2 = id1(corgi1);
10      Dog teddy2 = id1(teddy1);
11      Dog corgi3 = id2(corgi2);
12      corgi3.bark(); // query q
13  }
```

(a) Example program.

```
MayPointTo(u, obj) :- MayPointTo(v, obj), Assignment(u, v).
Assignment(v, u) :- AssignmentCI(v, u, i), CI(i).
Assignment(v, u) :- AssignmentCS(v, u, i), CS(i).
...
```

(b) Datalog rules of a parametric pointer analysis.

```
MayPointTo(0:main:corgi1, Corgi).
MayPointTo(0:main:teddy1, Teddy).
AssignmentCI( 0:id1:dog, 0:main:corgi1, 9:id1).
AssignmentCS( 9:id1:dog, 0:main:corgi1, 9:id1).
AssignmentCI(0:main:corgi2, 0:id1:dog, 9:id1).
AssignmentCS(0:main:corgi2, 9:id1:dog, 9:id1).
AssignmentCI( 0:id1:dog, 0:main:teddy1, 10:id1).
AssignmentCS(10:id1:dog, 0:main:teddy1, 10:id1).
...
```

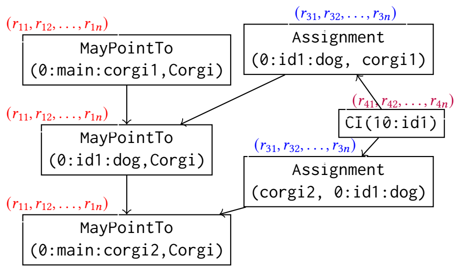
(c) Input tuples extracted from the program.

```
CI(9:id1). CI(10:id1). CI(11:id2).
```

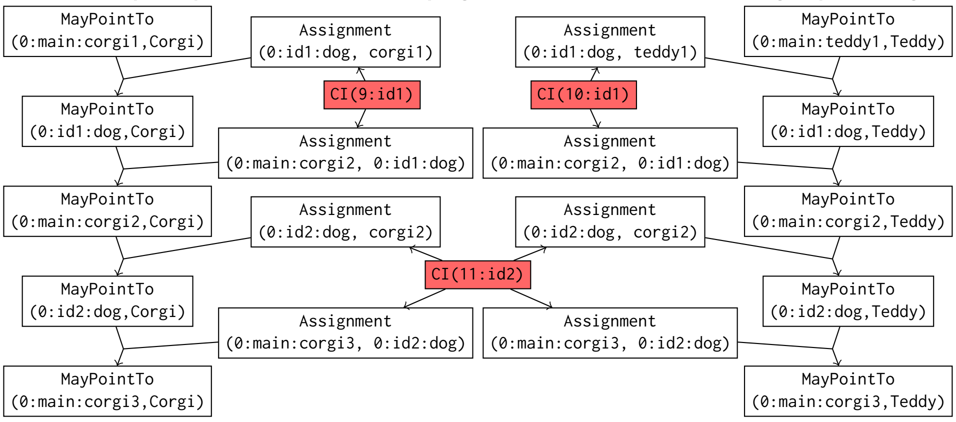
(d) Analysis parameters of the program.

```
Assignment(0:id1:dog, 0:main:corgi1).
MayPointTo(0:id1:dog, Corgi).
Assignment(0:id1:dog, 0:main:teddy1).
MayPointTo(0:id1:dog, Teddy).
Assignment(0:main:corgi2, 0:id1:dog).
MayPointTo(0:main:corgi2, Corgi).
MayPointTo(0:main:corgi2, Teddy).
Assignment(0:id2:dog, 0:main:corgi2).
MayPointTo(0:id2:dog, Corgi).
MayPointTo(0:id2:dog, Teddy).
Assignment(0:main:corgi3, 0:id2:dog).
MayPointTo(0:main:corgi3, Corgi).
MayPointTo(0:main:corgi3, Teddy).
```

(e) Tuples derived under context-insensitivity.


$(r_{11}, r_{12}, \ldots, r_{1n})$
MayPointTo (0:main:corgi1,Corgi)
$(r_{31}, r_{32}, \ldots, r_{3n})$
Assignment (0:id1:dog, corgi1)
$(r_{41}, r_{42}, \ldots, r_{4n})$
CI(10:id1)
$(r_{11}, r_{12}, \ldots, r_{1n})$
MayPointTo (0:id1:dog,Corgi)
$(r_{31}, r_{32}, \ldots, r_{3n})$
Assignment (corgi2, 0:id1:dog)
$(r_{11}, r_{12}, \ldots, r_{1n})$
MayPointTo (0:main:corgi2,Corgi)

(f) Embedding of part of Figure 2g.


MayPointTo (0:main:corgi1,Corgi)
Assignment (0:id1:dog, corgi1)
Assignment (0:id1:dog, teddy1)
MayPointTo (0:main:teddy1,Teddy)
CI(9:id1)
CI(10:id1)
MayPointTo (0:id1:dog,Corgi)
Assignment (0:main:corgi2, 0:id1:dog)
Assignment (0:main:corgi2, 0:id1:dog)
MayPointTo (0:id1:dog,Teddy)
MayPointTo (0:main:corgi2,Corgi)
Assignment (0:id2:dog, corgi2)
Assignment (0:id2:dog, corgi2)
MayPointTo (0:main:corgi2,Teddy)
CI(11:id2)
MayPointTo (0:id2:dog,Corgi)
Assignment (0:main:corgi3, 0:id2:dog)
Assignment (0:main:corgi3, 0:id2:dog)
MayPointTo (0:id2:dog,Teddy)
MayPointTo (0:main:corgi3,Corgi)
MayPointTo (0:main:corgi3,Teddy)

(g) Part of the derivation graph of the example in Figure 2a.

Fig. 2. A motivating example.

id1 cannot be determined. This is how context-insensitive pointer analysis works. To make the analysis more precise, a context-sensitive pointer analysis is needed. One well-known context-sensitive pointer analysis is k-CFA, it clones method bodies based on call sites [Shivers 1991] (equivalent to replacing id1 at Line 9 with a functionally equivalent function named 9:id1), so it can distinguish the dog variable in id1 called by Line 9 (denoted as 9:id1:dog) from the one called by Line 10 (denoted as 10:id1:dog). Therefore, pointer analysis can make sure that 9:id1:dog must only point to the Corgi object. Analogously, corgi2 and corgi3 in the main function must only point to the Corgi object. Finally, the analysis concludes that only the bark method of class Corgi will be invoked at Line 12. Though context sensitivity improves the precision of pointer analysis, it increases the time and space cost to run a pointer analysis. So for real-world programs, people only pick certain parts of program components (such as call sites in k-CFA) to be context-sensitive to trade off time and precision. In this example, analyzing the `id1` at Line 9 *or* Line 10 in a context-sensitive way can resolve the query, because it distinguishes the two `Dog` objects. Thus, refining `id2` at Line 11 is unnecessary.

We formalize the aforementioned *may-point-to* flows between variables in the Datalog rules presented in Figure 2b, where k of k-CFA can be either `0` or `1` for each call site (stands for context insensitivity and context sensitivity, respectively). *May-point-to* flow is transformed into tuples represented in Figure 2c and 2d. Certain rules related to context calculation, argument passing, and method return values are omitted for clarity, and their results are denoted using `AssignmentCI` and `AssignmentCS`. Variables are represented in the format `context:method:variable` to differentiate variables with the same name in different contexts or methods. The first rule in Figure 2b propagates the may-point-to relation along dataflows in the program. It means that if a variable `v` may point to an object `obj` and there is a value flow (assignment, argument passing, and so on) from `v` to `u`, `u` may also point to the object `obj`. The subsequent rules extract dataflows from argument passing and return values of functions. `AssignmentCI` and `AssignmentCS` are calculated beforehand, representing dataflows under context insensitivity and sensitivity, respectively. If a call site `i` is associated with `CI`, related `AssignmentCI` tuples will be appended to the set of `Assignment` tuples, so that the function call at `i` will be analyzed context-insensitively. Therefore, this analysis is parameterized in a way that each call site is associated with either a `CI` or `CS` tuple, representing context insensitivity and sensitivity. Those `CI` and `CS` tuples are called *(abstract) parameter tuples*, which stands for the parameters of the analysis. Given a set of parameter tuples, the analyzer will compute all tuples that can be derived from other tuples. For the parameter set shown in Figure 2d, the tuples derived are shown in Figure 2g (`AssignmentCI` and `AssignmentCS` are ignored for simplicity). The derivation forms a hypergraph where every hyperedge may have multiple heads. Every hyperedge stands for a derivation relation, it starts from the condition(s) and points to the conclusion.

To avoid analyzing all call sites context-sensitively, which is impractical for most real-world programs, our method effectively selects context sensitivity based on iterative counterexample-guided abstraction refinement: We start with the coarsest abstraction that analyzes all method calls context-insensitively, the Datalog solver computes tuples derived from input tuples and rules. Then, in each iteration, the MaxSat solver will select a new abstraction according to current counterexamples (paths to queries in the derivation graph). We then use the new abstraction to analyze the program again. This process stops until all queries are resolved, the abstraction cannot be further refined, or the time threshold is met. Since new abstractions are more precise than old abstractions, the selection of new abstractions is named abstraction refinement. Figure 2g shows the derivation of both `MayPointTo(0:id1:dog,Corgi)` and `MayPointTo(0:id1:dog,Teddy)`. This indicates that `dog` in `id1` may point to either a `Corgi` or a `Teddy` object. Consequently, the query `q` is irresolved under context-insensitivity.

Therefore, to analyze the program more precisely, we refine our abstraction by replacing some `CI` tuples with their corresponding `CS` tuples. Because paths in a derivation graph represent the derivation relation between tuples, to eliminate a query, we need to cut all paths to the query in the derivation graphs by refining certain parameter tuples. For example, if `CI(10:id1)` is removed, the path to `Assignment(0:id1:dog, corgi1)` will be cut off, and therefore `MayPointTo(0:main:corgi3, Corgi)` will also be eliminated. Constraint solvers are usually used to find the smallest set of parameter tuples (which is `CI` here) that cuts off the paths of as many queries as possible. According to the derivation graph shown in Figure 2g, the precision of `q` may be affected by all the parameter tuples. However, this strategy does not guarantee the elimination of the query. Because we do not simply remove the parameter tuple but replace it with a more precise one. The new (refined) parameter tuple may generate the query in another path. For example, although removing `CI(11:id2)` also cuts off the path to `MayPointTo(0:main:crogi3,Teddy)`, adding `CS(11:id2)` will create another path to it. As a result, refining it will not resolve the query. But by only looking at the derivation graph, as previous works [Zhang et al. 2014] do, the MaxSat solver cannot figure out unhelpful parameter tuples. Therefore, even though the MaxSat solver can return one of the smallest sets of `CI` that cut off all certain paths, that set may contain unhelpful parts, which makes abstraction refinement take more iterations. Furthermore, due to unnecessary refinements, the analyzer may take more time and space to analyze programs and the analysis may even become unscalable.

Refinement Space Pruning. Our approach utilizes a graph neural network to reduce the size of MaxSat problems and guide the MaxSat solver in selecting more effective parameter tuples. Unlike the aforementioned learning-based approaches, our approach does not rely on manual feature engineering. Instead, our graph neural network takes the automatically extracted derivation graph as input. It embeds nodes into real vectors based on their relations and neighborhoods (which represent tuples that derive it and tuples it derives). Then, our approach uses a fully connected neural network to transform these vectors into real values representing the usefulness of parameter tuples. This enables us to prune unhelpful tuples and improve the analysis efficiency.

