Producing Shorter Congruence Closure Proofs in a State-of-the-Art SMT Solver

Bruno Andreotti and Haniel Barbosa

Universidade Federal de Minas Gerais (UFMG),
Belo Horizonte, Brazil
{bruno.andreotti,hbarbosa}@dcc.ufmg.br

Abstract. An important component of SMT solving is the theory of equality and uninterpreted functions, which is traditionally modelled in solvers via a congruence closure algorithm. Oftentimes, these algorithms are instrumented to provide machine checkable proofs when determining why two terms are equivalent. In a recent work published at FMCAD'22, Flatt et al. presented a modified congruence closure algorithm that could effectively produce demonstrably shorter proofs. This new algorithm relies on computing *redundant* equalities, which are not necessary to prove the equivalence between two terms but can provide shorter proofs. While promising, the modified algorithm was only considered in an equality saturation tool. In this work, we have adapted this algorithm to apply it within an SMT solver, and implemented our approach in the state-of-the-art solver cvc5. We discuss the challenges faced when integrating this algorithm into the backtracking nature of an SMT solver, and how we have addressed them. We evaluate our implementation on a large set of SMT-LIB benchmarks from multiple theories, and demonstrate how this new technique can result in smaller SMT proofs, while having only a moderate impact on runtime performance.

Keywords: SMT solving · Congruence closure · SMT proofs

1 Introduction

Proof-producing SMT solvers have a large potential for increasing the trustworthiness of formal methods applications that rely on solvers to discharge proof obligations. However, proof certificates can be quite large and therefore challenging to be checked quickly, specially in tools with limited performance, such as formally verified checkers or proof assistants. One way to mitigate this issue is to employ solving techniques that can lead to shorter proofs within the same proof calculus, which would be cheaper to check. Flatt et al. [11] recently proposed such a procedure for the congruence closure algorithm [19], a key component of SMT solvers to reason about equality and uninterpreted functions [9, 17]. Their approach keeps *redundant* equalities that would normally be discarded, since they are not needed during solving, and takes advantage of them for finding shorter proofs. Their work however was in the context of the equality saturation [25] tool

`egg` [26]. Equality saturation, which has recently seen significant usage in e.g. program optimization, works by constructing a graph that represents all equivalent forms of a program, and selecting an optimized one. Aiming to generate shorter proofs for the congruence closure algorithm as used within SMT solvers, in this paper we adapt the procedure of Flatt et al. for the CDCL($\mathcal{T}$) architecture [20] of modern SMT solvers, and implement it in the state-of-the-art solver cvc5 [4].

Extending the core component of state-of-the-art SMT solvers is notoriously challenging [6], and adapting the algorithm for CDCL($\mathcal{T}$) involves supporting backtracking and orchestrating the congruence closure reasoning with the SAT solver. Since equality saturation tools do not involve backtracking, and in general their congruence closure implementations do not need to interact with external reasoning, these considerations were not relevant in the original implementation in `egg`. Another difference is positive, however: finding shorter proofs potentially can lead to shorter *conflict clauses*. These clauses are used by SMT solvers to guide the search of the SAT solver, and shorter explanations can prune the search space more aggressively and lead to better performance. This potential advantage is not present in the equality saturation context since it is not performing a backtracking search guided by explanations.

After introducing the necessary background (Section 2), we discuss in detail the various challenges specific to the SMT setting and how we tackled them (Section 3). We also cover the specific implementation decisions to effectively integrate the algorithm to cvc5 (Section 4). Our evaluation of the current implementation (Section 5) indicates an encouraging reduction in the size of proofs from the congruence closure algorithm, as well as some significant improvement in runtime for particular families of benchmarks, while also uncovering a number of future directions for improvements.

1.1 Related work

Producing smaller unsatisfiability proofs has long been a concern in the context of SAT solvers. Fontaine et al. [12] introduced techniques for compressing resolution proofs by optimizing the resolution derivation to more efficiently reuse pivots. Heule et al. [13] proposed producing smaller proofs directly from the SAT solver, by using a more expressive proof system. This new proof system could result in proofs that are not only smaller, but also faster to check. More recently, Reeves et al. [22] leveraged *proof skeletons* to reduce the storage requirements of proofs by only recording a subset of derived clauses, those deemed "important" according to different criteria, and reconstructing the missing clauses at checking-time.

While most of these works could be adapted to work in an SMT solver, they are mainly concerned with the challenge of storing large proofs, which is especially relevant to SAT solving. In a recent work, Otoni et al. [21] showed how theory-specific proof witnessess could be used to compactly certify the execution of specific SMT algorithms. This work encompassed the theories of linear integer and real arithmetic, and equality and uninterpreted functions. For the latter, the

authors relied on a compact representation of the equality reasoning, but did not meaningfully modify the proof search in the congruence closure algorithm. To the best of our knowledge, no recent work exists in the direction of producing shorter proofs and smaller explanations from congruence closure reasoning in SMT solvers.

2 Background

2.1 CDCL($\mathcal{T}$) SMT solvers

The problem of Boolean satisfiability (SAT) consists of determing whether a formula in propositional logic is satisfiable, that is, whether there exists an assignment to the formula's free variables that makes the formula true. It is commonly assumed that the formula is in conjunctive normal form (CNF). The problem of Satisfiability Modulo Theories (SMT) is a generalization of SAT, with a few key differences. Firstly, it operates over many-sorted first-order logic, allowing quantifiers, equality, and arbitrary function symbols; and secondly, the logic is complemented by a set of *theories*, which restrict the possible interpretations of the formula. These theories usually model a real-world domain in mathematics (like the theories of integer and real arithmetic) or in computer science (like the theories of strings and floating point numbers).

