On the Difficulty of Evolving Permutation Codes

Luca Mariot[1], Stjepan Picek[1], Domagoj Jakobovic[2], Marko Djurasevic[2], and Alberto Leporati[3]

[1] Digital Security Group, Radboud University
Postbus 9010, 6500 GL The Netherlands
{luca.mariot, stjepan.picek}@ru.nl
[2] Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, Zagreb, Croatia
{domagoj.jakobovic, marko.durasevic}@fer.hr
[3] DISCo, Università degli Studi di Milano-Bicocca,
Viale Sarca 336/14, 20126 Milano, Italy
alberto.leporati@unimib.it

Abstract. Combinatorial designs provide an interesting source of optimization problems. Among them, permutation codes are particularly interesting given their applications in powerline communications, flash memories, and block ciphers. This paper addresses the design of permutation codes by evolutionary algorithms (EA) by developing an iterative approach. Starting from a single random permutation, new permutations satisfying the minimum distance constraint are incrementally added to the code by using a permutation-based EA. We investigate our approach against four different fitness functions targeting the minimum distance requirement at different levels of detail and with two different policies concerning code expansion and pruning. We compare the results achieved by our EA approach to those of a simple random search, remarking that neither method scales well with the problem size.

Keywords: permutation codes, evolutionary algorithms, incremental construction, powerline communications, flash memories, block ciphers

1 Introduction

Permutation codes (also called permutation arrays) are a particular kind of error-correcting codes where the codewords are permutations. In particular, a permutation code $PA(n, d)$ is a set of permutations of length n such that any two permutations in it disagree in at least d positions.

These combinatorial objects have several applications, for example, as error-correcting codes in *powerline communications* [2]. The basic approach for powerline transmission is to encode the data by small voltage variations, with the requirement of keeping the power output as constant as possible. Further, transmission over powerlines is affected not only by white Gaussian noise but also by impulse and narrow-band noise due to electrical interference and magnetic

fields. Using permutation codes in a modulation scheme, as suggested by Han Vinck [10], provides a good trade-off between power variation and correcting errors introduced by these kinds of noise. A second domain where permutation codes have been extensively applied is *flash memories*, particularly in the so-called *rank-modulation scheme* [11]. In traditional designs, the cells in a flash memory encode the information using different charge levels, allowing them to store a set of discrete values. On the other hand, rank-modulation encodes the data in the cells with a permutation that specifies the relative ranks of the charges instead of directly using their absolute values. Using a permutation code in this scheme improves the writing speed and the correction of errors introduced by charge leakage, which becomes progressively frequent in aging memories.

Finally, permutation codes have also been applied to a smaller extent in cryptography, specifically for the design of *block ciphers* [24]. In the *Substitution-Permutation Network* (SPN) paradigm for block ciphers, the plaintext is encrypted by iteratively applying several times a *round function*. The round function, in turn, consists of a *confusion layer*, which aims at making the relationship between the ciphertext and the symmetric key as complicated as possible, and a *diffusion layer*, whose goal is to spread the statistical structure of the plaintext over the ciphertext. The resulting block is mixed with a *round key* to get the corresponding ciphertext, which is then given as input to the next application of the round function. The diffusion layer is usually implemented through a *Maximum Distance Separable (MDS) matrix* as it happens, for example, in AES [6]. An alternative approach is to use a set of different permutations coming from a permutation code $PA(n, d)$. By dynamically choosing a different permutation from the code at each round, two different input blocks are guaranteed to result in output blocks at Hamming distance of at least d, thereby implementing a *multi-permutation* as defined by Vaudenay [25].

Despite their simple definition, the construction of permutation codes is far from being a trivial problem. Indeed, finding the largest permutation code is a particular instance of the *sphere-packing problem* studied in coding theory [5], and of the Max-Clique problem in graph theory, which is known to be **NP**−complete [12]. In particular, one of the main open questions in this research field is to determine the largest permutation code for a given length n and minimum distance d, i.e., the maximum number of permutations that can partake in a $PA(n, d)$. Such a number is usually denoted as $M(n, d)$, and its exact value is known only for a few specific cases. Generally, one resorts to coding-theoretic results to provide lower and upper bounds on $M(n, d)$. Apart from algebraic constructions, for which the reader may find a survey in [3], a few heuristic algorithms have also been developed to construct large permutation codes [21,18], mostly based on branch and bound and iterative clique search approaches. As far as we know, up to now, there have been no attempts in the literature to employ Evolutionary Algorithms (EA) to address this problem, although some authors used EA in the past to evolve other kinds of combinatorial designs, such as *orthogonal Latin squares* [16], *orthogonal arrays* [17] and *disjunct matrices* [14].

