Experiments with 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`

Abstract. Reductions are perhaps the most useful tool in complexity theory and, naturally, it is in general undecidable to determine whether a reduction exists between two given decision problems. However, asking for a reduction on inputs of bounded size is essentially a Σ_2^p problem and can in principle be solved by ASP, QBF, or by iterated calls to SAT solvers. We describe our experiences developing and benchmarking automatic reduction finders. We created a dedicated reduction finder that does counter-example guided abstraction refinement by iteratively calling either a SAT solver or BDD package. We benchmark its performance with different SAT solvers and report the tradeoffs between the SAT and BDD approaches. Further, we compare this reduction finder with the direct approach using a number of QBF and ASP solvers. We describe the tradeoffs between the QBF and ASP approaches and show which solvers perform best on our Σ_2^p instances. It turns out that even state-of-the-art solvers leave a large room for improvement on problems of this kind. We thus provide our instances as a benchmark for future work on Σ_2^p solvers.

1 Introduction

Finding reductions between different decision problems is a central task in complexity theory. Polynomial-time reductions are perhaps the most traditional, and constructing such reductions generally involves creating certain gadgets or building some other form of structure on top of the instance that is to be reduced. The intuition that finding reductions resembles structured constructions can be captured formally: the reduction one finds is usually not only a polynomial-time function, but often a log-space one, or even a quantifier-free projection.

The class of quantifier-free projections, defined formally in the next section, is a very restricted subset of log-space functions. Still, they are sufficient to capture important complexity classes (see Chapter 11 of [13]). For example[3], P=NP iff SAT $\leq_{\text{qfp}}$CVP, and NL=NP iff SAT $\leq_{\text{qfp}}$REACH. The hope when focusing on weaker (but still sufficiently strong) reductions is that they will put new complexity-theoretic results within reach, and there are examples where

⋆ Supported in part by KAKENHI No. 25106501.

[3] We write X $\leq_{\text{qfp}}$Y if there is a quantifier-free projection from X to Y. CVP is the P-complete Circuit Value Problem, REACH is the NL-complete problem of directed reachability, and SAT is the NP-complete propositional satisfiability problem.

this has actually been accomplished [1]. Quantifier-free projections also have a significant advantage when trying to derive them automatically. They are by definition formulas of a simple form, so one can enumerate them easily once their dimension is fixed. In fact, instead of enumerating, one can write them in symbolic form using propositional variables. This opens the way for using propositional solvers to find such reductions automatically.

The problem of determining whether a quantifier-free projection exists between two given decision problems is still undecidable in general. But when we fix the dimension of the reduction we are looking for and only ask for it to be correct on inputs of bounded size, the question becomes essentially a Σ_2^p problem – it is of the form $\exists \overline{X}\ \forall \overline{Y}\ \varphi$ where $\overline{X}$ and $\overline{Y}$ are sets of propositional variables and φ is a quantifier-free propositional formula. This problem can then in principle be solved by a QBF or ASP solver, or by iterated calls to a SAT solver.

This paper describes our experiments with this kind of automated reduction finding. We present both a dedicated reduction finder called DE[4] and a generator[5] that allows to construct instances for QBF and ASP solvers that are equivalent to the given reduction finding problem. It is therefore a source of instances for which the hardness depends on the chosen parameters. To make it easy to use these problems for benchmarking, we provide both the generator (all source code is available as open-source) and the collection of `qdimacs`, `qpro`, and `lparse` files for the set of parameters we used in our experiments.[5]

In the long term, automatic reduction finders may help obtain unexpected complexity-theoretic results or re-discover stunning reductions. For example, the coNL-to-NL reduction behind the Immerman-Szelepcsényi Theorem, awarded the Gödel Prize in 1995, is in fact a dimension-8 quantifier-free projection and can in principle be found by DE. But current solvers do not perform sufficiently on high-dimensional instances: even dimension 3 is beyond reach of DE or any other solver with the present approach. Still, none of these solvers has been tuned for Σ_2^p problems and DE is a young project. We believe that our benchmarks are a source of meaningful, challenging SAT and 2QBF instances and we will work to include them in the next SAT and QBF evaluations. If solvers can be tuned to perform well on these kinds of instances, and improve their performance on Σ_2^p problems in general, we may be able to obtain interesting complexity-theoretic results in this way – so we encourage the community to experiment.

Related work. The idea of automatic reduction finding, together with the first automated ReductionFinder, was developed in [3]. ReductionFinder works on a database of decision problems specified in stratified Datalog and attempts to place the problems into classes based on the existence of reductions. It uses the ASP solver CMODELS to search for reductions, and it has not previously been compared to other reduction-finding attempts nor is it publicly available. We focus entirely on the problem of finding a reduction between two given problems,

[4] DE is available at `http://www-alg.ist.hokudai.ac.jp/~skip/de`

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

and thanks to a private copy of ReductionFinder we also compare our results to this previous approach.

2 Background in Descriptive Complexity

Classically, one defines a decision problem as a set of words and the complexity of a problem as the amount of computational resources (time, space) required to check on a Turing machine whether a word belongs to the set. In descriptive complexity, we take a higher-level view of decision problems. Instances do not need to be encoded as words, but are directly relational structures, for example graphs. The role of a Turing machine is in turn played by a formula in some logic, and the complexity of a problem is the expressive power required by the formula. It turns out that different logics correspond to different complexity classes and that all major complexity classes have logical characterizations.

