Dis/Equality Graphs

GEORGE ZAKHOUR, University of St. Gallen, Switzerland
PASCAL WEISENBURGER, University of St. Gallen, Switzerland
JAHRIM GABRIELE CESARIO, University of St. Gallen, Switzerland
GUIDO SALVANESCHI, University of St. Gallen, Switzerland

E-graphs are a data structure to compactly represent a program space and reason about equality of program terms. E-graphs have been successfully applied to a number of domains, including program optimization and automated theorem proving. In many applications, however, it is necessary to reason about disequality of terms as well as equality. While disequality reasoning can be encoded, direct support for disequalities increases performance and simplifies the metatheory.

In this paper, we develop a framework independent of a specific implementation to formally reason about e-graphs. For the first time, we prove the equivalence of e-graphs to the reflexive, symmetric, transitive, and congruent closure of the equivalence relation they are expected to encode. We use these results to present the first formalization of an extension of e-graphs that directly supports disequalities and prove an analytical result about their superior efficiency compared to embedding techniques that are commonly used in SMT solvers and automated verifiers. We further profile an SMT solver and find that it spends a measurable amount of time handling disequalities.

We implement our approach in an extension to egg, a popular e-graph Rust library. We evaluate our solution in an SMT solver and an automated theorem prover using standard benchmarks. The results indicate that direct support for disequalities outperforms other encodings based on equality embedding, confirming the results obtained analytically.

CCS Concepts: • **Theory of computation** → *Equational logic and rewriting*; *Program verification*; • **Computing methodologies** → *Theorem proving algorithms*.

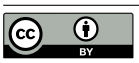
Additional Key Words and Phrases: Automated Theorem Proving, E-Graphs, Disequalities

ACM Reference Format:
George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proc. ACM Program. Lang.* 9, POPL, Article 77 (January 2025), 24 pages. https://doi.org/10.1145/3704913

1 Introduction

E-graphs [Nelson 1980] have become an integral data structure for automated reasoning. They enable representing equivalence relations of expressions that enjoy congruence, i.e., if two expressions e_0 and e_1 are equivalent, then any expressions constructed by applying the same operations to e_0 and e_1 remain equivalent. They have found applications in a variety of domains, including program optimization [Tate et al. 2009], program synthesis [Briggs and Panchekha 2022], and decision procedures [Barbosa et al. 2022; de Moura and Bjørner 2008]. They are a foundational data structure for performing *equality saturation* [Pal et al. 2023], a technique where expressions in a

Authors' Contact Information: George Zakhour, University of St. Gallen, St. Gallen, Switzerland, george.zakhour@unisg.ch; Pascal Weisenburger, University of St. Gallen, St. Gallen, Switzerland, pascal.weisenburger@unisg.ch; Jahrim Gabriele Cesario, University of St. Gallen, St. Gallen, Switzerland, jahrimgabriele.cesario@unisg.ch; Guido Salvaneschi, University of St. Gallen, St. Gallen, Switzerland, guido.salvaneschi@unisg.ch.



ACM 2475-1421/2025/1-ART77
https://doi.org/10.1145/3704913

program are systematically rewritten to explore equivalent forms. By representing expressions in a compact and structured manner, e-graphs enable the exploration of a vast solution space while avoiding redundant computations.

E-Graphs and Disequalities in Automated Reasoning. Enabling efficient reasoning about equalities makes e-graphs crucial for automated theorem proving. Yet, while e-graphs focus on handling equalities, many real-world problems involve a combination of both equalities and disequalities, such as SMT (Satisfiability Modulo Theories) and inductive theorem provers.

SMT solvers interpret formulae within predefined formal theories in first-order logic with equality. They delegate solving the boolean structure of such a formula to a SAT (Boolean Satisfiability) solver. When lifting the interpretation found by the SAT solver back into the SMT theory, SMT solvers must handle equalities for variables to which the SAT solver assigned *true* and disequalities for those assigned *false*.

Some SMT theories, such as the widely-used `ArraysEx` theory for arrays, expose disequalities to the developer in the axioms of the theory. For instance, modifying an array at index i does not affect the value at index j if i and j are *not equal*. Different methods exist for encoding disequalities into e-graphs. We show that a common encoding found in SMT solvers lends itself to easy integration with the underlying SAT solver and can use an off-the-shelf e-graph data structure, but is not the most efficient encoding of disequalities in terms of e-graph operations.

In inductive theorem provers, equalities and disequalities naturally arise in handling pattern matches. For example, to prove any property on the following code snippet, the proof assumes that $x = P_0$ holds in the first branch and $x \neq P_0$ holds in the second:

```
1 cases x of
2   P0 ⇒ e0
3   _  ⇒ e1
```

Integrating disequalities enables the system to prove a goal by absurdity (*ex falso quodlibet*) by deriving contradictions in the set of hypotheses. These contradictions are more apparent when equalities and disequalities are used together as opposed to using equalities alone.

In summary, automated proving would benefit from a unified framework that reasons about both equalities and disequalities. Surprisingly, a direct approach to support disequalities has been proposed since the early days of e-graphs. Nelson [1980] observes that *"Given a disequality $t \neq u$, we must add structure to the graph, that if the nodes representing t and u ever become equivalent, the program will detect it. The obvious way to do this is to store with the root of each equivalence class a circular list of nodes that are forbidden to join the class."* Yet, recent approaches [Barbosa et al. 2022; de Moura and Bjørner 2008] rather *embed* disequalities and – instead of a direct treatment – they assign the equality to $\bot$ (Section 2.2).

Introducing a Comprehensive Formalism for E-Graphs. The main contribution of this work is a novel and complete formal reasoning framework for e-graphs. Previous work is rather informal and tailored to specific use cases like SMT solving [de Moura 2008], or focused on concrete imperative implementations with long and complex proofs [Nelson 1980]. To the best of our knowledge, our framework is the first to formally prove correctness and cost properties for e-graphs and extensions that support disequalities. Our formalism is extendable with new e-graph features and can serve as a formal foundation for future theoretical developments on e-graphs, including extensions like inequalities (e.g., greater-than or less-than relations that enable reasoning by induction) and the formalization of the equality saturation algorithm.

Contributions. First, in Section 3.1, we present the formal framework use it to prove the equivalence of e-graphs to the reflexive, symmetric, transitive, and congruent closure of the equivalence relation they are expected to encode. Second, in Section 3.2, we use the framework to reason about three embedding techniques that enable reasoning about disequalities. Third, in Section 3.3, we generalize e-graphs to *die-graphs*, which add support for disequality reasoning directly in the structure of the e-graph. In die-graphs, equality is represented as in e-graphs. Disequality, on the other hand, being a symmetric relation between two e-nodes, is thus represented as an undirected *disequality edge* between two e-classes in the e-graph. Using the framework, we prove that die-graphs find any contradiction between the equality relation encoded by an e-graph and a set of disequalities. We further prove an analytical result about their superior efficiency compared to embedding techniques used in widespread SMT solvers and automated verifiers. Fourth, in Section 4, we describe our implementation of DieGraph on top of the egg e-graphs library [Willsey et al. 2021]. Finally, in Section 5, we empirically evaluate the performance of embedding techniques and disequality edges. For our empirical analysis, we use SMT solving and theorem proving as case studies for populating e-graphs with realistic data, e.g., existing SMT benchmarks. As our focus is on the e-graph data structure rather than proving techniques, our analysis is not targeted specifically to SMT solving.

In summary, this paper provides the following contributions.

- We provide a formalism for e-graphs, which also accounts for disequalities, and prove both the basic e-graph data structure and its extensions for disequalities correct.
- We use the theoretical framework to compare different solutions to treat disequalities in e-graphs and show that, analytically, direct support for disequalities performs better.
- We implement our approach in DieGraph, an efficient e-graph library based on the popular egg e-graph library [Willsey et al. 2021] that natively supports disequalities.
- We evaluate our approach in two case studies based on SMT solving and theorem proving comparing two different approaches to support disequalities in e-graphs, showing that (1) tracking disequalities greatly benefits automated reasoning and (2) one of the approaches performs consistently better throughout the experiments.
- We profile the Z3 SMT solver [de Moura 2008] on the SMT-LIB EUF benchmarks [Barrett et al. 2015], finding that it spends non-negligible amount of time time manipulating disequalities. Our evaluation suggests that this time can be optimized by using disequality edges as opposed to Z3's equality embedding.

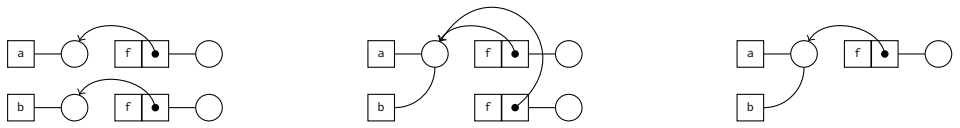
2 Equality and Disequality Graphs

In this section, we introduce e-graphs and their operations, and explain the existing approaches to support reasoning about disequalities.

2.1 E-Graphs

E-graphs compactly represent the equivalence of syntactically different terms according to an equivalence relation, such as program equivalence, e.g., the programs reduce to the same value according to the dynamic semantics of a programming language [Plotkin 2004].

The E-Graph Data Structure. An e-graph is a bipartite graph, i.e., a graph with two disjoint sets of unconnected vertices: *e-nodes* represent expressions that are either atoms or function applications, and *e-classes* represent sets of equivalent expressions. An e-graph – without support for disequalities – contains two kinds of edges. (1) An edge from an e-class to an e-node means that the e-node belongs to the e-class, i.e., the e-node is equivalent to every other e-node that belongs to the e-class. (2) An edge from an argument of an e-node to an e-class encodes congruence, i.e., equivalent



(a) Initial State. (b) Intermediate State During Merging. (c) Final Merged State.

Fig. 1. E-Graphs structure and basic operations.

expressions can be constructed by replacing any argument with any member of the e-class that the argument points to.

E-graphs focus on dynamically discovering and maintaining equivalences between expressions, making it efficient to check whether two expressions are equivalent. The process of dynamically discovering and maintaining equivalences involves managing the correct set of edges and nodes in the graph. Initially, each unique expression is represented as an individual e-node, and each e-node is placed in its own e-class. As new equivalences are discovered through algebraic transformations or logical simplifications – i.e., two e-classes are found to be equivalent – the e-graph merges the respective e-classes into a single e-class, ensuring that all semantically equivalent expressions are represented together.

