


EUF-based Solving Dyck-Reachability with Applications to Static Analysis

Yide Du, Zhenbang Chen(✉), Kunlin Liu, Guofeng Zhang, Xudong Wang, Ke Ma, Wei Dong, and Ji Wang

College of Computer Science and Technology, National University of Defense Technology, Changsha, China
{dyd1024,zbchen,klliu18,zhangguofeng17,wangxudong,ke,dongwei,wj}@nudt.edu.cn

Abstract. Static analysis plays a crucial role in program optimization, bug detection, and automated testing. Dyck-reachability provides a foundational formulation for static analysis, as Dyck grammars can model critical properties such as field and context sensitivity, thus offering broad applicability. This paper shows that static analysis problems modeled as Dyck-reachability on bidirected graphs can be encoded into the EUF SMT theory; consequently, all such problems admit efficient formulation and solution via EUF-based SMT solvers. By leveraging the optimized nature of modern SMT solvers, our method achieves efficiency comparable to state-of-the-art graph-based bidirected Dyck-reachability algorithms while eliminating the need for developing complex specialized graph reachability algorithms. Our approach opens new avenues for solving these classical static analysis problems, demonstrating the strong potential of SMT solvers in encoding static analysis solutions.

Keywords: Static Analysis · EUF · SMT · Dyck reachability

1 Introduction

Program analysis is essential for software engineering tasks such as optimization [23], bug detection [20], and automated testing [36]. It enables automated program understanding and is broadly categorized into dynamic and static approaches. Dynamic methods (*e.g.*, fuzzing [9] and runtime monitoring [22]) execute programs to analyze runtime behavior, offering precision but limited coverage and the requirement of specific program execution environments. In contrast, static analysis examines code without execution by employing techniques such as data-flow analysis [3] to over-approximate program behavior through abstract models. These methods benefit from soundness, meaning they guarantee coverage of all possible program behaviors, but face inherent imprecision

Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China.

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16557, pp. 296–316, 2026.
https://doi.org/10.1007/978-3-032-26220-2_15

due to over-approximation. This imprecision can lead to false positives, and improving precision typically compromises scalability, establishing a fundamental precision-scalability trade-off. Achieving an optimal balance between these competing objectives remains both critical and challenging in practice.

Various static analysis problems including inter-procedural data-flow analysis [35], alias analysis [43], and pointer analysis [39] can be formulated as context-free language (CFL) reachability on graphs [34,35]. In this formulation, programs are modeled as graphs (*e.g.*, control-flow or field-access graphs), and the analysis is framed as a reachability problem under path constraints: it checks for paths where the sequence of edge labels belongs to a specified formal language for instance, a language that requires balanced or matched operations. Constraints such as function call-return consistency and memory load-store matching are thus directly encoded. While providing a natural modeling framework, CFL-reachability's cubic time complexity [25] limits scalability, leading to approximations such as bidirected Dyck-reachability. Defined by the Dyck language grammar [21], this approach operates on bidirected graphs with matched opening/closing labels, forming node equivalence relations that trade precision for efficiency [44]. It has been successfully applied to field sensitivity in alias analysis [44] and call-return matching in data dependence analysis [10,26,27].

The state-of-the-art bidirected Dyck-reachability algorithm achieves a worst-case time complexity of $O(m + n \cdot \alpha(n))$, where n and m denote the number of nodes and edges in the graph, respectively, and $\alpha(n)$ is the inverse Ackermann function [2]. Its average-case complexity is $O(m)$, making whole-program analysis computationally tractable. In contrast, the LLVM compiler[1] employs an alternative algorithm for alias analysis via bidirected Dyck-reachability with complexity $O(n + m \log m)$ [44]. However, implementing static program analysis tools remains challenging. Beyond frontend issues, developers must comprehend intricate data structures and algorithmic details, such as the FDLL (Fast-Doubly-Linked-List) used in graph-based approaches [44], and subsequently abstract and implement the algorithms based on their understanding. This engineering process is labor-intensive and error-prone.

Our key argument in this paper is that well-optimized logical reasoning tools can significantly simplify the design and implementation of static analyses. Inspired by the equivalence relation property of bidirected Dyck-reachability [44], we propose an encoding of such problems using the Equality with Uninterpreted Functions (EUF) SMT theory. This approach provides a straightforward implementation path while enabling direct utilization of highly optimized SMT solvers. Although SAT/SMT techniques are well-established in areas like symbolic execution [5] and path-sensitive analysis [37], to our knowledge, *we are the first to apply SMT solving to static analyses modeled by bidirected Dyck-reachability*, such as pointer and alias analyses. Modern SMT solvers like Z3 [12], CVC5 [6], and Yices [18] incorporate sophisticated algorithms and low-level optimizations, making them ideal for reducing engineering effort while ensuring robustness.

[1] https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/CFLSteensAliasAnalysis.cpp.

Our evaluation on two representative tasks demonstrates that this EUF-based approach achieves comparable efficiency to an implementation [26] of the state-of-the-art graph algorithm [10] for bidirected Dyck-reachability.

The main contributions of this paper are as follows.

- We propose to utilize SAT/SMT for more static analysis problems. Specifically, we propose a novel approach to solving static analysis problems modeled as bidirected Dyck-reachability problems by EUF SMT theory, which covers a lot of static analysis problems, such as alias and pointer analyses. *Our approach opens a new door to solving these classical static analysis problems and their implementations.*
- We show that any program analysis problem modeled by bidirected Dyck-reachability can be reduced to solving formulas in the EUF SMT theory.
- We evaluated EUF SMT-based, graph-based, and Datalog-based approaches on three static analysis tasks: field-sensitive alias analysis for Java applications and large-scale open-source C programs, and context-sensitive data dependence analysis for Java programs. Experimental results demonstrate that the best-performing SMT solvers, Yices [18], Plat-smt [19] and Egg [42], achieve efficiency comparable to the optimal graph-based algorithm [10]. Notably, some SMT solvers exhibit even better performance in terms of query overhead, highlighting their strong potential for static analysis tasks.

The rest of this paper is organized as follows. Section 2 introduces the necessary background and motivation. Section 3 presents the reduction from bidirected Dyck-reachability to EUF SMT solving. Section 4 provides experimental results. Section 5 provides a discussion. Finally, Sect. 6 concludes the paper.

2 Preliminaries and Motivation

This section provides an overview of EUF SMT theory and bidirected Dyck-reachability. Subsequently, by comparing the solving algorithms for both problems, we present the motivation for this work.

2.1 The Theory of Equality with Uninterpreted Functions

The theory of Equality with Uninterpreted Functions (EUF) is a core component of SMT solvers like Z3 [12] and CVC5 [6], *etc.*. In EUF, uninterpreted functions are defined by their signatures (names and input/output types) while omitting semantic details. This abstraction preserves only the *congruence* property: identical inputs produce identical outputs.

The axioms of EUF logic include the three axioms of equivalence relations and the congruence axiom, which states that functions with the same inputs produce the same outputs.

$$\forall x. \quad x = x \qquad \text{(reflexivity)}$$
$$\forall x, y. \quad x = y \rightarrow y = x \qquad \text{(symmetry)}$$
$$\forall x, y, z. \quad x = y \wedge y = z \rightarrow x = z \qquad \text{(transitivity)}$$
$$\forall \bar{x}, \bar{y}. \quad \bigwedge\nolimits_i x_i = y_i \rightarrow f(\bar{x}) = f(\bar{y}) \qquad \text{(congruence)}$$

Abstraction of functions as uninterpreted functions inevitably loses some information. For example, if addition is abstracted as f, the commutative property is not preserved: $f(x, y)$ and $f(y, x)$ are not considered equivalent under EUF congruence, despite $x + y = y + x$ for numbers. However, this paper leverages the congruence property to model parenthesis matching in Dyck language without incurring any precision loss.

