

Teaching Problem Reduction: NP-Completeness via Sudoku

Andrea F. Lobo

Rowan University, Department of Computer Science
Glassboro, NJ, U.S.A., 08028
lobo@rowan.edu

Abstract

It is widely recognized that every Computer Science (CS) professional should understand and be able to apply some aspects of intractability and computational complexity. NP-completeness [8] is one of these fundamental concepts. The concept of problem reduction is central to the notion of NP-completeness and to NP-completeness proofs, but it seldom appears in the CS curriculum in any other context. We posit that encoding one problem as another is a problem solving technique, and that early experience with this technique would help undergraduate CS students understand NP-completeness. In this paper, we propose the use of the popular Japanese number puzzle Sudoku [19] to introduce and explain the concept of problem encoding. We present a straightforward translation of a Sudoku instance to an instance of the well-known NP-complete Satisfiability problem (SAT) [5]. We propose that CS students develop a Sudoku solver in a lower-division course such as Data Structures or Discrete Mathematics. They would accomplish this by translating a Sudoku puzzle into a SAT instance and using a public-domain SAT solver. We believe this will help students understand the concept of NP-completeness and proofs of NP-completeness, when they later encounter them in the CS curriculum.

Keywords: NP-Completeness, SAT, Computer Science Education, Propositional Logic.

1 Introduction

Intractability and computational complexity are fundamental Computer Science (CS) concepts. They are theoretically important and certainly of interest to academics. Additionally, there are aspects of intractability and computational complexity that every CS professional should understand and be able to apply: How to evaluate the efficiency of an algorithm for a problem; the fact that some problems are hard or intractable and cannot be solved efficiently in polynomial time, no matter how fast the processor or large the memory; the fact that there are some other problems, which are called NP-complete, whose exact complexity is not even known. It is now widely recognized and accepted that these topics have an important place in the CS undergraduate curriculum. The ACM-IEEE CC2001 core [7] includes 6 hours of basic computability in topic AL5. Topic AL6 of CC2001 is The Complexity Classes P and NP . The ABET accreditation standards for CS programs [1] also require theory.

The NP-complete problems [8] have practical as well as theoretical importance, and they arise in varied domains of science. Many NP-complete problems arise from natural applications, such as “Given a map with cities and roads with travel times, can a salesperson visit each city only once and return to the starting point given a number of days for the trip?” Another example of a NP-complete problem is “Given geographical locations for several cellular towers, their transmission powers, and a set of available transmission frequencies, can each tower be assigned a frequency so that overlapping coverage areas use different frequencies?” A third NP-complete problem, the well-known Satisfiability (SAT) [5], asks whether the boolean variables of a given propositional formula in Conjunctive Normal Form (CNF) can be assigned true/false values in such a way that makes the formula true. The SAT problem has been receiving renewed attention from researchers in recent years, driven in part by the lucrative application of efficient SAT solvers on today’s fast computers to real-life circuit verification problems. Several efficient SAT solvers such as Zchaff [14], Berkmin [9] and Walksat [18] are now publicly downloadable. Additionally, the annual SAT competition [15] features a contest among SAT solvers and sets a standard file format for SAT problem instances.

It is becoming increasingly important for CS undergraduates to understand NP-completeness, even for those students who are not theoretically inclined and are not planning to attend graduate school. The common curricular approach to computability in undergraduate CS programs is to address intractability and NP-completeness in an “Algorithm Design and Analysis” (AD&A) course. We have observed, anecdotally and from our experience with the accreditation of CS programs, that many AD&A courses and textbooks [2,4,6,10,11,13,16,17,20] address this topic towards the end. In many AD&A courses where this topic is discussed, the students are asked to write an NP-completeness proof. This is primarily an application of non-trivial mathematical skills, and many students do not develop an understanding of NP-completeness, of the fundamentally difficult nature of NP-complete problems and of the importance of the P vs. NP question. These shortcomings of the common curricular approach to computability have not been adequately addressed in the Computer Science Education literature. Our suggested remedy [12] for this pedagogical problem received the Best Paper Award at CCSCNE 2006.

In this paper, we posit that NP-completeness is conceptually hard to grasp at the undergraduate level largely due to the fact that most undergraduate students have never been exposed to the notion of problem reduction. Intuitively, a problem P_1 can be reduced to problem P_2 if there is a quick (polynomial runtime) translator that translates problem instances of P_1 into problem instances of P_2 in a manner that is “solution preserving”. This means that the problem instance of P_1 is solvable if and only if the translated instance (of P_2) is solvable. Most undergraduate students encounter problem reduction for the first time when they are presented with their first NP-completeness proof. Note that reduction can be viewed as a problem solving technique: If you have an efficient method for solving P_2 , then P_1 can also be solved efficiently if you can find an efficient way of reducing P_1 to P_2 . We propose that problem reduction be introduced to students early on as yet another abstract problem solving technique, in the same vein as divide and conquer or dynamic programming. Section 2 describes a programming project that exposes undergraduate CS students to problem reduction in a lower division course. The project involves translating the Japanese number puzzle Sudoku to SAT. We choose Sudoku because of its almost universal popularity. We choose SAT because we have found it to be very easy to describe and motivate to undergraduate students [12], there is a straightforward reduction from Sudoku to SAT, and there are efficient public domain SAT solvers. A translation of Sudoku to SAT is presented in Section 3. Section 4 concludes and outlines future work.