Most modern SMT solvers are based on the CDCL($\mathcal{T}$) algorithm [20], a variation of the *Conflict-Driven Clause Learning* (CDCL) algorithm [15] for SAT solving. This algorithm searches for a satisfying assignment by, for each step of the search, selecting an unassigned variable and assigning it a value among $\{\top, \bot\}$. Then, the algorithm computes all other assignemnts that are directly implied by this, in a process called *propagation*. While in SAT solvers only propositional reasoning is used during this step, SMT solvers often also incorporate theory-specific reasoning, in what is referred to as *theory propagation*.

If at any point in the search a clause in the formula is unsatisfied, meaning all its literals are set to $\bot$ by the current variable assignments, we reach what is called a *conflict*. In this case, the algorithm must *backtrack*, undoing a number of assignments until it reaches a previous branching point. The algorithm also performs *conflict analysis* to understand the root cause of the conflict and incorporate that information as a new learned clause, referred to as the *conflict clause*. If a satisfying assignment is found, the CDCL($\mathcal{T}$) algorithm then employs a series of *theory solvers*, to determine whether the assignment is *consistent* with the relevant theories. If it is, the formula is satisfied; otherwise, the theory solver must produce a conflict clause, and the search continues.

2.2 Congruence closure

Congruence is the property that, given terms $a_1, \ldots, a_n$, $b_1, \ldots, b_n$ and an arbitrary function f, $a_1 = b_1 \wedge \ldots \wedge a_n = b_n$ implies $f(a_1, \ldots, a_n) = f(b_1, \ldots, b_n)$. For a given set of equalities, we say that the corresponding congruence closure

is the minimal equivalence relation that satisfies these equalities as well as the property of congruence.

In the context of SMT solving, solving the theory of equality and uninterpreted functions (EUF) consists in determining whether a conjunction of equalities and disequalities between terms is consistent under the EUF theory. Typically, this is solved by constructing the congruence closure defined by the problem equalities, and verifying that the problem disequalities respect it.

Efficient algorithms for computing congruence closure were first described by Downey et al. [9] and Nelson and Oppen [17], in the context of program verification. The algorithm is based on a *Union-find* data structure [24], where each term is a vertex in a graph (called the *equality graph*, or *e-graph*), and each equivalence class forms a tree whose root is called the class *representative*.

At first, each term is part of an equivalence class containing only itself. The algorithm works by sequentially processing equalities, and merging these equivalence classes when they are found to be equal. When processing an equality $a = b$, if the terms are not already equivalent, the algorithm finds the representatives of a and b, and adds an edge between them (called an *equality edge*), selecting one to be the representative of the new merged class. Then, the algorithm merges all classes that have become equivalent due to congruence (e.g. $f(a)$ and $f(b)$). Edges added in this step are called *congruence edges*, and the terms that caused them to be added (in this case, a and b) are the edges' *justification*.

To avoid having to deal with functions with different numbers of arguments, we will take as a starting assumption that the terms given to the congruence closure algorithm are *currified* [18]. This process, borrowed from functional programming, turns a function application into a series of applications of a special "apply" function symbol, which we will denote f. For example, the term $g(a, b, c)$, when currified, becomes $f(f(f(g, a), b), c)$. From here on, we will assume that all terms have been currified, and are therefore either constants or an application of f over two arguments.

Once the equality graph is built, we can determine if two terms are equivalent by searching for a path between their corresponding vertices. However, for many applications (including SMT solvers), simply determining whether two terms are equivalent is not enough—it is also necessary to produce an *explanation* of their equivalence. Here, an explanation is defined as a subset of the input equalities that is sufficient to ensure the terms are equivalent. In SMT solvers that use the CDCL($\mathcal{T}$) algorithm [20], explanations from congruence closure are crucial, as they are used to construct conflict clauses for the solver. Furthermore, since a smaller conflict clause will prune a larger part of the search space, a small explanation is preferable. However, it is known that finding the smallest possible explanation for the equivalence of two terms is an NP-complete problem [10].

Besides providing an explanation, a congruence closure algorithm may also produce a structured *proof*. If an explanation is simply a set of equalities that justify the equivalence between two terms, a proof is a derivation, using these equalities and the properties of reflexivity, symmetry, transitivity and congruence to demonstrate that the two terms are equivalent. Proof-producing SMT solvers

have become increasingly important in the last few years, and are already used in many applications [2, 3, 5, 14, 21], including proof automation in interactive theorem provers [23].

An explanation-producing congruence closure algorithm was first described by Nieuwenhuis et al. in [19]. When explaining the equivalence of two terms, this algorithm traverses the path between respective nodes in the equality graph. When traversing an equality edge, the algorithm simply records the input equality associated with that edge. When traversing a congruence edge between the terms $f(a_1, a_2)$ and $f(b_1, b_2)$, the algorithm recursively explains the equivalence of the terms a_1 and b_1, and that of a_2 and b_2; then adds all returned equalities to the explanation. Figure 1 shows the pseudocode for this algorithm. While the algorithm as described can only produce explanations, it can be straightforwardly extended to produce structured proofs.

```
1  function get_explanation(start, end):
2      let explanation = []
3      let lca = find_lowest_common_ancestor(start, end)
4      explanation += explain_along_path(start, lca)
5      explanation += explain_along_path(end, lca)
6      return explanation
7
8  function explain_along_path(lower, upper):
9      let explanation = []
10     let current = lower
11     while current != upper:
12         let edge = current.edge_to_parent()
13         if edge is congruence edge between f(a1, a2) and f(b1, b2):
14             explanation += get_explanation(a1, b1)
15             explanation += get_explanation(a2, b2)
16         else:
17             explanation += edge
18         current = current.parent()
19     return explanation
```

Fig. 1: Pseudocode for a classical proof-producing explanation algorithm. The function `get_explanation` returns a list with the explanation for the equivalence of two terms. Since the equivalence class is represented by a rooted tree, the function works by explaining the path from each of the nodes to their lowest common ancestor.

