

Automatic Evaluation of Reductions between NP-complete Problems

Carles Creus, Pau Fernández, and Guillem Godoy *

Universitat Politècnica de Catalunya, Department of Software,
ccreuslopez@gmail.com, ggodoy@lsi.upc.edu, pau.fernandez@upc.edu

Abstract. We implement an online judge for evaluating correctness of reductions between NP-complete problems. The site has a list of exercises asking for a reduction between two given problems. Internally, the reduction is evaluated by means of a set of tests. Each test consists of an input of the first problem and gives rise to an input of the second problem through the reduction. The correctness of the reduction, that is, the preservation of the answer between both problems, is checked by applying additional reductions to SAT and using a state-of-the-art SAT solver. In order to represent the reductions, we have defined a new programming language called REDNP. On one side, REDNP has specific features for describing typical concepts that frequently appear in reductions, like graphs and clauses. On the other side, we impose severe limitations to REDNP in order to avoid malicious submissions, like the ones with an embedded SAT solver.

Keywords: SAT application, Reductions, NP-completeness, Self-learning

1 Introduction

Nowadays, there is an increasing interest in offering college-level courses online. Several websites like Khan Academy [1], Coursera [2], Udacity [3] and edX [4], provide online courses on numerous topics. The users/students have access to videos and texts explaining several subjects, as well as tools for automated evaluation by means of exercises. In the specific context of computer science, online judges for testing correctness of programs are used in several academic domains as a self-learning tool for students, as well as an objective method in exams for scoring their programming skills (see, e.g., [5]). This kind of judges are also used in online contests organised by several sites, like TopCoder [6] and Codeforces [7].

For the last two years we have developed a specific online judge for the subject of Theory of Computation [8], located at <http://racso.lsi.upc.edu/juez>. The site offers exercises about deterministic finite automata, context-free grammars, push-down automata, and reductions between undecidable problems. Users can submit their solutions, the judge evaluates them, and offers a counterexample when the submission is rejected. This is very useful to make students

* The authors were supported by an FPU grant (first author) and the FORMALISM project (TIN2007-66523) from the Spanish Ministry of Education and Science.

understand why their solutions are wrong, and to keep them motivated during the learning process. We have used the judge in the classroom, not only as a support tool for the students, but also as an evaluation method on exams. This has had a marked effect on the motivation and involvement of the students: during a fifteen-week course, each student has solved more than 150 exercises in average, with more than 680 submissions. This means that each exercise needed over 4 submissions to be accepted by the judge, and the students were motivated enough to perform new attempts to reach an acceptance verdict. These results have motivated us to extend the judge to evaluate other kinds of exercises related to the subject. In particular, we have developed an evaluator of reductions between NP-complete problems. In this paper we explain the main aspects of this new evaluator. In Section 2, we explain our approach for checking the reductions, based on the use of SAT solvers, and the need for a specialised programming language to describe the reductions. This language is described in Section 3. In Section 4 we analyse the performance of the judge.

2 Approach

We start by recalling the well-known concept of reduction [9]. Given two decisional problems P_1, P_2 , a reduction $P_1 \leq P_2$ is a program R that transforms a valid input of P_1 into a valid input of P_2 preserving the answer between the respective problems, i.e., that satisfies $P_1(I) \Leftrightarrow P_2(R(I))$ for any valid input I of P_1 . In the specific setting of reductions for proving NP-hardness, R must be a polynomial time algorithm. Also, recall that any NP-complete problem can be reduced to SAT in polynomial time.

Currently, our website has a list of exercises of reductions between NP-complete problems. Each exercise describes two problems P_1, P_2 , and the user must provide a program R , which is intended to be a polynomial time reduction $P_1 \leq P_2$. Internally, the judge has a set of tests, where each test is a valid input I_1 of P_1 , and uses two additional reductions $R_1 : P_1 \leq \text{SAT}$, $R_2 : P_2 \leq \text{SAT}$ to transform the respective $I_1, I_2 = R(I_1)$ into instances of SAT $S_1 = R_1(I_1)$, $S_2 = R_2(I_2)$. Next, it checks whether S_1, S_2 are logically equivalent using a state-of-the-art SAT solver. In the negative case, the judge offers I_1 and the corresponding I_2 to the user as a counterexample to the correctness of R .

