A Lattice of Union-Finds (with appendices)

Dorian Lesbre, Matthieu Lemerre

A Lattice of Union-Finds (with appendices)

Dorian Lesbre[1] [0000-0002-4328-6753] and Matthieu Lemerre[1] [0000-0002-1081-0467]

Université Paris-Saclay, CEA List, Palaiseau, France
first.last@cea.fr

Abstract. Union-find data-structures offer compact and fast representation of relations. To use them to represent facts in an abstract interpreter, the canonical framework is to turn the structure into a lattice: i.e. extend persistent union-find with a partial order and the associated join and meet. We formally define all three and show how each can be computed in almost-linear time relative to the number of nodes, or, in the presence of an efficient difference operator, in almost-linear time relative to the size of the difference. Benchmarks show these algorithms offer good performance even on large inputs, and can be integrated in an abstract interpreter with a small cost overhead.

Keywords: Union-find · Lattice · Persistence · Abstract Interpretation

1 Introduction

Union-find data-structures are a staple of most algorithmic courses. Indeed, the core algorithms of union-find are simple but efficiently solve a non-trivial problem, the *disjoint set* problem, which can be though of as keeping track of a changing partition of a given set. An alternative formulation of this problem sees union-find as maintaining an equivalence relation by computing a symmetric-transitive closure from a growing set of equivalences. Typically, the problem has queries (Which set is this node in? Is it equivalent to this other node?), called *find* operations, interleaved with *union* operations which merge two sets

This is the technical report accompanying the SAS 2026 publication. It has not undergone any post-submission improvements or corrections. The version of record of the contribution is published in *Lecture Notes on Computer Science* (LNCS).

together. Quite famously, union-find solves this with what is essentially the best complexity one could hope for. Tarjan and van Leeuwen [1984] show that, using union-by-rank and path compression, both operations have an amortized cost proportional to the inverse of Ackermann's function, which is almost constant.

Besides the rather abstract disjoint set problem, union-find has been used in a wide variety of practical applications: graph connectivity queries, type systems [Milner, 1978], e-graphs [Willsey et al., 2021], alias analysis [Steensgaard, 1996] and representing injective relations in program analysis [Lesbre et al., 2025].

We are particularly interested in using union-find to represent relations efficiently in program analysis. There, equalities have been used to perform congruence closure [Chang and Leino, 2005], to factor out equal variables, reducing the size of other analyses [Cox et al., 2015], to reason about sequences [Giet et al., 2023], and to analyze the control-flow of mobile systems [Feret, 2005].

However, these examples rarely use union-find to represent equalities. When they do, they share a common simplifying constraint: they all use a single, global instance of union-find, which is therefore limited to storing flow-insensitive facts. In a non program analysis lingo, this means union-find can only store facts that are always true, and not depend on a particular context. How to efficiently make union-find flow-sensitive, or contextual, has remained an open research problem.

One difficulty is that union-find operations are effective because they rely on imperative modification. Operations like path compression have invisible side effects that are important for efficiency, and union outright destroys the previous version. As such, the structure is *ephemeral* [Driscoll et al., 1986]. Research shows that union-find can be made *fully persistent* [Italiano and Sarnak, 1991] without sacrificing performance [Conchon and Filliâtre, 2007], which is a first step in making the data-structure contextual as it allows backtracking, and thus using union-find with a tree of contexts.

However, this does not suffice for flow-sensitive program analysis, where we also need to combine and compare union-finds coming from different contexts. For instance, computing what relations are true after merging two branches of an if-statement requires finding the set of relations that are true in both contexts. In short, **union-find contexts should have a lattice structure, and not just a tree structure. This requires defining lattice operations (ordering, join, and meet) on union-find instances**, i.e. making union-find not just fully persistent, but *confluently-persistent* [Fiat and Kaplan, 2001].

Defining these operations poses additional challenges. For one, the internal structure of a union-find is non-canonical, with many possible layouts representing the same equivalence relations. Operations on union-find must not depend on this interior structure, which is particularly challenging for the binary lattice operations, as they must consider and compare two instances at once.

Furthermore, union-finds are efficient because they represent the quadratic number of relations between any pairs of nodes by a linear spanning tree. This is a form of constraint factorization [Lesbre et al., 2025], where the structure avoids storing constraints that can be discovered by reduction [Miné, 2002]. This poses a challenge for join, as the best theoretical precision requires considering all

constraints, even those that are not explicitly stored in the structure. However, we want to avoid performing a full reduction first (i.e. computing the full transitive closure), as this expensive computation has quadratic complexity, and would thus overshadow the time and memory benefits of union-find.

Contributions. We formally define all three lattice operations: join, meet and ordering for persistent union-find data-structures in terms of the represented equivalence relation (Section 2). These definitions are thus independent of the internal structure of the union-finds. We then present algorithms for computing all three operations on any persistent union-find (Sections 3, 4 and 5 respectively). Our algorithms are simple to implement and efficient, but rely on some non-trivial correction arguments. They all run in $\mathcal{O}(|\mathbb{N}| \times \alpha(|\mathbb{N}|) \times \rho)$, which is *almost*[1]-linear time relative to the number of union-find nodes $|\mathbb{N}|$, times the persistent map access cost ρ (which is 1 for arrays, and logarithmic for functional maps). For join, we provide worst-case examples showing that this complexity is optimal.

Furthermore, we show that, when the underlying data-structure supports an efficient difference operator (Section 3.3), these algorithms can be modified to run in almost-linear time relative to the size of the difference, which can be substantially smaller. We specialize our algorithms to two concrete data-structures, Patricia trees [Okasaki and Gill, 1998] and persistent arrays [Conchon and Filliâtre, 2007], which implement such a difference operator (Section 6). We also present a version of these algorithms for labeled union-find (Section 7). Finally, we benchmark them on randomized graphs, and evaluate their use to represent equalities between SSA terms in an abstract interpreter (Section 8). Using SSA notably avoids having to remove relations as variables change. While deletions are possible with a modified union-find [Kaplan et al., 2002], we do not tackle this problem here.

These algorithms were implemented in OCaml and published as an `opam` library `union-find-lattice` [Lesbre and Lemerre, 2026b]. They are also available in the accompanying software artifact [Lesbre and Lemerre, 2026a].

2 Defining the Lattice Operators on Union-Find

2.1 Union-find

A *union-find* data-structure $\mathbb{UF}$ [Tarjan, 1975] is defined on a given set of *nodes* $\mathbb{N}$ (often integers $[\![0 : k-1]\!]$). The public interface contains three operations:

$$\texttt{make} : \texttt{int} \to \mathbb{UF} \qquad \texttt{find} : \mathbb{UF} \to \mathbb{N} \to \mathbb{N} \qquad \texttt{union} : \mathbb{UF} \to \mathbb{N} \to \mathbb{N} \to \texttt{unit}$$

where `make` k creates a new instance supporting up to k nodes, `find` $u\ x$ return a unique *representative* of the set that contains x (and may modify the structure to perform *path compression*, i.e. shortening the path from x to its representative, without changing the underlying relation), and `union` $u\ x\ y$ modifies u by joining

[1] Our algorithms pay an inverse Ackermann cost on top of the linear complexity.

the sets containing x and y. For our proofs, we introduce a fourth operator, $\texttt{repr} \in \mathbb{UF} \to \mathbb{N} \to \mathbb{N}$, which is similar to `find` but has no side effects.

Given a union-find structure $u \in \mathbb{UF}$, we can define the represented *equivalence relation* as $x \equiv_u y \triangleq \texttt{repr}\ u\ x = \texttt{repr}\ u\ y$. For a given node $x \in \mathbb{N}$, we define its *equivalence class* as the set of related nodes: $\{y \in \mathbb{N} \mid x \equiv_u y\}$.

2.2 Persistent Union-find

Persistent union-find [Conchon and Filliâtre, 2007] has the same interface, except that `union` returns a newly created instance and does not modify its argument:

$$\texttt{union} : \mathbb{UF} \to \mathbb{N} \to \mathbb{N} \to \mathbb{UF}$$

Concretely, this can be built using any *persistent map* data-structure, denoted $X \rightharpoonup Y$, which represents a partial function between elements of X and elements of Y. This map supports get and set operations, denoted $m[x]$ and $m[x \mapsto y]$ respectively, with set returning a new copy, without destroying the previous version. Section 6 will look into two concrete implementations of such maps. For the rest of the paper, we will use an unspecified persistent map, and denote by ρ an upper bound of the cost of its get and set operations (typically 1 for arrays, and logarithmic in the number of unions for functional maps). The data-structure is then simply a pointer (mutability is used for path compression) to such a map [Conchon and Filliâtre, 2007]:

$$\textbf{type}\ \ \mathbb{UF} \triangleq \{\ \textbf{mutable}\ \texttt{parent}\colon\ \mathbb{N} \rightharpoonup \mathbb{N}\ \}$$

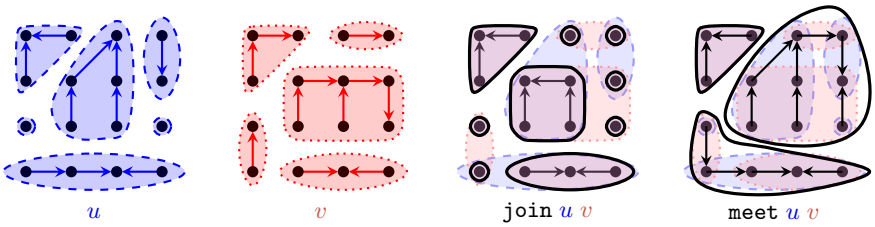


Fig. 1: Example of join and meet, each point represents a node, each potato a class and each arrow a parent. From left to right: two example union-finds u (blue dashed) and v (red dotted), their join, and their meet (both in black).

2.3 Lattice Operators

To turn union-find into a lattice, i.e. to make it confluently-persistent, we define three binary operators: a *partial order* $\cdot \sqsubseteq \cdot : \mathbb{UF} \to \mathbb{UF} \to \texttt{bool}$; a least-upper bound $\texttt{join} : \mathbb{UF} \to \mathbb{UF} \to \mathbb{UF}$; and a greatest lower-bound $\texttt{meet} : \mathbb{UF} \to \mathbb{UF} \to \mathbb{UF}$.

All of these can be defined in terms of the equivalence relations represented by their arguments. Inclusion is simply an implication between both relations:

$$u \sqsubseteq v \quad \triangleq \quad \forall\, x\ y \in \mathbb{N}^2,\ x \equiv_v y \Rightarrow x \equiv_u y \tag{DefIncl}$$

Given u and $v \in \mathbb{UF}$, join returns a union-find structure representing the conjunction of their equivalence relations, i.e. the equivalence class intersection:

$$\forall\, x\ y \in \mathbb{N}^2, \quad x \equiv_{(\texttt{join}\ u\ v)} y \quad \triangleq \quad x \equiv_u y \ \wedge\ x \equiv_v y \tag{DefJoin}$$

Finally, meet returns (the transitive closure of) the disjunction of the corresponding relations (as the disjunction may not be transitively closed), i.e. the union of the equivalence classes. It is defined inductively as:

$$\begin{aligned}\forall\, x\ y \in \mathbb{N}^2, \quad x \equiv_{(\texttt{meet}\ u\ v)} y \triangleq\ & x \equiv_u y \ \vee\ x \equiv_v y \\ & \vee\ \exists\, z,\ x \equiv_{(\texttt{meet}\ u\ v)} z \wedge z \equiv_{(\texttt{meet}\ u\ v)} y\end{aligned} \tag{DefMeet}$$

Both `join` u v and `meet` u v should maintain persistence. They must return a new version and leave u and v accessible.

The reader may be confused as to why a conjunction is called join, and a disjunction meet. This convention is consistent with the use of lattices in the theory of abstract interpretation, where smaller elements in lattice order are more precise. With this convention, the empty union-find (with no constraints, i.e. equality) is the maximal element of the lattice. The join corresponds to fewer known constraints between elements, so it loses precision, while the meet corresponds to more constraints, and therefore increases in precision.

Example 2.1. Figure 1 shows the result of computing the join and meet on two union-finds containing 16 nodes. We show both a possible internal structure (with parent arrows), and the represented classes (as potato sets). The join intersects the equivalence classes whereas the meet unions them.

Note that while these operations are straightforward when considering the equivalence classes, they are more subtle when one only considers the parent arrows. For instance, the top-left class is identical in u and v but has a different parent layout and representative in both.

