Finding Reductions Automatically

Arshdeep Singh Bhogal

June 3, 2022

1 Introduction

A reduction is a technique for mapping one problem to another. A good reduction can be used to demonstrate that the second problem is at least as complex as the first. However, for reductions to be beneficial, they must be simple. For example, a tough NP-complete problem like the Boolean satisfiability problem can be reduced to a trivial problem like determining if a number equals zero by having the reduction machine tackle the problem in exponential time and output zero only if a solution exists. This, however, accomplishes nothing since, while we can solve the new problem, doing the reduction is just as difficult as solving the old problem. Likewise, a reduction computing a non-computable function can reduce an undecidable problem to a decidable one. As Michael Sipser points out in 'Introduction to the Theory of Computation': "The reduction must be easy, relative to the complexity of typical problems in the class [...] If the reduction itself were difficult to compute, an easy solution to the complete problem wouldn't necessarily yield an easy solution to the problems reducing to it."

Many researchers have observed that natural problems remain complete for natural complexity classes under surprisingly weak reductions including log space reductions, one way log space reductions, projections, first order projections, and even the astoundingly weak quantifier free projections [Imm87, Jon73]. 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?". We will look at different ways to approach it.

The report is organised as follows: We start in Section 2 by understanding how reduction can be found manually between 3-colourable to 3-sat. With section 3-5, we establish background in descriptive complexity sufficient for the reader to understand all they need to know about reductions and the logical descriptions of decision problems. Sections 6 presents a way to find gadgets automatically between the problems described in section 2. And section 7 presents an approach to verify these reductions.

2 Manual Reduction from 3-SAT to 3-COLOURABLE

From 3-SAT to 3-coloring, we find a reduction. We accomplish this by constructing structures known as gadgets that follow a set of rules. This reduction will necessitate the use of three components:

1. Truth gadget: A triangle with three vertices T, F, and B, which intuitively stand for true, false, and base. which stand for true, false, and base, respectively. Because these vertices are all linked, any 3-coloring must have distinct colours for them. We'll call those colours True, False, and Basic. Thus, when we say that a node is coloured True, all we mean is that it must be coloured the same as the node T.

2. Variable gadget: A variable v is also a triangle joining two new nodes v and v to base in the truth gadget. Node v must be coloured either True or False, and so node v must be coloured either False or True, respectively.

3. Clause gadget: Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabelled nodes and ten edges; see the figure below. If all three literal nodes in the clause gadget are colored False, then the rightmost vertex in the gadget cannot have one of the three colors. Since the variable gadgets force each literal node to be colored either True or False, in any valid 3-coloring, at least one of the three literal nodes is colored True.

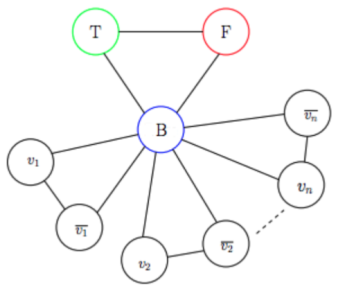

Figure 1: Variable Gadget for the reduction.

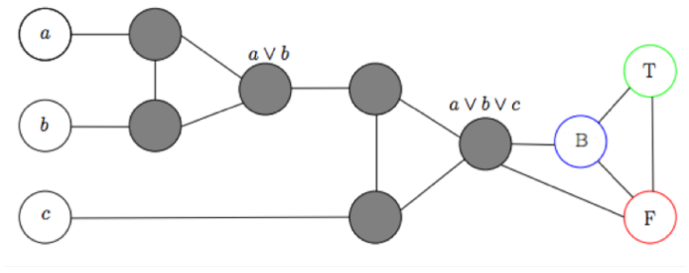

Figure 2: Clause Gadget for the reduction.

Applying the above transformation to a formula, we now prove that our CNF is satisfiable if and only if the above graph is 3-colourable. Suppose ϕ is satisfiable and let $(x_1^*, x_2^*, ..., x_n^*)$ be the satisfying assignment. If x_i^* is assigned True, we colour v_i as true and v_i as false (recall they're connected to the Base vertex, coloured B, so this is a valid colouring). Since ϕ is satisfiable, every clause $C_i = (a \vee b \vee c)$ must be satisfiable, i.e., at least of a, b, c is set to True. By the second property of the OR gadget, we know that the gadget corresponding to C_i can be 3-coloured so that the output node is coloured as true. And because the output node is adjacent to the False and Base vertices of the initial triangle only, this is a proper 3-colouring. Conversely, suppose G is 3-colourable. We construct an assignment of the literals of ϕ by setting x_i to True if v_i is coloured as true and vice versa. Now suppose this assignment is not a satisfying assignment to ϕ, then this means there exists at least one clause $C_i = (a \vee b \vee c)$ that was not satisfiable. That is, all of a, b, c were set as False. But if this is the case, then the output node of corresponding OR gadget of C_i must be coloured as false. But this output node is adjacent to the false vertex; thus, contradicting the 3-colourability of G.

