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 naive 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 $\Sigma$ be a set of function symbols and $V$ be a set of variables. $T(\Sigma, \emptyset)$ is the set of terms inductively constructed using symbols from $\Sigma$ 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 $(c_1, \ldots, c_n)$ 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 $\sigma$ 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 $(\sigma, r)$ 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 $p$.

For instance in Figure 1a, pattern $\sigma(\alpha, g(\alpha))$ 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 $\{\alpha \mapsto c_1\}$.

Existing approaches to e-matching rely on backtracking [2, 4, 14]. For example, de Moura and Björner [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 $\sigma(\alpha, g(\alpha))$ 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 $\alpha \rightarrow n_f.child_1$ to substitution $\sigma$, iterates through all $g$-application e-nodes $n_g$ in e-class $n_f.child_2$, and only yield $\sigma$ 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


---

Figure 1: (a) an e-graph over $T(\Sigma, \emptyset)$ and $\cong$ 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, $a \cong c$, $g(a) \cong g(b)$, and $f(a, g(a)) \cong g(f(a, a))$. The labels of e-classes are at bottom left. (b) relation representing of $f$. (c) relation representing of $g$. 

1To distinguish between variable and constant, we prefix variables with a question mark following [14].
We propose to view an e-graph as a set of relational tables: we exploit the equality constraints during query planning for query size. Particularly, for non-variable pattern \( \alpha \) to a CQ. The database instance from Figure 1b and 1c produces a single sub-query:\( \{ \} \to \alpha \). 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. For example, \( f(p_1, \ldots, p_n) \) 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 aux(\( p_i \)). For variable pattern \( x \), aux simply returns \( x \) and an empty list. Given a pattern \( \rho \), the compile function returns a CQ with body atoms from aux(\( \rho \)) and the head atom consisting of the root variable and variables in \( \rho \).

\[
\text{compile}(\rho) = Q(\text{root}, v_1, \ldots, v_n) \sim A
\]

where \( v_1, \ldots, v_n \) are variables in \( \rho \)

and aux(\( \rho \)) = root \sim A

\[
\text{aux}(f(p_1, \ldots, p_n)) = \alpha \sim R_f(v, v_1, \ldots, v_n), A_1, \ldots, A_n
\]

where \( v \) is fresh and aux(\( p_i \)) = \( v_i \sim A_i \)

\[
\text{aux}(x) = x \sim \emptyset \quad \text{where } x \text{ is a pattern variable}
\]

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

We use generic join [7] as the subroutine for solving CQs. Generic join guarantees optimal performance in worse cases and is practically efficient. Our translation of patterns to queries preserves the worst-case optimal guarantee for e-matching. Namely, given a pattern \( \rho \), 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 \( O(\max E \cdot |M(p, E)|) \).

4 RESULTS AND CONTRIBUTIONS

We run preliminary experiments on three representative e-matching queries, collected from egg’s test suite implementing equality satisfaction for mathematical expressions\(^2\). The three queries compile to linear acyclic, 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\(^\times\) 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.

\(^2\)https://github.com/egraphs-good/egg/tree/main/tests/math.rs
REFERENCES


