Colored E-Graph: Equality Reasoning with Conditions

Anonymous

Anonymous Anonymous
Anonymous

Abstract. E-graphs are a prominent data structure that has been increasing in popularity in recent years due to their expanding range of applications in various formal reasoning tasks. Often, they are used for equality saturation, a process of deriving consequences through repeatedly applying universally quantified equality formulas via term rewriting. They handle equality reasoning over a large spaces of terms, but are severely limited in their handling of case splitting and other types of logical cuts, especially when compared to other reasoning techniques such as sequent calculi and resolution. The main difficulty is when equality reasoning requires multiple inconsistent assumptions to reach a single conclusion. Ad-hoc solutions, such as duplicating the e-graph for each assumption, are available, but they are notably resource-intensive.
We introduce a key observation is that each duplicate e-graph (with an added assumption) corresponds to coarsened congruence relation. Based on that, we present an extension to e-graphs, called *Colored E-Graphs*, as a way to represent all of the coarsened congruence relations in a single structure. A colored e-graph is a memory-efficient equivalent of multiple copies of an e-graph, with a much lower overhead. This is attained by sharing as much as possible between different cases, while carefully tracking which conclusion is true under which assumption. Support for multiple relations can be thought of as adding multiple "color-coded" layers on top of the original e-graph structure, leading to a large degree of sharing.
In our implementation, we introduce optimizations to rebuilding and e-matching. We run experiments and demonstrate that our colored e-graphs can support hundreds of assumptions and millions of terms with space requirements that are an order of magnitude lower, and with similar time requirements.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

E-graphs are a versatile data structure that is used for various tasks of automated reasoning, including theorem proving and synthesis. They are especially effective in reasoning about equality. E-graphs have been popularized in compiler optimizations thanks to their ability to support efficient *rewrites* over a

large set of terms, while keeping a compact representation of all possible rewrite outcomes. This mechanism is known as *equality saturation*. It provides a powerful engine that allows a reasoner to generate all equality consequences of a set of known, universally quantified equalities. Possible uses include selecting the best version of an expression according to some desired metric, such as run-time efficiency [28], size [9,21], or precision [22] (when used as a compilation phase) and a generalized form of unification, called e-unification, for application of inference steps (when used for proof search).

In this work we focus on *exploratory reasoning* tasks such as theory exploration [25], rewrite rule inference [19], and proof search [15,5,13]. Exploratory reasoning is setting where some intermediate (or even final) conjectures or goals are not known *a priori*, and need to be discovered. For example, in the course of theory exploration, we traverse the domain of list expressions to find all valid equalities. One of the candidate equalities encountered may be:

$$\mathsf{filter}\ q\ (\mathsf{filter}\ p\ [x, y]) = \mathsf{filter}\ p\ (\mathsf{filter}\ q\ [x, y])$$

Where the definition of filter is

$$\mathsf{filter}\ p\ (x :: xs) = \text{if}\ p\,x\ \text{then}\ x :: \mathsf{filter}\ p\ xs\ \text{else}\ \mathsf{filter}\ p\ xs$$

We wish that the automatic reasoning technique simplify both terms to a common form, in order to prove that they are equal. A term like filter $p\ [x, y]$ is rewritten using the definition to if $p\,x$ then $x ::$ filter $p\ [y]$ else filter $p\ [y]$. In this example we assume an equality saturation engine, based on an e-graph, that can simplify terms of the form if true... and if false..., but cannot directly rewrite if $p\,x$... because $p\,x$ does not match either true or false; therefore, this term requires special treatment.

When encountering a term like if $p\,x$, previous work based on rewriting, specifically equality saturation, either leaves it as opaque, or performs *case splitting* by forking the entire state (in this case, the e-graph). Case splitting, a common reasoning step, is needed to introduce alternative assumptions $p\,x$ and $\neg p\,x$ and consider each one. With the forking case splitting method we will have one copy in which filter $p\ [x, y]$ rewrites to $x ::$ filter $p\ [y]$ and one copy in which it rewrites to filter $p\ [y]$. The two copies will have a shared term filter $p\ [y]$, but this cannot be taken advantage of, because the copies are disjoint; therefore any rewrite that applies to the term will now have to be carried out twice, doubling both time and memory consumption. Furthermore, rewriting of this latter term, filter $p\ [y]$, will similarly have to case-split on $p\,y$, so each copy will have to be cloned again. This, even though the rewriting of filter $p\ [y]$ depends only on $p\,y$ and is agnostic to which case of $p\,x$ is taken. Further rewriting of the terms in the equality above exacerbates problem, because $q\,x$, $q\,y$ also occur in conditions, perpetrating additional case-splits, and leading to even more duplication ($\times 16$) as well as redundant, repeated work. In larger examples, the reasoner will quickly exhaust the available memory as a result.

We note that in exploratory tasks it is often necessary to keep all the exploration paths rather than "finish off" proof branches iteratively. The reason is

that an incomplete path can be extended later in the exploration with the help of other conclusions. This makes the memory limitation acute to these scenarios.

Our key observation is that each assumption may lead to additional unions, but may never "break apart" classes of terms. An e-graph naturally represents a congruence relation $\cong$, which is an equality relation over terms (with function applications), which maintains $x \cong y \vdash f(x) \cong f(y)$. Any union done on the assumption-less congruence relation, which we call the "root" congruence relation, will also be correct on any forked e-graph's congruence relation that holds additional assumptions. This observation is important because such unions in the root can be shared across all other derived e-graphs.

We extend the e-graph data structure into a *Colored E-Graph* to maintain multiple congruence relations at once, where each relation is associated with a color. A colored e-graph contains multiple congruence relations without having to duplicate the shared terms, by holding a union-find structure for each relation. For the example above, we will use a colored e-graph with different colors for assumptions on the truth value of the if condition, $p\,x$. The root $\cong$ will be represented by the color black. The color blue will represent the assumption $p\,x$, which we denote $p\,x \cong_{\mathrm{b}} \mathsf{true}$, and the color red will represent $\neg p\,x$ which we denote $p\,x \cong_{\mathrm{r}} \mathsf{false}$. Equality saturation can now infer, via rewriting, the consequences $\mathsf{filter}\ p\ [x, y] \cong_{\mathrm{b}} x :: \mathsf{filter}\ p\ [y]$ and $\mathsf{filter}\ p\ [x, y] \cong_{\mathrm{r}} \mathsf{filter}\ p\ [y]$. When creating the new blue and red, the term $\mathsf{filter}\ p\ [x, y]$ will not be copied; instead, it is shared with the black color.

In addition, having a hierarchy between different colors can further benefit from such sharing. As mentioned, two new assumptions are needed for case splitting on the if condition, $p\,y$. Any conclusion in the preceding congruence relations, whether root or blue, is also true for new relations added for $p\,y$, on top of blue. In addition, terms will still remain uncopied, as the sharing persists for the additional congruence relations.

We also extend the colored e-graph's logic in several key ways, to support e-graph operations for all the colors. First, we set up a multi-level union-find where the lowest level corresponds to the root congruence. Higher levels define further unions of e-class representatives from the lower ones, forming *colored e-classes*, giving a coarsening of relations lower in the hierarchy. Second, we change how congruence closure is applied to the individual congruence relations while taking advantage of the sharing between each such relation and the root. Lastly, we present a technique for e-matching over all the relations at once.

Our contributions:

1. The observation that assumptions induce coarsened e-graphs that share much of the original structure.
2. Algorithms for colored e-graphs operations.
3. Optimizations on top of the basic algorithms to significantly improve resource usage.

4. A colored e-graph implementation[1] and an evaluation that shows an improvement factor in memory usage over the existing baseline, while maintaining similar run-time performance.

2 Background on E-graphs

This section presents some basic definitions and notations that will be used throughout the paper. We assume a term language L where terms are constructed using *function symbols*, each with its designated arity. We use $f^{(r)} \in \Sigma[L]$ to say that f is in the *signature* of L and has arity r. A term is then a *tree* whose nodes are labeled by function symbols and a node labeled by f has r children. (In particular, the leaves of a term have nullary function symbols.)

An e-graph $\mathcal{G}$ serves as a compact data structure representing a set $S \subseteq L$ of terms and a congruence relation $\cong \subseteq L \times L$. This congruence relation, in addition to being reflexive, symmetric, and transitive, is also closed under the function symbols of $\Sigma[L]$. That is, for every $f^r \in \Sigma[L]$, and given two lists of terms $t_{1..r} \in L$ and $s_{1..r}$, each of length r, if $t_i \cong s_i$ $(i = 1..r)$, then it follows that $f(t_1, \ldots, t_r) \cong f(s_1, \ldots, s_r)$. This property, known as *congruence closure*, is a key attribute of the data structure. The maintenance of this attribute as an invariant significantly influences the design and implementation of e-graph actions.