The previous approach has two main drawbacks. On the one side, passing a finite set of tests does not guarantee the preservation of the answer. Nevertheless, the same objection arises for programming judges, and experience shows that a good set of tests often suffices to avoid false positives (except for those malicious submissions that fail deliberately on a particular case). Typically, creating a good set of tests requires to anticipate wrong solutions to the exercise that can be apparently correct, and produce one or more tests that make those wrong reductions fail. Also, it is convenient to cover a large number of cases by generating several input instances systematically, perhaps randomly. On the other side, since the involved problems in the reduction are hard, we need to use small tests I_1 . In fact, these tests must be very small since, otherwise, the

composition of reductions might produce instances S_2 that are too complex for the SAT solver. However, an exercise $P_1 \leq P_2$ checked with small tests could be passed by a malicious submission that solves $P_1(I_1)$ directly (by, e.g., using an adequate backtracking that works fine for small inputs, or with a direct translation to SAT combined with the use of a simple embedded SAT solver), and produces a forged I_2 accordingly. Thus, an exponential time transformation might be wrongly accepted.

We have designed a concrete programming language specially suited for describing reductions between NP-complete problems called REDNP. The transformations can be described succinctly due to the management of the input and the output based on a predefined structure for each problem and the use of strings for naming basic objects. REDNP also has features for describing typical concepts that frequently appear in reductions, like graphs and clauses. In addition, we impose severe limitations to REDNP, like the use of structured intermediate memory, in order to avoid malicious submissions (this should not be an obstacle for describing reductions, since many natural reductions between NP-complete problems are just projections [10]). Also, we limit the maximum execution time and the value of integers (this makes it difficult to use integers as bitvectors representing sets of true/false values for performing some kind of exhaustive search).

3 REDNP

REDNP is a simple C-like programming language with the classical **if-else**, **for**, **while** instructions, and expressions with the usual integer and boolean operators. In our case, the assignment **=**, and the pre- and post-increment and decrement are also instructions. We also admit a **foreach** instruction. Except for two special variables **in** and **out**, all the variables have simple types **int** and **string**. This is to avoid the use of intermediate structured data for the computation of the reduction. For the same reason, strings can be concatenated and assigned, but their positions cannot be accessed with an index. Thus, strings should be used only to name objects in the result of the reduction. The string operator for concatenation is **+**, and integer values are automatically converted to strings when they are operated or assigned. In string literals like "**s**{**i**}{**0**} **iff** **s**{**i-1**}{**0**} **and not** **n**{**i**}", the expressions inside curly braces are evaluated, making it easier to describe indexed names. The two special variables **in** and **out** mentioned above refer to the input and output of the program, respectively. The contents of **in** can be accessed but cannot be modified, and the contents of **out** can be assigned but cannot be accessed.

For each exercise, the types of **in** and **out** are fixed. As an example, consider the exercise asking for a reduction **VERTEX-COVER** $\leq$ **DOMINATING-SET**. These problems receive a natural number k and an undirected graph G as input. In the first case, the question is whether there exists a choice of k vertices of G such that each edge of G contains at least one chosen vertex. In the second case, the question is whether there exists a choice of k vertices such that any other vertex is adjacent to a chosen one. A typical correct reduction consists in preserving

the same k , removing all isolated vertices, and for each edge (u, v) , adding a new vertex new_{uv} and two new edges (u, new_{uv}) and (v, new_{uv}) . Let's see how we represent this reduction in REDNP. For this exercise, we have the types:

```
in: struct {
    k: int
    nvertices: int
    edges: array of array [2] of int
}
out: struct {
    k: int
    edges: array of array [2] of string
    vertices: array of string
}
```