This paper investigates the suitability of EA to optimize permutation codes. We do so by from the previous permutations. The process is repeated until either a given fitness budget expires or an upper bound on $M(n,d)$ is reached (since this means that the code cannot be expanded further). We evaluate our incremental EA approach under four fitness functions and two *update policies*. The first update policy expands the code as soon as a suitable permutation is found by the EA. The second policy also removes some rows at random from the current code after a certain amount of fitness evaluations with no improvement has elapsed. The number of removed rows is decreased over time, similarly to the cooling schedule used in simulated annealing. For the sake of comparison, we also adopt a baseline random search (RS) method and perform experiments over 15 problem instances. The results show that both EA and RS cannot scale well on this optimization problem, with the largest codes found that lie far from the best-known lower bounds [20].

2 Preliminaries

We denote by $[n]$ the set $\{1, \cdots, n\}$ of the first $n \in \mathbb{N}$ positive integers. Next, S_n denotes the *symmetric group* of order n, i.e., the set of all permutations over $[n]$. Given a permutation $\pi \in S_n$, we encode it as a vector $\pi = (p_1, \cdots, p_n)$ of length n, where each coordinate $\pi[i]$ specifies the value of the permutation when evaluated on $i \in [n]$. Further, given two permutations $\pi, \sigma \in S_n$, the *Hamming distance* $d_H(\pi, \sigma)$ is the number of coordinates where π and σ differ.

Definition 1. *Let $n \in \mathbb{N}$ and $d \leq n$. A* permutation code *(also* permutation array*, PA) of length n and minimum distance d, denoted by $PA(n,d)$, is a subset P of the symmetric group S_n such that $d_H(\pi, \sigma) \geq d$ for every pair of distinct permutations $\pi, \sigma \in P$.*

Using the error-correcting codes terminology, the permutations in a $PA(n,d)$ are also called *codewords*. If P is composed of m codewords, one can represent it through a $m \times n$ matrix where each row corresponds to one of the permutations in P. The ordering of the rows in such a matrix is irrelevant since it does not change the pairwise Hamming distances of the permutations. In the following, we will mostly use this matrix-based notation to represent PA, although the set-theoretic notation will also be useful to describe the operations performed by our evolutionary algorithms to search for such arrays.

One of the main problems is determining the largest number of codewords that can partake in a permutation code. Following the notation from [3], given n and d we denote by $M(n,d)$ the maximum number of rows in a $PA(n,d)$. The values of $M(n,d)$ are generally unknown, but several theoretical bounds exist. Two well-known results in this direction, originating from coding-theoretical considerations, are the *Gilbert-Varshamov lower bound* and the *sphere-packing upper bound*, which we summarize below for permutation codes.

Theorem 1. *Let $n, d \in \mathbb{N}$ with $d \leq n$. Then, the following inequalities hold for the maximum number of codewords in a permutation code:*

$$\frac{n!}{\sum_{k=0}^{d-1} \binom{n}{k} D_k} \leq M(n,d) \leq \frac{n!}{\sum_{k=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{k} D_k} \quad , \tag{1}$$

where D_k is the number of derangements *of k elements (i.e., the number of permutations of length k without fixed points), which for all $k \in \mathbb{N}$ equals:*

$$D_k = k! \sum_{i=0}^{k} \frac{(-1)^i}{i!} \quad . \tag{2}$$

The Gilbert-Varshamov and sphere-packing bounds are rather crude, but in practice, they are helpful to decide whether a specific instance of n and d is suitable to construct a permutation code large enough for a specific application. Of course, tighter bounds have been proved in the related literature of permutation codes, either by combinatorial arguments or by providing concrete constructions. The latter case usually occurs for specific values of the minimum distance. For instance, if $d = n$, then it is rather easy to constructively prove that $M(n, n) = n$, by considering any *Latin square* of order n as an example of permutation code reaching this bound. Indeed, a Latin square of order n is a $n \times n$ array such that each number in $[n]$ appears exactly once in each row and each column. Thus, each row of the square is a permutation, and any two rows differ in all coordinates since there cannot be any repeated number in any column. A simple construction for a Latin square of order n is to take all *cyclic shifts* of the identity permutation $(1, 2, \cdots, n)$, which proves the existence of a $PA(n, n)$ for every $n \in \mathbb{N}$.