Our approach involves embedding nodes from derivation graphs into real vectors using a two-step process. First, we map each node to initial feature vectors based on their relation names. Then, we perform message passing to aggregate information from node neighborhoods and update the feature vectors accordingly. This captures information about high-order neighbors. Next, the feature vectors are passed through a two-layer fully connected neural network to generate scalar scores for each parameter tuple. Unhelpful tuples are excluded based on their scores, improving the efficiency of the refinement process. Overall, our approach effectively embeds nodes into real vectors and prunes unhelpful parameter tuples, leading to accelerated solving of the MaxSat solver and improved performance.

Learning. Since we aim to filter out *unhelpful* abstraction parameters, this problem can be modeled as a binary classification problem. We run a conventional refinement algorithm [Zhang et al. 2014] on small benchmarks and collect the derivation graphs and refinement traces that are identified using a MaxSat solver. Then we run a data processing algorithm as described in § 4.4 to mark helpful parameters as positive samples, while the remaining parameters are negative samples.

Concretely, our approach performs a backward iteration through the trace. For every refinement, we first filter out unhelpful parameters that are refined in this refinement. Then we look for other helpful parameters that are not refined and add them to the set of helpful parameters. In this example, suppose the trace chooses to refine `CI(11:id2)` at the first refinement and refine `CI(9:id1)` at the second refinement. We process the last refinement first. The second refinement in this example refines the abstraction from A_1 ={`CS(11:id2)`} to A_2 ={`CS(11:id2)`, `CS(9:id1)`} (`CI` tuples are omitted for simplicity). Since A_1 cannot eliminate the query, we can conclude that the parameter refined in this refinement (`CS(9:id1)`) is helpful. Then, we look for other helpful parameters that are not refined. We achieve this goal by trying all other abstractions like {`CS(11:id2)`, p}. Those parameters p that eliminate q will also be marked as helpful. In this example, only `CS(10:id1)` satisfies this condition. After processing this refinement, we add (A_1, {`CS(9:id1)`, `CS(10:id1)`}) to our training dataset, which means that our neural network should classify {`CS(9:id1)`, `CS(10:id1)`} as helpful given abstraction A_1 as its input.[1] Then we check the last but one refinement. Because neither A_0 ={} nor A_1 ={`CS(11:id2)`} can eliminate the query, we cannot determine whether

[1]Since analyzers generate derivation graphs using the given abstractions, derivation graphs can be determined by abstractions. Therefore, we denote abstractions as inputs for simplicity.

`CS(11:id2)` is helpful (Some queries may only be eliminated by the coexistence of multiple parameters. therefore, refining a parameter may not eliminate any query until some other parameters are also refined.). To tackle this problem, helpful parameters in later refinements (`CS(9:id1)` in this example) are added to current abstractions. By adding `CS(9:id1)` to A_0 and A_1, We get auxiliary abstractions A'_0 ={`CS(9:id1)`} and A'_1 ={`CS(9:id1)`,`CS(11:id2)`}. Since A'_1 can eliminate the query q, the helpfulness of `CS(11:id2)` can be determined by running the analysis with the abstraction A'_0, as $A'_0 = A'_1 \setminus$ {`CS(11:id2)`}. We run the analysis with A'_0 and find out that {`CS(9:id1)`} is enough to eliminate q, so we can safely mark parameters refined this refinement (`CS(11:id2)`) as unhelpful. Since no helpful parameter is refined in this refinement, we do not look for other helpful parameters. According to properties of pointer analysis (for example, monotonicity), helpful parameters in later refinements can be considered helpful for earlier refinements. Therefore, we add {`CS(9:id1)`, `CS(10:id1)`} to the set of helpful parameters of A_0. Finally, this refinement is added to the dataset as (A_0, {`CS(9:id1)`, `CS(10:id1)`}). After generating the dataset, we train our graph neural network using the standard settings of binary classification.

3 Preliminaries

In this section, we introduce the necessary notations before introducing our approach. In particular, we will define the syntax and semantics of Datalog(§ 3.1), parametric program analyses(§ 3.2), and the cheapest abstraction problem(§ 3.3), which is the problem that we try to solve approximately.

3.1 Datalog Syntax and Semantics

Figure 3 shows the syntax and denotational semantics of Datalog, where $\star$ represents zero or more occurrences of objects. Figure 3a shows the syntax. The topmost component is a program, which comprises zero or more constraints. Every constraint has a head and a tail, representing its conclusion and conditions, respectively. Head is a literal and tail is formed by zero or more literals. Every literal is a relation name followed by arguments, which are either variables or constants. Specially, tuples are literals whose arguments are all constants.

Figure 3b shows the denotational semantics. Every Datalog program C outputs a set of tuples. Datalog programs typically come with input tuples, but they can be encoded as constraints without conditions. So we omit them for simplicity. Every constraint of the form $l_0 : -l_1, \ldots, l_n$ is interpreted as a rule: for any substitution σ, if $\sigma(l_1), \ldots \sigma(l_n)$ are all derived, $\sigma(l_0)$ will be derived. For example, rule `A(x):-B(x,y),C(y)` means that `A(x)` is derived if there exists at least one `y` such that `B(x,y)` and `C(y)` are both derived. More specifically, if `B(1,2)` and `C(2)` are derived, then we can pick $\sigma = [x \longrightarrow 1, y \longrightarrow 2]$, so that $\sigma($`A(x)`$)$ = `A(1)` is derived. We here abuse the notation of σ so it can apply to a literal and substitute all variables in the literal with constants. In particular, if a constraint has no tail literal, we consider its head literal as derived. The denotation of a Datalog program C is the least fixpoint of denotations of its constraints. It is known that the denotation of a Datalog program is monotonic: if $C_1 \subseteq C_2$, then $[\![C_1]\!] \subseteq [\![C_2]\!]$.

3.2 Parametric Program Analysis

We now introduce parametric program analyses in Datalog, which include parameters that allow users to control the granularity of their abstractions, thereby controlling their precision and scalability. For example, k-CFA [Shivers 1991] is a typical family of such analyses. In k-CFA, every method invocation is combined with a parameter that determines how this invocation should be analyzed (context insensitively/context sensitively), like the example in the Overview.

In a parametric program analysis, we introduce a special set of tuples called *parameter tuples*, or parameters for short. A parametric program analysis in Datalog comprises three parts:

(program)	$C ::= c^\star$	(constraint)	$c ::= l : -l^\star$
(literal)	$l ::= r(a^\star)$	(argument)	$a ::= v \mid d$
(variables)	$v \in \mathbf{V} = \{x, y, \dots\}$	(constants)	$d \in \mathbf{D} = \{0, 1, \dots\}$
(relations)	$r \in \mathbf{R}$	(tuples)	$t \in \mathbf{T} = \mathbf{R} \times \mathbf{D}^\star$

(a) Syntax of Datalog.

$$\begin{aligned}
\llbracket C \rrbracket &= \text{lfp } F_C \in \mathcal{P}(\mathbf{T}) \\
F_C, f_c &\in \mathcal{P}(\mathbf{T}) \to \mathcal{P}(\mathbf{T}) \\
F_C(T) &= T \cup \bigcup \{f_c(T) \mid c \in C\} \\
f_c(T) &= \{\sigma(l_0) \mid \exists \sigma \in \mathbf{V} \mapsto \mathbf{D}. \\
&\qquad \sigma(l_k) \in T \text{ for } 1 \le k \le n\}
\end{aligned}$$

(b) Denotational semantics of Datalog.

Fig. 3. Syntax and denotational semantics of Datalog.

- a set of constraints that represent the analysis specification which is consistent across different programs;
- a set of input tuples extracted from a specific program that represent its facts;
- a set of parameter tuples specifying the precision of analysis across different components.

We denote running a parametric analysis as $\llbracket C \cup A \rrbracket$, where C denotes the first two parts and A denotes the last part. Throughout the paper, we assume that any abstraction A is sound.

Example. Consider the example Datalog analysis in the overview. Figure 2b shows the general analysis rules that hold across different programs. They represent how may-point-to tuples grow with assignments and how the analysis handles context sensitivities based on the parameters. Figure 2c shows the input tuples that represent program facts, and they are automatically extracted from a given program. Here, they are may-point-to facts that are generated from new statements, parameter passing, and so on. Figure 2d shows the parameters. Each represents whether a given call site is handled context-insensitively or -sensitively (1-CFA). Here, contents in Figure 2b and Figure 2c together constitute C, while Figure 2d shows A. $\llbracket C \cup A \rrbracket$ stands for the result of the analysis under context insensitivity, which is shown as Figure 2e.

A query is a special tuple that represents a program property of interest:

$$q \in \mathbf{Q} \subseteq \mathbf{T}.$$

The set of queries varies according to analyses. For example, for a pointer analysis, queries can be "variable `v` may point to null" and for typestate analysis, queries can be "file object `f` is never closed". When running an analysis on a program, we have a set of queries $\mathbf{Q}$ we want to answer. We say a query q is resolved using abstraction A if $q \notin \llbracket C \cup A \rrbracket$. We also say A is a viable abstraction to q. We use $R(A, C, Q)$ to denote the set of queries that are resolved using abstraction A: $R(A, C, Q) = Q \setminus \llbracket C \cup A \rrbracket$.

On a program, given a set of queries Q and a family of abstractions $\mathbf{A}$, we now define a precision preorder $\preceq$ and an efficiency (scalability) preorder $\sqsubseteq$. We say abstraction A_1 is not more precise than A_2 iff the analysis can not resolve more queries with A_1: $R(A_1, C, Q) \subseteq R(A_2, C, Q) \Leftrightarrow A_1 \preceq A_2$. Modeling scalability is more complex and we rely on domain knowledge to define the scalability preorder for a given analysis. For example, in k-CFA, an abstraction is an array of natural numbers with the fixed length, each of which represents the context depth for each call site and object. We say $A_1 \sqsubseteq A_2$ iff $\forall i \in [0, |A_1|[.A_1[i] \ge A_2[i]$, which means A_1 is more context-sensitive.

Example. Consider the example in the overview section. `MayPointTo(0:main:corgi3, Teddy)` is the only query (denoted as `q`). Abstraction A_1 = {`CI(9:id1)`, `CI(10:id1)`, `CI(11:id2)`} cannot resolve `q` while A_2 = {`CS(9:id1)`, `CI(10:id1)`, `CI(11:id2)`} can. Therefore, $A_1 \preceq A_2$. In terms of the scalability preorder, we follow the aforementioned preorder for k-CFA. In particular, `CS` is considered to have a context depth of 1 while `CI` is considered to have a context depth of 0. Therefore, $A_2 \sqsubseteq A_1$.

Algorithm 1 Iterative abstraction refinement with a graph neural network.

```
Require: Analysis $C$, abstraction family $\mathbf{A}$, queries $Q$
Ensure: Resolved queries $Q_R$, final abstraction $A$
 1: $A \leftarrow$ chooseInit($\mathbf{A}$)
 2: $Q_R \leftarrow \emptyset$
 3: repeat
 4:     $(D, \bar{Q}_R) \leftarrow$ analyze($C$, $A$, $Q$)    ▷ $D$ is a derivation.
 5:     $Q_R \leftarrow Q_R \cup \bar{Q}_R$
 6:     $G \leftarrow$ transform($D$, $Q$)    ▷ $G$ is a graph.
 7:     scores $\leftarrow$ GNN($G$)
 8:     $A, Q_F \leftarrow$ MaxSat($D$, scores)
 9:     $Q \leftarrow Q \setminus Q_R \setminus Q_F$
10: until Q = $\emptyset$
11: return $(Q_R, A)$
```

3.3 Problem Statement

The primary objective of our approach is to discover a cheap abstraction that can effectively resolve a maximum number of queries.