Descriptive complexity provides a particularly convenient framework for automatically finding reductions. The fact that instances do not need to be encoded as words allows us to express interesting reductions succinctly, and formulas, unlike Turing machines, have natural normal forms. In this section we introduce the background in descriptive complexity necessary for this paper; refer to [13] or Chapter 3 of [8] for a more detailed introduction and additional material.

A *relational signature* $\tau := (R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s)$ is a tuple of predicate symbols R_i with arities a_i and constant symbols c_j. A finite *τ-structure*

$$\mathfrak{A} := (U, R_1 \subseteq U^{a_1}, \ldots, R_r \subseteq U^{a_r}, c_1 \in U, \ldots, c_s \in U)$$

consists of a finite universe U, an a_i-ary relation for each predicate symbol of τ, and a definition – an element of U – for each constant symbol.

For example, the signature for directed graphs contains a single, binary predicate symbol E and so a directed graph consists of a finite set of vertices and a binary edge relation. For convenience, we generally identify an n-element universe U with the set $\{0, \ldots, n-1\}$.

Formulas of *first-order logic* over a signature τ have the form

$$\varphi \ := \ R_i(x_1, \ldots, x_{a_i}) \mid x_i = x_j \mid x_i = c_j \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i\, \varphi \mid \forall x_i\, \varphi,$$

where $x_1, x_2, \ldots$ are first-order variables, and the semantics, given an assignment of the variables x_i to elements e_i of the structure, is defined in the natural way. For example, $\exists x_1 R(x_1, x_2)$ holds for an assignment $x_2 \to e_2$ in $\mathfrak{A}$ if, and only if, there exists an element e_1 of U such that (e_1, e_2) is in the relation R in $\mathfrak{A}$.

Formulas without free variables define *properties* in the natural way. For example, $\forall x, y\ \neg E(x, y)$ defines the property of having no edges, i.e. being an empty graph. That is, the property defined by a formula φ is the set of all structures on which φ holds. We use properties to specify decision problems.

Queries and reductions. Reductions map σ-structures to τ-structures, defining the universe, relations, and constants by means of logical formulas. Reductions are a special kind of *query*, and so we begin by defining first-order queries.

A *first-order query* from σ-structures to τ-structures is an $r+s+2$-tuple,

$$q \;:=\; (k, \varphi_0, \varphi_1, \dots, \varphi_r, \psi_1, \dots, \psi_s)\,.$$

The number $k \in \mathbb{N}$ is the *dimension* of the query. Each φ_i, ψ_j is a first-order formula over the signature σ. Let $\mathfrak{A}$ be a σ-structure with universe $U^{\mathfrak{A}}$. The formula φ_0 has free variables $x_1, \dots, x_k$ and defines the universe U of $q(\mathfrak{A})$,

$$U \;:=\; \bigl\{(u_1, \dots, u_k) \mid u_i \in U^{\mathfrak{A}}, \mathfrak{A} \models \varphi_0(u_1, \dots, u_k)\bigr\}\,.$$

That is, the new universe consists of k-tuples of elements of the old universe, where φ_0 determines which k-tuples are included.

Each remaining φ_i has free variables $x_1^1, \dots, x_1^k, x_2^1, \dots, x_{a_i}^k$ and defines

$$R_i := \bigl\{(u_1^1, \dots, u_1^k), \dots, (u_{a_i}^1, \dots, u_{a_i}^k) \mid \mathfrak{A} \models \varphi_i(u_1^1, \dots, u_{a_i}^k)\bigr\} \cap U^{a_i}\,.$$

That is, φ_i determines which of the a_i-tuples of U are included in R_i. Finally, each ψ_i has free variables $x_1, \dots, x_k$ and defines c_i as the unique $(u_1, \dots, u_k) \in U$ such that $\mathfrak{A} \models \psi_i(u_1, \dots, u_k)$.

First-order queries therefore transform σ-structures into τ-structures. Given a property P of σ-structures and a property Q of τ-structures, a first-order *reduction* r from P to Q is a first-order query that satisfies an additional condition. Namely, reductions must satisfy $\mathfrak{A} \in P \iff r(\mathfrak{A}) \in Q$ for all σ-structures $\mathfrak{A}$.

There are various kinds of first-order reductions (see [13]). Quantifier-free projections are the weakest version usually considered. There, all formulas in the reduction must be quantifier-free, and the reduction must also be a *projection*, i.e., each bit of the output depends on at most one bit of the input, where the bit is selected in first-order.

First-order reductions are weaker than, for example, polynomial-time reductions, and quantifier-free projections are even more restricted. However, natural problems that are complete for natural complexity classes via polynomial-time reductions tend to remain complete via these weaker reductions (see, e.g., [13]). In this paper we consider parametrized classes of quantifier-free reductions where all formulas are in disjunctive normal form (DNF). Therefore, all formulas in the reductions we consider are quantifier-free, however, the reductions are not necessarily projections. We also consider only formulas ψ_i that directly define the constant c_i (i.e., a fixed tuple of constant symbols from σ and $U^{\mathfrak{A}}$) and require φ_0 to be always true. These restrictions are for simplicity, and have minimal impact from the complexity-theory perspective, given that constants can be omitted or replaced by monadic relations.

Extending first-order logic. So far, we have focused only on first-order logic. Although it is generally more than adequate for reductions, first-order logic has several drawbacks when used to express properties. First of all, it is not expressive enough to describe many relations that can easily be computed. This limitation stems from the *locality* of first-order formulas. This property implies that it is not possible to express the transitive closure of a relation in first-order logic, so, e.g., also the property that a graph is connected.