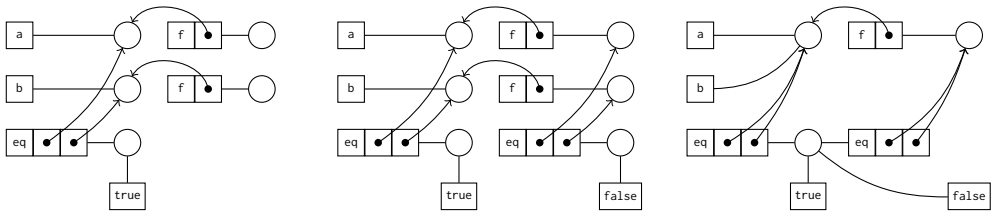
For example, Figure 1 shows an e-graph with two atoms `a` and `b` and a function `f` with arity 1. Rectangles denote e-nodes and circles denote e-classes. We visualize the two kinds of edges by representing the edge between e-class and e-node as undirected and the edge between an argument of an e-node and an e-class as directed. Initially, the e-nodes for `a`, `b`, `f(a)` and `f(b)` are in distinct e-classes (Figure 1a). Upon equating `a` and `b`, both atoms become members of the same e-class (Figure 1b). Crucially, as a result, the e-graph discovers that the e-nodes representing `f(a)` and `f(b)` are identical and thus equates their e-classes (Figure 1c).

Equality Saturation. Equality saturation [Pal et al. 2023] is a key technique in e-graphs used to discover equivalent expressions by exhaustively applying a predefined set of equivalence rules until no further progress can be made. As equality saturation is computationally expensive, research has focused on optimizing this process [Willsey et al. 2021].

The procedure starts with an initial e-graph and proceeds by iteratively applying equivalence rules to generate new e-nodes. Whenever two e-classes are found to be equivalent – i.e., they contain overlapping e-nodes after the introduction of new ones – they are merged into a single e-class. Ultimately, the process may reach a *saturated state* where no more equivalence rules can be applied. In practice, however, e-graphs are typically not fully saturated since this process is resource-intensive and is not guaranteed to terminate.

For example, saturating the e-graph of Figure 1c with the equivalence rule $\forall x.\texttt{f}(x) \equiv x$ equates the terms `a` with `f(a)`, `b` with `f(b)`, `f(a)` with `f(f(a))`, etc. Thus in the fully saturated graph there are two e-classes: the first containing the atom e-node representing `a` and the function application e-node $\texttt{f}(\cdot)$ with an edge to its own e-class representing `f(a)`, `f(f(a))`, etc; and the second similarly containing `b` and `f(b)`, `f(f(b))`, etc.

Applications of E-Graphs. The ability to compactly reason about the equivalence of expressions is crucial in several domains. In compiler optimizations, e-graphs help to identify and merge equivalent code fragments, reducing redundancy and selecting more efficient representations among equivalent programs. In automated theorem proving, they simplify expressions and discover equivalences, aiding progress towards proof goals. In symbolic computation, they enable the manipulation and



(a) Equated Terms. (b) Equated and Disequated Terms. (c) Saturated.

Fig. 2. Equality Embedding in E-Graphs.

simplification of algebraic expressions. In program analysis, e-graphs track and merge equivalent program states or expressions, supporting tasks such as optimization, verification, and refactoring.

2.2 Die-Graphs

We present the two prevalent methods to extend the functionality of e-graphs to handle disequalities: different variants of *equality embeddings* and *disequality edges*.

Equality Embedding. A straightforward *equality embedding* (EE) technique reifies the equality relation constructed by the e-graph explicitly into the term language of the e-nodes. We first introduce a special symbol `eq` into the language to represent equality and two symbols `true` and `false` to represent truth and falsehood. Two terms t_1 and t_2 are equal if the terms $\texttt{eq}(t_1, t_2)$ and `true` are equal, i.e., in the same class. And t_1 and t_2 are unequal if $\texttt{eq}(t_1, t_2)$ and `false` are equal.

Checking whether the given equalities and disequalities are consistent, i.e., that no contradiction exists, amounts to simply checking whether `true` and `false` are not equal. Yet, this check only yields the correct result after saturating the e-graph on the following rules: (1) $\forall x, y.\, \texttt{eq}(x, y) \equiv \texttt{true} \longrightarrow x \equiv y$ to lift the embedded equality to e-graph equivalence, (2) $\forall x, y.\, \texttt{eq}(x, y) \equiv \texttt{false} \longrightarrow \texttt{eq}(y, x) \equiv \texttt{false}$ to account for symmetry of disequality, and, (3) $\forall x, y.\, \texttt{eq}(\texttt{eq}(x, y), \texttt{false}) \equiv \texttt{false} \longrightarrow x \equiv y$ to account for double negation elimination. Further, when adding $\texttt{eq}(t_1, t_2) \equiv \texttt{false}$ to the e-graph, we need to also add the equalities $\texttt{eq}(t_1, t_1) \equiv \texttt{true}$ and $\texttt{eq}(t_2, t_2) \equiv \texttt{true}$. This is needed to discover equalities of disequated terms in the hopes of merging `true` and `false` when a contradiction exists. Thus, we further saturate on (4) $\forall x, y.\, \texttt{eq}(x, y) \equiv \texttt{false} \longrightarrow \texttt{eq}(x, x) \equiv \texttt{true}$ and (5) $\forall x, y.\, \texttt{eq}(x, y) \equiv \texttt{false} \longrightarrow \texttt{eq}(y, y) \equiv \texttt{true}$.

Overall, while EE can use an off-the-shelf e-graph, it comes at the cost of performing saturation to ensure consistent results when querying the e-graph.

Figure 2a shows an EE of the e-graph in Figure 1a after equating `a` and `b` – but before saturation. Note that – while the equality $\texttt{eq}(\texttt{a}, \texttt{b}) \equiv \texttt{true}$ is already encoded in the e-graph – this e-graph has not yet equated the terms `a` and `b`, i.e., the data structure does not yet contain the equality $\texttt{a} \equiv \texttt{b}$.

Upon disequating `f(a)` and `f(b)` on the e-graph of Figure 2a, we obtain the one shown in Figure 2b. The state of the e-graph hints at a contradiction as it states that $\texttt{eq}(\texttt{f}(\texttt{a}), \texttt{f}(\texttt{b})) \equiv \texttt{false}$ and $\texttt{eq}(\texttt{a}, \texttt{b}) \equiv \texttt{true}$ This is why saturation is crucial. The saturated graph is shown in Figure 2c, in which `f(a)` and `f(b)` are merged – following standard e-graph merging described in Figure 1c. As now `f(a)` and `f(b)` are treated equal, the disequality $\texttt{eq}(\texttt{f}(\texttt{a}), \texttt{f}(\texttt{b})) \equiv \texttt{false}$ that we added to the graph is equivalent to $\texttt{eq}(\texttt{f}(\texttt{a}), \texttt{f}(\texttt{a})) \equiv \texttt{false}$. Saturation further creates the node $\texttt{eq}(\texttt{f}(\texttt{a}), \texttt{f}(\texttt{a})) \equiv \texttt{true}$, which leads to `true` and `false` being merged into the same class, i.e., the e-graph can report the contradiction.

Optimized Equality Embedding. Instead of checking whether `true` and `false` are in the same class, as EE does, we can check whether $\texttt{eq}(t, t) \equiv \texttt{false}$ exists in the e-graph. This check is less

(a) Disequated Terms. (b) Equated and Disequated Terms.

Fig. 3. Negated Equality Embedding in E-Graphs.

efficient than the simple one in EE, but it reduces the amount of the performance-critical saturation performed in EE. In fact, this optimization exploits the key observation that EE's second saturation rule will have searched for such an equality already to guarantee that `true` ≡ `false`. With this check, the only saturating rules from EE that are needed for this *optimized equality embedding* (OEE) are $\forall x, y.\, \texttt{eq}(x, y) \equiv \texttt{true} \rightarrow x \equiv y$ to lift the embedded equality to e-graph equivalence and $\forall x, y.\, \texttt{eq}(\texttt{eq}(x, y), \texttt{false}) \equiv \texttt{false} \rightarrow x \equiv y$ to account for double negation elimination.

For the simple example of EE in Figure 2c, the same steps still apply to OEE, except that the condition to detect a contradiction in Figure 2 is that $\texttt{eq}(\texttt{f}(\texttt{a}), \texttt{f}(\texttt{a})) \equiv \texttt{false}$ is in the e-graph.

Negated Equality Embedding. EE and OEE need to saturate as user code might add equalities of the form $\texttt{eq}(t_1, t_2) \equiv \texttt{true}$ and the `eq` relation should faithfully represent equality. Instead, saturation can be avoided by extending the e-graph term language with a symbol that is not accessible to user code, ensuring that terms containing this symbol are added only when the embedding itself introduces them. This technique exploits the observation that the embeddings that are ever added are of the form $\texttt{eq}(t_1, t_2) \equiv \texttt{false}$. In other words, we are only interested in embedding disequality. To this end, in this *negated equality embedding* (NEE) approach, the symbol `ne` is used to reify the disequality relation. Two terms t_1 and t_2 are unequal if the term $\texttt{ne}(t_1, t_2)$ exists in the e-graph.

Checking whether the given equalities and disequalities are consistent, i.e., that no contradiction exists, amounts to checking whether $\texttt{ne}(t, t)$ is contained in the e-graph for any term t.

Figure 3a shows the e-graph in Figure 1a after disequating $\texttt{f}(\texttt{a})$ and $\texttt{f}(\texttt{b})$. Upon equating `a` and `b`, we obtain the e-graph shown in Figure 3b. The state of the e-graph shows a contradiction as it contains $\texttt{ne}(\texttt{a}, \texttt{a})$.

Note that `ne` only ever connects two e-classes or one e-class to itself in the case of a contradiction. Hence, instead of storing `ne` e-nodes, we can directly encode the disequality relation as an edge between two e-classes, which we describe next.

Disequality Edges. A more direct approach to supporting disequalities in e-graphs incorporates disequality edges (DE) – originally called "forbid lists" – directly into the e-graph data structure [Nelson 1980]. DE are class-to-class edges that explicitly express that the two e-classes may not be merged, i.e., that their members are unequal.

For example, when disequating $\texttt{f}(\texttt{a})$ and $\texttt{f}(\texttt{b})$ on the previously described e-graph of Figure 1a, with DE, we obtain the one shown in Figure 4a. The e-graph encodes that $\texttt{f}(\texttt{a}) \neq \texttt{f}(\texttt{b})$ by the DE between the e-classes of $\texttt{f}(\texttt{a})$ and $\texttt{f}(\texttt{b})$. Upon equating `a` and `b` on Figure 4a we obtain the one shown in Figure 4b. In this die-graph it is immediately possible to detect the contradiction, as the e-graph contains a disequality self-loop (denoted by the dashed edge) on one of its e-classes.

Disequalities in the Wild. We observe that common SMT solver implementations, like Z3 [de Moura and Bjørner 2008] or cvc5 [Barbosa et al. 2022] use embeddings, whereas OpenSMT [Bruttomesso et al. 2010] uses DE. We believe that embeddings are popular in SMT solvers because they make it straightforward to integrate the solver for the theory with the underlying SAT solver. These

(a) Consistent State. (b) Contradiction.