Importantly, the explanations returned by this algorithm might not be the smallest valid explanations. While there is only one unique path between the two nodes in the tree, if the input equalities were processed in a different order, or if different congruence edges were added, the explanation for the equivalence of these nodes might be different. More generally, the fact that the standard con-

gruence closure algorithm discards redundant equalities means that oftentimes shorter explanations are not found.

2.3 The Greedy congruence closure algorithm

In an effort to produce shorter explanations from congruence closure, Flatt et al. [11] developed two new congruence closure algorithms, called TreeOpt and Greedy, that do not discard redundant equalities, and instead use them to provide alternative, possibly shorter paths in the equality graph. Additionally, these new algorithms also compute extra congruence edges each time two classes are merged, contributing to even more alternative paths.

Keeping redundant edges means that the graph for each equivalence class is no longer a tree. Whereas before there was always only one path between two terms in an equivalence class, now the explanation algorithm must have a way to determine which of the possible paths represents the shortest proof. This is where the strategies used by the two algorithms diverge.

The TreeOpt algorithm is a $O(n^5)$ algorithm that finds an optimal proof for a slightly modified metric of proof size. On the other hand, Greedy is a $O(n \log n)$ heuristic algorithm, that attempts to find small explanations without incurring an increase in complexity when compared to traditional congruence closure algorithms.

Flatt et al. [11] convincingly showed that TreeOpt does not present a significant improvement in proof size when compared to Greedy, while having a substantial performance overhead. For this reason, we believe that TreeOpt is impractical for use in SMT solvers, and therefore focus our interest on Greedy.

Greedy is a greedy congruence closure algorithm that attempts to find a small explanation while keeping the same $O(n \log n)$ complexity as traditional algorithms. To do this, it first computes an estimate of the proof size for each edge in the equality graph. This estimate is obtained by computing the proof size of the edge with the traditional congruence closure algorithm, that is, ignoring redundant equalities.

Once the estimates are calculated, the algorithm simply finds the shortest path between the terms, using the estimates as weights for the edges. The algorithm is parameterized by an integer *fuel*, which, similarly to [11], we set to 10 as a default. When it encounters a congruence edge, if the fuel is greater than 0, the algorithm recurses (and decrements its fuel) to explain the justification of the congruence edge. However, if the fuel is 0, the algorithm explains the justification by simply using the classical congruence closure algorithm, i.e., ignoring redundant equalities.

2.4 Computing extra redundant edges

As mentioned, besides not discarding redundant equalities that are asserted, the new algorithms also compute extra redundant congruence edges everytime two equivalence classes are merged. This is done by finding all terms in the newly

merged class that have the same *canonical form*. For an application term $f(a, b)$, we say that its canonical form is the term $f(a', b')$, where t' is the representative of the equivalence class of the term t. For a constant term t, we say that its canonical form is just t. Notably, if two non-identical terms have the same canonical form, it means they are equivalent by congruence, and we may add a congruence edge between them.

```
1  function get_canonical_form(term):
2      if term is of the form f(a,b):
3          let a' = representative_of(a)
4          let b' = representative_of(b)
5          return f(a',b')
6      else:
7          return term
8
9  function compute_extra_edges(eclass):
10     let canonical_map = {}
11     for term in eclass:
12         let canon = get_canonical_form(term)
13         for other in canonical_map[canon]:
14             add_edge(term, other)
15             if number_of_redundant_edges > LIMIT:
16                 return
17         canonical_map[canon] += term
```

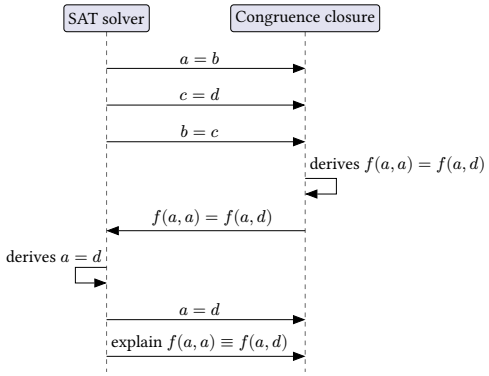
Fig. 2: The algorithm for computing extra congruence edges between the two equivalence classes.

Figure 2 shows the algorithm for computing extra congruence edges in an equivalence class. We keep a map from a canonical form term to a list of terms in the class that have that canonical form. When we process a term, we compute its canonical form, and add an edge between it and every other term that shares that canonical form. This process is repeated for every term in the equivalence class, or until an arbitrary limit is reached.

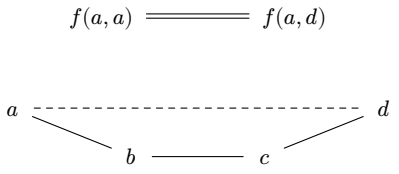
While the original work by [11] implemented the new algorithms in an equality saturation tool, implementing them in an SMT solver requires several changes due to the backtracking nature of the solver and the complex interactions between the congruence closure algorithm and other modules. Over the next sections, we describe in detail the challenges we faced when adapting the Greedy algorithm, and how we tackled them.

3 Handling implicit dependencies in a congruence closure within CDCL($\mathcal{T}$)

Modern SMT solvers work by orchestrating a CDCL SAT solver that finds models for an abstraction of the input formula, with multiple theory solvers that check if these models are consistent. In the EUF theory, the theory solver is also commonly used during *propagation*, that is, even before a full model is found, the EUF solver is queried to ensure that the partial model that is being constructed by the SAT solver is consistent, and also to derive new facts that can be propagated. This results in a back-and-forth interaction between the SAT solver and the theory solver, where the SAT solver adds propagated facts into the congruence closure algorithm, which in turn will derive new literals that are asserted to the SAT solver. Figure 3 shows an example of this interaction between the SAT solver and the congruence closure algorithm during propagation, and the e-graph after this interaction has played out.