Latin squares also provide a construction for a better lower bound on $M(n, d)$ when $d = n - 1$. In particular, two Latin squares are called *orthogonal* if their superposition yields all ordered pairs in the Cartesian product $[n] \times [n]$, and a set of k *mutually orthogonal Latin squares* (k-MOLS) is a family of k Latin squares of order n that are pairwise orthogonal. Colbourn et al. [4] showed how to construct a $PA(kn, n - 1)$ by using a set of k-MOLS of order n, thereby proving that $kn \leq M(n, n - 1)$. We emphasize that determining the maximum size of a MOLS family for a given n is also a long-standing open problem in design theory, but several results are known for specific cases [22].

It is also easy to determine the maximum number of rows in a permutation array for low minimum distances. Indeed, two permutations cannot differ in only one position since this would imply that both vectors have a repeated value (thus, not making it a permutation). Therefore, the minimum distance is always at least 2, i.e., when two permutations differ by a single *transposition*, or swap. This means that the largest $PA(n, 2)$ corresponds to the symmetric group S_n itself, hence $M(n, 2) = n!$. Additionally, any two distinct permutations in the *alternating group* A_n (i.e., the set of *even* permutations of length n) are always at a minimum distance of 3. Since the alternating group is exactly half of the size of the symmetric group, it follows that $M(n, 3) = n!/2$. We conclude this section by summarizing the above results as follows:

Theorem 2. *Let $n, d \in \mathbb{N}$ with $d \leq n$. Then:*
- $M(n,1) = 1$; $M(n,2) = n!$; $M(n,3) = n!/2$;
- $M(n,n) = n$; $kn \leq M(n, n-1)$, *if there exists a set of k-MOLS of order n.*

Tables reporting more refined lower and upper bounds for various values of n and d may be found in [20].

3 Incremental Construction with EA

From an intuitive point of view, it seems natural to cast the search of a permutation code as a combinatorial optimization problem. Given the length n of the permutations, the (minimum) distance d and the number m of desired permutations in the array, one needs to find a set of m elements from S_n such that the Hamming distance between any two permutations in it is at least d. Therefore, disregarding the bounds on m induced by the distance parameter, the size of the resulting search space $\mathcal{S}_{m,n}$ is $|\mathcal{S}_{m,n}| = \binom{n!}{m}$ since we need to pick m elements from a set of size $n!$. Exhaustively searching for a solution would be already prohibitive for very small values of n and m: for example, there are only $|S_5| = 120$ permutations of length $n = 5$. However, visiting all subsets of S_5 of size $m = 12$ would imply a search space of $\binom{120}{12} \approx 1.05 \cdot 10^{17}$ elements, which clearly cannot be explored in a reasonable amount of time. Consequently, it seems interesting to address this optimization problem with evolutionary algorithms.

3.1 Evolving Subsets of Permutations

Given n, m and d, a straightforward option is to set up an EA that searches for permutations codes by directly evolving a set of m permutations. A candidate solution in the population is represented as a matrix A of size $m \times n$, where each row is a permutation of the set $[n]$. Then, this candidate solution would be evaluated through a fitness function that measures how close is A from being a $PA(n,d)$. This could be accomplished, e.g., by counting the number of pairs of rows in A that are at Hamming distance at least d and maximizing such fitness. In this approach, one could use common operators for permutation-based chromosomes. For crossover, these include among others *partially mapped crossover* [9] and *cycle crossover* [19]. For mutation, the most natural solution is to apply a simple swap operator that randomly exchanges two values in a permutation [1], but other methods have been proposed such as, e.g., the *inversion operator* [8] and the *scramble operator* [23]. Still, when evolving permutation codes, one deals with sets of permutations. Thus, a possible solution for this problem would be to apply the variation operators in a *row-wise* manner. For example, given two $m \times n$ arrays A and B, define an offspring array C by first applying a permutation-based crossover to the first row of A and B, then to the second one, and so on until C is completed. Although straightforward, this optimization approach suffers from several drawbacks:

