Versioned E-Graphs

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

E-Graphs are an efficient encoding for discovering and maintaining sets of equalities, commonly adopted in the context of formal proofs, program analysis, and optimization. In several scenarios, equalities may hold only conditionally, i.e., under certain assumptions. For example, in automated provers the proof is often split into multiple branches, such that each branch considers a different set of equalities. A limitation of traditional e-graphs is that they can only encode a single set of equalities at a time. Conditional equalities are then handled by maintaining multiple e-graphs, e.g., one for each branch in the proof, which is inefficient as equalities shared among branches are simply replicated many times.

In this paper, we introduce versioned e-graphs, which efficiently encode multiple equivalence sets at the same time, maximizing shared information among them. We formalize for versioned e-graphs and prove their correctness. We evaluate our solution against widely-adopted solutions which maintain multiple e-graphs and show that versioned e-graphs are up to 5–30 % more memory efficient and up to 4× faster depending on the case study, especially when solution spaces are large both in explored program terms and number of branches.

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, Conditional Reasoning

ACM Reference Format:
Jahrim Gabriele Cesario, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2026. Versioned E-Graphs. *Proc. ACM Program. Lang.* 10, PLDI, Article 171 (June 2026), 23 pages. https://doi.org/10.1145/3808249

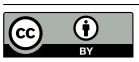
1 Introduction

E-graphs [16] are a core data structure in automated reasoning. They allow efficient maintenance and querying of equivalences among terms based on a set of equivalence rules. Specifically, if two terms are equivalent under these rules, applying the same operation to both will produce new terms that are also equivalent, thanks to congruence. E-graphs have been applied across various fields, including program optimization [24] and synthesis [4] and decision procedures [2, 9].

They play a crucial role for *equality saturation* [17, 24], which systematically rewrites program terms to explore equivalent forms. By organizing terms into a compact representation, e-graphs enable efficient exploration of a large solution space while preventing redundant computations.

Traditionally, e-graphs encode a single global equality relation, which limits their effectiveness in branching contexts such as symbolic execution or proof by case analysis. For example, in automated theorem proving, proofs are often organized across *cases*, such that each proof *branch* evolves into

Authors' Contact Information: Jahrim Gabriele Cesario, University of St. Gallen, St. Gallen, Switzerland, jahrimgabriele.cesario@unisg.ch; 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; Guido Salvaneschi, University of St. Gallen, St. Gallen, Switzerland, guido.salvaneschi@unisg.ch.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART171
https://doi.org/10.1145/3808249

a different set of valid equalities starting from a common base set. These contextual equalities are then handled by maintaining multiple e-graphs, e.g., one for each branch in the proof.

State-of-the-art theorem provers [14, 20] clone the entire e-graph for each proof goal before proceeding with the sub-proofs. Hence, equalities that hold unconditionally (e.g., those that are valid in all branches of the proof) are replicated many times, leading to excessive memory consumption and poor run time performance due to cloning. For this reason, recent research explored support for branching in e-graphs that leverages sharing of common elements to improve memory efficiency [8, 21]. Yet, to the best of our knowledge, no existing approach achieves *full sharing*, i.e., storing every term at most once in the graph across all contexts, and no existing approach provides a formal model to reason about the data structure and its correctness.

In this paper, we introduce *versioned e-graphs*, a data structure that encodes a hierarchy of equivalence relations on top of a shared term space. The hierarchy is encoded as a *version tree* that reflects on the underlying program code or the corresponding proof structure, where every version corresponds to a branch and the equivalence relation of each version is an extension to the equivalence relation of its parent version. Overall, this design improves efficiency by avoiding redundancy in encoding shared equalities across versions and improves performance compared to the state of the art [21] while, in contrast to the existing implementation, fully supporting *e-class analysis* [27], a technique to attach analysis data (that forms a semilattice) to each e-class.

We prove our approach to versioned e-graphs correct and realize it in the Veg implementation. We compare the performance of our implementation with traditional and state-of-the-art approaches. We conduct a parameter analysis on the impact of the number of versions and terms on memory and run time performance. We implement an EUF solver [13], which uses versioned e-graphs to reason about unquantified boolean propositions over equalities between uninterpreted functions. We modify the automated theorem prover TheSy [20] to compare directly against the state-of-the-art Easter Egg [21] approach. Our evaluation shows that versioned e-graphs generally improve performance in terms of memory (up to 5–30 %) and run time (up to 4×).

In summary, this paper makes the following contributions.

- We introduce *versioned union-finds*, an optimized data structure to encode multiple equivalence relations (Section 3).
- We analyze the amortized time and space complexity of the versioned union-find and demonstrate that it outperforms the cloning approach. (Section 4).
- We build *versioned e-graphs* with full sharing across versions on top of *versioned union-finds*, extending their equivalence relations with congruence (Section 5).
- We provide a *formalization* of versioned union-finds and versioned e-graphs and prove them correct (Section 6).
- We implement these ideas both in an optimized standalone Rust library Veg, largely inspired by the popular *egg* library (Section 7).
- We generalize persistent union-finds [7] to *persistent e-graphs* by using standard persistent data structures (Section 7), making cloning memory efficient through structural sharing.
- We evaluate the performance of versioned e-graphs and persistent e-graphs against cloning and the state of the art, showing that they scale better in memory and time (Section 8.1), improve memory and run time performance in verification use cases (Section 8.2), and support common e-graph extensions without a performance penalty (Section 8.3).

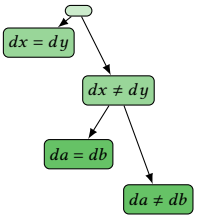
2 Background

In this section, we describe the *union-find* data structure [10, 11, 23] (Section 2.1) to encode equivalence relations. Based on that, we introduce *e-graphs* [16] (Section 2.2).

Running Example. We will use the `dist_diffs` function (Listing 1) as a running example, which computes the distance of the differences `dx` and `dy`, given a distance metric `f`. For example, `f` is the Euclidean distance if it computes the square of its argument (e.g., used for the "sum of squares" loss function) or the Manhattan distance if it computes their absolute value (e.g., used for the "sum of absolute differences" loss function). The code performs an optimized addition of `f(dx)` to `f(dy)`. If `dx == dy` or `f(dx) == f(dy)`, the code doubles `f(dx)` through a bit shift. Otherwise, it adds `f(dx)` to `f(dy)`.

Listing 1. Running Example.

```
1  fun dist_diffs(f, dx, dy) =
2    if dx == dy then
3      f(dx) << 1
4    else
5      da = f(dx)
6      db = f(dy)
7      if da == db then
8        da << 1
9      else
10       da + db
```



A property which we may want to prove is commutativity of the `dx` and `dy` arguments, i.e., `dist_diffs(f, dx, dy) = dist_diffs(f, dy, dx)`. The proof proceeds by case analysis over the program's branches (Lines 3, 8 and 10), showing the property holds in each. Branch-specific assumptions arise from `if` conditions and accumulate across nested lexical scopes, where each (nested) scope is represented as a (sub)version in the versioned e-graph. The green boxes in the figure visualize the nested lexical scopes that correspond to the tree structure of the versions in the proof (right side).

2.1 Union-Find

A *union-find* [10, 11, 23] efficiently represents equivalence classes as trees, the roots being the *representatives* of the classes. Edges denote that (symmetric and transitive) equivalence of elements. Initially, each element is in its own class (reflexively). The `union` operation merges two classes and `find` returns the representative of an element. With the *path compression* optimization, each `find` updates the queried element to point to the representative, reducing the cost of future `find` calls.

Running Example. Figure 1 shows the union-find for the running example (Listing 1) in the branch where `dx == dy` does not hold but `da == db` does. In subfigure A, `f(dx)` is unioned with `da` because of the binding `da = f(dx)` in the code. Similarly, `f(dy)` is unioned with `db` and `db` with `da`, making `dx`, `dy` and `da` the representatives of the equivalence classes. In B, path compression shortens the path from `f(dy)`, making every element point directly to its equivalence class.

2.2 From Union-Find to E-Graph

Beyond the equivalences encoded in a union-find, *e-graphs* (equivalence graphs) [16] ensure that function applications stay congruent when adding equivalences – the *congruence invariance* property, e.g., if $x = y$, then $f(x) = f(y)$. Their graph structure consists of *e-nodes* (program terms expressed as function applications) grouped into *e-classes* (sets of equivalent terms).

An e-graph's `find` retrieves the equivalence class of a term, and `union` merges two classes. Operationally, unioning an e-class α with β moves α's e-nodes into β, replaces all occurrences of α with β in e-node arguments, and recursively unions all e-classes containing the same e-node.

E-graphs are commonly built on top of a union-find in two ways: Either (1) e-nodes are the union-find elements and e-classes are the union-find equivalence classes or (2) the e-graph uses a *hash-cons* to map each e-node to its original e-class and the union-find maintains an equivalence relation over e-classes instead of e-nodes. After unioning, congruence invariance is recovered through a *rebuilding* process, for which a common approach is *e-node canonicalization*, which replaces every argument of an e-node with its representative.

Running Example. Figure 2 shows the e-graph for the running example (Listing 1) in the branch where `dx == dy` holds. Solid arrows denote membership of e-nodes in e-classes – transitively pointing to the canonical – and dashed edges denote the arguments to e-node function application. In

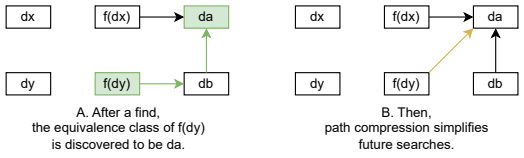


Fig. 1. Union-Find Example.

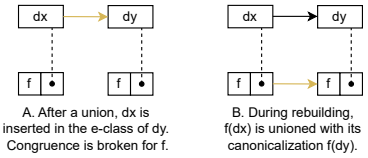


Fig. 2. E-Graph Example.

subfigure A, `dx` is unioned with `dy` due to the branch condition, breaking congruence of the function `f`. In B, rebuilding restores congruence by unioning `f(dx)` with its canonicalization `f(dy)`.