Theorem 2.2. *Join is the least upper bound and meet the greatest lower bound of our lattice order.*

Proof. $x \equiv_{(\texttt{join}\ u\ v)} y \Rightarrow x \equiv_u y \wedge x \equiv_v y$ by DefJoin, so join is a UB of u and v. For any UB $u, v \sqsubseteq r$, $x \equiv_r y$ implies both $x \equiv_u y$ and $x \equiv_v y$, so `join` u $v \sqsubseteq r$.

By DefMeet, $x \equiv_u y$ or $x \equiv_v y$ implies $x \equiv_{(\texttt{meet}\ u\ v)} y$, so meet is a LB. For any LB $r \sqsubseteq u, v$, we can break down $x \equiv_{(\texttt{meet}\ u\ v)} y$ into a chain of equivalences in u or v. Each link is preserved in r, therefore so will the whole chain. □

With these definitions, the lattice has finite height and size when the set of nodes is finite. Indeed, for any strict inclusion $u \sqsubset v$, v must have at least one more equivalence class then u, and the total number of classes is bounded by $|\mathbb{N}|$.

Algorithm 3.1: Almost-quadratic join algorithm.

```
let join : UF → UF → UF ≜ fun u v ↦
    let res : UF ≜ make |ℕ| in (* resulting union-find *)
    for x in ℕ do
        for y in ℕ do
            if find u x = find u y && find v x = find v y
            then res := union res x y
        done;
    done; return res
```

Algorithm 3.2: Almost-linear join algorithm.

```
let join : UF → UF → UF ≜ fun u v ↦
    let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
    let res : UF ≜ make |ℕ| in (* resulting union-find *)
    for x in ℕ do
        let r_u ≜ find u x and r_v ≜ find v x in
        match Hashtbl.lookup new_classes (r_u, r_v) with
        | Some r_j ↦ res := union res x r_j (*or set res.parent[x ↦ r_j]*)
        | None ↦ new_classes := new_classes[(r_u, r_v) ↦ x]
    done; return res
```

3 The Join Operator

This section describes increasingly performant join algorithms: a naive one inspired by DefJoin; another which is almost-linear in the number of nodes; and a third which is almost-linear in the size of the difference between versions. Finally, it shows how join can use and maintain union-by-rank, an important optimization to minimize union-find tree depth.

3.1 Almost-Quadratic Join

DefJoin can be adapted into a simple algorithm that iterates through every pair of nodes and unions those that are related in both u and v. This is presented in Algorithm 3.1 using an OCaml-like syntax, taking some liberties for legibility. It has an almost-quadratic complexity $\mathcal{O}\big(|\mathbb{N}|^2 \times \alpha(|\mathbb{N}|) \times \rho\big)$ where α is the inverse Ackermann function. This is similar to the transitive closure algorithm used to perform joins in weakly-relational domains [Miné, 2002], which has cubic complexity; but benefits from the fact that union-find computes transitive closures in almost-constant time.

3.2 Almost-Linear Join

We can improve on Algorithm 3.1 by exploiting more of the internal structure of the union-find (Algorithm 3.2). The crux of this algorithm is the construction of a map `new_classes` from pairs of representatives of u and v to a corresponding representative of `join` u v. It is built by a linear scan over all nodes. Another pass unions each node to its representative in `join` u v, again in linear time; for efficiency, we group the two passes in a single one. A conceptually similar algorithm is used by Chang and Leino [2005] to compute a join of e-graphs.

Intuitively, this algorithm works because each equivalence class of `join` u v corresponds to a unique pair of classes from u and v. Recall that the join can be thought of as computing all class intersections (Example 2.1). Furthermore, we can identify a class by its representative. So, for a given node $x \in \mathbb{N}$, the pair $(\texttt{repr}\ u\ x, \texttt{repr}\ v\ x)$ uniquely identifies the intersection of classes it belongs to.

This is formalized in the following theorem. Let `Reprs` u be the set of representative nodes in the union-find u. Then, `Reprs` (`join` u v) can be mapped one-to-one to a subset of `Reprs` $u \times$ `Reprs` v, specifically, to the subset which corresponds to non-empty class intersections. This subset, which we call *set of cross-representatives* of u and v is simply $\mathcal{R}_{u,v} \triangleq \{(\texttt{repr}\ u\ x, \texttt{repr}\ v\ x)\ :\ x \in \mathbb{N}\}$. Algorithmically, it can be constructed in linear time and space.

Theorem 3.1. *A join operator verifies* DefJoin *if and only if, for all* $u, v \in \mathbb{UF}^2$, *there exists a function* $f \in \mathcal{R}_{u,v} \to \mathbb{N}$ *such that:*

1. *for all* $x \in \mathbb{N}$, $\texttt{repr}\ (\texttt{join}\ u\ v)\ x = f(\texttt{repr}\ u\ x,\ \texttt{repr}\ v\ x)$
2. f *is injective*

In that case, the co-restriction $f \in \mathcal{R}_{u,v} \to \texttt{Reprs}\ (\texttt{join}\ u\ v)$ *is bijective.*

Proof. ($\Rightarrow$) Let x and y be nodes such that $\texttt{repr}\ u\ x = \texttt{repr}\ u\ y$ and $\texttt{repr}\ v\ x = \texttt{repr}\ v\ y$. By DefJoin, $x \equiv_{\texttt{join}\ u\ v} y$: they have the same repr in the join. Hence, this repr is a function of their reprs in u and v. This function also satisfies (2), as otherwise we would have nodes which are not related in u or v, but are related in the join.

($\Leftarrow$) The following statements are equivalent for any two nodes x and y:

$$
\begin{aligned}
& x \equiv_{\texttt{join}\ u\ v} y \\
\iff\ & \texttt{repr}\ (\texttt{join}\ u\ v)\ x = \texttt{repr}\ (\texttt{join}\ u\ v)\ y \\
\iff\ & f(\texttt{repr}\ u\ x,\ \texttt{repr}\ v\ x) = f(\texttt{repr}\ u\ y,\ \texttt{repr}\ v\ y) && \text{by (1)} \\
\iff\ & (\texttt{repr}\ u\ x, \texttt{repr}\ v\ x) = (\texttt{repr}\ u\ y, \texttt{repr}\ v\ y) && \text{by (2)} \\
\iff\ & \texttt{repr}\ u\ x = \texttt{repr}\ u\ y \wedge \texttt{repr}\ v\ x = \texttt{repr}\ v\ y \\
\iff\ & x \equiv_u y \wedge x \equiv_v y
\end{aligned}
$$

When both are true, (1) implies that f can be co-restricted to `Reprs` (`join` u v), and that it is surjective on that set. Combining with (2) it must be bijective. □

The theorem, like DefJoin says nothing of the internal structure of the result union-find. Indeed, there are many possible choices for a correct join: firstly we must choose one node to be the representative of the new classes (with different choices corresponding to different functions f in the theorem); and then we must choose a tree structure to map all other nodes to the chosen representative.

Lemma 3.2. *Algorithm 3.2 satisfies* DefJoin *and runs in* $\mathcal{O}\big(|\mathbb{N}| \times \alpha(|\mathbb{N}|) \times \rho\big)$.

Proof sketch. Apply Theorem 3.1 with $f \triangleq (\texttt{repr}\ \texttt{res}) \circ \texttt{new\_classes}$. Details in Appendix C. □

3.3 Join Based on a Difference Operator

Algorithm 3.2 is optimal in the sense that, in the worst cases highlighted in Examples 3.3 and 3.4, the representation of `join` u v as a union-find data structure has $\Omega(|\mathbb{N}|)$ differences with both u and v. Thus, it must require a linear time to produce by modifying either of the arguments. In Example 3.4, the join also requires linear time to produce when starting from a fresh, empty union-find.

Algorithm 3.2 already makes union-find join almost as fast as any non-relational domain [Miné, 2002], which typically perform join in $\mathcal{O}(|\mathbb{N}| \times \rho)$. A common optimization for these domains is to speed up the join by only iterating through the difference between the arguments. This improves the complexity when the arguments are similar (which is very important in static analysis applications [Blanchet et al., 2003, §6.1.2], where most joins merge branches corresponding to small conditionals). We can do the same thing in union-find: if most nodes point to the same parent in both arguments, then the total number of updates starting from either is proportional to the size of the difference between them, which can be much smaller than the number of nodes.

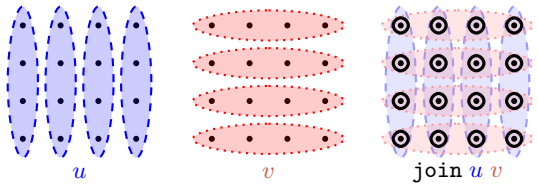
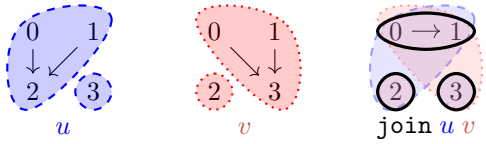


Fig. 2: Worst-case join, each class of u intersects with each class of v.

Example 3.3 (All classes intersect). Figure 2 presents a union-find on nodes which are integer pairs ($\mathbb{N} = [\![0:k]\!] \times [\![0:k]\!]$ with $k = 3$ in the figure). When joining a union-find representing first coordinate equality (u) with one representing second coordinate equality (v), the result is the equality union-find, where all classes are singletons. Obtaining such a structure by modifying either argument requires at least $k^2 - k$ writes, since the parents of all non-representatives must be updated.

Note that starting from a fresh union-find can require no writes at all, as long as `make` k^2 does not perform any writes, as is the case with some functional map based union-finds.

Example 3.4 (Worst-case join). Consider a set of 4 nodes $\mathbb{N} = \{0, 1, 2, 3\}$ and the union finds with the following parents arrays: $u.\texttt{parent} \triangleq [2, 2, 2, 3]$, $v.\texttt{parent} \triangleq [3, 3, 2, 3]$, as illustrated below:



Their join must keep 0 and 1 in the same class, which amount to modifying 3 parents when starting from u or v, or modifying one parent when starting from a fresh union-find. Repeating this pattern allows building an example with $4k$ nodes, whose join requires modifying $3k$ parents when starting from u or v, or k parents when starting from the empty union-find.

Suppose we have a *difference operator* $\texttt{diff} : \mathbb{UF} \to \mathbb{UF} \to \mathcal{P}(\mathbb{N})$ which, given two union-finds, iterates on all the points where they differ. Formally:

$$\forall\, x \in \mathbb{N}, \quad u.\texttt{parent}[x] \neq v.\texttt{parent}[x] \;\Rightarrow\; x \in \texttt{diff}\ u\ v \qquad \text{(DefDiff)}$$

Crucially, this operator should be faster than $\mathcal{O}(|\mathbb{N}|)$: when the number of unequal parents is small (relative to $|\mathbb{N}|$), the complexity and size of `diff` should also be small. In the following, we use δ as an upper bound of both the complexity and size of `diff`. Section 6 will define such an operator for two persistent map data-structures. Note that DefDiff only specifies a lower bound for the difference. Having too many nodes (i.e. nodes with the same parent) in the difference is still correct, it just makes computation slower.

With this new operation, we can reformulate Algorithm 3.2. Instead of iterating over every node, we only iterate over the difference. Indeed, there is no need to inspect or modify nodes which have the same parent in both arguments, we can keep them as-is, pointing to their common parent. For the nodes which appear in the difference, we first reset their parents to themselves, essentially putting each in its own class, and then union them as in the linear algorithm.

A tricky part of skipping shared nodes is that we are no longer free to choose an arbitrary node to represent the new classes. When considering an intersection of classes that contains both nodes that appear in the difference (which will be unioned to our chosen node) and nodes that do not (whose parents are unchanged), we need to ensure that the chosen representative matches the inherited one. The following example shows why an arbitrary choice could split a class in two:

Example 3.5. Consider the join of the two following union-finds:

Of the three nodes, z may be the only one that appears in the difference. It is thus the first seen node corresponding to the cross-representative pair (x, x). However, if we set $\texttt{new\_classes}[x, x] := z$ and point z to itself, as is done in Algorithm 3.2, the join would split $\{x, y, z\}$ in two classes $\{x, y\}$ and $\{z\}$, since the parents of x and y are unchanged, as they do not appear in the difference.

This problematic interaction can only occur for nodes t that have the same representative in both u and v (i.e. `repr` $u\ t =$ `repr` $v\ t$). Such nodes may have their paths to the representative completely unchanged (x and y in the example). This cannot happen if the representatives differ, as every node is then guaranteed to have at least one ancestor in the difference, and thus have its path be edited.

