

A Catalogue of the Steiner Triple Systems of Order 19

Petteri Kaski¹, Patric R. J. Östergård², Olli Pottonen², and
Lasse Kiviluoto³

¹Helsinki Institute for Information Technology HIIT
University of Helsinki, Department of Computer Science
P.O. Box 68, 00014 University of Helsinki, Finland

²Department of Communications and Networking
Helsinki University of Technology TKK
P.O. Box 3000, 02015 TKK, Finland

³Celtius Ltd
Pieni Robertinkatu 11, 00130 Helsinki, Finland

Abstract

A method for compressing Steiner triple systems is presented. This method has been used to compress the 11,084,874,829 Steiner triple systems of order 19 into approximately 39 gigabytes of memory. The compressed data can be obtained by contacting the authors.

1 Introduction

After the announcement of the completed classification of the Steiner triple systems of order 19—STS(19) for short—which was later published in [2], the authors got several requests for a complete collection of the systems in electronic form. Such requests were politely answered by saying that there are in fact more than 11 billion, or precisely

$$N(19) = 11,084,874,829$$

such objects, so these had not been saved during the search. Indeed, saving the generated systems as 19×57 binary incidence matrices would take about 1.5

terabytes of storage—not an altogether impossible amount of data for modern mass storage devices, but certainly cumbersome to handle.

It is an instructive exercise to think about how much this data can be practically compressed—not only in the case of the STS(19), but also for other collections of combinatorial objects.

Theoretically, assuming a collection of Q objects, on average at least $\log_2 Q$ bits are required to uniquely represent (identify) an object in the collection. Concatenating the compressed representations of all the objects, at least $Q \log_2 Q$ bits, or approximately 46 gigabytes for $Q = N(19)$, are required.

If one relaxes the requirement that each object be represented as a unique string of bits, other possibilities for compressing the objects become apparent. At one extreme, the phrase “list all nonisomorphic STS(19)” arguably compresses our objects with the human decompressor in mind. In a more theoretically disciplined setting (properly captured in the theory of Kolmogorov complexity [4]), one can ask for the shortest computer program that outputs a complete list of isomorphism class representatives. Considering the total size of the computer programs needed to carry out the original classification—about 200 kilobytes—one can obviously say that the objects can be compressed into this size in a modern computer, but extracting the objects is then rather slow.

In the present study we are interested in obtaining a practical trade-off between requirements in storage space and in decompression time. Here we anticipate that a user is interested in either (a) systematically searching through the objects, or (b) sampling the objects uniformly at random, for example, to test whether they have certain properties.

The compression scheme to be presented makes use of the fact that there is a strong degree of similarity between consecutive objects generated by the classification program. The Steiner triple systems of order 19 are compressed into about 39 gigabytes, or little more than 28 bits per system. Decompression of the entire collection of objects takes approximately 15 hours on a Linux PC with a 2-GHz CPU ($5\mu\text{s}$ per isomorphism class), which is about 0.5% of the time required by a classification from scratch with our current implementation.

2 Compression of Steiner Triple Systems

2.1 Preliminaries

A *Steiner triple system* of order v is a pair (P, \mathcal{B}) , where P is a v -element set of *points* and \mathcal{B} is a set of 3-element subsets of P , called *blocks*, such that every 2-subset of P occurs in exactly one block. Each point of P necessarily occurs in exactly $r = (v - 1)/2$ blocks, and the number of blocks is $b = v(v - 1)/6$.

For convenience in what follows, we assume that the point set is $P = \{1, 2, \dots, v\}$ and that pairs and triples over P are lexicographically ordered. More precisely, for

pairs $\{x, y\}$ and $\{u, v\}$ with $x < y$ and $u < v$, we define $\{x, y\} < \{u, v\}$ if and only if either $x < u$ or $x = u, y < v$. Similarly, for triples $\{x, y, z\}$ and $\{u, v, w\}$ with $x < y < z$ and $u < v < w$, we define $\{x, y, z\} < \{u, v, w\}$ if and only if $x < u$ or $x = u, y < v$ or $x = u, y = v, z < w$.