It is implicit that the vertices are numbered from 1 to `in.nvertices`, and hence, each cell of `in.edges` contains an array with 2 different values inside such interval. Note that the edges described by `in` are pairs of `int`'s, while the edges described by `out` are pairs of `string`'s. This is also the case for all the exercises of our site: names of objects are always `int`'s inside `in`, and `string`'s inside `out`. This is because, usually, reductions need to traverse all input objects in a simple way, and naming them from 1 to a certain value eases this task. Moreover, reductions usually have to name the output objects meaningfully, using names that keep some relation with input objects. Indexed names in the output where the indices refer to input objects is a typical case, like the `newuv` vertex. The field `out.vertices` is optional and does not need to be filled by the reduction, but it is provided for the cases where the user considers necessary to notify the existence of isolated vertices, since they are not present in any edge. A possible solution to this exercise described in REDNP is:

```
out.k = in.k;
foreach (edge; in.edges) {
    u = edge[0]; v = edge[1];
    out.edges.push = u, v;
    out.edges.push = u, "new{u}{v}";
    out.edges.push = v, "new{u}{v}";
}
```

Note that arrays of non-fixed size in the output can be extended using `.push` to add a new element. Also, note that values of several positions of arrays can be simultaneously assigned by separating the list of expressions by commas.

As we have mentioned in Section 2, the problem setter must provide additional reductions to SAT in order to validate the submissions. In particular, a reduction from `DOMINATING-SET` to SAT must be provided for the previous exercise, and this is also done using REDNP. Since this reduction works considering the input elements represented by integers, an internal process transforms the generated graph by converting strings to integers again. The following program could be the reduction provided by the problem setter, as well as a solution for the exercise `DOMINATING-SET ≤ SAT`:

```
for (i=1; i<=in.numnodes; ++i) {
    s = "n{i}";
    foreach (node; in.adjacents[i]) s = s + " or n{node}";
    insertsat(s);
}
insertsat("s{0}{0}");
```

```

for (j=1; j<=in.k; ++j) insertsat("not s{0}{j}");
for (i=1; i<=in.numnodes; ++i) {
    insertsat("s{i}{0} iff s{i-1}{0} and not n{i}");
    for (j=1; j<=in.k; ++j)
        insertsat("s{i}{j} iff (s{i-1}{j} and not n{i}) or " +
                    "(s{i-1}{j-1} and n{i})");
}
insertsat("s{in.numnodes}{in.k}");

```

The previous program appends clauses to `out` using `insertsat`, an additional feature provided by REDNP that allows to transform any given propositional formula into a list of clauses using the Tseitin transformation.

4 Performance

As mentioned in Section 2, it may be hard to solve the resulting SAT instance $S_2 = R_2(R(I_1))$ compared to the cost of solving I_1 and $S_1 = R_1(I_1)$. In this section we explain some experiments¹ that confirm this statement, and justify the need for considering only simple I_1 's as tests for the judge. MINISAT [11] has been the final choice used by the judge since PICOSAT [12] gave lower performance.

As a first experiment, we consider the family of tests $\langle k, \mathcal{K}_n \rangle$ as input for the exercise **VERTEX-COVER** $\leq$ **DOMINATING-SET**, where $\mathcal{K}_n$ represents the complete graph of size n . Figure 1 shows the time needed by MINISAT to solve S_1 and S_2 for n 's chosen among $\{12, 14, 16, 18, 20\}$ and all possible k 's. The critical k with respect to performance is the maximum number of vertices that cannot cover $\mathcal{K}_n$, that is $k = n - 2$. All the S_1 's are solved very fast, whereas for $n = 20$ and $k = 18$ the solver needs about 50 seconds to solve S_2 .