Therefore, we've given a reduction [Mou14] from 3-SAT to 3-COLORING.

3 Descriptive Complexity

In this section we present background and notation from descriptive complexity theory concerning the representation of decision problems and reductions between them. 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. This allows us to express interesting reductions succinctly, and formulas, unlike Turing machines, have natural normal forms.

Two fundamental translations from computational complexity to logic [Dea21] are the following:

- The parallel time needed to check whether an input structure has a property is equal to the depth needed to describe that property in a first order inductive definition; and
- The amount of memory needed to check whether an input structure has a property is characterized by the number of distinct variables needed to describe that property in first order logic.

In general, the computational complexity of checking whether an input has a given property can be exactly characterized as the richness of a logical language needed to express that property. Thus, the secrets of computation can be understood with logic. For this reason, it is of great use to have tools such as DE to help us manipulate and reason about the objects in question, i.e., structures and queries. The remaining section explains standard notations for descriptive complexity [Imm98].

Definition 3.1 (Vocabulary). $\tau = ({R_1}^{a_1}, \dots, {R_r}^{a_r}; c_1, \dots, c_r; R{f_1}^{r_1}, \dots, f_t^{r_t})$

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.

Definition 3.2 (Structures). $A = (|A|; R_1^A, \dots, R_r^A; c_1^A, \dots, c_8^A; f_1^A, \dots, f_t^A)$

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

Definition 3.3 (Queries). $I : STRUC[\sigma] \Rightarrow STRUC[\tau]$

Any mapping I is a query from structures of one vocabulary to structures of another vocabulary, that is polynomial bounded.

Definition 3.4 (Reductions). $A \in S \Leftrightarrow R(A) \in T$.

Any query that satisfies the above condition, can be termed as a reduction. In general, if S and T are finite sets of structures of vocabulary σ and τ, respectively, and R is a first-order query from $STRUC[\sigma]$ to $STRUC[\tau]$, then R is a first-order reduction from S to T if and only if for all finite structures $A \in STRUC[\sigma]$ above condition is satisfied.

However, first-order logic has several drawbacks when used to express properties. First, 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 $TC[x1, x2.\phi(x1, x2)](y1, y2)$. This formula takes the transitive and reflexive closure of the relation defined by $\phi(x1, x2)$ and then evaluates it on $(y1, y2)$.

4 Existing Work

4.1 Reduction Finder

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 problem of the form $\exists X \forall Y \phi$ where X and 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.

ReductionFinder searches for a small, simple reduction, R, by repeatedly calling a SAT solver. The following figure outlines the steps.

1. Find an R that is a correct reduction on the current example graphs.
2. Find a G (counterexample) on which the current R fails and restart.
3. If no example exists, we have found a valid reduction in out current search space.

While we would expect that such a search is exponential in the size of R, in their experience[CIM10] the difficulty is that the number of variables in the Boolean formulas grow linearly with the number of counter-example graphs. It takes a long time to find small reductions and cannot find medium-sized reductions.

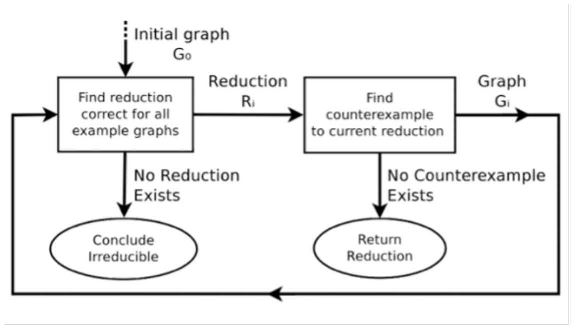

Figure 3: Algorithm for Reduction Finder.

4.2 What is DE?

In this section we will introduce DE and present its basic functionality. We will discuss queries (briefly described above but now with DE syntax): the formalism for expressing properties, transforming structures, and building reductions between problems.

