Benchmarks from Reduction Finding

Charles Jordan[1⋆] and Łukasz Kaiser[2]

[1] ERATO Minato Project, JST & Hokkaido University
[2] LIAFA, CNRS & Université Paris Diderot
skip@ist.hokudai.ac.jp, kaiser@liafa.univ-paris-diderot.fr

1 Introduction

Consider the problem of searching for a complexity-theoretic reduction between two decision problems, P and Q. Such a reduction is a function r satisfying $x \in P \iff r(x) \in Q$ for all structures x for which it makes sense to consider P. It is not difficult to show that determining the existence of reductions is undecidable in general. However, sufficiently restricting the class of reductions and structures considered results in a simpler, decidable variant that is still meaningful from a complexity-theory perspective.

Once we can encode these reductions and structures in a finite number of bits, the problem becomes a QBF instance with one quantifier alternation (2QBF):

$$\exists r \; \forall x \; (x \in P \iff r(x) \in Q)\,. \tag{1}$$

Therefore, reduction finding in this setting is essentially a Σ_2^p problem and it is possible to apply QBF solvers and ASP solvers supporting disjunctive logic programs. In this paper, we introduce various reduction-finding instances intended for benchmarking QBF solvers. For additional background, details and comparisons with other approaches, see [3]. Automatic reduction finding was first considered by [1]. More generally, reduction finding is a form of program synthesis, and similar approaches can be used more widely, see [4].

2 Background

Although polynomial-time reductions are perhaps the most traditional, we focus on a weaker class of quantifier-free logical reductions used in descriptive complexity. We do not introduce descriptive complexity and the full motivation for this class here, see [3] for the definitions we use and [2] for additional material.

The weaker class of logical reduction we consider has a number of advantages. First, the logical formulas defining the reduction make the explicit construction of (1) reasonable. Although the class of reductions we consider is extremely limited, it still suffices to characterize complexity classes and complete problems generally remain complete under these reductions. Proving that a computation

⋆ Supported in part by KAKENHI No. 25106501.

cannot be done in polynomial-time is notoriously difficult, while there are techniques for proving that something cannot be done in, e.g., first-order logic. So our reductions are strong enough that they suffice to determine the relationship between complexity classes, but weak enough to hope for progress.

2.1 Parameters

An instance of a reduction-finding problem is determined by several parameters, which thus determine the difficulty of the underlying problem. First, we must fix the two decision problems of interest: P and Q in (1). Our system supports defining these problems in first-order logic extended by a reflexive transitive-closure (TC) operator, corresponding to the complexity class NL.

The choice of properties is important: choosing, e.g., the trivial properties *always true* and *always false* results in an easy instance that is not interesting. From the complexity-theoretic perspective, the hardness of properties defined in first-order (without TC) is fairly well-understood. Therefore, formulas using TC are most interesting and seem significantly harder.

Next, there are several parameters that determine the class of reductions being considered. Several parameters determine which atomic formulas are allowed to occur in the reduction. In particular, these parameters control whether successor is allowed between variables ($x = y + 1$), and whether certain numeric constants ($\min, \max$) are available. Then, there is a parameter k fixing the *dimension* of the reduction (informally, the dimension determines how much larger the output of a reduction can be compared to the original structure). Our reductions are defined by tuples of formulas in DNF – there is also a parameter c determining how many conjunctions may occur in the DNF.

Finally, there is a parameter n determining the size of structure considered in (1); we search only[3] for reductions that are correct on structures of size n.

Given these parameters, our generator constructs the corresponding QBF instance of (1) in several formats: qdimacs, negated qdimacs (to avoid an additional quantifier alternation from CNF conversion), qpro and lparse. Increasing c provides additional power to the class of reductions (instances become more positive), and increasing k is similar. Increasing n usually places more restrictions on the reduction (instances become more negative). A balance between parameters is important: large c with small n is unlikely to be meaningful because the reduction is too powerful for the small class of structures.

3 Instances

Let us describe how we instantiate the parameters described above in practice, and present some experimental results on a few parameter sets.

For the two decision problems of interest, P and Q in (1), we use one of 48 properties that were translated from the first paper on automatic reduction

[3] We support searching for reductions correct on a range of sizes $n_1 \leq n_2$ when using CEGAR, see [3].

finding [1] to allow to compare performance to their system (c.f. [3]). The names we use for the problems correspond to the names used in the system from [1], and are always 6 characters long. Below, we give a list describing these 48 problems, to give an overview what is included.