3 Versioned Union-Find

Despite their generality, union-finds (and e-graphs) are limited by their inherent single-version model – they maintain one global equivalence relation. This is a poor fit for verification tasks involving branching logic, where each proof branch is based on a shared set of equivalent terms but evolves independently, aggregating potentially different terms to be equivalent in each branch while still retaining many shared elements. In these cases, maintaining multiple data structures is inefficient as shared information among them may be replicated many times. For this reason, we propose a novel *versioned union-find* to support multiple logical contexts within a shared structure, and then extend it to *versioned e-graphs* to support congruence.

Versioned union-finds extend union-finds to encode a *version tree* of equivalence relations. Each node in the tree represents a version and all the equivalences at a version also hold at its descendant versions. For example, we may discover that a certain set of equalities holds before an if-then-else-branch in a program. If the current version is labeled 0.1, we label the then-branch 0.1.1 and the else-branch 0.1.2. All the equalities that hold before branching (0.1) still hold in both branches (0.1.1 and 0.1.2), but we may discover different additional equalities in both branches. Therefore, we say that versions 0.1.1 and 0.1.2 *depend on* version 0.1, while versions 0.1.1 and 0.1.2 are *independent*. A versioned union-find labels its edges with the versions in which the connected nodes are in the same equivalence class. Thus, a versioned union-find is a labeled directed graph with *version edges*.

Listing 2 shows the versioned union-find algorithm by highlighting (in green) the changes to the standard union-find (in black). A `Node` contains a domain `Element` and a `parent` field pointing to the parent `Node` in the union-find. In the versioned union-find, `parent` is indexed by a `Version`. Self-references (within a version) indicate the representatives of an equivalence class (within the version). `Version` has a `parent` and a `children` field to navigate the version tree. A versioned union-find initially contains a single version, the *root version*.

The `find` operation (Line 1) returns the representative element (within a given version). As nodes may have different parents in different versions – due to different union operations performed in

Listing 2. Versioned Union-Find Algorithm.

```
1  fun vuf.find(x: Node, v: Version): Node =
2    p = vuf.cvp(x, v) // closest valid parent
3    x' = if p != x then vuf.find(p, v) else x
4    vuf.add_edge(x', x, v)
5    x'
6
7  fun vuf.union(x: Node, y: Node, v: Version) =
8    x' = vuf.find(x, v)
9    y' = vuf.find(y, v)
10   for sv in v.children do
11     vuf.union(x, y, sv)
12   vuf.add_edge(x', y', v)
13 fun vuf.add_node(e: Element): Node =
14   x = Node(e)
15   vuf.add_edge(x, x, Version.Root)
16   x
17
18 fun vuf.add_edge(p: Node, c: Node, v: Version) =
19   c.parent[v] = p
20
21 fun vuf.cvp(x: Node, v: Version): Node =
22   if v in x.parent
23   then x.parent[v]
24   else vuf.cvp(x, v.parent)
```

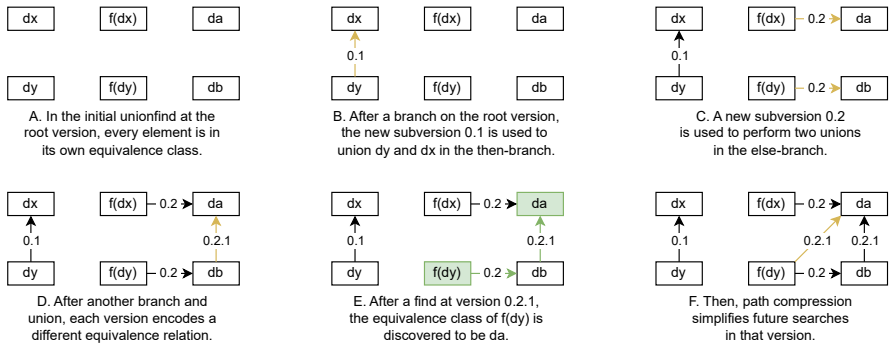


Fig. 3. Versioned Union-Find Example.

each version – finding the equivalence class of an element in a version requires following *valid* version edges (Line 21), i.e., edges labeled with the version itself or its closest ancestor in the version tree. A representative is reached when no more valid edges can be followed.

Crucially, union-finds perform the *path compression* optimization: Each `find` call updates the queried element to point directly to the representative of its equivalence class (within a given version), hence reducing the cost of future `find` operations (Line 4). In versioned union-finds, this optimization not only shortens the path traversed by subsequent finds, but also the searches for valid version edges in the version tree, i.e., the closest ancestor becomes the version itself.

The `union` operation (Line 7) makes the representative of one class a child of the representative of the other (within a given version). To preserve the ancestor relation among versions, unions must first be applied to all children recursively (Lines 10 and 11).

Running Example. Figure 3 shows an example of a versioned union-find based on the running example (Listing 1). In subfigure A, we show the initial versioned union-find for the domain of elements {`dx`, `dy`, `f(dx)`, `f(dy)`, `da`, `db`} where every element belongs to its own equivalence class in the root version. In B, `dx` is unioned with `dy` in the new child version 0.1 created for the first then-branch. In C, `f(dx)` is unioned with `da` and `f(dy)` with `db` in the other child version 0.2, due to the bindings in the code snippet. In D, `da` is unioned with `db` in child version 0.2.1 for the then-branch of the second if-then-else branching. In E, the equivalence class of `f(dy)` in version 0.2.1 is found to be `da`. Finally, in F, path compression shortens the paths traversed from element `f(dy)` in version 0.2.1.

Cycles. Our model allows unions in a version to be visible to all descendant versions. Naïvely supporting these unions, however, can cause infinite loops when finding the equivalence class of an element in the descendant version afterward.

Our algorithm breaks such cycles by keeping track of the representative of every equivalence class in each version separately. The traversal during `find` then stops as soon as a representative in the given version is reached (even if more valid edges could still be followed). In Listing 2, keeping track of versioned representatives is for free because `union` does not skip adding edges for equal elements (Line 12). Further, full path compression (Line 4) adds a self-referencing version edge to the representative. Note that an optimization to reduce the memory footprint of self-referencing version edges is selectively adding them just before unions.

Figure 4 presents an example for such a cycle – loosely based on our running example (Listing 1) but with a contrived sequence of unions and finds to expose the issue. The figure shows three

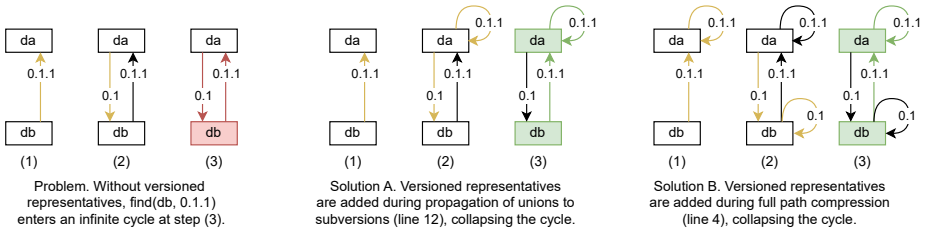


Fig. 4. Example of a Cycle and Cycle Avoidance in a Versioned Union-Find.

algorithms performing the same sequence of operations: (1) `da` is unioned with `db` in the child version 0.1.1, (2) `db` is unioned with `da` in its parent 0.1, (3) the equivalence class of `db` is queried in the child 0.1.1. In the Problem subfigure, we show an algorithm that does not track versioned representatives and enters an infinite loop when attempting to find the equivalence class of `db` in version 0.1.1. Solutions A and B show two alternative algorithms that collapse the infinite loop by maintaining versioned representatives and correctly determine the equivalence class of `db` to be `da` in version 0.1.1. Solution A adds versioned representatives as part of propagation to descendant versions during unions, while Solution B does it as part of full path compression. In its simplicity, our pseudocode in Listing 2 implements both solutions at Line 12 and Line 4 respectively.

Note that cycles can be avoided altogether by restricting the model to allow modifications only to leaf versions, i.e., forbidding modifications to any version once it has children. The restricted model also allows avoiding propagation of unions to descendant versions (Listing 2, Lines 10 and 11).

4 Analysis of the Versioned Union-Find

In this section, we first provide a correctness argument for versioned union-find. Second, we present an amortized complexity analysis in line with the classical complexity results [23]. We first analyze the best- and worst-case amortized space complexity of versioned union-finds, then the amortized time complexity of versioned union-finds by simulating it with a classical union find.

4.1 Correctness

This subsection outlines arguments for the correctness of the versioned union-find in Listing 2. Consider a user-defined list of versioned equalities, where each equality is a triple of two elements and a version, denoting that the two elements are equal in the given version. We consider the versioned union-find *correct* if it represents, in every version `v`, the smallest reflexive, symmetric, and transitive closure over the list, restricted to the equalities at `v` or any ancestor of `v`.

Two elements `a` and `b` are equivalent at version `v`, if and only if `find(a, v)` and `find(b, v)` are equal. This relation is reflexive, symmetric, and transitive by virtue of equality being so. The smallest closure property is addressed in Claim 1 and the restriction on the list in Claim 2 and 3.

The arguments in this section rely on `find` performing path compression (as defined in Listing 2) as it simplifies them. Path compression, however, is not required to guarantee the correctness of the versioned union-find. The mechanized proofs [6] of the theorems (Section 6), particularly the correctness theorem Theorem 6.10, do not make use of path compression.

Claim 1 (Smallest Closure Property). *Given `a`, `b`, and `x`. Let `a'` be `find(a, v)`, `b'` be `find(b, v)`, and `x'` be `find(x, v)`. If `a'`, `b'`, and `x'` are all distinct, after `union(a, b, v)` then `find(b, v)` remains equal to `b'`, `find(a, v)` becomes equal to `b'` (Closure), and `find(x, v)` remains equal to `x'` (Smallest).*

Proof Sketch. Let x_{i+1} be `cvp(`x_i`, v)` with x_0 being `x`, and let a_{i+1}, a_0 and b_{i+1}, b_0 be defined similarly. By the definition of `find(x, v)` it follows that `x'` is the constant point of the sequence x_i, i.e. the

point at which the sequence becomes constant, by inlining `p` at Line 2 in Line 3. Similarly, `a'` is the constant point of the sequence a_i and `b'` of the sequence b_i. By the distinction assumption, no two of the three sequences share an element, otherwise they would have the same constant point.

