Associativity and Commutativity in Equality Saturation

TARIK ROSIN, Saarland University, Germany
MARCEL ULLRICH, Saarland University, Germany
SEBASTIAN HACK, Saarland University, Germany

Equality saturation is a promising technique for program optimization which sidesteps the phase ordering problem. However, current e-graph implementations grow exponentially large, even for simple examples. Many practical applications involve associative and commutative (AC) operators. We present an extension of relational e-matching that handles AC operators natively by storing terms as multisets. Preliminary results show that equality saturation modulo AC uses asymptotically less memory in certain cases.

Additional Key Words and Phrases: E-Graph, Equality Saturation, E-matching

1 Motivation

In this paper we study the effects of associativity and commutativity (AC) rules on equality saturation. Equality saturation is a technique of exploring equivalent terms by rewriting, while storing deduplicated terms in an e-graph.

When used for program optimization, equality saturation allows for good flexibility. It does not require an ordering on terms or another kind of canonicalization strategy. Rules can be bidirectional without the danger of non-productive cyclic rewriting. This is because the e-graph acts as a compact database of all explored programs. Equality saturation can be interrupted at any time to extract the best program discovered so far.

However, e-graphs tend to explode in size, even for small programs and few rewrite rules. The e-graph is explicitly storing the entire search space. Two prominent rewrite rules, *associativity* and *commutativity*, are known to cause combinatorial blowup.

$$(x + y) + z = x + (y + z) \tag{A}$$

$$x + y = y + x \tag{C}$$

AC is ubiquitous in applications of equality saturation such as hardware synthesis [Coward et al. 2024; Wang et al. 2023], program optimization [Nandi et al. 2021; Panchekha et al. 2015], and theorem proving [de Moura and Bjørner 2007].

Example 1.1. Consider only the associative law and $x \cdot 0 = 0$ with the initial term $a \cdot 0$. First, equality saturation determines that $a \cdot 0 = 0$. In the next iteration, the associative pattern would match on $a \cdot (a \cdot 0)$ by backwards substituting $a \cdot 0$ for 0 in the initial term. This produces the new term $(a \cdot a) \cdot 0$ and the same process repeats indefinitely.

This paper reports on an extension of e-graphs to mitigate the blowup caused by AC rules. We first present bounds on the AC blow-up of standard e-graphs. We then show how to extend the database view of e-graphs to support efficient e-matching and leverage an existing algorithm for AC congruence closure to support efficient rebuilding.

1.1 Bounds

In traditional equality saturation the term $x_1 + x_2 + \ldots + x_n$ produces a saturated e-graph with exponentially many e-nodes and e-classes. In this section, we give bounds on the number of e-nodes and e-classes in the presence of AC rules.

Authors' Contact Information: Tarik Rosin, taro00001@uni-saarland.de, Saarland University, Saarbrücken, Germany; Marcel Ullrich, ullrich@cs.uni-saarland.de, Saarland University, Saarbrücken, Germany; Sebastian Hack, hack@uni-saarland.de, Saarland University, Saarbrücken, Germany.

Theorem 1.2. *For a root term $t_r = x_1 + x_2 + \cdots + x_n$, the number of e-classes in an AC-saturated e-graph is given by the function $2^n - 1$.*

Proof. Each equivalence class represents a single variable or a sum of distinct variables, regardless of order and brackets. A simple counting argument is that each e-class corresponds to a non-empty subset of variables and hence $|\mathcal{P}(\Sigma_n) \setminus \{\emptyset\}| = 2^n - 1$. □

Theorem 1.3. *For a root term $t_r = x_1 + x_2 + \cdots + x_n$, the number of e-nodes in an AC-saturated e-graph is given by the function $3^n - 2^{n+1} + n + 1$.*

Proof. Consider an arbitrary non-leaf e-node $t_k = c_1 + c_2$ which sums together $k \geqslant 2$ variables. Observe that the chosen variables can be distributed among c_1, c_2 in precisely $2^k - 2$ many ways. Thus, we can quantify the total number of e-nodes by counting each set of variables multiplied by the number of possible e-nodes.

$$\sum_{k=2}^{n} \binom{n}{k}(2^k - 2) = 1 + \sum_{k=0}^{n} \binom{n}{k} 2^k 1^{n-k} - 2 \cdot 2^n = 3^n - 2^{n+1} + 1$$

Adding the leaf e-nodes for the n variables yields $3^n - 2^{n+1} + n + 1$. □

2 Approach

Our algorithm builds on previous work by Zhang et al. [2022], which introduced relational e-matching. The modification we propose is adding specific indexed relations for AC operators. Before discussing our approach, we briefly recap the framework of existing relational e-matching approaches.