Definition 3.1 (Cheapest Abstraction Problem.). Given an analysis C, a set of queries Q, and a sound abstraction family $(\mathbf{A}, \preceq, \sqsubseteq)$, let $R(\mathbf{A}, C, Q)$ be a maximum set of queries that can be resolved by an abstraction in $\mathbf{A}$, that is

$$\exists A_1 \in \mathbf{A}.R(A_1, C, Q) = R(\mathbf{A}, C, Q) \quad \wedge \quad \forall A \in \mathbf{A}.R(\mathbf{A}, C, Q) \supseteq R(A, C, Q).$$

The cheapest abstraction problem is to find a cheapest A_{min} that can resolve these many queries

$$|R(A_{min}, C, Q)| = |R(\mathbf{A}, C, Q)| \quad \wedge \quad \forall A.|R(A, C, Q)| = |R(\mathbf{A}, C, Q)| \Rightarrow A \sqsubseteq A_{min}.$$

4 Our Framework

Algorithm 1 shows our general framework to solve the cheapest abstraction problem approximately. It starts with an abstraction by calling `chooseInit` at Line 1, whose implementation depends on the specific analysis – usually it returns the cheapest abstraction in the family (for example, fully context-insensitive for k-CFA). Then the framework enters the refinement loop (Line 3 - 10). During every iteration, it analyzes the program using the current abstraction (Line 4) and returns the derivation graph and queries resolved in this iteration, updates the set of all queries resolved (Line 5), and then transforms the derivation that is extracted from the Datalog solver into a graph (Line 6). Then our framework feeds the graph into our neural network which assigns a score to each parameter tuple (Line 7). Using the scores and the derivation, our framework formulates a MaxSat problem which modifies the one introduced by Zhang et al. [2014]. By solving the MaxSat problem, the framework gets the next abstraction to try and a set of queries that our framework gives up upon (Line 8). These queries either cannot be resolved with any abstraction in $\mathbf{A}$ or its viable abstractions are pruned by our graph neural networks. Our experiment shows that the latter case happens rarely in practice. Then our framework updates the query set Q by excluding the resolved queries and the queries that it gives up upon (Line 9). The iteration continues until Q becomes empty or the overall process exceeds a given time limit. It is obvious that our approach is always sound as long as abstractions in the family are sound.

4.1 From Derivation Graphs to GNN Inputs

We first explain how to extract derivations from Datalog solvers (the `analyze` function in Algorithm 1) and build an input graph to our GNN (the `transform` function in Algorithm 1).

A derivation is the set of instance constraints in the least fixpoint computation of Datalog:

$$D(C) = \{\sigma(l_0) : -\sigma(l_1), ..., \sigma(l_k) \mid l_0 : -l_1, ..., l_k \in C \wedge \forall i \in [1, k].\sigma(l_k) \in [\![C]\!] \wedge k \geq 0 \wedge \sigma \in \mathbf{V} \mapsto \mathbf{D}\}.$$

While some Datalog solvers have built-in support to return such derivations, we propose an extraction approach independent of solvers by modifying the given Datalog program. Briefly, for each constraint $l_0 : -l_1, ..., l_n$, we add a constraint in the following form:

$r_{cn}(v_1, ..., v_k) : -l_1, ..., l_n,$ where $v_1, ..., v_k$ are the variables that appeared in the original constraint.

Here r_{cn} is a new relation that is unique for each constraint. By inspecting r_{cn}, we can obtain all instances of the corresponding constraint that are triggered in the least fixpoint computation. For example, given the rule `A(y):-B(x,y),C(y,z).`, an additional rule is introduced as `A0(x,y,z):-B(x,y),C(y,z).`. Therefore, by inspecting elements of relation `A0`, we know not only on what elements `y` the relation `A` holds, but also how they are derived. For example, if `A0(0,1,2)` holds, then `A(1)` can be derived from `B(0,1)` and `C(1,2)`.

The derivation naturally forms a hypergraph, where each hyperedge corresponds to an instance constraint. For example, the instance constraint deriving `MayPointTo(0:id1:dog, Corgi)` in figure 2b corresponds to a directed hyperedge from $\{$`MayPointTo(0:main:corgi1, Corgi)`, `Assignment(0:id1:dog, 0:main:corgi1)`$\}$ to $\{$`MayPointTo(0:id1:dog, Corgi)`$\}$. Thus, the part of the derivation that derives `MayPointTo(0:main:corgi2, Corgi)` can be modeled as a hypergraph in Figure 4a.

Next, our framework translates a derivation hypergraph into a graph that can be consumed by our graph neural network. In particular, it converts the graph into a simple graph and removes information that cannot be generalized across programs to avoid over-fitting. More concretely, for each vertex, which is a tuple, we only keep the relation name as its information because the constants are specific to given programs. We represent the relation name using a one-hot encoding scheme, so that the vertex with the i-th relation name will be assigned a tuple $(0, \dots, 0, 1, 0, \dots, 0)$ where the only 1 is in the i-th position. Additionally, we assign two binary flags to present more information, one indicating whether the tuple is a query and the other indicating whether the tuple is a parameter. The query flag is important for identifying the goal of our analysis, while the parameter flag is important as it identifies the tuples that the neural network will evaluate. Consequently, the complete vertex information is encoded in a binary format using 0-1 values.

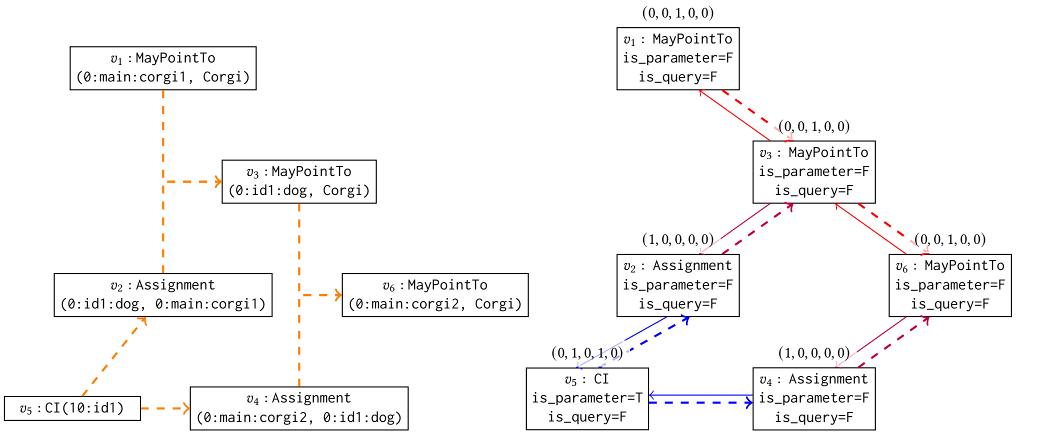
For each hyperedge, we add a directed edge from each of its source vertexes to its destination vertex. We also assign it a label $(forward, c, i)$ where c is the original (template) constraint, and i is the index of the source vertex in the constraint. In addition, a reverse edge is added for each directed edge with a label $(backward, c, i)$. This design allows bi-directional message passing in our network. So every tuple gets information from the tuples it derives and the tuples derive it.

Example. The derivation hypergraph in Figure 4a is converted into a simple graph shown in Figure 4b. Each vertex is labeled with its relation name, whether it is a parameter, and whether it is a query. The figure also shows the 0-1 encoding for each vertex. Since there are 3 relations in the graph, the first 3 bits stand for the relation name for every vertex: v_2 and v_4 belong to `Assignment`, so their first entries are 1; v_5 belongs to `CI`, so its second entry is 1. Other vertexes belong to `MayPointTo`, so their third entries are 1. Different colors or line styles (solid or dashed) stand for different edge labels. Following this, the hypergraph with 6 vertexes and 4 hyperedges is transformed into a graph with 6 vertexes in 3 vertex labels and 12 edges in 6 edge labels.

4.2 Graph Neural Network Architecture

We use a classical message-passing-style graph neural network workflow. First, we transform the discrete 0-1 encoding of vertexes into real vectors. Then, a process called message passing is repeated for a predefined number of iterations. During its execution, each vertexes' feature vector is updated based on the feature vectors of its neighbors and itself. We next explain the process in detail.

First, each vertex is assigned to a feature vector of reals. These vectors are initially calculated from the aforementioned 0-1 encodings. More concretely, we use a learnable matrix $E \in \mathbf{R}^{T \times H}$ to transform encodings of size T into real vectors of size H. For every vertex i, h_i^t represents its feature


v_1 : MayPointTo (0:main:corgi1, Corgi)
v_3 : MayPointTo (0:id1:dog, Corgi)
v_2 : Assignment (0:id1:dog, 0:main:corgi1)
v_6 : MayPointTo (0:main:corgi2, Corgi)
v_5 : CI(10:id1)
v_4 : Assignment (0:main:corgi2, 0:id1:dog)
(0, 0, 1, 0, 0)
v_1 : MayPointTo is_parameter=F is_query=F
(0, 0, 1, 0, 0)
v_3 : MayPointTo is_parameter=F is_query=F
(1, 0, 0, 0, 0)
v_2 : Assignment is_parameter=F is_query=F
(0, 0, 1, 0, 0)
v_6 : MayPointTo is_parameter=F is_query=F
(0, 1, 0, 1, 0)
v_5 : CI is_parameter=T is_query=F
(1, 0, 0, 0, 0)
v_4 : Assignment is_parameter=F is_query=F

(a) Part of the hypergraph of program in Figure 2a. (b) Transformed graph of the hypergraph in Figure 4a.

Fig. 4. Converting a derivation hypergraph into an input graph to our graph neural network.

vector in the t-th message passing (h_i^0 represents the initial feature vector), and t_i represents the 0-1 encoding of the vertex t. Then we have:

$$h_i^0 = E \times t_i.$$

Then, the algorithm enters message passing. In each iteration, the feature vector of each vertex is updated by combining its feature vector and information aggregated from its neighbors. To take edge labels into account, we apply the network architecture of R-GCN[Schlichtkrull et al. 2018], where each edge label l has its own learnable weight matrix M_l^t for the t-th iteration. We use $||$ to denote the concatenation of two vectors, m_i^t to denote the aggregated message that is passed to vertex i in the t-th iteration, $edge(i)$ to denote the set of edges that end with i. Each edge is a triple (j, i, l), where j is the source vertex and l is the label. Since our network has many layers, we add a residual connection to avoid the vanishing gradient problem [He et al. 2016]. Formally, we have:

$$h_i^{t+1} = \tanh\left(\Theta^t \times (h_i^t || m_i^t)\right) + h_i^t \quad \text{where} \quad m_i^t = \sum_{(j,i,l) \in edge(i)} M_l^t \times h_j^t.$$

Here, tanh is the Tanh activation function; Θ^t and M_l^t are learnable matrices of shape $2H \times H$ and $H \times H$, respectively. And they are different across iterations (as the superscripts suggest).

The message passing process terminates after a specified number of iterations, N. Let $h_i = h_i^N$ denote the final feature vector for vertex i. To compute the score for each parameter vertex i, we first process h_i through a two-layer perceptron to map it from a vector into a real number. We then apply a sigmoid function to scale this real number into the range $[0, 1]$, following common practice in classification tasks. The higher the resulting score s_i, the more helpful the neural network considers refining the corresponding parameter would be. The formal definition is as follows:

$$s_i = \text{sigmoid}\big(S_2 \times \text{LeakyReLU}(S_1 \times h_i)\big) \in [0, 1]$$

where $\quad \text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$ is a classical activation function [Maas et al. 2013],

S_1 and S_2 are learnable matrices of the perceptron.

Finally, a parameter i is classified as unhelpful if $s_i < 0.5$.

4.3 Formulating Refinement as MaxSat

We adopt the MaxSat formulation from Zhang et al. [2014] and prune parameter tuples that are considered unhelpful by our graph neural network. MaxSat problems contain two kinds of constraints: hard constraints (i.e., conventional SAT constraints) and soft constraints. A soft constraint, denoted by $w\ q$, means that if q is satisfied, the solution gains weight w. MaxSat solvers aim to find a solution that preserves all hard constraints while maximizing the sum of weights from satisfied soft constraints.