(a) Interaction between SAT solver and congruence closure algorithm.



(b) Equality graph at the end of the interaction.

Fig. 3: An example of a possible interaction between the SAT solver and the congruence closure algorithm, and the e-graph that is constructed after this interaction has played out. Here, we use simple lines to denote equality edges; double lines to denote congruence edges; and dashed lines to denote redundant edges.

In the example, the SAT solver first asserts into the congruence closure the equalities $a = b$, $c = d$ and $b = c$, in that order. When the latter is added, the terms a and d are in the same equivalence class, which means the terms $f(a, a)$ and $f(a, d)$ have become equivalent by congruence. The congruence closure algorithm adds the congruence edge $(f(a, a), f(a, d))$ accordingly. Then, as part of *theory propagation*, the congruence closure algorithm sends the lemma $f(a, a) = f(a, d)$ to the SAT solver. This in turn causes a propagation in the

solver, which results in the fact $a = d$ being derived by SAT reasoning. This equality is asserted to the congruence closure algorithm, which results in the redundant edge (a, d) being added.

Now, the edge $a = d$ depends on the equality $f(a, a) = f(a, d)$, even though that dependency is not represented in any way in the equality graph—we call these *implicit dependencies*. When queried for an explanation of $f(a, a) \equiv f(a, d)$, the congruence closure algorithm cannot use the equality $a = d$ in the explanation, as that would result in a circular proof. Instead, the only correct explanation for this equivalence is the one based on the longer path between a and d, namely the equalities $a = b$, $b = c$ and $c = d$.

In the traditional congruence closure algorithm, where no redundant edges are kept, after the path between two terms is first established it will never change. This means that equalities which are implicitly dependent on the equivalence between two terms will never influence the explanation that is returned for those terms' equivalence, preventing this kind of circular proof. In this modified version of the algorithm, however, we need to take special care to ensure this cannot happen.

3.1 Edge levels

To properly handle implicit dependencies, we need to augment the equality graph with information that encodes these dependency relations. The most precise way of doing this would be to construct the entire implication graph of the SMT solver, and restrict which edges can be used based on that. However, this is complicated to do in practice and might incur a performance cost, since modern SMT solvers rely heavily on lazily computing proofs. For example, most SAT solvers do not store the entire implication graph and instead keep a *trail*, which consists of a list of all assigned literals, in the order that they were assigned. Since a literal can only have been implied by literals that were assigned before it, this trail is a topological ordering of the implication graph.

We follow a similar strategy, and augment the equality graph based on the order in which the equalities are asserted into the congruence closure algorithm. Specifically, we assign to each edge in the e-graph a *level*[1], which is the index in which this edge was added to the graph; as such, these levels also correspond to a topological ordering of the implication graph. We also define the concept of two terms' *merge level*. For two terms in the same equivalence class, we say that their merge level is the level in which the terms were originally determined to be equivalent, i.e., the level of the edge that merged their equivalence classes. When searching for the explanation of the equivalence between two terms, we restrict the search to edges whose level is no greater than the terms' merge level. Effectively, this restricts the search to edges already present in the e-graph when the equivalence of the two terms was first derived.

[1] Although similar concepts, these levels are distinct from the SAT solver's *decision levels*. In the example in Figure 3, all of the shown interaction could have taken place during propagation, which is to say, during a single decision level.

Figure 4 shows the e-graph from Figure 3b, but with level information denoted in blue. Consider again querying this e-graph for the explanation of $f(a,a) \equiv f(a,d)$. Their equivalence was first determined when the congruence edge between them was added, so their merge level is the level of that edge, namely 3. Therefore, the redundant edge between a and d, which has level 4, and which implicitly depends on the congruence edge $(f(a,a), f(a,d))$, will not be included in the search for an explanation, preventing a circular proof.

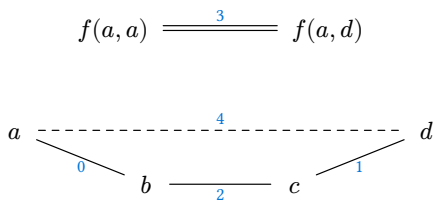


Fig. 4: The e-graph from Figure 3b, but with the level of each edge shown in blue.

Figure 5 shows the pseudocode for the GREEDY algorithm, modified to take into account the levels of edges. Now, besides finding the shortest path between the terms according to the computed weights, this shortest-path search must also ignore edges whose level is greater that `max_level`, set to be the merge level of the two terms.

```
function get_explanation(start, end, weights):
    let max_level = get_merged_level(start, end)
    let path = shortest_path(start, end, weights, max_level)
    let explanation = []
    for edge in path:
        if edge is congruence edge between f(a1, a2) and f(b1, b2):
            explanation += get_explanation(a1, b1, weights)
            explanation += get_explanation(a2, b2, weights)
        else:
            explanation += edge
    return explanation
```

Fig. 5: Pseudocode for the modified explanation algorithm. Here, we assume `shortest_path` is a function that returns a list of edges representing the shortest path between two nodes, given a set of edge weights, and rejecting all edges whose level is greater than `max_level`.

3.2 Alternative solutions

Restricting the explanation search with edge levels is an aggressive way of preventing circular proofs, and at first glance, it might seem that other, less restrictive solutions might be possible. In this section, we discuss some alternative solutions that were considered for the implicit dependencies problem, and why ultimately they were not feasible.