Therefore, all we need to maintain is the invariant that nodes with the same representative in both u and v should union to that representative in the join:

$$\forall\, x \in \mathbb{N},\ \texttt{repr}\ u\ x = \texttt{repr}\ v\ x \Rightarrow \texttt{new\_classes}[\texttt{repr}\ u\ x, \texttt{repr}\ v\ x] = \texttt{repr}\ u\ x \qquad \text{(KeepSameRepr)}$$

To do so, we initialize `new_classes` so that (x, x) is bound to x for all $x \in \mathbb{N}$. In practice, these bindings are not physically stored in `new_classes`, we simply change the `lookup` function to check for pair equality before performing a hash-table lookup.

The new Algorithm 3.3 is remarkably similar to Algorithm 3.2 with only three key differences:

1. the result `res` is not newly created by `make` (with all elements pointing to themselves), instead it is initialized by u, and we only reset the parents of nodes in `diff` $u\ v$ to point to themselves;
2. it iterates through the difference (`diff` $u\ v$) instead of every node;
3. it checks for pair equalities before performing a `Hashtbl.lookup`.

Despite the similarity between both algorithms, the correctness proof for Algorithm 3.3 is significantly more complex. Indeed, it must now consider the interactions between parents which are explicitly set and those that are inherited from u, using KeepSameRepr to ensure they agree on the new representative.

One final subtle point, the `find` calls on line 8 can modify `diff` $u\ v$ if find performs path compression. To ensure correction, it is important to initialize `res` before these calls, to operate on an unmodified copy of u.

Lemma 3.6. *Algorithm 3.3 satisfies DefJoin and runs in* $\mathcal{O}(\delta \times \alpha(|\mathbb{N}|) \times \rho)$.

Proof sketch. Use Theorem 3.1 with $f \triangleq$ (`repr res`) $\circ$ `lookup new_classes`. Details in Appendix C. □

3.4 Using Rank for More Efficient Unions

Union-find offers two choices of representative when computing `union` $u\ x\ y$: either make the representative of x point to that of y, or do the reverse. The chosen strategy impacts performance, as it leads to parent trees of different depths. Tarjan [1975] showed that an arbitrary choice leads to a logarithmic complexity for `find` and `union` with path compression, and linear complexity without. Two known improvements are *union-by-size* which tracks class sizes, and merges the smallest into the largest; and *union-by-rank* [Tarjan and van Leeuwen, 1984], which tracks (an upper bound of) class tree depth, called *rank*, and merges

Algorithm 3.3: Difference-based join, highlighting changes from Algorithm 3.2.

```
1  let lookup tbl (x,y) = if x = y then Some x
2                         else Hashtbl.lookup tbl (x,y)
3  let join : UF → UF → UF ≜ fun u v ↦
4      let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
5      let res : UF ≜ { parent = u.parent } in
6      for x in diff u v do res.parent := res.parent[x ↦ x] done;
7      for x in diff u v do
8          let r_u ≜ find u x and r_v ≜ find v x in
9          match lookup new_classes (r_u, r_v) with
10         | Some r_j ↦ res := union res x r_j
11         | None ↦ new_classes := new_classes[(r_u, r_v) ↦ x]
12     done; return res
```

the shallower tree into the deeper one. Both improve complexity to the inverse of the Ackermann function with path compression, and logarithmic without.

We will use union-by-rank as it requires fewer writes (all unions modify size, but not all unions modify rank), which decrease memory use in persistent union-find. The adaptation for union-by-size is similar. To support union-by-rank, we need to store an upper-bound of rank at all nodes, not just the representatives. The rank of a non-representative is an upper bound of the depth of the subtree rooted there. This way, when we reset the parents (Algorithm 3.3, line 6), we still have some rank information on these newly created representatives. With this, we can essentially re-use Algorithm 3.3 directly. As a small optimization, Algorithm 3.4 sets the ranks of elements of the difference to the minimum of their ranks in both arguments.

Lemma 3.7. *Algorithm 3.4 satisfies* DefJoin *and computes valid rank upper bounds. It runs in* $\mathcal{O}(\delta \times \alpha(|\mathbb{N}|) \times \rho)$.

Proof. Correction and complexity as in Lemma 3.6. For rank, take $x \in \texttt{diff}\ u\ v$. Any nodes pointing to x after the first loop are not in the diff. So the longest path to x is in both u and v, and thus bounded by $\min u.\texttt{rank}[x]\ v.\texttt{rank}[x]$ □

Another strategy is *random-union*, which simply chooses a class at random. It is as efficient as union-by-size in practice [Goel et al., 2014], and can be used directly with our join algorithms as it does not require any additional information. However, as it is non-deterministic, it can lead to a larger difference when joining branches which have common unions.

Appendix A shows a variant of join that edits the parents once instead of resetting them and then calling union. It has the same theoretical complexity but avoids the double write for parents in the difference. This can be a bit faster, however, supporting union-by-rank becomes more involved.

Algorithm 3.4: Difference-based join by rank, showing changes from Algorithm 3.3.

```
type 𝕌𝔽rank ≜ {mutable parent: ℕ ⇀ ℕ; mutable rank: ℕ ⇀ int}
let join : 𝕌𝔽rank → 𝕌𝔽rank → 𝕌𝔽rank ≜ fun u v ↦
    let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
    let res : 𝕌𝔽rank ≜ { parent=u.parent; rank=u.rank } in
    for x in diff u v do
        res.parent ≔ res.parent[x ↦ x];
        res.rank ≔ res.rank[x ↦ min u.rank[x] v.rank[x]];
    done;
    for x in diff u v do
        let ru ≜ find u x and rv ≜ find v x in
        match lookup new_classes (ru, rv) with
        | Some rj ↦ res ≔ union res x rj
        | None ↦ new_classes ≔ new_classes[(ru, rv) ↦ x]
    done; return res
```

Algorithm 4.1: Difference based meet algorithm.

```
let meet : 𝕌𝔽 → 𝕌𝔽 → 𝕌𝔽 ≜ fun u v ↦
    diff u v ▷ fold (fun res x ↦ union res x v.parent[x]) u
```

4 The Meet Operator

Meet is simpler than the join since it only amounts to adding more constraints to a union-find, that is to say, perform additional unions. In contrast, the join has to remove constraints from the union-find, which requires more careful consideration of the internal structure.

In essence, all `meet` u v has to do is perform in u all the unions from v. While finding that full set of union could be complex, it suffices to only add to u the unions between x and v.`parent`$[x]$ for all nodes x. These unions imply all others by transitivity and symmetry. Reusing the difference operator from Section 3.3, we can add these unions only for nodes whose parent in v is different from that in u, thus avoiding needless work (Algorithm 4.1).

A benefit of only modifying the union-find structure through `union`-calls is that the algorithm natively supports any union-find variant, and does not require special considerations for union-by-rank (or other strategies), unlike join.

Note that in the absence of a difference operator, Algorithm 4.1 can be transformed into a almost-linear meet operator by simply iterating on all nodes. The lemma below still applies, since we can define `diff _ _` $\triangleq \mathbb{N}$, which clearly satisfies DefDiff with complexity $\delta = |\mathbb{N}|$.

Lemma 4.1. *Algorithm 4.1 verifies* DefMeet *and runs in* $\mathcal{O}(\delta \times \alpha(|\mathbb{N}|) \times \rho)$.

Algorithm 5.1: Difference-based lattice partial order

```
let ( ⊑ ) : 𝕌𝔽 → 𝕌𝔽 → bool ≜ fun u v ↦
    diff u v ▷ forall (fun x ↦ find u x = find u v.parent[x])
```

Proof. The meet's relation is implied by the transitive closure of u or v as it only adds unions that hold in v to u. It implies u's relation (built on top of u). It also implies v's since it must contain all $x \equiv_v v.\texttt{parent}[x]$ (either present in u, or $x \in \texttt{diff}\ u\ v$ by DefDiff, and therefore added in the loop).

The algorithm calls `diff` and iterates through its result (size $\mathcal{O}(\delta)$). Each iteration calls `union` (amortized cost is $\alpha(|\mathbb{N}|) \times \rho$). □

Appendix B shows another way of computing meet by tracking a history of performed union. It is faster in theory but does not work with join, as we have not found a way to maintain the history when performing a join.

5 The Lattice Partial Order Operator

The quadratic definition of inclusion can be simplified into a linear check:

Lemma 5.1. *$u \sqsubseteq v$ is equivalent to $\forall\, x \in \mathbb{N},\ x \equiv_u v.\texttt{parent}[x]$*

Proof. ($\Rightarrow$) apply DefIncl to $x \equiv_v v.\texttt{parent}[x]$.

($\Leftarrow$) Let x, y s.t. $x \equiv_v y$, let $r_v = \texttt{repr}\ v\ x = \texttt{repr}\ v\ y$, Then, it suffices to show $x \equiv_u r_v$ and $y \equiv_u r_v$. Both cases are identical. To show the first, repeatedly apply the lemma along the path from x to r_v in v. □

With this new condition, we do not even need to check all the nodes, but simply the difference as defined Section 3.3. Therefore, the Algorithm 5.1 simply checks the condition of Lemma 5.1 on all nodes of the difference. In the absence of a difference operator, this algorithm can be made almost-linear by simply checking all nodes: recall that we can define $\texttt{diff}\ \_\ \_ \triangleq \mathbb{N}$, which clearly satisfies DefDiff with complexity $\delta = |\mathbb{N}|$.

Lemma 5.2. *Algorithm 5.1 verifies DefIncl and runs in $\mathcal{O}(\delta \times \alpha(|\mathbb{N}|) \times \rho)$.*

Proof. Correction by Lemma 5.1. For complexity, the algorithm iterates through the diff (δ), and only performs find ($\alpha(|\mathbb{N}|) \times \rho$) and a parent access ($\rho$). □

6 Two Data-Structures for Efficient Difference

The previous sections show how each lattice operator can run in almost-linear time relative to the size of the difference δ between two union-find instances. We now present two persistent map data-structures which support an efficient difference: Patricia trees using the known fast-merges [Okasaki and Gill, 1998]; and persistent arrays [Conchon and Filliâtre, 2007] using a new but straightforward difference operator.

6.1 Patricia Tree based Confluently-Persistent Union-Find

Patricia trees [Morrison, 1968] are binary radix tries on the binary representation of keys. Thanks to their structure as prefix tries, their internal representation only depends on their contents, and not on the order of insertion. This allows for a faster-than-linear intersection operator [Okasaki and Gill, 1998].

Indeed, we can iterate on a pair of trees simultaneously, which allows **skipping subtrees that only appear in one** of them and **skipping subtrees already known to be equal** (physically equal pointers). This occurs naturally if both trees derive from a common ancestor, but can also be enforced using hash-consing on the tree nodes [Filliâtre and Conchon, 2006], or tree operations that prioritize sharing whenever possible. The end result is an intersection bounded by the *intersection distance* d_{inter}, which counts all shared non-physically equal interior nodes, as well as all shared leaves with non-equal values. It ignores interior nodes and leaves present in only one of the trees.

While it is possible to construct a large intersection distance from trees that contain the same bindings but share no interior nodes, in practice, that distance is often proportional to the number of non-equal leaves times tree depth: i.e. $\mathcal{O}(\delta \times \log n)$ (similar to idempotent map operators [Blanchet et al., 2002, §6.2]).

We can use Patricia tree as our parent and rank map in union-find (combining both maps in a single tree). Their access cost for get and set operations is $\rho \triangleq \log n$, where n is the size of the tree. To save space and performance, we do not store every node in the tree, instead get assumes that absent nodes point to themselves with rank 0. Thus, we do not store nodes from singleton classes. The size of the tree is then $\mathcal{O}(u)$ where u is the *number of unions* performed, which is smaller than $|\mathbb{N}|$. Therefore, with Patricia trees, $\rho = \log u$ and $\delta = d_{\text{inter}}$.

Patricia Tree Join. To adapt the join loops from Algorithm 3.4, we use two Patricia tree operators:

$$\texttt{inter} : (\alpha, \beta)\ \texttt{PatriciaTree.t} \to (\alpha, \beta)\ \texttt{PatriciaTree.t} \to (\alpha \to \beta \to \beta \to \beta) \to (\alpha, \beta)\ \texttt{PatriciaTree.t}$$

$$\texttt{fold\_on\_neq\_inter} : (\alpha, \beta)\ \texttt{PatriciaTree.t} \to (\alpha, \beta)\ \texttt{PatriciaTree.t} \to (\alpha \to \beta \to \beta \to \gamma \to \gamma) \to \gamma \to \gamma$$

where `inter` u v f creates the intersection of u and v (which are both trees mapping keys of type α to values of type β), using f to merge differing bindings, while `fold_on_neq_inter` u v f x iterates through the non-equal bindings, calling f at each one to update the accumulator x (of type γ). Both operators iterate both trees simultaneously, skipping equal subtrees and trees present only in one of the arguments. Therefore, their cost is the intersection distance d_{inter}.