After the union, all newly added edges labeled with `v` are a result of the union itself (Line 12) and path compression (Lines 8 and 9 and Line 4). Other edges are added, but these are labeled with descendants of `v` (Line 11). Since `cvp`, and thus `find`, only inspects edges labeled by `v` (Line 23) or its ancestors (Line 24), only those newly added edges at `v` are relevant to the claim.

Closure. Thanks to path-compression, `cvp(`b_i`, v)` is `b'` for every i, and `cvp(`a_i`, v)` is `a'` for every i except when a_i is `a'` where `cvp(a',v)` is `b'`. Thus, `find(a, v)` and `find(b, v)` are both equal to `b'`.

Smallest. Moreover, `cvp` at `v`, and thus `find`, remain constant for all elements not in either sequences a_i and b_i. Since the sequence x_i shares no element with the sequence a_i and b_i then `find(x, v)` remains equal to `x'`. □

To analyze how a change in one version influences another, we consider two cases: either a union happens at an ancestor version (Claim 2), in which case its effect is observed at the descendant version; or not (Claim 3), in which case the union happens at a descendant or independent version and its effects must not be observed.

Claim 2 (Union at Ancestors). *Let version v_1 be an ancestor of v_2, after `union(a, b, `v_1`)` then `find(a, `v_2`)` is equal to `find(b, `v_2`)`.*

Proof Sketch. Observe that during `union(a, b, `v_1`)`, a call to `union(a, b, `v_2`)` will eventually be invoked during the recursive propagation (Lines 10 and 11). Let `a'` be `find(a, `v_2`)` as it occurs on Line 8 during `union(a, b, `v_2`)`. Similarly, let `b'` be `find(b, `v_2`)` as it occurs on Line 9 during the same union. Thanks to path compression (Line 4) there exists an edge e_1 labeled with v_2 from `a` to `a'` and similarly another edge e_2 from `b` to `b'`. Thanks to Line 12 there exists a third edge e_3 labeled with v_2 from `a'` to `b'`.

Observe that, for every descendant `v` of v_1, exactly one call to `union` and two calls to `find` happen at `v`. Moreover, if an edge is added at `v` then it can only be removed through path compression at `v`. Therefore, after all the unions are done, both e_2 and e_3 remain.

After `union(a, b, `v_1`)` is done, to prove the equality of interest we consider two cases. First, if e_1 was removed, then it must be during `find(b, `v_2`)` which occurred during `union(a, b, `v_2`)` (Line 9). In which case, `find(a, `v_2`)` was already equal to `find(b, `v_2`)` completing the proof.[1] Second, if e_1 remained, `find(a, `v_2`)` is `find(cvp(a, `v_2`),`v_2`)` by definition of `find` (by inlining `p` at Line 2 in Line 3). The latter is `find(a',`v_2`)` since adding e_1 modified the parent of `a` at v_2 (Line 19) and `cvp` returns the parent of `a` at v_2 (Line 23). Next, it is equal to `find(cvp(a',`v_2`),`v_2`)` by definition of `find`, which is `find(b',`v_2`)` since e_3 has been added. On the other hand, `find(b, `v_2`)` is equal to `find(cvp(b, `v_2`),`v_2`)` by definition of `find`, which is equal to `find(b', `v_2`)` since e_2 has been added. By transitivity, `find(a, `v_2`)` and `find(b, `v_2`)` are equal. □

Claim 3 (Union at Non-Ancestors). *Let version v_1 not be an ancestor of v_2, i.e. v_2 is a strict ancestor of v_1 or they are independent, and `find(a, `v_2`)` is different from `find(b, `v_2`)`, after `union(a, b, `v_1`)` then `find(a, `v_2`)` remains different from `find(b, `v_2`)`.*

Proof Sketch. Recall from the proof of Claim 1 that all edges added after `union(a, b, `v_1`)` are labeled with descendants of v_1. All of these versions are either descendants of or unrelated to v_2.

[1] This case is impossible. If `a` and `b` were not in the same equivalence class during `union(a, b, `v_2`)` then `find(b, `v_2`)` could not remove the edge. And if they were in the same equivalence class then e_1 would have been replaced by an identical edge.

Crucially, none are ancestors of v_2. Also recall that `cvp`, and thus `find`, only inspects ancestor versions, implying that `cvp` remains constant at v_2, from which the goal follows. □

4.2 Space Complexity

For the space complexity we assume a versioned union-find is constructed using unions and version additions. We assume that these operations are done such that every union comes after the addition of the version it unions at, and that every version addition comes after the addition of the parent version, unless the parent version is the root one.

Theorem 4.1. *A versioned union-find constructed from m unions and v version additions consumes between $O(m \log m + (v + m) \log v)$ and $O(vm(\log m + \log v))$ bits.*

Proof. In the best case, unions never propagate to descendant versions. This happens when unions only ever operate on leaf versions. Each of the v version additions adds one edge in the version tree which is represented as two pointers to versions, each pointer costing $\log v$ bits. In the worst case, all m unions introduce new elements to the union-find thus requiring $\log 2m$ bits for a pointer to an element. Each of the m unions is guaranteed to add a single edge thanks to the ordering described above. Each edge has two elements and one version pointer (its label) costing $2 \log 2m + \log v$ bits. In total, $O(m \log m + (v + m) \log v)$ bits are spent. In the truly best case, unions only ever operate on the same element, thus only adding one edge for the first union and leading to a total cost of $O(v \log v)$ bits, but this is a contrived scenario.

In the worst case, unions always propagate. This happens when all versions are added before applying any unions, all of which unioning at the root version. Each of the m unions now adds v edges thus costing $v(2 \log 2m + \log v)$ bits. In total, $O(vm(\log m + \log v))$ bits are spent. □

In the worst case, the versioned union-find consumes as much memory as cloning the union-find for each version, but in the best case it is linearly better in the number of versions.

4.3 Time Complexity

The classical theorem for the time complexity of the classical union-find is that path compression makes find $O(1)$ amortized – eventually, when enough calls to find have been made and when all paths have been compressed, the parent of every node is the root. The other optimization, union-by-rank, achieves the same amortized run time for find but earlier – the threshold after which a find becomes constant is smaller.

We have only presented the path-compression optimization for the versioned union-find. We will show that find is $O(1)$ amortized even though the versioned find must first traverse the version tree to find the closest valid parent before recursing (Line 1 of Listing 2).

We start by proving a variant on the classical time complexity theorem.

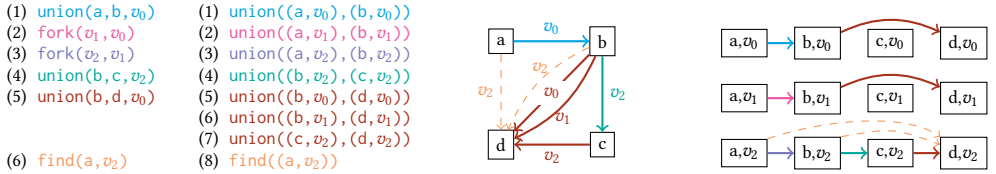
Lemma 4.2 (Classical Time Complexity). *A sequence of m unions and k finds on the classical union-find with path-compression takes*

$$O((k + m) \max(1, \log(m^2/(k + m))/\log(2(k + m)/m)))$$

Proof. In [23] the time complexity is $O(m \max(1, \log(m^2/k)/\log(2k/m)))$ for sequences of m unions *at roots* and k finds. When unions do not assume their arguments are roots and instead compute them by calling find (Lines 8 and 9 of Listing 2), each union calls find twice. Specializing the theorem in [23] to m unions and $k + 2m$ finds we recover the statement of this theorem. □

Observe that when $k \geq m(m - 1)$ distinct finds are made, the run time cost becomes $O(k)$ justifying the $O(1)$ amortized time complexity of find.

Next, we state and prove the time complexity theorem for versioned union-finds.



(a) Sequence. (b) Translation. (c) Versioned. (d) Simulated.

Fig. 5. Operations (a) in the versioned union-find (c) and their translation (b) in the classical union-find (d).

Theorem 4.3 (Versioned Time Complexity). *A sequence of m unions, k finds, and v version additions on the versioned union-find with path-compression takes*

$$O((k + mv)\max(1, \log(m^2v^2/(k + mv))/\log(2(k + mv)/mv)))$$

Proof. The proof proceeds as follows. We propose a translation of sequences of versioned union-find operations into sequences of classical union-find operations such that the classical union-find simulates the versioned one. In particular, we will show that this induces an injection from the edges of the versioned union-find to the edges of its equivalent classical union-find. This injection demonstrates that the cost of running the sequence on the versioned union-find is upper-bounded by the translated sequence on the classical union-find. In summary, the proof builds a simulation and shows that the cost of running the simulation is higher.

The elements of the classical union-find in which the simulation is run are pairs of the elements of the versioned union-find and versions. The simulation comes with the following invariant: edges between (x_1, v_1) and (x_2, v_2) are only allowed when $v_1 = v_2$. Given a sequence of m unions, k finds, and v version additions, we translate each versioned find $find_v(x, v)$ into the classical find $find_c((x, v))$ (we will use the subscript v and c to distinguish the operations done on the versioned union-find and the classical one). Figures 5a and 5b provide an example of translated operations. A priori, this translation is not correct: the closest valid parent of x at v may not be at v and $find_v$ will have to find the parent at earlier versions. The cost of this operation can be preemptively accounted for if all unions at versions earlier than v also exist at v. This is done by translating each version addition into a sequence of all the earlier unions but made on v (Operations (2) and (3) in Figure 5a correspond to operations (2), (3), and (4) in Figure 5b). Observe that version addition, a constant time operation, is translated into mv unions in the worst case. Finally, each versioned union must propagate unions to all descendant versions (Operation (5), which is translated into (5), (6) and (7)), and, at each version, it must perform two versioned finds and add one edge, i.e., at each version it does (the equivalent of) one classical union. Figures 5c and 5d show the versioned and the simulated union-finds for the given example operations. Edges that would be introduced by path compression are dashed.

Therefore, a sequence of m versioned unions, k versioned finds, and v version additions corresponds to $k + 2mv$ finds and $2mv$ unions. The statement of this theorem can be recovered by specializing Lemma 4.2 with the earlier numbers. □

In the worst case, the versioned union-find with v versions costs the same as maintaining v separate classical union-finds, meaning that its expected runtime can only be better than cloning.