To remove this limitation of first-order logic, one extends it with various operators. For example, the *transitive closure operator* allows us to write formulas of the form $\mathrm{TC}[x_1, x_2.\varphi(x_1, x_2)](y_1, y_2)$. This formula takes the transitive and reflexive closure of the (implicit) relation defined by $\varphi(x_1, x_2)$ and then evaluates it on (y_1, y_2). Adding the transitive closure operator removes some limitations of FO, but how can we know what other problems remain? Let us review some of the best-known correspondences between logics and complexity classes.

The oldest result [6] shows that the class NP is captured by existential second-order logic. More practically, polynomial-time computations are captured by the extension of FO by the least fixed-point operator (LFP) when a linear order relation is present [11,18]. The requirement of a linear order can be weakened when a counting mechanism is added to the logic, and LFP with counting captures P on many classes of structures, such as grids, planar graphs [9] and all classes that exclude a fixed minor [10]. Although LFP is presumably more expressive than the transitive closure logic (TC) we mentioned, TC captures all problems solvable in non-deterministic logarithmic space (NL) on ordered structures [12].

Example 1. Having introduced the transitive closure operator and our notion of reductions, let us give a simple example that our reduction-finding systems can find[6]. Consider the following formulas,

$$\text{Reach} := TC[x, y.E(x, y)](s, t) \quad \text{AllReach} := \forall x_1, x_2\, (TC[y, z.E(y, z)](x_1, x_2))\,.$$

Here, Reach expresses the NL-complete problem of reachability (there exists a directed path from s to t) and AllReach expresses the NL-complete problem of all-pairs reachability (there is a directed path from x to y for all vertices x, y).

Using the notation for reductions introduced above, a correct reduction from Reach to AllReach is

$$(k := 1,\ \varphi_0 := true,\ \varphi_1 := x_1 = s \vee x_2 = t \vee E(x_2, x_1))\,.$$

This reduction reverses all edges in the original graph, adds directed edges from s to all vertices and also adds directed edges to t from all vertices. It is not difficult to see that the result is strongly connected if, and only if, the original graph has a directed path from s to t. Note that a similar reduction exists without reversing the edges – however the above is the actual output of our program.

3 Finding Reductions

The fundamental problem that we want to solve is the following. Given two logical formulas φ_P and φ_Q, is there a reduction from the property defined by φ_P to that defined by φ_Q? Unfortunately, this problem is undecidable. In fact, it is also undecidable to determine whether a given reduction is correct for two fixed properties, or even whether two given properties are logically equivalent.

[6] With parameters $k = 1, c = 3, n = 4$ in $< 3s$.

Our fundamental approach is to fix an "outline" of the reduction we hope to find. For example, we may assume that all formulas[7] in the reduction are quantifier-free, in DNF, and are a disjunction of exactly c conjunctions. Once the signatures are fixed, there is only a finite number of atoms that could occur in these conjunctions. This is because there are only finitely many variables (we cannot introduce new variables with quantifiers) and constants, and only finitely many ways to combine these symbols with the relation symbols and equality. For each atom and conjunction, we introduce a Boolean variable representing whether or not the atom occurs in that conjunction of the reduction. Intuitively, we can now express the existence of a reduction as a logical formula

$$\exists \mathbf{r}\ \forall \mathfrak{A}\ \ (\mathfrak{A} \models \varphi_P \ \leftrightarrow\ \mathbf{r}(\mathfrak{A}) \models \varphi_Q)\,, \tag{1}$$

where $\mathbf{r}$ is the finite set of Boolean variables defining the reduction, and $\mathfrak{A}$ ranges over all structures having the same signature as φ_P.

Of course, there are infinitely many such structures, which explains why the problem is still undecidable. However, experience shows that it usually suffices to consider structures of fairly small size, at least for natural problems and natural classes of weak reductions. That is, although one can construct artificial properties where arbitrarily large examples are needed, it seems that if a simple reduction between natural problems is correct on all small instances, then it is usually correct on all instances.

Therefore we focus, as did [3], on finding reductions that are correct on all structures of size n. Here, n is a parameter and it is also possible to consider ranges of n. Once n is fixed, as well as the outline for $\mathbf{r}$, checking Formula (1) becomes decidable. In fact, it is natural to represent $\mathfrak{A}$ with Boolean variables, and so Formula (1) becomes a one-alternation quantified Boolean formula. Satisfiability of such formulas[8] is complete for Σ_2^p.

Example 2. Let us show how the QBF for Formula (1) is constructed in the following case. Let P be the class of non-empty graphs defined by $\varphi_P = \exists x, y\, E(x, y)$ and let Q be the class of non-complete graphs given by $\varphi_Q = \exists x, y\, \neg E(x, y)$. We ask whether there is a quantifier-free reduction from P to Q of dimension $k = 1$, with $c = 1$ conjunctions, and that is correct on all graphs of size $n = 2$.

First, let us fix the outline for our reduction with one conjunction. Note that $\sigma = \tau = \{E\}$, so in this case the reduction we are looking for has the following form: $(k := 1,\ \varphi_0 := \mathit{true},\ \varphi_1(x_1, x_2))$ for some formula $\varphi_1(x_1, x_2)$ which is a conjunction of literals. What atoms are possible over the signature $\{E\}$ with variables x_1, x_2? There are exactly 5 atoms in the basic syntax: $E(x_1, x_1)$, $E(x_1, x_2)$, $E(x_2, x_1)$, $E(x_2, x_2)$, and $x_1 = x_2$. So, in this most basic case, the outline for φ_1 has the form $\varphi_1(x_1, x_2) =$

$$\begin{aligned}&X_1 E(x_1, x_1)\ \wedge X_2 E(x_1, x_2)\ \wedge X_3 E(x_2, x_1)\ \wedge X_4 E(x_2, x_2)\ \wedge X_5 x_1 = x_2 \ \wedge\\ &Y_1 \neg E(x_1, x_1) \wedge Y_2 \neg E(x_1, x_2) \wedge Y_3 \neg E(x_2, x_1) \wedge Y_4 \neg E(x_2, x_2) \wedge Y_5 \neg x_1 = x_2.\end{aligned}$$

[7] As mentioned above, we always fix $\varphi_0 = \mathit{true}$ and define constants by fixed tuples.

