Minimal Superpermutation SAT Benchmarks

Martin Mariusz Lester
Department of Computer Science
University of Reading
Reading, United Kingdom
m.lester@reading.ac.uk
0000-0002-2323-1771

***Abstract*—This benchmark consists of several SAT-based encodings of instances of the minimal superpermutation problem.**

I. INTRODUCTION

The minimal superpermutation problem [1] asks, for a positive integer n, what is the smallest sequence of the digits 1–n that contains every permutation of $[1, n]$ as a subsequence?

For example, for $n = 4$, the minimal superpermutation has length $l = 33$. If the first permutation in the sequence is fixed to be 1234, then it is uniquely determined to be:

$$123412314231243121342132413214321$$

For $n = 5$, $l = 153$, but the sequence is not unique [2]. For higher values of n, the length of the minimal superpermutation is not known. For example, for $n = 6$, $861 \leq l \leq 872$.

An instance of the superpermutation problem can be elegantly encoded as an instance of the Travelling Salesman Problem (TSP). The best known automated methods for solving instances of the superpermutation problem use dedicated TSP solvers. This method was used to verify minimality for $n = 5$ and find the smallest known sequence for $n = 6$ [3]. As the TSP is NP-complete, instances can be translated into SAT instances, but the resulting SAT instances are often hard for current solvers.

The instances in this benchmark suite instead use direct encodings of the superpermutation problem into SAT. The instances encode the decision problem of whether a superpermutation (or a prefix of a superpermutation) of length l for n distinct digits exists, rather than the optimisation problem of finding the smallest l for a particular n.

II. ENCODINGS

The instances directly encode a sequence of l digits drawn from $[1, n]$ and a combinatorial circuit to recognise whether the sequence is a superpermutation.

The 1st layer of the circuit recognises individual permutations. Each permutation consists of $n!$ digits; a permutation recognising circuit checks whether the values of the digits match those expected for a particular permutation. A permutation could start at any of the l digits, except those at the end of the sequence, so roughly l copies of each permutation circuit are needed.

In the 2nd layer of the circuit, for each permutation, the outputs of each copy of the permutation recognising circuit are ORed together. The 3rd layer of the circuit ANDs together the outputs of the 2nd layer, evaluating to true only if all permutations exist in the sequence.

Different instances in the benchmark use different encodings of the digits and the circuit. There are 3 different encodings of the digits:

1) *Binary* encoding, using $\log n$ bits per digit.
2) *One-hot* encoding, using n bits per digit, with k encoded as bit k set to 1 and all other bits to 0.
3) *Unary* encoding, using n bits per digit, with k encoded using the k least significant bits set to 1 and all remaining bits set to 0.

For each encoding digit encoding, there are 2 variants:

1) A *non-strict* encoding, where a digit's bits are constrained only by the permutation recognising circuit.
2) A *strict* encoding, where extra clauses constrain a digit's bits only to valid encodings of a digit; this sometimes allow a smaller encoding of the permutation recognising circuit.

The 1st layer of the circuit has 2 variants:

1) *Flat:* The permutation recognising circuit is a large AND over equality of all digits.
2) *Tree:* The permutation recognising circuit is built from a tree of permutation prefix recognising circuits. Where two permutations share a common prefix, they share circuitry to recognise that prefix.

In total, this amounts to $3 \cdot 2 \cdot 2 = 12$ different encodings.

III. INSTANCES

The benchmark suite contains instances of 3 slightly different problems:

1) Find the minimal superpermutation for $n = 4$ with $l = 33$. These instances are easy, with MiniSAT 2.2.1 solving them in less than 1 minute.
2) Show that the minimal superpermutation for $n = 4$ with $l = 33$ is unique, once the first permutation is fixed. These instances add a clause to instances from the preceding set that forbids the known superpermutation, making them unsatisfiable. These instances are still relatively easy, with MiniSAT solving the hardest in just over 2 minutes.
3) Find a prefix of a superpermutation for $n = 5$ with either $l = 21$ and $g = 15$ permutations, or $l = 26$ and $g = 19$

permutations. These instances are harder, with MiniSAT solving 7 out of the 12 $l = 21$ instances in under 10 minutes and none of the $l = 26$ instances.

In the final set of instances, the check for g permutations was encoded by converting the SAT instance to a Pseudo-boolean (PB) instance, where it could easily be added as a cardinality constraint. Then the cardinality constraint was converted back to SAT using `pbencoder` from `pblib` [4].

The values of l for the final set of instances were chosen to be hard but plausible within the 5000 second time limit used in the SAT Competition. With the PB formulation, the default configuration of the PB solver *clasp* [5] was able to solve 8 out of 12 of the $l = 26$ instances within this time limit.

All timings are for an Intel i5-7500 CPU running at 3.40GHz.

REFERENCES

[1] D. A. Ashlock and J. Tillotson, "Construction of small superpermutations and minimal injective superstrings," in *Conference on Algebraic Aspects of Combinatorics and Sundance Conference and International Conference on Algol 68 Implementation*, ser. Congressus Numerantium, no. v. 93. Utilitas Mathematica Pub. Incorporated, 1993, pp. 91—98. [Online]. Available: https://books.google.co.uk/books?id=f5PgAAAAMAAJ

[2] N. Johnston, "Non-uniqueness of minimal superpermutations," *Discret. Math.*, vol. 313, no. 14, pp. 1553–1557, 2013. [Online]. Available: https://doi.org/10.1016/j.disc.2013.03.024

[3] R. Houston, "Tackling the minimal superpermutation problem," *CoRR*, vol. abs/1408.5108, 2014. [Online]. Available: http://arxiv.org/abs/1408.5108

[4] T. Philipp and P. Steinke, "Pblib – a library for encoding pseudo-boolean constraints into cnf," in *Theory and Applications of Satisfiability Testing – SAT 2015*, ser. Lecture Notes in Computer Science, M. Heule and S. Weaver, Eds. Springer International Publishing, 2015, vol. 9340, pp. 9–16.

[5] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "*clasp* : A conflict-driven answer set solver," in *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, ser. Lecture Notes in Computer Science, C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483. Springer, 2007, pp. 260–265. [Online]. Available: https://doi.org/10.1007/978-3-540-72200-7_23