The egg library [29] revolutionizes the application of e-graphs by explicitly supporting the equality saturation workflow. It enables the periodic maintenance of congruence closure, via *deferred rebuild*, allowing for the amortization of associated rebuilding costs. We give a short background on how egg achieves better performance by means of efficient data structure representation.

In egg, the authors present the e-graph as a union-find-like data structure, augmented to support operations on expressions. This implementation is primarily achieved through the utilization of three key structures: a hash-cons table, a union-find structure, and an e-class map. These structures collectively underpin the functionalities integral to the operation of the e-graph.

(a) The union-find component is responsible for keeping track of merged e-classes and maps each e-class id to a single representative for all (transitively) merged e-classes. This information is later used to canonicalize the keys and values of the hash-cons.
(b) The e-class map stores the structure of the e-graph. For each e-class id, the map keeps all the e-nodes that are contained therein. E-nodes are similar to AST nodes except that their children point to e-class ids instead of containing a single sub-term each. Through the contained e-nodes, child e-classes can be reached. For efficiency, the e-graph also keeps the parents of each e-class, i.e., those of whose it is a child of.
(c) The hash-cons table maps e-nodes to their containing e-class id. An important aspect of the hash-cons is that after rebuilding its keys and values

[1] [Anonymized]

are expected to be *canonical*. That is, whenever e-classes are merged one of their ids becomes "the" representative. A *canonical* hash-cons will only contain representative ids. For example, if x and y are merged into y, then $f(x, z) \mapsto x$ must become $f(y, z) \mapsto y$.

Note, that both the hash-cons and the parent e-classes can be derived from the e-nodes contained in the e-class map. But, they are important aspects of the efficient e-graph implementation presented, and therefore are mentioned as part of the data structure.

An e-class with id e represents a set of terms defined recursively as:

$$L(e) = \{f(t_1, .., t_k) \mid f(e_1, .., e_k) \in M(e), t_i \in L(e_i) \text{ for } i = 1..k\}$$

We will use the notation $[t]$ to refer to e-class id where $t \in L([t])$.

Example 1. The term $a \cdot (b + c)$ may be stored in an e-graph using five e-classes $\langle 1\rangle$ though $\langle 5\rangle$ (angle brackets will denote an e-class id in our example) with the following e-nodes:

$$\begin{array}{rcl} M & = & \langle 1\rangle \mapsto \{a\} \quad \langle 2\rangle \mapsto \{b\} \quad \langle 3\rangle \mapsto \{c\} \\ & & \langle 4\rangle \mapsto \{\langle 2\rangle + \langle 3\rangle\} \; \langle 5\rangle \mapsto \{\langle 1\rangle \cdot \langle 4\rangle\} \end{array}$$

The e-graph maintains a union-find, which, in this trivial example, is a bit boring since it is an identity relation.

The contents of the hash-cons (which can be easily discerned by inverting M) are:

$$H \quad = \quad a \mapsto \langle 1\rangle \quad b \mapsto \langle 2\rangle \quad c \mapsto \langle 3\rangle \qquad \langle 2\rangle + \langle 3\rangle \mapsto \langle 4\rangle \qquad \langle 1\rangle \cdot \langle 4\rangle \mapsto \langle 5\rangle$$

An e-graph where every e-class is a singleton, like this one, is just a forest of expression trees with sharing. The situation becomes more interesting once we start mutating the graph via its dedicated operations.

1. Insert - Adding a term t to the e-graph basically means creating an e-class per AST node of t; but this can potentially create many duplicates as some (or all) sub-terms of t may already be present in the graph. For this reason, insertion is done bottom-up where at each level the hash-cons is utilized to look up an existing, compatible e-node, in which case its containing e-class id is reused; otherwise, a fresh id is created. The end result is an e-class id corresponding to the root of t.
2. Union - Merging two e-classes is accomplished by applying the respective union operation of the union-find and concatenating the node and parent lists in the e-class map. This, however, temporarily invalidates the invariant of the hash-cons and e-class map that all e-class ids and e-nodes must be canonical.
3. Rebuilding (Congruence closure) - As explained before, a union of $[x]$ into $[y]$ necessitates replacing any e-node $f([x], [z])$ by $f([y], [z])$. Moreover, if $f([x], [z]) \in [w_1], f([y], [z]) \in [w_2]$, then, following this replacement, both

$[w_1]$ and $[w_2]$ now contain $f([y],[z])$, meaning that $[w_1] = [w_2]$ and evoking a cascading union of $[w_1], [w_2]$. These updates accumulate quickly and are very costly to keep up with during rewriting. A significant contribution by egg is the concept of deferred rebuilding, where rebuilding is performed periodically. The periodic rebuilding is highly efficient and well-suited for equality saturation; however, it does not provide significant gains for tasks that require the congruence invariant to be maintained at all times.

4. E-matching - Perhaps the most interesting and algorithmically involved operation is looking up a *pattern* in the set of terms represented by the e-graph. A pattern is a term with (zero or more) *holes* represented by metavariables $?v_{1..k}$. For example, $(?v_1 + 1)\cdot?v_2$ is a pattern. Pattern lookup is important for rewriting in equality saturation. Since the metavariable holes may be filled in by any term, existing matching algorithm operate in a top-down manner, traversing the e-nodes downward via the e-class map.

Rewriting. We expand a bit about equality saturation and rewriting. In this setting, we assume a background set of symbolic *rewrite rules* (r.r.), each of the form $t \dot{\rightarrow} s$, where t and s are patterns as explained in item (4) above. A *match* θ of pattern t on the e-graph, is an assignment mapping metavariables to e-class ids. For a pattern t, $t\theta$ delineates a set of terms obtained by all the combinations of substituting each v_i with some $t_i \in \theta(v_i)$. By virtue of the congruence closure property, all the terms in $t\theta$ are represented by a single e-class, which we will denote $[t\theta]$. Applying the r.r. is done by merging the e-classes containing the terms $s\theta$ and $t\theta$. Because the term $t\theta$ might be new, it needs to also be inserted resulting in $union([s\theta], insert(t\theta))$. Repetitively applying such rewrite rules to a set of terms can be used to generate growing sets of terms that are equivalent, according to rewrite semantics, to ones in the starting set. Ideally, the set eventually *saturates*, in which case the e-graph now describes *all* the terms that are rewrite-equivalent. We point out that in many situations, the e-graph keeps growing as a result of rewrites and never gets saturated—so the number of successive rewrite iterations, or "rewrite depth", has to be bounded.

A *conditional rewrite rule* (c.r.r.) [3] is a natural extension of a r.r. that has the following form:

$$\varphi \Rightarrow t \dot{\rightarrow} s$$

Where φ is a precondition for rewriting t to s. For example, the rules for max are:

$$?x > ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?x$$
$$?x \leq ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\rightarrow} ?y$$

The semantics of a precondition φ is defined such that a term matching the pattern of φ must be unified with Boolean true in order for the rewrite to be applied.

3 Colored E-Graphs: Overall Design

This study aims to extend automatic reasoning through equality saturation, by providing direct support for simultaneous reasoning about multiple cases (or

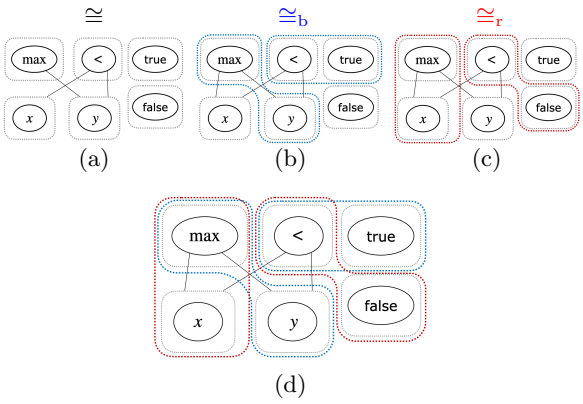

Fig. 1. Example e-graph (a), with two colored layers; (b) is blue, (c) is red, (d) shows them combined.

multiple congruence relations). The main use case for handling multiple relations in parallel arises when attempting to reason about an unknown goal in the presence of conditional expressions. Notably, existing systems such as TheSy [25] and Ruler [19] utilize equality saturation with e-graphs in an exploratory setup. In their setup, the primary objective is to discover new rewrite rules that are initially unknown at the start of the execution. In these exploratory reasoning tasks, we might want to work with additional assumptions. A good example is case splitting steps required to handle conditional expressions. For example, let $t := \mathsf{max}(x, y)$, then reasoning about the cases $x < y$ and $x \geq y$ separately is desirable: in the first case $t \cong x$, and in the second $t \cong y$. Without any assumptions, we can say neither and rewriting of t is blocked. The approach in [25] uses a prover with a case splitting mechanism that creates a *clone* of the e-graph for each such case, one for $x < y$ and one for $x \geq y$, but this runs at great expense to run-time and memory. In particular, the e-graph may consist of a large number of terms that are not related to x and y or their relative values. Such terms will be duplicated needlessly, and subsequent rewrites that apply to them will be repeated on the duplicates. In cases where further case splitting is required, e.g. $y < z$, that split must also be done on both clones, leading to a duplication factor of 4. Thus, the number of clones will grow exponentially with the number of nested case splits.