- As the aim is constructing a permutation array in an "all-at-once" fashion, an EA would directly explore m-subsets of the symmetric group S_n, which results in a very large search space already for small values of m and n.
- The fitness function would need to consider the Hamming distance of each pair of rows in the arrays. Hence, the computational complexity required to evaluate a single candidate solution would be quadratic in the number of permutations of the array, as there are $\binom{m}{2} = \frac{m(m-1)}{2} = \mathcal{O}(m^2)$ pairwise Hamming distances to compute in an $m \times n$ array.
- This optimization approach relies on the number of rows m composing the desired permutation to define the problem instance. This implies that one would need to check in advance if m rows are attainable by a permutation code of length n and distance d.

3.2 Iterative Approach

Given the problems featured by the "all-at-once" method, we chose to follow an *iterative* optimization approach, greatly reducing the search space handled at each step by the evolutionary algorithm. Given $n, d \in \mathbb{N}$, the idea is to start from an empty set and add a random permutation $p_1 \in S_n$ of length n: trivially, P forms a $PA(n, d)$ with $m = 1$ rows. Then, an EA evolves a single permutation $p_2 \in S_n$, until it finds one whose Hamming distance from p_1 is at least d. When it is found, p_2 is added to P, thereby expanding the permutation code to $m = 2$ rows. The process is repeated by evolving a new permutation until a general termination criterion is met, such as reaching a theoretical upper bound for $M(n, d)$ or a specified number of fitness evaluations. By construction, the obtained array will be a permutation code $PA(n, d)$ with a certain number of rows m. At each stage, the EA only explores the set S_n of all permutations of length n instead of the whole set of m-subsets of permutations.

More formally, given a $PA(n, d)$ $P = \{p_1, \cdots, p_m\}$ with m rows, the decoding of a candidate chromosome $p_{m+1} \in S_n$ results in the following phenotype: $P' = P \cup \{p_{m+1}\}$. Clearly, since P already satisfies the properties of a permutation code of minimum distance d, the fitness of the candidate solution P' encoded by p_{m+1} is evaluated only by taking into account the m Hamming distances $d_H(p, p_{m+1})$, with p ranging over P. This is a much more efficient fitness function since its computational complexity scales linearly with the number of rows in P. Further, suppose that $\chi : S_n \times S_n \to S_n$ is a crossover operator for single permutations. Then, given two parent permutations $p_1, p_2 \in S_n$, the phenotype C for the offspring child candidate solution is defined as $C = P \cup \{\chi(c_1, c_2)\}$, that is, crossover is limited only on the new row. Accordingly, the same approach is adopted for mutation by applying the corresponding operator $\mu : S_n \to S_n$ only on the new permutation optimized by the EA.

Algorithm 1 reports the pseudocode for the incremental EA informally introduced above. The input parameters are the length of the permutations n, the required minimum distance d, the fitness budget fb, the target number of rows M (which specifies, for example, a known upper bound for $M(n, d)$), the size

Algorithm 1 INCREMENTAL-EA-PA($n, d, fb, M, popsize, \theta$)

```
P ← {}
π ← GEN-RAND-PERMUTATION(n)
P ← P ∪ {π}
eval ← 0
while |P| < M AND eval < fb do
    pop ← INIT-POPULATION(n, popsize)
    EVALUATE-FITNESS(pop, P, n, d, ev)
    best ← UPDATE-BEST-IND(pop)
    while eval < fb AND (NOT IS-PA(n, d, P, best.π)) do
        pop ← UPDATE-POP-EA(n, d, P, pop, θ, eval)
        best ← UPDATE-BEST-IND(pop)
    if IS-PA(n, d, P, best.π) then
        P ← P ∪ {best.π}
return P
```

of the EA population *popsize*, and a vector θ specifying the parameters for the underlying evolutionary algorithm.