In essence, we use the set of non-equal bindings as our difference, and benefit from the shared tree structure to access it efficiently. There is a subtlety however, that set does not completely satisfy DefDiff: it skips nodes bound in one tree but not the other, even when they have different parents. This is no issue for join as such nodes correspond to singleton classes (in the tree where they are

Algorithm 6.1: Difference-based join for Patricia tree union-find, with ranks.

```
type 𝕌𝔽_pat_tree ≜ (ℕ,ℕ*int) PatriciaTree.t (*node → parent, rank *)
let PatriciaTree.join : 𝕌𝔽_pat_tree → 𝕌𝔽_pat_tree → 𝕌𝔽_pat_tree ≜ fun u v ↦
    let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
    let res ≜ PatriciaTree.inter u v
              (fun x (_,ranku) (_,rankv) ↦ x, min ranku rankv) in
    PatriciaTree.fold_on_neq_inter u v (fun x _ _ res ↦
        let r_u ≜ find u x and r_v ≜ find v x in
        match lookup new_classes (r_u, r_v) with
        | Some r_j ↦ union res x r_j
        | None ↦ new_classes ≔ new_classes[(r_u, r_v) ↦ x]; res
    ) res
```

unbound), so they must also be a singleton class in the join. There is thus no need to inspect them, the intersection can simply drop them from the result.

Lemma 6.1. *Algorithm 6.1 satisfies* DefJoin *and runs in* $\mathcal{O}(d_{inter} \times \alpha(|\mathbb{N}|) \times \log u)$.

Proof. Same as Lemma 3.6. The only difference is that we do not see nodes with differing parents when one is in a singleton class. Such nodes are removed from the join tree, so they map to themselves, as they should under DefJoin. □

Patricia Tree Meet and Inclusion. Both meet and inclusion need to iterate through the leaves of either tree, not just the shared leaves like join. However, they can still skip leaves that are equal in both trees. This can be done by simultaneously walking through both trees, skipping physically equal subtrees. Thus, all we need is the following operator:

```
fold_on_neq_union : (α,β) PatriciaTree.t → (α,β) PatriciaTree.t →
                    (α → β option → β option → γ → γ) →
                    γ → γ
```

which, given two Patricia trees, calls the argument function on their non-equal leaves, passing `None` when that binding is absent in either argument. The result (of type γ) is updated at each call. Essentially, this operator is a fold through our difference. Its cost is tied to the *union distance* d_{union} instead of the intersection distance, which is similar but also counts leaves present in only one of the trees.

Given that operator, meet is a straightforward translation of Algorithm 4.1. As a simple optimization, it does not call union on nodes who are their own parents in v. Inclusion is also straightforward adaptation of Algorithm 5.1. We simply raise and catch an exception when a counter example is found to get an early return. Algorithm 6.2 shows both adaptations.

Algorithm 6.2: Meet and inclusion algorithms for Patricia trees.

```
let PatriciaTree.meet : 𝕌_pat_tree → 𝕌_pat_tree → 𝕌_pat_tree ≜ fun u v ↦
  PatriciaTree.fold_on_neq_union u v (fun x _ r res ↦
      match r with
      | None ↦ res
      | Some (px,_) ↦ union res x px) u

let PatriciaTree.( ⊑ ) : 𝕌_pat_tree → 𝕌_pat_tree → 𝕌_pat_tree ≜ fun u v ↦
  try PatriciaTree.fold_on_neq_union u v (fun x _ r () ↦
      match r with
      | None ↦ ()
      | Some (px,_) ↦ if not x ≡_u px then raise Mismatch) ();
  true with Mismatch ↦ false
```

6.2 Persistent Array based Confluently-Persistent Union-Find

Another structure used for persistent union-find are *persistent arrays* [Baker, 1978, Conchon and Filliâtre, 2007]. They contain one "live" array (`PA_Root`) and a tree of pointers which detail changes between the current version and the live one (`PA_Diff` i v t, which represent $t[i \mapsto v]$):

```
type α parray ≜ α data ref
and α data ≜ PA_Root of α array | PA_Diff of int * α * α parray
```

Unlike Patricia trees, this is not a pure data-structure, but rather one that maintains the illusion of persistence by updating its internal pointers as it is accessed. Each get and set operation perform a *reroot*: they move the live array to the accessed pointer, inverting all `PA_Diff`s along the path. In practice, this means that the data-structure is as performant as a standard array when accessing the same version repeatedly, but pays a reroot cost for each version switch. This cost is linear in the number of (non no-op) set operations performed between both versions, which we call the *edit distance* d_{edit}.

We use the implementation of Conchon and Filliâtre [2007, §2.3]. It performs reroot before every get/set operation, using a tail-recursive reroot function written in continuation passing-style (the non tail-recursive version can overflow the stack). The set operation is also optimized to only create a `PA_Diff` node if the set value is not equal to the previous value, which is crucial for performance as path compression introduces a lot of these no-op sets.

Persistent Array Difference. Previous description of persistent arrays do not describe a difference operation. However, it is straightforward to define. To compute the difference between two persistent array versions, simply reroot the array at the left argument, and then follow pointers from the right argument back to the root, accumulating all keys along the way in a hash-set (Algorithm 6.3). This only works if both arrays are part of the same version tree, which we check

Algorithm 6.3: Persistent array difference.

```
1 let PArray.diff : α parray → α parray → int Hashset.t ≜ fun u v ↦
2     reroot u;
3     let rec fold diff data = match data with
4       | PA_Root _ ↦ assert(data == !u); diff
5       | PA_Diff(key, _, t) ↦ fold (Hashset.add diff key) !t
6     in fold (Hashset.new ()) !v
```

with an assertion (the root reached from the right argument is at the same address as the left argument, note that == in OCaml is pointer equality).

Lemma 6.2. *Algorithm 6.3 verifies* DefDiff *and runs in* $\mathcal{O}(d_{edit} + r)$ *where* d_{edit} *is the edit distance between both arguments and* r *is the edit distance between the first argument and the previously accessed array version.*

Proof. All $x \in \mathbb{N}$ s.t. $u.\texttt{parent}[x] \neq v.\texttt{parent}[x]$ must be modified along the path between u and v, thus must be seen by the diff.

For complexity, it reroots at u (cost r) and walks the path from v to u (length of d_{edit}) while only performing hash-table operations ($\mathcal{O}(1)$ amortized time). □

Persistent Array Join. For simplicity, we assume our nodes are integers: $\mathbb{N} = \texttt{int}$, so we can see the array as a direct mapping from node to parents. To maintain performance, Algorithm 6.4 reschedules Algorithm 3.3 to minimize reroots. Instead of interleaving `find` calls to both arguments in one loop, we will do multiple loops in sequence: the first performs all `find` calls on the left argument (and fetches ranks); the second performs the right `find` calls; the third resets the parents, and the fourth performs unions. Thus, we only reroot twice: once at u in the call to `diff` (line 5), and the second time at v in the second `find` loop (line 7). By initializing the result at v as well, the third loop does not require a reroot.

Lemma 6.3. *Algorithm 6.4 satisfies* DefJoin *and runs in* $\mathcal{O}(\alpha(|\mathbb{N}|) \times d_{edit} + r)$.

Proof. Correction follows from Lemma 3.6, complexity is also similar, it just requires checking when reroots occur. □

Join on the Nearest Common Ancestor. One issue with Algorithm 6.4 is that it builds the join on top of either u or v, essentially undoing all the changes between u/v and the *nearest common ancestor* (NCA) of u and v. This lengthens the chain of version changes unnecessarily, which results in worse performance when needing to reroot to that NCA (or any above ancestor). A better option is to build the join directly on top of that NCA.

Doing so requires the ability to identify that NCA. This can be done by adding a version tag to persistent arrays, with each set incrementing it by one.

Algorithm 6.4: Difference-based join for persistent array union-find, with ranks.

```
1  type 𝕌𝔽_parray ≜ { mutable parent : ℕ parray; mutable rank : int parray }
2  let PArray.join : 𝕌𝔽_parray → 𝕌𝔽_parray → 𝕌𝔽_parray ≜ fun u v ↦
3    let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
4    let (p,r) ≜ (v.parent, v.rank) in (*copy before calling find *)
5    let d ≜ PArray.diff u v
6     ▷ map (fun x ↦ (x, find u x, u.rank[x]))
7     ▷ map (fun (x,r_u,rku) ↦ (x, r_u, find v x, min rku v.rank[x])) in
8    let (p,r) ≜ fold d (fun (x,_,_,rk) (p,r) ↦
9                          p[x ↦ x],r[x ↦ rk]) (p,r) in
10   fold d (fun (x, r_u, r_v, _) res ↦
11       match lookup new_classes (r_u, r_v) with
12       | None ↦ new_classes := new_classes[(r_u, r_v) ↦ x]; res
13       | Some r_j ↦ union res x r_j) {parent=p,rank=r}
```

Algorithm 6.5: Meet and inclusion for persistent arrays

```
1  let PArray.meet : 𝕌𝔽_parray → 𝕌𝔽_parray → 𝕌𝔽_parray ≜ fun u v ↦
2    PArray.diff v u (*reroot at v*)
3    ▷ map (fun x ↦ (x, v.parent[x]))
4    ▷ fold_left (fun r (x,p) ↦ union r x p) u (*reroot at u*)
5
6  let PArray.( ⊑ ) : 𝕌𝔽_parray → 𝕌𝔽_parray → bool ≜ fun u v ↦
7    PArray.diff v u (*reroot at v*)
8    ▷ map (fun x ↦ (x, v.parent[x]))
9    ▷ forall (fun (x,p) ↦ find u x = find u p) (*reroot at u*)
```

This allows `diff` to also return the NCA (the element with the smallest tag found on the chain). The only change needed in Algorithm 6.4 is then to replace the vs on line 4 by this NCA. This modified algorithm is still correct and has the same complexity, it just performs an additional reroot from v to the NCA.

Similarly, the alternative version of join from Appendix A can also improve performance, as it avoids the double-write on parents, which results in longer `PA_Diff` chains in the result.

Persistent Array Meet and Inclusion. Both Algorithms 4.1 and 5.1 have the same issue of interleaving left and right queries in their loops. Just like join, we can improve their performance on persistent arrays by rescheduling the loop to limit reroots (Algorithm 6.5). Note that, for inclusion, this limits the worst-case cost but increases the best-case cost. Indeed, it is no longer possible to have an early return, as testing only starts after having run through the entire difference.

6.3 Performance Comparison Between Both Data-Structures

Table 1 summarizes the complexity results for both data-structures. Overall, persistent arrays are faster for the regular find/union operations as long as we limit version switches. For linear lattice operations, Patricia trees benefit from their sparse representation: there is no need to iterate through all nodes, only the u nodes that appear in unions. When that number is comparable to $|\mathbb{N}|$ however, persistent arrays become faster as they do not pay a logarithmic access cost.

For the difference-based operations, persistent arrays depend on the edit distance (the number of set operations between two branches), while Patricia tree depend on the intersection and union distances (number of different nodes and leaves), plus a logarithmic access cost. It is thus possible to find cases where either structure outperforms the other. In the worst cases, Patricia tree cost is bounded by $\mathcal{O}(|\mathbb{N}| \times \log(|\mathbb{N}|))$, while the edit distance is not really bounded (a same node may have its parent changed multiple times by successive unions/path compressions). Patricia trees also have an edge for join when both branches operate on separate nodes (as leaves absent from either of the trees are skipped) and for inclusion (as persistent arrays do not allow for an early return). In program analysis, the join cost is often the dominant factor in the overall program cost, making Patricia trees viable despite their cost overhead for union and find. Section 8 will present actual benchmarking results of both data-structures.

Table 1: Complexity upper bounds for both data-structures. u is the number of unions performed, r is the reroot cost for persistent array access (d_{edit} between the current version and the previously accessed one).

	Patricia tree	Persistent array
`find`	$\alpha(\|\mathbb{N}\|) \times \log u$	$\alpha(\|\mathbb{N}\|) + r$
`union`	$\alpha(\|\mathbb{N}\|) \times \log u$	$\alpha(\|\mathbb{N}\|) + r$
linear join/meet/incl	$\alpha(\|\mathbb{N}\|) \times u$	$\alpha(\|\mathbb{N}\|) \times \|\mathbb{N}\| + r$
difference join	$\alpha(\|\mathbb{N}\|) \times d_{\text{inter}} \times \log u$	$\alpha(\|\mathbb{N}\|) \times d_{\text{edit}} + r$
difference meet	$\alpha(\|\mathbb{N}\|) \times d_{\text{union}} \times \log u$	$\alpha(\|\mathbb{N}\|) \times d_{\text{edit}} + r$
difference inclusion	$\alpha(\|\mathbb{N}\|) \times d_{\text{union}} \times \log u$	$\alpha(\|\mathbb{N}\|) \times d_{\text{edit}} + r$ (no early return)
Memory use	u	$\|\mathbb{N}\|$

Furthermore, keeping persistent arrays efficient requires more careful consideration of side effects. One has to reason about the execution order to limit reroots, which can be difficult in large modular programs, and downright impossible in concurrent ones. Concurrent programs would also require synchronized access (using locks). Finally, persistent arrays difference only works for arrays which share a common ancestor. For two arbitrary arrays, one would have to use the linear version of the lattice operands.