This leads us to our proposed solution—*Colored E-graphs*, whose prime directive is to avoid duplication via sharing of the common terms, thus storing them only once when possible. The e-graph structure becomes *layered*: the lowermost layer represents a congruence relation over terms that is true in all cases (represented, normally, as e-classes containing e-nodes). On top of it are layered additional congruence relations that arise from various assumptions. Going back to our example, the lowermost layer is shown in Figure 1(a), containing the terms $\mathsf{max}(x, y)$, $x < y$, true and false. Layers corresponding to assumptions $x < y$ and $x \geq y$ are shown alongside it in 1(b) and 1(c). To evoke intuition, we

associate with each layer a unique *color*, and paint their e-classes (dotted outlines, in depicted e-graphs) accordingly. Conventionally, the lowermost layer is associated with the color black. In the sequel we will use blue for $x < y$ and red for $x \geq y$ when referring to the example. In the blue layer, $(x < y) \cong_{\text{b}} \mathsf{true}$ and $\mathsf{max}(x, y) \cong_{\text{b}} y$; in the red layer, $(x < y) \cong_{\text{r}} \mathsf{false}$ and $\mathsf{max}(x, y) \cong_{\text{r}} x$. This is shown via the corresponding blue and red dotted borders. 1(d) shows a depiction where both colors are overlain on the same graph, which is a more faithful representation of the concept of colored e-graphs, although this visualization is clearly not scalable to larger graphs. In Figure 2 a larger graph can be seen that includes the terms $\mathsf{max}(x, y) - \mathsf{min}(x, y)$ and $|x - y|$, after a few more rewriting steps. An overlain graph will be quite incomprehensible in this case, so the layers are shown separately; it can be easily discerned that $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\text{b}} |x - y|$ as well as $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong_{\text{r}} |x - y|$.

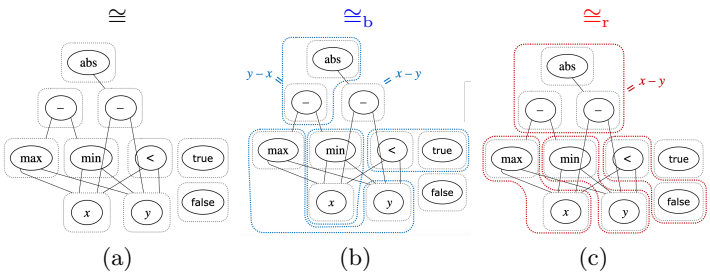

Fig. 2. Proof of $\mathsf{max}(x, y) - \mathsf{min}(x, y) = |x - y|$. The e-nodes corresponding to the two terms are in the same e-class both in the blue layer (b) and in the red (c). It is important to note that the layers are overlain, and that the black nodes are shared; they are separated here for ease of perception.

Observe that both additional layers, blue and red, utilize only the existing (black) e-nodes. Each additional color can be represented by simply specifying further unions of e-classes on top of those in the black congruence relation. We say that each color's congruence $\cong_c$ is a *coarsening* of the black congruence, $\cong$, in the simple logical sense, $\cong \subseteq \cong_c$.

Colored E-Graphs layered structure becomes significant in more complex examples. Generalizing the property $\mathsf{max}(x, y) - \mathsf{min}(x, y) \cong |x - y|$ to three variables, it becomes $\mathsf{max}(x, y, z) - \mathsf{min}(x, y, z) \cong \mathsf{max}(|x - y|, |x - z|, |y - z|)$. Reasoning about this property would require to reason about additional colors, or clones, on top of $x < y$ and $x \geq y$. The extra colors are due to the conditionals from max, min, and $|\cdot|$ leading to a total of six leaf cases as can be seen in Figure 3. The coarsening of congruence relation holds in this multi-level hierarchy as well. If we mark in blue the layer representing $(x < y) \cong_{\text{b}} \mathsf{true}$, and in green the layer representing $x < y \wedge y < z \cong_{\text{g}} \mathsf{true}$, then $\cong_{\text{b}} \subseteq \cong_{\text{g}}$. That is, any common sub-term and conclusion made in root, or one of the parent colors can be shared

to the descendants. Thus, further duplication, whether of terms or computation, can be prevented.

A crucial challenge to address is the implementation of operations (insert, union, congruence closure, e-matching) efficiently while preserving this invariant as well as the standard e-graph invariants. The colored layers require special support, as different e-classes may be united in some colored (non-black) layer but not in others (including the black relation). Notably, the black congruence relation can be implemented as a standard e-graph since all the necessary data structures are available to it.

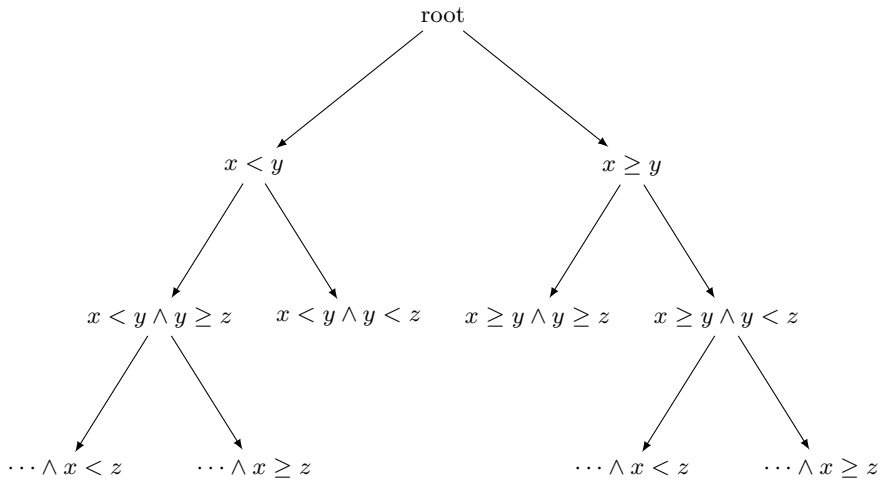

Fig. 3. All needed case split cases to reason about the property $\mathsf{max}(x, y, z) - \mathsf{min}(x, y, z) \cong \mathsf{max}(|x - y|, |x - z|, |y - z|)$

Before diving into the design of colored e-graphs, it is better to start with their expected semantics. One way to understand the semantics of colored e-graphs is by analogy to a set of clones, i.e. separate e-graphs $\mathcal{E}$; One e-graph represents the base congruence $\cong$, and one per color c represents $\cong_c$. All e-graphs in $\mathcal{E}$ conceptually represent the same terms partitioned differently into e-classes. Thus, they have the same e-nodes, except that the choice of e-class id (the representative) may be different according to the composition of the e-classes. We will call the e-classes of the color congruences *colored e-classes*. A union in any layer, black or colored, is in effect a union applied to the respective e-graph and all its descendants. Thus, a union in the black layer (i.e. the original e-graph) is analogous to a union in *all* of the e-graphs of the corresponding e-classes; this maintains the invariant that every colored e-class is a union of (one or more) black e-classes. The colored e-graph semantics of the other operations—insertion, congruence closure, and e-matching—are the same as if they were performed across all clones.

A naturally occurring situation in equality saturation and exploratory reasoning tasks is that the e-graph is extensive, and each assumption induces a relatively small number of additional congruences. Colored e-graphs are adapted to this scenario. Each color layer corresponds to a narrow assumption, such as $x < y$ in our overview example, and will therefore have some additional unions, but not drastically many. In such settings, the space reduction obtained by deduplication of the black e-nodes, outweighs the overheads associated with bookkeeping of the colored e-classes. With careful tweaks and a few optimizations, we show that we can keep these overheads quite modest.

For presentation purposes, we will describe the design and implementation of colored e-graphs in two steps. We start with a basic implementation that is not very efficient but is effective for understanding the concepts and data structures; then, we indicate some pain points, and move on to section 5 to describe optimization steps that can alleviate them.

In the basic implementation, all e-nodes reside in the "black" layer, which is represented by a "vanilla" e-graph of the e-graphs library egg, with the operations listed in section 2. The colored congruences do not have designated e-graphs of their own, and instead, the operations of union, congruence closure, and e-matching have *colored variants*, parameterized by an additional color c that are semantically analogous to the same operations having been applied, in clone semantics, to the e-graph associated with color c in $\mathcal{E}$. (For the time being, insertion does not have a colored variant, since insertions create e-nodes and all e-nodes are shared.)