Let the Datalog analysis be C, the current abstraction be A, and the unresolved queries be Q. The original MaxSat problem formulation consists of three parts. All three parts are in disjunctive normal forms. Each tuple is translated into a boolean variable. First, a set of hard constraints Ω_C encodes the derivation. Second, there is a set of soft constraints $\{w_a\ a \mid a \in A\}$ such that $\sum_{a \in A' \subseteq A} w_a$ represents the incurred cost by refining parameters in A'. Finally, there is a set of soft constraints $\{w\ \neg q \mid q \in Q\}$ where $w > \sum_{a \in A} w_a$. This constraint indicates the goal is to resolve each query but can only give up queries when even refining all parameters cannot help.

If our graph neural network considers a parameter tuple a unhelpful, its corresponding soft constraint will be replaced with a hard constraint, so that parameter a will not be refined in this refinement. In the original framework, queries given up are irresolvable under the given abstraction family, which means that those queries cannot be eliminated with any abstraction in that abstraction family. But in our settings, MaxSat solver may give up queries earlier when all parameters related to them are considered unhelpful, which can be both advantageous and disadvantageous. There exist queries that are irresolvable, which are real bugs or false positives that cannot be resolved using the given abstraction family. Giving up on them earlier avoids over-refinement, which is beneficial in terms of the analysis efficiency. On the other side, giving up resolvable queries affects the performance of our approach. Therefore, we inspect how this affects the performance of our approach in § 5.4 and the result shows that by giving up queries earlier, our approach observably decreases the number of iterations and the size of the final abstraction with little precision loss.

4.4 Training Data Generation

Directly using refinement traces from the prior work[Zhang et al. 2014] as training data is a possibility. However, as aforementioned, their approach of using MaxSat to produce refinement candidates is a greedy algorithm and may lead to unnecessary refinement. And even when all refinements are helpful, some viable parameters may be unexplored. To further improve the quality of training data, we want to filter out unhelpful parameter tuples from the refinement traces and add unexplored helpful parameter tuples. To have a solid foundation, we give a formal definition of helpful parameters.

Definition 4.1 (helpful parameters.). Given a sound abstraction family $(\mathbf{A}, \preceq, \sqsubseteq)$, a set of queries Q, an abstraction parameter p is helpful if

$$\exists A_1 \in \mathbf{A}.\ \exists q \in Q.\ q \notin R(A_1, C, Q) \land q \in R(\text{update}(A_1, p), C, Q)$$

where $\text{update}(A_1, p)$ updates the corresponding part of A_1 by the abstraction parameter p.

Determining whether a parameter is helpful is generally an NP-hard problem because it is a combinational problem. Therefore, it is impractical to iterate all abstractions and identify all helpful parameters. Our approach provides an estimation that performs well practically.

Our algorithm is shown as Algorithm 3. We first run a vanilla CEGAR algorithm (Line 1), detailed in Algorithm 2. This algorithm is similar to Algorithm 1 but includes additional data collections in each iteration. Specifically, for each iteration, we collect the derivation graph, the current abstraction, the refined parameters. We also collect all resolvable queries.

Algorithm 2 Generating the Refinement Trace (`GeneratingRefinementTrace`)

```
Require: Analysis $C$, abstraction family $\mathbf{A}$, queries $Q$.
Ensure: Refinement Trace $T$, resolvable queries $Q_R$
 1: $A \leftarrow$ chooseInit($\mathbf{A}$)
 2: $T \leftarrow []$                                  ▷ $T$ collects the trace of the refinement.
 3: $Q_R \leftarrow []$                                ▷ Resolved queries.
 4: repeat
 5:     $(D, \bar{Q}_R) \leftarrow$ analyze$(C, A, Q)$
 6:     $Q_R \leftarrow Q_R \cup \bar{Q}_R$
 7:     $A\prime, Q_F \leftarrow$ MaxSat$(D)$
 8:     $Q \leftarrow Q \setminus (Q_R \cup Q_F)$
 9:     $P \leftarrow A\prime - A$                     ▷ Get parameters get refined in this round.
10:     $T$.append$(\{A, P, D\})$
11:     $A \leftarrow A\prime$
12: until $Q = \emptyset$
13: return $(T, Q_R)$
```

Algorithm 3 Generating the Training Dataset

```
Require: Analysis $C$, abstraction family $\mathbf{A}$, queries $Q$.
Ensure: Training set $\hat{T}$.
 1: $T, Q_R \leftarrow$GeneratingRefinementTrace$(C, \mathbf{A}, Q)$
 2: $\hat{T} \leftarrow []$                            ▷ $\hat{T}$ is the training set.
 3: $P_1 \leftarrow \emptyset$                         ▷ $P_1$ is the set of helpful parameter.
 4: $P_3 \leftarrow \emptyset$      ▷ $P_3$ is the set of parameters *alternative* to helpful parameters seen so far.
 5: for $\{A, P, D\} \in$reversed$(T)$ do
 6:     $P_2 \leftarrow \emptyset$                     ▷ $P_2$ is the set of likely unhelpful parameters seen so far.
 7:     for $p \in P$ do
 8:         $A\prime \leftarrow (A \cup P_1) \setminus P_2 \setminus \{p\}$
 9:         $(\_, Q\prime) \leftarrow$ analyze$(C, A\prime, Q_R)$
10:         if $Q\prime = Q_R$ then
11:             $P_2 \leftarrow P_2 \cup \{p\}$        ▷ $Q\prime = Q_R$ means that removing $a$ does not affect precision.
12:         else
13:             $P_1 \leftarrow P_1 \cup \{p\}$
14:             for $\hat{p} \in$ Parameters$(A')$ do
15:                 $(\_, \hat{Q}) \leftarrow$ analyze$(C, A\prime \cup \hat{p}, Q_R)$
16:                 if $\hat{Q} = Q_R$ then
17:                     $P_3 \leftarrow P_3 \cup \{\hat{p}\}$   ▷ Replacing $p$ with $\hat{p}$ does not affect precision.
18:     $\hat{T}$.append$(P_3 \cup P_1, D)$            ▷ Helpful parameters and the derivation graph.
19: return $\hat{T}$
```

Then, we enter the process of filtering out likely unhelpful parameter tuples (stored in P_2) and looking for unexplored viable parameters (stored in P_3). We perform a backward iteration through the refinement trace to simplify the helpfulness test of parameters, since we will collect helpful parameters into P_1 so that $A \cup P_1$ is always precise enough to resolve all resolvable queries and starting from the last iteration naturally satisfies this condition because A alone is precise enough. We start from the last iteration and cancel parameters refined in this refinement (P) one by one

(Line 7 – Line 17). If the number of resolved queries remains unchanged, the canceled parameter is considered likely unhelpful. We add the parameter to the set of likely unhelpful parameters (Line 11). Therefore, it will be ignored in later iterations of P. Otherwise, we mark it as helpful and add it back to the abstraction (Line 13). Then, to look for other helpful parameters, we try to replace the helpful parameter p with another parameter $\hat{p}$ (Line 14 – Line 17). If the number of resolved queries equals Q_R, $\hat{p}$ will also be marked as helpful.

Example. Suppose there are 5 parameters: $a_1, a_2, \ldots, a_5$ and a query that can be resolved if and only if $(a_2 \vee a_3 \vee a_4) \wedge a_5$ is satisfied. Given a trace that contains two refinements that refine the abstraction from $\emptyset$ to $\{a_5\}$ and then $\{a_1, a_3, a_4, a_5\}$. Our algorithm first focus on the refinement that refines $\{a_5\}$ to $\{a_1, a_3, a_4, a_5\}$. Parameters refined in this refinement are $\{a_1, a_3, a_4\}$. Our algorithm first tries to remove a_1, resulting in the abstraction $\{a_3, a_4, a_5\}$, which satisfies the condition $(a_2 \vee a_3 \vee a_4) \wedge a_5$; therefore the number of resolved queries stays unchanged, and we mark a_1 as likely unhelpful. Then we try to remove a_3, results in the abstraction $\{a_4, a_5\}$. Since the condition is still satisfied, we also mark a_3 as likely unhelpful. Next, we try to remove a_4, resulting in the abstraction $\{a_5\}$ that cannot resolve the query. Therefore, a_4 will be marked helpful. Then we try $\{a_5, p\}$ where $p \in \{a_1, a_2, a_3\}$. We will find that when $p = a_2$ or $p = a_3$, the query can be resolved again. Therefore, a_2 and a_3 will be added to P_3. So, for this refinement, we mark $\{a_2, a_3, a_4\}$ as helpful, exactly what the condition implies (a_5 is already refined at the beginning of this refinement step so it is not a candidate of parameters. Therefore, it does not need to be labeled helpful or unhelpful). Last, we focus on the refinement step that refines $\emptyset$ to $\{a_5\}$. Since a_4 is added to P_1 in the last iteration, $A \cup P_1 = \{a_4, a_5\}$ here is precise enough to resolve the query, making it possible to test the helpfulness of a_5 by removing a_5. This removal results in the abstraction $\{a_4\}$ that cannot resolve the query. Therefore a_5 will be marked helpful. So, for this refinement, we mark a_2, a_3, a_4 and a_5 as helpful, exactly what the condition implies.

4.5 Training

We apply classical supervised learning to train our networks to judge whether a parameter is helpful or not. To get a binary classification, we classify parameters with scores greater than 0.5 as helpful parameters, and classify other parameters as unhelpful parameters.

One issue we encountered is that the training data is highly imbalanced in terms of positive and negative samples. The reason is that, due to locality, an analysis usually only needs to apply high precision for a very small fraction of parameter tuples to resolve a query. Thus, in the training data, the number of negative samples are overwhelming to that of positive samples. To address this challenge, we apply importance sampling. In the loss function, we use *(the number of negative samples)/(the number of positive samples)* as the weight of each positive sample, and 1 as the weight of each negative sample. So the weights of positive samples add up to the same value as those of negative samples. Let $S(x) \in [0, 1]$ be the score of x calculated by our neural network, $\mathcal{D}^+$ and $\mathcal{D}^-$ be the set of positive and negative samples, respectively. Then the loss function is defined as:

$$\frac{|\mathcal{D}^-|}{|\mathcal{D}^+|} \cdot \sum_{x \in \mathcal{D}^+} \left(S(x) - 1\right)^2 + \sum_{x \in \mathcal{D}^-} \left(S(x) - 0\right)^2$$

5 Experiments

We evaluate our approach by comparing it with the CEGAR framework proposed by Zhang et al. [2014] using a pointer analysis and a typestate analysis, which are exactly the same settings as Zhang et al. [2014] use. Since pointer analysis is a popular field in programming languages and there are many works focus on improving the scalability of pointer analyses [Jeon et al. 2020; Li et al.

2018a,b; Ma et al. 2023; Tan et al. 2016, 2017], we also compare our approach to a state-of-the-art pointer analysis, Cut-Shortcut [Ma et al. 2023]. Due to space limits, we mainly use the pointer analysis to illustrate the effectiveness of our approach and use the typestate analysis to illustrate the generality of our approach. In particular, we aim to answer the following research questions:

RQ1. Can our approach help resolve more queries on large programs?
RQ2. How effective is our approach in reducing the sizes of constraint problems?
RQ3. How often does our approach make a resolvable query end up being unresolved?
RQ4. Is our "GNN + MaxSat" architecture necessary? Will graph neural networks alone suffice?
RQ5. Do our networks learn meaningful heuristics? What kind of heuristics do they learn?
RQ6. Is our approach general enough to boost different analyses?

5.1 Experimental Setup

Implementation. We have built our system upon the system proposed by Zhang et al. [2014]. In particular, we use JChord [Naik 2011] as the analysis frontend for Java programs and the MiFuMaX MaxSat solver [Janota 2014] as the underlying constraint solver. In JChord, we also reimplement Cut-Shortcut [Ma et al. 2023] to enable a fair comparison with our approach by eliminating the difference in the underlying analysis infrastructure. We use the Deep Graph Library [Wang 2019] with PyTorch [Paszke et al. 2019] to build our neural networks. We use scikit-learn[Pedregosa et al. 2011] to implement decision trees for explaining our neural networks.