Patricia trees avoid all of these complications. They are truly immutable and can thus be accessed in any order with no performance loss, and concurrent

access does not suffer any data race risk. The sparse representation also allows for efficient operations when not deriving from a common ancestor. In fact, we can re-use the difference-based algorithms directly, they will only iterate through the u-nodes that are in unions in either argument, instead of all nodes.

7 Lattice Operations on Labeled Union-Find

In this section, we adapt our lattice algorithms to *labeled union-find* (LUF) [Lesbre et al., 2025], an extension of union-find where the parent pointers are annotated with *edge labels* $\mathbb{L}$. These labels form a mathematical group: they have an infix *composition* $\cdot \fatsemi \cdot \in \mathbb{L} \to \mathbb{L} \to \mathbb{L}$, an *inverse* $\cdot^{-1} \in \mathbb{L} \to \mathbb{L}$ and a *neutral element* $\mathtt{id} \in \mathbb{L}$. In LUF, `find` not only returns a representative, but also the label between its argument and the representative (which is `id` for self representatives), and the union takes a label to add between the nodes an extra argument:

$$\mathtt{find}_{\mathbb{L}} \in \mathbb{UF}_{\mathrm{lbl}} \to \mathbb{N} \to \mathbb{N} \times \mathbb{L} \qquad \mathtt{union}_{\mathbb{L}} \in \mathbb{UF}_{\mathrm{lbl}} \to \mathbb{N} \to \mathbb{N} \to \mathbb{L} \to \mathbb{UF}_{\mathrm{lbl}}$$

Labeled union-find represents more complex relations than equality. For instance, using the *constant difference* group, where labels are integers $\mathbb{L} = \mathtt{int}$ and composition is addition, it can represent equality up-to a constant: $x = y + k$. In fact, conventional union-find can be seen as a special case of labeled union-find, where the group of labels only has a single element ($\mathbb{L} = \mathtt{unit}$).

Making labeled union-find persistent is straightforward, simply use a persistent map to store the parent pointers and labels. To define the lattice operations, since nodes are now related to a representative via a label, we must handle nodes pointing to the same representative via different labels. Formally, we no longer have a simple equivalence relation $x \equiv_u y$, but instead a *labeled relation*:

$$x \xrightarrow{\ell}_u y \iff \begin{array}{l}\text{let } (r_x, \ell_x) = \mathtt{find}_{\mathbb{L}}\ u\ x \text{ and } (r_y, \ell_y) = \mathtt{find}_{\mathbb{L}}\ u\ y \text{ in} \\ r_x = r_y \wedge \ell = \ell_x \fatsemi {\ell_y}^{-1}\end{array}$$

7.1 Labeled Union-Find Join

DefJoin can be directly restated in terms of the labeled relation:

$$\forall\, x\ y\ \ell,\ x \xrightarrow{\ell}_{\mathtt{join}\ u\ v} y \iff x \xrightarrow{\ell}_u y \ \wedge\ x \xrightarrow{\ell}_v y \qquad \text{(LblDefJoin)}$$

With the added labels on the relation, we can no longer identify in which join class a node belongs solely by it pair of representatives in u and v, as we did in Theorem 3.1. There can be multiple nodes that point to the same pair, but through different labels, and are thus not related in the join, as shown in the following example.

Example 7.1. Figure 3 presents a join between two labeled union-find with 4 nodes. Despite all nodes being in a single class both u and v, the conflicting labels mean the join has to split this single class into two separate ones.

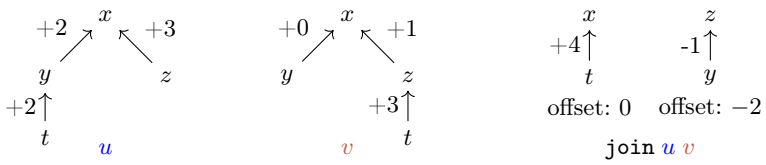


Fig. 3: LUF join splits a common class in two due to non-matching labels.

Algorithm 7.1: Difference-based LUF join, showing changes from Algorithm 3.4.

```
type 𝕌𝔽_lbl ≜ { mutable parent: ℕ ⇀ ℕ × 𝕃; mutable rank: ℕ ⇀ int }
let lookup tbl (x,y,ℓ) = if x = y && ℓ = id
                         then (x, id) else tbl[x,y,ℓ]
let join : 𝕌𝔽_lbl → 𝕌𝔽_lbl → 𝕌𝔽_lbl ≜ fun u v ↦
  let new_classes : (ℕ×ℕ×𝕃, ℕ×𝕃) Hashtbl.t ≜ Hashtbl.new () in
  let res : 𝕌𝔽_lbl ≜ { parent ≜ u.parent; rank ≜ u.rank } in
  for x in diff u v do
    res.parent := res.parent[x ↦ x, id];
    res.rank := res.rank[x ↦ min u.rank[x] v.rank[x]];
  done;
  for x in diff u v do
    let (r_u, ℓ_u) ≜ find_𝕃 u x and (r_v, ℓ_v) ≜ find_𝕃 v x in
    match lookup new_classes (r_u, r_v, ℓ_u^-1 ⨟ ℓ_v) with
    | Some(r_j, ℓ_j) ↦ res := union_𝕃 res x r_j (ℓ_u^-1 ⨟ ℓ_j)
    | None ↦ new_classes := new_classes[r_u, r_v, ℓ_u^-1 ⨟ ℓ_v ↦ (x, ℓ_u^-1)]
  done; return res
```

Fortunately, we can identify in which intersection of classes a node belongs by adding a single label to that representative pair. Indeed, all nodes of an intersection of classes will have the same *representative offset*: i.e. the label that would relate both cross-representatives of the pair when passing through the node. Thus, Algorithm 7.1 modifies Algorithm 3.3 so that `new_classes` is no longer a mapping $\mathcal{R}_{u,v} \rightharpoonup \mathbb{N}$, but a mapping $\mathcal{R}_{u,v} \times \mathbb{L} \rightharpoonup \mathbb{N}$. It modifies `new_classes` further to store the label between the new representative and the old one in u, as that label is needed to perform the union. Its correction derives from the following generalization of Theorem 3.1.

Lemma 7.2. *For all u, $v \in \mathbb{UF}_{lbl}{}^2$, nodes x, $y \in \mathbb{N}^2$ and label $\ell \in \mathbb{L}$, we have $x \xrightarrow{\ell}_u y$ and $x \xrightarrow{\ell}_v y$ if and only if the next two points hold:*

1. *x and y have the same representatives r_u in u and r_v in v*
2. *let $\ell_{xu}, \ell_{xv}/\ell_{yu}, \ell_{yv}$ be the labels between x/y and their representatives in u and v, as illustrated in Figure 4, then ${\ell_{xu}}^{-1} \fatsemi \ell_{xv} = {\ell_{yu}}^{-1} \fatsemi \ell_{yv}$*

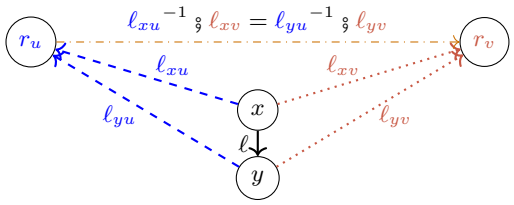


Fig. 4: Node relations in Lemma 7.2, blue dashed paths are present in u, red dotted ones in v, the black continuous path in both, and the orange dash-dotted one in neither. It is the implied offset between the cross-representatives.

Proof. ($\Rightarrow$) by definition of the labeled relations $\xrightarrow{\ell}_u$ and $\xrightarrow{\ell}_v$, x and y must have the same representatives and $\ell = \ell_{xu} \fatsemi \ell_{yu}{}^{-1}$ and $\ell = \ell_{xv} \fatsemi \ell_{yv}{}^{-1}$. Thus:

$$\begin{aligned}
\ell_{xu} \fatsemi \ell_{yu}{}^{-1} &= \ell_{xv} \fatsemi \ell_{yv}{}^{-1} \\
\ell_{yu}{}^{-1} &= \ell_{xu}{}^{-1} \fatsemi \ell_{xv} \fatsemi \ell_{yv}{}^{-1} && \text{(left product by } \ell_{xu}{}^{-1}\text{)} \\
\ell_{yu}{}^{-1} \fatsemi \ell_{yv} &= \ell_{xu}{}^{-1} \fatsemi \ell_{xv} && \text{(right product by } \ell_{yv}\text{)}
\end{aligned}$$

($\Leftarrow$) by replaying the above chain of equalities in the other direction. □

7.2 Labeled Union-Find Meet

One non-obvious limitation of the labeled relation definition $x \xrightarrow{\ell}_u y$ is that two nodes x and y can only be related through a single label. This means that union can fail when trying to add a label between nodes which are already related by a different label, as both cannot be stored internally. The meet suffers from the same issue: if the meet arguments contain different labels between nodes, the best the meet can do is choose one of them for the result, and signal the other as conflicting. For example, if we have $x \xrightarrow{\ell}_u y$ and $\xrightarrow{\ell'}_v y$ with $\ell \neq \ell'$, the meet can only keep either ℓ or ℓ'. Lesbre et al. [2025] show that labels have a flat lattice structure, so we arbitrarily decide to favor keeping relations from the left argument. This problem can also occur by transitivity, even if u and v have no directly conflicting relations.

Our best effort implementation therefore only guarantees the following:

$$\begin{aligned}
&(1)\ x \xrightarrow{\ell}_u y \ \Rightarrow\ x \xrightarrow{\ell}_{\texttt{meet}\ u\ v} y \\
&(2)\ x \xrightarrow{\ell}_v y \ \Rightarrow\ \exists\, \ell',\ x \xrightarrow{\ell'}_{\texttt{meet}\ u\ v} y \\
&(3)\ x \xrightarrow{\ell'}_{\texttt{meet}\ u\ v} y \ \Rightarrow\ \exists\, x_1..x_n\ \ell_0..\ell_n,\ \ell = \ell_0 \fatsemi \ell_1 \fatsemi \ldots \fatsemi \ell_n \wedge \\
&\qquad\qquad x \xrightarrow{\ell_0}_u x_1 \xrightarrow{\ell_1}_v x_2 \cdots x_n \xrightarrow{\ell_0}_u y
\end{aligned} \qquad \text{(LblDefMeet)}$$

Algorithm 7.2 adapts Algorithm 4.1 by handling the new labels and passing them to union. Just like Lesbre et al. [2025], we assume that $\texttt{union}_{\mathbb{L}}$ handles conflicts internally: i.e. it returns an unmodified data-structure when a conflicting union is added, optionally signaling the conflict by calling an external function.

Algorithm 7.2: Difference based labeled union-find meet algorithm.

```
let meet : 𝕌𝔽lbl → 𝕌𝔽lbl → 𝕌𝔽lbl ≜ fun u v ↦
    diff u v ▷ fold (fun res x ↦ let (y,ℓ) ≜ v.parent[x] in
                                 unionL res x y ℓ) u
```

Algorithm 7.3: Difference-based lattice partial order for labeled union-find

```
let ( ⊑ ) : 𝕌𝔽lbl → 𝕌𝔽lbl → bool ≜ fun u v ↦
  diff u v ▷ forall (fun x ↦ let (y,ℓ) ≜ v.parent[x] in x —ℓ→u y)
```

7.3 Labeled Union-Find Inclusion

The definition of inclusion can be restated for labeled relations with little change:

$$u \sqsubseteq v \quad \triangleq \quad \forall\, x\ y \in \mathbb{N},\ \forall\, \ell \in \mathbb{L},\ x \xrightarrow{\ell}_v y \Rightarrow x \xrightarrow{\ell}_u y \qquad \text{(LblDefIncl)}$$

Similarly to Lemma 5.1, it simplifies to a linear check:

Lemma 7.3. *LblDefIncl* $\iff \forall\, x \in \mathbb{N},\ \mathit{let}\ (y, \ell) \triangleq v.\texttt{parent}[x]\ \mathit{in}\ x \xrightarrow{\ell}_u y$

The proof is similar to that of Lemma 5.1. Thus, the adaptation of Algorithm 5.1 to labeled union-find (Algorithm 7.3) is straightforward and has the same complexity.