The subroutines GEN-RAND-PERMUTATION and INIT-POPULATION respectively generate at random a single permutation and a population of *popsize* candidate permutations. An individual *ind* in the population is assumed to be a record composed of two items, namely $ind.\pi$ (the vector specifying the permutation) and $ind.fit$ (the fitness value of the permutation). EVALUATE-FITNESS computes the underlying fitness function for each individual in the population, while UPDATE-BEST-IND returns a pointer to the best individual in the current population. The specific structures for these two subroutines depend on the details of the fitness function, which we will address in the next section. IS-PA is a predicate returning true if and only if the union of a $PA(n, d)$ and a new permutation is still a $PA(n, d)$, and it is used to determine when to exit from the inner while loop of the EA. When an optimal solution is found, IS-PA($n, d, P, best.\pi$)) returns true, and the permutation code P is extended by adjoining to it the permutation of the best individual in the population.

The actual EA is implemented by the UPDATE-POP-EA subroutine. In particular, depending on the underlying EA, the population might be updated completely, as in a generational approach (possibly coupled with elitism) or only partially, with only a few new offspring individuals entering into the population at each step. In our experiments, we adopted a steady-state genetic algorithm (GA) with tournament selection. This means that each time UPDATE-POP-EA is invoked, t individuals are drawn at random from the population, and the two with the best fitness values are selected for crossover. The resulting offspring then undergoes mutation with probability p_μ, that replaces the worst individual in the tournament. Algorithm 2 gives the pseudocode for our steady-state GA implementing the UPDATE-POP-EA subroutine. The parameters vector θ is replaced by the pair (t, p_μ), whose components respectively specify the tournament size and the mutation probability. Notice also that *eval*, which is a counter used to keep track of the number of fitness evaluations performed by the algorithm, is assumed to be a global variable: in fact, it is used in the invariants for the while loops in Algorithm 1.

Algorithm 2 UPDATE-POP-EA$(n, d, P, pop, (t, p_\mu), eval)$

$tourn \leftarrow$ RANDOM-SELECT(t, pop)
$(p_1, p_2) \leftarrow$SELECT-BEST$(tourn)$
$c \leftarrow$ CROSSOVER(p_1, p_2)
$c \leftarrow$MUTATION(c, p_μ)
$c.fit \leftarrow$ FITNESS(n, d, P, c)
$eval \leftarrow eval + 1$
$worst \leftarrow$ SELECT-WORST$(tourn)$
REPLACE$(worst, c)$
return pop

The subroutines RANDOM-SELECT, SELECT-BEST, and SELECT-WORST respectively return a random subset of t individuals from the population, the two best individuals and the worst one in the tournament concerning their fitness values. The offspring chromosome is created from p_1 and p_2 by first applying CROSSOVER and then MUTATION. After evaluating the fitness function – and increasing the counter of fitness evaluations – the subroutine REPLACE changes the worst individual in the tournament to the newly created offspring.

3.3 Fitness Functions

We defined four fitness functions to be optimized by the iterative EA described in the previous section, which we describe below. In what follows, we assume that the goal is to compute the fitness of a permutation $p \in S_n$ when adjoined to a $PA(n, d)$ of m rows, $P = \{p_1, \cdots, p_m\}$.

The first fitness function directly sums the Hamming distances of each pair (p, p_i) of permutations, but only if they are at least equal to the required minimum distance d:

$$fit_1(p) = \sum_{p_i \in P} \delta_i \cdot d_H(p, p_i), \text{ where } \delta_i = \begin{cases} 1, & \text{if } d_H(p, p_i) \geq d, \\ 0, & \text{otherwise} \end{cases} . \quad (3)$$

Note that this fitness function completely neglects the permutation pairs' information at Hamming distance lower than d. For this reason, the second fitness function has the same form of fit_1, but also takes into account the invalid pairs by discounting them through an exponential factor:

$$fit_2(p) = \sum_{p_i \in P} \delta'_i \cdot d_H(p, p_i), \text{ where } \delta'_i = \begin{cases} 1, & \text{if } d_H(p, p_i) \geq d, \\ 2^{d_H(p,p_i)-d}, & \text{otherwise} \end{cases} . \quad (4)$$

Indeed, when $d_H(p, p_i) < d$ the factor δ'_i is a number between 0 and 1, which decreases as the difference $d_H(p, p_i) - d$ gets smaller. In this way, the more a pair (p, p_i) is closer to the required minimum distance d, the more it contributes to the fitness function.