2.1 Relational E-Matching

In relational e-matching, the e-graph is a database with one relation per operator and the rewrite patterns are translated to conjunctive queries. We use the *generic join* algorithm due to its simplicity.

Example 2.1. Take the pattern $x + (-x)$ as an example. The pattern is lowered to the conjunctive query $Q(r, x) = R_{\text{add}}(r, x, y) \wedge R_{\text{neg}}(y, x)$. The first argument is always the e-class id, and the remaining arguments define the e-node. The variable r is the patterns root, the e-class id of the matched term. The variable y is the e-class id of $(-x)$ and was added during lowering. For an arbitrary variable ordering, we can compile the query to a concrete algorithm.

Algorithm 1: Generic Join for the pattern $x + (-x)$ with variable ordering $x < y < r$

```
X ← π₂R_add ∩ π₂R_neg            // Collect all candidates of x by intersecting the two relations
for x ∈ X do
    Y ← π₃σ_{2=x}R_add ∩ π₁σ_{2=x}R_neg            // Select x and then collect all y's given x
    for y ∈ Y do
        R ← π₁σ_{3=y}σ_{2=x}R_add                   // Collect r given x and y
        for r ∈ R do
            yield (r, x)                  // All candidates survived, so we output this match
```

The generic join algorithm proceeds one variable at a time and determines all values this variable can take. Next, it branches by fixing the variable to a concrete value and moves on to the next variable. In order to make this procedure efficient, trie indices are built before the matching phase.

```
pub trait TrieCursor {
    // advance the cursor down along the `key` edge
    fn select(&mut self, key: Id);
    fn unselect(&mut self);

    // tests whether the current level has `key`
    fn contains(&self, key: Id);

    // returns an iterator over all keys on the current level
    fn keys(&self) -> Iterator<Id>;
}
```

These four functions are sufficient to implement an index that supports generic join. If all functions run in worst-case logarithmic time, the resulting join algorithm is considered worst-case optimal. Our novelty lies in having a separate index structure for AC operators.

2.2 E-Graphs modulo AC

In equality saturation modulo AC, we do not run the associative and commutative patterns during the e-matching phase. Instead, these patterns are structural properties of AC relations.

An AC relation is a list of tuples, where tuples are allowed to have variadic arity. Each tuple stores an e-class id and a multiset of arguments of a term, e.g. $+(\{x_1, x_2, x_3\}) \in c$. When inserting a term into the e-graph, AC operators are flattened into a multiset. Two terms that are AC-equivalent now share the same multiset e-node.

We introduce two more concepts: *explicit* and *implicit* representation. An e-node is explicitly represented if the e-node is stored as a tuple in one of the relations. In contrast, an e-node may be implicitly represented if it is a subterm of an explicitly represented e-node. The distinction is useful because it allows us to only store a fraction of the total e-nodes.

We now describe how to implement the trie cursor interface of such relations. An incomplete but efficient scheme is given by using the multiset operations to implement the interface. Testing containment is the same as testing whether the multiset contains the key. Iterating over the keys becomes iterating over the unique elements of the multiset. Selecting a certain key is removing said key once from the multiset and reinserting it for the unselect operation. To support this kind of indexing, we need to focus on a single multiset at a time. Therefore, we add an auxiliary variable t to each AC operator when compiling the pattern. The variable t selects a multiset and is always first in the variable ordering.

Example 2.2. The pattern $x + (-x)$ will now be compiled to $Q(r, x) = R_{\text{add}}(r, x, y, t) \wedge R_{\text{neg}}(y, x)$. Here t ranges over all multisets in the relation and we enforce it to be first in the variable ordering.

This approach is not complete because each variable can only be bound to an e-class id but not to a subset of e-class ids. A counterexample would be trying to match the above pattern on $a + b + -(a + b)$. When the outer sum is flattened, the pattern variable x may not be bound to $a + b$. Further research is required to see how significant the loss of completeness is.

