Finding Reductions Automatically

Michael Crouch*, Neil Immerman* and J. Eliot B. Moss*

Computer Science Dept., University of Massachusetts, Amherst
{mcc,immerman,moss}@cs.umass.edu

Abstract. We describe our progress building the program Reduction-Finder, which uses off-the-shelf SAT solvers together with the Cmodels system to automatically search for reductions between decision problems described in logic.

Key words: Descriptive Complexity, First-Order Reduction, Quantifier-Free Reduction, SAT Solver

1 Introduction

Perhaps the most useful item in the complexity theorist's toolkit is the reduction. Confronted with decision problems $A, B, C, \ldots$, she will typically compare them with well-known problems, e.g., REACH, CVP, SAT, QSAT, which are complete for the complexity classes NL, P, NP, PSPACE, respectively. If she finds, for example, that A is reducible to CVP ($A \leq$ CVP), and that SAT $\leq B$, $C \leq$ REACH, and REACH $\leq C$, then she can conclude that A is in P, B is NP hard, and C is NL complete.

When Cook proved that SAT is NP complete, he used polynomial-time Turing reductions [4]. Shortly later, when Karp showed that many important combinatorial problems were also NP complete, he used the simpler polynomial-time many-one reductions [14].

Since that time, many researchers have observed that natural problems remain complete for natural complexity classes under surprisingly weak reductions including logspace reductions [13], one-way logspace reductions [9], projections [22], first-order projections, and even the astoundingly weak quantifier-free projections [11].

It is known that artificial non-complete problems can be constructed [15]. However, it is a matter of common experience that most natural problems are complete for natural complexity classes. This phenomenon is receiving a great deal of attention recently via the dichotomy conjecture of Feder and Vardi that all constraint satisfaction problems are either NP complete, or in P [7, 20, 1].

* The authors were partially supported by the National Science Foundation under grants CCF-0830174 and CCF-0541018 (first two authors) and CCF-0953761 and CCF-0540862 (third author). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Since natural problems tend to be complete for important complexity classes via very simple reductions, we ask, *"Might we be able to automatically find reductions between given problems?"*

Of course this problem is undecidable in general. However, we have made progress building a program called ReductionFinder that automatically does just that. Given two decision problems A and B, ReductionFinder attempts to find the simplest possible reduction from A to B. Using off-the-shelf SAT solvers together with the Cmodels system[8], ReductionFinder finds many simple reductions between a wide class of problems, including several "clever" reductions that the authors had not realized existed.

The reader might wonder why we would want to find reductions automatically. In fact, we feel that an excellent automatic reduction finder would be an invaluable tool, addressing the following long-term problems:

1. There are many questions about the relations between complexity classes that we cannot answer. For example, we don't know whether P = NP, nor even whether NL = NP, whether P = PSPACE, etc. These questions are equivalent to the existence of quantifier-free projections between complete problems for the relevant classes [11]. For example, P = NP iff SAT $\leq_{\text{qfp}}$ CVP. Similarly, NL = NP iff SAT $\leq_{\text{qfp}}$ REACH and P = PSPACE iff QSAT $\leq_{\text{qfp}}$ CVP. Having an automatic tool to find such reductions or determine that no small reductions exist may improve our understanding about these fundamental issues.
2. Another ambitious goal, well formulated by Jack Schwartz in the early 1980s, is to precisely describe a computational task in a high-level language such as SETL [21] and build a smart compiler that can automatically synthesize efficient code that correctly performs the task. A major part of this goal is to automatically recognize the complexity of problems. Given a problem, A, if we can automatically generate a reduction from A to CVP, then we can also synthesize code for A. On the other hand if we can automatically generate a reduction from SAT to A, then we know that A is NP hard, so it presumably has no perfect, efficient implementation and we should instead search for appropriate approximation algorithms.
3. Being able to automatically generate reductions will provide a valuable tool for understanding the relative complexity of problems. If we restrict our attention to linear reductions, then these give us true lower and upper bounds on the complexity of the problem in question compared to a known problem, K: if we find a linear reduction from A to K, then we can automatically generate code for A that runs in the same time as that for K, up to a constant multiple. Similarly if we find a linear reduction from K to A, then we know that there is no algorithm for A that runs significantly faster than the best algorithm for K.

It is an honor for us to have our paper appear in this Festschrift for Yuri Gurvich. Yuri has made many outstanding contributions to logic and computer science. We hope he is amused by what we feel is a surprising use of SAT solvers for automatically deriving complexity-theoretic relations between problems.

This paper is organized as follows: We start in Section §2 with background in descriptive complexity sufficient for the reader to understand all she needs to know about reductions and the logical descriptions of decision problems. In section §3 we explain our strategy for finding reductions using SAT solvers. In section §4 we sketch the implementation details. In section §5 we provide the main results of our experiments: the reductions found and the timing. We conclude in section §6 with directions for moving this research forward.

2 Reductions in Descriptive Complexity

In this section we present background and notation from descriptive complexity theory concerning the representation of decision problems and reductions between them. The reader interested in more detail is encouraged to consult the following texts: [10, 5, 16], where complete references and proofs of all the facts mentioned in this section may be found.

2.1 Vocabularies and Structures

In descriptive complexity, part of finite model theory, the main objects of interest are finite logical structures. A *vocabulary*

$$\tau = \langle R_1^{a_1}, \ldots, R_r^{a_r}; c_1, \ldots, c_s; f_1^{r_1}, \ldots, f_t^{r_t} \rangle$$

is a tuple of relation symbols, constant symbols, and function symbols. R_i is a relation symbol of arity a_i and f_j is a function symbol of arity $r_j > 0$. A constant symbol is just a function symbol of arity 0. For any vocabulary τ we let $\mathcal{L}(\tau)$ be the set of all grammatical first-order formulas built up from the symbols of τ using boolean connectives, $\neg, \vee, \wedge, \rightarrow$ and quantifiers, $\forall, \exists$.

A *structure* of vocabulary τ is a tuple,

$$\mathcal{A} \quad = \quad \langle |\mathcal{A}|; R_1^{\mathcal{A}}, \ldots, R_r^{\mathcal{A}}; c_1^{\mathcal{A}}, \ldots, c_s^{\mathcal{A}}; f_1^{\mathcal{A}}, \ldots, f_t^{\mathcal{A}} \rangle$$

whose universe is the nonempty set $|\mathcal{A}|$. For each relation symbol R_i of arity a_i in τ, $\mathcal{A}$ has a relation $R_i^{\mathcal{A}}$ of arity a_i defined on $|\mathcal{A}|$, i.e. $R_i^{\mathcal{A}} \subseteq |\mathcal{A}|^{a_i}$. For each function symbol $f_i \in \tau$, $f_i^{\mathcal{A}}$ is a total function from $|\mathcal{A}|^{r_i}$ to $|\mathcal{A}|$.

Let STRUC[τ] be the set of finite structures of vocabulary τ. For example, $\tau_g = \langle E^2; ; \rangle$ is the vocabulary of (directed) graphs and thus STRUC[τ_g] is the set of finite graphs.