The secrets of computation can be understood with logic. For this reason, it is of great use to have tools such as DE to help us manipulate and reason about the objects in question, i.e., structures and queries. Relatively simple queries suffice to construct many of the objects of interest. Furthermore, it is possible to program via reductions, i.e., build a carefully constructed and optimized program for a complete problem, C, and then solve other problems by simply writing the reduction to C. An extremely successful example of this point of view occurs when C = SAT: We already have general automatic problem solving via SAT solvers. DE lets us easily define vocabularies, structures, and queries. For example, let agraph be the vocabulary of graphs with two colours:

$$\textit{agraph is new vocabulary } \{A{:}\ 1,\ E{:}\ 2\}$$

DE makes it easy to use SAT solvers to solve a wide class of problems. For example, consider the following 3-ary reduction from 3COLOR to SAT:

$$\textit{thrcoltosat is new query } \{graph,\ sat,\ 3,\ x1 \leq 3,\ P{:}2 \textit{ is } x1 = 3 \wedge x2 = 0 \wedge x3 = x6 \wedge x4 = 0 \wedge x5 < 3,\ N{:}2 \textit{ is } x4 = 0 \wedge E(x2,\ x3) \wedge (x1 = x5 \wedge (x6 = x2 \mid x6 = x3)) \wedge x1 < 3\}.$$

Here, Boolean variable $\langle 0, c, x\rangle$ means "vertex x is colour c", clause $\langle 3, 0, x\rangle$ says that "vertex x is some colour" and if $E(x, y)$ then clause $\langle a, x, y\rangle(for 0 \leq a \leq 2)$ says that, "then x and y are not both colour a". We can then use the SAT solver to check 3-colorability [CIJ12].

It also allows formulas from SO∃(TC), i.e., in addition to first-order logic, second-order existential logic, a transitive closure operator, and arithmetic are all available. Thus, queries using TC and even second-order quantification may be written and evaluated. However, one important reason for creating DE remains[CIJ12]. A "draw" command that users can make changes to defined structures and then immediately observe their effect. This process mimics the visualization skills of the expert and will allow others to gain a deeper understanding of how structures are defined by logical formulas.

5 Finding gadgets automatically for a reduction from 3-sat to 3-color

In this section, we will try to look for gadgets that satisfy our constraints. As we have seen in section 2, finding a gadget for a problem can be a lengthy and tedious process. To make our work easy, we shall use Bule, a grounder that supports SAT and QBF solvers. For our problem, we look for the clause gadget with the minimum number of vertices. We will follow the following two steps to find the gadget.

Step 1: Finding a candidate gadget. We know that the gadget will have 3 vertices as input and a final vertex as output. To find the minimum number of internal vertices, we will have to test all graphs,$\forall g$, $g \subseteq G$, where g is a potential graph from the universal set G containing n vertices and satisfies the QBF. This can be written as

$$\exists E\, \exists n,\, g(E,n) \subseteq G.$$

Step 2: Validating a gadget. Once we have found a graph, we need to check if it is valid, meaning it needs to be 3-colourable. In our scenario, we create 8 copies(3 input nodes, each taking T/F) of the graph and check if it acts as an OR-gadget. This ensures that the output node(Fig 2. $a \vee b \vee c$ node) of the graph only shows true when one or more of the input nodes(Fig 2. a, b, c nodes) it true as well.

True outcome: Apart from satisfying 3-colourability, the graph should also act an an OR-gadget. However, we only check this for the first seven copies where at least one of the input nodes is true.

$$\exists c\, \forall V\; \phi(g(V), c),$$

where c is a potential colouring for graph g with vertices V if $\phi(V, c)$ is satisfied.

False outcome: For the eighth copy of the graph where all the input nodes are false, we ensure that there's no valid colouring that results in a true output. The following clause gives the required equation,

$$\exists c\, \forall V\, \exists c_c\, \exists c_e\; \psi(g(V), c_c) \vee \sigma(g(V), c_e),$$

where c_c give a cheat colour i.e., no colour for a vertex, c_e gives a cheat edge i.e., an edge having same colour at both end vertices. Therefore, a gadget that only satisfies a colouring when a cheat colour or cheat edge is present can never give a valid result when all input nodes are false.

The Bule code for the grounding set up is given in algo.bul and the true and false outcome is given in graph_coloring_postive_copy.bul and graph_coloring_negative_copy.bul respectively.

6 Finding reductions automatically

The above section casts a glimpse on the power of grounding and SAT solvers. In this section, we will try to take a step further and generalise the existence of a reduction between two given problems.

Lemma 6.1. *A reduction r exists between two problems when it correctly maps all true and false instances from one problem to another.*

Proof. Let us consider two problems P and Q and try to prove the above lemma.