Fig. 4. Disequality Edges in E-Graphs.

tools often treat the equality relation encoded by the e-graph to be the logical assignment that the SAT solver outputs. Yet, current state-of-the-art high-performance e-graph libraries like egg do not provide built-in support for disequalities in the form of DE. Encoding equalities manually using EE, however, is possible of course. We elaborate on the design choices of SMT solvers in Section 6.

3 Formal Reasoning for Dis/Equality Graphs

In this section, we develop a formalization of e-graphs and extensions that support disequalities.

3.1 E-Graphs

E-graphs compactly represent a set of equalities among terms. Throughout this section, we use *term* to refer to a tree of function applications. The function is a symbol chosen from a finite alphabet Σ. Each function is given an arity specified by the $arity : \Sigma \to \mathbb{N}$ function. We assume every function application in a term respects the function's arity. Formally, a term is drawn from the set $T = \bigcup_i T_i$ where $T_0 = \emptyset$ and $T_{i+1} = \bigcup_{f \in \Sigma} \{f(t_1, \cdots, t_n) \ : \ arity(f) = n, t_1 \in T_i, \ldots, t_n \in T_i\}$.

In Definition 3.1 we define e-graphs as a triple of e-classes, e-nodes, and node–class edges.

Definition 3.1. An e-graph G is a triple (V_c, V_n, E) such that:

(1) a set of e-classes $V_c \subseteq C$, where C is countably infinite,
(2) a set of e-nodes $V_n \subseteq \bigcup_{f \in \Sigma} \{f\} \times V_c^{arity(f)}$,
(3) a total function $E : V_n \to V_c$ representing e-class membership edges.

In our definition, e-classes can be represented using any object chosen from a countably infinite universe. This assumption is needed since adding terms to the e-graph results in adding each subterm to a freshly created e-class, i.e., in the worse case, there are as many e-classes as there are terms. Since terms are countably infinite, e-classes must be too. Moreover, we encode the edges as a total function whose domain are e-nodes and whose codomain are e-classes since all e-nodes must belong to one and only one e-class.

Notation. We use the s meta-variable for any symbol, the a, b, c meta-variables for symbols in Σ with arity 0, and f, f_1, f_2 for those with a strictly positive arity. Terms are denoted with t, t_1, t_2, e-nodes with n, n_1, n_2, and e-classes with c, c_1, c_2. Instead of using the notation $(f, c_1, \cdots, c_n)$ or (a) for terms as suggested by Definition 3.1, we will use the more intuitive notation $f(c_1, \cdots, c_n)$ and a, respectively.

We use $[e_1/e_2]$ to denote substitution that replaces e_2 with e_1, σ ranges over substitutions, and $e\sigma$ denotes the application of the substitution σ to the expression e. If A is a set of expressions, $A\sigma$ denotes the set of all expressions in A applied to σ. If f is a function, we use $f\sigma$ to denote the function that behaves as f with its domain and codomain applied to σ.

We use $[f(x) \mapsto g(x) \ : \ x \in A]$ for the function with domain being the pre-image of $f(A)$ w.r.t. f and mapping every $f(x)$ to $g(x)$. When A is a singleton set, we use $[f(x) \mapsto g(x)]$. We use $f \downarrow_A$ to be the function f with its domain restricted to A, i.e., we assume that A is a subset of the

domain of f. If f and g are functions with disjoint domains and codomains, we use the $f \cup g$ for the function that acts like f and g.

Moreover, for an equivalence relation $\sim$, we use the notation $[x]_\sim$ for the canonical representative of the equivalence class containing x. Given a set of pairs $\mathcal{E}$, we denote the reflexive, symmetric, and transitive closure with $\mathcal{E}^*$, which is an equivalence relation.

3.1.1 Operations. We now formally define the operations on the e-graph based on the previous concepts and definitions.

Lookup Term's Class. The *find* operation finds the e-class of a term t in an e-graph g, which we abbreviate with $g[t]$. The operation may fail with $\bot$ as it may be given a term which was never added to the e-graph. The *find* operation has the signature $G \times T \to V_c \cup \{\bot\}$ and is defined as follows:

Definition 3.2 (find).

$$g[t] = \begin{cases} E(x) & \text{if } t = a \wedge arity(a) = 0 \wedge a \in V_n \\ E(f(g[t_1], \ldots, g[t_n])) & \text{if } t = f(t_1, \ldots, t_n) \wedge f(g[t_1], \ldots, g[t_n]) \\ \bot & \text{otherwise} \end{cases}$$

The term membership test $t \in g$ can be defined to be $g[t] \neq \bot$.

Add Terms. Adding a term to an e-graph requires flattening the term by adding all the subterms first and replacing them with their classes in the parent term. The function's signature is $G \times T \to G$, which we abbreviate with $g \cup t$, and it is defined as follows:

Definition 3.3 (add).

$$\frac{t \in g}{g \cup t = g} \qquad \frac{\begin{array}{c} s \in \Sigma \qquad n = arity(s) \qquad s(t_1, \ldots, t_n) \notin g \\ g' = g \cup t_1 \cup \cdots \cup t_n = (V_c', V_n', E') \\ c \notin V_c' \qquad t' = s(g'[t_1], \ldots, g'[t_n]) \end{array}}{g \cup s(t_1, \ldots, t_n) = (V_c' \cup \{c\}, V_n' \cup \{t'\}, E' \cup [t' \mapsto c])}$$

Observe that the second rule also covers constants of arity 0. Now we can state *add*'s correctness lemma:

Lemma 3.4 (Terms are not forgotten after being added). *Let g be an e-graph and $t_0, \ldots, t_n$ be a sequence of terms, then for every i, $t_i \in g \cup t_0 \cup \cdots \cup t_n$.*

Merge Classes. An equality is asserted by merging two e-classes. Therefore, when equating two terms, they must first be added using *add* and later their classes can be merged using *merge*. We abbreviate the *merge* function with $g \cup \{c_1 = c_2\}$ and, by abuse of notation, we also use the abbreviation $g \cup \{t_1 = t_2\}$ for $g' \cup \{g'[t_1] = g'[t_2]\}$ where $g' = g \cup t_1 \cup t_2$. The signature of *merge* is $G \times V_c \times V_c \to G$ and it is defined as follows:

Definition 3.5 (merge).

$$\frac{}{g \cup \{c = c\} = g} \qquad \frac{\begin{array}{c} c_1 \neq c_2 \in V_c \qquad V_c = \{c_1, \ldots, c_k\} \qquad \sigma = [c_1/c_2] \\ N = \{n \in V_n : c_2 \in n\} \\ \mathcal{E} = \{(E(n_1), E(n_2)) : \forall n_1, n_2 \in V_n.\ n_1\sigma = n_2\sigma\}^* \\ E' = E \downarrow_{V_n \setminus N} \sigma \cup [n\sigma \mapsto [E(n)]_\mathcal{E}\sigma : n \in N] \\ g_0 = (\text{codom } E', \text{dom } E', E') \qquad g_i = g_{i-1} \cup \{c_i\sigma = [c_i]_\mathcal{E}\sigma\} \end{array}}{(V_c, V_n, E) \cup \{c_1 = c_2\} = g_k}$$

The *merge* function is defined in two rules. The first states that equating identical classes does not change the e-graph. The second merges the e-class c_2 into the e-class c_1 as follows. (1) On the first line, merge fixes an ordering on the e-classes of the e-graph and creates the substitution σ that replaces c_2 with c_1. (2) On the second line, merge collects all the e-nodes in the e-graph that refer to c_2 into N because c_2 will be removed at the end of merging. (3) On the third line, merge constructs the equivalence relation $\mathcal{E}$ of e-classes which informally either share e-nodes directly or indirectly via other e-classes. This is needed because after fixing an e-node, it may be discovered to belong in two different e-classes. To handle these cases, merge must recursively union such e-classes. (4) On the fourth line, we define the edges of the new e-graph E' in two parts. First, we move all the e-nodes of c_2 into c_1. Second, we fix all the e-nodes collected in N and map them to the canonical e-class per $\mathcal{E}$. However, as the canoncical may be c_2, we must apply it to the substitution σ. (5) On the fifth line, we fold the equalities of all the e-classes with their canonical per $\mathcal{E}$. The fold builds on the initial e-graph whose e-nodes are the domain of E', e-classes are the codomain of E', and edges are E'. We use this initial graph, as opposed to the expected $(V_c, V_n \setminus N \cup N\sigma, E')$ definition, to remove all empty e-classes. (6) Finally, we define the merge to be the result of the folding described earlier.

Finally, we state *merge*'s correctness lemma:

Lemma 3.6. *Let g be an e-graph and $g' = g \cup \{t_1 = t_2\}$, then $g'[t_1] = g'[t_2]$.*

3.1.2 E-Graph Correctness. With these functions defined on the e-graph, we can now postulate its correctness, i.e., that it encodes an equivalence relation with congruence.

First, we define the equivalence relation induced by an e-graph as follows:

Definition 3.7 (E-graph equivalence relation). Let g be an e-graph, then $\sim_g$ is the following relation:

$$\frac{t \notin g}{t \sim_g t} \qquad \frac{t_1 \in g \qquad t_2 \in g \qquad g[t_1] = g[t_2]}{t_1 \sim_g t_2}$$

As expected, the following lemma holds:

Lemma 3.8. *$\sim_g$ is an equivalence relation.*

Finally, we state in Theorem 3.10 that $\sim_g$ is also Σ-congruent, which we define as follows:

Definition 3.9. An equivalence relation $\sim$ is Σ-congruent when, for every $t_1 \sim t_1', \cdots, t_n \sim t_n'$ holds $f(t_1, \ldots, t_n) \sim f(t_1', \ldots, t_n')$ for $f \in \Sigma$ and $arity(f) = n$.

Theorem 3.10. *Given a set of equalities of terms $\mathcal{E} = \{t_1 = t_1', \ldots, t_n = t_n'\}$ and the empty e-graph $g_\emptyset = (\emptyset, \emptyset, \cdot)$, then $\sim_{g_\emptyset \cup \mathcal{E}}$ is Σ-congruent.*

In the following, we use our reasoning framework to establish that the equivalence relation induced by an empty e-graph seeded with a set of equalities $\mathcal{E}$ is the expected one: It is neither too big, e.g., equating terms that should not necessarily be equal, nor too small, e.g., not deriving the equivalence of two terms in the reflexive, symmetric, transitive, and congruent closure of $\mathcal{E}$.