2 The Programming Project

The classic Sudoku puzzle [19] consists of a 9x9 grid containing preset values in some of its 81 cells. The grid is subdivided into 9 3x3 boxes. The objective of the puzzle is to fill the blank cells so that each column, each row, and each box contains all the digits between 1 and 9. A Sudoku puzzle can be easy for humans, or difficult and time-consuming. Note that Sudoku stated in a general form for squares of size $n^2 \times n^2$, for arbitrary values of n , is NP-complete [21].

We propose the assignment of a programming project that solves Sudoku puzzles in a lower division course such as Data Structures or Discrete Mathematics. To solve this assignment, the students will need a basic understanding of propositional logic: Boolean variables, *and* and *or* operators, converting propositional formulae into Conjunctive Normal Form (an *and* of clauses that contain *or*'s of Boolean variables or their negation), DeMorgan's Law, etc. They will also require rudimentary programming skills to prepare an input file and then execute a program that was correctly implemented by another person. The project can be completed by CS freshmen or sophomores in three steps: Given a Sudoku puzzle,

1. Translate it into a SAT instance (one translation is described in section 3 below), and store the SAT instance in a file that conforms to the International SAT Competitions format [15],
2. Solve the SAT instance using an efficient public-domain SAT solver, and,
3. Translate the SAT instance solution into the Sudoku puzzle solution.

This is an exercise in problem solving and logic more than one on programming, since steps 2 and 3 are quite simple. This will become clear after examining a Sudoku to SAT translation.

3 A Sudoku to SAT Translation

There are several ways to translate or encode a given Sudoku problem into a SAT problem. In this section, we present one straightforward translation.

We create 729 ($9 \times 9 \times 9$) different propositional variables $v_{i,j,k}$, where i , j and k take on values from 1 to 9. Intuitively, the propositional variable $v_{i,j,k}$ is set to true if and only if the cell at row i and column j takes the value k . Recall the constraints on a Sudoku puzzle solution:

1. Each row must have all the digits 1 through 9,
2. Each column must have all the digits 1 through 9,
3. Each 3x3 box must have all the digits 1 through 9,
4. Each cell must have exactly one value in the range 1 through 9.

These three constraints are the same except that each refers to a different subset of cells.

We now present the SAT encoding for Constraint 1. Constraint 1, when applied to a row i , can be viewed as a conjunct (and) of the following nine constraints:

- 1.1. The number 1 is the value of exactly one cell in row i .
- 1.2. The number 2 is the value of exactly one cell in row i .
- ...
- 1.9. The number 9 is the value of exactly one cell in row i .

We will only address Constraint 1.1 since the other eight constraints can be translated in analogous manner.

Constraint 1.1, in turn, can be broken down as a conjunct of two constraints:

1.1.A. At least one cell of row i has the value 1

1.1.B. At most one cell of row i has the value 1

Constraint 1.1.A, for row i , yields the propositional clause:

$$(v_{i,1,1} \text{ OR } v_{i,2,1} \text{ OR } v_{i,3,1} \text{ OR } v_{i,4,1} \text{ OR } v_{i,5,1} \text{ OR } v_{i,6,1} \text{ OR } v_{i,7,1} \text{ OR } v_{i,8,1} \text{ OR } v_{i,9,1})$$

Constraint 1.1.B, for row i , yields a subformula comprising 36 clauses of the form

$$((\text{not } v_{i,j,1}) \text{ OR } (\text{not } v_{i,k,1}))$$

combined together with the *and* operator, for all distinct combinations of j and k , j and k in the range 1 through 9.

Constraints 2 and 3 can be encoded in a similar manner.

Constraint 4, for the cell at row i and column j , is encoded as a conjunction of two constraints:

4.1. The cell at row i and column j has at least one value in the range 1 through 9

4.2. The cell at row i and column j has at most one value in the range 1 through 9

Constraint 4.1, for the cell at row i and column j , yields the propositional clause:

$$(v_{i,j,1} \text{ OR } v_{i,j,2} \text{ OR } v_{i,j,3} \text{ OR } v_{i,j,4} \text{ OR } v_{i,j,5} \text{ OR } v_{i,j,6} \text{ OR } v_{i,j,7} \text{ OR } v_{i,j,8} \text{ OR } v_{i,j,9})$$

Constraint 4.2, for the cell at row i and column j , yields a subformula comprising 36 clauses of the form

$$((\text{not } v_{i,j,v1}) \text{ OR } (\text{not } v_{i,j,v2}))$$

combined together with the *and* operator, for all distinct combinations of $v1$ and $v2$, $v1$ and $v2$ in the range 1 through 9.