Client Analyses. Following the previous work [Zhang et al. 2014], we apply our approach to a context-sensitive, flow-insensitive pointer analysis and a context-/flow-sensitive typestate analysis. The diverse features of these two analyses highlight the generability of our approach.

The pointer analysis is based on k-object-sensitivity [Milanova et al. 2002]. It is similar to the aforementioned k-CFA analysis except that it replaces call sites in k-CFA with abstract objects. Briefly, it uses contexts in the form of $h_1, \ldots, h_n$ to differentiate calling contexts. Here, h_i is an abstract object that the method receiver object *this* in a language like Java may point to. Like other call-string-based context-sensitive analyses, the contexts are truncated to some length k. Here, we allow different k values for different allocation sites and $k \in \{0, 10\}$. These k values form the abstraction for the pointer analysis. In this way, it gives a more fine-grained way to control the context sensitivity compared to the standard k-object-sensitivity analysis. The efficiency preorder is defined as follows:

$$A_1 \sqsubseteq A_2 \Leftrightarrow \forall\, h.A_1(h) \geq A_2(h).$$

As for the queries, we use the analysis to check whether a virtual method call is a polymorphic call (that is, it can call different methods at runtime).

The typestate analysis is adapted from that by Fink et al. [2008]. It is fully flow- and context-sensitive. But its context sensitivity is implemented in a different style: using the tabulation algorithm [Reps et al. 1995]. Moreover, it not only tracks may-alias information like the pointer analysis does, but also tracks must-alias information. More concretely, at each program point, it computes a set of abstract states of the form (h, t, a) that overapproximate the typestates of all objects at that program point. Here, h is an allocation site in the program, t is the typestate in which a certain object that is allocated at h might be, and a is a finite set of accesspaths that must point to an object in h. The abstraction we use is the set of variables allowed to track in must sets. Tracking relevant variables allows the analysis to perform strong updates to avoid spurious typestates, and thus is crucial to the precision and scalability of the analysis. The efficiency preorder is defined as

$$A_1 \subseteq A_2 \Leftrightarrow A_2 \sqsubseteq A_1$$

Table 1. Benchmark characteristics. All numbers are computed using a 0-CFA call-graph analysis.

benchmark	description	# classes		# methods		bytecode (KB)	
		app	total	app	total	app	total
`hsqldb`	relational database engine	189	1,341	2,441	10,223	190	670
`batik`	SVG graphics library	1,391	3,170	8,178	18,919	590	1,290
`bloat`	Java bytecode analysis/optimization tool	277	1,269	2,651	9,133	195	586
`pmd`	Java source code analyzer	348	1,357	2,590	9,105	186	578
`xalan`	XML to HTML transforming tool	42	1,036	372	6,772	28	417
`rhino-a`	Javascript engine	66	1,142	687	7,136	64	452
`sablecc-j`	parser generator for jimple	294	2,011	1,743	12,538	74	754
`sablecc-w`	parser generator for Wig	294	2,011	1,743	12,538	75	754
`soot-c`	Java program analyzer	561	1638	2,273	8,627	114	500
`soot-j`	Java program analyzer	1,638	1,638	8,627	8,627	114	500
`antlr`	parser/translator generator	111	350	1,150	2,370	128	186
`luindex`	document indexing tool based on `lucene`	206	619	1,390	3,732	102	235
`lusearch`	text search tool over a corpus of data based on `lucene`	219	640	1,399	3,923	94	250
`sunflow`	photo-realistic image rendering system ETH	165	1894	1,328	13,356	117	934
`schroeder-s`	sampled audio editing tool with small input	109	936	617	6,435	37	352
`schroeder-m`	sampled audio editing tool with medium input	109	936	617	6,435	37	352

GNN Hyperparameters. We set the number of message passing to 12. The dimensions of feature vectors at every iteration of message passing are all set to 64. We use the Adam optimizer [Kingma and Ba 2014], where the learning rate is set to 10^{-4} and the decay ratio is set to 0.9995.

Benchmarks. We consider the benchmarks from the DaCapo suite [Blackburn et al. 2006] and the Ashes suite [ash 2000]. Table 1 shows the characteristics of these benchmarks. The top half contains large benchmarks that Zhang et al. [2014] has trouble scaling to. We conduct our main scalability and efficiency study on these benchmarks to answer RQ1. The bottom half contains small benchmarks all of whose queries can be resolved by Zhang et al. [2014]. They are also used to answer RQ3. For other RQs, all these benchmarks are used.

Training Dataset. We use Zhang et al. [2014] to generate refinement traces and then use the algorithm described in § 4.4 to generate training data. Since our algorithm focuses on refining resolvable queries faster, we filter out benchmarks without resolvable queries. To reduce memory costs during training and illustrate the generability of our approach, we use the smallest benchmarks (by the sizes of the largest derivation graphs during their refinements) as our training dataset. For the pointer analysis, the four smallest benchmarks in order are `lusearch`, `luindex`, `antlr` and `sunflow`. Considering `lusearch` and `luindex` share the library `lucene`, we ignore `luindex` and only consider the three remaining benchmarks. Algorithm 3 takes 8 to 14 hours to transform the refinement traces into a training dataset. Since using full traces of all these benchmarks exceeds the 256 GB memory limit, we train 3 models by using each pair of these 3 benchmarks. Training each model takes approximately 10 hours per epoch, with a total of 20 epochs. For the typestate analysis, since only a few benchmarks contain resolvable queries, we only use the smallest benchmark `antlr` to make sure that there are enough test benchmarks to illustrate the effectiveness of our method.

Experiment Environment. We run all experiments on a Ubuntu 18.04 machine with 2 Intel Xeon Gold 6240 CPUs (2.6 GHZ) and set the memory limit to 256 GB memory. The Java runtime environment is the Oracle HotSpot JVM 1.6. The graph neural works run on the DGL library 0.8.2 with PyTorch 1.8.1 as the backend and the Python version is 3.8.5. The decision trees are trained using sk-learn 1.2.0. For the pointer analysis, we set the timeout limit to 12 hours. For the typestate analysis, we set the timeout limit to 24 hours.

Baselines. As aforementioned, the CEGAR framework proposed by Zhang et al. [2014] is used as a baseline for the two analyses and Cut-Shortcut [Ma et al. 2023] is used for the pointer analysis.

Table 2. Results on the pointer analysis using different training sets. Timeout > 12 hours.

benchmark	Zhang et al. [2014]				Cut-Shortcut [Ma et al. 2023]		`lusearch + antlr`				`lusearch + sunflow`				`antlr + sunflow`			
	# res.	# its	runtime	|A|	# res.	runtime	# res.	# its	runtime	|A|	# res.	# its	runtime	|A|	# res.	# its	runtime	|A|
`avrora`	70	12	Timeout	1320	0	0h03m	**74**	20	1h48m	1690	71	13	Timeout	1350	**74**	12	**0h55m**	2480
`batik`	**60**	**10**	Timeout	1940	29	1h09m	42	12	Timeout	1750	15	7	Timeout	1370	16	10	Timeout	1450
`bloat`	0	4	Timeout	420	42	0h05m	**243**	10	**2h19m**	1360	178	6	Timeout	1290	234	5	Timeout	2020
`pmd`	77	9	Timeout	1010	15	0h09m	**144**	19	3h55m	1780	129	18	3h17m	1720	104	23	Timeout	2640
`sablecc-j`	25	5	Timeout	590	22	0h04m	**105**	7	0h30m	590	98	6	0h25m	530	96	7	0h28m	630
`sablecc-w`	25	5	Timeout	590	22	0h04m	**105**	7	0h28m	590	98	7	0h27m	520	96	7	0h28m	630
`soot-c`	31	5	Timeout	470	0	0h01m	**98**	11	0h57m	1900	77	14	1h06m	2250	*13*	7	0h26m	1020
`soot-j`	31	5	Timeout	470	0	0h01m	**109**	11	1h00m	2080	77	14	1h06m	2250	*13*	7	0h27m	1020
`xalan`	10	14	Timeout	630	14	0h04m	**61**	18	**2h10m**	1850	14	18	2h45m	2050	60	15	Timeout	1800
`antlr`	**5**	15	**0h34m**	920	0	0h01m	*N/A*				**5**	**12**	0h42m	**820**	*N/A*			
`luindex`	**68**	28	1h12m	1210	0	0h02m	**68**	13	0h37m	740	**68**	17	0h50m	900	*57*	8	0h21m	420
`lusearch`	**30**	21	0h56m	1580	0	0h02m	*N/A*				*N/A*				*23*	12	0h33m	1110
`schroeder-m`	**25**	10	3h21m	270	0	0h08m	**25**	**5**	0h37m	**160**	**25**	9	1h04m	220	*20*	5	0h32m	220
`schroeder-s`	**25**	10	2h28m	280	0	0h08m	**25**	**5**	0h34m	**160**	**25**	9	1h02m	220	*20*	5	0h32m	220
`sunflow`	**10**	17	1h11m	1030	0	0h08m	**10**	**9**	**0h52m**	**300**	*N/A*				*N/A*			

Cut-Shortcut [Ma et al. 2023] applies a novel pointer analysis strategy which is different from classical cloning-based context sensitivity: It cuts off dataflow edges that reduce precision and adds shortcut edges to maintain soundness by recognizing patterns that are common in object-oriented programming. Therefore, their approach is able to analyze Java programs fast with relatively high precision.

5.2 Scalability and Precision Results

The top half of Table 2 summarizes the scalability results of the baselines and our approach using all three models mentioned above on pointer analysis. The "# res." columns show the number of queries resolved by our approach and the two baselines. Numbers in bold belong to the approach superior to the other approaches on the same benchmark. For "# res.", the largest number on each benchmark will be bold. Numbers of other columns will be bold if they are the smallest on the same benchmark and the corresponding "# res." is also the largest. Since Cut-Shortcut [Ma et al. 2023] is not refinement-based, we only list the overall runtime and the number of queries it resolves.

Compared to Zhang et al. [2014], our approach can resolve more queries for 22 out of 27 (model, benchmark) pairs. Compared to Cut-Shortcut [Ma et al. 2023], our approach resolves fewer queries only on `batik`. Another noteworthy result is that Cut-Shortcut runs notably faster than our approach and Zhang et al. [2014]. This conclusion is reasonable as recent pointer analysis works focus more on scalability, while refinement-based approaches focus more on precision, especially in resolving hard queries. The two styles of approaches are complementary to each other and can be combined for better scalability and precision: One can first apply faster approaches (for example, Cut-Shortcut [Ma et al. 2023]) to resolve as many queries as possible, and then apply our approach to resolve the remaining queries by leveraging the query-driven nature of our approach.

For a high-level understanding of the performance of all those approaches, we calculate the number of queries resolved by each approach among all benchmarks. Since the Ashes Suite Collection contains multiple running configurations for some programs [2] (`sablecc-j` and `sablecc-w` for SableCC, `soot-c` and `soot-j` for Soot), we take the average number of resolved queries for them. Under this criteria, for all benchmarks, our models resolve 772.5, 582, and 557 queries, respectively, while Zhang et al. [2014] only resolves 273 queries and Cut-Shortcut [Ma et al. 2023] only resolves 144 queries. Since Zhang et al. [2014] resolves more queries than Cut-Shortcut [Ma et al. 2023] on benchmarks except `bloat`, we compare our performance mainly to Zhang et al. [2014] later. Our improvement over Zhang et al. [2014] is about 183%, 113%, and 104%, respectively. On average,

[2] Configurations will affect the dynamic execution of programs, which is used by JChord to estimate hard-to-analyze language properties like reflection.