[8] Note that Formula (1) has leading existential quantifiers; satisfiability of one-alternation formulas with leading universal quantifiers is complete for Π_2^p.

Above, X_i and Y_i are propositional variables that determine whether the literal after them will appear or not: $X_1 E(x_1, x_2)$ means "$E(x_1, x_2)$ if X_1 is set and *true* otherwise", as becomes clear below. An outline is thus a formula with these additional propositional variables used as guards.

In all our tests, we use an extended set of atoms, not only the basic ones presented above for readability. In the extended set, in addition to relations over variables and equality as above, we allow the following atoms: for a fixed enumeration of the elements of the structure, we say that x is the minimal one, the maximal one, or that $x = y + 1$. This allows to find more reductions with the same outline parameters.

Having constructed the outline, let $\mathfrak{A}$ be a 2-element structure and assume that the tuple (i, j), for $i, j \in \{0, 1\}$, is in the relation E in $\mathfrak{A}$ if, and only if, the propositional variable E_{ij} is set. Note that the part $\mathfrak{A} \models \varphi_P$ of Formula (1), in our case $\mathfrak{A} \models \exists x, y\, E(x, y)$, can now be written as a purely propositional formula: $\bigvee_{i,j\in\{0,1\}} E_{ij}$. To express that $\mathbf{r}(\mathfrak{A}) \models \varphi_Q$ we need to use the definition of E in $\mathbf{r}(\mathfrak{A})$ given by φ_1. In our case, for $\mathbf{r}(\mathfrak{A}) \models \exists x, y\, \neg E(x, y)$ we write $\bigvee_{i,j\in\{0,1\}} \neg\varphi_{ij}$. Here φ_{ij} is derived from the outline of φ_1 using the propositional variables E_{ij}. For the basic outline presented above, that means $\varphi_{ij} =$

$$\begin{array}{l}(\neg X_1 \vee E_{ii}) \;\wedge (\neg X_2 \vee E_{ij}) \;\wedge (\neg X_3 \vee E_{ji}) \;\wedge (\neg X_4 \vee E_{jj}) \;\wedge (\neg X_5 \vee i = j) \wedge \\ (\neg Y_1 \vee \neg E_{ii}) \wedge (\neg Y_2 \vee \neg E_{ij}) \wedge (\neg Y_3 \vee \neg E_{ji}) \wedge (\neg Y_4 \vee \neg E_{jj}) \wedge (\neg Y_5 \vee i \neq j),\end{array}$$

where $i = j$ and $i \neq j$ get substituted by *true* or *false* depending on i and j.

In this way, we obtain the following propositional formula, which we call the QBF corresponding to Formula (1) for $k = 1, c = 1, n = 2$.

$$\exists X_1 \dots X_5\, Y_1 \dots Y_5\; \forall E_{00} \dots E_{11} \;\left(\bigvee_{i,j\in\{0,1\}} E_{ij} \;\leftrightarrow\; \bigvee_{i,j\in\{0,1\}} \neg\varphi_{ij} \right).$$

Observe that this formula is satisfied exactly if there is a reduction with the specified outline correct on all structures of the specified size. Moreover, if satisfied, the outer-most existentially quantified variables allow to extract a reduction.

3.1 Approaches to Solving Σ_2^p Problems

Several approaches have been used to solve problems in Σ_2^p. Actually constructing the QBF for Formula (1) as above is fairly tedious, but poses no serious difficulty. This results in a QBF instance with one quantifier alternation where a satisfying assignment of the existential variables gives a reduction, so QBF solvers can be immediately applied.

Although they are perhaps not yet as mature as SAT solvers, in recent years there has been a great deal of work on efficient QBF solvers. See, for example, the QBFEVAL series of evaluations [17]. QBFEVAL'10 also included a 2QBF track, and we believe our instances could be attractive candidates for inclusion in that track. Still, as will be clarified below, in our case it is often more efficient to present the formula as a 3QBF instance.

In addition to QBF solvers, there are other approaches that have been used to solve Σ_2^p problems. For example, ASP solvers that support disjunctive logic programs can solve such problems using the reduction to disjunctive logic programs given by [4,16]. Examples of such solvers are CMODELS and CLASPD, and some previous work [5] has indicated that ASP solvers may outperform QBF solvers on Σ_2^p problems.

Another option is to expand the universal quantifier block of the formula using a conjunction over all possible assignments, resulting in a SAT instance. This allows SAT solvers to be used directly, however it entails an exponential increase in instance size. In our experience, this is impractical for large instances.

Finally, some recent work [3,14,15] has noted that one can use repeated calls to SAT solvers to solve Σ_2^p instances, essentially an application of counter-example guided abstraction refinement (CEGAR) [2]. This approach is also a finitely-truncated implementation of limit-learning [7], using guesses for the leading existential quantifiers as hypotheses and assignments to the universal quantifiers as counter-examples. Our dedicated reduction finder DE uses this approach.

