МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2024
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2024
Proceedings of the Fifty-Third Spring Conference
of the Union of Bulgarian Mathematicians
Borovets, April 1–5, 2024

SORTING IN CONSTRUCTION OF RESOLUTIONS OF COMBINATORIAL DESIGNS*

Antoaneta Tsvetanova[1], **Stela Zhelezova**[2]
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences,
Sofia, Bulgaria
e-mails: [1]`a.cvetanova@math.bas.bg`, [2]`stela@math.bas.bg`

There are different kinds of sorting algorithms. Each algorithm has its own advantages and disadvantages depending on the data they process. We are interested in the performance of sorting algorithms in the case of construction of resolutions of combinatorial designs. Studying their performance for this special class of problems will give us the opportunity to improve, if possible, the speed of our software for solving similar problems. We use C++ and, for part of our investigations, the computer algebra system GAP.
Keywords: sorting, classification algorithm, combinatorial design, design resolution.

СОРТИРАНЕ НА ДАННИ ПРИ КОНСТРУИРАНЕ НА РЕЗОЛЮЦИИ НА КОМБИНАТОРНИ ДИЗАЙНИ

Антоанета Цветанова[1], **Стела Железова**[2]
Институт по математика и информатика, Българска академия на науките,
София, България
e-mails: [1]`a.cvetanova@math.bas.bg`, [2]`stela@math.bas.bg`

Съществуват много и различни алгоритми за сортиране. Всеки алгоритъм има своите предимства и недостатъци в зависимост от данните, които обработва. Разглеждаме работата на някои алгоритми за сортиране при задачи за конструиране на резолюции на комбинаторни дизайни. Проучването на тяхното представяне, за този специален клас проблеми, ще ни даде възможност да подобрим, ако е възможно, скоростта на нашия софтуер за конструиране на резолюции на комбинаторни дизайни. Използваме C++ и също така, за част от нашите изследвания, системата за компютърна алгебра GAP.
Ключови думи: сортиране, класификационен алгоритъм, резолюция на комбинаторен дизайн.

*`https://doi.org/10.55630/mem.2024.53.057-064`
Partially supported by the Bulgarian National Science Fund under Contract No KP-06-H62/2-2022.
2020 Mathematics Subject Classification: 68P10, 05-08, 05B25, 05B07.

1. Introduction. This paper presents research on the applicability of a number of sorting techniques in the classification of resolutions of combinatorial designs. For the basic knowledge on resolutions of combinatorial designs refer, for instance, to [11].

Let $V = \{P_i\}_{i=1}^{v}$ be a finite set of *points*, and $\mathcal{B} = \{B_j\}_{j=1}^{b}$ – a finite collection of k-element subsets of V, called *blocks.* If any 2-subset of V is contained in exactly λ blocks of $\mathcal{B}$, then $D = (V, \mathcal{B})$ is a 2-(v,k,λ) *design.* Each point is in r blocks.

A parallel class is a partition of set of points by blocks. There are q blocks in a parallel class. A *resolution* of the design is a partition of the collection of blocks into *parallel classes.* The design is *resolvable* if it has at least one resolution. Two resolutions are isomorphic if there exists an automorphism of the design transforming each parallel class of the first resolution into a parallel class of the second one. An automorphism of a resolution is an automorphism of the underlying design which maps each parallel class to a parallel class of the same resolution. The set of all resolution automorphisms is a group.

Resolutions of combinatorial designs are used in different types of error-correcting codes [10, 12], in network coding [14], cryptography [16] etc.

Part of the research on the topic has been done by computational methods. Many interesting classification results are obtained by computer, for instance, in [1, 3, 20, 21, 23]. Some authors use classification algorithms based on canonical augmentation [2] and also algorithms for computing of the code weight distribution in which no sorting is needed [17]. Equally important the *orderly generation* algorithms are in use too [6, 7, 13, 26]. They are based on the backtrack search technique and thus are with exponential time complexity. Nevertheless, for relatively small parameter sets and in combination with specific parameter depending theoretical conditions, they can be very successful. Apart from the use of theoretical restrictions, the algorithm implementation can be improved by refining the computation too.

In this paper we are interested in the performance of an auxiliary algorithm, namely the involved sorting algorithm. It appears in an orderly generation algorithm because it is connected with a particular lexicographic order imposed on the constructed objects. Sorted objects facilitate search and comparison and improve data operations efficiency. This, together with the frequently great number of combinatorial objects which are considered, leads to the impact of time spent for sorting on the whole computation time.