8 Evaluation

In order to evaluate our approach, we have implemented all three algorithms, join, meet and inclusion, using both Patricia trees and persistent arrays. We seek to answer the following questions:

RQ1 How well do these approaches scale with large inputs?
RQ2 How do Patricia trees and persistent arrays compare with each other?
RQ3 What cost overhead do they bring to a static analysis tool?

To answer the first two, we have run benchmarks on randomly generated inputs. For the third, we integrated our library into the Codex static analysis library. All experiments were run on a 13th Gen Intel Core i7-13800H CPU (5.20 GHz), with 62 GB of RAM.

8.1 Synthetic benchmarks

We generate random union-finds based on the number of nodes $|\mathbb{N}|$, the *number of common unions* u, and the number of *additional branch unions* δ as follows:

1. Create a new union-find by calling `make` $|\mathbb{N}|$.

2. Generate u random pairs of nodes, and call union between all pairs
3. From this common point, create two branches, each performing an additional δ random unions
4. Call the difference-based join, meet and inclusion between the two branches.

For each part of this process, we record the runtime and memory allocations. To limit time variation from the garbage collector (GC), we run a major GC collection at before timing our functions. We have run two experiments, one where we timed the functions on their own, and one where we timed the function followed by a `GC.major` call. Both showed the same trends, with the second being 10-20% slower. The results presented here correspond to timing the functions without the GC call. We run this process 50 times for each input, the results plot the average measured values along with the standard deviation.

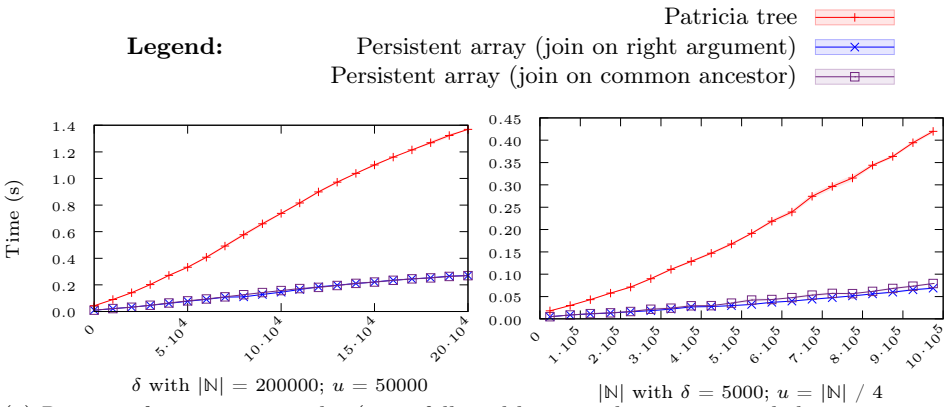
Results. The runtimes are plotted in Figure 5 as a function of δ (left) and $|\mathbb{N}|$ (right, with $u = |\mathbb{N}|/4$ to keep the union ratio constant). Unsurprisingly, Patricia tree is significantly slower for the build, as its union complexity is higher. It is comparable in performance for the lattice operations, and even significantly faster for inclusion. Our results confirm that δ is the dominant factor in performance cost, since much larger values for $|\mathbb{N}|$ and u often still run an order of magnitude faster than large δ. Note that Patricia trees cost also show an increase in the right graph, due to the increasing access cost in $\mathcal{O}(\log u)$.

Patricia trees fare better when layering joins on top of each other, as shown in Figure 6. In contrast, persistent arrays get slightly more expensive as the edit distance increases. Here we can also see the benefit of building the join on the nearest common ancestor (purple squares) instead of the left argument (blue crosses), as that minimizes the edit distance for future joins.

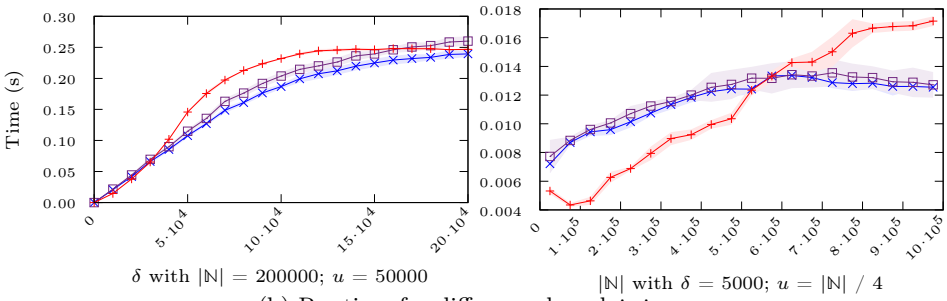
8.2 Comparing with other algorithms

Chang and Leino [2005] define a join operator on e-graphs, which includes a join of union-finds. Their algorithm is conceptually the same as our linear join (Algorithm 3.2), although more complex as their nodes are symbolic terms (they can contain other nodes), and so must be iterated in sub-term order. For comparison, we implement our simplified version on a standard array, performing explicit copies (LinearOps in Table 2), along with our linear meet and join (Algorithms 4.1 and 5.1 with `diff _ _` $\triangleq \mathbb{N}$).

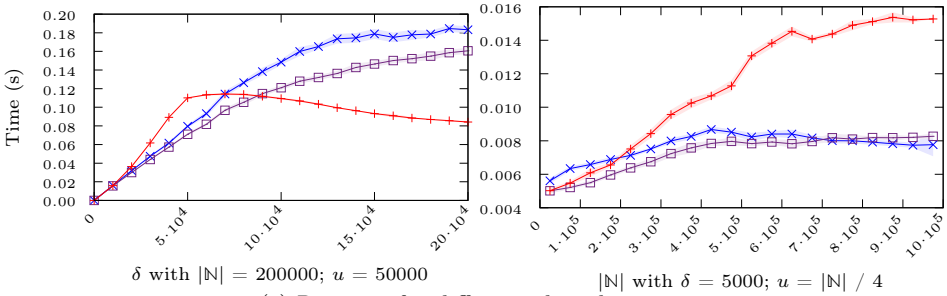
We have also found a join and inclusion in MemCAD [Cox et al., 2015, Giet et al., 2023]. The papers do not describe the algorithms used in detail, but we found two implementations in the source code, based on either AA trees [Andersson, 1993] or OCaml's balanced trees. The first uses a bi-directional map, storing both pointers to the parents and to all children. Its join corresponds to our quadratic Algorithm 3.1. It is very slow, taking over 30 minutes to run 100 random tests ($|\mathbb{N}| \leqslant 2000$, $u \leqslant 2000$, $\delta \leqslant 100$) where all our implementations run 1000 tests in less than a 0.5s. However, its inclusion is equivalent to our linear inclusion. The second implementation (ClassIntersection in Table 2, directly



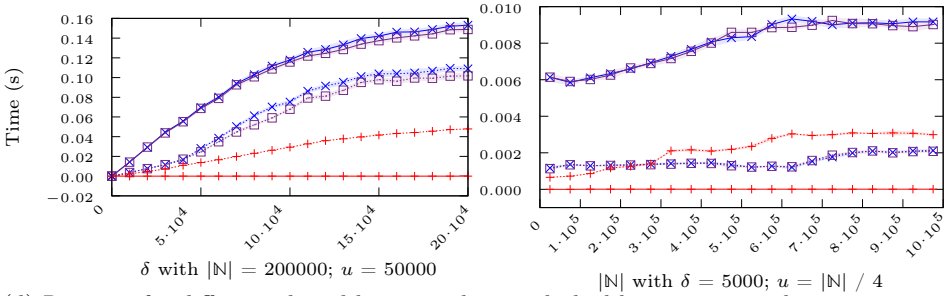
(a) Runtime for steps 1, 2 and 3 (`make` followed by u random unions and then creating 4 branches, each performing an additional δ random unions).



(b) Runtime for difference-based `join`.



(c) Runtime for difference-based `meet`.



(d) Runtime for difference-based lattice inclusion, dashed lines correspond to returning true (no early return possible), full lines to returning false.

Fig. 5: Runtime of our difference-based lattice operations, as a function of δ (left) and $|\mathbb{N}|$ (right). The curves show the average value on 50 runs, with the shaded region corresponding to ± 1 standard deviation.

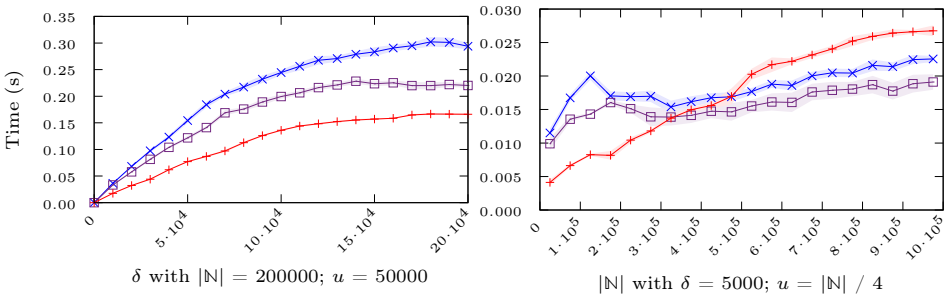


Fig. 6: Nested join runtime (specifically `join` (`join` (`join` u v) w) (`join` w z), only timing the outermost join).

Table 2: Runtime of lattice operations, in ms, $|\mathbb{N}| = 2e5$, $u = 50\,000$, $\delta = 5\,000$

Variant	join	meet	$\sqsubseteq$ (true)	$\sqsubseteq$ (false)
Patricia tree	4.5	7.1	2.1	0.002
Persistent array	9.2	7.3	1.3	6.4
Hashconsed Patricia tree	25	37.8	6.5	0.003
Sparse persistent array	17	12.6	2.3	9.4
LinearOps (array w/ copy)	114	7.3	2.9	0.002
ClassIntersection	9.9e5	–	28	0.009

taken from MemCAD) uses eager path compression and explicitly stores the equivalence classes as functional sets. Join performs the pairwise intersection of all classes, which is better than the previous version but still far worse than even the linear algorithms. Inclusion checks set inclusion on the classes.

We have also tried two other variants which had unsatisfying performance. The first was hash-consing the Patricia tree nodes to maximize sharing. It ran 3-5x slower than the non-hash-consed version, suggesting that the hash-consing cost largely overshadows the sharing benefits. The second was sparse persistent arrays: instead of allocating a huge array when the number of union is small, use a dynamic array (or vector) and only store the nodes that appear in unions (using a hash-map to renumber the nodes). This ran 1.5-2x slower while not providing any significant decrease in allocations.

8.3 Use in an Abstract Interpreter

We have run some preliminary experiment on integrating a union-find domain as part of the Codex abstract interpreter for C code. Codex translates C code to a single-static assignment (SSA) representation as part of the analysis [Lemerre, 2023, Lesbre and Lemerre, 2024], therefore it can use union-find without having to worry about variable re-assignment changing the stored relations. Algorithm 8.1 presents an example of a small C program analyzed by our analysis. We use a reduced product between the union-find domain (represented as equivalence

Algorithm 8.1: Example SSA-based analysis

```
1  int x = rand(1,7);  // {x₀} ↦ [1 : 7]
2  int y = rand(3,9);  // {x₀} ↦ [1 : 7], {y₀} ↦ [3 : 9]
3  int z = rand(0,5);  // {x₀} ↦ [1 : 7], {y₀} ↦ [3 : 9], {z₀} ↦ [0 : 5]
4  if (x == y) {       // {x₀, y₀} ↦ [3 : 7], {z₀} ↦ [0 : 5]
5    assume(y == z);   // {x₀, y₀, z₀} ↦ [3 : 5]
6    y = x + 4; }      // {x₀, y₀, z₀} ↦ [3 : 5], {y₁} ↦ [7 : 9]
7  else if (x == z)    // {x₀, z₀} ↦ [1 : 5], {y₀} ↦ [3 : 9]
8    y = z + 4;        // {x₀, z₀} ↦ [1 : 5], {y₀} ↦ [3 : 9], {y₂} ↦ [7 : 9]
9  else return; // The third branch is dropped.
10 // Join lines 8 and 6: y₃ = φ(y₁, y₂), terms y₁ and y₂ drop out of scope.
11 // {x₀, z₀} ↦ [1 : 5], {y₀} ↦ [3 : 9], {y₃} ↦ [7 : 9]
12 assert(x == z); // Proved, they are in the same class
13 assert(y == z+4); // Provable with labeled union-find
```

classes in the example) and a non-relational interval domain: instead of storing one interval per variable, we store one per equivalence class. As such, we benefit from all related variables sharing a single value, which offers some precision improvements, especially when in the presence of multiple transitive equalities. Note that, thanks to SSA terms being immutable, there is no need to forget equalities when the value of `y` updates.