Weighing the edges based on proof size The Greedy algorithm already relies on adding weights to the e-graph edges to enable finding smaller proofs. These weights attempt to capture the size of the proof required to justify an edge. Conceivably, it would make sense to make these weights also include the parts of the proof that go beyond congruence closure reasoning. Currently, an equality that is asserted into the congruence closure algorithm would receive the weight 1, even if it implicitly depends on an equivalence, since the weight cannot capture the external reasoning that was done to derive the asserted equality. If instead we could compute an appropriate weight for that edge, which took into account all the reasoning required to derive it, we could ensure that an edge which implicitly depends on an equivalence will have a weight bigger than the proof size for that equivalence. Thus, when explaining this equivalence, the shortest path found would not include this edge.

While this strategy might work in theory, it would require estimating the proof size for each edge, including parts of the proof beyond congruence closure reasoning. In a modern SMT solver, most of these proofs are computed in a lazy fashion [5], and changing that behaviour would greatly impact performance. Therefore, while this technique might work in different contexts, it is not practical for a high-performance SMT solver.

Computing levels based on *provenance* While constructing the entire implication graph for the SMT solver might be impractical, there are incomplete alternatives that are less restrictive than the arbitrary topological ordering imposed by the edge levels. In particular, when the SAT solver asserts a literal l into the EUF theory solver, it has access to the set of literals that immediately caused that literal to be propagated (in the SAT solver, this would be a clause). This set of literals, which we denote as l's *provenance*, represents the direct ancestors of l in the implication graph. Using this information, we can define the level of a literal l (which will determine the level of edges in the e-graph) to be one more than the maximum level of the literals in its provenance. More formally,

$$\text{level}(l) = 1 + max\,\{\, \text{level}(p) \mid p \in \text{provenance}(l) \,\}$$

Contrary to the previous level solution, this would allow multiple edges to share the same level, and in fact each level will correspond to one "layer" of the implication DAG. This would mean that equalities that have the same level have no dependency relation in the implication graph—one could be explained by the other and vice-versa.

While this solution is quite elegant[2], it would still result in explanations that are rejected by the SAT solver as circular. Let l_1 and l_2 be two literals with the same provenance level, and say the EUF solver produces an explanation for l_1 in terms of l_2, but l_2 is situated after l_1 in the SAT solver trail. Since the SAT solver does not store the full implication graph, it cannot ensure that the explanation given is valid, *even if it does respect the dependency relations in the implication graph*. In other words, the SAT solver considers the particular topological ordering of the implication graph that composes the SAT trail as the "source of truth", and will reject explanations that do not follow it, even if they wouldn't lead to circular proofs.

This invariant sits at the core of modern SAT solvers, and changing it would involve allowing arbitrary *reimplication* of propagated literals. There has been some recent work exploring the benefits of reimplication in SAT solving [8, 16], but these techniques are still not well established in SMT solvers. Until that changes, we believe that any correct solution for the implicit dependencies problem will necessarily be as strict as the above solution based on the edge levels.

4 Implementation

We have implemented the modified version of the congruence closure algorithm from the previous section in cvc5 [4], a state-of-the-art SMT solver. The existing theory solver for the theory of equality and uninterpreted functions, called its *equality engine*, implements a version of a proof-producing congruence closure algorithm as seen in Nieuwenhuis et al. [19]. As part of this work, we adapted this existing implementation to not discard redundant equalities, and implemented the Greedy explanation algorithm, with the adaptations detailed in the previous sections. This required rewriting a large part of the equality engine, with over 900 lines of code changed. In this section, we highlight some of the implementation details that went in to this work.

4.1 Computing the merge level of two terms

An important part in selecting which edges are allowed in an explanation is determining the merge level of the terms being explained. For a given path between two terms in the e-graph, we said that its *path level* is the maximum of the levels of all edges in path. This corresponds to the level that the path first appeared in the e-graph. Note that the level in which two terms were merged is the level in which any path first appeared between them. Thus, the merge level of two terms is the minimum path level of all the paths between them.

However, we know that when two terms are merged, the path between them with minimal path level will not contain any redundant edge (otherwise that edge would have caused a merge, and would not be redundant). Therefore, we only

[2] It does, however, require changing the API of the EUF theory solver to receive provenance information, and updating all users of that API to provide that information, which is a non-trivial engineering effort.

need to concern ourselves with paths that don't contain redundant edges. There is always only one such path between two terms, which is the path computed by the classical congruence closure algorithms, known as the *tree path*.

All put together, to compute the merge level of two terms we must simply traverse the tree path between them, and record the highest edge level encountered.

4.2 Backtracking-aware edge limits

As mentioned earlier, when computing extra redundant congruence edges, the modified congruence closure algorithm only adds edges up to a limit. This is intended to ensure the algorithm has a linear overhead when compared to traditional congruence closure algorithms, since in theory the extra edges could amount to $O(n^2)$ where n is the number of nodes in the e-graph. In the original implementation [11], this limit was set to $2n$.

When implementing these algorithms in cvc5, however, we found that setting a limit based on the current state of the graph was not sufficient. In particular, in many benchmarks we saw a situation where the SAT solver asserts an equality to the congruence closure algorithm, which caused a large number of redundant edges to be computed; then, the solver backtracks, removing all the redundant edges, as well as the asserted equality; and after that the process repeats, with the solver adding a small number of non-redundant edges followed by computing a large number of redundant edges.

In these benchmarks, we saw that the total number of redundant edges added greatly shadowed the number of non-redundant edges, and there was a substantial slowdown compared to the traditional congruence closure algorithm. To address this, we implemented a smarter limit to the number of computed redundant edges, which takes into account all the edges added throughout the solving process. Now, the total number of redundant edges added cannot exceed twice the number of non-redundant edges.

4.3 Lazy computation of weights and extra edges

Due to backtracking, it might be the case that edges added to the e-graph will be removed before an explanation using them is computed. Therefore, eagerly recomputing edge weights everytime an edge is added would be inefficient. Instead, we only compute edge weights when searching for an explanation.