We now illustrate the algorithm for solving EUF formulas through an example. Consider the unsatisfiable formula $\varphi : c = f(a) \wedge d = f(b) \wedge a = b \wedge c \neq d$. Its unsatisfiability can be illustrated via the construction of a congruence graph, as shown in Fig. 1. Graph nodes denote terms, with directed edges (labeled by function symbols, *e.g.*, f) representing function applications, and undirected red edges indicating term equivalence. Figure 1a shows the initial graph with all terms and their functional relationships. Subsequently, processing constraint $a = b$ introduces an equivalence edge between a and b (Fig. 1b). By the congruence property (equivalent inputs yield equivalent outputs), terms c and d become equivalent. Consequently, the graph is updated to Fig. 1c. This, however, contradicts the given assertion $c \neq d$, thus proving the unsatisfiability of φ.

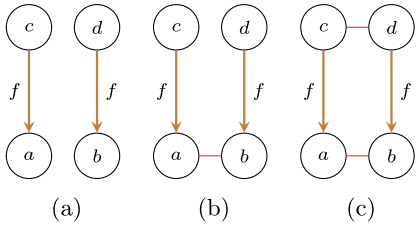


Fig. 1. Solving EUF SMT Formulas Using Congruence Graphs.

In summary, this algorithm first processes all equivalence relations by iteratively (1) merging equivalent classes and (2) discovering new equivalences via the congruence property, until a fixed point is reached. It then checks each inequality constraint to verify that its operands reside in distinct equivalence classes. The formula is *satisfiable* if no inequality conflicts with any of the computed equivalence classes; otherwise, it is *unsatisfiable*.

2.2 Bidirected Dyck-Reachability and Its Applications in Program Analysis Tasks

The context-free language (CFL) reachability problem on graphs extends the general graph reachability problem. We begin by the following definition:

Definition 1. *(labeled graph) A labeled graph is defined as a tuple $G = (V, E)$, where V is the set of vertices, and $E = V \times \Sigma \times V$ is the set of edges, with Σ being an alphabet.*

For simplicity, we introduce a function $\lambda : V \times V \to \Sigma$, where $\lambda(u, v) = l$ if and only if $(u, l, v) \in E$. Given a context-free grammar (CFG) $\mathcal{G}$ and a labeled graph G, where $L(\mathcal{G})$ represents the language accepted by the CFG $\mathcal{G}$, we say that two nodes $u, v \in V$ are *CFL-reachable* if and only if the following condition holds:

$$\exists t_1, t_2, \ldots, t_n \in V. \lambda(u, t_1) \cdot \lambda(t_1, t_2) \cdots \lambda(t_n, v) \in L(\mathcal{G}) \tag{1}$$

The Dyck language is a particular type of CFL. Its grammar is as follows, where $\Sigma = \{\epsilon, l_0, r_0, \cdots, l_i, r_i, \cdots\}$ are the terminal symbols, S is the start non-terminal symbol.

$$S \to l_0 S r_0 \mid \cdots \mid l_i S r_i \mid \cdots \mid SS \mid \epsilon \tag{2}$$

The language defined by this grammar requires balanced matching between terminal symbols, *i.e.*, symbols l_i and r_i appear in pairs, such as in the case of the parentheses matching language. Here, for every i, we refer to l_i as *opening labels* and r_i as *closing labels*, with the notation $\bar{l_i} = r_i$, $\bar{r_i} = l_i$, and $\bar{\epsilon} = \epsilon$.

Definition 2. *(bidirected graph) A bidirectional graph $G = (V, E)$ is a labeled graph where $(u, l, v) \in E$ if and only if $(v, \bar{l}, u) \in E$.*

Consider the Dyck-reachability problem on a general graph G. Relaxing G into a bidirected graph G' by adding reverse edges for each original edge (as Definition 2) yields the bidirected Dyck-reachability problem. This formulation constitutes a relaxation of the general Dyck-reachability problem, which may introduce imprecision. Specifically, if a node pair is Dyck-reachable in graph G, it remains Dyck-reachable in G'. However, the addition of bidirected edges may enable reachability between node pairs that are not connected in the original graph G, thus introducing imprecision. For example, consider the path $a \xrightarrow{l} b \xrightarrow{\bar{l}} c$: a is Dyck-reachable to c, but c is not Dyck-reachable to a in the original graph. However, when relaxed to a bidirected graph (Definition 2), the added edges $a \xleftarrow{\bar{l}} b$ and $b \xleftarrow{l} c$, making c reachable from a.

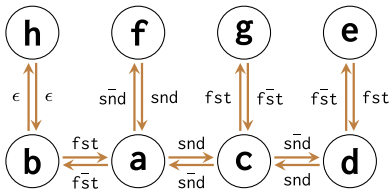
Theorem 1. *[44] The bidirected Dyck-reachability relation is an equivalence relation.*

This theorem states that the bidirected Dyck-reachability relation is *reflexive* (each node is Dyck-reachable to itself), *symmetric* (if nodes u and v are Dyck-reachable, then so are v and u), and *transitive* (if nodes u and v, and v and k are Dyck-reachable, then so are u and k).

Bidirected Dyck-reachability based field-sensitive alias analysis. Currently, many approaches utilize bidirected Dyck-reachability to model program analysis problems[10,24,26–29,44]. Figure 2 shows an example of alias analysis. Figure 2a shows a Java program, and Fig. 2b illustrates this program's corresponding labeled bidirected graph. In the graph, nodes correspond to program variables. Edges labeled with an *opening labels* represent field write operations, while those with a *closing labels* represent field read operations, with distinct

```
1  class Tree {
2      Tree fst;
3      Tree snd; }
4  void foo(){
5      Tree a, b;
6      Tree c, d;
7      Tree e, f, g;
8      ...
9      ...
10     a.fst = b;
11     c.snd = a;
12     a.snd = f;
13     d = c.snd;
14     e = d.fst;
15     c = g.fst;
16     h = b; }
```

(a) A Java program.



(b) Labeled bidirected graph for alias analysis.

Fig. 2. Bidirected Dyck-reachability based field-sensitive alias analysis.

labels denoting different field accesses. For example, the statement `a. fst=b` is encoded as an edge $\lambda(\texttt{b}, \texttt{a}) = \texttt{fst}$. Consequently, aliases arising from field accesses are represented by paths where opening and closing labels form matched pairs, corresponding to Dyck-reachable paths in the graph. In Fig. 2b, nodes {`a, d`} and {`h, b, e`} are Dyck-reachable, respectively (the equivalence class of each variable with itself is omitted), and the variables in each equivalence class are aliases in the original program.

Relaxing the graph to a bidirected graph introduces some approximation. For example, consider the program statements `a=c` and `a=b`. Under Andersen's pointer analysis [4], `b` and `c` are not considered aliases, but in bidirected Dyck-reachability based approach, `b` and `c` are considered aliases. However, the bidirected Dyck-reachability-based approach is more efficient than Andersen's pointer analysis. To the best of our knowledge, the most efficient algorithm for solving the bidirected Dyck-reachability problem may achieve linear time complexity [10] in many cases, which is significantly more efficient than the cubic lower bound [30] of Andersen's pointer analysis.

In this example, field access operations are modeled via Dyck-reachability. Similar matching properties exist in other program constructs, such as call/return matching in function invocations and reference/dereference matching in C programs. The Dyck-reachability modeling approach thus has broad applications in program analysis problems.