We accomplish this result by showing that the expected relation implies the e-graph's relation and vice-versa. To that end, we define the usual partial-order over equivalence relations as follows:

Definition 3.11. Let $\sim_1$ and $\sim_2$ be two equivalence relations over the same set T, then $\sim_1 \sqsubseteq \sim_2$ if for every $t \in T$ holds $[t]_{\sim_1} \subseteq [t]_{\sim_2}$, i.e., when the equivalence class of every term t w.r.t. $\sim_1$ is a subset of the equivalence class of t w.r.t. $\sim_2$.

To prove that one equivalence relation is less than the other, we can use the following lemma:

Lemma 3.12. *$\sim_1 \sqsubseteq \sim_2$ if and only if for every s, t, $s \sim_1 t$ implies $s \sim_2 t$.*

Lemma 3.13 states that the minimal element of equivalence relations is syntactic equality: the equivalence relation with as many equivalence classes as terms, each class containing a single term. To avoid notational confusion, instead of = we denote syntactic equality with $\emptyset^*$, the reflexive, symmetric, and transitive closure of the empty set.

Lemma 3.13. *For every equivalence relation $\sim$ holds $\emptyset^* \sqsubseteq \sim$.*

With these lemmas, we can state the completeness theorem in Theorem 3.14:

Theorem 3.14. *Let $\mathcal{E}$ be a set of equalities and $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph. Let $\mathcal{E}^\star$ be the reflexive, symmetric, transitive, and Σ-congruent closure of $\mathcal{E}$. Then, $\mathcal{E}^\star \sqsubseteq \sim_{g_\emptyset \cup \mathcal{E}}$.*

Theorem 3.15 is the soundness theorem that states that the equivalence relation induced from an e-graph is the smallest congruent equivalence relation:

Theorem 3.15. *Let $\mathcal{E}, g_\emptyset$, and $\mathcal{E}^\star$ be as in Theorem 3.14. Let $\sim$ be a Σ-congruent equivalence relation such that $\mathcal{E}^\star \sqsubseteq \sim$, then $\sim_{g_\emptyset \cup \mathcal{E}} \sqsubseteq \sim$.*

The following corollary that combines both theorems with the antisymmetry of $\sqsubseteq$ easily follows.

Corollary 3.16. *Let $\mathcal{E}, g_\emptyset$, and $\mathcal{E}^\star$ be as in Theorem 3.14. Then, $\mathcal{E}^\star = \sim_{g_\emptyset \cup \mathcal{E}}$.*

3.2 E-Graphs with Embeddings

With disequalities, we assume that the e-graph will be seeded with an equality set $\mathcal{E} = \{t_1 = t'_1, \cdots, t_n = t'_n\}$ and a disequality set $\mathcal{D} = \{t_1 \neq t'_1, \ldots, t_m \neq t'_m\}$.

3.2.1 Embedding Framework. Embeddings assumes that the following special symbols exist in the alphabet Σ: eq with $arity(eq) = 2$, $\top$ with $arity(\top) = 0$ and $\bot$ with $arity(\bot) = 0$.

Adding support for disequalities amounts to treating $\mathcal{D} = \{t_1 \neq t'_1, \ldots, t_m \neq t'_m\}$ as an additional equality set $\mathcal{E}^- = \{eq(t_1, t'_1) = \bot, \ldots, eq(t_m, t'_m) = \bot\}$ that the e-graph is seeded with.

The theorems and lemmas from the previous part must also be instantiated with $\mathcal{E}^-$ for the results to hold true.

We do not formalize a full saturation engine, but describe a specialized approach for saturating embeddings. First, the engine must guarantee that the semantics of the embedded equality match the e-graph's treatment of equality, i.e., if the embedded equality of two e-classes is true in e-graph, then they must be merged. This is done by saturating the e-graph with $\forall t_1, t_2, eq(t_1, t_2) = \top \rightarrow t_1 = t_2$. Second, it must guarantee that eq is symmetric. Third, it must guarantee that eq is reflexive. Finally, it must perform double negation elimination.

We define the core of the embedding saturation in the following definition:

Definition 3.17. The core of the embedding saturation is defined by the function $sem : G \rightarrow G$ as stated below with the assumption that $\top$ and $\bot$ are in the e-graph.

$$sem(g) = \begin{cases} sem(g \cup \{c_1 = c_2\}) & \text{if } c_1 \neq c_2 \wedge eq(c_1, c_2) \sim_g \top \\ sem(g \cup \{eq(c_2, c_1) = \bot\}) & \text{if } eq(c_1, c_2) \sim_g \bot \wedge eq(c_2, c_1) \notin g \\ sem(g \cup \{eq(c, c) = \top\}) & \text{if } eq(c, c) \notin g \\ sem(g \cup \{c_1 = c_2\}) & \text{if } eq(c, \bot) \sim_g \bot \wedge g[eq(c_1, c_2)] = c \\ g & \text{otherwise} \end{cases}$$

We can now state the correctness theorem of the embedding saturation:

Theorem 3.18. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ be a set of equalities, and $\mathcal{D}$ be a set of disequalities. Let $\mathcal{E}^-$ be the translation of $\mathcal{D}$ into equality embeddings. Let $g' = sem(g_\emptyset \cup \{\top, \bot\} \cup$*

$\mathcal{E} \cup \mathcal{E}^-$). Then, for all e-classes c_1, c_2 of g', holds $eq(c_1, c_2) \sim_{g'} \top$ if and only if $c_1 = c_2$. And, for all e-classes c_1, c_2 of g', if $eq(c_1, c_2) \sim_{g'} \bot$ then $eq(c_2, c_1) \sim_{g'} \bot$.

Moreover we can state the following cost claim of equality embedding:

Theorem 3.19. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ and $\mathcal{D}$ be a set of equalities and disequalities respectively. Let $\mathcal{E}^-$ be the embedding of $\mathcal{D}$. Let $(V_c, V_n, E) = sem(g_\emptyset \cup \{\top, \bot\} \cup \mathcal{E})$ and $(V_c', V_n', E') = sem((V_c, V_n, E) \cup \mathcal{E}^-)$. Then, $|V_c'| \leq |V_c|$ and $|V_n| + |V_c| \leq |V_n'| \leq |V_n| + |V_c| + |\mathcal{D}|$.*

The theorem states that the number of e-classes never increases as any new term added to the e-graph will be merged with an existing class. The number, however, may decrease, but only by using the double negation elimination saturation rule. The other rule that may decrease the number of e-classes is equality lifting in the first case of *sem* which, at first glance, may not apply as all equalities are equated to $\bot$ in $\mathcal{E}^-$. However, such an equality may be embedded within a double negation elimination that will trigger that first rule. For example the following disequality leads the saturation engine to use all four saturation rules: $eq(eq(c_1, c_2), \bot) \neq \bot$.

The theorem also describes the new nodes introduced by *sem* to belong to one of two sets: those embedding the reflexivity of *eq* for each e-class of which there is $|V_c|$, and those embedding the commutativity of *eq* of which there may be up to $|\mathcal{D}|$. Note that V_n already contains a copy of $\mathcal{D}$ and thus the upper bound must add only a second commutated copy. The upper bound on V_n is tight in the case when $\mathcal{D}$ does not include the symmetric and reflexive version of any of its disequalities, while the lower bound is tight when $\mathcal{D}$ is empty.

These bounds also hint at the runtime cost of saturation since each call performs a single merge. Thus, it follows that embedding saturation performs up to $|V_c| + |\mathcal{D}|$ merges.

3.2.2 Equality Embedding (EE). The use of equality embedding is to find contradictions, i.e., when $\top \sim_g \bot$. This is the case when an equality of the shape $eq(c, c) = \bot$ is discovered.

This hints at a potential improvement, particularly to the third case in the definition of *sem*, where the reflexive property of *eq* can be embedded for c when c occurs in a disequality embedding. We define the optimization:

Definition 3.20. The optimized core of the embedding saturation is defined by the function $sem_{EE} : G \longrightarrow G$ as stated below with the assumption that $\top$ and $\bot$ are in the graph.

$$sem_{EE}(g) = \begin{cases} sem_{EE}(g \cup \{c_1 = c_2\}) & \text{if } c_1 \neq c_2 \wedge eq(c_1, c_2) \sim_g \top \\ sem_{EE}(g \cup \{eq(c_2, c_1) = \bot\}) & \text{if } eq(c_1, c_2) \sim_g \bot \wedge eq(c_2, c_1) \notin g \\ sem_{EE}(g \cup \{eq(c, c) = \top\}) & \text{if } eq(c, c) \notin g \wedge \exists c', eq(c, c') \sim_g \bot \vee eq(c', c) \sim_g \bot \\ sem_{EE}(g \cup \{c_1 = c_2\}) & \text{if } eq(c, \bot) \sim_g \bot \wedge g[eq(c_1, c_2)] = c \\ g & \text{otherwise} \end{cases}$$

The cost improvement of this optimization is reflected in the following adaptation of the cost claim:

Theorem 3.21. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ and $\mathcal{D}$ be a set of equalities and disequalities respectively. Let $\mathcal{E}^-$ be the embedding of $\mathcal{D}$. Let $(V_c, V_n, E) = sem_{EE}(g_\emptyset \cup \{\top, \bot\} \cup \mathcal{E})$ and $(V_c', V_n', E') = sem_{EE}((V_c, V_n, E) \cup \mathcal{E}^-)$. Then, $|V_c'| \leq |V_c|$ and $|V_n| + \min(|V_c|, |\mathcal{D}|) \leq |V_n'| \leq |V_n| + \min(|V_c|, |\mathcal{D}|) + |\mathcal{D}|$.*

It follows as a corollary that, when $\mathcal{D} = \emptyset$, equality embedding has no cost on the e-graph.

Finally, we state the correctness theorem of equality embedding:

Theorem 3.22. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ and $\mathcal{D}$ be a set of equalities and disequalities respectively. Let $\mathcal{E}^-$ be the embedding of $\mathcal{D}$. Let $\mathcal{D}^s$ be the symmetric closure of $\mathcal{D}$. Let $g = sem_{EE}(g_\emptyset \cup \{\top, \bot\} \cup \mathcal{E} \cup \mathcal{E}^-)$. Then, $\mathcal{E}^\star \cap \mathcal{D}^s \neq \emptyset$ if and only if $\top \sim_g \bot$.*

3.2.3 Optimized Embedding Framework (OEE). The main observation for the optimization offered by OEE is that a contradiction can be found while performing equality saturation: Instead of guaranteeing that $\top \sim_g \bot$, saturation can stop when it first encounters $eq(c, c) \sim_g \bot$.

