Torch: Software Package For The Search Of Linear Binary Codes

Carolin Hannusch
Department of Computer Science
Faculty of Informatics
University of Debrecen
Kassai út 26
H-4028 Debrecen, Hungary
hannusch.carolin@inf.unideb.hu

Sándor Roland Major
Department of Information Technology
Faculty of Informatics
University of Debrecen
Kassai út 26
H-4028 Debrecen, Hungary
major.sandor@inf.unideb.hu

***Abstract*—We describe a software package created by the authors that can be used to search for linear binary codes with almost arbitrary conditions. The package is easily extensible and reconfigurable to suit the specific needs of the search. The main function can be used to search for currently unknown linear codes, or to quickly generate examples of known codes.**

***Index Terms*—binary codes, search algorithms, self-dual codes**

I. Introduction and Motivation

The search of linear binary codes is as old as Coding Theory is itself [11]. For a given codelength n, the question what (n, k, d)-codes exist, where k denotes the dimension and d the minimum distance, cannot be answered in general. For some reason, within Coding Theory self-orthogonal and self-dual codes enjoy special attention. Binary self-orthogonal codes are called to be of Type II if all codewords have weight divisible by 4, and to be of Type I if it has also codewords of weight not divisible by 4.

It is well-known that Type II codes exist only for codelength divisible by 8. Type I codes exist for all even codelengths. Further, we know [9] that if C is a self-dual code, then its minimum distance $d \leq 4\lfloor \frac{n}{24} \rfloor + 4$, if $n \not\equiv 22 \mod 24$ and $d \leq 4\lfloor \frac{n}{24} \rfloor + 6$, if $n \equiv 22 \mod 24$. If C is a Type I code and $n \equiv 0 \mod 24$, then $d \leq 4\lfloor \frac{n}{24} \rfloor + 2$. If C is an (n, k, d) code, where d reaches its upper bound, then C is called *extremal*. If d does not reach the upper bound, but there does not exist a code with the same length and dimension which would have higher minimum distance, then C is called *optimal*.

For several natural numbers n, there are still open research questions, the most regarding extremal and optimal self-dual codes, e.g. the existence of a self-dual $(72, 36, 16)$ Type II code [5] or the existence of a self-dual $(56, 28, 12)$ Type I code [6]. The search of these codes have aimed that a lot of codes were found ([8], [7]), but the original questions are still not answered. We also use the notation of neighbouring codes, which was introduced in [1]. Nowadays, code-based cryptography is gaining more and more importance, as it is also called post-quantum cryptography [10]. Our intention was to invent a code searching software, which can create different generator matrices for similar binary codes fast, so those matrices can be used in cryptographic schemes.

II. The Software

The software implementation described in this paper is used for research purposes to find linear codes with specific conditions. The goal is to provide a search algorithm that can be extended or configured in a flexible, reusable way, such that new research results can be quickly and easily incorporated. The package is named Torch.

In the following sections, we describe the technical requirements of the software, the high-level structure and interface, and some practical results are presented.

A. Dependencies

The package is written in Python 3.x, and uses the following software:

- Sagemath 9.x (*https://www.sagemath.org/*) is used for most basic linear code operations, and to provide an interface to other software packages
- the GAP Guava package (*https://www.gap-system.org/Packages/guava.html*) is used for some specific operations, such as minimum distance calculation
- Magma (*http://magma.maths.usyd.edu.au/magma/*) is used for operations where the provided function is more efficient than the Sagemath implementation

B. Modules, options, configurations

The top-level function of Torch is named $get\_codes()$. This function can be provided with a large number of parameters and options. A custom dependency injection framework is used to configure the search algorithm using the information given to the search function. The implementation contains a number of modules for each task in the search. The relationships between these modules are described in .json configuration files. Each config file describes one commonly used way of assembling the objects needed to start the search. Multiple configurations can be combined.

The default configuration defines a depth-first search. When searching for an *(n,k,d)* code, the starting node is a generator

matrix for an *(n,1,d')* code, $d' \geq d$. The search tries to append this matrix one row at a time, such that at a depth of κ', the current node will be a generator matrix for an *(n,k',d")* code, $d' \geq d'' \geq d$. When a depth of *k* is reached, the matrix is returned as a solution and the search continues.