4.4 Boundary Cases

Examples for the best case run time and memory are sequences of operations where only one kind of operation is performed. We achieve the best case for run time using only find operations or only adding versions, since such operations only traverse or add, respectively, a single edge. We

achieve the best case for memory using only find or union operations on existing elements, since, without multiple versions, both operations only move existing edges without adding new ones. An example for the worst case for both run time and memory is when propagation is maximal, e.g., a number of versions added, followed by a number of unions at the root version, causing all unions to propagate to all versions In such degenerate cases, our versioned union-find performs like cloning the union-find for each version and applying all operations to all clones. Thus, we conclude that versioned union-finds match cloning in the worst case and outperform it otherwise, and that the key to achieving the best-case complexity is to avoid propagation of unions to child versions.

5 Versioned E-Graphs

In this section, we present versioned e-graphs by building on top of versioned union-finds, and adapting the e-graph operations to be version-aware.

A *versioned e-graph* is an efficient representation of tree-structured equivalence relations among congruent functions, building on versioned union-finds (as e-graphs build on union-finds), where finding, unioning, canonicalization, and rebuilding are parametrized over the given versions.

Listing 3 shows the versioned e-graph algorithm by highlighting (in green) the changes to the standard e-graph (in black). The e-graph contains a versioned union-find (Line 1), which maintains the equivalence relations among e-classes across versions, and a hash-consing map (Line 2), which maps every `ENode` to its corresponding original `EClass`. An `ENode` consists of an operator and a list of `EClass` arguments. An `EClass` is a pointer to a `Node` in the underlying versioned union-find.

The `find` operation is a simple wrapper around the versioned union-find's `find` (Line 5). In particular, we use `find` to define the `canon` operation, which canonicalizes an `ENode` by replacing its arguments with their representatives in the given version (Line 27).

The classical `add` operation canonicalizes the `ENode` (Line 10), adds it to the hash-cons (Lines 11 and 12) and returns the corresponding `EClass`. For versioning, we need to propagate to all versions to maintain congruence invariance: the `ENode` is first canonicalized at the root version (Line 8), then propagated to all versions (Line 9), canonicalizing the `ENode` at each version (Line 10) and adding it to the hash-cons (Lines 11 and 12), and finally unioned with the existing `EClass` (Line 13).

The classical `union` operation delegates to the union-find's `union` (Lines 16 and 20) and *rebuilds*, i.e., restores congruence invariance by unioning every `ENode` with its canonicalization (Lines 21 to 24). An ordering (Line 19) is enforced to ensure termination of this rebuilding process for cyclic terms. For versioning, we also propagate the operation to all descendant versions (Lines 17 and 18).

Running Example. Figure 6 shows a versioned e-graph based on the running example (Listing 1). In subfigure A, each e-node is in its own e-class. In B, the e-classes of `dx` and `dy` are unioned in version

Listing 3. Versioned E-Graph Algorithm.

```
1  vuf = VersionedUnionFind()
2  hashcons = Map<ENode, EClass>()
3
4  fun veg.find(x: EClass, v: Version): EClass =
5    vuf.find(x, v)
6
7  fun veg.add(n: ENode): EClass =
8    n = veg.canon(n, Version.Root)
9    for v in versions do
10     nv = veg.canon(n, v)
11     if not nv in hashcons then
12       hashcons[nv] = vuf.add_node(nv)
13     veg.union(hashcons[n], hashcons[nv], v)
14   hashcons[n]
15 fun veg.union(x: EClass, y: EClass, v: Version) =
16   (x, y) = (veg.find(x, v), veg.find(y, v))
17   for sv in v.children do
18     veg.union(x, y, sv)
19   (child, parent) = if x < y then (x,y) else (y,x)
20   vuf.union(child, parent, v)
21   for (n, c) in hashcons do // rebuilding
22     if child in n.args then
23       c0 = veg.find(veg.add(veg.canon(n, v)), v)
24       if c != c0 then veg.union(c, c0, v)
25
26 fun veg.canon(n: ENode, v: Version): ENode =
27   args = n.args.map(fun c => veg.find(c, v))
28   ENode(n.op, args)
```

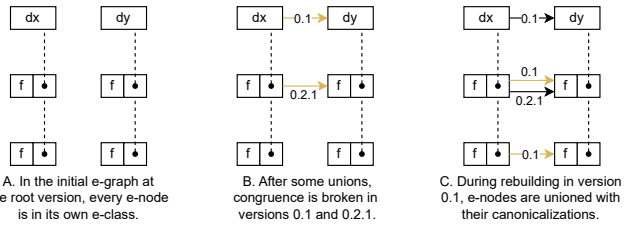


Fig. 6. Versioned E-Graph Example.

0.1 (then-branch of the first if-then-else), and `f(dx)` with `f(dy)` in version 0.2.1 (then-branch of the second if-then-else), temporarily breaking congruence in both versions. In C, congruence is restored in version 0.1, discovering `f(dx) == f(dy)` – it is not yet restored in version 0.2.1. Importantly, all e-nodes are shared across versions, ensuring *full-sharing* of the term space.

Extensions. Many classical e-graph extensions can easily be ported to versioned e-graphs, including deferred rebuilding, e-class analysis, e-matching, equality saturation, and extraction. The insight is that they are designed to work on a single e-graph, and thus can be adapted to operate on a single version of a versioned e-graph, i.e., using `find` and `union` operations at a fixed version.

We support the same *deferred rebuilding* introduced by *egg* [27], by tracking performed unions and dependencies between e-nodes and e-classes. Differently from *egg*, we avoid canonicalizing the global hash-cons at a specific version, as it would cause inconsistencies in other versions. This memory optimization would require duplicating e-nodes across versions, which would defeat the purpose of the global hash-cons in the first place.

We support egg's *e-class analysis* to maintain additional user-defined information on e-classes. We maintain a separate set of analyses for each e-class for each version, since the analyses may diverge across versions. While separating analyses per versions incur memory overhead, it is necessary for correctness. As an instance of analysis, we implemented versioned *disequality edges* [16, 30] to efficiently represent disequalities among e-classes in versioned e-graphs, which is an important optimization in automated theorem proving [31, 32].

We also support *e-matching*, *equality saturation*, and *greedy extraction*. These extensions require a cache mapping e-classes to their e-node members at a specific version. We compute this cache on-demand and then maintain it incrementally during adds and unions. Thanks to versioning, we can also maintain only the cache for hot versions and drop it for cold versions, thus saving memory.

Versioned e-graphs lend themselves to extensions specific to versioning that make use of the additional information encoded through version relations to distribute resources more effectively across versions and to implement version-specific operations. One simple example is *version removal*, where versioned e-graphs generalize over traditional push/pop approaches [3] for handling conditionals and backtracking, e.g., in SMT solvers. We also see the opportunity to design more efficient and expressive version-aware extensions, e.g., rewrite rules that can populate new versions during equality saturation, or versioned proof production [29].

6 Formalism

In this section we develop the necessary formalism for versioned union-finds (Section 6.1) and versioned e-graphs (Section 6.2) to prove the two main correctness theorems: (1) versioned union-finds are the smallest reflexive, symmetric, and transitive closure of a set of equalities (Theorem 6.10) and (2) versioned e-graphs are the smallest reflexive, symmetric, transitive, and congruent closure (Theorem 6.19). The paper's artifact [6] contains mechanized proofs in Lean4.

6.1 Versioned Union-Find

In Definition 6.1 we define the version tree.

Definition 6.1 (Version Tree $\mathcal{V}$). A version tree $\mathcal{V}$ a quadruple $(V, root, parent, depth)$ with V being the set of versions in the tree, $root$ being the root of the tree, $parent : V \to V$ mapping each node to its unique parent (we assume $parent(root) = root$), and $depth : V \to \mathbb{N}$ mapping each node to its depth in the tree, i.e. the length of the path from the root to the node.

We denote the children of a node v with $children(v) = \{v' \ : \ parent(v') = v\}$, and the ancestor relation with $v \preceq v'$ (reads "v is ancestor of v'").

Next, in Definition 6.2 we define versioned union-find.

Definition 6.2 (Versioned Union-Find $\mathcal{U}$). The set of versioned union-finds $\mathcal{S}_{\mathcal{U}}$ contains all quintuples $\mathcal{U}$ of the form $(\mathcal{V}, U, E, parent, depth)$, such that the following holds:

(1) $\mathcal{V} = (V, root, parent_{\mathcal{V}}, depth_{\mathcal{V}})$ is a version tree,
(2) U is a finite set of elements in the versioned union-find,
(3) $E \subseteq U \times V \times U$ is the set of version-labeled directed edges from elements to their parents,
(4) $E \ni (x, v, y)$, for every $(x, v) \in U \times V$ and at most one $y \in U$,
(5) $E \ni (x, root, y)$, for every $x \in U$ and exactly one $y \in U$,
(6) $parent : U \times V \to U$ is the closest valid parent (`cvp` in Listing 2),
(7) $parent(x, v) = y$, for every $(x, v, y) \in E$,
(8) $parent(x, v) = parent(x, parent_{\mathcal{V}}(v))$, for every $(x, v, y) \notin E$. Together with (7), the definition is well-founded, by observing that the $depth_{\mathcal{V}}$ of the version decreases, with the base case being handled by (5),
(9) $depth : U \times V \to \mathbb{N}$ is the depth of some element at some version in the forest of $\mathcal{U}$,
(10) $depth(x, v) = 0$, for every $(x, v) \in U \times V$ such that $x = parent(x, v)$,
(11) $depth(x, v) = depth(parent(x, v), v) + 1$, for every $(x, v) \in U \times V$ such that $x \neq parent(x, v)$,
(12) For every $x, y \in U$ and $v \in V$, if $find(x, parent_{\mathcal{V}}(v)) = find(y, parent_{\mathcal{V}}(v))$ then $find(x, v) = find(y, v)$, where $find$ is as defined later in Definition 6.3. This is the expectation that the all the equivalences at a version also hold at its descendants.

The last requirement of Definition 6.2 is needed to forbid constructions such as Figure 7, where an equality at version 0.1 does not hold in its child version 0.1.1: in the parent $find(a, 0.1) = e = find(b, 0.1)$, but in the child $find(a, 0.1.1) = c \neq d = find(b, 0.1.1)$.