Definition 3.23. The equality embedding saturation for OEE is defined as follows:

$$sem_{OEE}(g) = \begin{cases} sem_{OEE}(g \cup \{c_1 = c_2\}) & \text{if } c_1 \neq c_2 \wedge eq(c_1, c_2) \sim_g \top \wedge \nexists c, eq(c, c) \sim_g \bot \\ sem_{OEE}(g \cup \{c_1 = c_2\}) & \text{if } eq(c, \bot) \sim_g \bot \wedge g[eq(c_1, c_2)] = c \\ g & \text{otherwise} \end{cases}$$

There are two modifications compared to Definition 3.20: The saturation rules that add reflexive and symmetric equalities are removed and an additional guard is added that guarantees that, as soon as a contradiction – in its new form – is found, the saturation is done.

As we modified the check for contradiction, the correctness theorem must be restated as follows:

Theorem 3.24. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ and $\mathcal{D}$ be a set of equalities and disequalities respectively. Let $\mathcal{E}^-$ be the embedding of $\mathcal{D}$. Let $\mathcal{D}^s$ be the symmetric closure of $\mathcal{D}$. Let $g = sem_{OEE}(g_\emptyset \cup \{\top, \bot\} \cup \mathcal{E} \cup \mathcal{E}^-)$. Then, $\mathcal{E}^\star \cap \mathcal{D}^s \neq \emptyset$ if and only if there exists an e-class c in the e-graph such that $eq(c, c) \sim_g \bot$.*

The bounds on the cost theorem are improved by the fact that reflexive and symmetric terms are not added anymore. In fact, the additional guard does not contribute an improvement as in the worse-case scenario – when no contradiction exists – the guard is always satisfied.

Theorem 3.25. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ and $\mathcal{D}$ be a set of equalities and disequalities respectively. Let $\mathcal{E}^-$ be the embedding of $\mathcal{D}$. Let $(V_c, V_n, E) = sem_{OEE}(g_\emptyset \cup \{\top, \bot\} \cup \mathcal{E})$ and $(V_c', V_n', E') = sem_{OEE}((V_c, V_n, E) \cup \mathcal{E}^-)$. Then, $|V_c'| \leq |V_c|$ and $|V_n| \leq |V_n'| \leq |V_n| + |\mathcal{D}|$.*

3.2.4 Negated Equality Embedding (NEE). The main observation of NEE is that in the absence of user-provided equality embeddings, all embeddings in the e-graph will always have the form $eq(c_1, c_2) = \bot$. Thus by embedding disequality instead, the existence of a disequality e-node of the form $ne(c, c)$ is enough to signal a contradiction.

The result is drastic as no saturation is needed anymore. This greatly simplifies the statement of the correctness theorem as follows.

Theorem 3.26. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ be a set of equalities, and $\mathcal{D}$ be a set of disequalities. Let $\mathcal{N}^\neq = \{ne(t_1, t_2) : t_1 \neq t_2 \in \mathcal{D}\}$ be the disequality embedding terms. Let $\mathcal{D}^s$ be the symmetric closure of $\mathcal{D}$. Let $g = g_\emptyset \cup \mathcal{N}^\neq \cup \mathcal{E}$. Then, $\mathcal{E}^\star \cap \mathcal{D}^s \neq \emptyset$ if and only if $ne(c, c) \in g$ for some e-class c in g.*

The cost theorem is equally simpler to state.

Theorem 3.27. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $\mathcal{E}$ be a set of equalities, and $\mathcal{D}$ be a set of disequalities. Let $\mathcal{N}^\neq = \{ne(t_1, t_2) : t_1 \neq t_2 \in \mathcal{D}\}$ be the disequality embedding e-nodes. Let $(V_c, V_n, E) = g_\emptyset \cup \mathcal{E}$, and $(V_c', V_n', E') = (V_c, V_n, E) \cup \mathcal{N}^\neq$. Then, $|V_c'| = |V_c| + |\mathcal{N}^\neq|$ and $|V_n| \leq |V_n'| \leq |V_n| + |\mathcal{N}^\neq|$.*

The additional cost in the number of e-classes results from the fact that the disequality e-nodes are never stated to be equal to any other expression, i.e., each exists in its own e-class and this e-class only has a single e-node: the disequality one.

At first glance, the cost in the number of e-nodes may appear identical to the one in Theorem 3.25. However, observe that the previous cost theorem does not provide a bound with respect to the first saturation loop, but rather only with respect to the second. The additional e-classes introduced in the first round is thus not accounted for. As no saturation is done in NEE, these additional unaccounted ones no longer exist.

3.3 Die-Graphs: E-Graphs with Disequality Edges

A die-graph is an e-graph with an additional set storing the disequality edges. We do not represent disequality edges using a function, but rather through a general relation since an e-class may be unequal to multiple other e-classes. Moreover, since disequality enjoys symmetry and congruence, i.e., if $t_1 = t_2$ and $t_3 \neq t_1$, then $t_3 \neq t_2$, the disequality relation is one among e-classes rather than e-nodes.

Therefore, we define die-graphs formally as follows:

Definition 3.28. A die-graph is a quadruple (V_c, V_n, E, D) such that (V_c, V_n, E) is an e-graph and $D \subseteq V_c \times V_c$.

All operations defined earlier on e-graphs can be applied as-is to die-graphs by leaving D unchanged, with the exception of merging.

To state that two e-classes are unequal, the die-graph adds an undirected edge between both e-classes. The operation to do so is $\mathit{disequate} : G \times V_c \times V_c \to G$ and, per the notational convention we built thus far, we use $g \cup \{c_1 \neq c_2\}$ as an abbreviation for this function. Moreover, $g \cup \{t_1 \neq t_2\}$ is an abbreviation for $g' \cup \{g'[t_1] \neq g'[t_2]\}$ where $g' = g \cup \{t_1, t_2\}$.

We formally define *disequate* as follows:

Definition 3.29. Let (V_c, V_n, E, D) be a die-graph, then

$$(V_c, V_n, E, D) \cup \{c_1 \neq c_2\} = (V_c, V_n, E, D \cup \{(c_1, c_2), (c_2, c_1)\})$$

When the *merge* operation is given two different e-classes, it is guaranteed that one of them will be removed from the e-graph. As the set of disequality edges must be a subset of the cartesian product of e-classes, *merge* cannot leave D as-is and it must be redefined. In Definition 3.30, we recreate the *merge* definition from Definition 3.5 with the necessary changes highlighted in gray:

Definition 3.30 (merge).

$$\frac{}{g \cup \{c = c\} = g} \qquad \frac{\begin{array}{c} c_1 \neq c_2 \in V_c \qquad V_c = \{c_1, \ldots, c_k\} \qquad \sigma = [c_1/c_2] \\ N = \{n \in V_n : c_2 \in n\} \\ \mathcal{E} = \{(E(n_1), E(n_2)) : \forall n_1, n_2 \in V_n.\ n_1\sigma = n_2\sigma\}^* \\ E' = E \downarrow_{V_n \setminus N} \sigma \cup [n\sigma \mapsto [E(n)]_{\mathcal{E}}\sigma : n \in N] \\ g_0 = (\operatorname{codom} E', \operatorname{dom} E', E', \boxed{D\sigma}) \qquad g_i = g_{i-1} \cup \{c_i\sigma = [c_i]_{\mathcal{E}}\sigma\} \end{array}}{(V_c, V_n, E) \cup \{c_1 = c_2\} = g_k}$$

Cost. The cost of the disequality edges follows from Definitions 3.29 and 3.30 and can be formulated as follows:

Theorem 3.31. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot)$ be the empty e-graph, $g'_\emptyset = (\emptyset, \emptyset, \cdot, \emptyset)$ be the empty die-graph, $\mathcal{E}$ be a set of equalities, and $\mathcal{D}$ be a set of disequalities. Let $(V_c, V_n, E) = g_\emptyset \cup \mathcal{E} = (V_c, V_n, E)$ and let $(V'_c, V'_n, E', D) = g'_\emptyset \cup \mathcal{E} \cup \mathcal{D}$. Then, $V'_c = V_c$, $V'_n = V_n$, $E' = E$, and $|\mathcal{D}| \leq |D| \leq 2|\mathcal{D}|$.*

The bound on the size of disequality edges is tight: $|D| = |\mathcal{D}|$ when $\mathcal{D}$ is equal to its symmetric closure and $|D| = 2 \cdot |\mathcal{D}|$ when no disequality is present in its symmetric form.

Finally we can prove the correctness theorem of the die-graph:

Theorem 3.32. *Let $g_\emptyset = (\emptyset, \emptyset, \cdot, \emptyset)$ be the empty die-graph, $\mathcal{E}$ be a set of equalities, and $\mathcal{D}$ be a set of disequalities. Let $(V_c, V_n, E, D) = g_\emptyset \cup \mathcal{E} \cup \mathcal{D}$. Let $\mathcal{E}^\star$ be the reflexive, symmetric, transitive, and Σ-congruent closure of $\mathcal{E}$ and let $\mathcal{D}^s$ be the symmetric closure of $\mathcal{D}$. Then, $\mathcal{E}^\star \cap \mathcal{D}^s \neq \emptyset$ if and only if there exists $c \in V_c$ such that $(c, c) \in \mathcal{D}$.*

3.3.1 Comparison of Embeddings and Disequality Edges. Theorems 3.21, 3.25, 3.27 and 3.31 give grounds on which embeddings and disequality edges can be compared w.r.t. memory usage.

For all embedding techniques, we must choose between either increasing the number of e-nodes (OEE) linearly in the size of disequalities or increasing the number of e-classes (NEE) linearly in the size of disequalities. Instead, with disequality edges, the number of e-nodes and e-classes remain constant at the cost of storing separately a pair of e-classes for each disequality. Therefore, disequality edges perform better memory-wise.

A similar argument can be made for an improved run time with disequality edges. For EE and OEE, saturation must be done and takes an order of iterations linear to the number of disequalities. While NEE does not perform saturation, and thus incurs no fundamental cost on the basic operations of the e-graph, the cost is transferred to e-matching and invariant maintenance where the set of e-classes and e-nodes are searched.

4 Implementation

We implement DieGraph as an extension of the egg Rust library [Willsey et al. 2021], which implements e-graphs as follows: E-graphs are defined in the `EGraph` struct, which includes a vector of e-classes `classes`. E-classes are defined in the `EClass` struct, which includes a unique identifier `id` of type `Id`, a vector of e-class identifiers `parents`, and a vector of its e-node members `nodes`. E-nodes are instances of the `Language` trait, which offers a symbol and an array of e-class arguments.

Notably, the `EGraph` contains a `unionfind` field that efficiently records the history of e-class merges using a union-find data structure [Galler and Fisher 1964; Hopcroft and Ullman 1973]. With this approach, egg avoids updating e-class identifiers in the e-graph operations that require to do so. Instead, the update can be delayed to run on-demand using the `find(id)` method, which is known to be constant amortized time [Tarjan 1975].