2.2 Ordering

It is often convenient to assume that structures are ordered. An *ordered structure* $\mathcal{A}$ has universe $|\mathcal{A}| = \{0, 1, \ldots, n-1\}$ and *numeric* relation and constant symbols: $\leq, \mathrm{Suc}, \min, \max$ referring to the standard ordering, successor relation, minimum, and maximum elements, respectively (we take $\mathrm{Suc}(\max) = \min$). ReductionFinder may be asked to find a reduction on ordered or unordered structures. In the former case it may use the above numeric symbols. Unless otherwise noted, we from now on assume that all structures are ordered.

2.3 Complexity Classes and their Descriptive Characterizations

We hope that the reader is familiar with the definitions of most of the following complexity classes:

$$\mathrm{AC}^0 \subset \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \tag{1}$$

where $\mathrm{L} = \mathrm{DSPACE}[\log n]$, $\mathrm{NL} = \mathrm{NSPACE}[\log n]$, P is polynomial time, and NP is nondeterministic polynomial time. AC^0 is the set of problems accepted by uniform families of polynomial-size, constant-depth circuits whose gates include unary "not" gates, together with unbounded-fan-in "and" and "or" gates. NC^1 is the set of problems accepted by uniform families of polynomial-size, $O(\log n)$-depth circuits whose gates include unary "not" gates, together with binary "and" and "or" gates.

Each complexity class from Equation 1 has a natural descriptive characterization. Complexity classes are sets of decision problems. Each formula in a logic expresses a certain decision problem. As is standard, we write $C = \mathcal{L}$ to mean that the complexity class C is equal to the set of decision problems expressed by the logical language $\mathcal{L}$. The following descriptive characterizations of complexity classes are well known:

Fact 1 $\mathrm{FO} = \mathrm{AC}^0$*;* $\mathrm{NC}^1 = \mathrm{FO}(\mathit{Regular})$*;* $\mathrm{L} = \mathrm{FO}(\mathrm{DTC})$*;* $\mathrm{NL} = \mathrm{FO}(\mathrm{TC})$*;* $\mathrm{P} = \mathrm{FO}(\mathrm{IND})$*; and* $\mathrm{NP} = \mathrm{SO}\exists$*.*

We now explain some of the details of Fact 1. For more information about this fact the reader should consult one of the texts [10, 5, 16].

2.4 Transitive Closure Operators

Given a binary relation on k-tuples, $\varphi(x_1, \ldots, x_k, y_1, \ldots, y_k)$, we let $\mathrm{TC}_{\overline{x},\overline{y}}(\varphi)$ express its transitive closure. If the free variables are understood then we may abbreviate this as $\mathrm{TC}(\varphi)$. Similarly, we let $\mathrm{RTC}(\varphi)$, $\mathrm{STC}(\varphi)$, and $\mathrm{RSTC}(\varphi)$ denote the reflexive transitive closure, symmetric transitive closure, and symmetric and reflexive transitive closure of φ, respectively.

We next define a deterministic version of transitive closure DTC. Given a first order relation, $\varphi(\overline{x}, \overline{y})$, define its deterministic reduct,

$$\varphi_d(\overline{x}, \overline{y}) \quad \stackrel{\text{def}}{=} \quad \varphi(\overline{x}, \overline{y}) \wedge (\forall \overline{z})(\neg\varphi(\overline{x}, \overline{z}) \vee (\overline{y} = \overline{z}))$$

That is, $\varphi_d(\overline{x}, \overline{y})$ is true just if $\overline{y}$ is the unique child of $\overline{x}$. Now define $\mathrm{DTC}(\varphi) \stackrel{\text{def}}{=} \mathrm{TC}(\varphi_d)$ and $\mathrm{RDTC}(\varphi) \stackrel{\text{def}}{=} \mathrm{RTC}(\varphi_d)$.

Let $\tau_{gst} = \langle E^2; s, t; \rangle$ be the vocabulary of graphs with two specified points. The problem $\mathrm{REACH} = \{G \in \mathrm{STRUC}[\tau_{gst}] \mid G \models \mathrm{RTC}(E)(s,t)\}$ consists of all finite graphs that have a path from s to t. Similarly, $\mathrm{REACH}_d = \{G \in \mathrm{STRUC}[\tau_{gst}] \mid G \models \mathrm{RDTC}(E)(s,t)\}$ is the subset of REACH such that there is a unique path from s to t and all vertices along this path have out-degree

one. $\text{REACH}_u = \{G \in \text{STRUC}[\tau_{gst}] \mid G \models \text{STC}(E)(s,t)\}$ is the set of graphs having an undirected path from s to t.

It is well known that REACH is complete for NL, and REACH_d and REACH_u are complete for L [10, 19]. A simpler way to express deterministic transitive closure is to syntactically require that the out-degree of our graph is at most one by using a function symbol: denote the child of v as $f(v)$, with $f(v) = v$ if v has no outgoing edges. In this notation, a problem equivalent to REACH_d, and thus complete for L, is $\text{REACH}_f = \{G \in \text{STRUC}[\tau_{fst}] \mid G \models \text{RTC}(f)(s,t)\}$.

If $\mathcal{O}$ is an operator such as TC, let $\text{FO}(\mathcal{O})$ be the closure of first-order logic using $\mathcal{O}$. Then $\text{L} = \text{FO(DTC)} = \text{FO(RDTC)} = \text{FO(STC)} = \text{FO(RSTC)}$ and $\text{NL} = \text{FO(TC)} = \text{FO(RTC)}$.

2.5 Inductive Definitions

It is useful to define new relations by induction. For example, we can express the transitive closure of the relation E inductively, and thus the property REACH, via the following Datalog program:

$$\begin{aligned} E^*(x,x) &\leftarrow \\ E^*(x,y) &\leftarrow E(x,y) \\ E^*(x,y) &\leftarrow E^*(x,z), E^*(z,y) \\ \text{REACH} &\leftarrow E^*(s,t) \end{aligned} \tag{2}$$

Define FO(IND) to be the closure of first-order logic using such positive inductive definitions. The Immerman-Vardi Theorem states that $\text{P} = \text{FO(IND)}$. In this paper we will use stratified Datalog programs such as Equation 2 to express problems and then use ReductionFinder to automatically find reductions between them. Thus ReductionFinder can handle any problem in P or below. In the future we hope to handle problems in NP, but this will require us to go beyond SAT solvers to QBF solvers.

2.6 Reductions

Given a pair of problems $S \subseteq \text{STRUC}[\sigma]$ and $T \subseteq \text{STRUC}[\tau]$, a *many-one reduction* from S to T is an easy-to-compute function $f : \text{STRUC}[\sigma] \to \text{STRUC}[\tau]$ such that for all $\mathcal{A} \in \text{STRUC}[\sigma]$,

$$\mathcal{A} \in S \quad \Leftrightarrow \quad f(\mathcal{A}) \in T \ .$$

In descriptive complexity we use *first-order reductions* which are many-one reductions in which the function f is defined by a sequence of first-order formulas from $\mathcal{L}(\sigma)$, one for each symbol of τ. For example, the following is a reduction from REACH_f to REACH_u that ReductionFinder automatically found. Here $\sigma = \langle ; s, t; f^1 \rangle$ and $\tau = \langle E^2; s, t; \rangle$. The reduction, R_{fu}, is as follows:

$$\begin{aligned} E'(x,y) &\equiv y \neq t \ \wedge \ f(y) = x \\ s' &\equiv s \\ t' &\equiv t \end{aligned} \tag{3}$$

Note that the three formulas in R_{fu}'s definition (Equation 3) have no quantifiers, so R_{fu} is not only a first-order reduction, it is a *quantifier-free reduction* and we write $\text{REACH}_f \leq_{\text{qf}} \text{REACH}_u$.