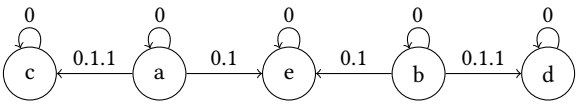


Fig. 7. A versioned union-find that satisfies all the requirements of Definition 6.2 but the last one.

Now, we complete the definition by introducing $find$.

Definition 6.3 ($find$). Given a versioned union-find $(\mathcal{V}, U, E, parent, depth)$, $find : U \times V \to U$ is defined as $find(x, v) = x$ if $x = parent(x, v)$ and $find(x, v) = find(parent(x, v), v)$ otherwise.

Its termination is guaranteed by observing that the $depth$ of the first argument decreases with the base case being handled by property (10) of Definition 6.2.

From $find$, we define the equivalence relation encoded by a versioned union-find at some version. Here and below, the dot notation explicitly indicates the data structure an operation applies to.

Definition 6.4 ($\equiv_v$). Given a versioned union-find $\mathcal{U}$ and a version v, the relation $\equiv_v \subseteq U \times U$ is:

$$x \equiv_v y \text{ in } \mathcal{U} \quad \text{if and only if} \quad \mathcal{U}.find(x, v) = \mathcal{U}.find(y, v)$$

Theorem 6.5 ($\equiv_v$ is an equivalence relation). *$\equiv_v$ is a reflexive, symmetric, and transitive relation.*

Proof. Reflexivity, symmetry, and transitivity follow from those of equality. □

The formalism in [6] includes a *find* with path compression. We omit it here since it is not required for the correctness of the versioned union-find.

To help with the following definitions, we introduce the descendant relation $x \sqsubseteq_v y$ (reads "x is a descendant of y at version v") over the elements of the versioned union-find at a given version v.

Definition 6.6 ($\sqsubseteq_v$). Given a versioned union-find $(\mathcal{V}, U, E, parent, depth)$, a version $v \in V$, and $x, y \in U$, then $x \sqsubseteq_v y$ if and only if $parent_v^{(d)}(y) = x$ for some $d \geq 0$ where $parent_v = parent(-, v)$.

Next we define how to union elements at some version. We assume that the arguments of *union* are roots to simplify the presentation without loss of generality.

Definition 6.7 (union). Given a versioned union-find $\mathcal{U}_0 = (\mathcal{V}, U, E, parent, depth)$, a version $v_0 \in V$ with $children(v_0) = \{v_1, \ldots, v_n\}$, and roots a and b at v_0, then $union : U \times U \times V \to \mathcal{S}_\mathcal{U}$ is:

$$\mathcal{U}_0.union(a, b, v_0) = (\mathcal{U}_n.\mathcal{V}, \mathcal{U}_n.U, \mathcal{U}_n.E \setminus \{(a, v_0, \mathcal{U}_n.parent(a, v_0))\} \cup \{(a, v_0, b)\}, parent', depth')$$

where

$$parent'(x, v) = \begin{cases} b & \text{if } x = a \wedge v = v_0 \\ \mathcal{U}_n.parent(x, v) & \text{otherwise} \end{cases}$$

and

$$depth'(x, v) = \begin{cases} \mathcal{U}_n.depth(x, v) + 1 & \text{if } x \sqsubseteq_v a \wedge v = v_0 \\ \mathcal{U}_n.depth(x, v) & \text{otherwise} \end{cases}$$

and

$$\mathcal{U}_{i+1} = \mathcal{U}_i.union(\mathcal{U}_i.find(a, v_{i+1}), \mathcal{U}_i.find(b, v_{i+1}), v_{i+1}) \qquad \text{for every } 0 \leq i < n$$

Unioning a and b at a version v_0 in the versioned union-find $\mathcal{U}_0$ starts by recursively unioning at every descendant version to obtain $\mathcal{U}_n$. Then, we add an edge from a to b labeled with v after removing any previous outgoing edge from a labeled with v. Finally, as a is now a child of b and is thus no longer a root (if $a \neq b$), we shift the *depth* of every element in the tree of a by one.

Next, we prove that *union* merges the equivalence classes of its arguments at all descendant versions.

Theorem 6.8 (*union* is correct). *Let $\mathcal{U}$, v_0, a, and b be given as in Definition 6.7. For every x, y, and v such that $v_0 \preceq v$, $x \equiv_v a$ in $\mathcal{U}$, and $y \equiv_v b$ in $\mathcal{U}$, then $x \equiv_v y$ in $\mathcal{U}.union(a, b, v_0)$.*

Proof. Follows because *union* merges argument roots, preserves equalities, and is well-defined. □

Now, we are ready to state the main correctness theorem of versioned union-finds: they represent the smallest reflexive, symmetric, and transitive closure of a set of equalities. First, we construct a versioned union-find from the trivial one and a sequence of equalities, proving its equivalence relation to be the reflexive, symmetric, and transitive closure of the sequence.

Definition 6.9 (Closure). Let $\mathcal{E} = e_0 :: \ldots :: e_{n-1}$ be a sequence of equalities at a version, where $e_i \in U \times V \times U$. Let $\mathcal{U}_0 = (\mathcal{V}, U, \{(x, root, x) \;:\; x \in U\}, \lambda x\, v.x, \lambda x\, v.0)$ and $\mathcal{U}_{i+1} = \mathcal{U}_i.union(\mathcal{U}_i.find(a_i, v_i), \mathcal{U}_i.find(b_i, v_i), v_i)$ with $(a_i, v_i, b_i) = e_i$. Then, the closure of $\mathcal{E}$ is the versioned union-find $\mathcal{U}_\mathcal{E} = \mathcal{U}_n$.

Finally, we show that the versioned union-find is correct, i.e., $\equiv_v$ in $\mathcal{U}_\mathcal{E}$ is the smallest relation.

Theorem 6.10 (Versioned Union-find is correct). *Given a sequence of versioned equalities $\mathcal{E} = e_0 :: \ldots :: e_{n-1}$, then $\equiv_v$ in $\mathcal{U}_\mathcal{E}$ is the smallest reflexive, symmetric, transitive closure of $\mathcal{E}$.*

Proof. $\mathcal{U}_\mathcal{E}$ closes over $\mathcal{E}$ by Theorem 6.8. $\equiv_v$ is a reflexive, symmetric, and transitive closure of $\mathcal{E}$ by Theorem 6.5. To prove that it is the smallest, we will assume we are given an arbitrary relation $\sim_v$ with the same properties and we will show that whenever $x \equiv_v y$ then $x \sim_v y$. We prove this by induction on $\mathcal{E}$, the sequence of equalities. When $\mathcal{E}$ is empty, $x \equiv_v y$ holds only when $x = y$ by definition of $\mathcal{U}_0$. By the assumed reflexivity property of $\sim_v$, it follows that $x \sim_v y$. When $\mathcal{E} = \mathcal{E}' :: (a, b, v')$, the induction hypothesis states that $\mathcal{U}'_\mathcal{E}$ is the smallest relation that closes over $\mathcal{E}'$. By case analysis, if $v \not\trianglelefteq v'$ then the equivalence relation after unioning is the same as before since union preserves unrelated versions. If it is not, then we consider the four cases whether $x \equiv_v a$ or not and whether $y \equiv_v a$ or not. In all four cases a straightforward argument can be made using the induction hypothesis, Theorem 6.8, and that union preserves equalities. □

6.2 Versioned E-Graph

We define the versioned e-graph as a versioned union-find and a hash-cons in Definition 6.11. We assume a finite set of function symbols Σ, omitting their arity as it does not contribute to the results.

Definition 6.11 (Versioned E-Graph $\mathcal{G}$). The set of versioned e-graphs $\mathcal{S}_\mathcal{G}$ contains all triples $\mathcal{G}$ of the form $(\mathcal{U}, N, H)$ with the following properties:

(1) C is a finite set of e-classes,
(2) $\mathcal{U} = (\mathcal{V}, C, E, \mathit{parent}, \mathit{depth})$ is a versioned union-find as in Definition 6.2 over e-classes,
(3) $N \subseteq \Sigma \times C^*$ is a finite set of e-nodes, i.e., function applications over e-classes,
(4) $H : N \to C$ is a bijection from e-nodes to their original e-class (i.e., *hash-cons*),
(5) *Congruence*: In every version v, if the canonical forms of two e-nodes n and m are equal, i.e. $\mathit{canon}(n, v) = \mathit{canon}(m, v)$ then they must be in the same equivalence class, i.e. $\mathit{find}(H(n), v) = \mathit{find}(H(m), v)$ where *find* and *canon* are defined in Definitions 6.12 and 6.13 respectively.

We can immediately complete the definition by introducing *find* in Definition 6.12, which retrieves the canonical e-class (i.e. root) from the underlying versioned union-find, and *canon* in Definition 6.13, which canonicalizes the arguments of an e-node at some version.

Definition 6.12 (find). Given $\mathcal{G} = (\mathcal{U}, H, N)$, then $\mathit{find} : C \times V \to C$ is $\mathcal{G}.\mathit{find} = \mathcal{U}.\mathit{find}$.

By definition, *find* inherits all the properties from the underlying versioned union-find.

Definition 6.13 (canon). Given a versioned e-graph $\mathcal{G} = (\mathcal{U}, H, N)$, an e-node $n = f(c_1, \ldots, c_a) \in N$, and a version $v \in V$, then $\mathit{canon} : N \times V \to N$ is: $\mathcal{G}.\mathit{canon}(n, v) = f(\mathit{find}(c_1, v), \ldots, \mathit{find}(c_a, v))$.

As before, the equivalence relation of a versioned e-graph at some version is defined using find.

Definition 6.14 ($\equiv_v$). Given a versioned e-graph $\mathcal{G}$ and a version v, the relation $\equiv_v \subseteq N \times N$ is:

$$n \equiv_v m \text{ in } \mathcal{G} \quad \text{if and only if} \quad \mathcal{G}.\mathit{find}(H(n), v) = \mathcal{G}.\mathit{find}(H(m), v)$$

We can already prove that $\equiv_v$ has most of the properties we wish to prove:

Theorem 6.15 ($\equiv_v$ is a congruent equivalence relation). *$\equiv_v$ is a reflexive, symmetric, transitive, and congruent relation.*

Proof. The first three are inherited from the definition of *find* and Theorem 6.5. Congruence follows from the congruence invariance (5) of Definition 6.11. □

Next, we present the *merge* operation, which merges two e-classes at some version, thereby extending one of the equivalence relations in the versioned e-graph.