2.2 The Compression Algorithm

Consider the blocks of an STS in increasing lexicographical order, and observe that the two smallest points of each block form the lexicographically smallest pair that does not occur in any of the preceding blocks. Consequently, the list of the largest elements of the blocks then uniquely determines the STS. This observation leads to effective compression, and we compress a list of STSs by applying the following algorithm to each system:

1. Sort the blocks in ascending lexicographical order $B_1 < B_2 < \dots < B_b$.
2. If the system under consideration is the first one, let $i = 0$. Otherwise, compute the largest i such that the first i blocks are equal to the first i blocks of the previous STS and store the result.
3. For $j = i + 1, i + 2, \dots, b$, consider $B_j = \{x, y, z\}$ with $x < y < z$. Compute the set P of all points p for which neither $\{x, p\}$ nor $\{y, p\}$ occurs in any of the blocks B_1, B_2, \dots, B_{j-1} . Obviously P is nonempty since $z \in P$. Let $n = |P|$. With $P = \{p_1, p_2, \dots, p_n\}, p_1 < p_2 < \dots < p_n$, find the index k for which $p_k = z$ and store it.

Decompression is achieved with the straightforward inverse operation.

Before proceeding further we note that the block B_{i+1} is never equal to the corresponding block of the previous STS. This fact can be used to further compress the data slightly; however, here we ignore this observation to enable a more streamlined implementation.

Using the aforementioned scheme, an STS is transformed into a sequence of $1 + b - i$ integers consisting of the value i and the indices k . The related values of n are obtained by the algorithm. Rough bounds for the values of i and k are $0 \leq i \leq b$ and $1 \leq k \leq n \leq v - 2$.

We use Huffman coding (see [1] or any other textbook treating source coding) for storing the integer sequences obtained from the STSs. Since different values of n lead to different distributions of the values of k , we have designed one code for each possible value of n . For $n = 1$ the only codeword is the empty string. A separate code is used to encode the values of i .

2.3 Implementation Details

To enable storage on file systems with restrictions on individual file size, and to make a distribution on media with limited storage capacity possible, the com-

pressed data is partitioned into several files. To simplify random access, we further divide the STSs into sets of equal size and compress each set independently (several such sets may still be stored in the same file). We used sets of 500 STSs, but other sizes are also supported. The last set may obviously have fewer STSs.

The details of the file format for compressed data are as follows. A file consists of three parts. The first part is simply the byte offset at which the third part begins. The second part consists of the sets of compressed STSs. The third part contains, in this order, a magic number `0xf0e86dcdd` included as a validity check, the number of STSs contained in the file (which is used for detecting the last STS), the number of STSs in the sets that are independently compressed (currently 500, as mentioned above), and the byte offsets for each set, starting with the first one. The data in the first and third part is represented as 4-byte unsigned integers with bytes written starting with the most significant byte, that is, in big-endian order. (Due to the 4-byte size for byte offsets, the files cannot be arbitrarily large.) In Huffman coding and decoding, a byte is considered as a string of bits starting with the least significant bit.

2.4 Decompression Software

The compressed data is accompanied by decompression software. The software contains a simple command-line utility and a library which allows other programs to access the compressed data.

The command-line utility, called `stsc`, can be used in the following ways:

```
stsc -u <inputfile> <outputfile>  Decompression
stsc -c <inputfile> <outputfile>  Compression
stsc -s <inputfile> <index>       Decompression of single STS
```

Here the index specifies which of the STSs in the input file is to be decompressed; the indices start at 1. To use standard input or output streams instead of ordinary files, specify '-' as input or output file, respectively.

In input and output, each STS is represented as a list of blocks without delimiters. The points are denoted by characters `a,b,...,s`. For example,

$$\{1,2,4\}, \{2,3,5\}, \{3,4,6\}, \{4,5,7\}, \{1,5,6\}, \{2,6,7\}, \{1,3,7\}, \dots$$

would be represented as `abdbcecdfdegaefbfgacg...` Each STS is followed by a newline. The described algorithm works for any order v , but the software is tailored for order 19. Nontrivial changes—like defining Huffman codes—are needed for other orders.

