Faster and Worst-Case Optimal E-matching via Reduction to Conjunctive Queries

Yihong Zhang*
yz489@cs.washington.edu
University of Washington
Seattle, WA, USA

1 PROBLEM AND MOTIVATION

An e-graph is a data structure that efficiently represents sets of congruent terms [6]. It has a long history of applications in automated theorem proving [2, 4, 6, 8, 11] and is re-purposed in the last decade for program optimization, known as equality saturation [5, 9, 10, 12–14]. A fundamental query operation on e-graph is e-matching [2, 4], which finds the set of terms in the e-graph matching a given pattern. E-matching is the bottleneck of many equality saturation–based program optimizers [9, 13, 14] and is a central procedure in state-of-the-art SMT solvers, including Z3 [3] and CVC4 [1]. Therefore, the performance of e-matching is critical. Several optimizations are proposed for e-matching [2, 4]. However, to our knowledge, existing e-matching algorithms are based on naïve backtracking, which is suboptimal in many cases. For example, they do not exploit the equality constraints implied by multi-occurrences of a variable, which is nonetheless common in practice.

To tackle this inefficiency, we propose to take a relational view of e-graphs. Under this view, e-matching corresponds naturally to a restricted class of relational queries, known as conjunctive query (CQ). By treating e-matching as an instance of CQ, we benefit from decades of research from the database community. In particular, by using the recently discovered generic join algorithm [7], our e-matching algorithm guarantees worst-case optimality and achieves more than 400× speed-up in our preliminary experiments.

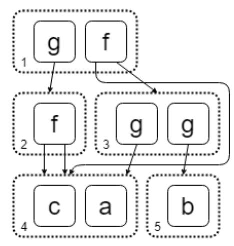
2 BACKGROUND AND RELATED WORK

We define the e-matching problem and briefly review related works in this section.

Let Σ be a set of function symbols and V be a set of variables. $T(\Sigma, V)$ is the set of terms inductively constructed using symbols from Σ and variables from V. Particularly, a *ground term* is a term that contains no variables, a non-ground term is called a *pattern*, and an f*-application* term is one whose top-level function symbol is f.

A *congruence relation* $\cong$ is an equivalence relation over ground terms where $f(t_1, \ldots, t_n) \cong f(t'_1, \ldots, t'_n)$ whenever $t_i \cong t'_i$. An e-graph E efficiently represents sets of terms in a congruence relation. It consists of a set of e-classes. Each e-class consists of a set of e-nodes, and each e-node consists of an n-ary function symbol f and n children e-classes. Similar to terms, an f*-application* e-node is an e-node associated with function symbol f.

An e-node $f(c_1, \ldots, c_n)$ is said to *represent* a term $f(t_1, \ldots, t_n)$ if e-class c_i represents term t_i. An e-class c represents a term t if an e-node a in c represents term t. Terms represented by the same e-class are congruent. A *substitution* σ is a function that maps variables to e-classes. Given a pattern p, We use $\sigma(p)$ to denote the set of terms obtained by replacing every occurrence of variable v_i in p with terms in $\sigma(v_i)$. Given an e-graph and a pattern p, e-matching is the task of finding the set of pairs (σ, r) such that every term in $\sigma(p)$ is represented in the "root" e-class r. Terms in $\sigma(p)$ are said to be matched by pattern p.



(a)

eclass-id	child_1	child_2
1	4	3
2	4	4

(b)

eclass-id	child_1
1	2
3	4
3	5

(c)

Figure 1: (a) an e-graph over $T(\Sigma, \emptyset)$ and $\cong_\Sigma$ where $\Sigma = \{f, g, a, b, c\}$ and a, b, c are nullary functions. Each solid box denotes an e-node and each dashed box denotes an e-class. Every term represented by an e-class is mutually equivalent. For example, $a \cong_\Sigma c$, $g(a) \cong_\Sigma g(b)$, and $f(a, g(a)) \cong_\Sigma g(f(a, a))$. The labels of e-classes are at bottom left. (b) relation representing of f. (c) relation representing of g.