2.3 Motivation

This subsection will first introduce the basic idea of the bidirected Dyck-reachability solving algorithm. Then, we will compare this method with the algorithm for computing the congruence closure of EUF SMT formulas on congruence graphs, motivating our approach.

Figure 3a illustrates a bidirected graph in which the Dyck grammar requires matching parentheses. By observing a Dyck-reachable path between nodes a and c, we deduce that a and c are Dyck-reachable. Figure 3b depicts the subgraph derived from Fig. 3a by retaining only edges labeled with $\langle$. In this subgraph, nodes a and c have edges labeled with $\langle$ directed toward node b. According to

Fig. 3. Bidirected graph and its subgraph containing only *opening label* edges.

the definition of a bidirected graph, the existence of such a configuration implies that the complete edge relationship among these three nodes must include the edges shown in Fig. 3a. Moreover, Theorem 1 establishes that the bidirected Dyck-reachability relation is an equivalence relation. Specifically, if nodes a and c are Dyck-reachable, then every node Dyck-reachable from a must also be Dyck-reachable from c, and conversely, every node Dyck-reachable from c must be Dyck-reachable from a. Therefore, nodes a and c can be merged in the graph. Similar to Strongly Connected Component (SCC), a Dyck Strongly Connected Component (DSCC) [10] represents a set of mutually Dyck-reachable nodes, forming a Dyck-reachable equivalence class.

Overall, the state-of-the-art algorithm for solving bidirected Dyck-reachability [10,44] consists of two fundamental steps. The first step identifies configurations where multiple nodes have edges with identical *opening labels* directed toward a common node, as illustrated in Fig. 3b. The second step merges these nodes into the same equivalence class. The process iterates these two steps until no additional node pairs can be merged. As a result, all nodes within a final equivalence class are mutually Dyck-reachable.

By examining this process, we observe an analogy with the congruence closure algorithm used in EUF SMT solving. Specifically, edges labeled with *opening labels* in bidirected graphs correspond to uninterpreted functions in EUF. The rule for merging nodes based on bidirected Dyck-reachability corresponds to the inference of new equivalence classes via the function congruence property in the congruence closure algorithm. Consequently, the resulting DSCCs correspond to the equivalence classes derived in EUF SMT solving.

Insight: *For the bidirected Dyck-reachability problem, if we consider the nodes in the graph as variables and edges with opening label as a function relationship, each edge with an opening label can be encoded as an EUF SMT formula. The equivalence classes obtained in EUF SMT solving correspond to the sets of Dyck-reachable nodes in the bidirected graph.*

3 EUF-based Dyck-Reachability Encoding

In this section, we will present a reduction of the bidirected Dyck-reachability problem to the satisfiability problems of EUF SMT formulas. This demonstrates

that all program analysis problems that can be modeled as bidirected Dyck-reachability problem can also be encoded using EUF SMT formulas, allowing them to be solved using SMT solvers.

3.1 Reduction

We present a reduction from the bidirected Dyck-reachability problem to the satisfiability of EUF SMT formulas. We first outline the specifications of both problems, then describe the reduction function that encodes any bidirected Dyck-reachability instance into EUF SMT theory.

Bidirected Dyck-reachability problem.

- **Input:** A Dyck CFG $\mathcal{G}$, a bidirected graph $G = (V, E)$, and a pair of node (u, v) .
- **Output:** u and v are *Dyck-Reachable* or *not*.

Satisfiability of EUF SMT formulas.

- **Input:** An EUF SMT formula.
- **Output:** *SAT* or *UNSAT*

Reduction function R. The reduction function R maps each instance of the Dyck-reachability problem to an EUF SMT formula satisfiability problem. In this mapping, graph nodes correspond to formula variables, each *opening label* in Σ is represented as an uninterpreted function, and edges labeled with *opening labels* or ϵ are encoded as equations. The queried node pairs are represented as inequalities between corresponding variables.

Algorithm 1: *Encoding*$(\mathcal{G}, G)$

Input: A CFG $\mathcal{G}$, a labeled bidirected graph $G(V, E)$
Output: An EUF formula ϕ_{EUF}.

```
1 φ_EUF ← True
2 foreach (s, l, t) ∈ E do
3     if l is an opening label in G then
4         φ_EUF ← φ_EUF ∧ (s = F_l(t))
5     if l = ε then
6         φ_EUF ← φ_EUF ∧ (s = t)
7 return φ_EUF
```

As shown in Algorithm 1, the algorithm encodes the input Dyck grammar $\mathcal{G}$ and labeled graph G into an EUF SMT formula. The algorithm initializes ϕ_{EUF} to True (Line 1), then iterates over the edges E of G. For each edge (s, l, t) (Line 2), the encoding depends on the edge label: opening parenthesis l is encoded as $s = F_l(t)$ (Line 4), while ϵ is encoded as $s = t$ (Line 6). After processing all edges, the resulting formula ϕ_{EUF} contains approximately $|E|/2$ conjunctive equality constraints. The reachability query for nodes u and v corresponds to checking the satisfiability of $\phi_{\text{EUF}} \wedge (u \neq v)$, denoted as $R(\mathcal{G}, G, u, v)$. The nodes u and v are Dyck-reachable if and only if $R(\mathcal{G}, G, u, v)$ is *unsatisfiable*, completing the reduction function R.

The reduction algorithm has a time complexity of $O(|E|)$. *However, in practical applications such as field-sensitive alias analysis, a program can be encoded*

directly as an EUF formula without first being converted to an explicit graph representation. Furthermore, the EUF SMT-based approach to bidirected Dyck-reachability does not require explicit enumeration of all $|V|^2$ node pairs to determine reachability. SMT solvers utilizing the congruence closure algorithm [16,32] first compute an equivalence closure from ϕ_{EUF}, which *exactly matches* the closure derived by graph-based bidirected Dyck-reachability algorithms. Reachability queries are then handled via *incremental solving*: the base equivalence closure is computed once by solving ϕ_{EUF}, after which constraints like $(u \neq v)$ are pushed to check satisfiability for individual queries and subsequently popped to reset the solver state.

Theorem 2. *Let $\mathcal{G}$ be a Dyck grammar and $G = (V, E)$ a bidirected graph. For any nodes $u, v \in V$, u and v are bidirected Dyck-reachable if and only if the EUF formula $R(\mathcal{G}, G, u, v)$ is unsatisfiable.*

Proof. **Direction1 : When (u, v) are Dyck-reachable in graph G, then $R(\mathcal{G}, G, u, v)$ is unsatisfiable.**
When (u, v) are Dyck-reachable in G, *i.e.*, $\exists t_1, t_2, \ldots, t_{n-1} \in V.\lambda(u, t_1) \cdot \lambda(t_1, t_2) \cdots \lambda(t_{n-1}, v) \in L(\mathcal{G})$, we let p represent this reachable path of length n and $Encoding(\mathcal{G}, p)$ denote the formula generated by encoding p. We proceed by induction on the path length n. **Direction 1** holds when the nodes u and v are the same, *i.e.*, when $n = 0$. When $n = 1$, we have $\lambda(u, v) = \epsilon$, and encoding this edge yields $u = v$. This formula establishes the equivalence of u and v, indicating that $R(\mathcal{G}, G, u, v)$ is unsatisfiable.

