Towards Producing Shorter Congruence Closure Proofs in a State-of-the-art SMT Solver (Extended Abstract)

Bruno Andreotti, Haniel Barbosa, Oliver Flatt

PAAR 2024 at Nancy, France, 2024–07–02

Congruence closure

- The property of congruence states that, for any terms x, y, and any function f,

$$x = y \to f(x) = f(y)$$

- For a given set of equalities, the congruence closure is a minimal equivalence relation that satisfies them, as well as reflexivity, symmetry, transitivity and *congruence*

Congruence closure in SMT solvers

- Congruence closure algorithms are essential for solving the theory of equality and uninterpreted functions (EUF)

- The solver will provide a a set of equalities and inequalities, and the congruence closure algorithm must determine if it is consistent

Congruence closure in SMT solvers

- However, for SMT solvers, it is not sufficient to just determine whether two terms are equivalent

- We also require an *explanation*, that is, a minimal set of equations that makes the terms equivalent

Congruence closure in SMT solvers

- However, for SMT solvers, it is not sufficient to just determine whether two terms are equivalent
- We also require an *explanation*, that is, a minimal set of equations that makes the terms equivalent
- Furthermore, we might also want a structured *proof*, that uses these equalities as assumptions to derive the equivalence of the two terms

Proof-producing congruence closure

- A proof-producing congruence closure algorithm was presented by Nieuwenhuis et al. in 2005

- It is based on a union-find data structure, and contructs an equality graph to represent the equivalence relation

Proof-producing congruence closure

- A proof-producing congruence closure algorithm was presented by Nieuwenhuis et al. in 2005
- It is based on a union-find data structure, and contructs an equality graph to represent the equivalence relation
- Then, finding the explanation for the equivalence of two terms consists in finding a path between them in the graph

Proof-producing congruence closure: Example

- TODO

Keeping redundant equalities

- You may have noticed that we discard equalities between terms we already know to be equivalent
- However, keeping these equalities might allow us to find shorter paths between terms, and thus sorter proofs

Keeping redundant equalities

- Flatt et al. at FMCAD'22 presented two proof producing congruence closure algorithms that make use of redundant equalities to find smaller proofs

Keeping redundant equalities

- Flatt et al. at FMCAD'22 presented two proof producing congruence closure algorithms that make use of redundant equalities to find smaller proofs

- In the original work, they were implemented in an equality saturation tool

Keeping redundant equalities

- Flatt et al. at FMCAD'22 presented two proof producing congruence closure algorithms that make use of redundant equalities to find smaller proofs

- In the original work, they were implemented in an equality saturation tool

- Here, we present our effort to implement these algorithms in cvc5, a state-of-the-art SMT solver

Keeping redundant equalities: Example

- TODO

A side note: tree-size vs DAG-size

- Finding the minimal proof for the equivalence of two terms is an NP-hard problem

- However, this problem becomes easier if we don't allow the reuse of proof steps

A side note: tree-size vs DAG-size

- Finding the minimal proof for the equivalence of two terms is an NP-hard problem
- However, this problem becomes easier if we don't allow the reuse of proof steps
- We call this the *tree size* of the proof, in contrast to the *DAG size*

TreeOpt algorithm

- Optimal algorithm (with regards to proof tree size)
- Computes the weight of each congruence edge by finding the size of the explanation of its justification, until a fixed point
- When asked to explain the equivalence between two terms, simply finds the sortest path between them considering these weights

TreeOpt algorithm

```
function compute_weights():
    let weights = {}
    for edge in edges:
        if not edge.is_congruence_edge():
            weights[l, r] = 1

    until fixed point:
        for (l, r) in congruence_edges:
            weights[l, r] = find_shortest_path(l, r, weights).size()
    return weights
```

TreeOpt algorithm

```
function get_explanation(start, end, weights=compute_weights()):
    let explanation = []
    let path = find_shortest_path(start, end, weights)
    for edge in path:
        if edge.is_congruence_edge():
            let (j1, j2) = edge.justification()
            explanation += get_explanation(j1, j2, weights)
        else:
            explanation += edge
    return explanation
```

GREEDY algorithm

- First, estimates the proof size of each congruence edge

- To explain the equivalence between two terms, finds the sortest path between them using these estimates as edge weights, but recurses when it encounters a congruence edge

GREEDY algorithm

```
function compute_weights():
    let weights = {}
    for edge in edges:
        if edge.is_congruence_edge():
            weights[l, r] = 1
        else:
            weights[l, r] = unoptimized_explanation(l, r).size()

    return weights
```

GREEDY algorithm

```
function get_explanation(start, end, weights=compute_weights(), fuel):
    if fuel == 0:
        return unoptimized_explanation(start, end)

    let explanation = []
    let path = find_shortest_path(start, end, weights)
    for edge in path:
        if edge.is_congruence_edge():
            let (j1, j2) = edge.justification()
            explanation += get_explanation(j1, j2, fuel - 1)
        else:
            explanation += edge

    return explanation
```

Dealing with backtracking

- Modern SMT solvers work by trying many possible (partial) solutions, and backtracking when a solution is determined to be invalid

- So, a congruence closure algorithm must be able to efficiently revert to a previous state when backtracking

Dealing with backtracking

- In our case, this is done by carefully recording the steps the congruence closure engine did, and undoing them when backtracking

- This includes clearing any cache that became invalid because of the backtrack (e.g., the TreeOpt and Greedy edge weights)

Avoiding circular explanations

- When we were discarding redundant equalities, there could never be a redundant explanation

