Big proof engines as little proof engines: new results on rewrite-based satisfiability procedures (Extended abstract)*

Alessandro Armando†
DIST
Università degli Studi di Genova, Italy

Maria Paola Bonacina‡
Dipartimento di Informatica
Università degli Studi di Verona, Italy

Silvio Ranise§
LORIA & INRIA-Lorraine
Nancy, France

Stephan Schulz¶
Dipartimento di Informatica
Università degli Studi di Verona, Italy

The application of automated reasoning to verification has long shown the importance of decision procedures for satisfiability in decidable theories. For instance, consider theories of data types, such as *arrays*, *lists*, *records*, for software and also hardware verification (e.g., arrays are used to model registers in the verification of pipelined microprocessors).

Much research in automated reasoning has revolved on two paradigms, that may be called *"big proof engines"* and *"little proof engines,"* respectively (e.g., recently [7, 8]). The "big engine" approach aims at procedures for the (semi-decidable) validity problem in full first-order logic (with equality), and applies them across all theories that can be presented in the logic. All formulæ are acceptable input. *Ordering-based methods* (e.g., ordered resolution, paramodulation/superposition, rewriting), *subgoal-reduction methods* (e.g., clausal normal-form tableaux, model elimination, Prolog technology theorem proving) and *instance-based methods* (e.g., hyperlinking, hypertableaux, disconnection method) all belong to this category. The "little engine" approach aims at procedures for specific decidable theories, by building each theory into a dedicated inference engine. Acceptable formulæ are severely restricted: a $\mathcal{T}$*-satisfiability procedure* decides the satisfiability of conjunctions of ground literals in theory $\mathcal{T}$. Basic examples are the congruence closure algorithm, for satisfiability of sets of ground equalities and inequalities, and procedures that build axioms of equational theories into the congruence closure algorithm.

Clearly, there has always been a continuum between big and little engines of proof. On the big engine side, much research has been devoted to *reasoning modulo a theory*, including methods that build axioms into first-order inference rules, unification and rewriting modulo an equational theory. On the little engine side, methods for the combination of theories have been investigated to build larger engines by combining little ones. Much work can be understood equally well from either the big or the little engine perspective.

*Supported in part by the Ministero per l'Istruzione, l'Università e la Ricerca (grant no. 2003-097383).

†Viale Causa 13, I-16145 Genova, Italy, `armando@dist.unige.it`

‡Strada Le Grazie 15, I-37134 Verona, Italy, `mariapaola.bonacina@univr.it`

§615 Rue du Jardin Botanique, F-54600 Villers-lès-Nancy, France, `Silvio.Ranise@loria.fr`

¶Strada Le Grazie 15, I-37134 Verona, Italy, `schulz@eprover.org`

This extended abstract reports on on-going research that aims at replacing the apparent dichotomy ("little engines *versus* big engines") by a cross-fertilization ("big engines *as* little engines"). The general idea is to explore how the technology of big engines (e.g., inference rules, search plans, algorithms, data structures, implementation techniques) may be applied selectively and efficiently "in the small." Even a cursory examination suggests how fruitful this might be under several aspects. First, most verification problems involve more than one theory, so that the little engine approach requires procedures for *combinations of theories*. Combination is complicated, and combining theories by combining concrete decision algorithms may lead to rather *ad hoc* procedures, that are hard to modify, extend, integrate into, or even interface with, other systems. Second, satisfiability procedures need to be proved correct and complete: a key part is to show that whenever the algorithm reports "satisfiable," its ouput represents a model of the input. Most model-construction arguments are specialized for concrete procedures, so that each new procedure requires a new proof. Third, although some systems offer some support for adding theories, a developer usually has to write a large amount of new code for each procedure, with little software reuse and high risk of errors.

If one could use first-order theorem-proving strategies, combination might become much simpler, because in several cases it would be sufficient to give as input the union of the presentations of the theories. No *ad hoc* correctness and completeness proofs would be needed, because a sound and complete strategy is a *semi-decision procedure* for unsatisfiability. Existing first-order provers, that embody the results of years of research on data structures and algorithms for deduction, could represent a repository for code reuse. The crux is *termination*: in order to have a *decision procedure*, one needs to prove that a complete theorem-proving strategy is bounded to terminate on satisfiability problems in the theories of interest. It was shown in [1] that a standard, refutationally complete, *rewrite-based inference system*, named $\mathcal{SP}$ from superposition, is guaranteed to terminate on satisfiability problems in the theories of *ground equality*[1], *lists*, *arrays with and without extensionality*, *sets with extensionality*, and the combination of lists and arrays.