- `trivial`, `ntrivil`, `trueque`, `falsequ`, `query15`, `query30` (trivial or almost trivial problems that we include for correctness testing)
- `reachqu` (reachability, NL-complete), `nreachq` (negated reachability, coNL-complete), `query10` (symmetric reachability, SL-complete),
- `query06`, `query31`, `query33`, `query34`, `query36` (L-complete), and `query71` (negated query06, coL-complete),
- `query42`, `query44` (non-reflexive reachqu/nreachq, NL/coNL-complete),
- `query48`, `query49` (non-reflexive SL/coSL-complete),
- `query54` (strongly connected, NL-complete), `query55` (symmetric query54),
- `query57`, `query58`, `query60`, `query64` (minor variations of the above),
- the others[4] are defined in plain FO and thus correspond to AC^0 problems

Above, we mention the classes coNL, coL, SL and coSL, however, NL=coNL and L=coL=SL=coSL. Instances corresponding to discovering these famous collapses are interesting and appear hardest among the instances we consider.

We use the largest class of reductions supported: we allow min and max and successor in the atoms (`-min -max -succ`) and also the use of numbers when defining constants (`-nbrs`). As for the dimension k (`-dim`), number of conjunctions c (`-cls`), and structure size n (`-elems`), the most basic set of parameters we used was $k = 1, c = 1, n = 3$. Our generator is `ReductionTest.native`[5] and the following command can be used to generate the `qdimacs` file for the above parameters and $P =$ `nreachq`, $Q =$ `reachqu`.

```
./ReductionTest.native -min -max -succ -nbrs
  -from nreachq -to reachqu -dim 1 -cls 1 -elems 3 -qdimacs
```

It is also possible to directly generate files for all problems we consider for a given parameter set using the `-gen [directory]` option.

```
./ReductionTest.native -min -max -succ -nbrs
  -dim 1 -cls 1 -elems 3 -gen .
```

In addition to generating files, the generator can test the existence of a reduction. This is done with CEGAR using a specified SAT-solver (at present `-minisat`, `-glueminisat` or the standard solver from Intel's Decision Procedure Toolkit). For example, one can check the existence of a reduction from $P =$ `nreachq` to $Q =$ `reachqu` with the specified parameters as follows.

[4] `exquery`, `eequery`, `nxquery`, `axquery`, `query01`, `query02`, `query03`, `query04`, `query05`, `query07`, `query08`, `query09`, `query11`, `query21`, `query23`, `query24`, `query25`, `query26`, `query27`, `query23`, `query45`, `query50`, `query51`, `query52`

[5] The generator with instructions and the collection of generated files we used for testing are available from `http://toss.sf.net/reductGen.html`.

```
./ReductionTest.native -min -max -succ -nbrs
  -from nreachq -to reachqu -dim 1 -cls 1 -elems 3 -glueminisat
```

From the 48 properties included, the method above can generate $48 \times 48 = 2304$ `qdimacs` files for each parameter set. To avoid an extra quantifier alternation due to CNF conversion in `qdimacs`, we also tested negating the QBF formula before CNF conversion – but results were worse than with the added alternation (c.f. [3]). To give an intuition about the hardness of the generated QBF instances, we present below the results from [3] showing the number of timeouts on these 2304 instances for 5 QBF solvers. The tests were performed on an Opteron 1385 (using 1 core) and with a timeout of 120s.

	$c=1\ n=3$	$c=2\ n=3$	$c=3\ n=3$	$c=1\ n=4$	$c=2\ n=4$	$c=3\ n=4$
RAREQS	0	0	16	19	65	204
DEPQBF	0	142	547	16	297	711
QUBE	10	536	949	82	760	1082
CIRQIT	58	673	1138	511	1092	1357
SKIZZO	522	1058	1156	975	1327	1434

On our instances, RAREQS, a recently introduced CEGAR solver, performed best. For non-CEGAR solvers, DEPQBF and QUBE outperform SKIZZO and CIRQIT. Between DEPQBF and QUBE the situation is less clear, some instances work much better with one of these solvers, others with the other. The comparison between SKIZZO and CIRQIT is difficult as well. As to the dominance of DEPQBF and QUBE over SKIZZO and CIRQIT, it holds for almost all queries. Still, there are a few outliers such as the reduction from `query26` to `query01`, on which DEPQBF and QUBE time out, but SKIZZO answers almost immediately.

Of course, our generator is not restricted to these problems – any problem in NL can be defined in first-order logic with transitive closure, and these formulas can be directly used with the `-from` and `-to` options.

Although some interesting reductions can be expressed with dimension 1, usually this does not suffice. However, even parameters such as $k=2, c=1, n=4$ or $k=3, c=1, n=3$ seem to produce very hard instances. Instances with $k>1$ where the properties use transitive-closure are often hard, but achieving better performance on such instances is an important step to finding *new* reductions and hopefully new complexity-theoretic knowledge.

References

1. Crouch, M., Immerman, N., Moss, J.E.B.: Finding reductions automatically. In: Fields of Logic and Computation – Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday. LNCS, vol. 6300, pp. 181–200. Springer (2010)
2. Immerman, N.: Descriptive Complexity. Springer-Verlag (1999)
3. Jordan, C., Kaiser, Ł.: Experiments with reduction finding. In: Proc. SAT 2013
4. Jordan, C., Kaiser, Ł.: Learning programs as logical queries. In: LTC 2013 (2013)