Colored union-find. In an e-graph, the union-find data structure holds all the e-class ids that were ever inserted into the e-graph. Instead of having this information replicated per layer, colored e-graphs save a master copy onto which black unions are applied, and on top of that *smaller* union-finds, one per color, whose elements are only the representatives of the parent layer e-classes that have been merged.

Colored e-matching. The e-class map is only saved for the black layer. This is sufficient, because an e-class in color c is always going to be a union of black e-classes, and all that is required for e-matching is finding e-nodes with a particular root (operator) in the course of the top-down traversal. So the union can be searched on demand by collecting all the "c-color siblings" of the e-class and searching them as well.

Colored congruence closure. In egg, the e-graph will fix the congruence invariant by repeatedly going through a work list of changed classes, and for all the parents of an e-class, re-canonise them, and find unions needed to complete the congruence by finding duplicates. In colored e-graphs the root will behave the same, but for colored layers there is no single e-class, as the colored e-classes are a equality class of concrete e-classes. Therefore for each color we will maintain an additional work list, and collect the relevant concrete parents from the concrete e-classes on demand. In practice, after collecting and canonising the parents, we get the exact same rebuild algorithm as in egg, but without updating the hashcons, as there isn't one colored layers.

Example 2. We walk through the steps needed to carry out the case splitting shown in Figure 2. The system contains the conditional rewrite rules shown on the right of Figure 4, which constitute the definitions of max and min, plus some prior knowledge about $|\cdot|$ and $-$.

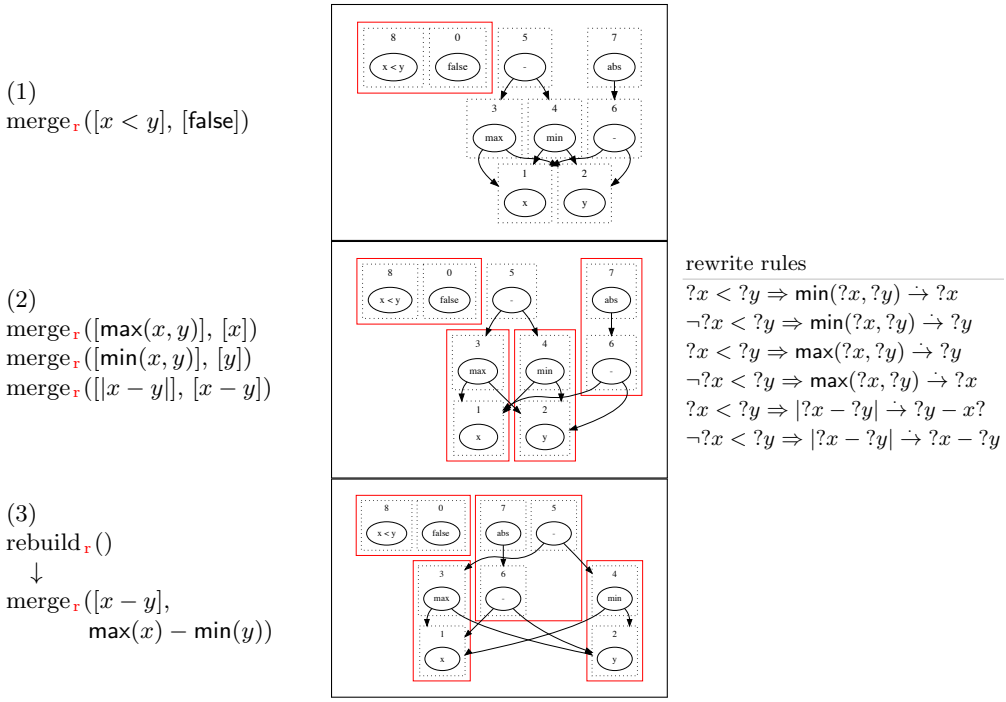

Fig. 4. Rewriting with case-split in a colored e-graph.

The semantics of a conditional rewrite rule in the domain of an e-graph is that the condition pattern should be matched and its root must be in the same e-class as true, and, additionally, the left-hand side should be matched as normal. For simplicity of presentation, we pretend that $\neg$ is a special case were the negated condition is e-matched and the e-class should contain false.

Starting with the base graph, Figure 2(a), we describe the operation of Colored Egg on the red color, corresponding to the case $\neg x < y$. The complement blue case $(x < y)$ is analogous.

1. The value of $x < y$ is declared as false via a colored merge. This yields a new red e-class.
2. Colored e-matching is performed against the premise of the c.r.r. $\neg ?x < ?y \Rightarrow \mathsf{max}(?x, ?y) \dot{\to} ?x$. The condition of the rule, $?x < ?y$, matches against the class $[x < y]$, which is indeed in the same red e-class as false.

Similar e-matches are carried out for the rules $\neg ?x < ?y \Rightarrow \mathsf{min}(?x, ?y) \dot{\to} ?y$ and $\neg ?x < ?y \Rightarrow |?x - ?y| \dot{\to} ?x - ?y$.

3. The children of $\langle 3\rangle - \langle 4\rangle$ ($\in M(\langle 5\rangle)$) are red-equivalent to those of $\langle 1\rangle - \langle 2\rangle$ ($\in M(\langle 6\rangle)$), and, as a consequence, red congruence closure kicks in and performs a red union there.

The process for blue is analogous. The case-split semantics is defined such that it records the fact that blue and red are *complements*, and as such extends $\equiv$ with the common equivalences, $\cong_{\mathrm{b}} \cap \cong_{\mathrm{r}} = \{\langle\langle 5\rangle, \langle 7\rangle\rangle, \dots\}$.

When using the above operations in the context of equality saturation, e-matching is applied for all colors to discover matches for the left-hand sides of rules. For each match, the right-hand side of the rule needs to be inserted into the e-graph and union-ed or color-union-ed with the left-hand side. Inserting the e-nodes to the e-graphs makes them available to all layers. We assume that the mere *existence* of a term in an e-graph does not in itself have the semantics of a judgement—it is only the placing e-nodes in the same e-class that asserts an equality. Thus, under this assumption, inserting all the e-nodes to the e-graph is sound. However, in the presence of many colors, and thus many colored matches, the result would be a large volume of e-nodes that are in black e-classes of size 1, because they were created just to serve one of the colors. As opposed to a standard, single e-graph where merging e-classes shrinks the space of e-nodes (because non-equal e-nodes may become equal as a result of canonization), in colored unions it is required that the e-graph maintain both original e-classes, thus losing this advantage. This can put a growing pressure on subsequent e-matching and rebuild operations *in all colors*.

4 Functional Description

We will now introduce some notations and definitions that will formalize the description of the e-graph brought in section 3.

e-class ids	E
e-nodes	$N = \{f(e_1, .., e_r) \mid f^r \in \Sigma, e_i \in E\}$
union-find	$\equiv_{\mathrm{id}} \subseteq E \times E$, $\equiv_{\mathrm{id}}$ is an equivalence relation
e-class map	$M : E \to \mathcal{P}(N)$
parent map	$P = \{e \mapsto \{(n, e') \mid e' \in E \wedge n \in M(e') \wedge n = f(\dots, e, \dots)\} \mid e \in E\}$
hashcons	$H = \{n \mapsto e \mid n \in M(e)\}$

The union-find offers an operation, $\mathit{find}(e)$, that returns a unique representative of id of the equivalence class (of $\equiv_{\mathrm{id}}$) that contains e. That is, $\mathit{find}(e) \equiv_{\mathrm{id}} e$ and for all $e_1 \equiv_{\mathrm{id}} e_2$, $\mathit{find}(e_1) = \mathit{find}(e_2)$.

We introduce a set of *colors*. As explained in section 3, colors are organized in a tree whose root is the initial color ("black"). We mark the root color $\varnothing$ and assign to every non-root color c a *parent color* $p(c)$.

$$\begin{aligned} &\text{colors} && C = \{\varnothing, \ldots\} \\ &\text{parent colors} && p : C \setminus \{\varnothing\} \to C \end{aligned}$$

The colored e-graph will now hold multiple union-find structures, one per color. They define a family of equivalence relations $\equiv_c$ by induction on the path from c to $\varnothing$.

- $\equiv_\varnothing = \equiv_{\text{id}}$; $\mathit{find}_\varnothing(e) = \mathit{find}(e)$
- $\equiv_c \subseteq E_{p(c)} \times E_{p(c)}$, where $E_{p(c)} = \{\mathit{find}_{p(c)}(e) \mid e \in E\}$ is the set of all representatives from $\equiv_{p(c)}$. $\mathit{find}_c(e)$ for $e \in E_{p(c)}$ returns a unique identifier in the normal manner of union-find, i.e., $\mathit{find}_c(e) \equiv_c e$ and for all $e_1 \equiv_c e_2$, $\mathit{find}_c(e_1) = \mathit{find}_c(e_2)$.