Assume that **Direction 1** holds for $n < k$. For $n = k$, since p of length n is a reachable path, the labels on p concatenate to form a well-parenthesized string. According to the syntax of the Dyck language (Rule 2), there are two ways to construct the path p:

1. There exists a well-parenthesized path p' of length $n-2$, such that $p = l \cdot p' \cdot \bar{l}$;
2. There exist two well-parenthesized subpaths p_1 and p_2, both satisfying $|p_i| < n$ and $p = p_1 \cdot p_2$.

By the induction hypothesis, $Encoding(\mathcal{G}, p')$, $Encoding(\mathcal{G}, p_1)$, and $Encoding(\mathcal{G}, p_2)$ can all be shown to establish the equivalence of their endpoint nodes. Furthermore, by the transitivity (Axiom transitivity) and congruence (Axiom congruence) axioms, it follows that $Encoding(\mathcal{G}, p)$ forms a proof sequence for $u = v$, thus $R(\mathcal{G}, G, u, v)$ is unsatisfiable.

Direction2 : If $R(\mathcal{G}, G, u, v)$ is unsatisfiable, then (u, v) are Dyck-reachable in graph G.
When $R(a, b)$ is unsatisfiable, there exists a proof sequence for $u = v$. Based on this proof sequence, we can derive a Dyck-reachable path in the graph G from u to v. The proof process is similar to that of **Direction 1**, and the complete proof can be found in the full version of this paper [17].

3.2 Example

In Sect. 2.2, field-sensitive alias analysis was modeled via Dyck-reachability. In the previous subsection, we present a reduction from bidirected Dyck-reachability problems to EUF SMT solving, demonstrating that EUF-based methods can solve such problems. As an illustration, we now detail the corresponding EUF encoding approaches using the programs in Fig. 2.

The results from encoding the Java program in Fig. 2a using EUF SMT formulas are as follows:

$$\texttt{h} = \texttt{b} \wedge \texttt{b} = F_1(\texttt{a}) \wedge \texttt{f} = F_2(\texttt{a}) \wedge \texttt{a} = F_2(\texttt{c}) \wedge \texttt{c} = F_1(\texttt{g}) \wedge \texttt{d} = F_2(\texttt{c}) \wedge \texttt{e} = F_1(\texttt{d})$$

In this case, each program variable maps to a formula variable, and each field is modeled as an uninterpreted function to represent field accesses. Program statements are encoded as follows: assignments like `h=b` as equalities; store operations like `a. fst=b` as $\texttt{b} = F_1(\texttt{a})$; and load operations like `d=c. snd` as $\texttt{d} = F_2(\texttt{c})$. Solving the formula yields equivalence classes, *e.g.*, {`a,d`} and {`h,b,e`} (excluding trivial self-equivalences), matching the Dyck-reachability results in Fig. 2b.

4 Experiment and Evaluation

Solving bidirected Dyck-reachability with EUF SMT solvers only requires designing an appropriate encoding, leveraging their mature implementations and avoiding the need for specialized algorithms. This section evaluates the efficiency of our approach by comparing SMT-based, graph-based, and Datalog-based methods on static analysis tasks. A typical static analysis consists of two phases: program analysis, which computes the *complete analysis results* (*e.g.*, all alias relations), and query resolution, which *answers specific queries* (*e.g.*, determining whether two given variables are aliases). We evaluate all three approaches from both perspectives, addressing the following research questions:

- **RQ1**: Efficiency of analysis, *i.e.*, given a program and a static analysis task, how efficient is the SMT solver-based approach compared to the other two approaches in completing the analysis?
- **RQ2**: Efficiency of querying, *i.e.*, after completing analysis, how efficient is the SMT solver-based approach compared to the other two approaches in answering specific queries?

4.1 Experimental Setup

Baseline. We use graph-based and Datalog-based approaches as baselines. The graph-based baselines include: `FD` by Zhang *et al.* [44] with complexity $O(n+m\cdot\log m)$, and `Optimal` by Chatterjee *et al.* [10] with worst-case complexity $O(m + n \cdot \alpha(n))$. We use the open-source implementation from Krishna *et*

al. [26], which focuses on dynamic bidirected Dyck-reachability and implements both algorithms. For fairness, we compiled the code with -O3 optimization, whereas the original compilation commands in [26] used no optimization flags.

Datalog tools are also widely used in static analysis, computing relational closures by deriving new *facts* from initial *facts* according to analysis *rules*. For our experiments, we selected two well-known Datalog tools as baselines: Soufflé [15] and Bddbddb [41].

SMT solvers. SMT solvers are widely used in program analysis and verification, with EUF theory playing a central role as it bridges different theories in combined theory solving [31]. To solve EUF formulas, SMT solvers typically employ congruence closure algorithms [13,16,32,33] (core procedures are outlined in Sect. 2.1, though implementations vary). The following SMT solvers were used in our experiments for EUF formula solving:

- **Z3** [12]**:** A high-performance C++ SMT solver from Microsoft Research, is widely adopted in academia and industry. Given its significant impact, we selected Z3 for EUF SMT solving in our experiments.
- **CVC5** [6]**:** A well-known SMT solver developed in C++ by Stanford University and the University of Iowa, it demonstrated outstanding performance in the QF_UF category at SMT-COMP 2024 [38].
- **Yices** [18]**:** Yices, developed in C by SRI International, is an efficient SMT solver that excelled in the SMT-COMP 2024 competition [38], winning both the Single Query Track and the Incremental Track in the QF_UF category.
- **Plat-smt** [19]**:** Plat-smt, a Rust-based QF_UF SMT solver, employs Egg's [42] implementation for congruence closure solving and demonstrated strong performance in that category at SMT-COMP 2024 [38].
- **Egg** [42]**:** Egg is a Rust-based tool for equality saturation that uses e-graphs as its core data structure to support congruence closure for deriving equivalence classes. It innovatively introduces a *rebuilding* operation that maximizes equivalence class merging while delaying the upward search for congruent terms, significantly improving computational efficiency.

Benchmarks. The application of Dyck-reachability in static analysis is well-established [10,24,26–29,44], including the baseline tools used in our experiments. Our initial benchmarks are drawn from these prior works. For further evaluation, we added 10 widely-used open-source C programs. The transformation from program to graph/EUF formulas incurs minimal overhead, as it requires only a single pass over the intermediate representation. Benchmark details are provided below.

Table 1. Field-sensitive Alias Analysis for Java Programs

benchmark	#Locs(K)	#Nodes	#Edges	#Labels
antlr	170	23031	21353	1246
bloat	206	26656	23598	1360
chart	448	51356	44501	3132
eclipse	181	24004	21943	1346
fop	431	46253	39125	2857
hsqldb	156	21646	20271	1160
jython	216	28033	24889	1398
luindex	164	22631	20915	1228
lusearch	175	23344	21569	1275
pmd	193	24586	22522	1322
xalan	155	21574	20186	1152

Field-sensitive alias analysis for Java. Existing work on bidirected Dyck-reachability [10,26,27,44] has selected the DaCapo 2006 [8] benchmark for field-sensitive alias analysis tasks, which have been modeled as bidirected Dyck-reachability problems [44]. Table 1 shows the statistical information for the graphs, including the lines of code (#Locs in thousands), the number of nodes (#Nodes), edges (#Edges), and variable fields (#Labels) in each graph.

Table 2. Field-sensitive Alias Analysis for C Programs

benchmark	#Locs(K)	#Nodes	#Edges	#Labels
git	188	525717	630839	167
libssh2.a	27	73667	81278	346
tmux	49	154992	197807	164
vim	255	489719	612678	512
lighttpd	62	50892	60381	102
sqlite3	268	323952	405705	362
strace	94	140191	164456	92
wrk	601	369384	439726	253
darknet	28	94266	110077	352
libxml2.a	72	159308	196230	199

Field-sensitive alias analysis for C Programs. To further evaluate our approach, we selected 10 real-world C programs, including several large-scale ones, as shown in Table 2. We used lotus's [14] encoding scheme to convert these C programs into graph representations. In this encoding, labels capture not only field accesses but also pointer dereference and reference operations.