2.3 Rebuilding

Rebuilding refers to restoring key invariants of the E-Graph. The most significant such invariant is *congruence closure* $a = b \implies f(a) = f(b)$, meaning two terms are equal if their constituent terms are equal.

Algorithm 2: Generic Join for the pattern $x + (-x)$ with variable ordering $t < x < y < r$

```
T ← π_4 R_add                          // t selects a multiset to simplify matching in the inner loops
for t ∈ T do
    X ← π_2 σ_{4=t} R_add ∩ π_2 R_neg             // R_add now has a fixed multiset to search in
    for y ∈ X do
        Y ← π_3 σ_{2=x} σ_{4=t} R_add ∩ π_1 σ_{2=x} R_neg
        for x ∈ Y do
            R ← π_1 σ_{3=y} σ_{2=x} σ_{4=t} R_add
            for r ∈ R do
                yield (r, x)
```

In equality saturation we might discover that $a = b$ via a rewrite rule. The process of rebuilding then traverses the relations to find parents, which are now also equal. There is a long history of congruence closure algorithms and in particular AC congruence closure algorithms. In the literature these are often presented as completion-like procedures that deal with sets of ground equations. We can understand the e-graph as a set of ground rewrite rules by viewing each e-node as a term that rewrites to its e-class id. For example, let $f(a, b, c) \in u$ be an e-node contained in the e-class u. In the completion-based presentation, this is indicated by a rule $f(a, b, c) \to u$ where e-class ids are additional constants distinct from the signature.

In light of this, two terms are proven equivalent by the e-graph if they rewrite to the same canonical constant. The task of rebuilding is to restore convergence of the set of ground rewrite rules that constitute the e-graph. This setting extends to AC symbols, namely by rewriting modulo AC.

We use the modular AC congruence closure algorithm by Kapur [2021]. The algorithm is modular in the sense that each relation is being rebuilt separately. If one relation finds a new equivalence during rebuilding, it updates the union-find data structure, causing the other relations to rebuild with the new equivalence. For a single AC relation, the algorithm works as follows:

(1) For an AC relation, we mark all e-nodes in the relation as dirty.
(2) Take an arbitrary dirty e-node $f(M) \to c$ and canonicalize it using the union-find.
(3) Next, simplify $f(M) \to c$ using any clean e-nodes.
(4) Collect all critical pairs between $f(M) \to c$ and any clean e-node. Non-trivial critical pairs create new e-nodes which get added to the relation and marked as dirty.
(5) Mark $f(M) \to c$ as clean and reduce all clean e-nodes with it.
(6) If no more e-nodes are dirty we are done, otherwise repeat from step 2.

Kapur's rebuilding algorithm produces a reduced convergent rewrite system. The e-graph will contain only minimal e-nodes such that all terms are still represented.

3 Preliminary Results

We evaluate how equality saturation modulo AC scales compared to egg [Willsey et al. 2021] as baseline implementation. We insert a term into an empty e-graph and measure the size after saturating. The theory is an abelian group extended with a homomorphism $h(ab) = h(a) \cdot h(b)$. For equality saturation modulo AC we only count the explicitly represented e-nodes, because implicitly represented e-nodes require no memory. Figure 1 displays the number of e-nodes and e-classes on small instances. Table 1 summarizes the asymptotic scaling behaviour of equality saturation modulo AC. Our results demonstrate a significant reduction in the size of an e-graph compared to standard equality saturation.

Fig. 1. E-graph sizes for selected terms comparing egg and eqsat-mod-ac.