Any preset values that appear in a Sudoku puzzle lead to additional clauses in its translation. For instance, if the cell in row 3 and column 7 has the value 6, we add the clause $(v_{3,7,6})$ to the formula. These are the clauses that will differ from one puzzle to another.

A Sudoku puzzle translates in this manner into a SAT instance with 729 variables and approximately 12000 clauses: 37 $(36 + 1)$ clauses from the translation of Constraint 1.1; 333 $(37 * 9)$ clauses from Constraint 1; 8991 $(333 * (9+9+9))$ clauses from all 9 rows, 9 columns, and 9 boxes; 2997 $((9*9)*(36+1))$ clauses from Constraint 4; and a few more for the preset values. Encodings resulting in smaller formulae are possible but we think the translation presented here is more intuitive.

We wrote a simple program that translates a Sudoku puzzle into its corresponding SAT instance as described above, and writes the SAT instance into a file in the International Sat Competitions format [15]. This file format is described in the appendix. We used this program to translate several Sudoku puzzles, ranging in difficulty from easy to extremely hard, into their corresponding SAT instances.

For Step 2 of the assignment, we used the Zchaff SAT solver [14] to solve the SAT instances. In all cases, the downloaded solver executing on a 480 MHz Sun UltraSPARC II found a solution within milliseconds. Step 3, translating the SAT solution into the solution to the Sudoku puzzle is trivial: If $v_{i,j,k}$ is set to true in the satisfying assignment, the cell located at row i and column j gets the value k .

4 Conclusions and Future Work

We suggested teaching problem encoding/translation early in the CS curriculum. We proposed that it be presented and used as a problem solving technique. We presented an assignment that requires students to apply their propositional logic knowledge and introduces SAT, a well-known, relevant, NP-complete problem. We plan a simple “with/without” assessment to evaluate the effectiveness of the proposed approach in the near future.

References

- [1] ABET at <http://www.abet.org/>.
- [2] A. Aho, J. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley. 1974.
- [3] Sara Baase and Adam Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Third Edition, Addison-Wesley. 1999.
- [4] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*, Prentice Hall. 1996.
- [5] Stephen A. Cook. The complexity of theorem proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151-158.
- [6] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. Second Edition, MIT Press. 2001.
- [7] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*. ACM and IEEE. 2001.
- [8] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co. 1979.
- [9] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE)*, 2002, pp. 142-149.
- [10] Michael Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis and Internet Examples*. John Wiley and Sons. 2001.
- [11] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press, New York. 1996.
- [12] Andrea F. Lobo and Ganesh R. Baliga, “NP-Completeness for All Computer Science Undergraduates: A Novel Project-Based Curriculum”, *Journal on Computing Sciences in Colleges*, Vol 21, No. 6, (June 2006), pp. 53-63. Also received Best Paper Award at the *Eleventh Annual CCSC Northeastern Conference*, April 2006.
- [13] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley. 1989.
- [14] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering and Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC01)*, 2001.
- [15] International SAT Competitions. See <http://www.satcompetition.org/>

- [16] Clifford Schaffer. *A Practical Introduction to Data Structures and Algorithm Analysis*. Second Edition, Prentice Hall. 2001.
- [17] Robert Sedgewick. *Algorithms*. Addison-Wesley. 1988.
- [18] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI94)*, Seattle, 1994, pp. 337-343.
- [19] Will Shortz. *Will Shortz presents the little black book of Sudoku: 400 puzzles*. St. Martin's Griffin Publishing. 2006.
- [20] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Second Edition, Addison Wesley. 1999.
- [21] Takayuki Yato. Complexity and completeness of finding another solution and its application to puzzles, Masters thesis, University of Tokyo, 2003.

Appendix

Input files for the International SAT Competitions [15] have the following format:

- Comment lines at the beginning of the file start with the character 'c'. There may be 0 or more comment lines.
- The first data line has the format "p cnf number-of-variables number-of clauses".
- The clauses are then specified, one clause per line. A clause is specified by a list of the numbers of the variables that occur in it, followed by the value 0. Please note that variables are numbered starting at 1, and negated variable number *i* is specified as **-i**. Thus, the clause (not-**x12** or **x41**) is specified by the line
-12 41 0

A sample file follows:

c Specification of the formula

c (x1 or x2 or not-x3) and (not-x2 or x4) and (not-x1 or not-x4)

p cnf 4 3

1 2 -3 0

-2 4 0

-1 -4 0