Problem P: For any instance I and candidate solution S. S is a valid solution for I if $\phi(I, S)$ is satisfied.

Problem Q: For any instance J and candidate solution T. T is a valid solution for J if $\psi(J, T)$ is satisfied.

From the above, we can write, $\exists r \forall I, I \models \phi_P \Leftrightarrow r(I) \models \phi_Q$.

Simplifying it further,

$$\exists r \forall I, (\exists S^1, \phi_P(I, S^1)) \Leftrightarrow (\exists T^1, \phi_Q(r(I), T^1))$$
$$\exists r \forall I, (\exists S^1, \phi_P(I, S^1)) \Rightarrow (\exists T^2, \phi_Q(r(I), T_2)) \wedge (\exists S^2, \phi_P(I, S^2)) \Leftarrow (\exists T^1, \phi_Q(r(I), T^2))$$
$$\exists r \forall I, \neg(\exists S^1, \phi_P(I, S^1)) \vee (\exists T^2, \phi_Q(r(I), T^2)) \wedge (\exists S^2, \phi_P(I, S^2)) \vee \neg(\exists T^1, \phi_Q(r(I), T^1))$$
$$\exists r \forall I, (\forall S^1, \neg\phi_P(I, S^1)) \vee (\exists T^2, \phi_Q(r(I), T^2)) \wedge (\exists S^2, \phi_P(I, S^2)) \vee (\forall T^1, \neg\phi_Q(r(I), T^1))$$
$$\exists r \forall I, (\forall S^1, \neg\phi_P(I, S^1)) \vee (\exists T^1, \phi_Q(r(I), T^1)) \wedge (\exists S^2, \phi_P(I, S^2)) \vee (\forall T^2, \neg\phi_Q(r(I), T^2))$$
$$\exists r \forall I, \forall S^1 T^1 \exists S^2 T^1 \neg\phi_P(I, S^1) \vee \phi_Q(r(I), T^1) \wedge \phi_P(I, S^2) \vee \neg\phi_Q(r(I), T^2)$$
$$\exists r \forall I, \forall S^1 T^2 \exists T^1 S^2 J, r(I, J) \wedge \neg\phi_P(I, S^1) \vee \phi_Q(J, T^1) \wedge \phi_P(I, S^2) \vee \neg\phi_Q(J, T^2)$$
$$\exists r \forall I, \forall S^1 T^2 \exists T^1 S^2 J, r(I, J) \wedge (\neg\phi(I, S^1) \vee \psi(J, T^1)) \wedge (\phi(I, S^2) \vee \neg\psi(J, T^2))$$

Hence, we get the final result as

$$\exists r \forall I \exists J y, \exists S^1 T^1 \forall S^2 T^2, \tau(r, I, J) \wedge (y \Rightarrow \phi(I, S^1)) \wedge (y \Rightarrow \psi(J, T^1)) \wedge (\neg y \Rightarrow \neg\phi(I, S^2)) \wedge (\neg y \Rightarrow \neg\psi(J, T^2)).$$

Therefore, a reduction r exists between all instances I and J if and only if it maps a yes instance to a yes and no to a no. □

7 Conclusion

We have seen some of the work to find reductions and provided ways to automate it. In section 7, we put forward a QBF that can be solved to find these reductions automatically. Unfortunately, current solvers can't handle instances/structures of reasonable size. Hence, why the problem is still undecidable. However, experience shows that it usually suffices to consider structures of 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. Another way [[CIM10]], involves finding a middle problem M in the middle, so that reduction from P to M and M to Q are simpler. Furthermore, we expect that the merits of each strategy may be combined with future advancements to obtain superior performance.

References

[CIJ12] Marco Carmosino, Neil Immerman, and Charles Jordan. Experimental descriptive complexity. In *Logic and Program Semantics*, pages 24–34. Springer, 2012.

[CIM10] Michael Crouch, Neil Immerman, and J Eliot B Moss. Finding reductions automatically. In *Fields of Logic and Computation*, pages 181–200. Springer, 2010.

[Dea21] Walter Dean. Computational Complexity Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Fall 2021 edition, 2021.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[Imm98] Neil Immerman. *Descriptive complexity*. Springer Science & Business Media, 1998.

[Jon73] ND Jones. Reducibility among combinatorial problems in log n space. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems*, pages 547–551. Dept. of Electrical Engineering, Princeton University Reading, Mass., 1973.

[Mou14] Lalla Mouatadid. Introduction to complexity theory: 3-colouring is np ..., 2014.