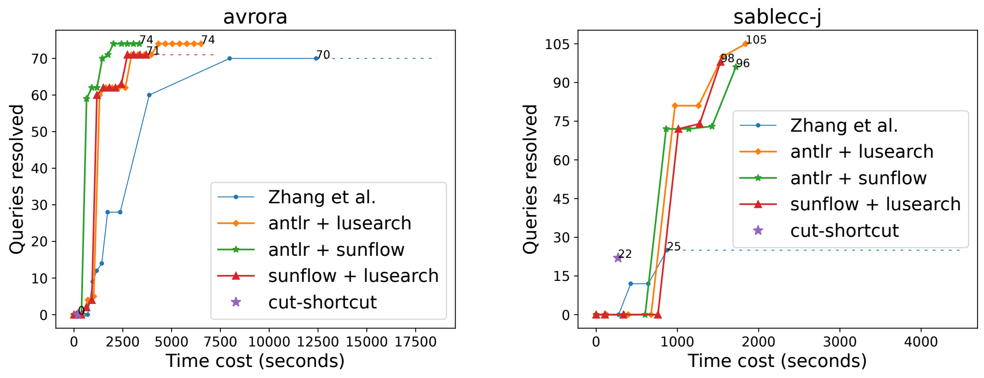


Fig. 5. Number of resolved queries and time consumption on `avrora` and `sablecc-j`.

our approach achieves an improvement of 133%. On `bloat`, the number of resolved queries even achieves a breakthrough of zero. Such results demonstrate the ability of our approach to resolve more queries on large benchmarks and the robustness of our approach to different training datasets. However, on `batik`, all of our models resolve less queries than Zhang et al. [2014], and two of them even resolve less queries than Cut-Shortcut[Ma et al. 2023]. We speculate the reason is that our training dataset is not diverse enough, so there are some unseen patterns that our models do not know how to deal with.

To further illustrate our effectiveness, we shall analyze the data presented in the "# its" columns and the "|A|" columns. The former denotes the number of refinement iterations completed before either terminating or reaching the twelve-hour time limit. The latter represents the sizes of the final abstractions achieved by each approach. These sizes are computed by summing all the k values (as described in § 5.1). On some benchmarks, Our approach demonstrates an ability to run for more iterations and therefore resolves more queries. For example, on both `soot` benchmarks and both `sablecc` benchmarks, the numbers of iterations have grown from 5 to 7 $\sim$ 14 and 5 to 6 $\sim$ 7, respectively. Moreover, the final abstractions of `soot` programs have grown five times in size. This shows that our approach empowers the refinement framework to explore more refined abstractions without incurring excessive time costs.

To look at the refinement process more closely, Figure 5 shows the number of resolved queries and time consumption per iteration for all approaches on `avrora` and `sablecc-j`, two of the larger benchmarks. (Other benchmarks are shown in Figure 6. As depicted in Figure 6 shows, most approaches either successfully finish the analysis or reach a stalemate before 4 hours. Setting the time limit to 12 hours provides those approaches with ample time to illustrate their inefficiency.) Dashed lines indicate that the approach does not terminate until the twelve-hour time limit is reached. For our approach, each iteration time consists of the running time of the analysis, the inference time of our graph neural network, and the constraint-solving time, while the iteration time of Zhang et al. [2014] only consists of the running time of the analysis and the constraint-solving time. As we can see, for `avrora`, our approach iterates much faster than Zhang et al. [2014]. Since our approach contains the overhead of a graph neural network, this boost indicates that our approach can quickly prune out useless parameters and provide better abstractions, so that the overhead of GNN can be reduced by the program analyzer and MaxSat solver. Furthermore, Zhang et al. [2014] gets stuck after only 12 iterations because its MaxSat solver gets stuck. However, Our approach is capable of completing up to 20 iterations, and with two of the models it terminates

Table 3. Parameter reduction using our graph neural network.

benchmark	Minimum number of parameter tuples		Maximum number of parameter tuples		average
	Before pruning	After pruning	Before pruning	After pruning	pruning rate
`avrora`	5186	390(92.5%)	5738	326(94.3%)	93.8%
`batik`	9584	1145(88.1%)	10193	1143(88.8%)	88.3%
`bloat`	4235	310(92.7%)	4939	330(93.3%)	93.2%
`pmd`	4211	439(89.6%)	4915	392(92.0%)	91.3%
`sablecc-j`	4253	297(93.0%)	4669	336(92.8%)	93.2%
`sablecc-w`	4253	297(93.0%)	4669	336(92.8%)	93.2%
`soot-c`	2379	221(90.7%)	3035	202(93.3%)	91.4%
`soot-j`	2379	221(90.7%)	3030	207(93.2%)	91.4%
`xalan`	4103	476(88.4%)	5021	460(90.8%)	89.8%
`luindex`	3023	296(90.2%)	3614	332(90.8%)	90.3%
`schroeder-m`	4335	500(88.5%)	4461	487(89.1%)	88.7%
`schroeder-s`	4335	500(88.5%)	4461	487(89.1%)	88.7%
`sunflow`	4876	420(91.4%)	5059	411(91.9%)	91.6%

within 2 hours. The termination shows that giving up queries is beneficial for scalability and our neural networks learn good strategies for giving up queries. Also, the iteration time of our approach does not grow significantly across iterations, unlike that of Zhang et al. [2014], where each of the last two iterations takes more than an hour. On `sablecc-j`, our advantage is even clearer, all our models terminate in half an hour, while Zhang et al. [2014] gets stuck after 15 minutes. Furthermore, the size of final abstractions of `avrora` and `sablecc-j` are of the same order of magnitude. This shows that our approach finds abstractions that are just precise enough and do not lead to much over-refinement. Cut-Shortcut [Ma et al. 2023] resolves 22 queries in 5 minutes, which reinforces our prior conclusion that recent pointer analysis works focus more on scalability while refinement-based approaches focus more on precision.

Answer to RQ1: by effectively pruning away unhelpful parameters, our approach enables the iterative refinement framework to resolve more queries on large benchmarks by trying more precise abstractions. Also, it does not lead to expensive over-refinement. Models trained with different training sets show that our approach is robust to different training sets.

5.3 Constraint Problem Simplification Results

Table 3 shows the number of parameters that can be refined (those without hard constraints) in the MaxSat problems before and after applying our graph neural network trained with `lusearch` and `antlr` during every step of refinement loops. As the table shows, our neural network drastically decreases the number of parameters that the MaxSat solver needs to consider to refine. Our method achieves an average reduction rate of around 90%, highlighting its effectiveness and the presence of many unnecessary abstraction tuples. This result explains why our approach iterates faster than Zhang et al. [2014]. Additionally, the average pruning ratios remain consistent across different benchmarks and abstractions, demonstrating the generalizability of our approach to unseen benchmarks of varying sizes and abstraction settings. These findings align with our earlier observation that our method resolves more queries and explores more expensive abstractions compared to Zhang et al. [2014].

Answer to RQ2: our graph neural network can significantly reduce the number of parameters that a MaxSat solver needs to consider, which is the key to making iterative abstraction refinements scale.

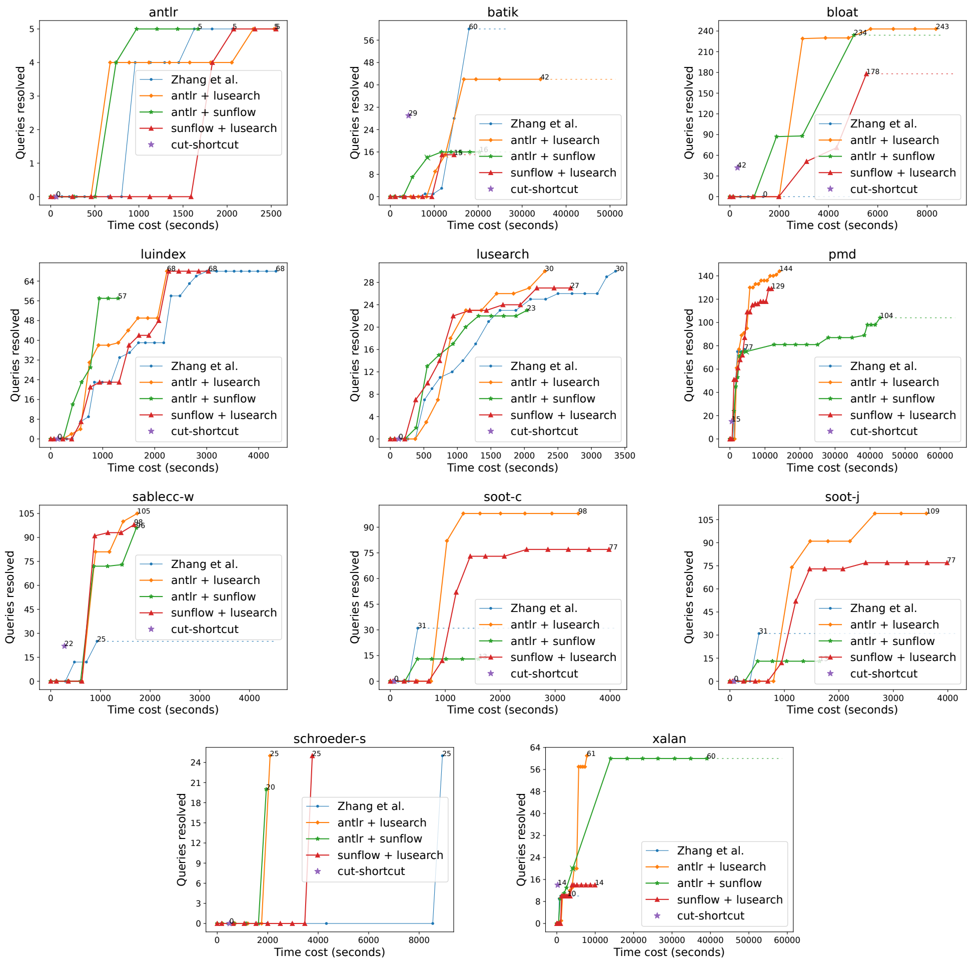


Fig. 6. Numbers of resolved queries and time consumptions across all benchmarks (in alphabetic order).

5.4 Controlled Precision Loss Study

As aforementioned, our approach may potentially hinder the precision of the analysis by accidentally pruning away helpful parameters. That is, the refinement process might give up some resolvable queries if all viable parameters are pruned by our approach. To estimate its potential effect, we inspect how often this happens by performing a controlled study using smaller benchmarks. On these benchmarks, Zhang et al. [2014] can successfully terminate so we know exactly which queries are resolvable.

The results[3] summarized in the bottom half of Table 2 indicate that our approach successfully resolves nearly all resolvable queries with minimal precision loss incurrred by parameter pruning. It also performs fewer iterations and consumes less time compared to Zhang et al. [2014]. This is illustrated by the examples of `lusearch` and `sunflow`, where our method achieves faster query resolution by pruning unhelpful abstraction tuples, as shown in Figure 7. Overall, our approach

[3]As a convention of machine learning, we ignore results on the training sets and mark the corresponding cells with *N/A*.

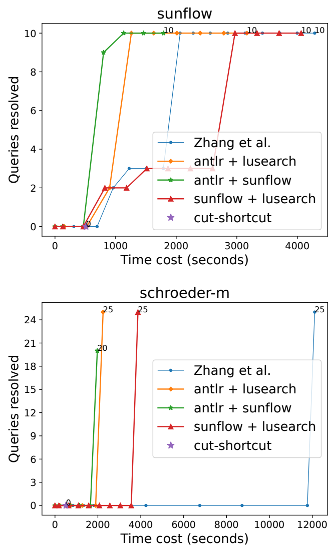


Fig. 7. Number of resolved queries and time consumption on `sunflow` and `schroeder-m`.

recall	98.55%	98.69%	98.47%	98.26%	98.64%
precision	98.55%	98.95%	98.86%	98.66%	98.67%

Fig. 8. Recall and precision of trained decision trees.

