PDPAR 2004

Union-Find and Congruence Closure Algorithms that Produce Proofs

Robert Nieuwenhuis   Albert Oliveras

Technical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain
`www.lsi.upc.es/~roberto|~oliveras`

Abstract

Congruence closure algorithms are nowadays central in many modern applications in automated deduction and verification, where it is frequently required to recover the set of merge operations that caused the equivalence of a given pair of terms. For this purpose we study, from the algorithmic point of view, the problem of extracting such small proofs.

Union-find data structures maintain the equivalence relation induced by a given sequence of *Union* operations between pairs of elements. Similarly, *congruence closure* algorithms maintain a *congruence* relation given by a sequence of pairs of *terms* (i.e., equations) without variables. The difference between equivalence closure and congruence closure is that the congruence relation, in addition to reflexivity, symmetry and transitivity, also satisfies the *monotonicity* axioms saying, for all f, that $f(x_1 \ldots x_n){=}f(y_1 \ldots y_n)$ whenever $x_i{=}y_i$ for all i in $1 \ldots n$.

Example 0.1 The equation $a{=}b$ belongs to the congruence generated by the three equations: $b{=}d$, $f(b){=}d$, and $f(d){=}a$. □

Decision procedures based on congruence closure are used in numerous deduction and verification systems, where the generation of *proof objects* is highly desirable if not required. For instance, this is crucial in the so-called *lazy* approaches to decision procedures for Boolean formulae over theory atoms. In these decision procedures, the Boolean formulae frequently include equality atoms; see, e.g., [dMR02,BDS02,FJOS03] and CVC, at `verify.stanford.edu/CVC`. These approaches are *lazy* in the sense that initially each equality atom is simply abstracted by considering it as a distinct propositional variable, and the resulting propositional formula is sent to a SAT solver. If the SAT solver returns a model that is not a congruence, an additional propositional clause (a *lemma*) precluding that model is added; this is iterated (*many* times) until the SAT solver finds a congruence model, or all assignments have been explored.

Example 0.2 Assume that, in such a lazy approach, the model being built by the SAT solver is fed into the congruence closure algorithm as a (long!) sequence of atoms that, in particular, includes $b = d$, $f(b) = d$, and $f(d) = a$. Then, if additionally $a \neq b$ is given, it is no longer a congruence (see Example 0.1). At that point, the congruence closure algorithm has to generate as a lemma the clause $b = d \wedge f(b) = d \wedge f(d) = a \longrightarrow a = b$, because the first three atoms are the *explanation* of $a = b$. It is hence crucial in these applications to efficiently recover small explanations among the (thousands of) merge operations that have taken place. □

Another recent approach for the flexible generation of decision procedures is given in [GHN+04]. It also heavily relies on incremental congruence closure with intermixed *explanation* operations. The basic idea is similar to the $CLP(X)$ scheme for constraint logic programming: to provide a clean and efficient integration of specialized theory solvers within the Davis-Putnam-Logemann-Loveland procedure [DLL62]. A general engine DPLL(X) is used, where X can be instantiated with a solver for a given theory T, thus producing a system DPLL(T). Each time the DPLL(T) procedure produces a conflict, explanations need to be generated by the theory solver for building the *conflict graph* that is used for *non-chronological backtracking* in modern SAT solvers such as Chaff [MMZ+01]. The fact that this approach currently outperforms previous techniques on logics with equality is largely due to the efficient algorithm for congruence closure with explanations described here (see [GHN+04] for details about the DPLL(T) approach and experiments).

We study from the algorithmic point of view the problem of efficiently recovering these explanations, showing that it can be done in quasi-optimal time $O(k\ \alpha(k,k))$ for a k-step explanation, without increasing the overall $O(n \log n)$ runtime of the fastest known congruence closure algorithms. As far as we know, this had not been done before, although several authors have addressed the problem of how to describe congruence closure proofs in different logical formats (see, e.g., [SD99]), and others have addressed union-find with different extensions such as backtracking.

References

[BDS02] Clarke Barrett, David Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation into sat. In *Procs. 14th CAV*, LNCS 2404, 2002.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Comm. of the ACM*, 5(7):394–397, 1962.

[dMR02] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Procs. 5th SAT Symp*, pages 244–251, 2002.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explanation. In *Procs. 15th CAV*, LNCS 2725, 2003.

[GHN^{+}04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Procs. CAV*, LNCS, 2004. To appear. Available at `www.lsi.upc.es/~oliveras`.

[MMZ^{+}01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automation Conference (DAC'01)*, 2001.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, 1999.