The definitions over $E_{p(c)}$ are naturally extended to E by (recursive) application of *find*; i.e., $\mathit{find}_c(e) = \mathit{find}_c(\mathit{find}_{p(c)}(e))$ and $e_1 \equiv_c e_2 \Leftrightarrow \mathit{find}_{p(c)}(e_1) \equiv_c \mathit{find}_{p(c)}(e_2)$. Thus it holds, by construction, that $\equiv_c \supseteq \equiv_{p(c)}$.

The colored e-graph also supports a $\mathit{merge}_c(e_1, e_2)$ operation for each color c where $e_1, e_2 \in E_c$. When performing a merge, the congruence relation invariants may be broken for c and all its descendants, and thus needs to be fixed. The merged classes are added to $\mathit{worklist}(c')$ for all c' where c' is c or its descendant. In egg [29], the invariants are restored periodically by performing a REBUILD pass. To accommodate the colors, we adjust the REBUILD logic to a multi-congruence-relation setting, so that it restores a congruence closure for each color during REBUILD. We update the auxiliary function REPAIR to work on colored e-classes, and introduce two new helper functions: COLLECT_PARENTS and UPDATE_HASHCONS, as presented in Algorithm 3. COLLECT_PARENTS extract the parents of a colored e-class by combining the sets of parents of all the (root) e-classes contained therein. UPDATE_HASHCONS is used to make sure that the hashcons entries are in canonical forms. It was already a part of REPAIR in egg; it is only repeated here to point out that it only updates the hashcons for the root color, since no canonization is required for colored layers.

Another important colored e-graph operation is e-matching. Colored e-matching is a modification of the e-matching abstract machine presented in [18]. E-matching is performed by an abstract machine M which consists of a program counter, array of registers reg, and backtracking stack bs, in combination with a sequence of instructions that represents a pattern p. The machine will go over the instructions one by one. Each instruction may either fail if its assertion is not met, or produce a set of continuation states. In the latter case, the machine picks the first state produced and pushes the current instruction onto the stack; in the former case, the machine backtracks and tries the most recently pushed state, trying the next remaining continuation.

To better present our modifications in colored egg, we first shortly introduce some of the original instruction types:

- $\mathsf{init}(f)$ — Expects an e-node representing an application $f(x_1, \ldots, x_n)$ in $reg[0]$, and pushes its children into $reg[1..n]$.
- $\mathsf{bind}(in, f, out)$ — Matches any e-node of the form $f(x_1, \ldots, x_n)$ that resides in the e-class saved in $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out+n-1]$.
- $\mathsf{compare}(i, j)$ — Asserts $reg[i] == reg[j]$.
- $\mathsf{check}(i, term)$ — Asserts that the e-class $reg[i]$ represents $term$.
- $\mathsf{continue}(f, out)$ — Match any e-node $f(x_1, \ldots, x_n)$ (in *any* e-class), storing its children $x_{1..n}$ in $reg[out..out + n - 1]$.
- $\mathsf{join}(in, reverse\_path, out)$ — Match any e-node $f(x_1, \ldots, x_n)$ that is reachable through $reverse\_path$ from the e-class $reg[in]$, storing its children $x_{1..n}$ in $reg[out..out + n - 1]$. (This is used to improve the performance of multi-patterns, see below.)

To support matching over different congruence relations, we modify the machine M to also include the current colored assumption *color* as part of its state. Respecting *color* requires modifications to the compilation and the instructions. There are two main situations where the colored assumption matters. The first is during $\mathsf{compare}(i, j)$, where we need to assert $reg[i] \equiv_{color} reg[j]$. The second case is when matching a function application of p which is represented by a bind instruction. Before each 'bind' instruction, a modified compilation procedure will insert a new '$\mathsf{colored\_jump}$' instruction. The instruction '$\mathsf{colored\_jump}(i)$' yields all the "colored siblings" of $reg[i]$ in the current *color*, replacing $reg[i]$ with the result. Thus, the machine will try matching the full colored equality class, one "root" e-class at a time. The algorithms for COLORED_JUMP(i) and the updated COMPARE(i, j) are given in Algorithm 1. The instruction 'check' can be likewise adjusted, but we point out that, in fact, it can be implemented as a sequence of 'bind's (with respective interleaved '$\mathsf{colored\_jump}$'s).

Multipatterns are originally supported in the abstract machine, allowing e-matching against a set of patterns with shared variables. This is particularly useful in the case of c.r.r.s to also match the precondition of the rule. Multipatten matching is accomplished with the '$\mathsf{continue}$' instruction, issued after each sub-pattern (except the last one), which will pick a new root for the following pattern. This instruction remains unchanged in the colored setting. However, for reasons of performance, '$\mathsf{continue}$' is sometimes replaced by 'join', which is similarly used to select a new root but limits the selection to e-nodes that can reach, via child edges in the e-graph, a given e-class corresponding to some hole that has already been filled. A *reverse path* is provided to further restrict the upward search needed to find such e-nodes. We do not go too deep into the details, but its colored variant will invoke a $\mathsf{colored\_jump}$ at every level. We point out that egg does not currently implement 'join', and our colored egg supports a special (though frequent) case in which $reverse\_path$ is empty.

5 Optimizations

In this section, we explain in more detail some important optimizations that address various challenges encountered when applying operations such as e-

matching and rebuilding in colored e-graphs. These optimizations aim to mitigate the negative impact of having e-nodes from multiple colored layers, inefficient rebuild processes, duplicate results in e-matching, and the re-adding of e-nodes. We will first explain these problems in more detail, and then offer some solutions

Rebuilding in colored e-graph, as discussed in section 3, can be significantly slower compared to a separate, minimized e-graph. The two main causes are that building a colored hash-cons (which will be presented shortly) requires going over all the e-classes, and that the colored e-graph contains duplicate e-nodes compared to a separate minimized one.

E-matching for each color may produce duplicate results due to the e-graph not being minimized according to the color's congruence relation; that is, colored-congruent terms are not always merged under a single e-class. To illustrate this, consider a simple e-graph representing the terms $1 \cdot 1$, $1 \cdot x$, $1 \cdot y$, and $x \cdot y$. Introduce a color, blue, where $x \cong_b y$. A simple pattern such as $1 \cdot ?v$ would have three matches, with assignments $?v \mapsto 1, ?v \mapsto x, ?v \mapsto y$. If the blue layer were a separate e-graph, x and y would have been in the same e-class, so one of the matches here is redundant (as far as the blue layer is concerned). Of course, in the black layer they are different matches; the point is, that many terms are added to the graph only as a result of a colored match, so matching them in the black e-graph is mostly useless to the reasoner. On the other hand, their *presence* in the black layer means they cannot ever be union-ed, leading to duplicate matches, as seen above, even in the respective colored layer(s).

When inserting e-nodes to the e-graph, the hash-cons is used to prevent duplication based on it being canonized. Adding an e-node from a colored conclusion, that is the e-matching relied on the colored congruence relation, does not benefit from canonization. In fact, each e-node $f(x_1, \ldots, x_n)$ has a multitude of equal black representatives that are $\cong_b$ -equivalent. Each child x_i in the e-node can be presented by any black id such that $e \in [x_i]_b$, so there are $\prod_i |[x_i]_b|$ representations. And, each representative may be re-added to the e-graph without blue canonization.

To address these issues, we present a series of optimizations to the colored e-graph data-structure and the procedures. These optimizations aim to reuse the "root" and ancestor layers as much as possible, in both memory use and also in compute. Thus, we can achieve a memory efficient, but effective colored e-graph.

5.1 Data-structure optimizations

Colored e-nodes. Adding e-nodes resulting from colored e-matches to the root e-graph is the source of many inefficiencies in the simple implementation as described in section 3. Instead, the optimized version has *colored e-nodes* as well; that is, e-nodes that are introduced following a colored match are associated with that color. Each colored layer includes additional specialized colored hash-cons and colored e-class map that contains colored e-nodes and colored parents. These structures should only hold the difference from the parent layer's structures, reusing as much as possible from them. The new mappings added are:

$$\begin{array}{ll} \text{e-class color map} & EC : E \to C \\ \text{colored parent map} & P_c = \{(n, e') \mid (n, e') \in P \wedge EC(e') = c\} \\ \text{colored hashcons} & H_c = \{n \mapsto e \mid n \in M(e) \wedge EC(e) = c\} \end{array}$$

Note that the colored mappings include the base parents and hashcons that were present in the non-optimized colored e-graph as $P_\varnothing$ and $H_\varnothing$.

The optimization is such that the hierarchy is used for all operations. For example, while inserting an e-node to a color c, it is looked up in the colored hash-cons for c and all its ancestors, $p^*(c)$, and finally if no match is found it is inserted into a new e-class e, setting $EC(e) = c$. An important quality of the colored hash-cons H_c is that it is canonized to its specific color c. Whenever a new colored e-node is added, it is checked against this canonical colored hash-cons to prevent duplicate insertion, thus partially overcoming the multitude of possible representations in the $\varnothing$ layer. We still might add unnecessary colored e-nodes, as the "root" layer might contain an equivalent e-node, but we will at most add each colored e-node one unnecessary time.