There are different kinds of sorting algorithms. Their performance and parameters are studied in depth by Knuth [15]. In our research we do not focus on their time complexity. It is known for all of them. Also there are papers on comparative analysis of various sorting algorithms in the general case [25, 30] and in case of data with a given specific distribution [4, 22]. We are interested in the performance of comparison-based unstable sorting algorithms suitable for the classification problems we consider. We study one and two-dimensional sorting which appears in the construction of resolutions of combinatorial designs. For each of them we apply a Cocktail Sort algorithm [15], the Introsort algorithm [18] (the built-in function in C++ [24]) and the Pattern-defeating quicksort algorithm [19] (the built-in *GAP* function [8]). For each sorting algorithm the maximal, minimal and the average time for execution of the sorting tasks are measured.

2. Sorting problems. We consider the construction of resolutions of a given combinatorial design with a prescribed automorphism group of order i (G_i). The generation method supposes the employment of a chosen lexicographic order. The elements involved in sorting are points or/and blocks of a particular combinatorial design. We can always denote them by integers. Design points can be enumerated $V = \{1, 2, \ldots, v\}$. Point labels are used to define a lexicographic order on the blocks and next to assign numbers to the blocks according to this order $\mathcal{B} = \{1, 2, \ldots, b\}$. The place and the size of a sorting task depends on the particular problem we consider.

2.1. Problem 1. In [27] parallelisms of $PG(3, 7)$ invariant under an automorphism group (G_{49}) of order 49 with some additional properties are constructed. They comprise resolutions of the point-line 2-$(400, 8, 1)$ design. It has $b = 2850$ blocks, each parallel class has $q = 50$ blocks and a resolution has $r = 57$ parallel classes. The design is known and the resolution construction starts with already lexicographically ordered design blocks. One way to derive resolutions invariant under G_{49} is by construction of all possibilities for parallel classes firstly. This can be done by backtrack search on the block orbits under G_{49}. These block orbits are considered on the lexicographically ordered design blocks. Therefore the first block in each orbit is the smallest one but the other blocks appear depending on the action of the considered automorphism group. Thus the blocks in each orbit are not necessarily in lexicographic order. In this case a parallel class consists of entire block orbits under G_7 – a particular subgroup of G_{49} and hence its blocks cannot appear in the chosen lexicographic order by construction. The structure of a parallel class under the given above conditions is presented in the first row of Table 1. Here f stands for a fixed under a particular G_7 block while O^i_j denotes j-th design block from the i-th orbit of length 7 under G_7. The second row corresponds to a particular parallel class with this structure. Each block is represented by its consecutive number in the chosen lexicographic order. For the purpose of our research we work with 3000 parallel classes. Each parallel class can be stored in a proper one-dimensional data container of integers named `Par_class` with size $q + 1 = 51$.

Table 1. A parallel class of a point-line 2-$(400, 8, 1)$ design fixed by a G_7

f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	...	O^5_1	O^5_2	O^5_3	O^5_4	O^5_5	O^5_6	O^5_7	O^6_1	O^6_2	O^6_3	O^6_4	O^6_5	O^6_6	O^6_7
9	58	142	212	240	331	359	429	...	716	1062	1408	1747	2093	2432	2778	779	1125	1471	1810	2156	2495	2841

2.2. Problem 2. In [28] the classification of point-cyclic resolutions of cyclic 2-$(45, 3, 1)$ designs is considered. A design is cyclic if it has a cyclic automorphism group of order v. A resolution with an automorphism permuting its points in one cycle is called point-cyclic. There are 11616 cyclic designs with these parameters [5]. It is well known that G_{45} partition the design blocks in one short block orbit of length $v/3 = 15$ and 7 block orbits of length $v = 45$. For tightness of the stored data such a design can be presented only by one block from each orbit (the smallest one in the chosen lexicographic order). In our case 8 blocks are enough. Each design block is a triple of points. With the agreement that the 8 block orbit representatives start with the first point they can be written by the numbers of the other two points only. The designs are stored in a file where each cyclic design corresponds to a row. The second row of Table 2 shows one such record with 8 groups of numbers. At the third row the corresponding design blocks are written. The first block represents the short orbit while the next 7 blocks are representatives of the long block orbits. We want to construct all point-cyclic resolution of this cyclic design.

One approach is to start with getting all design blocks by G_{45}. As can be seen from the last block in each orbit (the last row of Table 2) a cyclic shift of the block points leads to blocks with unordered points. The other problem is that if we write the orbits one after another to obtain all 330 blocks of a considered cyclic 2-$(45, 3, 1)$ design they will appear not in the chosen lexicographic order. Therefore we need to perform two-dimensional sorting on all design blocks. They can be stored in a proper two-dimensional data container of integers named `Deign_blocks` with size $b + 1 = 331$ and $k + 1 = 4$.