The second assertion in the example is provable with labeled union-find. With that variant, $\mathtt{y}_1$ and $\mathtt{y}_2$ would be in the same relational class as $\mathtt{z}_0$, with the offset $+4$, and thus that common relation would be preserved by $\mathtt{y}_3$. Note that merging the classes of $\mathtt{y}_1$ and $\mathtt{z}_0$ forgets an interval value, but does not lose precision since interval values can be exactly shifted by addition.

Test bench. We have run the analyzer both with and without that union-find domain on 626 files totaling 177k lines of code (282 LoC on average, as counted by `cloc`). We used 20 files randomly generated by CSmith [Yang et al., 2011], 603 files from the ReachSafety category of the SV-Comp benchmarks [Beyer, 2024], and 3 are small whole-program examples.

We ran our test three times: once as a control run, with only a standard non-relational domain, once with a Patricia tree based union-find lattice, and once with a persistent array based one. Since Codex does not know in advance how many terms it will create, we modified persistent arrays to be extendable, similarly to a vector, instead of having to specify the size when calling `make`.

Results. Using Patricia tree based union-find only increases runtime cost by 5-10% on average compared to the non-relational analysis. This confirms the relatively cheap nature of this domain. In contrast, the persistent array version fared much worse: it exceeded the 20s timeout 212 of the analyses, and increased runtime by a factor of 3-4×. This is unsurprising, as Codex is not designed

to only access versions one at a time, and limit version switches, a vital key to maintaining persistent array performance.

Of the 638 files, only 19 showed precision differences. Across those files, the union-find domain proves 34 new alarms (most of which are overflow checks) and 1 new assertion.

One subtlety we discovered while implementing this domain performing union during conditional back-propagation can lead to precision loss as sub-terms of non-representatives would no longer be explored. We are investigating a solution that attaches the set of all sub-terms of class elements to each equivalence class.

9 Related Work

Union-Find in Abstract Interpretation. Chang and Leino [2005] provide a join operator on e-graphs as part of a congruence closure domain. Their algorithm is conceptually the same as our linear join (Algorithm 3.2), with a slightly more complex iteration. As their nodes contain other nodes, they iterate in the sub-term order. Thus, they do not have a single representative map (our `new_classes`), but two (from join representatives to left/right representatives).

Cox et al. [2015] use union-find in the MemCAD analyzer in an equality functor, to reduce the number of variables sub-domains have to handle, while Giet et al. [2023] use it to represent equality between sequence variables in the sequence abstract domain. The papers do not describe the union-find lattice operands, but the implementations we found in the code are far slower than ours.

Lesbre et al. [2025] use an imperative, pointer-based labeled union-find to represent affine relations, but only in a flow insensitive context and so do not need lattice operations. They store relations between SSA terms [Lesbre and Lemerre, 2024] such as $x + 1 \xrightarrow{+1} x$. This, allows a reduced product with other domains, decreasing the number of terms the other domain has to handle.

Contextual E-Graphs. While e-graph are commonly used to perform full-program analysis and thus do not incorporate contexts, there has been recent research into tackling this problem, namely to allow conditional rewrites. Various solutions have been proposed. *Context-aware rewrite rules* [Drewery et al., 2025] extend e-graph by annotating e-nodes with a list of possible contexts, with conditional rewrite copying the node (and any parent). *Assume nodes* [Coward et al., 2023] create a new version of the terms in if-branches, essentially avoiding context-sensitivity by creating new terms for each context in a manner similar to SSI forms [Ananian, 1999, Boissinot et al., 2012]. *Colored e-graphs* [Singher and Itzhaky, 2024] introduce context-sensitivity by having multiple layers of union-find instances, with each version introducing a new layer. Unlike our version, they do not allow for join or meet, but allow changes from one version (path compression, new unions) to propagate to its children, which our version does not. Finally, Cesario et al. [2026] present a versioned e-graph based on a single, versioned union-find data structure. It extends union-find with version

tags, and is thus able to support a tree of versions, like persistent union-find, but within a single data-structure instance. It is thus more compact and has better sharing between versions. However, none of these methods incorporate any lattice operations. They all operate on a tree of contexts only.

Constraint Programming. Conchon and Filliâtre [2007] present persistent union-find to help with backtracking in constraint programming [Schulte and Carlsson, 2006]. Example use cases include unification, such as in type systems [Milner, 1978] or in Prolog; and decision procedures [Shostak, 1984]. For these use cases as well, the solution only allow for a tree of versions, not a lattice as our method does.

10 Conclusion

We have presented three lattice operations on persistent union-find data structures: join, meet and inclusion. We provide formal definitions for each in terms of the represented equivalence relations, as well as algorithms to compute them. A key idea of our algorithms is to use a difference operator to minimize the amount of work, with our algorithm running in almost-linear time relative to the size of that difference. We describe two data-structures that provide such a difference, Patricia trees and persistent arrays. Experimentally, our algorithms offer great performance even with fairly large inputs, and outperform the previous implementations we could find in literature, and can be integrated in existing analyzer with a fairly low cost overhead.

Data-Availability Statement

The algorithms presented in this paper are publicly available as an OCaml library, published on opam under the name `union-find-lattice` [Lesbre and Lemerre, 2026b]. The library is open-source under a LGPLv2.1 license.

Alongside the library, which may be subject to change and was trimmed down to only keep the best performing variants, we released a software artifact [Lesbre and Lemerre, 2026a] containing the benchmarks from Section 8, as well as a version of the Codex library which include the experimental union-find domain.

Acknowledgements

This research was supported in part by the Agence Nationale de la Recherche (ANR) grant agreement ANR-22-CE39-0014-03 (EMASS project).

Bibliography

C. S. Ananian. The Static Single Information Form. Master's thesis, Princeton University, Sept. 1999.

A. Andersson. Balanced search trees made simple. In *Algorithms and Data Structures*, WADS '93, pages 60–71, Berlin, Heidelberg, 1993. Springer. ISBN 978-3-540-47918-5. https://doi.org/10.1007/3-540-57155-8_236.

H. G. Baker. Shallow binding in Lisp 1.5. *Commun. ACM*, 21(7):565–569, July 1978. ISSN 0001-0782. https://doi.org/10.1145/359545.359566.

D. Beyer. SV-Benchmarks: Benchmark Set for Software Verification (SV-COMP 2024), 2024. https://doi.org/10.5281/zenodo.10669723.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In *The Essence of Computation: Complexity, Analysis, Transformation*, pages 85–108. Springer, Berlin, Heidelberg, 2002. ISBN 978-3-540-36377-4. https://doi.org/10.1007/3-540-36377-7_5.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38(5):196–207, May 2003. ISSN 0362-1340. https://doi.org/10.1145/780822.781153.

B. Boissinot, P. Brisk, A. Darte, and F. Rastello. SSI Properties Revisited. *ACM Trans. Embed. Comput. Syst.*, 11S(1):21:1–21:23, June 2012. ISSN 1539-9087. https://doi.org/10.1145/2180887.2180898.

J. G. Cesario, G. Zakhour, P. Weisenburger, and G. Salvaneschi. Versioned E-Graphs. *Proc. ACM Program. Lang.*, 10, 2026. https://doi.org/10.1145/3808249.

B.-Y. E. Chang and K. R. M. Leino. Abstract Interpretation with Alien Expressions and Heap Structures. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, VMCAI '05, pages 147–163, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-30579-8. https://doi.org/10.1007/978-3-540-30579-8_11.

S. Conchon and J.-C. Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, pages 37–46, New York, NY, USA, Oct. 2007. Association for Computing Machinery. ISBN 978-1-59593-676-9. https://doi.org/10.1145/1292535.1292541.

S. Coward, G. A. Constantinides, and T. Drane. Automating Constraint-Aware Datapath Optimization using E-Graphs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, DAC '23, pages 1–6, July 2023. https://doi.org/10.1109/DAC56929.2023.10247797.

A. Cox, B.-Y. E. Chang, H. Li, and X. Rival. Abstract Domains and Solvers for Sets Reasoning. In *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '15, pages 356–371, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-48899-7. https://doi.org/10.1007/978-3-662-48899-7_25.

A. Drewery, T. P. Jensen, and D. Pichardie. Contextual Equality Saturation. In *Static Analysis*, SAS '25, pages 34–61, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-032-07106-4. https://doi.org/10.1007/978-3-032-07106-4_3.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 109–121, New York, NY, USA, Nov. 1986. Association for Computing Machinery. ISBN 978-0-89791-193-1. https://doi.org/10.1145/12130.12142.

J. Feret. *Analyse Des Systèmes Mobiles Par Interprétation Abstraite.* Thesis, École Polytechnique, Feb. 2005. URL https://pastel.hal.science/pastel-00001303.

A. Fiat and H. Kaplan. Making data structures confluently persistent. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 537–546, USA, Jan. 2001. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-490-6. https://doi.org/10.5555/365411.365528.

J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 12–19, New York, NY, USA, Sept. 2006. Association for Computing Machinery. ISBN 978-1-59593-483-3. https://doi.org/10.1145/1159876.1159880.

J. Giet, F. Ridoux, and X. Rival. A Product of Shape and Sequence Abstractions. In *Static Analysis*, SAS '23, pages 310–342, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-44245-2. https://doi.org/10.1007/978-3-031-44245-2_15.

A. Goel, S. Khanna, D. H. Larkin, and R. E. Tarjan. Disjoint set union with randomized linking. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1005–1017, USA, Jan. 2014. Society for Industrial and Applied Mathematics. ISBN 978-1-61197-338-9. https://doi.org/10.5555/2634074.2634149.

G. F. Italiano and N. Sarnak. Fully Persistent Data Structures for Disjoint Set Union Problems. In *Algorithms and Data Structures*, volume 519 of *WADS '91*, pages 449–460, Berlin, Heidelberg, 1991. Springer. ISBN 978-3-540-47566-8. https://doi.org/10.1007/BFB0028283.

H. Kaplan, N. Shafrir, and R. E. Tarjan. Union-find with deletions. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 19–28, USA, Jan. 2002. Society for Industrial and Applied Mathematics. ISBN 978-0-89871-513-2.

M. Lemerre. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.*, 7(POPL):65:1895–65:1924, Jan. 2023. https://doi.org/10.1145/3571258.

D. Lesbre and M. Lemerre. Compiling with Abstract Interpretation. *Proc. ACM Program. Lang.*, 8(PLDI):162:368–162:393, June 2024. https://doi.org/10.1145/3656392.

D. Lesbre and M. Lemerre. Artifact for "A Lattice of Union-Finds", July 2026a. https://doi.org/10.5281/zenodo.21217874.

D. Lesbre and M. Lemerre. Union-Find Lattice Library, July 2026b. Version 0.1.0, URL https://github.com/codex-semantics-library/union-find-lattice.

D. Lesbre, M. Lemerre, H. R. Ait-El-Hara, and F. Bobot. Relational Abstractions Based on Labeled Union-Find. *Proc. ACM Program. Lang.*, 9(PLDI):195:1194–195:1219, June 2025. https://doi.org/10.1145/3729298.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, Dec. 1978. ISSN 00220000. https://doi.org/10.1016/0022-0000(78)90014-4.

A. Miné. A Few Graph-Based Relational Numerical Abstract Domains. In *Static Analysis*, SAS '02, pages 117–132, Berlin, Heidelberg, 2002. Springer. ISBN 978-3-540-45789-3. https://doi.org/10.1007/3-540-45789-5_11.

D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968. ISSN 0004-5411. https://doi.org/10.1145/321479.321481.

C. Okasaki and A. Gill. Fast Mergeable Integer Maps. ML '98, 1998.

C. Schulte and M. Carlsson. Chapter 14: Finite Domain Constraint Programming Systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 512–514. 1st edition edition, Oct. 2006. ISBN 978-0-444-52726-4.

R. E. Shostak. Deciding Combinations of Theories. *J. ACM*, 31(1):1–12, Jan. 1984. ISSN 0004-5411. https://doi.org/10.1145/2422.322411.

E. Singher and S. Itzhaky. Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, FMCAD '24, pages 70–83, Oct. 2024. https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_13.

B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, Jan. 1996. Association for Computing Machinery. ISBN 978-0-89791-769-8. https://doi.org/10.1145/237721.237727.

R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, Apr. 1975. ISSN 0004-5411. https://doi.org/10.1145/321879.321884.

R. E. Tarjan and J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, Mar. 1984. ISSN 0004-5411. https://doi.org/10.1145/62.2160.