This optimization is especially important for e-matching, as previously e-nodes were shared across all layers. When trying to match a function application f, all f-e-nodes in N were candidates. With this separation, only an e-node n such that $\exists e.\ n \in M(e) \wedge EC(e) \in p^*(c)$ is considered.

Pruning. Recall that having only coarsening relations means that any result found in a finer relation (ancestor color) is also true for the coarsened relation(s). And so, following unions, some of the colored e-nodes could actually become subsumed by e-nodes that already exist in an ancestor layer. To overcome these redundant e-nodes, we present an efficient deferred pruning method.

Normal e-graph minimization relies on having all e-nodes canonized. A colored e-graph usually does not canonize all e-nodes to a specific color c (except for $\varnothing$). Rather, H_c contains only the difference from previous layers. To find unnecessary e-node the colored e-graph will build a transient hash-cons during rebuild from all relevant e-nodes that are not c colored. That is a new hash-cons $H'_c = \{canonoize_c(n) \mapsto find_c(e) \mid EC(e) \in p^+(c) \wedge n \in M(e)\}$ is created. A c-colored class $e, EC(e) = c$, can be reduced by removing all e-nodes that already exist in $H'c$: $\{n \mid n \in M(e) \wedge EC(e) = c \wedge n \in \text{dom } H'_c\}$. While pruning is promising, its usability is limited to cases where the colored e-node is not immediately added back.

Colored minimization. Another improvement is having multiple colored e-nodes (of the same color) in a single (black) e-class. As mentioned previously, any e-node that resulted from a colored insert had to be in their own e-classes, as no black unions would be performed on them. But, given that $e \equiv_c e' \wedge EC(e) = EC(e') = c$, then the two black e-classes e, e' can be merged as both contain colored e-nodes of the same color and are in the same colored e-class (of the same color). Thus an invariant is kept that each colored equality class has at most one black e-class containing colored e-nodes.

5.2 Procedure optimizations

Rebuild. When rebuilding, we first reconstruct the congruence relation of the "root" layer. Even though a color, for example blue, will need to rebuild its own congruence, it still holds that $\cong \subseteq \ \cong_{\text{b}}$. So, any union induced by $\cong$ can be applied to the blue relation. To understand the implications, consider the e-graph representing the terms x, y, $f(x)$, $f(y)$, $f(f(x))$, and $f(g(y))$ where the blue color contains the additional assumption that $g(y) \ \cong_{\text{b}} f(y)$. If we union x and y, the black congruence will include $f(x) \cong f(y)$ which also holds in the blue relation. But, the depth of the blue rebuilding is once more, as $g(y) \ \cong_{\text{b}} f(y)$, we need to conclude the $f(f(x)) \ \cong_{\text{b}} f(g(y))$. This demonstrates how reusing parent relations is useful; the rebuild depth required can be reduced by first rebuilding finer relations.

E-match. In e-matching a similar optimization is applied. Any conclusion of e-matching on the root layer is also applicable to any higher layer in the e-graph. Thus, during e-matching, a pattern matched against the current layer should not be repeated for a higher layer. We modify the instructions presented in Algorithm 1 to prevent such repeating. This is done by starting to e-match only from $\varnothing$, and adding colored assumptions on demand. There are two situations where a colored assumption will be added. The first is during *compare(i, j)*. If $reg[i] \not\equiv_{color} reg[j]$, then we don't necessarily need to backtrack, as there might be a descendant of *color*, c, for which $reg[i] \equiv_c reg[j]$. Therefore, for each such c the current state with the new assumption $color \leftarrow c$ is added to the backtracking stack bs. On demand "coloring" is done such that when performing colored_jump, it is also possible to jump to any color c such that $M.color \in p^+(c)$ and the resulting e-class being jumped is not reachable otherwise. The set of new new assumptions added is minimized such that no unnecessary colors are added. That is, during 'compare', if a color c is sufficient, we won't also add its descendants to bs, and for 'colored_jump', an e-class will be matched only for the topmost (closest to root) descendant(s) of *color* for which it is equal to $reg[i]$. Following minimization, all additional matching paths are unique, as at least one (different) e-class is chosen at each fork.The modified instructions are given in Algorithm 2. Although all expected matches are found, and there are no duplicate paths, duplicate colored matches will still be present due to the e-graph not being fully minimized.

6 Evaluation

In this section, we evaluate the performance and effectiveness of our colored e-graphs and the different optimizations we propose. For this purpose we implemented two versions of colored e-graphs containing different improvements described in section 5. The simple version only uses procedural improvements, while the optimized version uses all optimizations.

6.1 Objectives and Evaluation Method

The objective of our evaluation is to assess the potential usefulness of colored e-graphs for exploratory tasks, specifically focusing on their ability to efficiently run equality saturation for multiple assumption sets simultaneously. To accomplish this we designed an experiment consisting of multiple test cases, each containing a distinct set of rewrite rules and proof goals. Our evaluation metrics include measuring the size of the e-graph created, and the time taken for equality saturation. To the best of our knowledge, a purely e-graph based automated theorem prover does not exist, and theory exploration tools have limited support for conditions. Therefore, for the experiments, we constructed an equality saturation-based prover (based on the code from [25]) that incorporates an automatic case-splitting mechanism.

The case-splitting mechanism is only used when it will potentially contribute to progress in equality saturation—that is, when it enables additional rewrite rules that were previously blocked. Each case split creates additional, coarsened e-graphs. As a baseline, this prover uses a set of e-graphs (clones) to represent the different congruence relations. We then replace the underlying e-graph to use colors instead of separate graphs when performing the case splits. We then compare the time and memory consumption of these variants. To more closely simulate an exploratory reasoning scenario, we consider the accumulated size of all the clones in our comparison.

We evaluate our implementation on inductive proof test suites introduced in [23], which are also used in [25]. Since the instances are relatively small, we introduced a slight variation: typically, only the terms occurring in the proof goal are used as input to the equality saturation process. These graphs are too small to demonstrate significant exploration tasks. In our experiments, for each proof goal, we identified all other benchmarks within the same test suite that have similar goals; that is, have a common vocabulary and common rewrite rules. The prover thus explores the e-graph of all the proof terms emerging from these starting set, which gives rise to larger e-graphs and more meaningful measurements. Importantly, the prover does not stop even if it proves the goal early, but continues to explore the space until saturation or resource cap.

All the experiments were conducted on 64 core AMD EPYC 7742 processor with 512 GB RAM, ensuring consistent testing conditions for accurate comparisons and measurements.

6.2 Experimental Setup

Using the modified prover, we run each test case, collecting the e-graph size and running time. The measurement of e-graph sizes involved counting the number of e-nodes present in the e-graph. For the colored layers, we specifically measured the additional colored e-nodes in the e-graph for each layer. In contrast, for the separate set of e-graphs, we counted the e-nodes in the original e-graph as well as each of the coarsened e-graphs, and take the difference. To manage memory usage during the evaluation, we incorporate the Cap library, a Rust library specifically

designed to limit memory usage. Each test case was limited to 32 GB of memory and 1 hour run-time.

In our evaluation, we ran experiments using a basic implementation of colored e-graphs as described in section 3, but without the data-structure optimizations ("monochrome e-nodes"), as well as the fully optimized one. We compare each of them against a baseline of using separate e-graphs.

As for other forms of ablation, in our experiments, we also measured the optimized implementation without pruning. It shows results that are very similar to the version that includes pruning. This is expected because in this experimental setup most of the pruned e-nodes will be re-added by running the rewrite rules again.

By conducting experiments with these three different flavors of e-graphs with additional assumptions, we aim to compare their performance and measure their impact on e-graph size and run-time. The results of these experiments provide insights into the advantages and limitations of each approach.

6.3 Results

In our setup, all assumptions emerge from case splits done by the prover. We filter out cases where no case splits were applied, since these have no assumptions introduced and thus colored e-graphs have no impact.

For each benchmark instance, we measure the *relative e-node overhead* as the number of additional e-nodes that are required, normalized by the number of different assumptions. That is, $(|\text{total e-nodes}| - |\text{base e-nodes}|)/|\text{assumptions}|$. "Base e-nodes" represent the contents of the graph before case splits. (For the colored e-graph with monochrome e-nodes we use the base e-nodes present in the separate e-graphs case.) Figure 5 summarizes the results, pitting colored e-graphs (with and without colored e-nodes) against the baseline of separate clones. In some cases one configuration times out or runs out of memory, while the other does not; we only compare cases where both configurations finished the run successfully. In both comparisons, we see roughly around $10\times$ lower overheads, where in the monochromatic case samples are more dispersed around the y axis, and the optimized case shows clear advantage to the colored e-graph implementation.