In our view, the connection to limit-learning gives a particularly nice perspective on our problem. For example, removing the finite-size restriction needed for decidability results is *exactly* an attempt to learn reductions from counter-examples in the limit. Of course, it is undecidable whether such a learner has converged to a correct hypothesis. Techniques from inductive inference may provide valuable guidance on efficient learning, i.e., minimizing total computation time, or number of counter-examples required.

4 Reduction Finding using QBF and ASP Solvers

In this section, we compare the performance of various QBF and ASP solvers on our problem. Our approach to generating QBF instances for reduction finding was outlined above. Essentially, we view the problem from the perspective of Formula (1) as a QBF instance and apply the above-mentioned approaches.

Note that Formula (1) is of the form $\exists r \, \forall \mathfrak{A} \, \psi$. The reduction r contributes existential propositional variables, the structure $\mathfrak{A}$ is the source of universal variables, and ψ is a quantifier-free propositional formula. While it is quantifier-free, in general this formula is not in conjunctive normal form (CNF). Most QBF solvers require their input to be in CNF and CNF conversion usually involves introducing new existentially quantified variables, which must be innermost.

We investigate three approaches to dealing with the conversion to CNF. First, there are QBF solvers that only require the formula to be in negation normal form (NNF), not CNF. The most common input format for such solvers is called `qpro` and in this case we generate the formula directly.

The second approach is to convert the propositional part of Formula (1) to CNF adding new existential variables.[9] In this case, we produce output in

[9] We used the standard Plaisted-Greenbaum technique for this, which in our tests slightly outperformed the often used Tseitin method.

qdimacs format. The resulting formula is of the form $\exists\overline{X}\ \forall\overline{Y}\ \exists\overline{Z}\ \psi_{\mathrm{CNF}}$ – it has one more quantifier alternation than strictly necessary.

The third approach is to first negate Formula (1), which then has the form $\forall r\,\exists\mathfrak{A}\,\psi$. We then convert ψ to CNF as above, and generate a qdimacs file with only one quantifier alternation.

Finally, we also generate lparse files for ASP solvers, as described in Subsection 3.1. This results in the following four cases that we test and benchmark.

qpro	Constructing QBF for Formula (1) and using QBF solvers that support non-CNF QBF (CIRQIT).
qdimacs	Constructing QBF for Formula (1) and converting directly to CNF, then using QBF solvers (RAREQS, DEPQBF, QUBE7.2, SKIZZO, CIRQIT).
nqdimacs	Negating Formula (1) before CNF conversion to avoid one quantifier alternation, and using QBF solvers (RAREQS, DEPQBF, QUBE7.2, SKIZZO, CIRQIT).
lparse	Constructing Formula (1) and using ASP solvers that support disjunctive logic programs (LPARSE or GRINGO, and CMODELS, CLASPD or GNT2).

Some of the combinations listed above performed quite poorly even on very simple reduction finding instances. In particular, CMODELS and GNT2 almost always abort (but produce correct output if they do not abort), and using CIRQIT with nqdimacs is very slow (but produces correct output). We therefore omit these combinations from our experimental results.

4.1 Comparing the QBF Approaches

We first concentrate on the following question: which of the three approaches to using QBF solvers, qdimacs, nqdimacs, and qpro, performs best? It turns out that there is a clear answer: qdimacs is the best option.

Comparing qdimacs and nqdimacs. Given the explanation above, one might think that nqdimacs is a more promising formulation for our instances – it expresses the same problem but avoids one quantifier alternation. We were mildly surprised to see that all QBF solvers we tested consistently performed worse on nqdimacs than on qdimacs instances with one more quantifier alternation. In Table 1 below we present the number of timeouts for qdimacs and nqdimacs instances for the easiest set of parameters we tested: $k = 1$, $c = 1$, and $n = 3$. Clearly, both DEPQBF and QUBE performed much better on qdimacs than on nqdimacs, e.g. there were no timeouts (set to 120s in this section) on any qdimacs instance for these two solvers but several hundred (out of 2304[10]) for nqdimacs, and the completed instances also took longer to finish. For SKIZZO the situation is less clear but the general pattern is the same. The CEGAR-based solver RAREQS had only 2 timeouts on this instance, both in the negated setting, but exhibits the same pattern for $c = 2$ (0 vs. 304 timeouts).

[10] We used 48 decision problems from [3] to be able to compare to ReductionFinder. They range from very simple, like the empty graph, to the NL-complete reachability and coNL-complete unreachability problems.

	`qdimacs`	`nqdimacs`
DEPQBF	0	300
QUBE	10	285
SKIZZO	522	706

Table 1. Number of timeouts for `qdimacs` and `nqdimacs` solvers.

Comparing `qdimacs` and `qpro`. When comparing `qdimacs` and `qpro` we must note that most QBF solvers accept `qdimacs` input while we found only one, CIRQIT, that accepts `qpro` and is freely available. This is crucial: either DEPQBF or QUBE reading `qdimacs` outperform CIRQIT uniformly, on all instances, with 1s margin of error, and both these solvers have far fewer timeouts than CIRQIT. On the other hand, when comparing only CIRQIT on `qdimacs` input with CIRQIT on `qpro` input, neither has a clear advantage – there are numerous instances where one times out and the other does not, and also instances where the opposite occurs. In Table 2 we show the number of timeouts of different solvers on three parameter sets of increasing difficulty.

	$k = 1$ $c = 1$ $n = 3$	$k = 1$ $c = 2$ $n = 3$	$k = 1$ $c = 3$ $n = 3$
RAREQS (`qdimacs`)	0	0	16
DEPQBF (`qdimacs`)	0	142	547
QUBE (`qdimacs`)	10	536	949
CIRQIT (`qdimacs`)	58	673	1138
CIRQIT (`qpro`)	157	523	903

Table 2. Number of timeouts for `qdimacs` and `qpro` solvers.