In this extended abstract, we summarize work in progress on *rewrite-based satisfiability procedures*, that includes: (1) new termination results extending the approach of [1] to more theories and combinations thereof; (2) experimental results showing that, against expectations, it is not obvious that validity checkers with the theories built-in are much faster than theorem provers that take theory presentations as input.

1 New termination results for rewrite-based satisfiability

A fundamental feature of inference systems such as $\mathcal{SP}$ (superposition, paramodulation, reflection, equational factoring, subsumption, simplification and tautology deletion [1]) is the usage of a *complete simplification ordering (CSO)* $\succ$ on terms and atoms to restrict inferences. We write $\mathcal{SP}_{\succ}$ for $\mathcal{SP}$ with a given $\succ$, and call $\mathcal{SP}_{\succ}$*-strategy* any theorem-proving strategy (inference system + search plan) with inference system $\mathcal{SP}_{\succ}$. The rewrite-based methodology for satisfiability procedures consists of the following phases:

[1]Also known as quantifier-free theory of equality, or *EUF* for Equality with Un-interpreted Function symbols.

1. $\mathcal{T}$*-reduction:* depending on the theory $\mathcal{T}$, this step consists of removing certain literals or symbols in such a way to obtain an equisatisfiable $\mathcal{T}$*-reduced* problem.

2. *Flattening:* all ground literals are flattened by introducing new constants, yielding an equisatisfiable $\mathcal{T}$-reduced *flat* problem.

3. *Ordering selection and termination:* any fair $\mathcal{SP}_\succ$-strategy is shown to terminate when applied to a $\mathcal{T}$-reduced flat problem, provided $\succ$ is "good" for $\mathcal{T}$, or $\mathcal{T}$*-good*.

An $\mathcal{SP}_\succ$-strategy is $\mathcal{T}$-good if $\succ$ is. $\mathcal{T}$-reduction is based on standard properties such as those of skolemization. For all theories mentioned in the following except arrays (for which we refer to [1]), a CSO $\succ$ is $\mathcal{T}$*-good* if $t \succ c$ for all ground compound terms t and constants c. Most orderings satisfy this requirement. The following theorem summarizes our results:

Theorem 1 *A fair $\mathcal{T}$-good $\mathcal{SP}_\succ$-strategy is a satisfiability procedure for $\mathcal{T}$ being any of the following theories: the theory of* records with or without extensionality*, the theory of* integer offsets*, the theory of* integer offsets modulo*, and any combination of any of the above with the theory of* arrays (with or without extensionality) *and the* quantifier-free theory of equality.

The proofs and all technical details can be found in the full version of the paper. The axioms for arrays appear in [1]. Records, $\textsc{rec}(id_1 : \textsc{t}_1, \ldots, id_n : \textsc{t}_n)$, abbreviated $\textsc{rec}$, associating an element of type $\textsc{t}_i$ to the field identifier id_i, for $1 \leq i \leq n$, are axiomatized by

$$\forall x, v. \quad \text{rselect}_i(\text{rstore}_i(x, v)) \simeq v \quad \text{for all } i,\ 1 \leq i \leq n \tag{1}$$

$$\forall x, v. \quad \text{rselect}_j(\text{rstore}_i(x, v)) \simeq \text{rselect}_j(x) \quad \text{for all } i, j,\ 1 \leq i \neq j \leq n \tag{2}$$

$$\forall x, y. \quad (\textstyle\bigwedge_{i=1}^{n} \text{rselect}_i(x) \simeq \text{rselect}_i(y) \supset x \simeq y) \quad \text{(extensionality)} \tag{3}$$

with x, y variables of sort $\textsc{rec}$ and v of sort $\textsc{t}_i$. The *theory of integer offsets* is presented by:

$$\forall x. \quad \text{s}(\text{p}(x)) \simeq x \tag{4}$$

$$\forall x. \quad \text{p}(\text{s}(x)) \simeq x \tag{5}$$

$$\forall x. \quad \text{s}^i(x) \not\simeq x \quad \text{for } i > 0 \tag{6}$$

where s and p are successor and predecessor. A presentation of the *theory of integer-offsets modulo* (modulo k, for $k > 0$) is obtained from the above by replacing (6) with:

$$\forall x. \quad \text{s}^i(x) \not\simeq x \quad \text{for } 1 \leq i \leq k-1 \tag{7}$$

$$\forall x. \quad \text{s}^k(x) \simeq x \tag{8}$$

2 An experimental appraisal

We designed six sets of *synthetic benchmarks*, on *arrays with extensionality* (`STORECOMM`, `SWAP`, `STOREINV`), the combination of *arrays* and *integer offsets* (`IOS`), the combination of *arrays*, *records* and *integer offsets* (`QUEUE`), and the combination of *arrays*, *records*, and *integer offsets modulo*