Definition 6.16 (merge). Given a versioned e-graph $\mathcal{G}_0 = (\mathcal{U}, H, N)$, and two classes $a, b \in C$ to merge at a version $v_0 \in V$ with $children(v_0) = \{v_1, \ldots, v_n\}$, then $merge : C \times C \times V \to \mathcal{S}_{\mathcal{G}}$ is:

$$\mathcal{G}_0.merge(a, b, v_0) = (\mathcal{U}', H, N)$$

where

$$\mathcal{G}_{i+1} = \mathcal{G}_i.merge(a, b, v_{i+1}) \qquad \text{for every } 0 \le i < n$$

$$W_0 = \{([a], [b])\} \qquad W_{k+1} = \{([H(e_1)], [H(e_2)]) \ : \ (x, y) \in W_k,\ e_1 \in N,\ e_2 = e_1[x \mapsto y] \in N\}$$

$$\forall c.[c] = \mathcal{G}_n.find(c, v_0) \qquad W = \cup_k W_k \qquad \mathcal{U}' = W.fold\left(\mathcal{G}_n.\mathcal{U},\ \ \lambda \mathcal{U}_i.\ \mathcal{U}_i.union\left(-, -, v_0\right)\right)$$

To merge two e-nodes a and b at v_0, Definition 6.16 constructs a sequence of e-graphs $\mathcal{G}_{i+1}$ which merges a and b at the i-th child of v in $\mathcal{G}_i$. Afterward, to guarantee congruence (Definition 6.11 (5)), a worklist W, a set of pairs of e-classes, is computed as follows. Starting with W_0 containing the e-classes of a and b, we gather in W_{k+1} all the pairs of e-classes which would become equal after unioning the pairs of e-classes in W_k. Observe that for every k, $W_k \subseteq C \times C$ and therefore there is a $\kappa \in \mathbb{N}$ for which $\cup_k^{\kappa} W_k = W$, i.e., the computation of the worklist is done in finite time. The hash-cons and the e-nodes of the resulting versioned e-graph are unchanged while the versioned union-find is the one where all the unions of the worklist are applied on top of $\mathcal{G}_n$.

We prove that *merge* unions the equivalence classes of its arguments at all descendant versions.

Theorem 6.17 (*merge* is correct). *Let $\mathcal{G}$, a, b, v be as Definition 6.16. For every $x, y \in N$ and v' such that $v \preceq v'$, if $x \equiv_v H^{-1}(a)$ and $y \equiv_v H^{-1}(b)$ in $\mathcal{G}$, then $x \equiv_{v'} y$ in $\mathcal{G}.merge(a, b, v)$.*

Proof. It follows by the definition of *merge* in Definition 6.16 and Theorem 6.8. □

Now, we are ready to state the main correctness theorem of versioned e-graphs: they represent the smallest reflexive, symmetric, transitive, and *congruent* closure of a set of equalities. First, we define how to construct a versioned e-graph from the trivial one and a sequence of equalities.

Definition 6.18 (Closure). Let $\mathcal{E} = e_0 :: \ldots :: e_{n-1}$ be a sequence of equalities at a version, where $e_i \in N \times V \times N$. Let $\mathcal{G}_0$ be the empty versioned e-graph and $\mathcal{G}_{i+1} = \mathcal{G}_i.merge(a_i, b_i, v_i)$ with $(a_i, v_i, b_i) = e_i$. Then, the closure of $\mathcal{E}$ is the versioned e-graph $\mathcal{G}_{\mathcal{E}} = \mathcal{G}_n$.

Finally, we show that $\equiv_v$ is the smallest equivalence relation with congruence.

Theorem 6.19 ($\mathcal{G}_{\mathcal{E}}$ is correct). *Given a sequence of versioned equalities $\mathcal{E} = e_0 :: \ldots :: e_{n-1}$, then $\equiv_v$ in $\mathcal{G}_{\mathcal{E}}$ is the smallest reflexive, symmetric, transitive, and congruent closure of $\mathcal{E}$.*

Proof. By the fact the $\mathcal{G}_{\mathcal{E}}$ closes over $\mathcal{E}$ and Theorem 6.15, it follows that $\equiv_v$ is a reflexive, symmetric, transitive, and congruent closure. To prove that it is the smallest, we will assume we are given an arbitrary relation $\sim_v$ with the same properties and we will show that whenever $x \equiv_v y$ then $x \sim_v y$. We prove this fact by induction on $\mathcal{E}$, the sequence of equalities.

When it is empty, $x \equiv_v y$ holds only when $x = y$ by definition of $\mathcal{G}_{\emptyset}$. By the assumed reflexivity property of $\sim_v$, it follows that $x \sim_v y$. When it is non-empty, i.e., when $\mathcal{E} = \mathcal{E}' :: (a, v', b)$, we may assume by the induction hypothesis that $\mathcal{G}'_{\mathcal{E}}$ is the smallest relation that closes over $\mathcal{E}'$. By case analysis, if $v \not\preceq v'$ then the equivalence relation after unioning is the same as before by definition of *merge* and the observation that it does not union at unrelated versions. If it is not, then we consider the four cases whether $x \equiv_v a$ or not, and whether $y \equiv_v a$ or not. In all four cases a straightforward argument can be made using the induction hypothesis, congruence in Theorem 6.15, and the fact that merge unions argument roots and preserves equalities. □

7 Implementation

We developed the Veg library of versioned e-graphs in Rust. Following Sections 3, 5 and 6, we present the implementation incrementally, beginning with the versioned union-find.

Versioned Union-Find. A union-find can be represented as an array mapping elements to their parent, such that elements are array indices and each element has exactly one parent. First, we extend this representation to support versioning in Veg. In our implementation, a version is simply a number, and the version tree is implemented by an array. Second, we label the edges in the union-find by versions. A versioned union-find is encoded as an array mapping each version to a map from elements to their parent in that version. Notably, the map is sparse because not all elements have a parent in every version. Finally, we modify the `union` and `find` operations to target a specific version (cf. Section 3). For these operations, it is critical that versions in the version tree can efficiently be traversed along its parent relationship.

Versioned E-Graph. We implement traditional e-graphs similarly to the *egg* library [27] for a fair comparison in our evaluation. An e-graph is built on a hash-cons and a union-find: the hash-cons maps e-nodes to their own e-class, while the union-find maintains the equivalence relation among e-classes. All operations performed on the e-graph are eventually delegated to the union-find.

To support versioning, we replace the underlying union-find with a versioned union-find, while sharing a single global hash-cons across versions. Restoring congruence invariance is delegated to the versioned union-find by canonicalizing e-nodes at a specific version and unioning them with their canonicalized counterparts in that version.

8 Evaluation

This section evaluates versioned e-graphs in different scenarios. For comparison, we consider two naïve baselines, the state-of-the-art for versioning in e-graphs, and our approach:

Cloning A baseline based on a standard e-graph where different versions are represented by independent copies of the entire e-graph.

Persistent An optimization of the cloning approach based on persistent data structures, namely RBB Trees [22] for arrays and HAMT [1] for maps, from the `im` Rust library.

Easter Egg The state-of-the-art *Easter Egg* [21] implementation that extends the popular *egg* [27] library to support versions.

Versioned Our versioned e-graph approach, implemented as a standalone library.

First, we perform a parameter analysis to investigate how the different approaches scale in terms of memory and run time for different factors, e.g., number of e-nodes and number of versions, and to provide insights into the average complexity of versioned e-graphs. Second, we implement a small EUF (Equalities and Uninterpreted Functions) solver and evaluate it on the benchmarks for uninterpreted functions from the standard SMT-LIB benchmark suite [19]. Third, we evaluate on running the TheSy [20] theorem prover on its own benchmarks.

The last case study makes extensive use of expensive equality saturation runs, which dominate the performance. As such, the advantage of versioning e-graphs is less evident in this case study – both for Easter Egg and our approach. We show, however, that versioning does not incur a performance overhead for implementing common extensions to e-graphs such as e-matching, extraction or saturation, and it shows substantial performance benefits in the other case studies.

8.1 Parameter Analysis

We assess how the three variants of supporting versioning in e-graphs scale regarding memory usage and run time for varying different parameters, focusing on the main influencing factors:

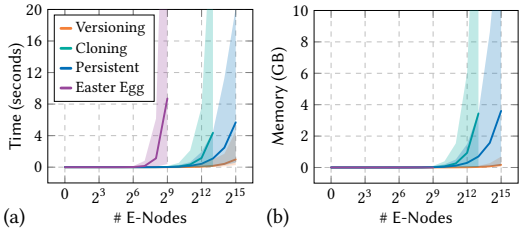


Fig. 8. Parameter Analysis for Amount of E-Nodes: Min-Avg-Max run time and memory per iteration.

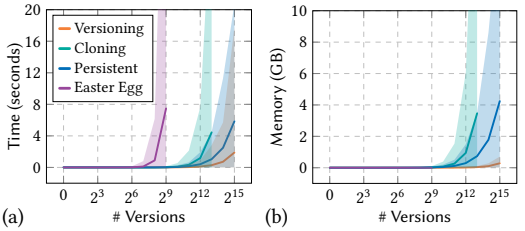


Fig. 9. Parameter Analysis for Amount of Versions: Min-Avg-Max run time and memory per iteration.

number of e-nodes and number of versions. To isolate the impact of each factor, we use synthetically generated equivalence relations, allowing us to control these parameters independently.

To synthesize such relations, we consider a set of operations containing e-node additions (as many e-nodes as configured), version additions (as many versions as configured), and unions and finds equal to the maximum between the number of e-nodes and versions. First, we perform all e-node additions by generating random e-nodes of a fixed maximum arity (5 in our experiment, with an average of 2.5) and adding them to the root version. Then, we iteratively sample and apply random operations from the aforementioned set until it is exhausted. Before any operation, we randomly select an existing version to operate on. Congruence is restored after every union.