The behavior of CIRQIT on `qdimacs` and `qpro` instances shows that one can benefit from knowing the structure of the formula. Together with the fact that `qdimacs` outperforms `nqdimacs`, these seem to indicate that a more careful handling of the innermost formula could lead to more efficient solvers.

Comparing QBF solvers. Having settled on `qdimacs`, we now compare the performance of the five QBF solvers on different parameter sets. In Table 3 we report the number of timeouts for each solver and each parameter set.

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 on which DEPQBF and QUBE time out, but SKIZZO answers almost immediately. This allows to hope that tailoring the strategies of QBF solvers towards Σ_2^p problems might still lead to significant performance gains.

	$c = 1$ $n = 3$	$c = 2$ $n = 3$	$c = 3$ $n = 3$	$c = 1$ $n = 4$	$c = 2$ $n = 4$	$c = 3$ $n = 4$
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

Table 3. Number of timeouts for different QBF solvers, $k = 1$.

4.2 Comparing Different Approaches to Reduction Finding

Having discussed the QBF approaches and chosen the best QBF solvers, let us finally compare the QBF approach with the approach using ASP solvers, and also with our reduction finder DE and ReductionFinder from [3].

Different ASP solvers. We consider three different ASP solvers (CMODELS, GNT2, and CLASPD) and two different grounding programs (LPARSE and GRINGO). Grounding is performed before the solver is started, and may take significant time. However, we time the total of grounding and solving and so it is possible for the grounding program to timeout before the solver begins.

Two of the solvers (CMODELS and GNT2) abort very frequently, even on simple instances and regardless of the choice of grounding program. Therefore we only show results for CLASPD with LPARSE and GRINGO. Interestingly, while the total number of timeouts was similar for the two grounders, there were numerous instances where one timed out and the other finished quickly. This may give some reason to hope that significantly better performance may be possible with this approach with more careful grounding.

Results. In Table 4 we compare the different reduction finding approaches that we tested. For the SAT-solver based DE runs, we have chosen DE-GMS[11] as it performs best on hard instances. For BDD-based DE runs, we show DE-CUDD, the only reduction finder we tested that used BDDs. For QBF solvers we have chosen the two best performers, RAREQS and DEPQBF. Note, that RAREQS is a CEGAR-based solver, more similar to DE-MS than DEPQBF. For ASP solvers, we show CLASPD both with LPARSE and GRINGO, as explained above.

We also include the results for ReductionFinder [3]. ReductionFinder only considers reductions of a slightly different form – this usually results in a simpler instance, so it is shown here only for comparison. All other approaches we present find reductions of exactly the same form – naturally, the answers (whether there is a reduction or not) agree on all parameter sets that we tested.

The CEGAR approach, whether in DE or in RAREQS, outperforms the others. Other QBF solvers (DEPQBF, QUBE) match the performance of the original ReductionFinder and in general perform better than the ASP approach.

[11] DE can use MiniSat2 (DE-MS), GlueMiniSat (DE-GMS), CryptoMiniSat (DE-CMS) or BDDs (DE-CUDD).

	$c = 1\ n = 3$	$c = 2\ n = 3$	$c = 3\ n = 3$	$c = 1\ n = 4$	$c = 2\ n = 4$	$c = 3\ n = 4$
DE-GMS	0 (100.0%)	0 (100.0%)	10 (99.6%)	0 (100.0%)	5 (99.8%)	103 (95.5%)
DE-CUDD	0 (100.0%)	116 (95.0%)	537 (76.7%)	0 (100.0%)	186 (91.9%)	722 (68.7%)
RAREQS	0 (100.0%)	0 (100.0%)	16 (99.3%)	19 (99.1%)	65 (97.1%)	204 (91.1%)
DEPQBF	0 (100.0%)	142 (93.8%)	547 (76.2%)	16 (99.3%)	297 (87.1%)	711 (66.1%)
GRINGO	40 (98.3%)	393 (82.9%)	590 (74.4%)	72 (96.9%)	593 (74.3%)	836 (63.7%)
LPARSE	51 (97.8%)	396 (82.8%)	605 (73.7%)	75 (96.7%)	635 (72.4%)	850 (63.1%)
RedFind	1 (99.9%)	152 (93.4%)	396 (82.8%)	2 (99.9%)	347 (84.9%)	547 (76.3%)

Table 4. Number of timeouts (% solved) for tested reduction finding approaches, $k = 1$.

5 Reduction Finding using CEGAR

We now compare CEGAR approaches to reduction finding. We begin by describing the reduction finding procedure used in DE and then compare the reduction-finding implementations in DE with RAREQS, focusing on difficult instances.

5.1 Finding Reductions in DE

DE finds reductions by first choosing a candidate reduction r_0, then searching for a counter-example (i.e., a structure $\mathfrak{A}$ such that $\mathfrak{A} \models \varphi_P \iff r_0(\mathfrak{A}) \not\models \varphi_Q$). If a counter-example is found, it searches for a reduction that is correct on all examples seen so far, i.e., if we have seen examples $\{\mathfrak{A}_0, \ldots, \mathfrak{A}_i\}$, then we search for an assignment of the Boolean variables $\mathbf{r}$ such that

$$(\mathfrak{A}_0 \models \varphi_P \leftrightarrow r(\mathfrak{A}_0) \models \varphi_Q) \wedge \cdots \wedge (\mathfrak{A}_i \models \varphi_P \leftrightarrow r(\mathfrak{A}_i) \models \varphi_Q) \ , \qquad (2)$$

and iterate until either no counter-example or no candidate reduction is found.