Run-time is measured as the the total run-time for completed test cases, and 1 hour for cases that timed out. We do not include runs that did not finish due to out-of-memory exceptions (we report the latter separately). As can be seen in Figure 6, the monochrome e-nodes lead to many timeouts, whereas the optimized case exhibits running times similar to separate clones. This is in line with our expectation: colors provide lower memory sizes at the expense of run-time.

Finally, in Table 1 we present the number of out-of-memory exceptions, the number of timeout exceptions, and total run-time for each configurations and test suite. The monochrome e-graph, as expected, exhibits many timeouts. Even though it has more errors than the other e-graph versions, it still has much longer run-times.

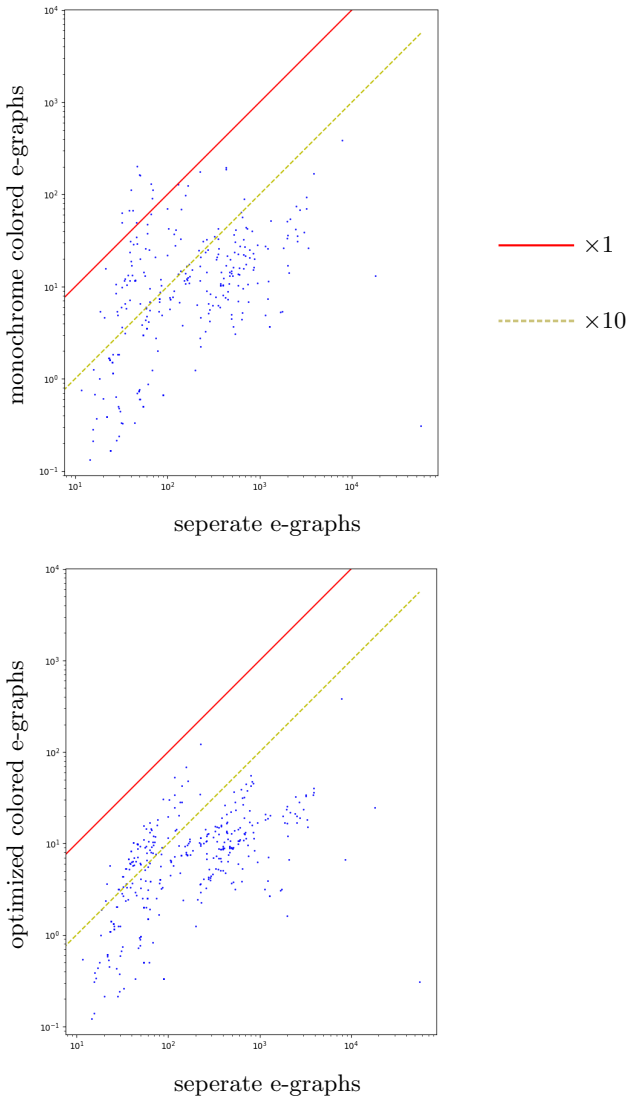

Fig. 5. Size comparison: relative e-node overhead in clones *vs.* color e-graph variants.

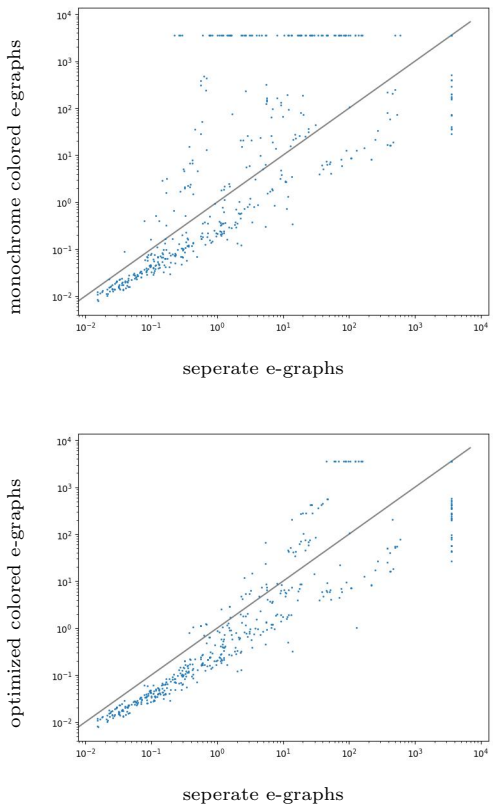

Fig. 6. Run-time comparison: run-time of clones vs. color e-graphs

Table 1. Run-time and exceptions for each test suite

Type	Test Suite	Run-time (seconds)	OOM	Timeout
Seperate	clam	70.1	0	0
	hipspec-rev-equiv	34.1	0	0
	hipspec-rotate	3880.3	1	1
	isaplanner	8454.4	0	60
	leon-amortize-queue	187356.4	52	0
	leon-heap	1735.9	0	0
Monochrome	clam	277.8	0	5
	hipspec-rev-equiv	139.0	0	17
	hipspec-rotate	1871.4	0	6
	isaplanner	6068.4	0	70
	leon-amortize-queue	14.8	0	57
	leon-heap	1201.8	0	25
Optimized	clam	23.6	0	0
	hipspec-rev-equiv	57.0	0	0
	hipspec-rotate	17.4	0	3
	isaplanner	20486.3	3	28
	leon-amortize-queue	10854.3	3	49
	leon-heap	4949.2	0	13

The results for the optimized e-graph show an improvement over those of the separate e-graphs, both in run-time and number of failed runs (as is shown in Table 1). The optimized section managed to complete more of the test cases than the baseline (with 99 failures, instead of 114). But the significant difference is that errors that used to be out-of-memory in the separate e-graphs rubric are replaced by timeouts, which is especially visible in the leon-amortize-queue test suite. The leon-heap suite is the worse one for colored e-graphs, with 13 additional timeouts even for the optimized version. The isaplanner suite is the most favorable, with half as many failures in the optimized version than in the baseline.

7 Related Work

Theory exploration and its applications. Interest in exploratory reasoning in the context of functional calculi started with IsaCoSy [12], a system for lemma discovery based in part on CEGIS [27]. In a seminal paper, QuickSpec [26] propelled applicability of such reasoning for inferring specifications from implementations based on random testing, with deductive reasoning to verify generated conjectures [6,11]. TheSy [25] and Ruler [19] have both incorporated e-graphs to some extent in the exploration process: they are used to speed up equivalence reduction of the space of generated terms, and, in [25], also the filtering and qualification phases using symbolic examples. The evaluation of the latter

shows quite clearly that case splitting is a major obstacle to symbolic exploratory reasoning, due to the large number of different cases and derived assumptions.

In the area of conditional rewrite discovery, Speculate [4] naturally builds on the techniques from QuickSpec and depends on property-based testing techniques to generate inputs that satisfy some conditions. SWAPPER [24] is a relatively early example of exploring using SyGuS with a data-driven inductive-synthesis approach with emphasis on finding rules that are most efficient for different problem domains. It requires a large corpus of similar SMT problems to operate.

Other e-graph extensions. E-graphs were originally brought into use for automated theorem proving [8], and were later popularized as a mechanism for implementing low-level compiler optimizations [28], by extending them with "φ-nodes" to express loops. Relational e-matching [31] makes use of Datalog seminaïve evaluation to harness the power of query planning in database systems. Subsequently, Datalog-powered e-matching has been recently fused with core Datalog semantics to allow richer logic programming by exposing equality saturation as a building block in a framework called egglog [30]. Since Datalog is based on Horn clauses, this meshes very well with conditional rewriting. It should be noted, though, that it is still a monotone framework, and does not allow backtracking or simultaneous exploration of alternative assumptions.

ECTAs [14,10] are another, related compact data structure that extends e-graphs, Version-Space Algebras [16,17], and Finite Tree Automata [1], with the concept of "entanglement"; that is, some choices of terms from e-classes may depend on choices done in other e-classes. Since the backbone of ECTAs is quite similar to an e-graph, the colors extension is applicable to this domain as well.

Uses of e-graphs in SMT. E-graphs are a core component for equality reasoning in SMT solvers [7,2], in most theory solvers such as QF_UF, linear algebra, and bit-vectors. E-matching is also used for quantifier instantiation [20], which is, in its essence, an exploratory task and requires efficient methods [18]. In these contexts, implications and other Boolean structures are treated by the SAT core (in CDCL(T)), and the theory solver only handles conjunctions of literals.

8 Conclusion

In conclusion, this paper has introduced the concept of colored e-graphs as a memory-efficient method for maintaining multiple congruence relations in a single e-graph. It provides support for equality saturation with additional assumptions over e-graphs, thereby enabling efficient exploratory reasoning of multiple assumptions simultaneously. The development of several optimizations based on the egg library and deferred rebuilding, and subsequent evaluation has validated our approach, demonstrating a significant improvement in memory utilization and a modest one for run-time performance compared to the baseline.