Both parameters – number of e-nodes and versions – are varied exponentially from 2^0 to 2^{15}. We run the four approaches for every parameter combination on 100 randomly generated e-graphs on an AMD EPYC-Genoa Processor 2.4 GHz, 32 GiB memory, and 30 seconds timeout per e-graph, for a total of 12 hours of computing time. We compare execution time and memory using `gnu-time`.

For all four variants (Cloning, Persistent, Easter Egg, Versioning), Figure 8 shows the minimum, average, and maximum execution time and memory consumption (y-axis) for completing the benchmark with a fixed amount x of e-nodes (x-axis), varying versions up to x. Conversely, Figure 9 shows the minimum, average, and maximum execution time and the memory consumption (y-axis) for a fixed amount x of versions (x-axis), varying e-nodes up to x.

The memory consumption for the naïve baselines grows linearly (note the logarithmic scale) but substantially, using 1.5 to 3.5 GiB at 2^{13} e-nodes (Figure 8b) or versions (Figure 9b). In contrast, versioned e-graphs and Easter Egg consume 25 to 70 MiB – negligible by comparison.

Yet, Easter Egg's memory performance comes at a run time penalty which exceeds 6.7 to 8 s for only 2^9 e-nodes (Figure 8a) or versions (Figure 9a), while versioned e-graphs compute the same benchmarks in 7 to 18 ms. Note, that the cloning approach also shows a measurable impact on run time performance, which we attribute to excessive memory accesses. In fact, the persistent approach shows better performance than cloning, as its memory efficiency compensates for the higher time complexity of its operations.

8.2 Case Study: EUF Solving

As the first case study, we implemented an EUF solver that checks the satisfiability of boolean formulas with Equalities and Uninterpreted Functions. The EUF theory can be found in common SMT solvers such as Z3 [9] and cvc5 [2], is quantifier-free, i.e., formulas have no universal or existential quantification, a restriction that makes decision procedures tractable and efficient. The EUF domain allows us to evaluate versioned e-graphs on the typical input to such solvers.

Our EUF solver accepts as input a boolean formula in conjunctive normal form (CNF) and a list of equalities as pairs of terms over a finite alphabet. Each variable in the formula represents such an equality. This does not restrict the set of formulas the solver can handle since a boolean variable x can be converted to the equality $x = \texttt{true}$. In the solver, if a variable is `true` in some model, its corresponding equality holds, otherwise the disequality holds.

The solver first creates an e-graph with all the terms in the list of equalities. Second, it picks a variable, and forks two versions from every leaf in the version tree: one version where the variable's corresponding equality holds, and another version where the disequality holds. To encode such disequalities, our solver relies on *disequality edges* [16, 30], which we implemented using *e-class analysis* [27]. Third, it evaluates the input CNF formula using the model, i.e., the variable assignment induced by the forked version in the e-graph. If the CNF formula is `false`, the current version is discarded to avoid exploring all its corresponding submodels. Otherwise, the version is checked for consistency, i.e., the versioned e-graph checks if two e-nodes are simultaneously equal and unequal. The solver discards the current version if the consistency check fails.

We further implemented the solver with Cloning, Persistent and Easter Egg. For Cloning and Persistent, we clone the e-graph for every time a new version is forked. For Easter Egg, the approach is similar to our versioned e-graph, except Easter Egg does not support versioned e-class analyses, which we use to represent disequalities. Thus, we adopt the next best alternative, i.e., *disequality embedding* [30], which represents every disequality edge as a special e-node in the e-graph. Crucially, disequality embedding incurs negligible run time and memory overhead.

Benchmark Data. As input to our solver, we use benchmarks from the standard SMT-LIB benchmark suite [19] which focus on the core logic and the theory of uninterpreted functions (QF_UF and UF). QF_UF is quantifier-free and can thus be translated faithfully into CNF. UF contains universal and existential quantifiers, which we solve under the assumption that every sort is finite and only populated by the constants declared in each benchmark. In total, we run our solver on 6 145 benchmarks (3 920 UF and 2 225 QF_UF).

Experimental Setup. We run our solver on a machine with an AMD EPYC-Genoa Processor 2.4 GHz, 16 cores, 32 GiB memory, and 60 seconds maximum time for each benchmark, for a total of about 240 hours of computing time. We measure run time and memory required to check each formula using `gnu-time`. The latter peaks at the end of solving just before the program frees the list of all the leaves of the version tree.

Experimental Results. Figure 10a shows the cumulative execution times for all four variants (Cloning, Persistent, EasterEgg, Versioning), i.e., the number of benchmarks completed (y-axis) within a given time span (x-axis). Therefore, the closer the curve is to the top left corner the better the approach performs, as more benchmarks completed with fewer resources. Figures 10b to 10d each compare the execution time of every individual benchmark of versioned e-graphs (x-axis) to another approach (y-axis), one per figure. A dot on the bottom-left-to-top-right dashed diagonal signifies a benchmark that used an equal amount of resources in both compared variants. Dots above the diagonal are the benchmarks where versioned e-graphs outperform the other variant, with dashed guides for 1:2 and 2:1 ratios. We distinguish timeouts (TO), out-of-memory errors (OOM), and crashes (ERR) with black, blue, and red boundaries, respectively. We also add some noise to better visualize overlapping data points. Figure 11 shows the same for memory consumption.

The charts show that for a given time (Figure 10a) and a given amount of memory (Figure 11a), versioned e-graphs and Persistent always manage to run more benchmarks (about 4.15 K) than Cloning (about 3.9 K) and Easter Egg (about 1.5 K).

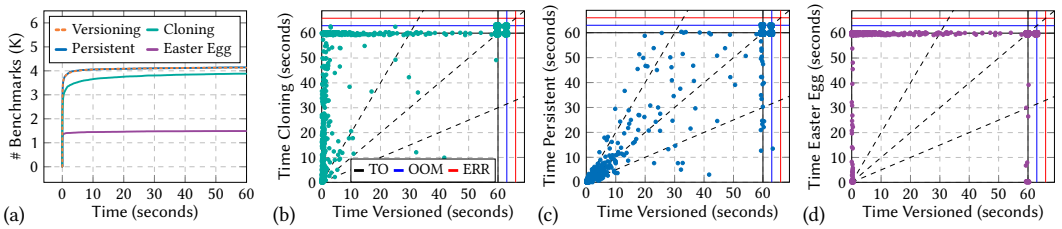


Fig. 10. Run time Comparison of the EUF Solver.

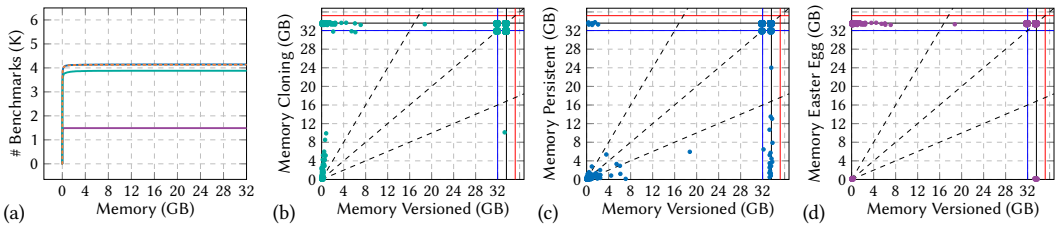


Fig. 11. Memory Comparison of the EUF Solver.

Figures 10b and 11b show that versioned e-graphs are substantially faster (4× on average) and more memory-efficient (0.75× on average) than Cloning for successful benchmarks, sharing about 2 K TOs and OOMs, with Cloning timing out much more frequently. Figures 10c and 11c show that versioned e-graph perform similarly to Persistent, with slightly more TOs and OOMs, while being slightly faster (1.2× on average) and more memory-efficient (0.8× on average) for successful benchmarks. Figures 10d and 11d confirm the findings from the parameter analysis (Section 8.1) that Easter Egg exhibits a substantial amount of timeouts, failing to complete most benchmarks.

8.3 Case Study: Automated Theorem Prover

For our second case study, we adapted the TheSy theorem prover [20], which was used to evaluate Easter Egg's approach to encode multiple equivalence relations in a single e-graph [21], i.e., the current state of the art. For our comparison, we swapped TheSy's e-graph implementation both with our versioned e-graph and with the cloning-based baseline in addition to Easter Egg.

Experimental Setup. We run the same 311 benchmarks of in the original TheSy paper [20] on a machine with an AMD EPYC-Genoa Processor 2.4 GHz, 16 cores, 32 GiB memory and 300 seconds maximum time for each benchmark, for a total of about 140 hours of computing time. We account for TheSy's non-determinism by re-running each benchmark until the maximum time is exhausted. If a single run takes longer than the limit, it is terminated by a timeout. We measure the time and memory of each run and report for each benchmark the average across all runs.

Experimental Results. Like for the EUF case study (Section 8.2), we show the cumulative and individual execution times (Figure 12) and memory consumption (Figure 13) for the TheSy case study, using the same layout and notation.

Figures 12a and 13a show versioned e-graphs solving the most benchmarks within a given time and memory limit (200 out of 311), followed closely by Cloning (191), then Persistent (161), and finally Easter Egg (67). While versioned e-graphs and Cloning complete a similar number of benchmarks, versioned e-graphs do so using noticeably less memory.

Regarding the individual benchmarks, Figures 12b and 13b show that versioned e-graphs have similar run time but are more memory-efficient than Cloning (0.95× on average) for successful benchmarks, sharing about 100 TOs and OOMs, with slightly more TOs and OOMs for Cloning.

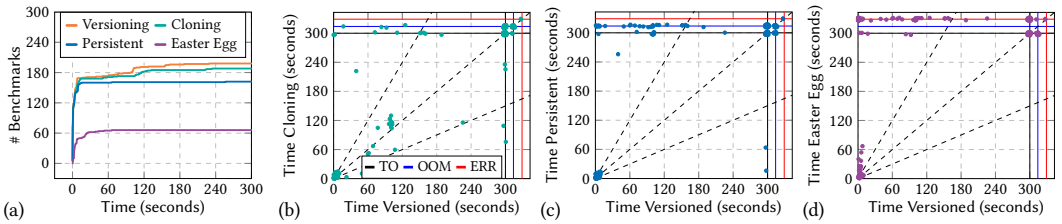


Fig. 12. Run time Comparison of TheSy.