More explicitly, for each structure $\mathcal{A} \in \text{STRUC}[\sigma]$, $\mathcal{B} = R_{fu}(\mathcal{A}) = \langle |\mathcal{A}|, E^{\mathcal{B}}, s^{\mathcal{B}}, t^{\mathcal{B}} \rangle$ is a structure in $\text{STRUC}[\tau]$ with universe the same as $\mathcal{A}$, and symbols given as follows:

$$\begin{aligned} E^{\mathcal{B}} &= \left\{ \langle a, b \rangle \ \middle| \ (\mathcal{A}, a/x, b/y) \models y \neq t \ \wedge \ f(y) = x \right\} \\ s^{\mathcal{B}} &= s^{\mathcal{A}} \\ t^{\mathcal{B}} &= t^{\mathcal{A}} \end{aligned}$$

In this paper we restrict ourselves to quantifier-free reductions. In general, a first-order reduction R has an arity which measures the blow-up of the size of the reduction. In [10] a first-order reduction of *arity* k maps a structure with universe $|\mathcal{A}|$ to a structure of universe $\left\{ \langle a_1, \ldots a_k \rangle \ \middle| \ (\mathcal{A}, a_1/x_1, \ldots, a_k/x_k) \models \varphi_0 \right\}$, i.e., a first-order definable subset of $|\mathcal{A}|^k$. However, increasing the arity of a reduction beyond two is rather excessive – arity two already squares the size of the instance. In this paper, in order to keep our reductions as small and simple as possible, we use a triple of natural numbers, $\langle k, k_1, k_2 \rangle$, to describe the universe of the image structure, namely

$$|R(\mathcal{A})| = |\mathcal{A}|^k \times \{1, \ldots, k_1\} \ \cup \ \{1, \ldots k_2\} \ . \tag{4}$$

That is in addition to raising the universe to the power k, we also multiply it by the constant k_1 and then we may add k_2 explicit constants to the universe. In this notation the above reduction R_{fu} has arity $\langle 1, 1, 0 \rangle$. It will become apparent in our many examples in the sequel how these extra parameters keep the reductions simple and small.

3 Strategy

We are given a pair of problems $S \subseteq \text{STRUC}[\sigma]$ and $T \subseteq \text{STRUC}[\tau]$, both expressed in Datalog. We want to know if there is a quantifier-free reduction from S to T.

It is not hard to see that this problem is undecidable, and in fact complete for the second level of the arithmetic hierarchy. It asks whether there exists some reduction that is correct for all inputs from $\text{STRUC}[\sigma]$, with no bounds on the size of the reduction nor the input.

We first make the problem more tractable by bounding the complexity of the reduction: We choose a triple $a = \langle k, k_1, k_2 \rangle$ describing the arity of the reduction and a tuple of parameters p bounding the size and complexity of the quantifier-free formulas expressing the reduction (e.g. how many clauses, the maximum

size of each clause, etc.). This reduces the complexity of the problem to co-r.e. complete: it is still undecidable.

To make the problem decidable, we choose a bound, n, and ask whether there exists a reduction of arity a and parameters p that is correct for all structures $A \in \text{STRUC}_{\leq n}[\tau]$, i.e, whose universes have cardinality at most n. Given such a reduction we can hope to prove by machine or hand that it works on structures of all sizes. On the other hand, being told that no such small reduction exists, we learn that in a precise sense there is no "simple" reduction from S to T.

Now our problem is complete for Σ_2^p – the second level of the polynomial-time hierarchy. Let $\mathbf{R}_{a,p}$ be the set of quantifier-free reductions of arity at most a and with parameter values at most p. The following formula asks whether there exists a quantifier-free reduction of arity a and parameters p that correctly reduces S to T on all structures of size at most n:

$$(\exists R \in \mathbf{R}_{a,p})(\forall \mathcal{A} \in \text{STRUC}_{\leq n}[\sigma])(\mathcal{A} \in S \leftrightarrow R(\mathcal{A}) \in T) \tag{5}$$

3.1 Solving a Σ_2^p Problem via Repeated Calls to a SAT Solver

We solve the problem expressed in Equation 5 by starting with a random structure $G_0 \in \text{STRUC}_{\leq n}[\sigma]$ and asking a SAT solver to find a reduction $R \in \mathbf{R}_{a,p}$ that works correctly on G_0, i.e., $G_0 \in S \leftrightarrow R(G_0) \in T$. If there is no solution, then our original problem is unsolvable.

Otherwise, we ask a new question to the SAT solver: is there some other structure, $G_1 \in \text{STRUC}_{\leq n}[\sigma]$ on which R fails, i.e, $G_1 \in S \leftrightarrow R(G_1) \notin T$. If not, then we know that R is a candidate reduction that is correct for all structures of size at most n.

However, if the SAT solver produces an example G_1 where R fails, we go back to the beginning, but now searching for a reduction that is correct on our full set of candidate structures, $\mathcal{G} = \{G_0, G_1\}$.

In summary, our algorithm proceeds as follows, with $\mathcal{G}$ initialized to $\{G_0\}$:

1. Using a SAT solver, search for a reduction correct on $\mathcal{G}$:

$$\textbf{find } R \in \mathbf{R}_{a,p} \text{ s.t. } \bigwedge_{G \in \mathcal{G}} G \in S \leftrightarrow R(G) \in T \tag{6}$$

   If no such R exists: **return**("no such reduction")

2. Using a SAT solver, search for some structure G on which R fails:

$$\textbf{find } G \in \text{STRUC}_{\leq n}[\sigma] \text{ s.t. } G \in S \leftrightarrow R(G) \notin T \tag{7}$$

   If no such G exists: **return**(R)
   Else: $\mathcal{G} = \mathcal{G} \cup \{G\}$; go to 1

Figure 1 shows a schematic view of this algorithm.

This procedure is correct because each new structure G eliminates at least one potential reduction. In our experience, the procedure works within a tractable

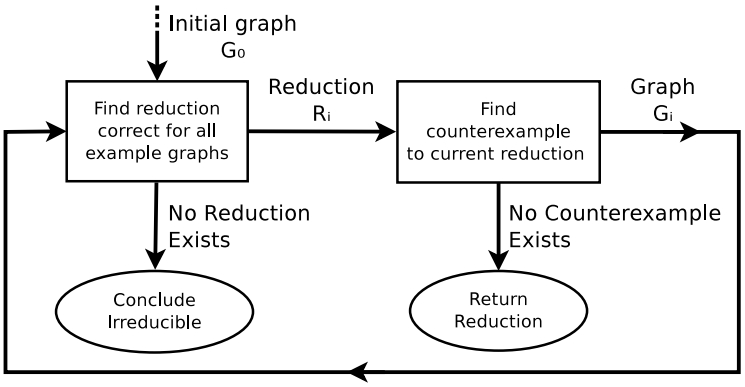

Fig. 1. A schematic view of the above algorithm.

number of structures; "smaller" searches have often completed after 5-15 sample structures, while the largest spaces searched by the program have required 30-50 iterations.