Table 2. Block orbits of a cyclic 2-$(45, 3, 1)$ design

orbit length	short(15)	long(45)	long	long	long	long	long	long
a record	16,31	2,4	5,13	8,22	7,29	11,27	10,33	6,19
block orbits	1,16,31	1,2,4	1,5,13	1,8,22	1,7,29	1,11,27	1,10,35	1,6,19
2$^{\text{nd}}$ block	2,17,32	2,3,5	2,6,14	2,9,23	2,8,30	2,12,28	2,11,36	2,7,20
last block	15,30,45	45,1,3	45,4,12	45,7,21	45,8,30	45,10,26	45,9,34	45,5,18

3. Sorting algorithms. Our programs are written in C++. Microsoft Visual C++ 2022 is used and the results are obtained on on Intel i3-7100U @ 2x2.4 GHz. The data we have to sort comprises only integers. Its entire size is fixed by the design parameters and hence is known in advance. The records in which they are included are not so long. The size of a Problem 1 like task is determined by q and usually is less than one hundred. For a Problem 2 like task the size is limited by r or by b and for the parameter sets we consider does not exceed one thousand. But the number of considered objects typically is very big. For instance there are 14227090 possible parallel classes invariant under a particular automorphism group of order 8 of the point-line 2-$(156, 6, 1)$ design (Problem 1 like task) and there are 2353310 cyclic 2-$(57, 3, 1)$ designs (Problem 2 like task). These are the reasons to choose as suitable for our investigations a Cocktail Sort algorithm, the Introsort algorithm and one of the sorting algorithms included in *GAP*. An orderly algorithm for construction of design resolutions generally includes one and two-dimensional sorting therefore it will be more effective if the data is stored in one and the same type containers for both problems.

3.1. Cocktail Sort. The *Cocktail Sort* algorithm (CSA) is based on the *Bubble Sort* algorithm (BSA). A BSA passes through an array from the beginning (left) to the end (right), compares elements and swaps them if their order is not correct [15, 24]. In the first iteration the largest element is settled on its exact place. Next the second largest element is moved to the right up to find its place etc. until all data is sorted. CSA does the same but in both directions. The first iteration is the same as for BSA while the second starts from the element before the just sorted one and goes from right to the left such that the smallest element is settled on place. If N is the number of sorted elements in the best case, the CSA time complexity is $O(N)$ and in the worst case is $O(N^2)$ as is for BSA. Nevertheless, there is an empirical analysis of CSA which shows that it is less than two times faster than BSA [29].

We implement CSA for Problem 1 in a standard manner in the function `CocktailSort (int arg[], int right)`. An one-dimensional array of integers `int Par_class[q+1];` is used to store each parallel class in Problem 1. It is passed to the function as a parameter. Its length (`right`) also has to be passed because the one-dimensional sort function is used for different data in many places in the implementation of our construction algorithm. For Problem 1 the function is called in the following way: `CocktailSort (Par_class, q);`.

For Problem 2 design blocks are stored in `int Deign_blocks[b+1][k+1]` – a standard two-dimensional array of integers. It is passed to the function as parameter but further CSA needs one more parameter. For a particular sorting task regarding design parameters it is known in how many positions two rows of the two-dimensional data array differ. The parameter `dept` depends on the number of these positions. The function declaration is `CocktailSort(int arg[][k+1], int right, int dept)`. Before swapping two unlike elements in position `dept` of rows i and $i+1$ we have to check if they coincide in all positions before `dept`. Thus for Problem 2 the function is called as:

```
for (int i = 1; i < k; i++)
        CocktailSort(Deign_blocks[][k+1], b, i);
```

3.2. Introsort. *Introsort* (Introspective sort) is a hybrid sorting algorithm incorporating *Insertion sort* for small set lengths, *Heapsort* for a recursion depth greater than a particular level and *Quicksort* in all other cases. More about the algorithm can be found in [18]. This combination ensures good time complexity in the worst and in the average case $O(N \log N)$, where N is the number of sorted elements.

In our algorithm implementation we use the Introsort algorithm incorporated in the C++ Standard Library as a built-in `std::sort()` function. It is included in the `<algorithm>` header file of C++. The `sort()` function cannot be used with a fixed two-dimensional array because an array cannot be assigned to another array. Therefore to apply the Introsort algorithm by the `sort()` function the data should be stored in `std::array` or `std::vector` classes. The main difference between them is that the first class is a fixed-size container while the second is a dynamic one. We use vectors but since the size of the data is known in advance the dynamic properties of the vectors are not employed here. This is important because a vector growth can lead to growth of the processing time.