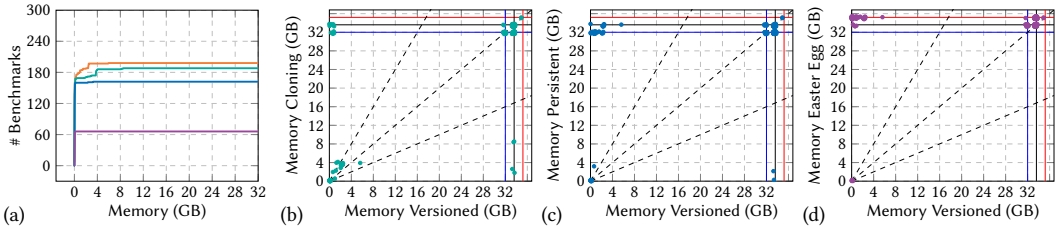


Fig. 13. Memory Comparison of TheSy.

Figures 12c and 13c show that versioned e-graph are slightly faster (1.1× on average) and more memory-efficient (0.7× on average) than Persistent on successful benchmarks. While the performance results are in accord with the EUF case study, the number of TOs and OOMs is significantly higher for Persistent. We believe this is due to the difference in workload in TheSy, which heavily relies on equality saturation, possibly reducing the effectiveness of sharing across the persistent data structures. Finally, Figures 12d and 13d maintain the same trends as before, with the additional observation that Easter Egg crashes on a significant number of benchmarks (178). Two of these crashes are shared with all other variants due to the original parser of TheSy. The other crashes result from broken invariants within Easter Egg's implementation.

Discussion. As a fully featured theorem prover, TheSy relies extensively on equality saturation. To make saturation efficient, all four e-graph variants maintain a cache mapping e-classes to the set of canonicalized e-nodes they contain – similar to an inverse union-find structure. Our implementation does not optimize this particular data structure, as the focus of this work is on the foundational union-find and e-graph data structures. Unfortunately, the benchmarks of TheSy heavily stress the inverse mapping structure, which limits the advantage of our approach in this specific case study. Nevertheless, our versioned e-graph achieves performance on par with both baselines and Easter Egg, and even outperforms Easter Egg on a substantial number of benchmarks. These results suggest that versioning does not introduce significant performance overhead for compute-intensive e-graph extensions such as saturation, while substantially improving performance in other case studies. We believe further research is warranted to encode this cache more efficiently.

9 Related Work

Easter Egg. Easter Egg [21] extends e-graphs with a hierarchy of "colored" union-finds, where colors correspond to our notion of versions. In contrast, our approach maintains a single versioned union-find data structure. A hierarchy of union-finds requires cascading lookups on the parent union-finds, hindering the crucial path compression optimization, impairing performance. Our approach of using a single union-find with versioned edges allows paths to be fully compressed. Furthermore, Easter Egg does not describe if and how cycles are handled as described in Section 3. Additionally, it does not support versioned e-class analyses, as each e-class can only be associated

with a single unversioned data. Finally, adding the same e-nodes to different versions in Easter Egg generates different e-classes, violating the notion of *full-sharing* as described in Section 1.

Easter Egg extends the popular egg framework [27], which offers a highly efficient implementation of e-graphs geared towards program optimization. It introduces e-class analyses, which enables straightforward semantic analyses over the e-graphs to tackle the issue that purely syntactic approaches struggle to define sound rewrite rules.

Assume Nodes. Assume nodes [8] are an extension to e-graphs to embed conditional expressions. This approach has been applied to encode constraints (e.g., on area, latency, power) for optimizing hardware designs, but it can generally be used for representing different versions depending on a condition into an e-graph. Assume nodes are special e-nodes with two parts. The first is a reference to the e-class of a boolean expression representing the condition or assumption under which equalities hold. The second is a reference to a sub e-graph that can never share e-nodes with the assume node's e-graph. To see why, consider the e-graph with four e-nodes, 0, x, $abs(x)$, and $x > 0$ each in their own e-class. When the $assume(x > 0, abs(x) = x)$ e-node is added, then the e-classes of x and $abs(x)$ must be unioned. If that is done in the same e-graph and then $x = -1$ is added, the e-graph asserts that $abs(-1) = -1$, violating the intention of the user. Having the assume node store a sub-e-graph that maintains copies of x and $abs(x)$ helps in resolving this issue.

To maximize sharing, assume nodes are pushed down as deep as possible in their bodies, which effectively reduces the size of the sub-e-graph each node has to maintain. This solution, however, requires saturating the e-graph on rewrite rules that perform this pushing, which is computationally intensive operation. Assume nodes are better suited when there is a need to reason about the equivalence of conditions. However, it is difficult to devise a sound and complete set of rewrite rules for applying them effectively to a specific domain, without exploding the size of the e-graph.

Applications. E-Graphs have been applied to support SMT solvers in handling theories involving equality [16]. Their capability to efficiently representing program spaces makes them particularly suitable for program optimization [12]. E-Graphs address two primary challenges: constructing the program space and retrieving elements from it (Section 2.2). A key insight from Tate et al. [24] introduced the concept of equational, monotone rewrites – known as equality saturation – as a method for constructing the program space.

Prominent applications of e-graphs in program optimization include tensor graphs [28], vectorization for signal processors [25], linear algebra optimization [26], and improved accuracy for floating-point expressions [18]. In program synthesis, e-graphs have been applied to tasks like CAD model synthesis [15] and "library learning", where common structures are extracted from a collection of programs to create reusable library functions [5].

10 Conclusion

E-graphs group together program terms, enabling one to efficiently merge term classes that proven equivalent and query for term equivalence, hence letting algorithms reason about rewrites for fast optimization and analysis. Traditional e-graphs can represent only one set of equalities at a time, while in many use cases equalities hold *conditionally* – for example under the hypothesis defined by each case in a proof with a case-by-case structure. To handle conditional equalities, multiple e-graphs are typically maintained, with, e.g., each e-graph corresponding to a different branch in a proof. However, this method is highly inefficient, as equalities common to multiple branches are redundantly duplicated across the e-graphs.

In this paper, we introduce *versioned e-graphs*, which encode multiple equivalence sets in the same data structure. We prove the correctness of versioned e-graphs and show that they are generally more efficient in both memory and time compared to existing approaches.

Artifact Availability

The artifact is publicly available on Zenodo [6]. It includes the Rust implementation of versioned e-graphs and a Lean4 mechanization of the definitions and proofs of versioned union-finds presented in Section 6.1. Moreover, all the benchmarks provided in Section 8 have dedicated scripts which can be executed to verify our reported results. The included README file provides a guide on how to interpret the output of the benchmark results.

Acknowledgments

This work has been co-funded by the Swiss National Science Foundation (SNSF, Grant No. 10001777), by ArmaSuisse Science and Technology, and by European Union's Horizon research and innovation program (CAPE Project, Grant No. 101189899).

References

[1] Phil Bagwell. 2001. Ideal Hash Trees. https://infoscience.epfl.ch/handle/20.500.14299/221731

[2] 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. doi:10.1007/978-3-030-99524-9_24

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[4] 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 2022)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3520308.3534506

[5] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. doi:10.1145/3571207

[6] Jahrim Gabriele Cesario, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2026. *Artifact for Versioned E-Graphs.* doi:10.5281/zenodo.18769553

[7] Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on Workshop on ML* (Freiburg, Germany) *(ML '07)*. Association for Computing Machinery, New York, NY, USA, 37–46. doi:10.1145/1292535.1292541

[8] Samuel Coward, Theo Drane, and George A. Constantinides. 2024. Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 14 pages. doi:10.1109/TCAD.2024.3483096

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer-Verlag, Berlin/Heidelberg, Germany, 337–340. doi:10.1007/978-3-540-78800-3_24

[10] Bernard A. Galler and Michael J. Fisher. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (May 1964), 301–303. doi:10.1145/364099.364331

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

[12] 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)*. Association for Computing Machinery, New York, NY, USA, 304–314. doi:10.1145/512529.512566

[13] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures: An Algorithmic Point of View* (2nd ed.). Springer Publishing Company, Incorporated.

[14] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs. *Proc. ACM Program. Lang.* 8, ICFP, Article 264 (Aug. 2024), 27 pages. doi:10.1145/3674653

[15] 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 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. doi:10.1145/3385412.3386012

[16] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford, CA, USA. AAI8011683.
[17] 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. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 258 (oct 2023), 29 pages. doi:10.1145/3622834
[18] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.* 50, 6 (jun 2015), 1–11. doi:10.1145/2813885.2737959
[19] Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. 2024. SMT-LIB release 2024 (non-incremental benchmarks). doi:10.5281/ZENODO.11061097
[20] Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 125–148. doi:10.1007/978-3-030-81688-9_6
[21] 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. doi:10.34727/2024/isbn.978-3-85448-065-5_13
[22] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB vector: a practical general purpose immutable sequence. *SIGPLAN Not.* 50, 9 (Aug. 2015), 342–354. doi:10.1145/2858949.2784739
[23] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. doi:10.1145/321879.321884
[24] 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)*. Association for Computing Machinery, New York, NY, USA, 264–276. doi:10.1145/1480881.1480915
[25] 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. doi:10.1145/3445814.3446707
[26] 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. *Proc. VLDB Endow.* 13, 12 (jul 2020), 1919–1932. doi:10.14778/3407790.3407799
[27] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. doi:10.1145/3434304
[28] 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 Fourth Conference on Machine Learning and Systems, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2021/hash/cc427d934a7f6c0663e5923f49eba531-Abstract.html
[29] George Zakhour, Jahrim Gabriele Cesario, Pascal Weisenburger, and Guido Salvaneschi. 2026. Proof Primitives for Equality Saturation-based Automated Provers *(VeriLang '26)*. ACM, New York, NY, USA, 7 pages.
[30] George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 77 (June 2025), 24 pages. doi:10.1145/3704913
[31] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proc. ACM Program. Lang.* 7, PLDI, Article 162 (jun 2023), 24 pages. doi:10.1145/3591276
[32] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *Proc. ACM Program. Lang.* 8, PLDI, Article 178 (jun 2024), 24 pages. doi:10.1145/3656408

Received 2025-11-14; accepted 2026-04-03