(CIRCULAR_QUEUE). The latter two exploit the observation that *queues* can be modelled as records, uniting a partially filled array with two indices (head and tail), and *circular queues* of length k are queues whose indices take integer values modulo k. These sets are designed to assess *scalability*, because they are *parametric* with respect to a parameter n, that determines the size of the instances[2]. To complete our appraisal with "real-world" problems that test performance on huge sets of literals, we used haRVey[3] to extract $\mathcal{T}$-satisfiability problems from the UCLID suite [5]. Over 55,000 proof tasks in the combination of integer offsets and ground equality were generated. We conducted experiments with the theorem prover E [6], that implements $\mathcal{SP}$, the validity checker CVC [9] and its successor CVC Lite [2], that are the only state-of-the-art tools implementing a complete procedure for arrays with extensionality[4]. In the following, *native input* means flattened, T-reduced files for E, and plain, unflattened files for CVC and CVC Lite.

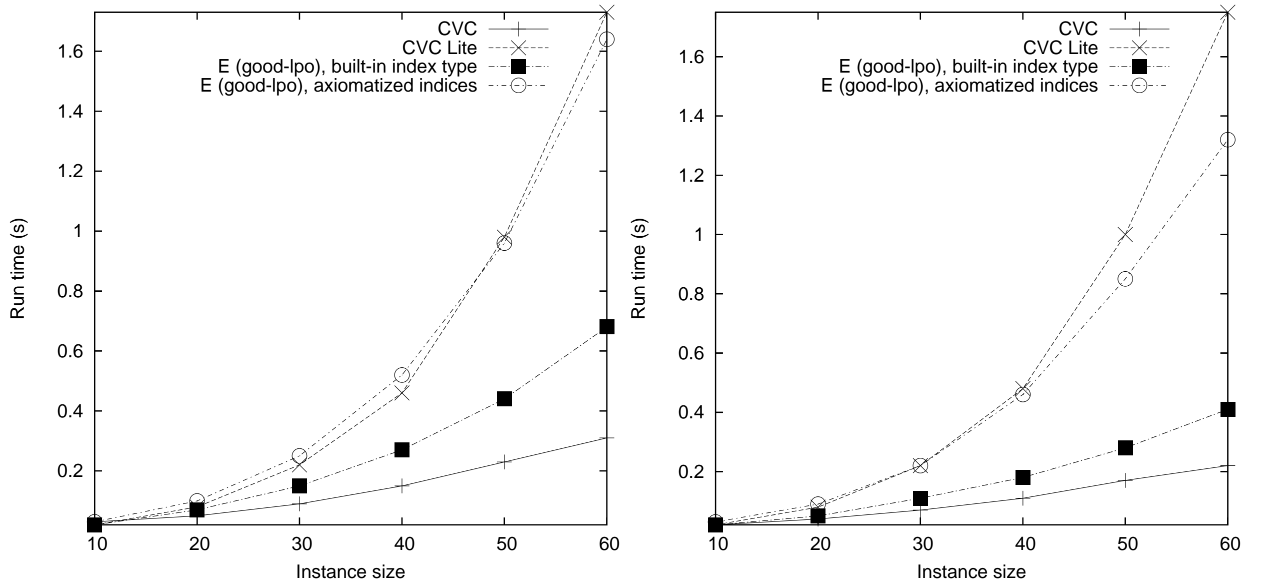

Figure 1: Performance on valid (left) and invalid (right) STORECOMM instances with native input.

In Fig. 1, "good-lpo" means a $\mathcal{T}$-good lexicographic recursive path ordering, where $\mathcal{T}$ is the theory of arrays, while "built-in index type" refers to a feature of E that allows it to recognize that all constants of sort index occurring in the problem are distinct: "axiomatized indices" refers to runs where this feature is not used and the disequalities stating that all indices are distinct are included in the input. On valid instances, *E(good-lpo)* with axiomatized indices and CVC Lite show nearly the same performance, with E slightly ahead in the limit. *E(good-lpo)* with built-in indices outperforms CVC Lite by a factor of about 2.5. CVC performs best, but it is surprising that E, which is optimized for *unsatisfiability*, performs better on invalid (i.e., satisfiable) instances, where *E(good-lpo)* with built-in indices comes closer to CVC. If *all* systems run on flattened input (Fig. 2), CVC and E with built-in indices are fastest: on valid instances, their performances are

[2]The problems are available upon request and will soon be submitted to TPTP `http://www.tptp.org/` or SMT-Lib `http://combination.cs.uiowa.edu/smtlib/`.