Adding the `forbid` List. Our implementation encodes disequality edges as "forbid lists" in e-classes as proposed by Nelson [1980]. Thus, at the core of the implementation is a change to the `EClass` struct[1] that adds a `forbid` field which is a list of e-class identifiers `Id`, as shown in Listing 1.

Listing 1. Adding the `forbid` field to `EClass`.

```
pub struct EClass<L, D> {
  // ...
  /// The eclasses known to be unequal to this one.
  pub forbid: Vec<Id>,
  // ...
}
```

Moving `forbid` Lists After Unioning. When equating two terms, the e-class of one is removed and all its children are moved to the e-class of the other. Analogously, all of the e-classes in the forbid list of the former are moved to forbid list of the latter. In egg, the `perform_union` method of the `EGraph` structure[2] is responsible for performing the union. Line 7 of Listing 2 presents the necessary addition to that function.

[1] https://github.com/egraphs-good/egg/blob/3231b86/src/eclass.rs
[2] https://github.com/egraphs-good/egg/blob/3231b86/src/egraph.rs

Generally, all identifiers in every forbid list must be updated since unioning removes e-classes. This update is expensive: At best, it is on the order of the size of the forbid list and, at worst, on the order of the number of e-classes. However, thanks to the union-find in `EGraph`, we can avoid this update.

Adding Disequalities. To add a disequality to the e-graph, DieGraph finds the identifier of the e-classes and pushes each in the `forbid` list of the other. DieGraph exposes the function `disunion` to the user, which is defined in Listing 3.

Checking Disequalities. Listing 4 defines the function `are_unequal` for `EGraph` which can be used to check if two terms are known to be unequal in DieGraph. It finds the e-class identifiers of both arguments, then searches the smaller of the two forbid lists for the identifier of the e-class with the larger forbid list.

Checking Consistency. To check for consistency, i.e., that no contradiction exists, DieGraph implements the function `is_consistent` as shown in Listing 5. For each e-class, the function checks that its identifier does not appear in its forbid list after fixing the identifiers using the union-find as mentioned earlier.

Listing 2. Updating the `forbid` list.

```
fn perform_union(
  // ...
) -> bool {
  // ...
  concat_vecs(&mut class1.nodes, class2.nodes);
  concat_vecs(&mut class1.parents, class2.parents);
  concat_vecs(&mut class1.forbid, class2.forbid);
  // ...
}
```

Listing 3. Adding a disequality.

```
pub fn disunion(&mut self,
     id1: Id, id2: Id) {
  let id1 = self.find(id1);
  let id2 = self.find(id2);
  self.classes.get_mut(&id1).unwrap()
    .forbid.push(id2);
  self.classes.get_mut(&id2).unwrap()
    .forbid.push(id1);
}
```

Listing 4. Checking a disequality.

```
pub fn are_unequal(&self, id1: Id, id2: Id) -> bool {
  let mut id1 = self.find(id1);
  let mut id2 = self.find(id2);

  if self.classes[&id1].forbid.len() >
     self.classes[&id2].forbid.len() {
    std::mem::swap(&mut id1, &mut id2);
  }

  self.classes[&id1].forbid.iter().any(|id| {
    self.find(*id) == id2
  })
}
```

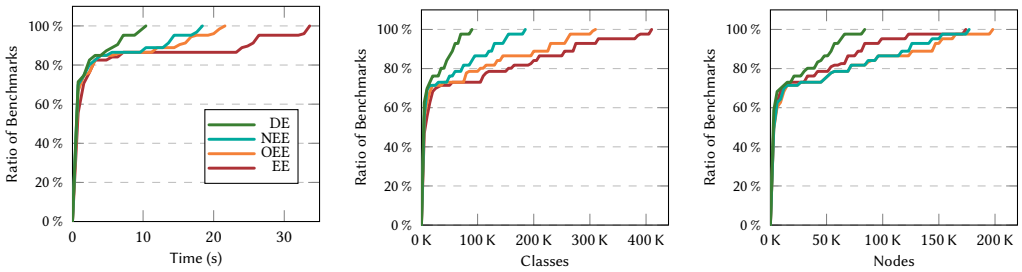
Listing 5. Consistency checking.

```
pub fn is_consistent(&self) -> bool {
  self.classes.values().all(|c| {
    let cid = self.find(c.id);
    c.forbid.iter().all(|id| {
      self.find(*id) != cid;
    })
  })
}
```

5 Evaluation

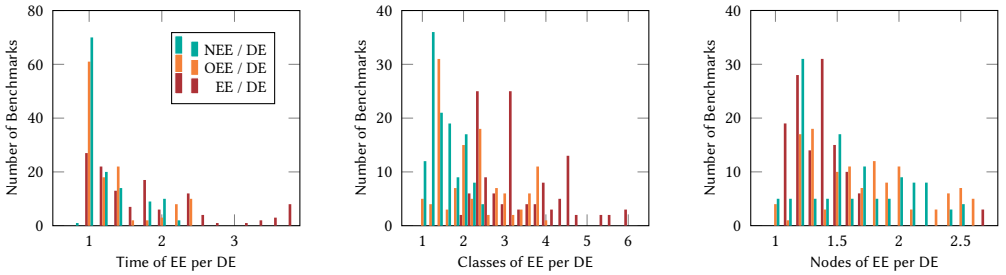
In this section, we compare the three equality embedding approaches and disequality edges (as discussed in Section 2.2).

First, we use automated theorem proving as a case study where disequalities arise during pattern matching (Section 5.1). Second, we use SMT solving as a case study where disequalities come from interpreting the result of the SAT solver (Section 5.2). Third, we perform a parameter analysis on all equality embeddings and on disequality edges, showing the effect of the number of asserted disequalities on the number of e-nodes and e-classes, and on the time to build the e-graph



(a) Time. (b) Classes. (c) Nodes.

Fig. 5. Cumulative Comparison Between DE and EE for Inductive Prover Benchmarks.



(a) Time Ratio. (b) Class Ratio. (c) Node Ratio.

Fig. 6. Histogram for Ratios Between DE and EE for Inductive Prover Benchmarks.

(Section 5.3). Last, we profile the disequality handling in a widely used industry-strength SMT solver (Section 5.4).

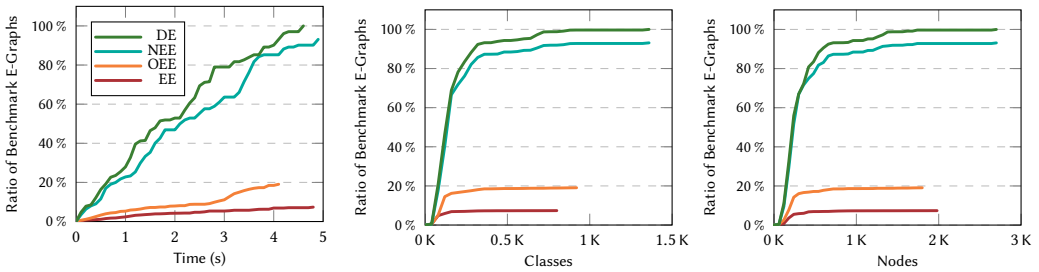
5.1 Inductive Theorem Prover Case Study

This case study is based on the inductive theorem prover Propel [Zakhour et al. 2024a]. Propel focuses on a set of fundamental algebraic laws, such as commutativity, associativity, idempotence, etc. We replaced Propel's implementation of equalities, disequalities, and saturation with a die-graph. In the experiments, we compare the DE (disequality edges) and the EE, OEE, and NEE (embedding) approaches.

We used Propel's 128 benchmark theorems [Zakhour et al. 2023], which contain the subset of the TIP 2015 benchmarks for inductive theorem provers [Claessen et al. 2015] that checks the algebraic properties supported by Propel.

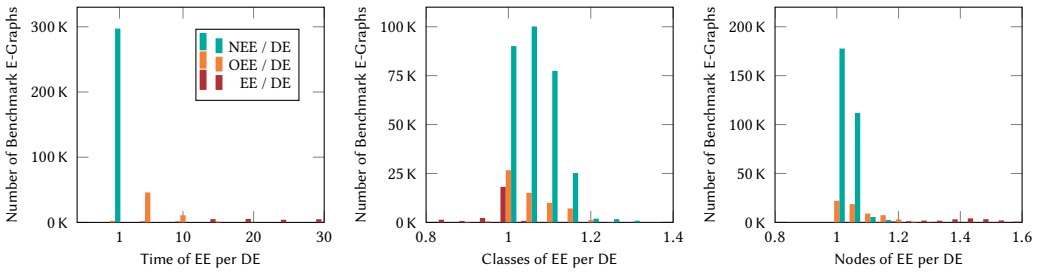
Results. The cumulative charts in Figure 5 show the number of completed benchmarks (y-axis) given a certain number of resources (x-axis). The resources that we consider are: the time (Figure 5a), the number of e-classes (Figure 5b) and the number of e-nodes (Figure 5c). In these charts, the faster the line grows the better the approach is as it completes more benchmarks with fewer resources. The results show that, in all metrics, disequality edges outperform the embeddings.

The histograms in Figure 6 show the improvements and regressions of DE over the other approaches (NEE, OEE and EE) w.r.t. a certain cost. In particular, they represent the number of benchmarks (y-axis) for a certain range of improvement/regression (x-axis). In these charts, one approach is better than another when the bars before one are few and small while those after one are many and high.



(a) Time. (b) Classes. (c) Nodes.

Fig. 7. Cumulative Comparison Between DE and EE for SMT Benchmarks.



(a) Time Ratio. (b) Class Ratio. (c) Node Ratio.

Fig. 8. Histogram for Ratios Between DE and EE for SMT Benchmarks.

The results show that in the vast majority of cases, DE exhibits slight to substantial improvements over all embeddings. The outliers are benchmarks which require a small amount of e-nodes during proof search.

5.2 SMT Case Study

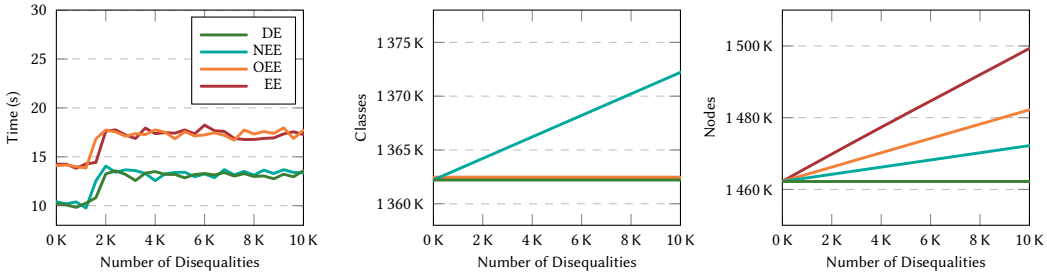
For this case study we implemented an SMT solver for the EUF (Equality and Uninterpreted Functions) theory. Its backend uses MiniSat [Eén and Sörensson 2003] for solving boolean propositions and an e-graph for handling EUF propositions. The e-graph implementation can be swapped to use egg or our implementation as described in Section 4.