We begin searching for reductions at a very small size ($n = 3$); for search spaces without a correct reduction, even this small size is often enough to detect irreducibility. When a reduction is found at a particular size n, we examine larger structures for counterexamples; currently we look at structures of size at most $n + 2$. If a counterexample is found, we add it to $\mathcal{G}$, increment n and return to step 1.

Search time increases very rapidly as n increases. Of the 10,422 successful reductions found, 9,291 of them were found at size 3, 1076 at size 4, 38 at size 5, and 17 at sizes 6-8. See Section §5 for details of results. See Section §6 for more about the current limits of size and running time and our ideas concerning how to improve these.

4 Implementation

Figure 1 shows a schematic view of ReductionFinder's algorithm. The program is written in Scala, an object-oriented functional programming language implemented in the Java Virtual Machine[1]. ReductionFinder maintains a database of problems via a directed graph, G, whose vertices are problems. An edge (a, b) indicates that a reduction has been found from problem a to problem b, and is labelled by the parameters of a minimal such reduction that has been found so far.

When a new problem, c, is entered, ReductionFinder systematically searches for reductions to resolve the relationships between c and the problems already categorized in G.

[1] http://www.scala-lang.org

Given a pair of problems, c, d, specified in stratified Datalog, and a search space $\mathbf{R}_{a,p}$ specifying the arity a and parameters p, ReductionFinder calls the Cmodels 3.79 answer-set system[2] to answer individual queries of the form of Equations (6), (7). Cmodels in turn makes calls to SAT solvers. The SAT solvers we currently use are MiniSAT and zChaff [6, 18].

4.1 Problem input

Queries in ReductionFinder are input as small stratified-Datalog programs; a query on vocabulary τ has the symbols of τ available as extrinsic relations. The query is responsible for defining a single-bit intrinsic relation `satisfied`, representing the truth of the query. Input queries may use lparse rules without choice rules or disjunctive rules. When the input vocabulary contains function or constant symbols, these are translated by ReductionFinder into purely relational statements.

Equation (8) gives the ReductionFinder input for the directed-graph reachability query REACH $\subseteq$ STRUCT$[\langle E^2; s, t\rangle]$, corresponding to the inductive definition (2). We define an intrinsic relation `reaches` to compute the transitive closure of the edge relation `E`.

```
reaches(X, X).
reaches(X, Y) :- E(X, Y).
reaches(X, Y) :- reaches(X, Z), reaches(Z, Y).
satisfied :- reaches(s, t).
```
(8)

4.2 Search spaces

ReductionFinder restricts itself to searching for quantifier-free reductions, i.e. reductions defined by a set of quantifier-free formulas. The complexity of these quantifier-free formulas is restricted by several search parameters. The three arity numbers $\langle k, k_1, k_2\rangle$ of Section 2.6 each limit the search. The set of numeric predicates available (Section 2.2) is also a configurable parameter. The number of levels of nested function application available is a parameter.

Finally, the length of each quantifier-free formula is a parameter. Relations are defined by formulas represented in DNF; the number of disjuncts is a parameter, as is the number of conjuncts in each clause. Functions are defined as an if/else-if/else expression; the conditional of each statement is a conjunction of atomic formulas, and the resultant is a closed term. Again, the number of clauses is a parameter, as is the number of conjuncts in each clause.

The expressivity of the search space increases monotonically with most of our search parameters, inducing a natural partial ordering on search spaces. The search server respects this partial ordering, and avoids performing a search when any more-expressive space has previously been searched. The server is not

[2] http://www.cs.utexas.edu/users/tag/cmodels.html

restricted to increasing parameters one-at-a-time; since there are many search parameters, performing a single "large" search may be more efficient than performing many small searches. When a successful reduction is found, the server can automatically search smaller spaces to determine the smallest space containing a reduction.

4.3 The searching process

Once a search space and a pair of problems are fixed, ReductionFinder performs the iterative sequence of search stages described in section 3.1. Within each stage, ReductionFinder outputs a single lparse/cmodels program expressing Equations (6) or (7), and calls the Cmodels tool. The **find** statements in these equations are quantified explicitly using lparse's choice rules. The majority of the program is devoted to evaluation rules defining the structure $R(G)$ in terms of the sets of boolean variables R and G.

Figure 2 gives lparse code for a single counterexample-finding step (equation (7)). This code attempts to find a counterexample to a previously-generated reduction candidate. The specific code listed is examining reductions from REACH (Section 2.4) to its negation. The reduction candidate was $E'(x, y) \equiv (E(y, x) \wedge x = s) \vee E(x, x)$, $s' \equiv t$, $t' \equiv \mathrm{Suc}(\mathrm{min})$ (lines 7-9).

The counterexample is found using lparse's choice rules as existential quantifiers, directly guessing the relation `in_E` and the two constant symbols `in_s` and `in_t` (lines 12-13). Since lparse does not contain function symbols, these constants are implemented as degree-1 relations which are true at exactly one point. We specify the constraint that we cannot have `in_satisfied` == `out_satisfied` (line 16); these boolean variables will be defined later in the program, and this constraint will ensure that our graph is a counterexample to the reduction candidate.

Defining `in_satisfied` and `out_satisfied` in terms of the input and output predicates (respectively) is easy. We have already required the user to input lparse code for the input and output queries. We do some minimal processing on this code, disambiguating names and turning function symbols into relations. The user's input for directed-graph reachability, listed in Equation (8), is translated into the input query block of lines 19-22. Similarly, the output query is translated into lines 25-28.

The remainder of the lparse code exists to define the output predicates (in this case `out_E`, `out_s`, `out_t`) in terms of the input predicates and the reduction. In building the output reduction `out_E(X, Y)`, we first build up a truth table for each of the atomic formulas used; for example, line 31 states that term `e_y_x` is true at point `(X, Y)` exactly if $E(Y, X)$ in the input structure. Each position in the DNF definition is true at `(X, Y)` exactly if the atomic formula chosen for that position is true (lines 36-37). The output relation `out_E(X, Y)` is then defined via the terms in the DNF (lines 38-39). The code in lines 30-39 thus defines the output relation `out_E(X, Y)` in terms of the input relations `in_E`, `in_s`, `in_t` and the reduction candidate `reduct_E`.

```
node(n1; n2; n3; n4).                                                    1
atomic(e_x_x; e_x_y; ...; x_eq_t; y_eq_t).                               2
closedterm(fn_s; fn_t; fn_min; fn_succ_min; fn_max).                     3
position(pos_0_0; pos_0_1; pos_1_0).                                     4
                                                                         5
%%% Import reduction candidate from previous stage.                      6
reduct_E(pos_0_0, e_y_x). reduct_E(pos_0_1, x_eq_s).                     7
reduct_E(pos_1_0, e_x_x).                                                8
reduct_s(fn_t).          reduct_t(fn_succ_min).                          9
                                                                         10
%%% Guess input relations E, s, t.                                       11
{ in_E(X, Y) }.                                                          12
1 { in_s(X) } 1. 1 { in_t(X) } 1. % Choose exactly one s, t.             13
                                                                         14
%%% A constraint on the entire program:                                  15
:- out_satisfied == in_satisfied.                                        16
                                                                         17
%%% Translated version of input query.                                   18
in_Reaches(X, X).                                                        19
in_Reaches(X, Y) :- in_E(X, Y).                                          20
in_Reaches(X, Y) :- in_Reaches(X, Z), in_Reaches(Z, Y).                  21
in_satisfied :- in_Reaches(X, Y), in_s(X), in_t(Y).                      22
                                                                         23
%%% Translated version of output query.                                  24
out_Reaches(X, X).                                                       25
out_Reaches(X, Y) :- out_E(X, Y).                                        26
out_Reaches(X, Y) :- out_Reaches(X, Z), out_Reaches(Z, Y).               27
out_satisfied :- not out_Reaches(X, Y), out_s(X), out_t(Y).              28
                                                                         29
%%% Define a truth table for each atomic relation in the reduction.      30
true(e_y_x, X, Y) :- in_E(Y, X).                                         31
true(x_eq_s, X, Y) :- in_s(X).                                           32
true(e_x_x, X, X) :- in_E(X, X).                                         33
                                                                         34
%%% Use these truth tables to evaluate output relations.                 35
true(P, X, Y) :- reduct_E(P, A), true(A, X, Y),                          36
                 position(P), atomic(A).                                 37
out_E(X, Y) :- true(pos_0_0, X, Y), true(pos_0_1, X, Y).                 38
out_E(X, Y) :- true(pos_1_0, X, Y).                                      39
                                                                         40
%%% Similarly, define the evaluation of each closed term.                41
eval_term(fn_s, X) :- in_s(X).                                           42
eval_term(fn_succ_min, n2).                                              43
                                                                         44
%%% Define the output relations.                                         45
out_s(X) :- reduct_s(F), eval_term(F, X), closedterm(F).                 46
out_t(X) :- reduct_t(F), eval_term(F, X), closedterm(F).                 47
```