- This is because, after a congruence edge is added, the path between the terms of its justification will not change anymore

Avoiding circular explanations

- When we were discarding redundant equalities, there could never be a redundant explanation
- This is because, after a congruence edge is added, the path between the terms of its justification will not change anymore
- Now that we keep redundant edges, this is not true anymore

Avoiding circular explanations: Example

- TODO

Avoiding circular explanations

- To prevent this problem, we store in each edge the *level* in which it was added

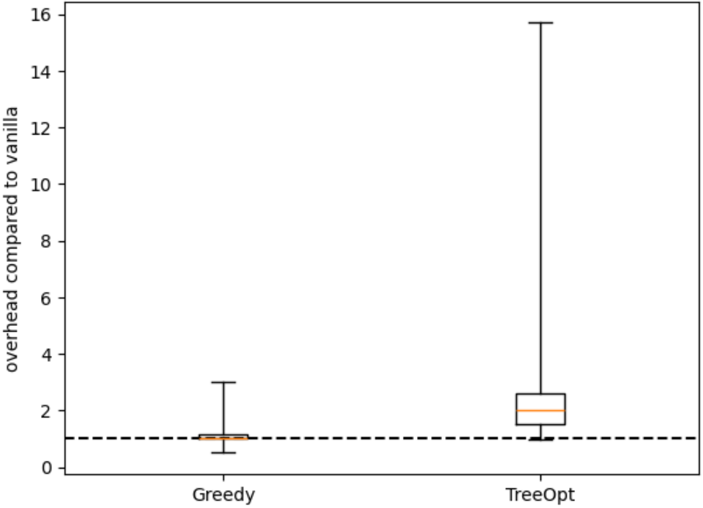
- Then, when explaining a congruence edge that was added in level n, we can only use edges whose level is no greater than n

Evaluation

- TODO

Results

- Average runtime overhead was 1.18x for GREEDY, 2.68x for TREEOPT



Results

- The new algorithms had no meaningful overall impact in the final proof size
- On average the proofs from GREEDY and TREEOPT were 0.2% and 0.1% larger that the VANILLA proofs
- They were 80% smaller than the VANILLA proofs in the best case, and 70% bigger in the worst case

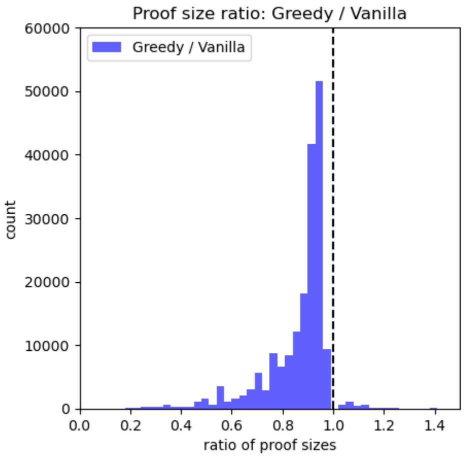
Results

- We also measured the size of the proof returned by each call to `get_explanation`
- In 83% of cases, the proofs returned by the three algorithms had the same size
- On average, the "local" proofs from the two new algorithms are 2% smaller than the VANILLA proofs

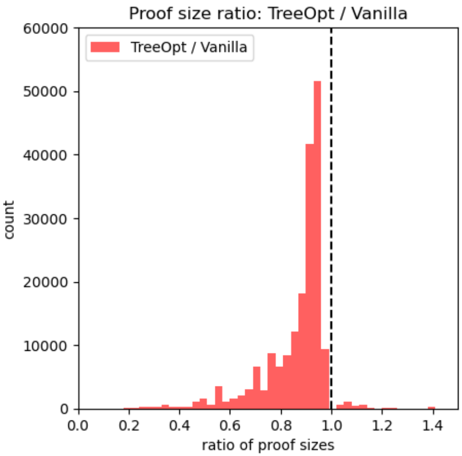
Results

- We also measured the size of the proof returned by each call to `get_explanation`
- In 83% of cases, the proofs returned by the three algorithms had the same size
- On average, the "local" proofs from the two new algorithms are 2% smaller than the VANILLA proofs
- If we exclude identical proofs, they are on average 14% smaller

Results



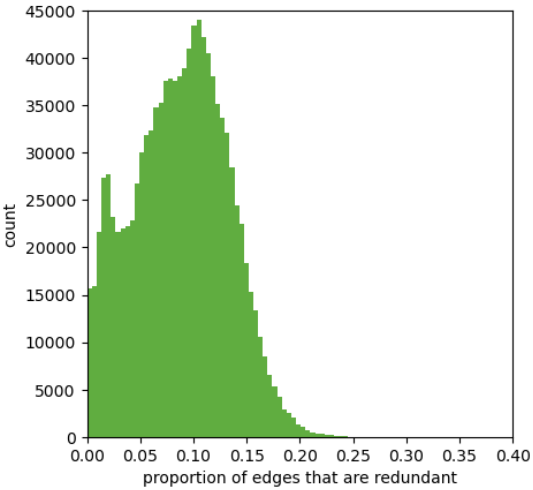
Proof size ratio: Greedy / Vanilla
Greedy / Vanilla
count
ratio of proof sizes
0
10000
20000
30000
40000
50000
60000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4



Proof size ratio: TreeOpt / Vanilla
TreeOpt / Vanilla
count
ratio of proof sizes
0
10000
20000
30000
40000
50000
60000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Results

- We also measured the number of redundant equalities added
- On average, 8.7% of the equality graph edges were redundant, and in the most extreme case, 44.63%



Conclusion

- TODO