Each parallel class in Problem 1 is stored in a vector `vector<int> Par_class(q+1);` and the design blocks in Problem 2 are written in a vector of vectors `vector<vector<int>> Deign_blocks(b+1, vector<int>(k+1));`. The range of sorting is given by iterators to the begin and to the end of the corresponding vector. We do not specify the function's third parameter as ascending order is used by default and it is suitable for the considered objects. Thus for Problem 1 we have `sort(Par_class.begin(), Par_class.end());` and for Problem 2 `sort(Deign_blocks.begin(), Deign_blocks.end());`.

3.3. Sorting data in GAP. The system for computational discrete algebra (*GAP*) is a freely distributed software. It is especially useful in group computations. Generally we use *GAP* to obtain the required for the constructive algorithm automorphism groups and their subgroups. In this paper we employ *GAP* in the needed sorting task. Three different sorting algorithms are involved in *GAP*. We apply the default one which is implemented in *GAP*'s function `Sort()`. It comprises the *Pattern-defeating quicksort* algorithm [19]. This quicksort modification is a hybrid sorting algorithm which contains the same sorting algorithms as Introsort, but with a different combining strategy. A novel scheme for partitioning in the quicksort part of the algorithm and a concept of a bad partitioning instead of exploiting a recursion depth are among the divergences from the Introsort algorithm. For a few common patterns this sorting algorithm achieves linear time complexity but in all other cases its time complexity is like that of the Introsort algorithm.

The syntax of the function is `Sort(list,func)` [9]. The parameter `func` gives the rule for comparing the elements and is omitted if the regular ascending order is adopted. Therefore to do necessary sort tasks in *GAP* we have to store our data in data structure called `list`. Its elements are written between square brackets [,] and separated by commas. For Problem 1 the sort function is called as `Sort(Par_class);`. For Problem 2 the data is stored in list of lists. We need to define a specific sorting function which answers how to compare two sub-lists that coincide in the first position. In this case the sort function is called as follows:

```
Sort(Deign_blocks, function(v, w)
        return v[1] < w[1] or (v[1] = w[1] and v[2] < w[2]);
```

4. Comparison. The times taken by the execution of each of the considered sorting algorithms for the problems mentioned above are given in Table 3. For each of them the minimal, maximal and average time for sorting an object is presented. To be more precise we get the running time in five identical trials. The values in the table are the averages of these trials.

It looks like the Introsort algorithm implemented in *C++* standard library has the best performance in the case of the one-dimensional Problem 1. Our Cocktail sort algorithm implementation is almost two times slower. The data size here is relatively small thus this result can be expected. The minimal time for one-dimensional sorting is the best for the Cocktail sort algorithm implementation because it refers to a case of almost sorted object.

For the two-dimensional Problem 2 the implementation of the Pattern-defeating quicksort algorithm in *GAP* is with the best performance. In this case our Cocktail sort algorithm implementation is almost 13 times slower. In Problem 2 we have to process 11616 objects so this difference is very important.

Our investigation shows that the implementation of the main constructive algorithm should better use the sort function from the C++ standard library. Thus the standard C++ arrays have to be replaced by vectors. Another possibility is to use the Pattern-defeating quicksort algorithm implementation for C++ which is under a zlib license and can be found at `https://github.com/orlp/pdqsort`.

Table 3. Times taken by the considered algorithm implementations in microseconds

Task	Algorithm	min	max	average
Problem 1	Cocktail sort	1	30999.7	50.4
	std::sort()	6.4	7700.4	27.3
	GAP Sort()	12.4	670	37.8
Problem 2	Cocktail sort	8844.4	186013.6	13307.5
	std::sort()	1588.6	115924.8	3092.2
	GAP Sort()	643.3	51162.7	1064.7

REFERENCES

[1] A. Betten. The packings of PG(3,3). *Des. Codes Cryptogr.*, **79**, no. 3 (2016), 583–595.

[2] I. Bouyukliev, S. Bouyuklieva, S. Kurz. Computer Classification of Linear Codes. *IEEE T Inform. Theory*, **67**, no. 12 (2021), 7807–7814.

[3] M. Braun. Construction of a point-cyclic resolution in PG(9,2). *Innovations in Incidence Geometry: Algebraic, Topological and Combinatorial*, **3**, no. 1 (2006), 33–50.

[4] J. Clement, Th. Nguyen Thi, B. Vallee. Towards a Realistic Analysis of Some Popular Sorting Algorithms. *Combinatorics, Probability and Computing*, **24**, no. 1 (2015), 104–144.

[5] C. J. Colbourn, A. Rosa. Triple Systems. Oxford, Clarendon Press, 1999.