In Formula (2), whether $\mathfrak{A}_j \models \varphi_P$ is known, so some simplifications can be easily performed. In our experience (and that of [3]), finding candidate reductions is more difficult than finding counter-examples. We therefore focus mostly on finding candidate reductions. However, we have optional heuristics to help choose *good* counter-examples. The times reported for DE alternate greedy minimization and maximization of counter-examples[12], except for CryptoMiniSat.

We implement candidate-finding using incremental SAT solvers or BDDs (using CUDD). Formula (2) is a natural candidate for incremental SAT solving: there are comparatively few Boolean variables $\mathbf{r}$ which are re-used, and at each stage we simply add the restriction corresponding to the new counter-example.

Using BDDs for candidate-finding is similar. At each stage, we have a BDD representing the set of candidate reductions that are consistent with the previous counter-examples. Given a new counter-example, we build a BDD of the hypotheses consistent with it and take the intersection of the two sets. However, to acquire a new counter-example, we must have a particular hypothesis.

[12] Given a counter-example, we greedily remove or add tuples to relations while checking that it remains a counter-example.

We take the "simplest" candidate reduction as our hypothesis, i.e., we select from the set of candidates consistent with the examples we have seen a candidate with the *minimal* number of atomic formulas appearing. This has an advantage: there are often several correct reductions in a search space. In this case, the BDD implementation will give a simplest correct reduction. The difference in size (and clarity) between this reduction and others can be substantial. We therefore often prefer the output of the BDD version. However, it is usually slower and much more memory-intensive than the SAT versions (see Subsection 5.2 below).

There is a known bug[13] in MiniSat2 related to simplification. This affects us, and so we disable simplification in MiniSat2. The same bug appears to be present in Glucose, so we do not benchmark Glucose. CryptoMiniSat and GlueMiniSat appear to be unaffected, and we leave simplification enabled for them.

5.2 Performance Results

In this subsection, we focus on a particular difficult instance, searching for reductions from $\overline{\text{REACH}}$ (the coNL-complete problem of checking whether there is no directed path from s to t in a graph) to REACH (the NL-complete problem of checking whether there is such a path). Finding a (correct) reduction between these problems proves the Immerman-Szelepcsényi Theorem. It is known that a dimension-8 QFP exists, it is interesting to determine whether $k = 8$ is required.

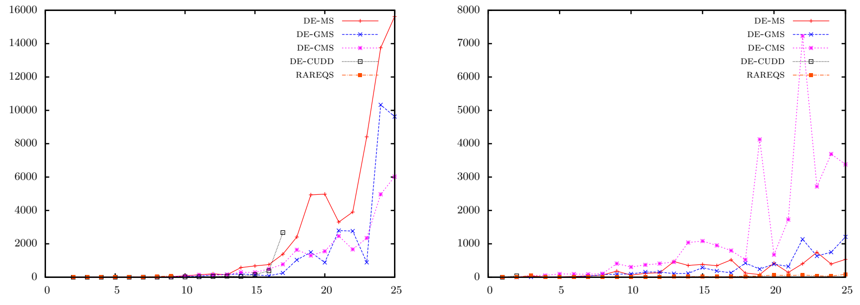

Fig. 1. CEGAR performance (in seconds) scaling n (left), c (right)

Our implementations do not approach $k = 8$. To begin, we fix simple parameters of $k = 1, c = 1, n = 3$ and examine performance when scaling k, c, n. The results are in Figure 1 and Table 5, with a (minimum) timeout of four hours. In our experience, RAREQS performs best when scaling c and the GlueMiniSat version performs best when scaling n. This is due to differences in how we handle common subformulas in DE and QBF generation – it is possible to unify these.

When actually searching for reductions, we can range over counter-example sizes (starting with $n = 1$ or $n = 2$). The small counter-examples exclude many

[13] `https://github.com/niklasso/minisat/issues/3`

	DE-MS	DE-GMS	DE-CMS	DE-CUDD	RAREQS
$k=1, c=1, n=3$	0.05	0.06	0.08	0.07	0.03
$k=2, c=1, n=2$	0.06	0.11	0.28	6.30	0.06
$k=2, c=1, n=3$	3562.14	1696.26	1755.03	timeout	3267.10

Table 5. Times (in seconds) as k increases.

candidates, giving a large performance improvement[14]. For this example, no reduction in this space exists for $n = 5$, so our implementations would stop at that point. Increasing only n decreases the likelihood of a reduction existing, making the instance more strongly negative.

Scaling only c is similar; a "reduction" (correct on graphs of size 3 but not in general) exists at $c = 4$. In this example, the instance becomes more strongly positive as c increases. However, scaling these parameters shows the limit of our current approaches. Even with scaling, our implementations do not perform reasonably on hard instances (properties which use transitive closure) with $k > 2$.

Above we considered baseline settings of $k = 1, c = 1, n = 3$ and scaled a single parameter. In Figure 2, we scale n with a slightly-modified baseline of $k = 1, c = 2$ to show performance with more than one conjunction.