[3]`http://www.loria.fr/equipes/cassis/softwares/haRVey/`

[4]To our knowledge, neither Simplify [4] nor ICS [3] are complete in this regard: cfr. Sec. 5 in [4] and a personal communication from H. Rueß to A. Armando in April 2004.

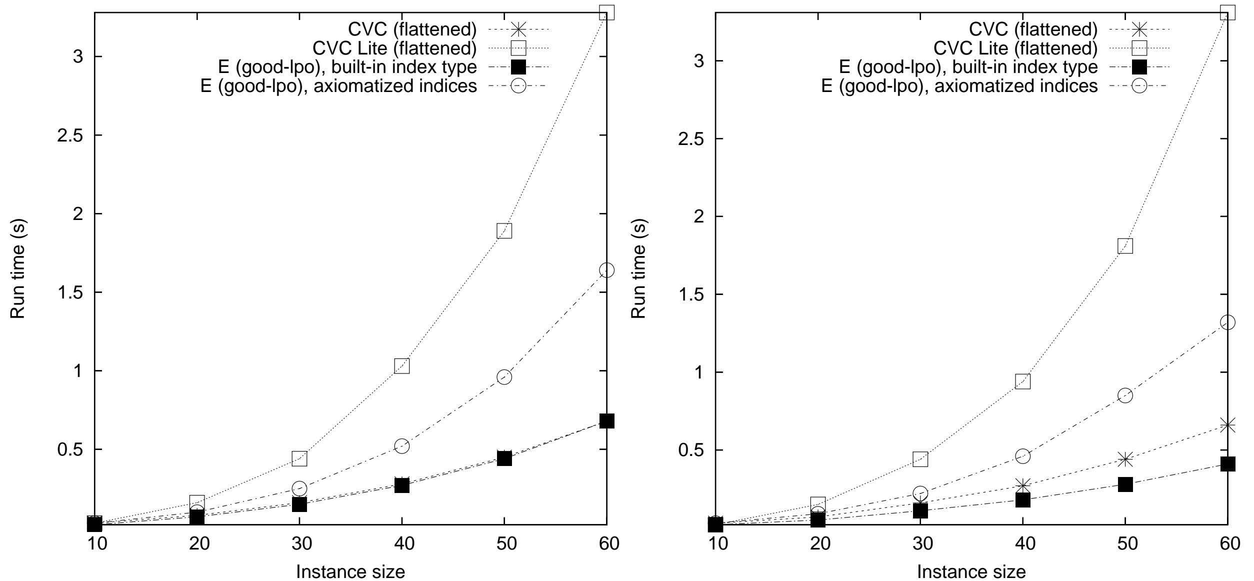

Figure 2: Performance on valid (left) and invalid (right) STORECOMM instances with flat input.

so close, that the plots are barely separable, but E leads on invalid instances.

As shown in Fig. 3 (left), the systems are very close up to instance size 5 for SWAP. Beyond this point, E leads up to size 7, but then is overtaken by CVC and CVC Lite. E can solve instances of size 8, but is much slower than CVC and CVC Lite, which solve instances up to size 9. No system can solve instances of size 10. For invalid instances, E solves easily instances up to size 10 in less than 0.5 sec, while CVC and CVC Lite are slower, taking 2 sec and 4 sec, respectively, and showing worse asymptotic behaviour. Fig. 4 displays performances on valid instances, when the input for E includes an additional lemma. Although this addition means that the theorem prover is no longer a decision procedure, the termination behavior in practice improves, as E solves instances of size 9 and 10, and suggests a better asymptotic behaviour.

The comparison becomes even more favorable for the prover on STOREINV (Fig. 5). CVC solves valid instances up to size 8, CVC Lite goes up to size 9, but E solves instances of size 10, the largest generated. A comparison of run times at size 8 (the largest solved by all systems), gives 3.4 sec for E, 11 sec for CVC Lite, and 70 sec for CVC. For invalid instances, E does not do as well, but the run times here are minimal (e.g., the largest, for instance size 10, is about 0.1 sec).