Fig. 2. Lparse code for a single search stage. This code implements equation (7), searching for a 4-node counterexample for a candidate reduction from REACH (Section 2.4) to its negation. Variables `X`, `Y`, `Z` range over `node`s.

Lines 41-47 similarly define the output constants `out_s` and `out_t`. Since lparse does not provide function symbols, we define these constants as unary relations `out_s(X)`, making sure that these relations are true at exactly one point. We are thus able to define the output constants in terms of the input symbols `in_s`, `in_t` and the the reduction candidate's definitions of s', t' (`reduct_s`, `reduct_t`).

The code for finding a reduction candidate (equation (6)) is very similar to the counterexample-finding code in Figure 2. We import the list $\mathcal{G}$ of counterexample graphs, and must guess a reduction. The input query, output vocabulary, and output query are evaluated for each graph. Truth tables must be built for each relation which might appear in the reduction, and for each graph.

4.4 Timing

ReductionFinder uses the Cmodels logic programming system to solve its search problems. The Cmodels system solves answer-set programs, such as those in the lparse language, by reducing them to repeated SAT solver calls. Direct translations from answer-set programming (ASP) to SAT exist[2, 12], but introduce new variables; Lifschitz and Razborov have shown that, assuming the widely-believed conjecture P $\not\subseteq$ NC^1/poly, any translation from ASP must either introduce new variables or produce a program of worst-case exponential length [17].

The Cmodels system first translates the lparse program to its Clark completion [3], interpreting each rule `a :- b` as merely logical equivalence (`a` $\Leftrightarrow$ `b`). Models of this completion may fail to be answer sets if they contain *loops*, sets of variables which are true only because they assume each other. If the model found contains a loop, Cmodels adds a *loop clause* preventing this loop and continues searching, keeping the SAT solver's learned-clause database intact. A model which contains no loops is an answer set, and all answer sets can be found in this way.

The primary difficulty in finding large reductions with ReductionFinder has been computation time. The time spent finding reductions dominates over the time spent finding counterexamples; reductions must be true on each of the example graphs, and the number of lparse clauses and variables thus scales linearly with the number of example graphs. The amount of time required by Cmodels seems highly correlated with the number of loop formulas which must be generated; Figure 3 shows the time for each reduction-finding stage during a several-hour arity 2 search, versus the number of loop formulas generated in the stage. The final reduction-finding step generated an lparse program with 399,900 clauses, using 337,605 atoms.

5 Results

5.1 Size and timing data

We have run ReductionFinder for approximately 5 months on an 8-core 2.3 GHz Intel Xeon server with 16 GB of RAM. As of this writing, ReductionFinder has

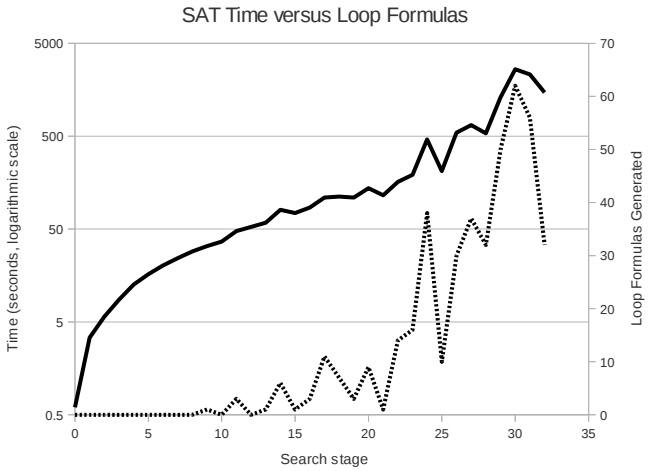

Fig. 3. Timing data for a run reducing $\neg\mathbf{RTC}[f](s,t) \leq \mathbf{RTC}[f](s,t)$ at arity 2, size 4. The solid line shows time to find each reduction candidate in seconds, on a logarithmic scale. The dotted line shows the number of loop formulas generated by Cmodels, and thus the number of SAT solver calls for each reduction candidate. This run was successful in finding a reduction.

performed 331,036 searches on a database of 87 problems. Of the 7482 pairs of distinct problems, we explicitly found reductions between 2698; an additional 803 reductions could be concluded transitively. 23 pairs were manually marked as irreducible, comprising provable theorems about first-order logic plus statements that $\mathrm{L} \lneq$ (co-)$\mathrm{NL} \lneq \mathrm{P}$. From these 23, an additional 3043 pairs were transitively concluded to be irreducible. 915 pairs remained unfinished.

For many of the pairs which we reduced successfully, we found multiple successful reductions. Sometimes this occurred when we first found the reduction in a large search space, then tried smaller spaces to determine the minimal spaces containing a reduction. More interestingly, some pairs contained multiple successful reductions in distinct minimal search spaces, demonstrating trade-offs between different measures of the reduction's complexity. Some of these trade-offs were uninteresting: a reduction which simply needs "some distinguished constant" could use min, max, or c_1. Others, however, began to show non-trivial trade-offs between the formula length required and the numerics or arity available. See Equations (9), (10) for an example. Of the 12,149 correct reductions found between the 2698 explicitly-reduced pairs of problems, 5091 were in some minimal search space.