This work thus serves as a stepping stone, advancing the current state of the art and setting a foundation for works on exploratory reasoning tools and

techniques. By extending e-graph capabilities, we hope to drive new innovation in the realm of symbolic reasoning and its applications.

References

1. Adams, M.D., Might, M.: Restricting grammars with tree automata. Proc. ACM Program. Lang. **1**(OOPSLA), 82:1–82:25 (2017). https://doi.org/10.1145/3133906, https://doi.org/10.1145/3133906
2. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24, https://doi.org/10.1007/978-3-030-99524-9_24
3. Bergstra, J., Klop, J.: Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences **32**(3), 323–362 (1986). https://doi.org/https://doi.org/10.1016/0022-0000(86)90033-4, https://www.sciencedirect.com/science/article/pii/0022000086900334
4. Braquehais, R., Runciman, C.: Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In: Diatchki, I.S. (ed.) Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017. pp. 40–51. ACM (2017). https://doi.org/10.1145/3122955.3122961, https://doi.org/10.1145/3122955.3122961
5. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7705, pp. 350–367. Springer (2012). https://doi.org/10.1007/978-3-642-35182-2_25, https://doi.org/10.1007/978-3-642-35182-2_25
6. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: International Conference on Automated Deduction. pp. 392–406. Springer (2013)
7. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102, https://doi.org/10.1145/1066100.1066102
9. Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 75–83. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13
10. Gissurarson, M.P., Roque, D., Koppel, J.: Spectacular: Finding laws from 25 trillion programs. In: ICST. vol. 6. Association for Computing Machinery, New York, NY, USA (2023)

11. Johansson, M.: Automated theory exploration for interactive theorem proving: - an introduction to the hipster system. In: Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. pp. 1–11 (2017). https://doi.org/10.1007/978-3-319-66107-0_1, https://doi.org/10.1007/978-3-319-66107-0_1
12. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. Journal of Automated Reasoning **47**, 251–289 (2010)
13. Jones, E., Ong, C.H.L., Ramsay, S.: Cycleq: an efficient basis for cyclic equational reasoning. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 395–409 (2022)
14. Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N.: Searching entangled program spaces. Proc. ACM Program. Lang. **6**(ICFP) (aug 2022). https://doi.org/10.1145/3547622, https://doi.org/10.1145/3547622
15. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)
16. Lau, T.A., Domingos, P.M., Weld, D.S.: Version space algebra and its application to programming by demonstration. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000. pp. 527–534. Morgan Kaufmann (2000)
17. Lau, T.A., Wolfman, S.A., Domingos, P.M., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1-2), 111–156 (2003). https://doi.org/10.1023/A:1025671410623, https://doi.org/10.1023/A:1025671410623
18. de Moura, L.M., Bjørner, N.S.: Efficient e-matching for SMT solvers. In: Pfenning, F. (ed.) Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007). https://doi.org/10.1007/978-3-540-73595-3_13, https://doi.org/10.1007/978-3-540-73595-3_13
19. Nandi, C., Willsey, M., Zhu, A., Wang, Y.R., Saiki, B., Anderson, A., Schulz, A., Grossman, D., Tatlock, Z.: Rewrite rule inference using equality saturation. Proc. ACM Program. Lang. **5**(OOPSLA), 1–28 (2021). https://doi.org/10.1145/3485496, https://doi.org/10.1145/3485496
20. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Syntax-guided quantifier instantiation. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 145–163. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_8, https://doi.org/10.1007/978-3-030-72013-1_8
21. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) 22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022. pp. 65–74. IEEE (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12, https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_12
22. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. ACM SIGPLAN Notices **50**(6), 1–11 (2015)

23. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 80–98. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
24. Singh, R., Solar-Lezama, A.: SWAPPER: A framework for automatic generation of formula simplifiers based on conditional rewrite rules. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016. pp. 185–192. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886678, https://doi.org/10.1109/FMCAD.2016.7886678
25. Singher, E., Itzhaky, S.: Theory exploration powered by deductive synthesis. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 125–148. Springer (2021)
26. Smallbone, N., Johansson, M., Claessen, K., Algehed, M.: Quick specifications for the busy programmer. J. Funct. Program. **27**, e18 (2017). https://doi.org/10.1017/S0956796817000090, https://doi.org/10.1017/S0956796817000090
27. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415 (2006). https://doi.org/10.1145/1168857.1168907
28. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: A new approach to optimization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 264–276. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480915, https://doi.org/10.1145/1480881.1480915
29. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304
30. Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better together: Unifying datalog and equality saturation. In: PLDI '23: 44rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2023). https://doi.org/10.48550/arXiv.2304.04332, https://doi.org/10.48550/arXiv.2304.04332
31. Zhang, Y., Wang, Y.R., Willsey, M., Tatlock, Z.: Relational e-matching. Proc. ACM Program. Lang. **6**(POPL), 1–22 (2022). https://doi.org/10.1145/3498696, https://doi.org/10.1145/3498696

A Algorithms Pseudo Code

Algorithm 1 Instructions: compare and colored_jump

```
1: function COMPARE(i, j)
2:     if find(color, reg[i]) ≠ find(color, reg[j]) then
3:         backtrack
4:     end if
5: end function
6:
7: function COLORED_JUMP(i)
8:     siblings ← {e|e ∈ E ∧ e ≡_color eclass}
9:     for sibling in siblings do
10:        reg[i] = sibling
11:        bs.push(current_state)
12:    end for
13:    backtrack
14: end function
```

Algorithm 2 Instructions: optimized compare and colored_jump

```
1: function COMPARE(i, j)
2:     if find(color, reg[i]) ≠ find(color, reg[j]) then
3:         descendants ← {c | color ∈ p⁺(c) ∧ reg[i] ≡_c reg[j]}
4:         minimal ← {c | c ∈ descendants ∧ ¬∃c′ ∈ descendants. c′ ∈ p⁺(c)}
5:         for c in minimal do
6:             color = c
7:             bs.push(current_state)
8:         end for
9:         backtrack
10:    end if
11: end function
12:
13: function COLORED_JUMP(i)
14:    siblings ← {e | e ∈ E ∧ e ≡_color eclass}
15:    for sibling in siblings do
16:        reg[i] = sibling
17:        bs.push(current_state)
18:    end for
19:    descendants ← {(c, e) | color ∈ p⁺(c) ∧ reg[i] ≡_c e ∧ e ∉ siblings}
20:    minimal ← {(c, e) | (c, e) ∈ descendants ∧ ¬∃(c′, e′) ∈ descendants.(c′ ∈ p⁺(c) ∧
       e′ ≡′_c e)}
21:    for (c, e) in minimal do
22:        color = c
23:        reg[i] = e
24:        bs.push(current_state)
25:    end for
26:    backtrack
27: end function
```

Algorithm 3 Colored Rebuilding

```
1: function REBUILD
2:     for color in self.colors do
3:         while self.worklist(color).len() > 0 do
               ▷ empty the worklist into a local variable
4:             todo ← TAKE(self.worklist(color))
               ▷ canonicalize and deduplicate the eclass refs to save calls to repair
5:             todo ← {self.find(color, eclass) | eclass ∈ todo}
6:             for each eclass in todo do
7:                 SELF.REPAIR(color, eclass)
8:             end for
9:         end while
10:    end for
11: end function
12:
13: function REPAIR(color, eclass)
14:    parents ← COLLECT_PARENTS(color, eclass)
15:    UPDATE_HASHCONS(color, parents)
       ▷ deduplicate the parents; note that equal parents get merged and put on the
   worklist
16:    new_parents ← {}
17:    for each (p_node, p_eclass) in parents do
18:        p_node ← self.canonicalize(color, p_node)
19:        if p_node is in new_parents then
20:            self.merge(color, p_eclass, new_parents[p_node])
21:            new_parents[p_node] ← self.find(color, p_eclass)
22:        end if
23:    end for
24:    if color = ∅ then
25:        eclass.parents ← new_parents
26:    end if
27: end function
```

Algorithm 4 Colored Rebuilding (auxiliary methods)

```
1: function UPDATE_HASHCONS(color, parents)
2:     if color = ∅ then
3:         for each (p_node, p_eclass) in parents do
4:             self.hashcons.remove(p_node)
5:             p_node ← self.canonicalize(color, p_node)
6:             self.hashcons[p_node] ← self.find(color, p_eclass)
7:         end for
8:     end if
9: end function
10:
11: function COLLECT_PARENTS(color, eclass)
12:     all_parents ← ∅                          ▷ Initialize an empty set for parents
13:     relevant_eclasses ← {e | e ∈ E ∧ e ≡_color eclass}
14:     for e in relevant_eclasses do
15:         all_parents ← all_parents ∪ e.parents          ▷ Add parents of e to the set
16:     end for
17:     return all_parents
18: end function
```