The IOS problems are encoded for CVC and CVC Lite by using their built-in linear arithmetic (on the reals for CVC and on the integers for CVC Lite). We tried to use inductive types in CVC, but it performed badly and even reported incorrect results[5]. In terms of performance (Fig. 6), CVC clearly wins, as expected from a system with built-in arithmetic. *E(good-lpo)* is no match, although it solves all tried instances (Fig. 6, left). *E(std-kbo)* features a standard Knuth-Bendix ordering for first-order theorem proving and the same search plan of *E(good-lpo)*, except for privileging ground clauses and not preferring input clauses. *E(std-kbo)* does much better,

[5]This is a known bug, that will not be fixed since CVC is no longer supported (personal communication from A. Stump to A. Armando, February 2005). CVC Lite 1.1.0 does not support inductive types.

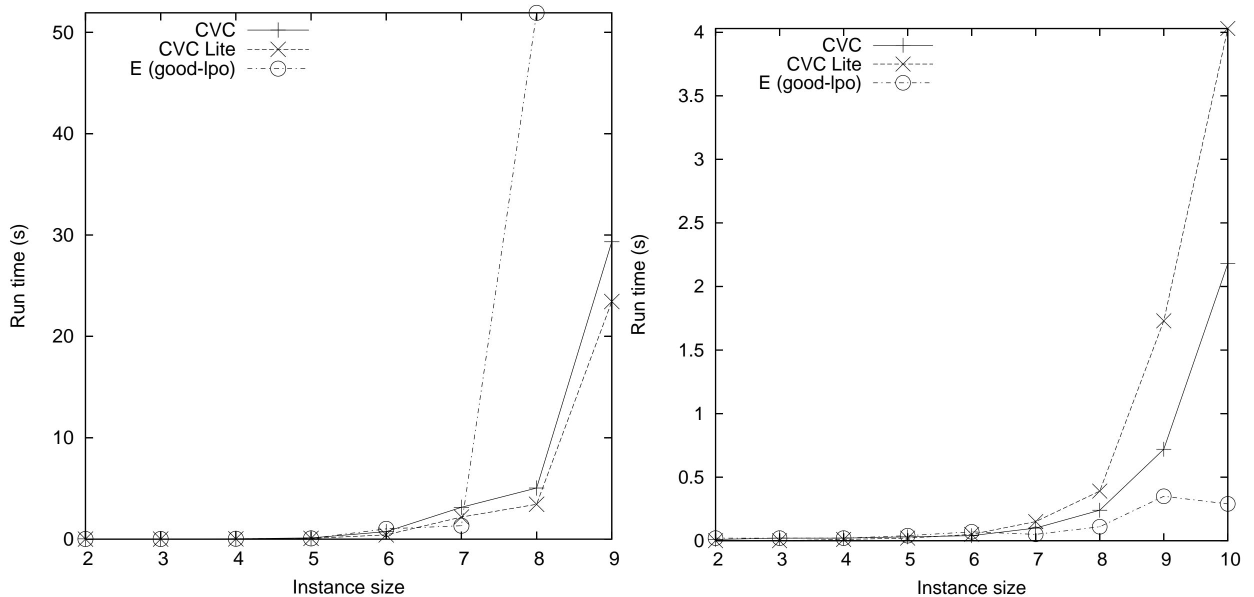

Figure 3: Performance on valid (left) and invalid (right) `SWAP` instances, native input.

probably because it does not consider the axioms early. It also does better than CVC Lite: it scales smoothly, while CVC Lite displays oscillating run times, showing worse performance for even instance sizes than for odd ones (Fig. 6, right).

Fig. 7 (left) shows performances on plain queues: as expected, CVC is the fastest system, but *E(good-lpo)* competes very well with the systems with built-in linear arithmetic. Fig. 7 (right) does not include CVC, because CVC cannot handle the *modulo-k* integer arithmetic required for circular queues. Between CVC Lite and E, the latter demonstrates a clear superiority: it shows nearly linear performance, and proves the largest instance in less than 0.5 sec, nine times faster than CVC Lite.

E in automatic mode, where the prover chooses automatically ordering and search plan, solved *all* the problems from UCLID, taking less than 4 sec on the hardest one, with average 0.372 sec and median 0.25 sec. Fig. 8 shows an histogram of run times: the vast majority of problems is solved in less than 1 sec and very few need between 1.5 and 3 sec. An optimized search plan was found by testing on a random sample of 500 problems, or less than 1% of the full set. With this search plan, very similar to *E(std-kbo)*, the performance improved by about 40% (Fig. 8, right): the average is 0.249 sec, the median 0.12 sec, the longest time 2.77 sec, and the vast majority of tasks is solved in less than 0.5 sec. We expect these problems to be easy also for UCLID, and we present them as evidence that a prover can meet the expectation one has for a verification tool.