5.2 A map of complexity theory

Figure 4 shows classes of queries within the ReductionFinder database. Each class contains one or more query which ReductionFinder has shown equivalent via quantifier-free reductions. An edge from class I to class J indicates that

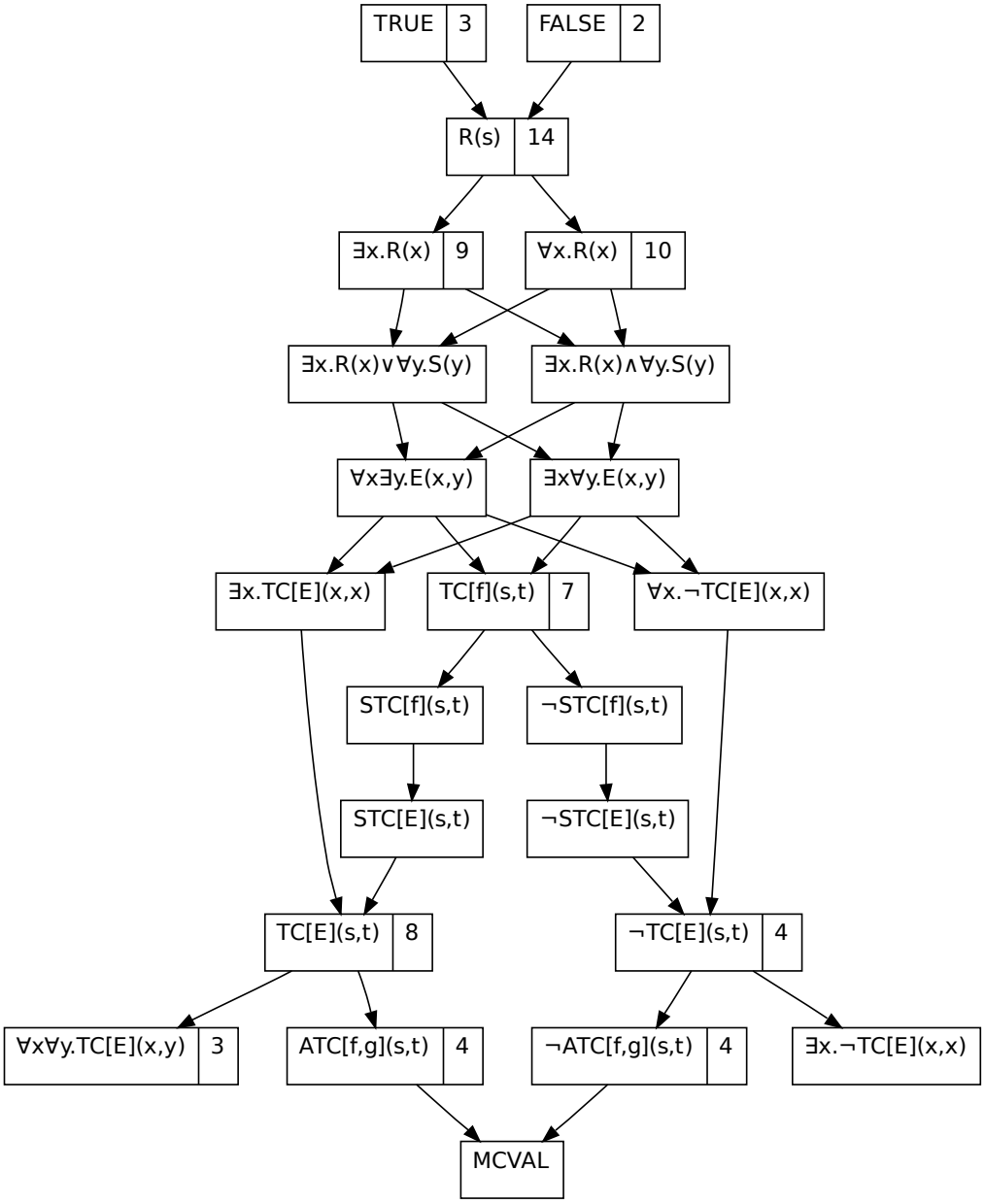

Fig. 4. A map of reductions in the query database. Nodes without numbers represent a single query. A node with number n represents n queries of the same complexity. Some queries are elided for clarity.

ReductionFinder has reduced $I \leq_{\text{qfp}} J$. Numbers on the graph indicate the number of queries the class contains; the contents of these classes are listed in Figure 5.

FALSE	$R(s) \wedge \neg R(s)$		
TRUE	$R(s) \vee \neg R(s)$	$\exists x.\mathbf{TC}[f](x,x)$	
$R(s)$	$\neg R(s)$	$R(f(s))$	
$E(s,t)$	$E(s,s)$	$E(s,t) \vee E(t,s)$	
$s = t$	$s \neq t$	$f(s) = t$	$f(s) \neq t$
$f(s) = s$	$f(s) = g(s)$	$f(s) = t \wedge f(t) = s$	$f(s) = t \vee f(t) = s$
$\exists x.R(x)$	$\exists x.R(x) \wedge sS(x)$	$\exists x.R(x) \vee S(x)$	
$\exists xy.E(x,y)$	$\exists xy.\neg E(x,y)$	$\exists xy.E(x,y) \wedge E(y,x)$	$\exists x.E(x,s)$
$\exists x.f(x) = x$	$\exists x.f(x) = s$		
$\forall x.R(x)$	$\forall x.\neg R(x)$	$\forall x.R(x) \wedge S(x)$	$\forall x.R(x) \vee S(x)$
$\forall xy.E(x,y)$	$\forall xy.\neg E(x,y)$	$\forall x.E(x,s)$	
$\forall x.f(x) = s$	$\forall x.f(x) = x$	$\forall x \neq y.\ f(x) \neq f(y)$	
$\mathbf{TC}[f](s,t)$	$\mathbf{RTC}[f](s,t)$	$\mathbf{TC}[f](s,s)$	
$\neg\mathbf{TC}[f](s,t)$	$\neg\mathbf{RTC}[f](s,t)$	$\neg\mathbf{TC}[f](s,s)$	
$(\exists y.T(y))\ \mathbf{RTC}[f](s,y)$			
$\mathbf{TC}[E](s,t)$	$\mathbf{RTC}[E](s,t)$	$\mathbf{TC}[E](s,s)$	
$\mathbf{TC}[f,g](s,t)$	$\mathbf{RTC}[f,g](s,t)$	$\mathbf{RTC}[f,g](s,s)$	
$(\exists y.T(y))\ \mathbf{RTC}[E](s,y)$		$(\exists xy.\ S(x) \wedge T(y))\ \mathbf{RTC}[E](x,y)$	
$\neg\mathbf{TC}[E](s,t)$	$\neg\mathbf{RTC}[E](s,t)$	$\neg\mathbf{TC}[E](s,s)$	$\neg\mathbf{RTC}[f,g](s,t)$
$\forall xy.\mathbf{TC}[E](x,y)$	$\forall x.\mathbf{TC}[E](x,t)$	$\forall x.\mathbf{TC}[E](x,x)$	
4 variations of **ATC**			

Fig. 5. A list of problems in the complexity classes of Figure 4. ReductionFinder has found a reduction between each pair of problems in each box. Each problem is expressed as a logical formula.

ReductionFinder has placed all of the computationally-simple problems into their correct complexity classes. The trivially-true query and trivially-false query were reduced to all other queries. The class $R(s)$ contains twelve queries which lack the power of even one first-order quantifier. The classes $\exists x.R(x)$ and $\forall x.R(x)$ contain many variations of first-order quantifiers; for example, $\exists x.R(x)$ includes $\exists xy.E(x,y)$, $\exists x.f(x) = s$, $\exists x.E(s,x)$. Below this, the structure of **FO** under quantifier-free reductions is correctly represented up to two quantifier alternations.