Our implementation does not aim to be competitive. In particular, we do not implement fully-fledged equality saturation since it is not strictly necessary to explore disequalities in e-graphs. Instead, it is crucial that the variants of the solver are identical in all regards but their treatment of disequality, so that we can accurately compare the different approaches on the same data.

We run our solver on the well-known SMT-LIB benchmark [Barrett et al. 2015] for EUF, which amounts to a total of 7 590 benchmark files. We use a timeout of five seconds for every benchmark which is enough to report on 6 441 of the benchmarks.

Results. Similarly to the inductive prover case study (Section 5.1), the cumulative charts in Figure 7 show the number of completed benchmarks (y-axis) given a certain number of resources (x-axis). The results are consistent across all resources and show that DE costs less than NEE, which costs less than OEE, which costs less than EE.

The histograms in Figure 8 show the improvements (greater than one on the x-axis) or regressions (less than one on the x-axis) of DE over the other approaches (NEE, OEE and EE) w.r.t. a certain cost.



(a) Time. (b) Classes. (c) Nodes.

Fig. 9. Comparison Between DE and EE for Different Ratios of Disequalities.

The single high bar at one indicates that DE performs similar to an embedding approach for these cases. Most cases, however, show modest to significant improvements. NEE performs better than EE and OEE as it generates fewer e-nodes. However, it is still outperformed by DE. We attribute this result to the higher overhead of maintaining the e-graph invariant that NEE suffers from.

Overall, our empirical results for SMT are in line with the results for the inductive prover and our analytical results, suggesting that disequality edges substantially outperform equality embedding. There are consistent improvements of DE over NEE, OEE, and EE in terms of the running time, the number of e-classes, and the number of e-nodes across both SMT solving and inductive theorem proving.

5.3 Parameter Analysis

We further perform a parameter analysis for all the equality embedding approaches and the disequality edges, varying the number of asserted disequalities from 0 to 10 000 for synthetically generated benchmarks.

For the disequality edge approach, we use our implementation as described in Section 4 and for the embedding approaches we use egg.

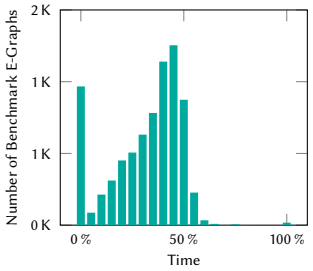
We generate 110 K pairs of s-expressions with a maximum depth of 5. The first 100 K pairs are asserted as equalities. The remaining 10 K pairs are disequality candidates, which are chosen to be subexpressions of the first 100 K equalities. The atoms of the s-expressions range over the numbers 1 to 5 (inclusive) and over three functions, `f`, `g`, and `h` of arity 1, 2, and 3, respectively. An example of such a pair is the following: `(h 4 2 1)` and `(f (h (g 5 1) (f (h 2 4 3)) (g 5 5)))`.

For each variant, we start with an empty e-graph in which we assert the equality of all 100 K pair of expressions. As s-expressions treat the numerals 1, 2, 3, 4, and 5 as symbols and not integers, we then assert the pairwise disequality of all the five numerals.

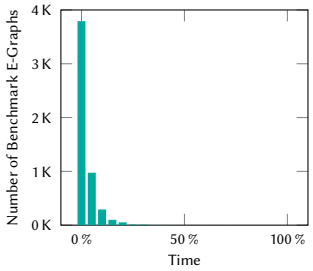
Figure 9a shows the full time required to populate the empty e-graph with the first 100 K equalities and the varying number of disequalities (x-axis), followed by saturation for EE and OEE and the check for consistency. The graph shows clearly the cost of saturation to be around 3.5 seconds. Moreover, it is also clear that EE and OE perform similarly, as do NEE and DE.

NEE, on the other hand, can be clearly distinguished in Figure 9b, which shows that it is the only technique that shows an increase in its number of e-classes linearly with the number of disequalities.

Figure 9c illustrates clearly the progressive improvement sketched in Section 2: OE optimizes the number of nodes required for EE by removing a saturation rule, NEE does the same w.r.t. OE, and DE removes the e-node encoding all together.



(a) Z3 E-Graph Time Ratio.



(b) Z3 Disequality Time Ratio.

Fig. 10. Histogram for Z3 Running EUF Benchmarks.

These plots are consistent with Theorems 3.21, 3.25, 3.27 and 3.31, which state that the number of classes is constant for EE, OEE, and DE but grows linearly for NEE. And the number of nodes grows linearly for EE, OEE, and NEE but remains constant for DE.

5.4 Disequalities in Existing Tools

In this section, we evaluate the run time cost of e-matching in the e-graph, e-graph saturation and – in particular – the treatment of disequality in real-world solvers. We are thus interested in the following questions: (1) How much of the total runtime is spent manipulating the e-graph and (2) how much of the total runtime is spent reasoning about disequalities.

To answer these questions, we compiled Z3 v4.13.0 with debug and instrumentation enabled. We ran Z3 on the 7 590 SMT-LIB EUF benchmark [Barrett et al. 2015] with a five second timeout. The generated profile for each benchmark contains the percentage of time spent in each function. We manually inspected the 1 385 different functions that the solver spent more than 0.01 % of its time in. We identified 205 functions that manipulate e-graphs, among which six handle disequalities.

In Figure 10a, we plot the number of benchmark files that spent a given percentage of its runtime manipulating an e-graph. In Figure 10b, we plot the percentage of running time manipulating disequalities. In the latter plot, we exclude the files that spent less than 5 % of their time on disequalities leaving 1 453 or 19 % of the benchmark files.

The average time spent manipulating an e-graph is 33.8 %, while the average time spent manipulating disequalities is 4 % across all benchmarks and 10 % across those spending more than 5 % of their time on disequalities. This indicates that Z3 might benefit from treating disequalities like OpenSMT does and use DE instead of embedding.

The contributors of the e-graph implementations of Z3 and OpenSMT confirmed that the respective e-graphs were designed with a focus on correctly representing disequalities. Yet, so far there has been no performance comparison between different approaches. We believe further investigation is needed to fill this gap, for example, adapting Z3 to use DE would allow us to reach more decisive results.

6 Discussion: Disequalities in State-of-the-Art SMT Solvers

During our work on DieGraph, we observed that industry-strength SMT solvers use different approaches for representing disequalities: Z3 [de Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022] use an embedding approach close to NEE, while OpenSMT [Bruttomesso et al. 2010] uses DE. In this section we elaborate on the implementation choices of each of these tools.

Z3. E-graphs in Z3 are defined in the `euf::egraph` class and its e-nodes are defined in the `euf::enode` class. E-classes do not have their own representation as a class. Instead, they are represented by their canonical e-node.

E-nodes in Z3 have a `m_is_equality` boolean flag that indicates whether the e-node embeds an equality or not. Similar to EE and OEE, Z3 maintains both `l_true` and `l_false` e-nodes, representing truth or falsehood. When the EUF solver is alerted by the SAT solver that an e-node representing an equality is `false` – by setting the e-node's `m_value` to `l_false` – the `new_diseq` method is called on the e-graph. If the SAT solver alerts the EUF solver that such an e-node is instead `true`, the arguments of the embedded equality are merged in the e-graph. Crucially, these equality embedding e-nodes are never merged with the e-nodes representing truth or falsehood, similar to NEE. Instead, the `m_value` indicates whether the equality is true or not. Thus, Z3 embeds both equality and disequality.

While propagating equalities, when two e-classes become equal, they are unioned in the e-graph as expected, but their equality is also communicated back to the SAT solver. In particular, if both sides of a disequality become equal, the SAT solver discovers the contradiction and halts the search for a model with the `unsat` result.

cvc5. E-graphs in CVC5 are defined in the `EqualityEngine` class and its e-nodes are defined in `EqualityNode` and identified with `EqualityNodeId`. Similar to Z3, e-classes are not represented by their own class. Instead, the identifier of the canonical e-node is used.

A disequality is added by calling `EqualityEngine::assertEquality` with an AST-like node representing an equality with negative polarity. Internally, this method merges the given node with the e-node representing `false`.

While merging and propagating equalities, the method `propagateTriggerTermDisequalities` of `EqualityEngine` is called. It loops over all the disequalities and checks if any of them have equal arguments, i.e., violate equality's reflexivity. If that is the case, propagation stops, `merge` returns `false`, an inconsistency flag is raised, and thereafter the method `consistent()` returns `false`.

OpenSMT. As the previous two solvers, OpenSMT also does not define an e-class and represents each e-class with its canonical e-node.

The `Enode` class uses the same implementation technique proposed by Nelson [1980] to encode disequality edges by keeping a `forbid` list. Asserting a disequality between two e-nodes is done via the `assertNeq` method of the `Egraph`, which adds each e-node to the forbid list of the other's root. The `unmergable` method of `Egraph` takes two e-nodes and checks if they cannot be merged by traversing their `forbid` lists.

Rebuilding the e-graph invariants is done with the `mergeLoop` method of `Egraph`, which is the method that finds contradictions if two unmergeable e-nodes are to be merged.

7 Related Work

E-graphs were introduced as a data structure for reasoning about program equivalence [Nelson 1980]. Later, the possibility of compactly representing program spaces has led to an increasing interest in the use of e-graphs for program optimization [Bytecode Alliance 2016; Joshi et al. 2002].

E-graphs address two fundamental problems: the construction of the program space, and retrieving elements from such space. We discuss the operations related to these problems in Section 3.1.1. The key insight of Tate et al. [2009] is the use of equational, monotone rewrites (equality saturation) for building the program space. Based on this insight, equality saturation has been used successfully for program manipulation. Some noticeable applications in program optimization include tensor graphs [Yang et al. 2021], vectorization for signal processors [VanHattum et al. 2021], linear algebra optimization [Wang et al. 2020], and improved accuracy for floating point expressions [Panchekha

et al. 2015]. In program synthesis, e-graphs have been used to synthesize CAD models [Nandi et al. 2020] and for *library learning*, that is extracting common structure from a corpus of programs into reusable library functions [Cao et al. 2023].

The egg library [Willsey et al. 2021] is a Rust implementation of e-graphs. It introduces e-class analyses, which allow simple semantic analyses over e-graphs, overcoming the issue that sound rewrite rules are difficult to define in a purely syntactic way. Subsequent work [Zhang et al. 2023] further improved expressivity by allowing rules to be defined compositionally.

While the key problem solved by e-graphs is construction and retrieving, several authors studied the related *proof problem* which is about not only showing that two terms are equivalent but also exhibiting a proof that they are [Flatt et al. 2022; Nieuwenhuis and Oliveras 2005].