3 Discussion

Our experiments show that the prover terminates in many cases beyond the known termination results: e.g., Fig. 4, and the runs with *E(std-kbo)*, whose ordering is not $\mathcal{T}$-good, if $\mathcal{T}$ includes arrays. Thus, theorem provers are not as brittle as one may fear with respect to termination,

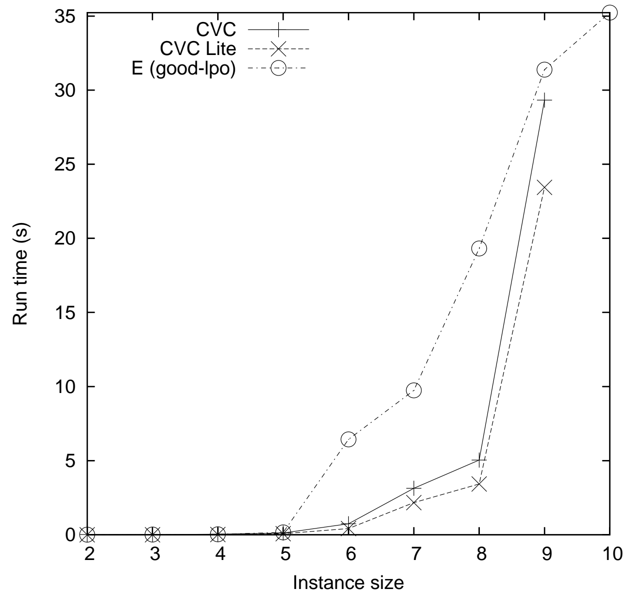

Figure 4: Performance on valid `SWAP` instances with added lemma for E.

and offer the flexibility of adding useful lemmata to the presentation. Also, these problems are very different from those theorem provers are optimized for (e.g., many axioms, many universal variables), which suggests there is further room for improvement.

References

[1] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Inform. and Comput.*, 183(2):140–164, 2003.

[2] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *CAV-16*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[3] L. de Moura, S. Owre, H. Rueß, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In D. Basin and M. Rusinowitch, editors, *IJCAR-2*, volume 3097 of *LNAI*, pages 218–222. Springer, 2004.

[4] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL–2003–148, HP Labs, 2003.

[5] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. A. Peled, editors, *CAV-16*, volume 3114 of *LNCS*, pages 475–378. Springer, 2004.

[6] S. Schulz. E – a brainiac theorem prover. *J. of AI Commun.*, 15(2–3):111–126, 2002.

[7] N. Shankar. Little engines of proof, 2002. Invited talk, 3rd FLoC, Copenhagen, Denmark; and course notes of Fall 2003, `http://www.csl.sri.com/users/shankar/LEP.html`.

[8] M. E. Stickel. Herbrand Award speech, 2002. 3rd FLoC, Copenhagen, Denmark.

[9] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a Cooperating Validity Checker. In K. G. Larsen and E. Brinksma, editors, *CAV-14*, LNCS. Springer, 2002.