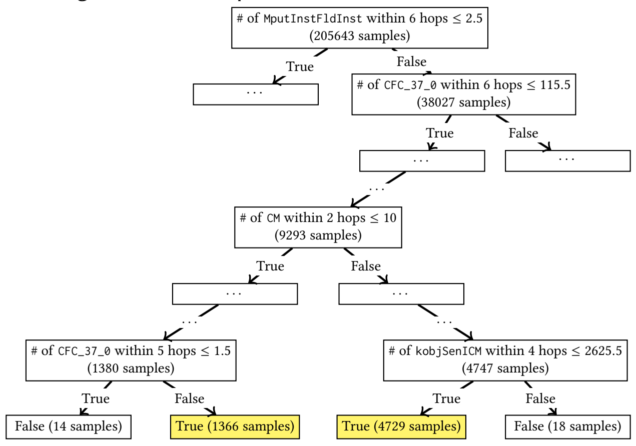

# of MputInstFldInst within 6 hops ≤ 2.5 (205643 samples)
True
False
...
# of CFC_37_0 within 6 hops ≤ 115.5 (38027 samples)
True
False
...
...
...
# of CM within 2 hops ≤ 10 (9293 samples)
True
False
...
...
...
...
# of CFC_37_0 within 5 hops ≤ 1.5 (1380 samples)
# of kobjSenICM within 4 hops ≤ 2625.5 (4747 samples)
True
False
True
False
False (14 samples)
True (1366 samples)
True (4729 samples)
False (18 samples)

Fig. 9. Part of the learned decision tree.

demonstrates effective query resolution with improved analysis time. On `schroeder-m`, two of our models finish the abstraction refinement within a time frame of 37 minutes to 64 minutes, whereas it takes Zhang et al. [2014] 201 minutes.

The precision losses all happen on the model trained with `antlr` and `sunflow`. This suggests that the absence of `lusearch` in the training dataset may lead to overly aggressive parameter pruning by the model. This indicates that larger training datasets with a more diverse range of benchmarks could further enhance the effectiveness of our approach. Overall, our approach boosts the refinement process with minimal risk of pruning out resolvable queries. In addition, it is effective on small benchmarks as well as large benchmarks.

Answer to RQ3: Though our approach can hinder the precision of the overall analysis theoretically, it has a limited impact in practice. Furthermore, a larger amount of training data may alleviate or even solve this problem.

5.5 Ablation Study

To study the effectiveness and necessity of our "GNN+MaxSat" architecture, we try to refine abstractions directly according to the output of our graph neural network, instead of using the output to filter out unhelpful parameters. We run the pointer analysis using only the GNN on the same benchmarks.

Table 4 presents the statistics for using only the GNN in the refinement process. The results show that the refinement process gets stuck quickly even on some smaller benchmarks, as the new architecture can only complete 2 iterations on most benchmarks. Those runtimes in red indicate that the memory limit is reached, which has never happened on other approaches. Additionally, the sizes of abstractions are much higher than those from other approaches. This conclusion is in agreement with the prior conclusion that some helpful parameters are interchangeable, which means that refining a part of those helpful parameters is enough to resolve desired queries. Given a

Table 4. Results of using graph neural network only. Timeout > 12 hours.

benchmark	Zhang et al. [2014]				lusearch + antlr				lusearch + antlr without MaxSat			
	# res.	# its	runtime	|A|	# res.	# its	runtime	|A|	# res.	# its	runtime	|A|
avrora	70	12	Timeout	1320	74	20	1h48m	1690	72	6	1h18m	5120
batik	60	10	Timeout	1940	42	12	Timeout	1750	0	2	Timeout	11450
bloat	0	4	Timeout	420	243	10	2h19m	1360	0	2	2h53m	3100
pmd	77	9	Timeout	1010	144	19	3h55m	1780	0	2	Timeout	4390
sablecc-j	25	5	Timeout	590	105	7	0h30m	590	105	5	0h44m	4140
sablecc-w	25	5	Timeout	590	105	7	0h28m	590	105	5	0h44m	4140
soot-c	31	5	Timeout	470	98	11	0h57m	1900	43	7	0h55m	3470
soot-j	31	5	Timeout	470	109	11	1h00m	2080	43	7	0h56m	3470
xalan	10	14	Timeout	630	61	18	2h10m	1850	0	2	2h10m	4760
luindex	68	28	1h12m	1210	68	13	0h37m	740	48	7	0h48m	3950
schroeder-m	25	10	4h50m	270	25	5	0h37m	160	0	2	3h35m	5000
schroeder-s	25	10	2h28m	280	25	5	0h34m	160	0	2	4h36m	5000
sunflow	10	17	1h11m	1030	10	9	0h52m	300	9	3	1h14m	5110

set of parameters that are considered helpful by the neural network, using GNN with MaxSat will only refine a part of those parameters while using GNN alone will refine them all. Despite those over-refined abstractions, some benchmarks can still be resolved within the time limit. However, these findings highlight the limitations of relying solely on the GNN for abstraction refinement, as it hinders the scalability and efficiency.

Answer to RQ4: Though graph neural networks can filter out unhelpful parameter tuples, they cannot capture exactly what parameter tuples to refine. Because some parameters are interchangeable, refining part of them is enough. As a result, they tend to make the abstraction too large to analyze. So, the combination of GNN and MaxSat solver is necessary.

5.6 Explaining Our Graph Neural Network

To explain what our neural network has learned and analyze whether it has learned reasonable patterns of parameter tuples, as well as to determine its ability to provide useful intuitions for selecting abstractions in program analysis, we train several decision trees to mimic our neural network. Since the rules of decision trees are transparent, we can analyze the behavior of our neural networks by analyzing nodes of the decision tree.

Feature Selection and Decision Tree Learning. Since we assign feature vectors for vertexes according to their relations, those feature vectors are just used to distinguish different types of nodes by the graph neural network and do not contain meaningful information. Therefore, we speculate that our graph neural network learns patterns more about graph structure. By randomly removing edges in the graphs, we observe that the neural network alters its output when the connectivity of certain nodes is disrupted. According to this observation, we speculate the patterns our neural network learns can be explained using numbers of vertices of some specific relations within specific hops. Therefore, we use the numbers of tuples of each relation in k hops ($k \leq 10$) from every vertex as our features. We denoted $x_{t,j,u}$ as the number of vertices of kind t in j hops from vertex u, where $t \in T, j \in \{1, \ldots, 10\}$. Thus, the dimension of every feature vector is $|T| \times 10$, which is 760 for the pointer analysis.

To learn and test those decision trees, we generate derivation graphs by running the pointer analysis on the benchmarks, and run a BFS (breadth-first search) for every parameter tuple u to get its $x_{t,j,u}$ as the input feature of decision trees. Then we run our graph neural network on those graphs to generate the network's scores of those parameter tuples as the labels for decision trees to learn. Since it takes a huge amount of time to run such BFS on large graphs, we only take derivation graphs from four small benchmarks: `antlr`, `lusearch`, `luindex` and `sunflow`. Then, we train decision trees to fit this data. To avoid over-fitting and better test how our decision trees imitate the neural network, we split the data into training sets and test sets using 5-fold cross-validation to train five decision trees without limiting their maximum depth (Our splitting is conducted at the level of abstract parameters. Therefore, approximately 80% of abstract parameters of each program are included in the training set). Table 8 displays the recall and precision of the trained decision trees. The results show that all decision trees achieve nearly 100% recall and precision, demonstrating their effective imitation of the neural network.

Running abstraction refinement with decision trees. To evaluate how well these decision trees learn and how well they perform, we replace our graph neural networks with decision trees to filter out unhelpful parameters in abstraction refinements. Results are shown in Table 5. For the decision trees, we provide their averages followed by standard errors in parentheses. For example, a cell entry of 5 (1.0) indicates an average of 5 with a standard error of 1.0. For benchmarks where only a subset of decision trees experience timeouts, we provide the number of trees that timed out and the completion time for those that either finished execution or reached the memory limit (indicated in red). Those 5 decision trees resolve 571, 490, 308, 392, 290 queries, respectively. The average 410.2 is much smaller than that of the model they try to fit (772.5), but still 50% higher than that ofZhang et al. [2014](273). This demonstrates that decision trees have the ability to partially fit the neural network.

For small benchmarks except two `schroeder` benchmarks, the decision trees resolve nearly all the queries. However, it takes significantly longer time compared to both Zhang et al. [2014] and our graph neural networks. This is largely attributed to our implementation. The extraction of input features (DFS) is implemented in Python. So even we use 64 cores parallelly to accelerate the process, it is still slower. Conversely, DGL uses a backend of C, so the overhead of graph neural networks is small. Despite the long runtime, the final sizes of abstractions quite close to those of our graph neural networks, and still smaller than Zhang et al. [2014]. All these results suggests that those decision trees capture most of the patterns learned by our graph neural networks, though some rules may be challenging to encapsulate in this feature representation. For the two `schroeder` benchmarks, only 1 of the 5 decision trees success to resolve those queries. This may be because those benchmarks are relatively different from those in the training data. However, the training data of those decision trees are already larger than those graph neural networks (4 benchmarks v.s. 2 benchmarks), implying that decision trees may not generalize as effectively as our graph neural networks.

For large benchmarks, the behavior of those decision trees aligns with our graph neural networks: they resolve more or about the same number of queries compared to Zhang et al. [2014] on most benchmarks except `batik`. The consistent underperformance of both learning-based approaches on this benchmark suggests that the diversity of the training set is important: There may be some rare patterns unseen in our training sets. It will be an interesting topic to generate training sets coverring more patterns. Despite the inefficiencies in our implementation, decision trees are able to successfully terminate on `sablecc` benchmarks, while the baseline cannot.

Inspecting a decision tree. We randomly select a trained decision tree from the 5-fold cross-validation and analyze its structure to see if we could gain any intuition from it. Because we care more about abstract parameters that are considered useful by the graph neural network, and they only account for 10% of the total, which are easier to classify, we only analyze leaves corresponding to `True`. The structure is shown in Figure 9, Related Datalog relations and domains are explained in Table 6 and Table 7. We mark two leaves that most positive samples go to in yellow; other paths

Table 5. Statistics about abstraction refinements with decision trees. Timeout > 12 hours.

benchmark	Zhang et al. [2014]				lusearch + antlr				decision trees			
	# res.	# its	runtime	$\|A\|$	# res.	# its	runtime	$\|A\|$	# res.	# its	runtime	$\|A\|$
avrora	70	12	Timeout	1320	74	20	1h48m	1690	60.6 (9.97)	11.2 (4.12)	Timeout	1760 (702.14)
batik	60	10	Timeout	1940	42	12	Timeout	1750	7.8 (5.04)	5.2 (0.40)	Timeout	948 (373.81)
bloat	0	4	Timeout	420	243	10	2h19m	1360	73.2 (87.96)	5.0 (2.61)	3 Timeouts,1h35m,1h49m	1368 (981.64)
pmd	77	9	Timeout	1010	144	19	3h55m	1780	72.8 (16.62)	5.6 (1.96)	4 Timeouts,3h38m	1224 (806.89)
sablecc-j	25	5	Timeout	590	105	7	0h30m	590	111.8 (12.61)	11.2 (1.47)	$2h1.2m$ ($20.95m$)	1326 (380.14)
sablecc-w	25	5	Timeout	590	105	7	0h28m	590	111.8 (12.61)	11.6 (1.85)	$2h10.2m$ ($37.17m$)	1350 (395.17)
soot-c	31	5	Timeout	470	98	11	0h57m	1900	63.0 (26.80)	10.2 (1.94)	2 Timeouts,1h54m,2h31m,3h59m	2032 (487.01)
soot-j	31	5	Timeout	470	109	11	1h00m	2080	63.0 (26.80)	9.8 (1.60)	2 Timeouts,2h37m,2h19m,3h10m	2024 (512.51)
xalan	10	14	Timeout	630	61	18	2h10m	1850	21.0 (21.55)	7.8 (1.94)	Timeout	1122 (893.38)
antlr	5	15	0h34m	920	5	12	0h42m	710	4.8 (0.40)	12.2 (0.75)	$2h5.8m$ ($10.23m$)	782 (42.61)
lusearch	30	21	0h56m	1580	30	12	0h38m	1270	28.0 (1.79)	12.2 (1.33)	$1h23.6m$ ($11.74m$)	1346 (57.83)
luindex	68	28	1h12m	1210	68	13	0h37m	740	63.8 (7.91)	15.2 (1.17)	$1h54.8m$ ($9.56m$)	862 (59.80)
schroeder-m	25	10	3h21m	270	25	5	0h37m	160	5 (10.00)	7.8 (1.72)	$3h51.0m$ ($99.72m$)	290 (72.94)
schroeder-s	25	10	2h28m	280	25	5	0h34m	160	5.0 (10.00)	7.2 (1.72)	$3h39.0m$ ($88.52m$)	254 (84.99)
sunflow	10	17	1h11m	1030	10	9	0h52m	300	9.6 (0.49)	13.6 (2.50)	$4h15.8m$ ($58.57m$)	506 (124.84)

Table 6. Relations in decision trees and their meanings

Relation	Meaning
`MputInstFldInst(m:M, b:V, f:F, r:V)`	There is a field-put operation that put `r` into the field `f` of base variable `b` in method `m`.