Context-sensitive data dependence analysis. Data dependency analysis focuses on identifying Def-Use chains in programs, with applications in various static analysis tasks. Context sensitivity requires distinguishing between different function calls, where a necessary condition for a valid call is the matching of call and return. Existing work models this requirement using bidirected Dyck-reachability [10,26,27]. It builds upon an approach from [40] that represents SPECjvm2008 benchmark programs [1] as bidirected graphs. In this paper, we adopt the same bidirected graph representation following [10]. Table 3 summarizes the statistical information of the graphs derived from these programs, where (#Labels) denotes the number of function calls in each program.

Table 3. Context-sensitive Data Dependence Analysis for Java Programs

benchmark	#Locs(K)	#Nodes	#Edges	#Labels
btree	5	1811	1757	801
check	10	5240	5267	2167
compiler	9	4189	4101	1646
compress	10	4375	4238	1721
crypto	21	6202	6300	2540
derby	21	6116	5948	2358
helloworld	9	4074	3969	1596
mpegaudio	40	9650	9931	3564
mushroom	2	899	809	376
parser	5	1686	1561	690
sample	2	931	834	389
scimark	10	4583	4429	1782
startup	11	5493	5367	2165
sunflow	9	3891	3792	1520
xml	47	23922	24391	9128

Evaluation method. This paper experimentally investigates the efficiency of three approaches: SMT solver-based, graph-based, and Datalog-based approaches. The input for the graph-based method is a graph derived from the compilation of the target program, and the graph models we evaluate are from [27] and [26]. To ensure a unified testing standard, the inputs for both the SMT solver-based approach and the Datalog tools are derived from this graph model.

For the SMT solver-based method, the input is a standard SMT-LIB [7] format file. We convert the existing graph file into an SMT formula using Algorithm 1, with the logic set to QF_UF. The querying functionality is implemented via incremental solving. For instance, to check the equivalence of two variables (u, v)—*i.e.*, whether nodes u and v are Dyck-reachable—the commands $(push\ 1)$, $(assert\ (not\ (=\ u\ v)))$, $(check\text{-}sat)$, and $(pop\ 1)$ are appended to the SMT-LIB file. An UNSAT result indicates that u and v are equivalent.

The experiments were conducted on a desktop running Ubuntu 22.04, equipped with an i7-9700 3.00GHz CPU and 16GB of RAM. Each experiment was performed three times, and the reported results are the averages.

4.2 Efficiency of Analysis

This section evaluates SMT-based, graph-based, and Datalog-based methods for computing all equivalence classes, using the programs in Tables 1–3. Results in Tables 4–6 distinguish parsing and solving time (denoted as parsing + solving) for graph-based and SMT tools, with gray values in parentheses showing 10,000-query times. Plat-smt solves each formula immediately upon parsing, interleaving the two phases; the total time is therefore reported. The red-highlighted numbers in the figures indicate the total time (parsing + solving) and query time for the best-performing tool in each category. Since Datalog tools such as Soufflé and Bddbddb explicitly store all relation pairs—requiring $O(n^2)$ storage for n elements in an equivalence class—their analysis and query efficiency is more than two orders of magnitude slower. We therefore omit their results in this section; detailed experimental results can be found in the full version of this paper [17].

Table 4. Results of field-sensitive alias analysis for Java programs, showing total analysis time (Parsing+Solving, ms) and total time for 10k queries (ms, in parentheses). Best results are highlighted in red. TO = Time Out (10 minutes).

benchmark	Graph based(ms)		SMT based(ms)				
	#FD	#Optimal	#Z3	#CVC5	#Yices	#Plat-smt	#Egg
antlr	29.3+10.4(6.0)	16.0+4.8 (6.1)	67.0+431.4 (380.1)	175.8+209.2 (499.3)	25.0+8.6 (28.3)	26.41 (14.8)	20.5+10.9 (4.0)
bloat	32.9+11.2(6.1)	17.7+6.1 (5.9)	75.1+558.7 (419.0)	154.9+245.7 (506.9)	27.5+9.5 (30.4)	29.82 (14.8)	24.4+13.1 (4.2)
chart	65.4+25.2(6.7)	54.9+14.2(6.2)	152.4+294.5 (679.1)	310.1+511.1 (581.8)	53.8+17.9 (30.0)	64.31 (15.0)	50.8+28.8 (4.2)
eclipse	30.7+10.4(6.1)	28.7+5.8 (6.2)	71.5+462.6 (384.6)	143.1+219.9 (505.1)	26.3+9.0 (27.7)	34.77 (14.7)	24.7+11.6 (4.0)
fop	59.4+21.6(6.6)	46.7+11.7(6.1)	138.4+1403.9 (617.7)	280.6+437.9 (539.3)	46.9+16.8 (29.5)	56.37 (15.0)	46.4+25.5 (4.0)
hsqldb	27.7+9.7 (6.0)	15.3+4.4 (6.4)	64.6+408.4 (364.5)	130.0+194.2 (484.5)	27.2+9.2 (28.7)	29.57 (15.7)	19.4+11.1 (4.2)
jython	34.7+12.4(6.1)	18.3+7.0 (6.0)	76.3+642.4 (425.9)	162.7+260.7 (512.1)	28.3+10.1 (28.8)	32.92 (14.6)	26.1+14.7 (4.2)
luindex	37.5+10.0(6.0)	14.9+5.3 (6.1)	66.5+102.4 (372.4)	136.9+205.4 (494.5)	22.6+8.5 (29.1)	28.54 (14.8)	21.4+11.6 (4.1)
lusearch	32.8+10.1(6.1)	15.8+5.4 (6.3)	69.0+107.3 (375.5)	138.3+213.9 (495.5)	26.0+8.5 (28.4)	29.91 (14.7)	21.6+11.7 (4.0)
pmd	38.9+11.2(6.0)	16.8+5.6 (5.9)	69.6+471.8 (403.7)	144.9+225.1 (508.1)	25.1+9.3 (29.2)	30.2 (15.0)	22.4+12.3 (4.3)
xalan	28.4+10.1(6.0)	21.1+4.8 (6.4)	63.7+362.5 (364.9)	129.7+193.0 (483.4)	22.6+8.3 (27.4)	26.21 (14.9)	19.3+11.1 (4.2)
Total Time	417.7+142.3 (67.7)	266.2+75.0(67.6)	914.2+5246.1(4787.4)	1906.9+2916.1 (5610.5)	331.2+115.6 (317.5)	389.0 (164)	297.0+162.3 (45.4)

The data in Tables 4–6 indicate that `Optimal` achieves the best performance in both total and solving time across nearly all benchmarks. The sole exception is in Table 5, where Plat-smt slightly outperforms `Optimal` in total time (3.29 s vs. 3.34 s). Among the SMT solvers, Yices, Plat-smt, and Egg demonstrate performance comparable to `Optimal` across the three benchmarks. Specifically, Plat-smt achieves 88% of `Optimal`'s efficiency in Table 4 (389 ms vs. 341.2 ms), while Yices reaches 65% of `Optimal`'s solving efficiency in the same table (115.6 ms vs. 75 ms) and 90% in Table 5 (1.67 s vs. 1.51 s). Compared to the second-best graph-based algorithm `FD`, Yices and Plat-smt outperform `FD` in both solving and total time in Tables 4 and 5, although they perform slightly worse than