E-Graph Extensions. While, in this paper, we studied an extended model of e-graphs with direct support for disequalities, other researchers have augmented e-graphs with additional features, which we discuss next. Program Expression Graphs (PEGs) have been applied in the context of compiler optimizations [Tate et al. 2009]. The resulting data structure, an E-PEG includes operator nodes and built-in nodes for representing conditionals and loops as well as *dataflow* edges that specify where operator nodes get their arguments from, and groups together PEG nodes that are equal into equivalence classes.

More recently, relational e-matching [Zhang et al. 2022] reduces pattern matching over e-graphs to relational queries – borrowing techniques from database query execution – to make the matching procedure orders of magnitude faster and guarantee a complexity bound in the matching algorithm. Yet, this method explicitly keeps both the e-graph and its relational representation. Later, Datalog was combined with e-matching in egglog [Zhang et al. 2023], a fixpoint reasoning system that unifies Datalog and equality saturation (EqSat). Egglog views the whole equality saturation algorithm in a relational setting avoiding the synchronization between the two representations of e-graphs. Egglog supports efficient incremental execution (through classical techniques of incremental view maintenance in databases), cooperating analyses, and lattice-based reasoning line Datalog as well as term rewriting, efficient congruence closure, and extraction of optimized terms like EqSat.

Equality-Constrained Tree Automata (ECTA) [Gissurarson et al. 2023; Koppel et al. 2022] are a data structure that compactly represents a program spaces combining Version-Space Algebras [Lau et al. 2000, 2003] and Finite Tree Automata [Adams and Might 2017] with "entanglement", which enables the choices of terms from e-classes to depend on choices done in other e-classes.

Colored e-graphs [Singher and Shachar 2024] are an extension to e-graphs, motivated by the limits of e-graphs in handling of case splitting, e.g., when equality reasoning requires multiple inconsistent assumptions to reach a single conclusion. Colored e-graphs represent all of the coarsened congruence relations in a single structure, resulting in a memory-efficient equivalent of multiple copies of an e-graph. Colored e-graphs maximize sharing between different cases and track which conclusion is true under which assumption, resulting into multiple "color-coded" layers on top of the original e-graph.

E-Graphs in Theorem Provers and SMT Solvers. Disequalities in SMTs and theorem provers naturally arise from the treatment of negation in equality predicates in equality logic and by pattern matching in term algebras [Echenim and Peltier 2020]. E-graphs have been successfully used for equality reasoning in SMT solvers and theorem provers [Barbosa et al. 2022; de Moura and Bjørner 2008; Detlefs et al. 2005] for theories such as QF UF, linear algebra, and bit-vectors. E-matching has been also adopted for quantifier instantiation [Niemetz et al. 2021], which is exploratory in nature and requires efficient methods [de Moura and Bjørner 2007]. In SMT solvers, the SAT core handles the boolean structure with the standard in CDCL(T) algorithms and the theory solver handles the literals that appear in the formula.

E-Graph Formalisms. De Moura [2008] applies formal treatment to e-graphs. Yet, their goal was not to develop a formal reasoning framework for e-graphs. Hence, central aspects like the congruence-closure algorithm are only described using natural language. Further, many of the definitions are specific to SMT solving, while e-graphs have more general applications. Additionally, the formalism leaves open how disequalities are represented, e.g., as an additional set on the side, as the formalism suggests but which would be inefficient, or embedded like Z3 does, etc. Overall, the formalism cannot be used out-of-the-box to prove theorems about the performance of disequality edges vs. embedding approaches. Instead, the key of our formalism is to define the essential concepts of e-graphs, such as e-nodes and e-classes.

The formalism by Nelson [1980] is very specific, essentially defining a concrete imperative implementation. Instead, our formalism is a specification of the properties that an implementation has to satisfy, yielding a more elegant formal model with simpler proofs. For example, Nelson's correctness argument is a two-pages long proof [Nelson 1980, p. 361–362] whereas our formalism achieves the same result using a sequence of smaller lemmas, each with an elementary proof. Further, their analysis covers correctness and time complexity but not space complexity, as our formalism does.

8 Conclusion

In this paper, we have introduced a formal reasoning framework for e-graphs and the first formalization of an extension to e-graphs that directly supports disequalities. Our formal treatment allows us to prove the correctness of e-graphs and their disequality extension as well as an analytical model of their performance. To demonstrate the practical benefits of this approach, we implemented these ideas in DieGraph, based on the state-of-the-art egg Rust library. The evaluation in both an SMT solver and an automated inductive theorem prover using standard benchmarks shows that direct disequality support outperforms traditional widely-used disequality encoding methods. Our study of profiling the Z3 SMT solver on standard benchmarks reveals that the solver spends a non-negligible amount of time handling disequalities, suggesting it might benefit from disequality edges.

Acknowledgments

The authors wish to thank the anonymous reviewers of POPL 2025 for their valuable comments. In particular we wish to thank Reviewer D for suggesting the Negated Equality Embedding (NEE) approach. This work is supported by the Swiss National Science Foundation (SNSF), grant 200429.

Artifact Availability

The artifact is available on Zenodo [Zakhour et al. 2024b]. It includes the implementation of DieGraph on top of the egg e-graphs library [Willsey et al. 2021], the modification of Propel [Zakhour et al. 2023, 2024a] to use e-graphs with and without disequality edges, and our SMT solver for the EUF theory. Moreover, all the benchmarks provided in Section 5 have dedicated scripts which can be executed to verify our reported results. The included README file provides a guide on how to reproduce and interpret the output of the benchmarks.

References

Michael D. Adams and Matthew Might. 2017. Restricting Grammars with Tree Automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 82 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133906

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on*

Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I (Munich, Germany). Springer-Verlag, Berlin/Heidelberg, Germany, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. The Satisfiability Modulo Theories Library (SMT-LIB). Retrieved November 20, 2024 from https://smt-lib.org/.

Ian Briggs and Pavel Panchekha. 2022. Synthesizing Mathematical Identities with E-Graphs. In *Proceedings of the 1st ACM SIGPLAN International Symposium on E-Graph Research, Applications, Practices, and Human-Factors* (San Diego, CA, USA) *(EGRAPHS '22)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/3520308.3534506

Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT Solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Paphos, Cyprus) *(TACAS '10)*, Javier Esparza and Rupak Majumdar (Eds.). Springer-Verlag, Berlin, Heidelberg, 150–153. https://doi.org/10.1007/978-3-642-12002-2_12

Bytecode Alliance. 2016. Cranelift. Retrieved November 20, 2024 from https://cranelift.dev/. Optimizing compiler backend.

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 14 (Jan. 2023), 29 pages. https://doi.org/10.1145/3571207

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2015. TIP: Tons of Inductive Problems. In *Intelligent Computer Mathematics*, Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (Eds.). Springer International Publishing, Cham, Switzerland, 333–337. https://doi.org/10.1007/978-3-319-20615-8_23

Leonardo de Moura. 2008. SMT Solvers: Theory and Implementation. Retrieved November 20, 2024 from https://leodemoura.github.io/files/oregon08.pdf. Presentation at the Summer School on Logic and Theorem Proving in Programming Languages, Oregon.

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *s* (Bremen, Germany) *(CADE '21)*, Frank Pfenning (Ed.). Springer-Verlag, Berlin, Heidelberg, 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS '08)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473. https://doi.org/10.1145/1066100.1066102

M. Echenim and N. Peltier. 2020. Combining Induction and Saturation-Based Theorem Proving. *Journal of Automated Reasoning* 64, 2 (Feb. 2020), 253–294. https://doi.org/10.1007/s10817-019-09519-x

Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing* (Santa Margherita Ligure, Italy) *(SAT '03, Vol. 2919)*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small Proofs from Congruence Closure. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design* (Trento, Italy) *(FMCAD '24)*, Alberto Griggio and Neha Rungta (Eds.). TU Wien Academic Press, 75–83. https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13

Bernard A. Galler and Michael J. Fisher. 1964. An Improved Equivalence Algorithm. *Commun. ACM* 7, 5 (May 1964), 301–303. https://doi.org/10.1145/364099.364331

Matthías Páll Gissurarson, Diego Roque, and James Koppel. 2023. Spectacular: Finding Laws from 25 Trillion Terms. In *Proceedings of the 16th IEEE Conference on Software Testing, Verification and Validation* (Dublin, Ireland) *(ICST '23)*. IEEE, Piscataway, NJ, USA, 293–304. https://doi.org/10.1109/ICST57152.2023.00035

J. E. Hopcroft and J. D. Ullman. 1973. Set Merging Algorithms. *SIAM J. Comput.* 2, 4 (Dec. 1973), 294–303. https://doi.org/10.1137/0202024

Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proceedings of the ACM on Programming Languages* 6, ICFP, Article 91 (Aug. 2022), 29 pages. https://doi.org/10.1145/3547622

Tessa Lau, Pedro Domingos, and Daniel S. Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration. In *Proceedings of the 17th International Conference on Machine Learning (ICML '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 527–534.

Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53 (Oct. 2003), 111–156. https://doi.org/10.1023/A:1025671410623

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford, CA, USA. AAI8011683.

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. 2021. Syntax-Guided Quantifier Instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems* (Luxembourg, Luxembourg), Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer-Verlag, Berlin, Heidelberg, 145–163. https://doi.org/10.1007/978-3-030-72013-1_8

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications* (Nara, Japan) *(RTA'05)*. Springer-Verlag, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 258 (Oct. 2023), 29 pages. https://doi.org/10.1145/3622834

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

Gordon D. Plotkin. 2004. A Structural Approach to Operational Semantics. *The Journal of Logic and Algebraic Programming* 60–61 (2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001

Eytan Singher and Itzhaky Shachar. 2024. Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions. In *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design* (Prague, Czech Republic) *(FMCAD '24)*, Nina Narodytska and Philipp Rümmer (Eds.). TU Wien Academic Press, Wien, Austria, 70–83. https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_13

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 1919–1932. https://doi.org/10.14778/3407790.3407799

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of the 4th Conference on Machine Learning and Systems (MLSys '21)*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). MLSys.

George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2024b. *Dis/Equality Graphs.* https://doi.org/10.5281/zenodo.13938878

George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 162 (June 2023), 24 pages. https://doi.org/10.1145/3591276

George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024a. Automated Verification of Fundamental Algebraic Laws. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 178 (June 2024), 24 pages. https://doi.org/10.1145/3656408

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI, Article 125 (June 2023), 25 pages. https://doi.org/10.1145/3591239

Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational E-Matching. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 35 (Jan. 2022), 22 pages. https://doi.org/10.1145/3498696

Received 2024-07-11; accepted 2024-11-07