`CFC(c1:C, f:F, c2:C)`	The field `f` of object `c1` may point to object `c2`.
`CM(c:C, m:M)`	Method `m` is reachable under context `c`.
`kobjSenICM(i:I, c:C, m:M)`	There is a context-sensitive invocation `i` in method `m` using `c` as the receiver object.

Table 7. Related domains in decision trees and their meanings

Domain	Meaning
V	local variables of reference type.
I	Invocation quads.
M	Methods.
F	Fields.
C	Contexts, also used to represent objects.

are ignored for simplicity (since there are 1351 nodes in the decision tree). The two leaves cover 6094 positive samples out of 19143, which is about 31.8%. Other leaves are ignored since each of them covers less than 1000 positive samples.

4729 positive samples go to the rightmost yellow leaf, while 1366 positive samples go to the leftmost yellow leaf. Their paths share heuristics like "there are more than 2 field store operations for this object in the function that allocates this object" and "contexts ending with this object will not be used to call static methods". Except for these rules, those leaves disagree on some conditions. The leftmost yellow leaf contains heuristics like "there will be less than 10 (context, function) pairs such that the context ends with this object", while the rightmost yellow leaf is on a path where this condition is not satisfied. This rule stands for an elaborate consideration in the heuristics that consider parameter tuples by different patterns.

Answer to RQ5: The results show that decision trees learnt can learn most of the strategies of the graph neural network. And our graph neural networks learned meaningful (though complex) heuristics, and there are some elaborate conditions to classify different kinds of patterns.

Table 8. Results on the typestate analysis. Timeout > 24 hours.

benchmark	Zhang et al. [2014]				our method			
	# res.	# its	runtime	|A|	# res.	# its	runtime	|A|
`batik`	2	2	Timeout	87	**3**	6	Timeout	149
`bloat`	0	14	Timeout	227	0	17	11h39m	115
`antlr`	11	13	**0h46m**	66	11	13	0h59m	**45**
`rhino-a`	3	11	0h26m	35	3	**8**	0h26m	**25**
`sablecc-j`	22	11	1h45m	47	22	**6**	**1h12m**	**38**
`sablecc-w`	22	11	1h48m	47	22	**6**	**1h09m**	**38**

5.7 Generality over Different Analyses

To demonstrate the versatility of our method, we train another neural network for the typestate analysis. Since most benchmarks lack resolvable queries, we use only the `javasrc-p` dataset as the training set to ensure sufficient benchmarks for testing. The statistics are shown in Table 8.

Our method achieves a complete resolution of resolvable queries in small benchmarks and improves performance in terms of iteration count and tracking set size. In larger benchmarks, our method outperforms the baseline by resolving 3 queries instead of 2 on `batik`. Although the improvement is not as substantial as in the pointer analysis, it serves as evidence of the effectiveness of our method. The moderate improvement may be attributed to the limited abstraction family used in our typestate analysis.

Answer to RQ6: By running on a typestate analysis, we show that our approach can successfully generalize to different analyses.

6 Related Work

Our work is most related to CEGAR techniques that are based on constraint solving, program analyses that are augmented with learning, and constraint-solving techniques that are augmented with learning.

Constraint-based CEGAR. Originally, CEGAR was proposed to scale model checkers to handle larger spaces compared to existing symbolic approaches (i.e., those based on BDDs) [Clarke et al. 2003]. Constraint solvers have been heavily applied for different purposes. Notably, SAT solvers have been applied to perform model checking itself [Clarke et al. 2002; McMillan 2002], and decide whether a concrete counterexample is valid [Biere et al. 1999; Chaki et al. 2004]. Other solvers include integer-linear-programming solvers [Clarke et al. 2002] and SMT solvers [Komuravelli et al. 2016, 2013]. Our approach cannot be directly applied to these works. However, following the same spirit, it would be interesting to apply learing-based techniques to scale up the solving process.

Later on, CEGAR is broadly applied to tune parametric program analyses. Notably, Zhang et al. [2014] proposed a framework that applies a MaxSat solver to refine abstractions of Datalog-based program analyses. Our work is built upon this framework. Grigore and Yang [2016] proposes an approach that also extends this framework with a learning component. However, their focus is how to reduce the number of iterations in CEGAR by adding learned biases to the MaxSat formulation. In that sense, our approach and their approach are orthogonal and complimentary .

Scaling Pointer Analysis. It has been an active field to select abstractions that balance precision and scalability for pointer analysis [Jeon et al. 2019, 2020; Li et al. 2018a,b; Ma et al. 2023; Tan et al. 2016, 2017]. These works all propose techniques based on domain knowledges that are specified to the problem. On the other hand, our work works with any parametric program analysis that is expressed in Datalog. Furthermore, most recent works focus more on scalability like Cut-Shortcut [Ma et al. 2023] does, while our approach focuses more on precision. Some works try to generalize to a broader range of analyses using cleverer algorithms and system-level optimizations [Shi et al. 2018; Zuo et al. 2021]. These works are orthogonal to ours and can be combined with ours to scale an analysis even better.

Learning-Based Program Analysis. Before our paper, there has been significant interest in applying learning to tune abstractions or knobs of program analyses [Chae et al. 2017; Heo et al. 2018; Jeong et al. 2017; Oh et al. 2015]. However, they differ from our approach in two aspects. First, the input to our approach is the derivation graph of an analysis which can be automatically extracted without feature engineering. On the other hand, existing approaches rely on manually selected syntax features and therefore need more work to apply to a new analysis. Second, these works aim to find a good abstraction in one step while our approach is integrated with CEGAR. While their approaches typically run faster, our approach can scale to resolve harder queries on large programs. There are also works in learning loop invariants [Sharma et al. 2013; Yao et al. 2020; Zhu et al. 2018]. Obviously, the targeted problems are different. Moreover, while our approach uses abstract executions of the analysis as input features, these approaches use concrete executions of the program. Finally, one extreme is to use a learning component to completely replace a conventional analysis [Galassi et al. 2018; Wang et al. 2020]. These approaches do not guarantee soundness.

Learning-Aided Constraint Solving. Our work can be viewed as a preprocessor for a MaxSat solver in a particular domain. There has been a growing interest in applying deep learning to accelerate constraint solving or even replace conventional constraint solvers. Popescu et al. [2022] gives a comprehensive survey on this topic. Our work is tailored to the program analysis domain and therefore more effective in solving related problems.

7 Conclusion

We have proposed a framework to scale counterexample-guided abstraction refinement for Datalog-based program analysis using graph neural networks. Our networks can effectively identify part of a given abstraction, refining which will not likely help resolve the target queries. Compared to existing learning-based techniques, our network takes the execution of the program analysis as input, which can be automatically extracted from a Datalog solver. Our experiment shows that our approach scales significantly better than existing approaches on two representative analyses.

Data-Availability Statement

The software that supports section 5 is available on Zenodo [Yan et al. 2024].

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 62172017, the Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (Grant No. TD2019001), and Ant Group Research Fund.

References

2000. Ashes Suite Collection.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 101:1–101:25.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular Verification of Software Components in C. *IEEE Trans. Software Eng.* 30, 6 (2004), 388–402.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. Springer, 154–169.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794.

Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Strichman. 2002. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, 265–279.

Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 1–34.

Andrea Galassi, Michele Lombardi, Paola Mello, and Michela Milano. 2018. Model Agnostic Solution of CSPs via Deep Learning: A Preliminary Study. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10848)*, Willem Jan van Hoeve (Ed.). Springer, 254–262.

Radu Grigore and Hongseok Yang. 2016. Abstraction refinement guided by a learnt probabilistic model. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 485–498.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA, 2016-06). IEEE, 770–778. https://doi.org/10.1109/CVPR.2016.90

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2018. Learning analysis strategies for octagon and context sensitivity from labeled data generated by static analyses. *Formal Methods Syst. Des.* 53, 2 (2018), 189–220.

Mikoláš Janota. 2014. MiFuMax—a literate MaxSat solver. , 83–88 pages.

Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A machine-learning algorithm with disjunctive model for data-driven program analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 2 (2019), 1–41. Publisher: ACM New York, NY, USA.

Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, and Anna Potapenko. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589. Publisher: Nature Publishing Group.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205.

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and

Helmut Veith (Eds.). Springer, 846–862.

Gil Lederman. 2021. Neural Guidance in Constraint Solvers. (2021).

Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *MaxSAT, Hard and Soft Constraints*. Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press. https://doi.org/10.3233/FAIA201007

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018a. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. Publisher: ACM New York, NY, USA.

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018b. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 129–140.

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–40.

Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 539–564. https://doi.org/10.1145/3591242

Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. Atlanta, Georgia, USA, 3.

Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From datalog to flix: A declarative language for fixed points on lattices. *ACM SIGPLAN Notices* 51, 6 (2016), 194–208. Publisher: ACM New York, NY, USA.

Kenneth L. McMillan. 2002. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2404)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.). Springer, 250–264. https://doi.org/10.1007/3-540-45657-0_19

Kenneth L. McMillan. 2003. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*. Springer, 1–13.

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 1–11.

Mayur Naik. 2011. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*.

Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 572–588.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

Andrei Popescu, Seda Polat Erdeniz, Alexander Felfernig, Mathias Uta, Müslüm Atas, Viet-Man Le, Klaus Pilsl, Martin Enzelsberger, and Thi Ngoc Trang Tran. 2022. An overview of machine learning techniques in constraint solving. *J. Intell. Inf. Syst.* 58, 1 (2022), 91–118. https://doi.org/10.1007/s10844-021-00666-5

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 49–61.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. Springer, 593–607.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 574–592.

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 693–706.

Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda*. Carnegie Mellon University.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 485–495.

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–291.

Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27.

Zhenyu Yan, Xin Zhang, and Peng Di. 2024. *Scaling Abstraction Refinement for Program Analyses in Datalog Using Graph Neural Networks (Artifact)*. https://doi.org/10.5281/zenodo.12663344 Zenodo.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. 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, 106–120.

Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program Analysis via Efficient Symbolic Abstraction. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 118 (oct 2021), 32 pages. https://doi.org/10.1145/3485495

Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. 2020. NLocalSAT: Boosting local search with solution prediction. *arXiv preprint arXiv:2001.09398* (2020).

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. 239–248.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721.

Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. 2021. Chianina: An Evolving Graph System for Flow-and Context-Sensitive Analyses of Million Lines of C Code. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 914–929.

Received 2024-04-06; accepted 2024-08-18