Beyond **FO**, ReductionFinder has made significant progress in describing the complexity hierarchy. A class of 7 L-complete problems is visible at $\mathbf{TC}[f](s,t)$ (deterministic reachability), including its complement ($\neg\mathbf{TC}[f](s,t)$) and deterministic reachability with a relational target ($\exists y.T(y) \wedge \mathbf{TC}[f](s,y)$). Unfortunately, the L-complete problems of cycle-finding ($\exists x.\mathbf{TC}[E](x,x)$) and its nega-

tion have not been placed in this class; nor has deterministic reachability with relations as both source and target ($\exists xy.S(x) \wedge T(y) \wedge \mathbf{TC}[E](x,y)$).

Below this level, ReductionFinder had limited success. We succeeded in reducing several problems to reachability (see Figure 5), including degree-2 reachability (reduction described in section 5.3. Not surprisingly, we did not discover a proof of the Immerman-Szelepcsényi theorem (showing co-NL $\leq$ NL by providing a reduction $\neg\mathbf{TC}[E](s,t) \leq \mathbf{TC}[E](s,t)$). We similarly did not prove Reingold's theorem [19], showing SL $\leq$ L by reducing $\mathrm{STC}[E](s,t) \leq \mathbf{TC}[f](s,t)$. These two results were historically elusive, and may require reductions above arity 2, or longer formulas than we were able to examine. Considering P-complete problems, we proved the equivalence of several variations of alternating transitive closure (**ATC**); however, we did not show the problem equivalent to its negation, or to the monotone circuit value problem (**MCVAL**).

5.3 Sample reductions

We now list a few of the reductions that ReductionFinder has produced.

Example 1 ReductionFinder found two arity-1 reductions showing $\mathbf{RTC}[E](s,t) \leq \forall x.\mathbf{TC}[E](x,x)$. The first of these problems is simply REACH; the second states that every node of a directed graph is on some (nontrivial) cycle. The two reductions are good examples of the arity-1 reductions we have found, and also show a clear tradeoff between the formula length required to define E' and the arity parameters:

$$\begin{aligned} |R(\mathcal{A})| = &\quad \{a_1, a_2, \ldots, a_n, c_1\} \\ E'(x,y) \equiv &\quad x = t \\ &\vee\ y = s \\ &\vee\ E(x,y) \end{aligned} \tag{9}$$

The output structure $R(\mathcal{A})$ has all of the elements of the input structure $\mathcal{A}$, plus one new point c_1. The new edge relation is true wherever the old edge relation was true; in addition, all possible edges into the source and out of the target are added.

Since the new point c_1 was not part of the original edge relation, it has only one outgoing edge (to s), and only one incoming edge (to t). Therefore c_1 is on a cycle iff there is a path in the original graph from s to t. Similarly, if such a path does exist, *every* node in $R(\mathcal{A})$ is on a similar cycle. Thus the input graph satisfies $\mathbf{RTC}[E](s,t)$ iff the output satisfies $\forall x.\mathbf{TC}[E](x,x)$.

In addition to this reduction, ReductionFinder found a second arity-1 reduction. The second reduction does not use a distinguished constant element, but requires a longer formula:

$$
\begin{aligned}
|R(\mathcal{A})| = &\quad \{a_1, a_2, \ldots, a_n\} \\
E'(x,y) \equiv &\quad y \neq s \wedge E(x,y) \\
&\vee\ x \neq s \wedge x = y \\
&\vee\ x = t
\end{aligned}
\tag{10}
$$

This reduction can be viewed as manipulating the graph as follows: we first remove all edges into s. We then add a self-loop on every edge except s. Finally, we add all possible edges out of t. Since the edge (t, s) is the only edge into node s, we then have that the node s is on a cycle iff there is a path from s to t. (Every other node is on a trivial cycle by construction.)

ReductionFinder has verified that neither reduction can be shortened; there is a tradeoff between the availability of the extra element c_1 and the required formula length. ReductionFinder can detect such tradeoffs, because in the partial ordering induced by our various search parameters, each of these reductions is in a minimal reduction-containing space.

Example 2 ReductionFinder successfully reduced the first-order problem $\forall x \exists y.E(x,y)$ to deterministic reachability ($\mathbf{TC}[f](s,t)$). This is a simple example of an arity-2 reduction where the successor relation is used to iteratively check all elements.

$$
\begin{aligned}
|R(\mathcal{A})| &= \{\langle a_1, a_1\rangle, \langle a_1, a_2\rangle, \ldots, \langle a_n, a_n\rangle\} \\
f'(\langle x, y\rangle) &\equiv \begin{cases} \text{if} & E(x,y) & \text{then} & \langle \mathrm{Suc}(x), \mathrm{Suc}(x)\rangle \\ \text{else if} & (\mathrm{Suc}(y) \neq x) & \text{then} & \langle x, \mathrm{Suc}(y)\rangle \\ \text{else} & & & \langle x, y\rangle \end{cases} \\
s' &\equiv \langle \min, \min\rangle \\
t' &\equiv \langle \min, \min\rangle
\end{aligned}
\tag{11}
$$

Recall that each element in the output structure is a pair of elements in the input structure.

Deterministic non-reachability to deterministic reachability Like all deterministic classes, L is closed under complement. The canonical L-complete problem is deterministic reachability. ReductionFinder was able to find a version of the canonical reduction from deterministic non-reachability to deterministic reachability, showing co-L $\leq$ L.

$$
\begin{aligned}
|R(\mathcal{A})| &= \{\langle a_1, a_1\rangle, \langle a_1, a_2\rangle, \ldots, \langle a_n, a_n\rangle, c_1, c_2\} \\
f'(\langle x, y\rangle) &\equiv \begin{cases} \text{if} & (x = t) & \text{then} & c_2 \\ \text{else if} & (y = \max) & \text{then} & c_1 \\ \text{else} & & & \langle f(x), \mathrm{Suc}(y)\rangle \end{cases} \\
f'(c_i) &\equiv c_i \\
s' &\equiv \langle s, \min\rangle \\
t' &\equiv c_1
\end{aligned}
\tag{12}
$$

An input graph $G = \langle f; s, t\rangle$ contains no path from s to t iff the output graph $I(G) = \langle f'; s', t'\rangle$ contains a path from s to t. This arity-2 reduction walks through the original graph in the sequence $\langle s, 0\rangle$, $\langle f(s), 1\rangle$, $\ldots$, $\langle f^n(s), n\rangle$. If t is ever found, we move to the point c_2, representing a reject state; if t is not found after n steps, we move to the target node c_1.