Recall that the weight of an edge is based on the proof size for that edge, without using redundant equalities. Therefore, the only edges that affect the weight of a given edge are those that appeared before it. cvc5's equality engine backtracks by removing edges in reverse order, meaning that if an edge was not removed during backtracking, no edge before it was removed either. As such, when backtracking, we don't need to invalidate any computed edge weight, aside from removing the weights of edges that were removed.

We also compute the extra congruence edges as described in Section 2.4 in a lazy fashion, only when queried for an explanation. Importantly, we must take

care to set the edge level correctly. For a redundant congruence edge between the terms a and b, we can safely set its level to the merge level of a and b. This is because, although we potentially compute this edge much later, it could have conceivably been added as soon as the terms' equivalence classes were merged—it does not have any implicit dependencies.

5 Evaluation

In order to evaluate the effectiveness and performance of our implementation, we tested it against a large set of benchmarks from the SMT-LIB benchmarks library [7], an industry-standard set of benchmarks for SMT solvers. We used the 162,228 SMT problems from a set of 14 logics, notably involving, besides the theory of equality and uninterpreted functions, the theories of strings and arrays[3]. These particular theories were selected because they make heavy use of equality reasoning, and are good testing grounds for the congruence closure algorithm. The results were generated with a cluster equipped with 32 x Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz, 256 GiB RAM machines, with one core per solver/benchmark pair, 120s time limit, and 8 GiB memory limit.

We focus our evaluation on a comparison between the Greedy and Vanilla algorithms. For each benchmark, we recorded the time taken to solve the benchmark with each of the algorithms, as well as the size of the final proof produced. We also measure directly, for each benchmark, the explanation size returned by all calls to `get_explanation`.

Table 1 shows an overview of our main results. The benchmarks are divided in three groups, based on the logic: the logics that include the theory of arrays, those that include the theory of strings, and those that include neither. For each logic group, the table presents the relative change in runtime, proof size and explanation of the Greedy algorithm, when using Vanilla as a baseline.

The table shows that the perfomance of the modified algorithm is heavily theory-dependent. For the theory of strings, the change in explanation size is very minor, but the total proof size is moderately smaller when using Greedy than with Vanilla. However, the runtime overhead is very large for these logics. On the other hand, for the other two benchmark groups, the explanation size difference is more pronounced, but this difference did not translate into a large difference in the total proof size. The runtime overhead for these groups was significantly smaller, but still substantial.

In the next few sections, we take a more detailed look into each of the evaluated metrics.

5.1 Runtime

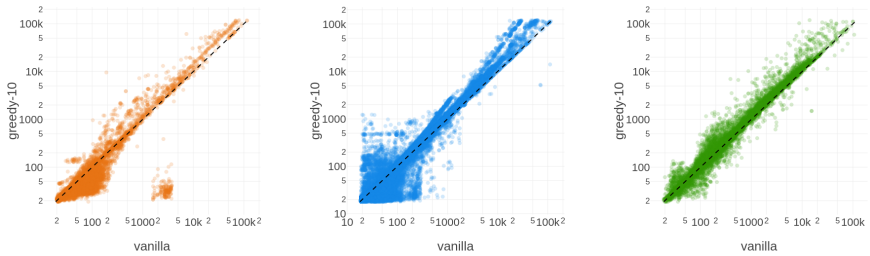
Figure 6 presents scatter plots of the runtime of each benchmark, when run with Greedy and Vanilla. The benchmarks are again divided in three groups,

[3] The specific logics used were ALIA, AUFLIA, AUFLIRA, QF_ALIA, QF_AUFLIA, QF_AX, QF_S, QF_SLIA, QF_UF, QF_UFLIA, QF_UFLRA, UF, UFLIA, and UFLRA.

Fragment	Runtime	Proof size	Explanation size
Arrays-based logics	25.96 %	-3.26 %	-7.97 %
Strings-based logics	56.52 %	-4.34 %	-0.83 %
Remaining logics	29.85 %	-3.16 %	-21.27 %
Total	45.59 %	-3.33 %	-7.91 %

Table 1: Relative change in runtime, proof size and explanation size of Greedy when compared to Vanilla, for different groups of logics.

depending on the logic. Points below the diagonal represent benchmarks in which Greedy outperformed Vanilla, and points above the diagonal are the opposite.



(a) Runtime for arrays-based logics. (b) Runtime for strings-based logics. (c) Runtime for remaining logics.

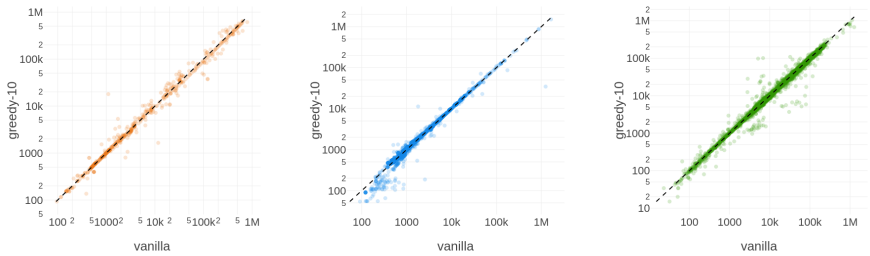
Fig. 6: Scatter plots of the runtime of each benchmark, when run using Greedy and Vanilla.

The plot shows that, for most benchmarks, and in particular larger benchmarks, Greedy presents a signifcant overhead when compared to Vanilla. In total, solving all benchmarks with Greedy took 45.6% more time than with Vanilla.

Interestingly, there were a number of benchmarks that took significantly less time to run with Greedy. In the most extreme case, the plot shows a set of benchmarks in the array-based logics that take a few seconds to be solved with Vanilla, but only several milliseconds when using Greedy. Over all logics, there were 5,347 benchmarks that were at least twice as fast with Greedy when compared to Vanilla (out of 162,228 total benchmarks). We speculate that this is related to smaller congruence closure explanations resulting in smaller conflict sets, which in turn results in a more efficient SAT search.