To use the decompression library, the header file `compression.h` must be included and the library `libstsc.a` must be linked. This header file defines the preprocessor constants `STS_V` and `STS_B` as the number of points and blocks of

an STS, respectively. The following functions are used to decompress data. If an error is detected, an error message is printed and the program is terminated with `abort()`.

- `file_r *init_decompress(FILE *)`
Initializes and returns a new file reader instance. An error is reported in case of a memory allocation failure, an I/O error, or an invalid file.
- `void free_decompress(file_r *)`
Deallocate the file reader.
- `const int *decompress(file_r *)`
Decompresses the next STS and returns a pointer to it. Reports an error if the end of the file has been reached, an I/O error occurs, or the file contains invalid data. The pointer remains valid until the next call to `decompress()` or `free_decompress()`.
- `void open_set(file_r *, int)`
Causes decompression to continue at the beginning of the specified set of STSs. Indexing of the sets starts at 0. An error is reported if the index is negative, the index is too large, or an I/O error occurs.
- `int get_sts_count(const file_r *)`
Returns the number of STSs the file contains.
- `int get_set_size(const file_r *)`
Returns the size of the sets to which the STSs are divided.
- `int get_offset(const file_r *)`
Returns the index of the next STS to be decompressed. Indexing starts at 0.

As an example, we include the following function, which implements the essential functionality of `stsc -s`. That is, the function takes a file and an index, and gets the STS in the file at the given index. The indices start at 0.

```
#include "compression.h"

const void get_sts(FILE *f, int index, int *sts) {
    file_r *r = init_decompress(f);

    if(index < 0 || index >= get_sts_count(r)) {
        fprintf(stderr, "get_sts: index out of bounds\n");
        abort();
    }

    int set_size = get_set_size(r);
```

```

open_set(r, index/set_size);
index %= set_size;

const int *s;
for(i=0 ; i<=index ; ++i)
    s = decompress(r);

memcpy(sts, s, sizeof(int[3*STS_B]));
free_decompress(r);
}

```

The source code of `stsc` can be considered as a more complete version of this example. The decompression library is also accompanied by `sample.c`, which generates a sample of the STS(19) uniformly at random.

3 Conclusions

A practical compression scheme for Steiner triple systems has been presented and used for compressing the complete set of 11,084,874,829 nonisomorphic systems of order 19 into approximately 39 gigabytes.

There is very little, if any, previous work concerning compression of combinatorial objects. One possible idea for further research would be to utilise an existing listing/search algorithm for the objects to enable further compression. In particular, to describe the objects, it suffices to compress the paths in the search tree of the algorithm that lead to the objects—one can expect the descriptions of such paths to compress better than the descriptions of the objects themselves.

Work is in progress [3] to collect results on various properties of Steiner triple systems of order 19. The authors of [3] are glad to invite colleagues to join this project by contributing with an investigation of a property deemed interesting.

The compressed STS(19) data and the decompression software can be obtained from the authors—to this end, please send an email to `petteri.kaski@cs.helsinki.fi`, `patric.ostergard@tkk.fi`, or `olli.pottonen@tkk.fi`, and ask for instructions.

Acknowledgments

The third author was supported in part by the Graduate School in Electronics, Telecommunication and Automation and a grant from the Foundation of Technology (Tekniikan edistämissäätiö). The research was supported in part by the Academy of Finland, Grants 107493, 110196, and 117499. The Department of Computer Science at the University of Helsinki is gratefully acknowledged for

providing computing resources. The authors thank Esa Seuranen for fruitful discussions.

References

- [1] T. M. Cover and J. A. Thomas, Elements of Information Theory, 2nd ed., Wiley, New York, 2006.
- [2] P. Kaski and P. R. J. Östergård, The Steiner triple system of order 19, *Math. Comp.* **73** (2004), 2075–2092.
- [3] P. Kaski, P. R. J. Östergård, and O. Pottonen, Properties of the Steiner triple systems of order 19, in preparation.
- [4] M. Li and P. Vitanyi, An Introduction to Kolmogorov Complexity and Its Applications, 2nd ed., Springer, New York, 1997.