Table 5. Results of field-sensitive alias analysis for C programs, showing total analysis time (Parsing+Solving, s) and total time for 10k queries (s, in parentheses). Best results are highlighted in red. TO = Time Out (10 minutes).

benchmark	Graph-based(s)		SMT-based(s)				
	#FD	#Optimal	#Z3	#CVC5	#Yices	#Plat-smt	#Egg
git	0.77+1.11 (0.007)	0.4+0.32 (0.007)	2.32+85.69 (1.422)	3.85+11.78 (29.286)	0.83+0.43 (0.033)	0.74 (0.016)	0.6+0.5 (0.004)
libssh2.a	0.1+0.12 (0.007)	0.06+0.03 (0.007)	0.26+7.13 (1.203)	0.49+1.15 (0.812)	0.08+0.04 (0.029)	0.09 (0.015)	0.09+0.06 (0.004)
tmux	0.24+0.3 (0.008)	0.12+0.1 (0.008)	0.61+17.51 (2.537)	1.14+3.69 (9.615)	0.21+0.1 (0.033)	0.21 (0.017)	0.23+0.14 (0.004)
vim	0.74+1.16 (0.009)	0.39+0.38 (0.009)	1.76+38.26 (7.219)	3.59+10.93 (32.387)	0.75+0.38 (0.033)	0.72 (0.018)	0.56+0.47 (0.005)
lighttpd	0.07+0.09 (0.007)	0.04+0.03 (0.006)	0.16+2.68 (0.873)	0.34+1.01 (2.204)	0.06+0.03 (0.030)	0.06 (0.015)	0.06+0.04 (0.004)
sqlite3	0.49+0.73 (0.009)	0.26+0.22 (0.008)	1.31+63.69 (4.923)	2.37+7.57 (45.547)	0.48+0.24 (0.038)	0.47 (0.017)	0.48+0.32 (0.004)
strace	0.21+0.26 (0.008)	0.1+0.08 (0.007)	0.55+0.65 (2.13)	0.95+2.86 (2.272)	0.17+0.07 (0.042)	0.17 (0.017)	0.13+0.11 (0.004)
wrk	0.54+0.75 (0.009)	0.29+0.22 (0.008)	1.4+56.88 (5.944)	2.62+8.28 (17.034)	0.54+0.25 (0.035)	0.52 (0.018)	0.5+0.32 (0.004)
darknet	0.14+0.17 (0.007)	0.07+0.05 (0.007)	0.31+3.23 (1.422)	0.6+1.77 (1.346)	0.11+0.04 (0.033)	0.12 (0.016)	0.1+0.06 (0.004)
libxml2.a	0.23+0.31 (0.008)	0.12+0.1 (0.008)	0.61+16.71 (2.467)	1.12+3.52 (8.747)	0.22+0.1 (0.034)	0.2 (0.016)	0.21+0.14 (0.004)
Total Time	3.52+5.0 (0.079)	1.83+1.51 (0.075)	9.29+292.4 (30.14)	17.08+52.57 (149.25)	3.45+1.67 (0.34)	3.29 (0.165)	2.96+2.16 (0.04)

Table 6. Results of context-sensitive data dependence analysis for Java programs, showing total analysis time (Parsing+Solving, ms) and total time for 10k queries (ms, in parentheses). Best results are highlighted in red. TO = Time Out (10 minutes).

benchmark	Graph based(ms)		SMT based(ms)				
	#FD	#Optimal	#Z3	#CVC5	#Yices	#Plat-smt	#Egg
btree	2.9+0.2 (8.6)	2.6+0.3 (5.0)	8.2+7.5 (156.4)	13.2+19.4 (512.3)	2.3+1.2 (29.4)	6.02 (17.4)	1.9+1.3 (4.3)
check	6.8+0.6 (5.9)	6.3+0.7 (5.4)	25.9+25.1 (204.7)	60.8+63.1 (575.0)	7.1+3.2 (33.5)	12.52 (14.2)	7.2+3.4 (4.4)
compiler	5.5+0.5 (6.2)	7.9+0.8 (5.3)	24.1+22.5 (187.6)	29.0+46.9 (533.6)	5.2+2.6 (30.7)	13.73 (14.2)	17.2+4.0 (4.2)
compress	5.9+0.5 (5.3)	7.9+0.8 (5.3)	13.7+19.8 (186.5)	31.6+49.5 (523.3)	5.4+2.6 (30.0)	13.82 (14.3)	7.7+2.9 (4.6)
crypto	9.1+1.0 (5.7)	8.4+1.0 (5.0)	18.2+35.7 (218.3)	45.7+76.7 (591.2)	8.0+4.2 (31.1)	16.38 (14.5)	9.1+3.6 (3.9)
derby	9.0+0.8 (5.4)	7.3+0.9 (5.1)	23.7+27.2 (214.2)	47.8+70.1 (540.6)	7.8+3.9 (31.0)	8.88 (14.7)	15.4+4.2 (4.3)
helloworld	5.7+0.4 (6.4)	4.7+0.6 (5.0)	13.3+20.1 (187.1)	31.6+44.4 (524.7)	5.1+2.5 (30.8)	5.17 (14.4)	5.4+2.3 (4.2)
mpegaudio	12.6+1.3 (6.0)	10.6+1.3 (5.1)	34.3+47.6 (259.8)	69.9+121.5 (581.7)	12.0+5.9 (30.1)	12.46 (14.5)	12.6+5.4 (4.3)
mushroom	2.3+0.2 (4.7)	0.8+0.1 (4.8)	6.3+3.6 (143.7)	12.2+10.7 (487.4)	3.4+1.3 (30.2)	0.97 (14.5)	2.5+1.7 (4.6)
parser	2.8+0.2 (5.1)	1.5+0.2 (4.9)	8.0+6.4 (156.1)	28.3+18.3 (493.3)	7.6+3.4 (34.7)	1.85 (15.4)	5.1+2.4 (6.1)
sample	1.2+0.1 (4.9)	0.8+0.1 (4.8)	6.5+3.5 (150.1)	14.6+11.8 (488.4)	3.4+1.6 (31.3)	1.02 (14.3)	1.8+1.3 (5.0)
scimark	6.2+0.5 (7.2)	4.1+0.5 (5.1)	15.5+22.0 (190.6)	53.4+52.1 (528.9)	17.4+5.9 (31.5)	5.73 (14.3)	10.8+3.2 (4.4)
startup	8.0+0.7 (6.0)	5.2+0.7 (5.2)	35.4+27.3 (208.6)	38.4+67.0 (572.8)	14.5+4.9 (30.4)	6.84 (14.4)	13.9+4.5 (4.1)
sunflow	5.5+0.5 (7.9)	3.5+0.4 (5.0)	12.7+18.2 (185.7)	30.9+42.4 (526.0)	6.4+2.9 (30.5)	5.56 (14.8)	5.4+2.1 (5.3)
xml	37.7+7.1 (6.1)	23.6+5.3 (5.7)	83.6+90.4 (452.4)	184.9+347.1 (580.3)	46.9+13.0 (30.9)	39.1 (14.7)	32.0+16.1 (4.3)
Total Time	121.04+14.43 (91.4)	95.15+13.8 (76.7)	329.33+376.9 (3101.8)	692.26+1040.98 (8059.5)	152.31+59.07 (466.1)	150.03 (220.6)	148.22+58.42 (68)

FD in Table 6 yet remain within the same order of magnitude. Notably, Egg is more than twice as fast as FD in Table 5, although it performs slightly worse on other benchmarks. In contrast, Z3 and CVC5 exhibit relatively poor performance across all benchmarks. We now analyze the factors contributing to these performance differences.