5.2 Proof size

Figure 7 presents, for each benchmark group, a scatter plot of the final proof size of each benchmark, when run with Greedy and Vanilla. Benchmarks where the proof size was exactly the same with both algorithms are omitted.



(a) Proof size for arrays-based logics. (b) Proof size for strings-based logics. (c) Proof size for remaining logics.

Fig. 7: Scatter plots of the final proof size of each benchmark, when run using Greedy and Vanilla.

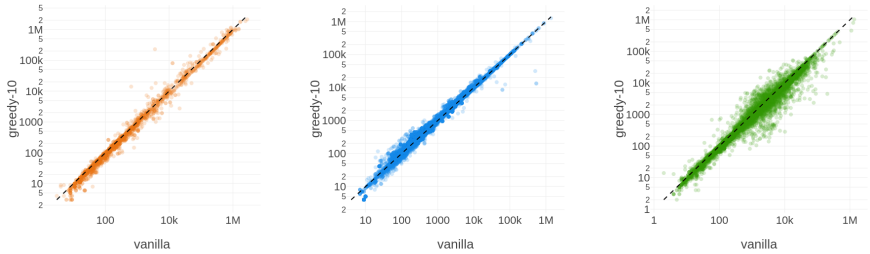
For most benchmarks, the final proof size did not change substantially between the two algorithms. However, since this is a measure of the final proof size, which includes reasoning steps besides congruence closure, the impact of the congruence closure algorithm is mitigated. In total, the proof size of all proofs was 3.3% smaller when generated with Greedy, when compared to Vanilla.

5.3 Explanation size

To more closely analyze the impact of the modified algorithm, we also looked at, for each benchmark, the size of all explanations returned by the congruence closure algorithm. Figure 8 presents scatter plots of the total explanation size of each benchmark, when run with Greedy and Vanilla. The benchmarks were grouped similarly to the previous plots, and benchmarks where the total explanation size was exactly the same with both algorithms are omitted.

This plot more clearly shows that the explanations returned by Greedy are generally smaller than those of Vanilla. However, this is not consistent across theories. In particular, benchmarks form string-based logics seem to benefit less from the new algorithm, while benchmarks from other logics show a more clear improvement.

Overall, the total explanation size for the Greedy algorithm was 7.9% smaller that that of Vanilla.



(a) Explanation size for arrays-based logics.

(b) Explanation size for strings-based logics.

(c) Explanation size for remaining logics.

Fig. 8: Scatter plots of the total explanation size of each benchmark, when run using Greedy and Vanilla.

5.4 Comparison with baseline implementation from cvc5

As mentioned earlier, implementing the Greedy algorithm in cvc5 required rewriting a large portion of the equality engine. This involved changing even how the traditional algorithm was implemented, which we refer to as Vanilla. To guarantee our changes did not degrade substantially the performance of cvc5's equality engine, we also compared the new Vanilla implementation with the baseline implementation, before any changes. These results are shown in Figure 9.

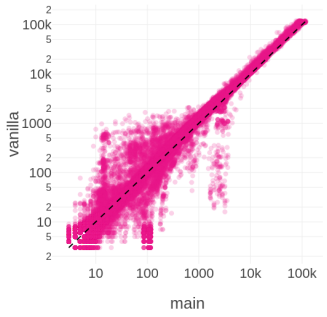


Fig. 9: Scatter plots of the runtime of each benchmark, when run using Vanilla versus the existing implementation in cvc5.

While there is a lot of noise in the results, it is possible to see that there is a small consistent overhead in the implementation of Vanilla when compared to the baseline, especially in larger benchmarks. On average, benchmarks take 8%

longer to solve when using Vanilla. While this overhead is not negligible, it is realtively modest, and could be improved with further optimization.

5.5 Other results

To get a more detailed view into how the redundant edges affect the returned explanations, we also recorded a series of *redundancy* measurements. Over all benchmarks, the number of redundant equalities included in the returned explanations was 3.53% of the total explanation size. Similar to other metrics, this was also theory-dependent: For strings-based logics this ratio was only 0.78%, while for arrays and other logics, it was 5.79% and 6.61%, respectively. We also measured the proportion of all e-graph edges that were redundant. Over all benchmarks, 43.58% of added edges were redundant—this was largely consistent across all logic groups, staying between 41.92% and 47.78%.

6 Conclusion and future work

We presented our effort to adapt a new congruence closure algorithm to work in a state-of-the-art SMT solver. We evaluated our implementation on a large, industry-standard set of benchmarks, and obtained mixed but promising results.

Our work shows that congruence closure algorithms that keep redundant equalities can be efficiently implemented in the context of a backtracking SMT solver, with reasonable overhead. Furthermore, we show that these techniques have a measurable and at times substantial impact on explanation and proof size. In particular, for logics involving the theories of arrays and equality and uninterpreted functions, we show that this new algorithm can result in consistently smaller explanations, with a small impact on final proof size. For some benchmarks, we also observed that the new algorithm can result in a drastic reduction in runtime. As for logics involving the theories of strings, the new algorithm did not produce significantly smaller explanations, but it did result in moderately smaller final proofs. However, the runtime overhead was also larger for these benchmarks.

Future opportunities for research may include improving the heuristics or developing new algorithms that make use of redundant edges. For example, the current results seem to show that this technique struggles with very large benchmarks, and maybe a more sophisticated heuristic for limiting the number of redundant edges added might be beneficial.

While our work was initially focused in treating the interaction between the equality engine and the SAT solver, congruence closure reasoning is used by many theories in SMT solving, and future work might explore how to optimize these techniques for the specific interactions between those theories and the congruence closure algorithm. Finally, this work is still somewhat restricted by the specific topological ordering of the implication graph that the SAT solver selects as its trail. One possible avenue for future research is investigating how reimplication, when adapted for an SMT context, might improve this restriction.