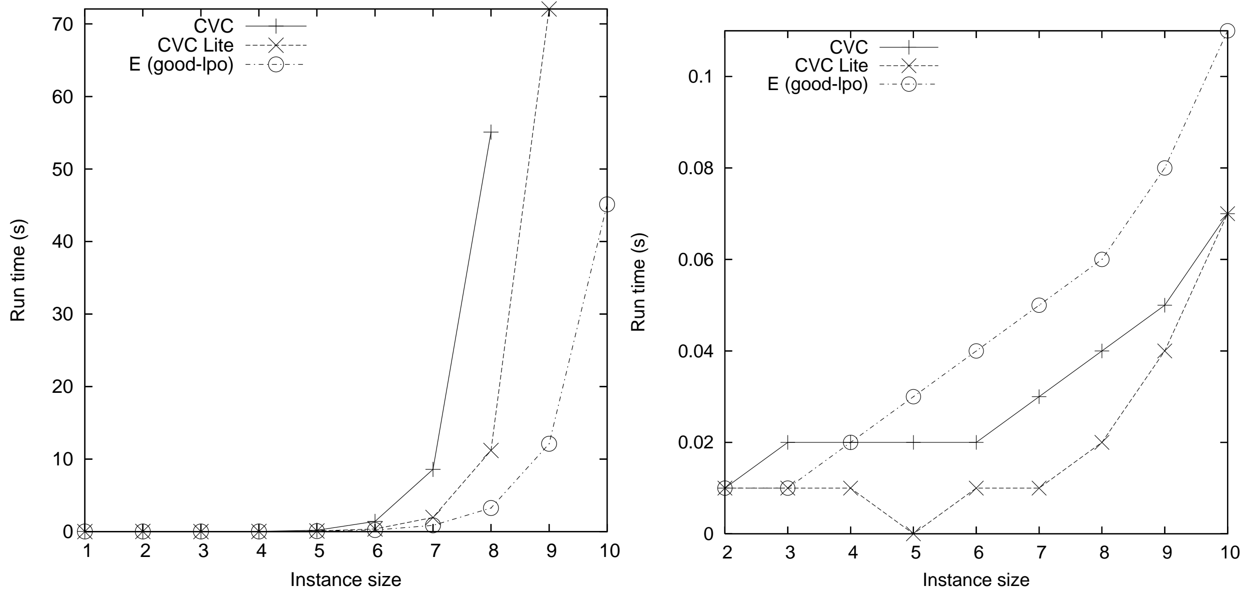

Figure 5: Performance on valid (left) and invalid (right) `STOREINV` instances with native input.

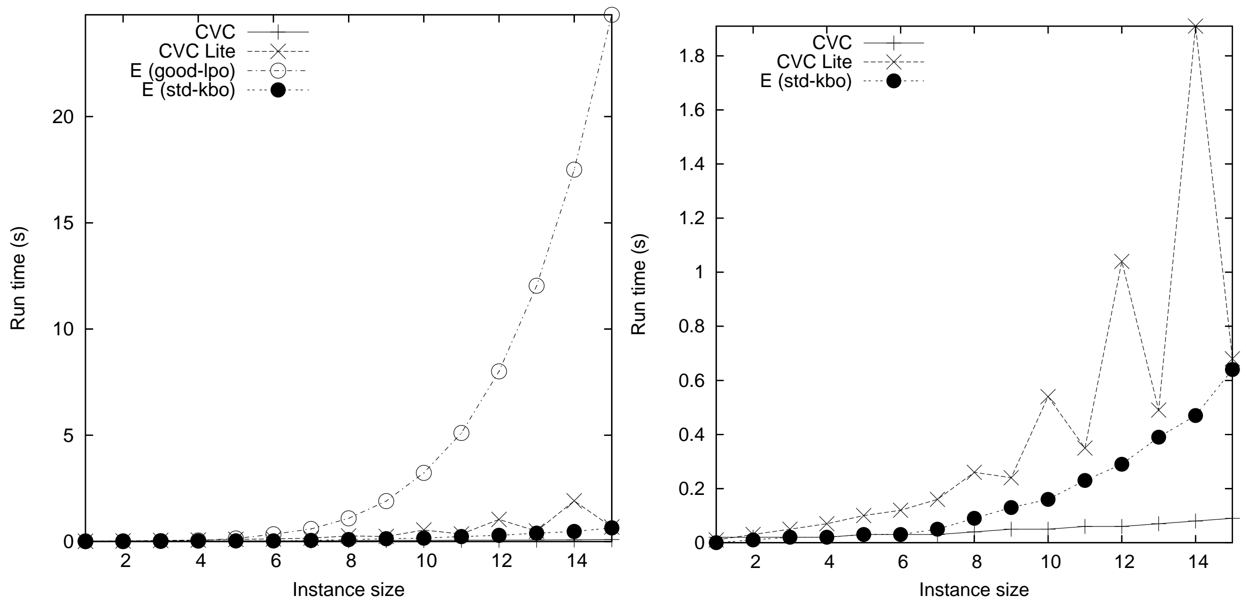

Figure 6: Performance on `IOS` instances: the graph on the right shows a rescaled version, including only the three fastest systems, of the same data on the left.