To carry out the search, the software needs to generate candidate vectors that could be appended to the current matrix. Each time a new candidate vector is found, a list of conditions are checked, which are specific to the type of linear code being searched. If all conditions are fulfilled, a new node in the search is created where the parent node's matrix has been appended by the candidate vector. The efficiency of the search greatly depends on how fast the candidate vectors can be created, and how many branches of the search tree can be pruned by the list of conditions.

To ensure that the default search is exhaustive, the conditions included are such that at least one linear code should be returned from each permutation equivalence class.

The following modules are important for instantiating the default search algorithm:

- *weights*: Generates the permitted Hamming weight values for the codewords of a given code. This module is used in multiple other modules to determine possible weights.
- *mu_table*: Let *a* and *b* be two codewords, and let $\mu(a, b)$ be the number of coordinates *i* where both a_i and b_i have a value of 1. For any *a* and *b*, $w(a + b) = w(a)+w(b)-2\mu(a, b)$. For any pair of codeword weights, the possible μ values for codewords with those weights can be precomputed before the search begins. This information is computed and stored by the *mu_table* module, and used when generating candidate vectors.
- *starters*: Generates the possible root nodes of the search. yielding *(n,1,d')* matrices. The number of possible starting matrices depend on the permitted Hamming weights as determined by the *weights* and *mu_table* modules.
- *solutions*: Given an *(n,k',d')* generator matrix *G*, the *solutions* module creates the possible candidate vectors that can be appended to *G*. The most commonly used implementation for this module generates these vectors using a recursive algorithm to distribute a number of 1 values among the vector's coordinates. The vectors are generated sorted in ascending or descending order of Hamming weight. The algorithm is recursive to avoid generating the same prefixes for vectors multiple times. The algorithm also checks a number of conditions during the intermediate steps of generating a vector, in order to abandon incorrect prefixes.

  Another implementation of this module, used in searches for self-orthogonal codes, uses the dual code of *G* to find possible vectors.

  The search algorithm can dynamically switch between implementations during the search when it detects that another implementation would be more efficient. By default, the decision to switch is based on the size of the dual code.
- *children*: Given a node containing an *(n,k',d')* generator matrix, the *children* module generates the possible *(n,k'+1,d")*, $d \leq d'' \leq d'$, matrices that can be created by appending the input matrix with one new vector. This module uses the *solutions* module to find the possible candidate vectors. The *children* module is also configured with a list of conditions that must be fulfilled in order to accept an appended matrix as a new node. These conditions depend on the specific properties of the code being searched for. New conditions can be easily created by extending an abstract base class named *AbstractCheck* and adding the new class to the appropriate configuration file. Adding new conditions like this is the main way of incorporating new research results in the search in order to speed up the algorithm.
- *descendants*: Given a node containing an *(n,k',d')* generator matrix *G*, this module generates the possible *(n,k,d)* matrices that can be created by appending *G* with $k - k'$ new vectors. This module uses the *children* module for the intermediate steps of the search. The default search is done by calling the *descendants* module using the root nodes returned by the *starters* module as inputs. The most basic implementation of this module builds a tree from the nodes created by the other modules.

  Another implementation creates a graph from the nodes to avoid generating a matrix that is just a permutation of vectors of a previous matrix.

  Like the *solutions* module, the *descendants* module can also switch between implementations during the search when certain conditions are met.
- *saver*: Saves the returned linear code objects. The object is serialized to a file using the python package *pickle*. A function to deserialize the objects is also provided. Using this module is optional.
- *logger*: Logs the activity of the search algorithm. The default implementation writes the log to console. Using this module is optional.

For testing and experimental purposes, the modules can also be instantiated independently of the search algorithm, using the dependency injection framework. This framework also handles choosing between multiple implementations for a module during build time, using the parameters given to the search function.

The basic parameters given to the search algorithm are the *(n,k,d)* and *type* properties of the linear code. The *type* parameter can be *type=None*, which adds no other conditions to the code, *type=1* when searching for a simply even self-orthogonal code, or *type=2* when searching for a doubly even self-orthogonal code.