In conclusion, while the observed experimental results are overall mixed, there is promising evidence that these technique might be useful in some contexts and for some theories.

Data Availability Statement The source code relevant to this research, as well as the tools and data used for the experimental evaluation are available at [1].

References

1. Andreotti, B., Barbosa, H.: Artifact for "producing shorter congruence closure proofs in a state-of-the-art smt solver" (Sep 2025). https://doi.org/10.5281/zenodo.17181344, https://doi.org/10.5281/zenodo.17181344
2. Andreotti, B., Lachnitt, H., Barbosa, H.: Carcara: An efficient proof checker and elaborator for smt proofs in the alethe format. In: Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part I. p. 367–386. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30823-9_19, https://doi.org/10.1007/978-3-031-30823-9_19
3. Barbosa, H., Barrett, C., Cook, B., Dutertre, B., Kremer, G., Lachnitt, H., Niemetz, A., N ötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Tinelli, C., Zohar, Y.: Generating and exploiting automated reasoning proof certificates. Commun. ACM **66**(10), 86–95 (sep 2023). https://doi.org/10.1145/3587692, https://doi.org/10.1145/3587692
4. 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 Construction and Analysis of Systems (TACAS), 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
5. Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C., Barrett, C.: Flexible proof production in an industrial-strength smt solver. In: Automated Reasoning: 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8–10, 2022, Proceedings. p. 15–35. Springer-Verlag, Berlin, Heidelberg (2022). https://doi.org/10.1007/978-3-031-10769-6_3, https://doi.org/10.1007/978-3-031-10769-6_3
6. Barbosa, H., Reynolds, A., Ouraoui, D.E., Tinelli, C., Barrett, C.W.: Extending SMT solvers to higher-order logic. In: Fontaine, P. (ed.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 11716, pp. 35–54. Springer (2019). https://doi.org/10.1007/978-3-030-29436-6_3, https://doi.org/10.1007/978-3-030-29436-6_3
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
8. Coutelier, R., Fleury, M., Kovács, L.: Lazy reimplication in chronological backtracking. In: Chakraborty, S., Jiang, J.R. (eds.) 27th International Conference

on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India. LIPIcs, vol. 305, pp. 9:1–9:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2024). https://doi.org/10.4230/LIPICS.SAT.2024.9, https://doi.org/10.4230/LIPIcs.SAT.2024.9
9. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM **27**(4), 758–771 (oct 1980). https://doi.org/10.1145/322217.322228, https://doi.org/10.1145/322217.322228
10. Fellner, A., Fontaine, P., Paleo, B.W.: Np-completeness of small conflict set generation for congruence closure. Form. Methods Syst. Des. **51**(3), 533–544 (dec 2017). https://doi.org/10.1007/s10703-017-0283-x, https://doi.org/10.1007/s10703-017-0283-x
11. Flatt, O., Coward, S., Willsey, M., Tatlock, Z., Panchekha, P.: Small proofs from congruence closure. In: 2022 Formal Methods in Computer-Aided Design (FMCAD). pp. 75–83 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_13
12. Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Compression of propositional resolution proofs via partial regularization. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23. pp. 237–251. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
13. Heule, M.J.H., Kiesl, B., Biere, A.: Short proofs without new variables. In: de Moura, L. (ed.) Automated Deduction – CADE 26. pp. 130–147. Springer International Publishing, Cham (2017)
14. Hoenicke, J., Schindler, T.: A simple proof format for SMT. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) International Workshop on Satisfiability Modulo Theories (SMT). CEUR Workshop Proceedings, vol. 3185, pp. 54–70. CEUR-WS.org (2022), http://ceur-ws.org/Vol-3185/paper9527.pdf
15. Marques Silva, J., Sakallah, K.: Grasp-a new search algorithm for satisfiability. In: Proceedings of International Conference on Computer Aided Design. pp. 220–227 (1996). https://doi.org/10.1109/ICCAD.1996.569607
16. Nadel, A.: Introducing intel(r) SAT solver. In: Meel, K.S., Strichman, O. (eds.) 25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 236, pp. 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.SAT.2022.8, https://doi.org/10.4230/LIPIcs.SAT.2022.8
17. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (apr 1980). https://doi.org/10.1145/322186.322198, https://doi.org/10.1145/322186.322198
18. Nieuwenhuis, R., Oliveras, A.: Congruence closure with integer offsets. In: Vardi, M.Y., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 78–90. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
19. Nieuwenhuis, R., Oliveras, A.: Proof-producing congruence closure. In: Giesl, J. (ed.) Term Rewriting and Applications. pp. 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
20. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). J. ACM **53**(6), 937–977 (Nov 2006). https://doi.org/10.1145/1217856.1217859, http://doi.acm.org/10.1145/1217856.1217859
21. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-specific proof steps witnessing correctness of SMT executions. In: Design Automation Conference (DAC). pp. 541–546. IEEE (2021). https://doi.org/10.1109/DAC18074.2021.9586272, https://doi.org/10.1109/DAC18074.2021.9586272

22. Reeves, J.E., Kiesl-Reiter, B., Heule, M.J.H.: Propositional proof skeletons. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 329–347. Springer Nature Switzerland, Cham (2023)
23. Schurr, H., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) Proc. Conference on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 12699, pp. 450–467. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_26, https://doi.org/10.1007/978-3-030-79876-5_26
24. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (apr 1975). https://doi.org/10.1145/321879.321884, https://doi.org/10.1145/321879.321884
25. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. SIGPLAN Not. **44**(1), 264–276 (Jan 2009). https://doi.org/10.1145/1594834.1480915, https://doi.org/10.1145/1594834.1480915
26. 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