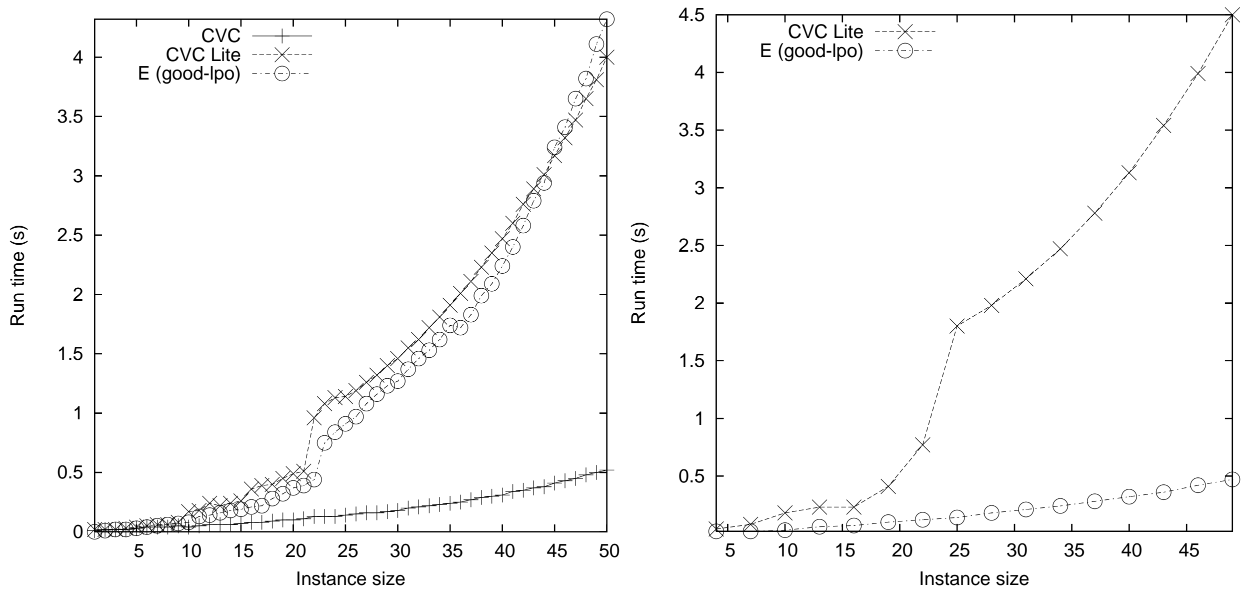

Figure 7: Performance of E, CVC and CVC Light on `QUEUE` (left) and `CIRCULAR_QUEUE` (right) with $k = 3$.

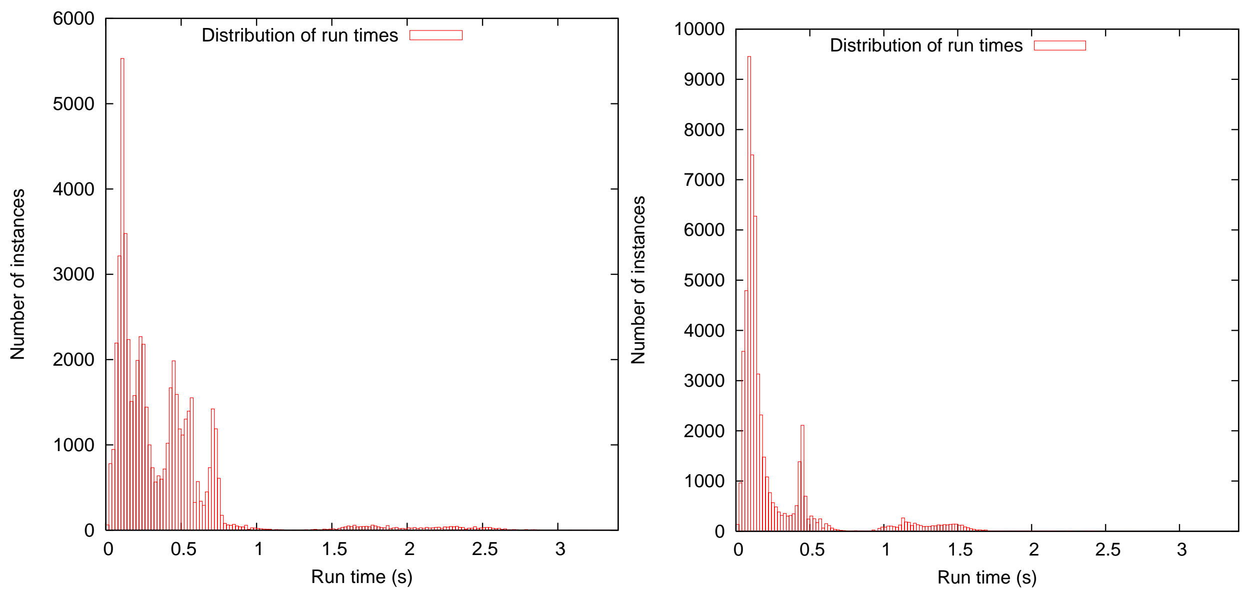

Figure 8: Distribution of E's run times in automatic mode (left) and with an optimized strategy (right) on the problems from UCLID (all valid).