[6] M. Dzhumalieva-Stoeva, I. G. Bouyukliev, V. Monev. Construction of self-orthogonal codes from combinatorial designs. *Probl. Inf. Transm.*, **48** (2012), 250–258 .

[7] I. A. Faradžev. Constructive enumeration of combinatorial objects, In Problèmes Combinatoires et Théorie des Graphes, (Université d'Orsay, July 9 – 13, 1977). *Colloq. Internat. du C.N.R.S.*, **260** (1978), 131–135.

[8] GAP – Groups, Algorithms, Programming – a System for Computational Discrete Algebra, `http://www.gap-system.org/`. Last accessed 21 December 2023.

[9] GAP – Reference Manual, Release 4.12.2, 2022-12-18, the GAP Group. Last accessed 21 December 2023.

[10] A. Gruner, M. Huber. New Combinatorial Construction Techniques for Low-Density Parity-Check Codes and Systematic Repeat-Accumulate Codes. *IEEE T Commun.*, **60**, no. 9 (2012), 2387–2395.

[11] Handbook of Combinatorial Designs, C. Colbourn, J. Dinitz,(eds.) 2nd edn. In: Rosen, K. (eds.) Discrete mathematics and its applications, Boca Raton, FL., CRC Press 2007.

[12] S. J. Johnson, S. R. Weller. Resolvable 2-designs for regular low-density parity-check codes. *IEEE T Commun.*, **51**, no. 9 (2003), 1413–1419.

[13] P. Kaski, P. Östergård. Classification algorithms for codes and designs. Berlin, Springer, 2006.

[14] R. Koetter, F. R. Kschischang. Coding for errors and erasures in random network coding. *IEEE T Inform. Theory*, **54** (2008), 3579–3591.

[15] D. E. Knuth. The art of computer programming 3: Sorting and Searching, 2nd edn. Addison-Wesley, 1998.

[16] A. Kumar, S. Maitra. Resolvable block designs in construction of approximate real MUBs that are sparse. *Cryptogr. Commun.*, **14** (2022), 527–549 .

[17] M. Markov, Y. Borissov. Computing the Weight Distribution of the Binary Reed-Muller Code R(4,9). arXiv:2309.10462.

[18] D. R. Musser. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience*, **27**, no. 8 (1997), 983–993.

[19] O. R. Peters. Pattern-defeating Quicksort. arXiv./abs/2106.05123, 2021.

[20] A. R. Prince. The cyclic parallelisms of PG(3,5). *Eur. J. Combin.*, **19**, no. 5 (1998), 613–616.

[21] G. F. Royle. An orderly algorithm and some applications in finite geometry. *Discrete Math*, **185**, no. 1–3 (1998), 105–115.

[22] A. Sabah, S. Abu-Naser, Y. Helles, R. Abdallatif, F. Y. A. Abu Samra, A. H. Abu Taha, N. M. Massa, A. A. Hamouda. Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics. *Int. J. Academic Eng. Research (IJAER)*, **7**, no. 6 (2023), 76–84.

[23] J. Sarmiento. Resolutions of PG(5,2) with point-cyclic automorphism group. *J. Combin. Des.*, **8**, no. 1 (2000), 2–14.

[24] Software Testing Help, Sorting Techniques In C++, `https://www.softwaretestinghelp.com/sorting-techniques-in-cpp/`. Last accessed 21 December 2023.

[25] B. Subbarayudu, L. L. Gayatri, P. S. Nidhi, P. Ramesh, R. G. Reddy, K. K. Reddy C. Comparative Analysis on Sorting and searching Algorithms. *Int. J. Civil Eng. Technol. (IJCIET)*, **8**, no. 1 (2017), 955–978.

[26] S. Topalova, S. Zhelezova. Backtrack Search for Parallelisms of Projective Spaces, In: Flocchini P., Moura L. (eds.) Combinatorial Algorithms, IWOCA 2021. Lect. Notes Comput. Sci., vol. **12757**, (2021), 544–557.

[27] S. Topalova, S. Zhelezova. Transitive Deficiency One Parallelisms of PG(3,7). *Mathematics*, **11** (2023), 2458.

[28] S. Topalova, S. Zhelezova. Point-cyclic KTS(45). 18th Annual Meeting of the Bulgarian Section of SIAM, BGSIAM'23 Extended abstracts (2023), 48–49.

[29] A Computer Science portal, `https://www.geeksforgeeks.org/cocktail-sort/`. Last accessed 21 December 2023.

[30] CodersLegacy, `https://coderslegacy.com/comparison-of-sorting-algorithms/`. Last accessed 21 December 2023.