For instance in Figure 1a, pattern $f(?\alpha, g(?\alpha))$[1] matches four terms in e-class c_1: $f(a, g(a))$, $f(a, g(c))$, $f(c, g(c))$, and $f(c, g(a))$; all of which are witnessed by the substitution $\{?\alpha \mapsto c_4\}$.

Existing approaches to e-matching rely on backtracking [2, 4, 14]. For example, de Moura and Bjørner [2] proposed a backtracking-based e-matching algorithm that is used by Z3 [3] and egg [14], two state-of-the-art e-graph implementations. To match the pattern $f(?\alpha, g(?\alpha))$ on the e-graph in Figure 1a, their algorithm does a depth-first search over the e-graph: it searches for all f-application e-nodes n_f, adds $?\alpha \mapsto n_f.child_1$ to substitution σ, iterates through all g-application e-nodes n_g in e-class $n_f.child_2$, and only yield σ if $n_g.child_1 = \sigma(?\alpha)$. In a large e-graph, there may be thousands of pairs of n_f and n_g where n_g is in e-class $n_f.child_2$, but only a few satisfy the constraint $n_f.child_1 = n_g.child_1$. Even worse, practical query patterns may involve many variables that occur at several places, which makes naïve backtracking extremely slow, even though the output size may be small. This inefficiency is due to the fact that naïve backtracking does not use the equality constraints to

*Research advisors: Dan Suciu and Zachary Tatlock. ACM student member number: 2830435. Category: Undergraduate.

[1] To distinguish between variable and constant, we prefix variables with a question mark following [14].

$$
\begin{aligned}
\mathrm{compile}(p) &= Q(\mathit{root}, v_1, \ldots, v_n) \coloneq A \\
&\quad \text{where } v_1 \ldots v_n \text{ are variables in } p \\
&\quad\quad \text{and } \mathrm{aux}(p) = \mathit{root} \sim A \\
\mathrm{aux}(f(p_1, \ldots, p_n)) &= v \sim R_f(v, v_1, \ldots, v_n), A_1, \ldots, A_n \\
&\quad \text{where } v \text{ is fresh and } \mathrm{aux}(p_i) = v_i \sim A_i \\
\mathrm{aux}(x) &= x \sim \emptyset \qquad \text{where } x \text{ is a pattern variable}
\end{aligned}
$$

Figure 2: The algorithm for compiling a pattern to a CQ.

prune the search space *globally*. This is in contrast to our approach, which exploits the equality constraints during query planning for greater performance and guarantees worst-case optimality with respect to the output size.

3 APPROACH AND UNIQUENESS

We propose to view an e-graph as a set of relational tables: we represent every n-ary function symbol f as a relation R_f with $n+1$ columns. Every f-application e-node is now a tuple in R_f. The first column denotes the e-class label of n_f; the next n columns denote the e-class labels of children of n_f. Figure 1b and 1c shows the relational representation of the e-graph in Figure 1a.

Under this relational view, an e-matching problem comes out as a conjunctive query (CQ) naturally. CQ is a restricted class of relational queries that only uses conjunctive operators. For example, given schema $R_f(\mathit{eclass\text{-}id}, \mathit{child}_1, \mathit{child}_2)$ and $R_g(\mathit{eclass\text{-}id}, \mathit{child}_1)$, our example pattern $f(?\alpha, g(?\alpha))$ corresponds to the following CQ:

$$Q(\mathit{root}, ?\alpha) \coloneq R_f(\mathit{root}, ?\alpha, x), R_g(x, ?\alpha).$$

This CQ finds the set of tuples $t_f \in R_f$, $t_g \in R_g$ such that $t_f.\mathit{child}_2 = t_g.\mathit{eclass\text{-}id} \land t_f.\mathit{child}_1 = t_g.\mathit{child}_2$ and produces the substitutions $\{\mathit{root} \mapsto t_f.\mathit{eclass\text{-}id},\ ?\alpha \mapsto t_f.\mathit{child}_1\}$. Note that the root e-class is now part of the substitution. Running the CQ on database instance from Figure 1b and 1c produces a single substitution $\{\mathit{root} \mapsto 1, ?\alpha \mapsto 4\}$, denoting the e-matching result $(\{?\alpha \mapsto c_4\}, c_1)$. In this example, the query Q can be computed by a binary join of R_f and R_g, which exploits the equality constraints on $?\alpha$ and x at the same time.