Reachability to Degree-2 Reachability Directed-graph reachability is the canonical NL-complete problem, and it is well-known that restricting ourselves to graphs with outdegree ≤ 2 suffices for NL-completeness. We chose to represent outdegree-2 reachability with two unary function symbols; we define $\mathbf{TC}[f, g](s, t)$ on the vocabulary $\langle; f^1, g^1; s, t\rangle$, with the semantics that nodes can be reached through any combination of f-edges and g-edges. ReductionFinder succeeded in reducing $\mathbf{TC}[E](s, t) \leq \mathbf{TC}[f, g](s, t)$ via an arity-2 reduction:[3]

$$
\begin{aligned}
|R(\mathcal{A})| &= \{\langle a_1, a_1\rangle, \langle a_1, a_2\rangle, \ldots, \langle a_n, a_n\rangle\} \\
f'(\langle x, y\rangle) &\equiv \begin{cases} \text{if} & E(x, y) & \text{then} & \langle y, y\rangle \\ & & \text{else} & \langle x, y\rangle \end{cases} \\
g'(\langle x, y\rangle) &\equiv \langle x, \mathrm{Suc}(y)\rangle \\
s' &\equiv \langle s, t\rangle \\
t' &\equiv \langle t, t\rangle
\end{aligned}
\tag{13}
$$

This reduction uses the traditional technique of using successor to iterate through possible neighbors. Each node $\langle x, y\rangle$ of the output structure can be read as "we are at node x, considering y as a possible next step". If there is an edge $E(x, y)$, we nondeterministically either follow this edge (moving along f to $\langle y, y\rangle$) or

[3] The reduction above has undergone some syntactic simplification. ReductionFinder originally reported the reduction:

$$
\begin{aligned}
f'(\langle x, y\rangle) &\equiv \begin{cases} \text{if} & E(x, y) & \text{then} & \langle y, y\rangle \\ & & \text{else} & \langle x, \mathrm{Suc}(x)\rangle \end{cases} \\
g'(\langle x, y\rangle) &\equiv \begin{cases} \text{if} & \mathrm{Suc}(y) = x & \text{then} & \langle x, \mathrm{Suc}(x)\rangle \\ & & \text{else} & \langle x, \mathrm{Suc}(y)\rangle \end{cases} \\
s' &\equiv \langle s, t\rangle \\
t' &\equiv \langle t, t\rangle
\end{aligned}
$$

move along g to the next possibility $\langle x, \mathrm{Suc}(y)\rangle$. If there is no edge $E(x, y)$, our only nontrivial movement is along g, to $\langle x, \mathrm{Suc}(y)\rangle$.

6 Conclusions and Future Directions

The ReductionFinder program successfully finds quantifier-free reductions between computational problems. The program maintains a database of known reductions between problems. Strongly connected components in this database correspond to complexity classes. When presented with a new problem, we can perform searches to automatically place the problem within the existing reduction graph.

This project has demonstrated that it is possible to find reductions between problems by using a SAT solver to search for them. Right now, ReductionFinder takes a long time to find small reductions and cannot find medium-sized reductions. We suggest some directions for future work aimed at taking automatic reduction finding to the next stage.

1. ReductionFinder searches for a small, simple reduction, R, by repeatedly calling a SAT solver as outlined in §3.1. The tasks involved are:
   - Find an R that is a correct reduction on the current example graphs, $G_0, \ldots, G_k$ (Equation 6).
   - Find a G_{k+1} on which the current R fails (Equation 7).

   While, we would expect that such a search is exponential in the size of R, in our experience the difficulty is that the number of variables in the boolean formulas grow linearly with the number of counter-example graphs, k, and unfortunately the running time seems to increase exponentially in k. (The search for counter-example graphs in the second case does not have this problem.) Since the problem we are trying to solve is Σ_2^p – there exists a small reduction, for all small graphs – we hope to speed up our search by using strategies similar to those employed by QBF solvers. Related to this is the question of what makes a good set of counter-example graphs.
2. To show that there is a reduction from problem A to problem B, it may be that we can find a problem in the middle, M, so that reductions from A to M and M to B are simpler. We believe that finding such intermediate problems will be invaluable in searching for reductions. However, we have only found limited evidence of this so far in our work with ReductionFinder. It will be valuable to develop heuristics to find or generate appropriate intermediate problems.
3. Sufficient progress on the above two points may enable us to automatically generate linear reductions. This would have great benefits for automatic programming of optimal algorithms as discussed in Item 3 near the end of Section 1.

References

1. Eric Allender, Michael Bauland, Neil Immerman, Henning Schnoor, and Heribert Vollmer, "The Complexity of Satisfiability Problems: Refining Schaefer's Theorem" *J. Comput. Sys. Sci.* 75 (2009), 245–254.
2. R. Ben-Eliyahu and R. Dechter, "Propositional semantics for disjunctive logic programs", *Annals of Mathematics and Artificial Intelligence* 12 (1996) 53–87.
3. K. Clark, "Negation as Failure", *Logic and Data Bases*, H. Gallaire and J. Minker, Eds., Plenum Press, New York 293–322.
4. Stephen Cook, "The Complexity of Theorem Proving Procedures," *Proc. Third Annual ACM STOC Symp.* (1971), 151–158.
5. Heinz-Dieter Ebbinghaus and Jörg Flum, *Finite Model Theory, Second Edition,* 1999, Springer-Verlag.
6. Niklas Eén, Niklas Sörensson, "An Extensible SAT-solver [extended version 1.2]", *SAT*, 2003.
7. Tomás Feder and Moshe Vardi, "The Computational Structure of Monotone Monadic SNP and Constraint Satisfaction: A Study Through Datalog and Group Theory," *SIAM J. Comput.* 28 (1999) 57–104.
8. Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea, "SAT-Based Answer Set Programming," *Proc. AAAI* (2004), 61-66.
9. Juris Hartmanis, Neil Immerman, and Stephen Mahaney, "One-Way Log Tape Reductions," *IEEE Found. of Comp. Sci. Symp.* (1978), 65–72.
10. Neil Immerman, *Descriptive Complexity*, 1999, Springer Graduate Texts in Computer Science, New York.
11. Neil Immerman, "Languages That Capture Complexity Classes," *SIAM J. Comput.* **16**, No. 4 (1987), 760–778.
12. T. Janhunen, "A counter-based approach to translating normal logic programs into sets of clauses", *Proc. ASP'03 Workshop*, 166–180.
13. Neil Jones, "Reducibility Among Combinatorial Problems in Log n Space," *Proc. Seventh Annual Princeton Conf. Info. Sci. and Systems* (1973), 547–551.
14. Richard Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computations,* R.E.Miller and J.W.Thatcher, eds. (1972), Plenum Press, 85–104.
15. Richard Ladner, "On the Structure of Polynomial Time Reducibility," *J. Assoc. Comput. Mach.* 2(1) (1975), 155–171.
16. Leonid Libkin, *Elements of Finite Model Theory,* 2004, Springer.
17. Vladimir Lifschitz and Alexander A. Razborov, "Why are there so many loop formulas?", *ACM Trans. Comput. Log.* 7(2) (2006), 261–268.
18. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malike, "Chaff: Engineering an Efficient SAT Solver," in *Design Automation Conference 2001.*
19. Omer Reingold, "Undirected ST-connectivity in Log-Space," *ACM Symp. Theory Of Comput.* (2005), 376–385.
20. Thomas Schaefer, "The Complexity of Satisfiability Problems," *ACM Symp. Theory Of Comput.* (1978), 216–226.
21. J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: an Introduction to SETL*, 1986, Springer-Verlag, New York.
22. Leslie Valiant, "Reducibility By Algebraic Projections," *L'Enseignement mathématique,* **T. XXVIII**, 3–4 (1982), 253–268.