Chatterjee *et al.* [10] established that Optimal solves the bidirected Dyck-reachability problem with complexity $O(m + n \cdot \alpha(n))$, proving its optimality. In contrast, known congruence closure algorithms for EUF have complexity $O(n \cdot \log n)$ [16,32,33]. However, since encoding bidirected Dyck-reachability into EUF formulas produces formulas that contain only unary functions with a constant number of function symbols, congruence closure algorithms can achieve lower complexity under this constraint. In Sect. 5.2, we provide a brief comparison between the congruence closure implementation in Egg [42] and the

`Optimal` algorithm. We show that Egg's implementation also attains a complexity of $O(m + n \cdot \alpha(n))$ under the same restriction, and we highlight the notable similarities in their algorithmic designs.

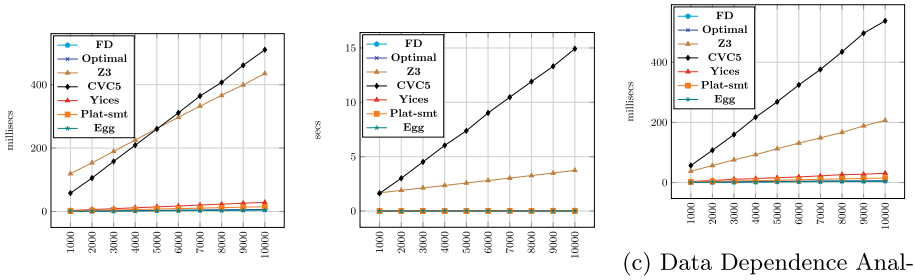
Algorithmic design choices significantly impact performance. First, regarding implementation languages, experiments on union-find sequences show that C and C++ implementations are roughly twice as fast as their Rust counterpart. This directly explains the two-fold performance gap between Egg and `Optimal`, despite their shared asymptotic complexity. Detailed results and analysis are provided in Sect. 5.1. Moreover, various tools employ distinct optimization strategies. For instance, Egg uses a *rebuilding* operation that performs unions first, followed by a single-level upward search for congruent terms, thereby lowering amortized complexity. Yices, on the other hand, flattens EUF formulas into terms of depth at most two [33]. This allows immediate equivalence detection after merging variables: once x and y are merged via $x = y$, terms such as $f(x)$ and $f(y)$ are directly recognized as equal, enabling m and n (from $m = f(x)$ and $n = f(y)$) to be merged without explicit congruence-finding steps.

Answer to RQ1: *On the tasks of field-sensitive alias analysis and context-sensitive data dependence analysis, SMT solvers such as Yices, Plat-smt, and Egg achieve performance comparable to the best graph-based algorithm,* `Optimal`*. These results demonstrate the high solving efficiency of the EUF SMT-based approach, confirming its suitability for practical static analysis.*

4.3 Efficiency of Querying

In this section, we compare the efficiency of SMT-based and graph-based methods for answering a specific type of queries (*i.e.*, the equivalence of two variables). For each program, we generated 10 query sequences with lengths ranging from 1,000 to 10,000 in increments of 1,000, where approximately 50% of the variable pairs are Dyck-reachable. In Tables 4–6, the gray numbers in parentheses report the time per 10,000 queries. Figure 4 plots the trend of query time as query sequence length increases across the three benchmarks; each point shows a tool's average query time for a given length across all programs in that benchmark. Complete results can be found in the full version of this paper [17].

We first compared the query results of the SMT solver-based methods with graph-based methods, and across tens of thousands of queries, both produced identical results, providing experimental support for Theorem 2. The results in Tables 4–6 show that Egg achieves the best performance, with speedups of up to 1.88 × (0.04 s vs. 0.075 s) compared to graph-based methods in Table 5. As shown in Fig. 4, the query time of every tool grows linearly with the query length. While Plat-smt and Yices perform slightly worse than graph-based tools, their average per-query overhead remains within the same order of magnitude. In contrast, Z3 and CVC5 are noticeably slower; nonetheless, their average per-query time remains within an acceptable range.



(a) Alias Analysis for Java (b) Alias Analysis for C (c) Data Dependence Analysis for Java

Fig. 4. Average query time for different analysis types

Answer to RQ2: *In terms of query efficiency, Egg outperforms graph-based methods. Although performance varies among solvers, the average per-query overhead remains within acceptable limits. This demonstrates the high efficiency of the EUF SMT-based approach in query processing, making it directly applicable to practical static analysis tasks.*

5 Discussion

Since the union-find operation is fundamental to congruence closure, this section presents the performance of various tools on union-find sequences, along with a concise comparison between the algorithms of the `Optimal` and Egg tools.

5.1 Experimental Results on Union-Find Sequences

This experiment evaluates the performance of various tools on Union-Find sequences, highlighting differences in implementation approaches and programming languages. For a set of 50,000 variables, we randomly generated equality equations ranging from 5,000 to 25,000 in increments of 5,000. These equations involve only variable equalities (representable as ϵ-labeled edges in graph-based tools) and contain no function applications. All results, averaged over three runs and reported in milliseconds, are presented in Table 7.

We first examine the effect of implementation language. Each tool our C++-based `unionfind`, Yices (in C), and Egg (in Rust)implements the classic Union-Find algorithm with path compression and union by rank [11]. The experiments reveal that the C and C++ implementations perform with comparable efficiency, and both outperform the Rust version by more than a factor of two.

Regarding algorithmic implementations, graph-based tools model Union-Find equations as graph edges, use reachability algorithms to identify equivalence classes, and then merge them via standard Union-Find structures, which hinders the overall efficiency. SMT solvers typically employ linked-list–based

Table 7. Results on Union-Find sequences of varying lengths, showing total analysis time (Parsing+Solving, ms). Best results are highlighted in red.

Length	#FD(C++)	#Optimal(C++)	#unionfind(C++)	#Z3(C++)	#CVC5(C++)	#Yices(C)	#Plat-smt(Rust)	#Egg(Rust)
5000	15.16+8.11	6.13+2.95	2.12+1.81	22.53+10.11	33.66+50.97	6.02+1.48	7.05	1.47+3.23
10000	20.65+15.13	11.07+7.56	4.23+3.33	48.0+21.24	64.53+123.3	12.45+2.88	13.64	3.49+7.37
15000	31.02+23.93	16.8+11.65	6.72+5.02	57.48+33.49	120.19+198.31	46.99+5.13	18.02	5.4+10.74
20000	46.79+33.15	20.86+17.59	8.64+5.95	70.39+49.9	126.58+285.8	22.26+6.01	23.88	7.68+13.07
25000	49.71+43.27	25.87+22.2	10.79+7.03	80.58+76.83	151.88+375.4	29.47+6.57	31.76	10.91+16.2
Total Time	163.34+123.59	80.73+61.96	32.49+23.13	278.98+191.57	496.84+1033.78	117.20+22.06	94.37	28.95+50.61

Union-Find algorithms to support backtracking, forgoing path compression and incurring $O(n \log n)$ complexity—leading to degraded performance. In contrast, Yices adopts a hybrid strategy: it applies the optimized Union-Find algorithm for processing pure equalities and switches to a linked-list representation for congruence-derived equivalence classes to enable backtracking, thereby achieving the highest efficiency.