The algorithm also accepts a large number of optional parameters. The most commonly used ones are the following:

- *verbose*: Default value is *True*. If set to *False*, this option turns of the logger module.
- *ascending*: Default value is *False*. Determines if the vectors in the generator matrix should be generated in

ascending or descending order of Hamming weight.
- *save*: Determines the path of the save folder where the serialized linear code objects will be stored. If set to *None*, it turns off the saver module.
- *config*: Determines the configuration file to be used in the search. This parameter is a list of filenames when combining multiple configurations together. If multiple files in the list contain information regarding the same module, the file later in the list overwrites the settings from files earlier in the list.
- *standard*: Default value is *True*. Determines if the generator matrices created by the algorithm should be in standard form or not.
- *wsorted*: Default value is *True*. Determines if the rows of the generator matrices created should be sorted by Hamming weight.
- *ordered*: Default value is *True*. Determines if the generator matrices should use a special ordered form to reduce the number of codes created that are permutation equivalent to each other.

Most modules also have optional parameters specific only to them, to further fine-tune their functions. These parameters can also be given to the search algorithm, so the dependency injection framework can add them to the modules during build time.

The package includes a number of other available configurations used for more specific code searches. These searches are often used in experiments to test new research results. These configurations modify the default configuration described previously, by adding new conditions for pruning the search tree, or new implementations to certain modules. These configurations include:

- *wlimits*: Used for searches where the Hamming weight of the rows of the generator matrix being searched for is limited. Two optional parameters are introduced: *min_line_weight* is used to set the lower weight limit of the rows in the generator matrix, and *max_line_weight* is used to set the upper limit. A common application of this configuration is searching for generator matrices with a minimum total weight, or searching for matrices where all rows have equal weight.
- *alternating*: Using the default configuration, matrices are generated such that the Hamming weights of the rows are sorted in descending order. Using the *alternating* configuration, the matrices will alternate between high weight and low weight for each row. This configuration is used for experimental purposes. In some search cases it is observed that using this construction yields linear codes faster than the default settings.
- *no_all_one*: This configuration is used to create generator matrices such that the linear code does not contain the codeword where every coordinate's value is 1. Since self-dual codes always contain the all-1 codeword, this configuration cannot be used to create such codes. The most common use case of this configuration is creating codes that will be used as building blocks for other, larger codes.
- *morningstar*: This configuration is used to create generator matrices such that the linear code always contains the all-1 codeword. It uses modified implementations of the *starters* and *descendants* modules, making sure that at each step, the sum of all rows of the generator matrix is the all-1 codeword. This configuration cannot be used together with the *no_all_one* configuration. The most common use case is searching for self-dual codes, or self-orthogonal codes that can be later appended using other methods.
- *neighbor*: This configuration is based on a research result regarding self-dual codes. The typical use case is searching for Type I self-dual codes. Let *C* be an *(n,k,d)* Type I self-dual linear code, where *n* is divisible by 8. Let C_{max} be the maximal doubly-even subcode of *C*. The dimension of C_{max} is *k*-1. The dual code of this subcode is $C_{max}^{\perp}$, and its dimension is *k*+1. Thus, the number of codewords in $C_{max}^{\perp}$ is 2^{k+1}. It can be shown that the number of singly even codewords in $C_{max}^{\perp}$ is 2^{k-1} and the number of doubly even codewords in $C_{max}^{\perp}$ is $2^{k+1} - 2^{k-1}$. The configuration uses this result as a condition to prune the search tree.
- *perm_equiv*: The conditions in the default configuration are such that a search typically finds multiple linear codes that are permutation equivalent to each other. This configuration checks to make sure that only codes that are not equivalent are yielded. The current implementation compares nodes to previosly found ones, making this configuration very memory-intensive when searching for large codes. This configuration is used when the goal of the search is to confirm the existence of a code, and not to generate a large number of examples.

III. Proof of concept

In [13], page 98, the question arose when is the first codelength n, such that a Type I code is better than the best known Type II code. No such code is known up to now. Our searching software is doing an exhaustive search for each case. From here it follows, that for given (n, k, d) there may be permutation equivalent codes among all found codes, but there does not exist a code which is not equivalent to at least one of the found codes. By using our software we could easily compute the following facts, which are known results ([4], [12]):

1) There exist $(16, 8, 4)$ Type I and Type II codes. (In the case of Type II, this is a Reed-Muller code.)
2) There is no $(16, 8, 6)$ Type I code.
3) There exists at least one $(24, 12, 8)$ Type II code. (Known as Golay-code.)
4) There is no $(24, 12, 8)$ Type I code.
5) There is at least one $(24, 12, 6)$ Type I code.
6) There is at least one $(32, 16, 6)$ Type I code.
7) There is no $(32, 16, 10)$ Type I code. ($(32, 16, 8)$ Type II is a Reed-Muller code.)