Generally, we use the algorithm in Figure 2 to compile a pattern to a CQ. The aux function returns a variable and a CQ atom list. Particularly, for non-variable pattern $f(p_1, \ldots p_n)$, aux produces a fresh variable v and a concatenation of $R_f(v, v_1, \ldots, v_n)$ and atoms from A_i, where $v_i \sim A_i$ is the result of calling $\mathrm{aux}(p_i)$. For variable pattern x, aux simply returns x and an empty list. Given a pattern p, the compile function returns a CQ with body atoms from $\mathrm{aux}(p)$ and the head atom consisting of the root variable and variables in p.

We use generic join [7] as the subroutine for solving CQs. Generic join guarantees optimal performance in worst cases and is practically efficient. Our translation of patterns to queries preserves the worst-case optimal guarantee to e-matching. Namely, given a pattern p, let $M(p, E)$ be the number of substitutions yielded by e-matching on e-graph E with size n, our algorithms runs in time $\tilde{O}(\max_E |M(p, E)|)$.

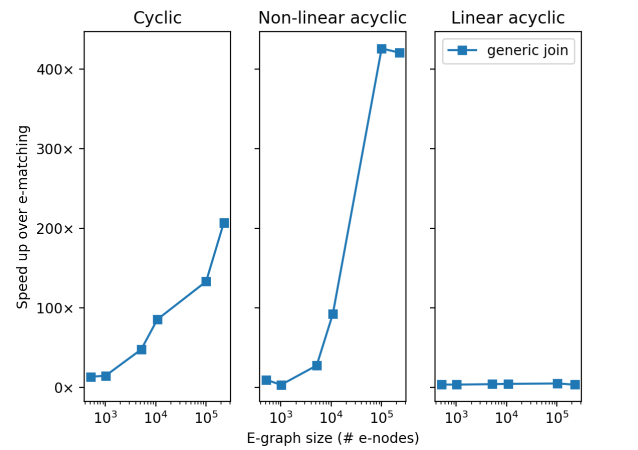


Figure 3: Speed-up for three queries relative to backtracking-based e-matching

4 RESULTS AND CONTRIBUTIONS

We run preliminary experiments on three representative e-matching queries, collected from egg's test suite implementing equality saturation for mathematical expressions[2]. The three queries compile to linear acylic, nonlinear acyclic, and cyclic CQ respectively. Cyclic and non-linear acyclic are two kinds of CQs an e-matching pattern with multi-occurrence of variables can generate, while linear acyclic queries correspond to e-matching patterns with no equality constraints. The baseline e-matching algorithm is based on an efficient virtual machine [2]. We currently implement generic join manually for each specific query.

Figure 3 shows the result. On cyclic and non-linear acyclic queries, the generic join algorithm achieve asymptotic speed-ups up to 426× over the baseline e-matching algorithm by utilizing the equality constraints during query planning. In the linear acyclic case, because no variable occurs more than once, generic join achieves similar performance as the baseline e-matching.

In summary, we propose a relational representation of e-graphs. Under this representation, e-matching comes out as CQs naturally. This allows us to apply techniques from the database community to optimize the e-matching problem. In this paper, we utilize the generic join algorithm for e-matching solving, which guarantees worst-case optimality. Preliminary experiments show that our algorithm is asymptotically faster for e-matching queries with equality constraints.

REFERENCES

[1] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, Heidelberg, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[2] 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.

[2] https://github.com/egraphs-good/egg/tree/main/tests/math.rs

[3] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

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

[5] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. 2006. Denali: A Practical Algorithm for Generating Optimal Code. *ACM Trans. Program. Lang. Syst.* 28, 6 (Nov. 2006), 967–989. https://doi.org/10.1145/1186632.1186633

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

[7] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. https://doi.org/10.1145/3180143

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

[9] 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

[10] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[11] Daniel Selsam and Leonardo Moura. 2016. Congruence Closure in Intensional Type Theory. In *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Springer-Verlag, Berlin, Heidelberg, 99–115. https://doi.org/10.1007/978-3-319-40229-1_8

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

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

[14] 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