5.2 Comparison of `Optimal` and Egg

This section provides a concise comparison between the congruence-closure algorithms in `Optimal` and Egg. The core procedures of the bidirected Dyck-reachability and congruence-closure algorithms are essentially identical, as both consist of two main steps noted in Sect. 2.3: (1) merging reachable/equivalent nodes, *e.g.*, via $union(a, b)$; and (2) identifying newly reachable/equivalent nodes by examining the predecessor/parent sets of the merged nodes. The algorithm repeats these steps until a fixed point is reached, thereby computing the congruence closure.

The basic operations include $union(a, b)$, merging the predecessor/parent sets of $find(a)$ and $find(b)$, and searching for congruent terms in the merged set. Both `Optimal` and Egg store predecessor/parent sets using maps, where elements are representatives of equivalence classes. Because the number of edge-label/function types is constant, these sets also have constant size—note, however, that if functions with arity greater than one are allowed, the size of parent sets can no longer be bounded by a constant. Consequently, both merging the parent sets and searching for congruent terms require only $O(1)$ time.

`Optimal` uses a worklist that stores pending unions, processes them each iteration, and adds newly discovered equivalent nodes to the worklist for the next round. This corresponds to Egg's *rebuilding* operation, which accumulates multiple unions and then performs a batch upward search for congruent terms via the `rebuild` function only one level upwarditerating to completion. Rebuilding thereby avoids many redundant upward merges, significantly accelerating congruence-closure computation.

In both algorithms, nodes in the worklist to be processed are preprocessed by applying *find* to obtain their equivalence-class representatives, eliminating duplicates and redundant operations. Chatterjee *et al.* [10] proved (Lemma 3.4) that `Optimal` executes $O(m)$ union-find operations. A similar argument shows

Egg also uses $O(m)$ union-find operations. Moreover, because the number of edge-label/function types is constant, a sparse graph with $m = O(n)$ can be constructed in $O(m)$ preprocessing time [10]—a step required by Egg as well. Hence, both algorithms achieve a worst-case time complexity of $O(m+n\cdot\alpha(n))$.

Although designed for different problem domains, the two algorithms exhibit a striking consistency in their underlying principles. This consistency is formally revealed by our proposed encoding of bidirected Dyck-reachability into EUF formulas, as introduced in this paper.

6 Conclusion and Future Work

This paper presents a novel static analysis approach that leverages SAT/SMT solvers. We demonstrate that the EUF theory can encode all problems solvable by bidirected Dyck-reachability, thereby offering a unified framework for various static analysis tasks. Empirically, our SMT-based method achieves efficiency comparable to state-of-the-art graph-based algorithms, with some solvers exhibiting superior query performance. The combined ease of modeling program semantics in EUF and the efficiency of modern solvers underscore the practical viability of this SMT-based paradigm. Further exploration of this paradigm could be a promising direction for future research.

In the future, we plan to further explore the application of SAT/SMT solvers to static analysis. This includes investigating a broader range of problems solvable by EUF SMT theory, examining connections between other SMT theories and static analysis problems, and exploring the potential of SMT solvers as general-purpose static analysis tools.

Data Availability. Our artifact is available at the following URL: https://zenodo.org/records/18630949

Acknowledgments. This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429 and 62032024).

References

1. SPECjvm2008 Benchmark Suit (2008). http://www.spec.org/jvm2008/
2. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. Math. Ann. **99**(1), 118–133 (1928)
3. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**(3), 137 (1976)
4. Andersen, L.O.: Program analysis and specialization for the c programming language (1994)
5. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. (CSUR) **51**(3), 1–39 (2018)
6. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 415–442. Springer (2022)

7. Barrett, C., et al.: The SMT-lib standard: version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK), Vol. 13, p. 14 (2010)
8. Blackburn, S.M., et al.: The DaCapo benchmarks: java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, And Application, pp. 169–190 (2006)
9. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2329–2344 (2017)
10. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal Dyck reachability for data-dependence and alias analysis. Proc. ACM Program. Lang. **2**(POPL), 1–30 (2017)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
12. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
13. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM (JACM) **52**(3), 365–473 (2005)
14. Lotus developers: lotus, program analysis and verification framework. https://github.com/ZJU-Automated-Reasoning-Group/lotus
15. Developers, S.: Soufflé, logic defined static analysis. https://souffle-lang.github.io/
16. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM (JACM) **27**(4), 758–771 (1980)
17. Du, Y., et al.: EUF-based solving DYCK-reachability with applications to static analysis (2026). https://doi.org/10.5281/zenodo.18871949
18. Dutertre, B., De Moura, L.: The YICES SMT solver. Tool paper at **2**(2), 1–2 (2006). http://yices.csl.sri.com/tool-paper.pdf
19. Ewert, D.: platsat (2024), toy 'QF_UF' SMT solver written in Rust. https://github.com/dewert99/plat-smt
20. Habib, A., Pradel, M.: How many of all bugs do we find? A study of static bug detectors. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 317–328 (2018)
21. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Addison-Wesley Longman Publishing Co., Inc (1969)
22. Jin, D., Meredith, P.O., Lee, C., Roşu, G.: JavaMOP: efficient parametric runtime monitoring framework. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 1427–1430. IEEE (2012)
23. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 194–206 (1973)
24. Kjelstrøm, A.H., Pavlogiannis, A.: The decidability and complexity of interleaved bidirected DYCK reachability. Proc. ACM Program. Lang. **6**(POPL), 1–26 (2022)
25. Koutris, P., Deep, S.: The fine-grained complexity of CFL reachability. Proc. ACM Program. Lang. **7**(POPL), 1713–1739 (2023)
26. Krishna, S., Lal, A., Pavlogiannis, A., Tuppe, O.: On-the-fly static analysis via dynamic bidirected DYCK reachability. Proc. ACM Program. Lang. **8**(POPL), 1239–1268 (2024)
27. Li, Y., Satya, K., Zhang, Q.: Efficient algorithms for dynamic bidirected DYCK-reachability. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022)

28. Li, Y., Zhang, Q., Reps, T.: On the complexity of bidirected interleaved DYCK-reachability. Proc. ACM Program. Lang. **5**(POPL), 1–28 (2021)
29. Li, Y., Zhang, Q., Reps, T.: Fast graph simplification for interleaved-DYCK reachability. ACM Trans. Program. Lang. Syst. (TOPLAS) **44**(2), 1–28 (2022)
30. Mathiasen, A.A., Pavlogiannis, A.: The fine-grained and parallel complexity of andersen's pointer analysis. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
31. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. (TOPLAS) **1**(2), 245–257 (1979)
32. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM (JACM) **27**(2), 356–364 (1980)
33. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. **205**(4), 557–580 (2007)
34. Reps, T.: Program analysis via graph reachability. Inf. Softw. Technol. **40**(11–12), 701–726 (1998)
35. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61 (1995)
36. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. ACM SIGSOFT Softw. Eng. Notes **30**(5), 263–272 (2005)
37. Shi, Q., et al.: 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, pp. 693–706 (2018)
38. SMT-COMP organizers: 2024 SMT competition results. Online Resource (2024). https://smt-comp.github.io/2024/results/
39. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. ACM SIGPLAN Notices **41**(6), 387–400 (2006)
40. Tang, H., et al.: Summary-based context-sensitive data-dependence analysis in presence of callbacks. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 83–95 (2015)
41. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp. 131–144 (2004)
42. Willsey, M., et al.: Egg: fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021)
43. Yan, D., Xu, G., Rountev, A.: Demand-driven context-sensitive alias analysis for java. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 155–165 (2011)
44. Zhang, Q., Lyu, M.R., Yuan, H., Su, Z.: Fast algorithms for DYCK-CFL-reachability with applications to alias analysis. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 435–446 (2013)