8) There is at least one $(40, 20, 6)$ Type I code.

From here it follows, that if there exists such an (n, k, d) Type I code, which is better than the best (n, k, d) Type II code, then $n \geq 40$. Further, we conjecture that $n \geq 56$ if such a code would exist, but exhaustive search was not finished for those cases.

Furthermore, in [2] a possible construction for a self-dual $(72, 36, 16)$ code is given, which uses a $(56, 21, 16)$ Type II code. By computing with our software, we could find seven non-equivalent $(56, 20, 16)$ Type II codes, but no $(56, 21, 16)$-code.

Additionally, we found an extremal $(32, 16, 8)$ Type I code, whose existence was known before [3].

Some results of our computations, especially the generator matrices of the found codes mentioned before are available at

https://arato.inf.unideb.hu/hannusch.carolin/gm.txt

and some screenshots of the nonexistence of the codes mentioned before are available at

https://arato.inf.unideb.hu/hannusch.carolin/kepernyokepek.jpg

A strength of the software is to compute many different generator matrices for given (n, k, d) values in short time. The codes generated may be permutation equivalent and they may be not equivalent. This fact makes the Code Search Software attractive for the use in code-based cryptographic schemes.

In the following Table I we give some results of our computations, using different configurations of the Torch Code Searching Software. All computations were done on a normal-household laptop with Intel Core i7-7700HQ CPU (2.80GHz). Our results can be regarded as a proof of concept. For future research, we plan to use HPC supercomputer in order to use the Torch Code Searching Software in parallel computations and to fasten the searching procedure.

TABLE I
COMPUTATIONAL RESULTS

(n,k,d)	Type	using default configuration	perm_ equiv configuration
(16,8,4)	Type I	43 codes in 13.9 sec	1 code in 9.3 sec
(16,8,4)	Type II	27 codes in 7.5 sec	2 codes in 4.9 sec
(24,12,8)	Type II	not ending 1000 codes in 673 sec	1 code in 73.4 sec
(24,12,6)	Type I	not ending 1000 codes in 2524 sec	1 code in 2642 sec
(32,8,6)	Type I	not ending 1000 codes in 35.3 sec	not ending 11 codes in 5 hours

REFERENCES

[1] Brualdi, R. A., Pless, V. S. (1991). Weight enumerators of self-dual codes. IEEE transactions on information theory, 37(4), 1222-1225.

[2] Bouyuklieva, S. (2008). Self-Dual Codes with Some Applications to Cryptography. NATO Advanced Research Workshop, 6-9 October 2008, Veliko Tarnovo

[3] Bouyuklieva, S., Willems, W. (2012) IEEE Trans. Inf. Theory 58, No. 6, 3856–3860

[4] Conway, J. H., Pless, V. (1980). On the enumeration of self-dual codes. Journal of Combinatorial Theory, Series A, 28(1), 26-53.

[5] Conway, J. H., Sloane, N. J. (1990). A new upper bound on the minimal distance of self-dual codes. IEEE Transactions on Information Theory, 36(6), 1319-1333.

[6] Dougherty, S. T., Kim, J. L., Solé, P. (2015). Open problems in coding theory. Contemp. Math, 634, 79-99.

[7] Gulliver, T. A., Harada, M. (2008). On doubly circulant doubly even self-dual [72, 36, 12] codes and their neighbors. Australasian Journal of Combinatorics, 40, 137.

[8] Harada, M., Saito, K. (2018). Singly even self-dual codes constructed from Hadamard matrices of order 28. Australas. J. Combin, 70(2), 288-296.

[9] Joyner, D., Kim, J. L. (2011). Selected unsolved problems in coding theory. Birkhäuser.

[10] Overbeck, R., Sendrier, N. (2009). Code-based cryptography. In Post-quantum cryptography (pp. 95-145). Springer, Berlin, Heidelberg.

[11] Pless, V., Brualdi, R. A., Huffman, W. C. (1998). Handbook of coding theory. Elsevier Science Inc..

[12] Pless, V., Sloane, N. J. (1975). On the classification and enumeration of self-dual codes. Journal of Combinatorial Theory, Series A, 18(3), 313-335.

[13] Rains, E. M., Sloane, N. J. A. (1998). Self-dual codes, Handbook of coding theory, Vol. I, II.