We have conducted additional experiments with similar results. For instance, when using matrix-like graphs with size $n \times n$ as input for the same exercise, S_1 is solved quickly and S_2 takes significantly longer for the critical k . In the exercise **CLIQUE** $\leq$ **INDUCED-SUBGRAPH**, we have modified the test set $\langle k, \mathcal{K}_n \rangle$ removing n edges from each graph to guarantee that there is no clique of size $n/2 + 1$. We observed that, in this case, solving S_2 was either easy or very difficult, with an abrupt transition at $k \approx n/2$. Another interesting example is the reduction of **HAMILTONIAN-CIRCUIT** from directed to undirected graphs. In this case, with a relatively small matrix-like graph of 4×4 , the composition of reductions produces an S_2 which is very difficult to solve.

These experiments show that very small instances give rise to SAT instances that are difficult for state-of-the-art SAT solvers after applying the composition of reductions to them. This seems to happen when the initial small instance belongs to a parameterized family of conceptually difficult instances. This means that the SAT solver is not able to detect the backdoor variables in such cases. SAT instances generated in this way could be included in the benchmarks for the SAT competitions, since they have a practical interest.

¹ The data is available at <http://www.lsi.upc.edu/~ggodoy/publications.html>

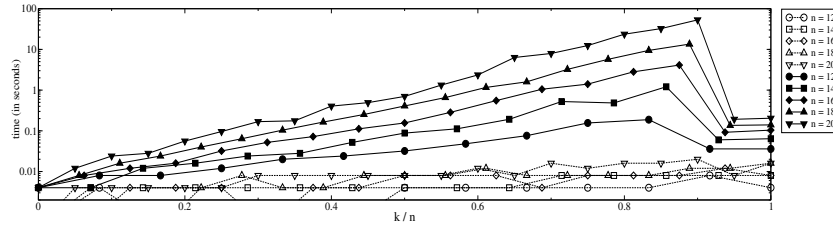


Fig. 1. Performance of MINISAT for $\text{VERTEX-COVER} \leq \text{DOMINATING-SET}$ (continuous lines; dotted lines correspond to solving VERTEX-COVER) with input $(k, \mathcal{K}_n)$.

5 Conclusions

We have developed an online judge for automatic evaluation of exercises on reductions between NP-complete problems. To this end, we have designed the programming language REDNP, which facilitates the task of describing such reductions, imposing limits to the algorithmic descriptions in order to avoid malicious submissions. This last feature was mandatory since, due to the hardness of the testing process, the tests used by the judge must be very simple.

Several questions arise from this work. On the one side, it could be interesting to design an alternative non-imperative language for the description of reductions, since some students might find more natural to express reductions in a declarative way. Nevertheless, one should decide how to restrict the language in order to avoid non-polynomial time reductions. On the other side, variants of our approach using different kinds of solvers (e.g., SMT solvers) should be experimented, since they might offer better performance and ease of use for reductions that need to express arithmetic properties.

References

1. Khan, S.: Khan Academy. www.khanacademy.org (2006)
2. Ng, A., Koller, D.: Coursera. www.coursera.org (2012)
3. Thrun, S., Stavens, D., Sokolsky, M.: Udacity. www.udacity.com (2012)
4. Agarwal, A.: edX. www.edx.org (2012)
5. García, C., Revilla, M.A.: UVA online judge. uva.onlinejudge.org (1997)
6. Hughes, J.: TopCoder. www.topcoder.com (2000)
7. Mirzayanov, M.: Codeforces. www.codeforces.com (2010)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley (2006)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
10. Skyum, S., Valiant, L.G.: A complexity theory based on boolean algebra. *Journal of the ACM* **32**(2) (1985) 484–502
11. Sörensson, N., Eén, N.: MiniSat 2.1 and MiniSat++ 1.0. Technical report, SAT Race 2008 Editions (2008)
12. Biere, A.: PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation* **4**(2-4) (2008) 75–97