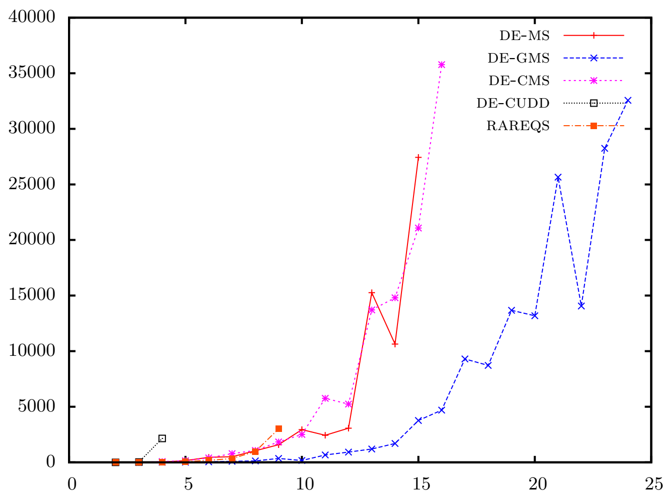

Fig. 2. CEGAR performance scaling n with $k = 1, c = 2$.

[14] We do not include range benchmarks here. They give additional advantages to the CEGAR approach.

SAT vs BDD. The largest difference in performance between DE versions is between the BDD implementation and the SAT implementations. The BDD implementation maintains a structure explicitly representing all reductions consistent with the examples seen so far. This is memory-intensive and slower than the SAT versions, which only find a single explicit candidate at each stage.

However, the BDD version chooses a simplest candidate at each step. This usually results in the BDD version requiring fewer iterations of the counter-example/candidate loop, although each iteration is more expensive. For example, if we look for a reduction from the query $R(s)$ to the query $\exists x\, R(x)$ with parameters $k = 3, c = 4, n = 6$, DE with CryptoMiniSat uses 7 counter-examples and finds a correct, but unnecessarily complicated, reduction. DE with CUDD uses 3 examples and finds the simplest reduction in this search space.

Comparing SAT solvers. The SAT-based DE implementations and RAREQS outperform the other approaches we consider. However, differences between the SAT solvers are visible in the results above. On the hardest examples we consider, DE with GlueMiniSat is best for large n. Our generator handles common subformulas better than DE, visible in the performance of RAREQS when scaling c.

On easy instances, DE using MiniSat often outperforms the others. This is likely because we disable simplification to avoid a bug in MiniSat and simplification is not needed for these instances. However, in practice we prefer the BDD-based implementation in such cases: any approach suffices for these instances and we prefer the more-understandable reductions found by this version.

When searching over ranges of counter-example sizes, GlueMiniSat usually performs best and we prefer GlueMiniSat on hard instances. CryptoMiniSat supports parallel SAT-solving – we do not use this, but it may improve performance.

6 Conclusions

We have developed and benchmarked several approaches to the problem of automatically discovering reductions between decision problems[15]. For each approach, it is possible to find instances where it outperforms the others. However, it is possible to state several clear conclusions. The dedicated CEGAR approach is generally the best, and GlueMiniSat performs best on hard instances. Our BDD-based approach gives the nicest output. The performance of QBF solvers depends heavily on the way in which the innermost formula is converted to CNF. For ASP solvers, much depends on the grounder used before the solver starts.

Our experiments show that each of the tested approaches – QBF solvers, ASP solvers, and the CEGAR method – still leaves a large room for improvements. We provide our testing instances and their generator, we will submit them to relevant competitions, and we encourage the community to use them for testing and optimization of all mentioned solvers. Moreover, we hope that it is possible to combine the strengths of each approach together with new improvements, in order to achieve better performance on hard, meaningful instances.

[15] Visit `http://toss.sf.net/reduct.html` to test the CEGAR approach online.

References

1. Allender, E., Balcázar, J.L., Immerman, N.: A first-order isomorphism theorem. SIAM J. Comput. 26(2), 539–556 (1997)
2. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
3. 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)
4. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence 15(3-4), 289–323 (1995)
5. Faber, W., Ricca, F.: Solving hard ASP programs efficiently. In: Proc. of LP-NMR'05. LNCS, vol. 3662, pp. 240–252. Springer (2005)
6. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Complexity of Computation, SIAM-AMS Proceedings. vol. 7, pp. 43–73. Amer. Mathematical Soc. (1974)
7. Gold, E.: Language identification in the limit. Inform. Control 10(5), 447–474 (1967)
8. Grädel, E., Kolaitis, P.G., Libkin, L., Marx, M., Spencer, J., Vardi, M.Y., Venema, Y., Weinstein, S.: Finite Model Theory and Its Applications. Texts in Theoretical Computer Science, Springer (2007)
9. Grohe, M.: Fixed-point logics on planar graphs. In: Proc. of LICS'98. pp. 6–15. IEEE Computer Society (1998)
10. Grohe, M.: Fixed-point definability and polynomial time on graphs with excluded minors. J. ACM 59(5), 27:1–27:64 (2012)
11. Immerman, N.: Relational queries computable in polynomial time. Inform. Control 68, 86–104 (1986)
12. Immerman, N.: Languages that capture complexity classes. SIAM J. Comput. 16(4), 760–778 (1987)
13. Immerman, N.: Descriptive Complexity. Springer-Verlag (1999)
14. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Proc. of SAT'12. LNCS, vol. 7317, pp. 114–128. Springer (2012)
15. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: Proc. of SAT'11. LNCS, vol. 6695, pp. 230–244. Springer (2011)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)
17. Peschiera, C., Pulina, L., Tacchella, A., Bubeck, U., Kullmann, O., Lynce, I.: The seventh QBF solvers evaluation (QBFEVAL'10). In: Proc. of SAT'10. LNCS, vol. 6175, pp. 237–250. Springer (2010)
18. Vardi, M.Y.: The complexity of relational query languages. In: Proc. of STOC'82. pp. 137–146. ACM (1982)