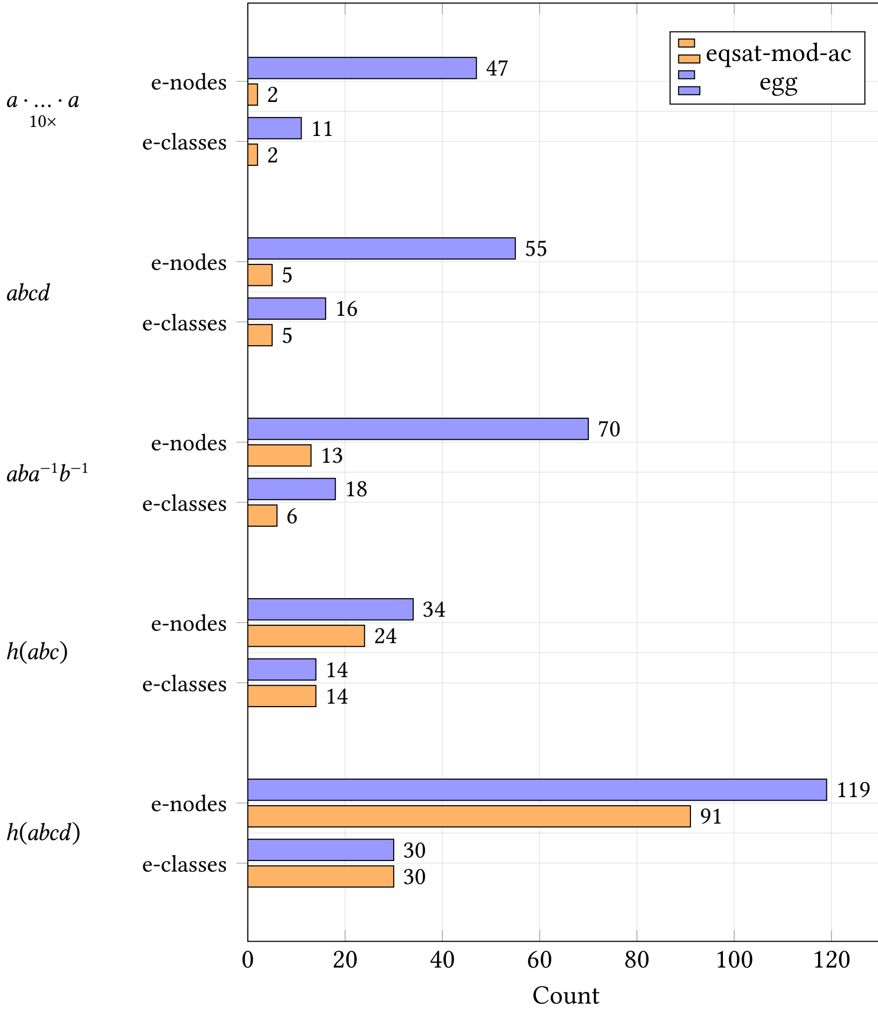

Table 1. Asymptotic e-graph size bounds for selected terms

	egg		eqsat-mod-ac		
Term	e-nodes	e-classes	e-nodes	e-classes	Theory
$x_1 + x_1 + \ldots + x_1$	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	AC
$x_1 + x_2 + \ldots + x_n$	$O(2^n)$	$O(2^n)$	$O(n)$	$O(n)$	AC
$h(x_1 + x_2 + \ldots + x_n)$	$O(2^n)$	$O(2^n)$	$O(2^n)$	$O(2^n)$	AC, homomorphism
$x \cdot 0$	∞	∞	$O(1)$	$O(1)$	AC, $x \cdot 0 = 0$

4 Conclusions

Equality saturation is a powerful method for equational reasoning, but its practical adoption is limited due to its exponential blow-up. Our investigation quantifies the effect of AC on classic equality saturation, formally proving $O(2^n)$ bounds for both e-nodes and e-classes.

By extending relational e-matching with specific indexed relations for AC operators, we can store terms as multisets and manage their subterms implicitly. Our approach outperforms traditional e-graphs, using asymptotically less memory. We observed that the memory savings are theory-dependent. In the worst case, our approach still requires an exponential amount of memory.

Further research is required to better understand the border at which our approach breaks down and to find remedies such that e-graphs become scalable.

References

Samuel Coward, Theo Drane, and George A. Constantinides. 2024. ROVER: RTL Optimization via Verified E-Graph Rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 12 (2024), 4687–4700. https://doi.org/10.1109/TCAD.2024.3410154

Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*, Frank Pfenning (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–198.

Deepak Kapur. 2021. A Modular Associative Commutative (AC) Congruence Closure Algorithm. In *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:21. https://doi.org/10.4230/LIPIcs.FSCD.2021.15

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (Oct. 2021), 28 pages. https://doi.org/10.1145/3485496

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

Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. 2023. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3582016.3582032

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (Jan. 2022), 22 pages. https://doi.org/10.1145/3498696

Received 17 April 2026