M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):23:1–23:29, Jan. 2021. https://doi.org/10.1145/3434304.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, June 2011. Association for Computing Machinery. ISBN 978-1-4503-0663-8. https://doi.org/10.1145/1993498.1993532.

Algorithm A.1: Difference-based join which directly edits parents.

```
let lookup tbl (x,y) = if x = y then Some x
                       else Hashtbl.lookup tbl (x,y)
let join : UF → UF → UF ≜ fun u v ↦
    let new_classes : (ℕ × ℕ, ℕ) Hashtbl.t ≜ Hashtbl.new () in
    let res : UF ≜ { parent = u.parent } in
    for x in diff u v do
        let r_u ≜ find u x and r_v ≜ find v x in
        match lookup new_classes (r_u, r_v) with
        | Some r_j ↦ res.parent := res.parent[x ↦ r_j]
        | None ↦ res.parent := res.parent[x ↦ x];
                  new_classes := new_classes[(r_u, r_v) ↦ x]
    done; return res
```

A Join by Editing Parents

The join algorithms from Section 3 proceed by calling union to build the new classes. This has the benefit of supporting multiple union-find variant, like union-by-rank, with very few modifications. However, it does extra work: we call union in a state where the representative of the classes we union are known. For the difference-based version, we also write the parents twice: once to reset the parents in the first loop, and once when performing union.

We can modify our algorithm to edit the parents directly instead of calling union, thus avoiding some of this work. Algorithm A.1 adapts our difference-join (Algorithm 3.3) to use this technique, and is fairly similar.

However, this requires more careful bookkeeping when supporting variants like union-by-rank (Algorithm A.2). In that case, we must both maintain the rank information and use it to choose the representative for the new classes (which union used to do for free). Thus, we iterate through the difference once to find the best representative, keeping track of found ranks in `new_classes`, along with a boolean which specifies whether the class contains an element of equal rank, in which case rank will have to be increased. We then iterate a second time to set the parents.

Experimentally, these improve algorithms have similar performance for a single join, but show up to a 10% runtime improvement when layering three joins on top of each other.

B Meet on Union History

Another option to compute the meet to keep track of the performed unions, i.e. to record the which calls to `union` (between unrelated nodes) were performed in each version as a linked list. This leads to a tree like structure, with all versions rooted at the empty list in the empty union-find. Meet finds the common ancestor

Algorithm A.2: Difference-based join by rank with direct parent edits, showing changes from Algorithm A.1.

```
let join : 𝕌𝔽_rank → 𝕌𝔽_rank → 𝕌𝔽_rank ≜ fun u v ↦
  let lookup tbl (x,y) ≜ if x = y
      then Some(x, min u.rank[x] v.rank[x], false)
      else Hashtbl.lookup tbl (x,y) in
  let new_classes : (ℕ×ℕ, ℕ×int×bool) Hashtbl.t ≜ Hashtbl.new() in
  for x in diff u v do (* initialize new_classes *)
      let r_u ≜ find u x and r_v ≜ find v x in
      let rx ≜ min u.rank[x] v.rank[x] in
      match lookup new_classes (r_u, r_v) with
      | None ↦ new_classes := new_classes[r_u,r_v ↦ x,rx,false]
      | Some(r_j, r, _) ↦ if r = rx
          then new_classes := new_classes[r_u,r_v ↦ r_j,r,true]
          else if r < rx then
              new_classes := new_classes[r_u,r_v ↦ x,rx,false]
  done;
  let res : 𝕌𝔽 ≜ { parent=u.parent; rank=u.rank } in
  for x in diff u v do (* point nodes to their new parents *)
      let r_u ≜ find u x and r_v ≜ find v x in
      match lookup new_classes (r_u, r_v) with
      | Some (r_j, r, b) ↦ res.parent := res.parent[x ↦ r_j];
          let rx = if x = r_j then r + Bool.to_int b
                   else min u.rank[x] v.rank[x] in
          res.rank := res.rank[x ↦ rx]
      | None ↦ failwith "Unreachable"
  done; return res
```

Algorithm B.1: History based meet algorithm.

```
type 𝕌𝔽_with_history ≜ { mutable parent: ℕ ⇀ ℕ; hist: (ℕ × ℕ) SList.t }
let union u x y ≜ {
  parent ≜ ... (* do union as usual *)
  history ≜ if x ≡_u y then u.hist else (x,y)::u.hist; }

let rec walk_history res hist1 hist2 = match hist1, hist2 with
  | _, _ when &hist1 = &hist2 ↦ res (*equal pointer ⇔ common ancestor*)
  | (a,b)::h1, _::h2 ↦ walk_history (union res a b) h1 h2
let meet : 𝕌𝔽_with_history → 𝕌𝔽_with_history → 𝕌𝔽_with_history ≜ fun u v ↦
  let lu ≜ SList.length u.hist and lv ≜ SList.length v.hist in
  if lu < lv
  then walk_history v u.hist (SList.tail_n (lv−lu) v.hist)
  else walk_history u v.hist (SList.tail_n (lu−lv) u.hist)
```

between two versions by unstacking these lists, and then applies the unions from one of the branches of that ancestor to the result of the other branch.

This alternative to Section 4 takes more memory space but has several advantages. For one, the difference it examines is only due to new unions, and not changes from path compression. It is therefore smaller. Secondly, by tracing both arguments to their common ancestor, it can easily choose which element would require the least modification (the one with the longest path to the ancestor), thus performing fewer unions. In the case of a meet with an ancestor, it can easily return that value unchanged. Algorithm 4.1 would only do that if the ancestor is passed as a second argument.

In fact, when using linked lists that store their size (for a constant time `length` operation), which we call `SList`, it is possible to do the unions directly as part of the walk to the common ancestor. A further optimization when performing many meets would be to use lists which contain skip-pointers[2] allowing for a faster-than-linear `tail_n` operation. This is presented in Algorithm B.1 (to clarify the subtle difference between OCaml's `=` and `==` operators, we write pointer equality using C-like address operators: `&x = &y`).

The big downside of this meet algorithm is that, since it requires additional information, it cannot be performed on top of join. Specifically, it could be used with Algorithm 3.1, as it only modifies the union-find through calls to union, but the more efficient Algorithms 3.2 and 3.3 directly modify the parents. Thus, they would have to be modified to construct a valid union history, which is a non-trivial problem we have not looked into.

[2]For example, add a pointer to the k-th tail every k nodes, or, every 2^k nodes, add pointers to previous powers of 2.

Lemma B.1. *Algorithm B.1 satisfies* DefMeet *and runs in* $\mathcal{O}(\alpha(|\mathbb{N}|)\times\rho\times d_{hist})$ *where the* history distance d_{hist} *is the number of unions performed since the latest common ancestor. Furthermore,* $d_{hist} < |\mathbb{N}|$.

Proof. By symmetry, it suffices to prove the case where `lu` $<$ `lv`. `walk_history` is called with v, u.hist and the last `lu` elements of v.hist. Both lists thus have the same size. `walk_history` then walks up both list simultaneously, maintaining the invariant that their sizes are always equal. It is thus guaranteed to find the nearest common ancestor (NCA), since it examines every pair of elements with the same depth until it finds physically equal elements (or two empty lists, which are always physically equal). Doing so, it adds to v all unions from u.hist, which, by definition of union, are all new unions (not implied by previous unions). Therefore, it must add exactly all the union performed on u since the version split at the NCA (albeit in reverse order).

Hence, the results has all union of v (initialized at v), and adds all union performed by u since its nearest common ancestor with v. Thus, it must also contain all union of u.

For complexity, `SList` have a constant-time size operation, `tail_n`'s complexity is bounded by its argument size, and walk_history performs a linear number of steps up-to that common ancestor. Furthermore, each step performs union, so has a $\mathcal{O}(\alpha(|\mathbb{N}|)\times\rho)$ cost.

Since history only grows when new unions (not previously implied by the union-find) are added, the length of a history list is at most one smaller than the number of nodes (as performing $|\mathbb{N}|-1$ non-redundant unions will necessarily relate all nodes). □

C Proofs

Proof of Lemma 3.2.

Lemma 3.2. *Algorithm 3.2 satisfies* DefJoin *and runs in* $\mathcal{O}\big(\,|\mathbb{N}|\times\alpha(|\mathbb{N}|)\times\rho\big)$.

Proof. Using Theorem 3.1 with $f \triangleq$ `repr res` $\circ$ `new_classes`. For (1), let $x\in\mathbb{N}$, there are two cases:

- (`repr` u x, `repr` v x) is not in `new_classes`, in which case it is added, pointing to x
- otherwise, we union x and `new_classes`[(`repr` u x, `repr` v x)]

Therefore, in both cases, $x \equiv_{\texttt{res}}$ `new_classes`[(`repr` u x, `repr` v x)], so our choice for f, `repr res` $\circ$ `new_classes` gives the representative of x in the join.

For (2), the values in `new_classes` are all disjoint since at most one value is added per iteration, and that value is the current iteration's node. Furthermore, all performed unions are between the current iteration's node (which was not seen before and is thus in a singleton class) and a value from `new_classes`. Therefore, two values from `new_classes` can never be in the same class: their representatives remain disjoint.

For complexity: the algorithm loops through all nodes. Iteration can call `find` and `union`, whose amortized complexity is $\mathcal{O}(\alpha(|\mathbb{N}|) \times \rho)$ [Tarjan, 1975] and performs hash-table operations (constant-time amortized complexity). □

Proof of Lemma 3.6.

Lemma 3.6. *Algorithm 3.3 satisfies* DefJoin *and runs in* $\mathcal{O}(\delta \times \alpha(|\mathbb{N}|) \times \rho)$.

Proof. We prove correction using the characterization of Theorem 3.1. We start by proving (1), using $f \triangleq \texttt{repr res} \circ \texttt{lookup new\_classes}$. Let $x \in \mathbb{N}$:

If $x \in \texttt{diff}\ u\ v$, then by the same reasoning as the above proof, x is related to $\texttt{new\_classes}[(\texttt{find}\ u\ x, \texttt{find}\ v\ x)]$, and so share its `repr`.

If $c \notin \texttt{diff}\ u\ v$, then $u.\texttt{parent}[x] = v.\texttt{parent}[x]$ (by DefDiff). By symmetry, we can suppose that the path between x and its representative is shorter in u than in v. We reason by induction on the length k of that path between x and its representative in u, to prove the following:

$$H(k) \triangleq \forall\, x \notin \texttt{diff}\ u\ v, \text{ such that the path from } x \text{ to } \texttt{repr}\ u\ x \text{ has length } k,\\ \texttt{lookup new\_classes}\ (\texttt{repr}\ u\ x, \texttt{repr}\ v\ x) \equiv_{res} x$$

- If that path has length 0, then $x = u.\texttt{parent}[x] = v.\texttt{parent}[x]$. Thus, $\texttt{repr}\ u\ x = \texttt{repr}\ v\ x$ and so the equality check in `lookup` ensures that $\texttt{lookup new\_classes}\ (\texttt{repr}\ u\ x, \texttt{repr}\ v\ x) = x$
- Otherwise, $u.\texttt{parent}[x] = v.\texttt{parent}[x] \neq x$.
  We have $res.\texttt{parent}[x] = u.\texttt{parent}[x]$ since the algorithm only modifies parents of elements in $\texttt{diff}\ u\ v$, or, through union, the parents of representatives. It then suffices to show the result for that parent. Either it is in $\texttt{diff}\ u\ v$, and the result is proved above, or it is not, and we can apply the induction hypothesis.

For injectivity (2), we start by showing $\texttt{lookup} \circ \texttt{new\_classes}$ is injective on $\mathcal{R}_{u,v}$. The values returned by `lookup` when $x = y$ are clearly distinct from each other. Those added in the loop are also distinct from each other, since at most one value is added per iteration and that value is unique to that iteration. Furthermore, that added value, x, must have two different representatives in u and v (since it is added in the None branch of the match). Thus, (x, x) is not a valid pair of representatives from (u, v), and it does not conflict with the values added by `lookup`.

Next, $\texttt{repr res}$ does not break injectivity. Indeed, all performed `union` occur between two elements which have the same image by $\texttt{lookup} \circ \texttt{new\_classes}$.

The complexity is straightforward. The algorithm calls `diff` and thus inherits its complexity δ. Furthermore, that complexity must be an upper-bound of the size of the result, and thus the number of iterations for both loops. The first loop does parent accesses (which are $\mathcal{O}(\rho)$). The second loop calls `find` and `union` (whose amortized complexity is $\mathcal{O}(\rho\,|\mathbb{N}|)$), and performs hash-table operations (constant amortized complexity). □