The third fitness function considered in our experiments corresponds to the minimum Hamming distance between p and each permutation in P, that is,

$$fit_3(p) = \min_{p_i \in P} \{d_H(p, p_i)\} \quad . \quad (5)$$

Hence, the permutation p is an optimal solution as soon as $fit_1(p) \geq d$, since this is precisely the characterizing property of a $PA(n,d)$. Although straightforward, this fitness function suffers from a limited range of possible values (especially for small values of d), making many candidate solutions very similar. This may hamper, in turn, the EA's ability to exploit specific regions of the search space.

The three fitness functions described up to now are all meant to be maximized as an optimization objective. On the contrary, the fourth fitness function is based on counting the number of pairs that do not meet the minimum distance requirement, clearly with the objective of minimizing them:

$$fit_4(p) = |\{(p, p_i) : p_i \in P,\ d_H(p, p_i) < d\}| \quad . \tag{6}$$

4 Experimental Evaluation

A problem instance for the permutation array problem is defined by the length of the permutation n and the (minimum) distance d. To evaluate the suitability of the incremental EA on this problem, one possibility is to compare the maximum number of rows obtained by it for a $PA(n,d)$ and the corresponding lower/upper bounds known in the literature. As far as we are aware, Smith and Montemanni [20] report the most up-to-date table that reports such bounds for $6 \leq n \leq 18$ and $4 \leq d \leq 18$. Since d must always be less than or equal to n, the total number of problem instances to be tested for a complete comparison is $\sum_{i=3}^{15} i = 117$, which might be unfeasible depending on how much time a single run of the EA takes. Therefore, it makes sense to perform the experiments on a subset of instances, limiting the size of the permutations to $n = 10$ and minimum distance $n - 2 \leq d \leq n$. In this way, we get a total of 15 problem instances to test. These instances also have practical relevance in the design of modulation schemes for powerline communications (see, e.g., [7], where PA of length at most 8 are considered for this task) and for the design of block ciphers, where $n = 8$ is a popular permutation size in the diffusion layers of lightweight block ciphers [15]. The instances where $n = d$ correspond to the problem of finding a Latin square of order n. Furthermore, although the size of the symmetric group S_n, for $6 \leq n \leq 10$, is sufficiently limited to be completely explored, recall that the unfeasibility of the exhaustive search approach stems from the fact that we are trying to construct *subsets* of permutations. This already yields a search space of size $\binom{6!}{120} \approx 3.07 \cdot 10^{140}$, for the $PA(6,4)$ instance, and thus it cannot be exhaustively explored. For each considered combination of n and d, Table 1 reports the size of the corresponding search space computed as $\binom{n!}{M(n,d)}$ and the corresponding best value known for $M(n,d)$ taken from [20]. The search space size decreases as the minimum distance approaches the length, with $n = d$ giving the smallest sizes – although still not amenable to exhaustive search.

4.1 Experimental Settings

In our experiments, we evaluated our evolutionary approach to construct PA along three different components: namely, the *fitness functions* described in

Table 1: Approximate search space size $\mathcal{S}_{n,d}$ and code size bound $M(n,d)$ for each considered problem instance. Bold values represent non-tight lower bounds.

$d\backslash n$		6	7	8	9	10
$n-2$	$\mathcal{S}_{n,d}$	$3.07 \cdot 10^{140}$	$2.31 \cdot 10^{277}$	$1.81 \cdot 10^{843}$	$1.20 \cdot 10^{1\,658}$	$3.83 \cdot 10^{2\,978}$
	$M(n,d)$	120	**77**	336	504	720
$n-1$	$\mathcal{S}_{n,d}$	$3.41 \cdot 10^{36}$	$1.91 \cdot 10^{106}$	$1.10 \cdot 10^{184}$	$3.26 \cdot 10^{297}$	$1.61 \cdot 10^{453}$
	$M(n,d)$	18	42	56	72	49
n	$\mathcal{S}_{n,d}$	$1.89 \cdot 10^{15}$	$1.63 \cdot 10^{23}$	$1.73 \cdot 10^{34}$	$3.01 \cdot 10^{45}$	$1.10 \cdot 10^{60}$
	$M(n,d)$	6	7	8	9	10

Section 3.3, the underlying *search algorithm*, and the adopted *update policy*. The search algorithm refers to the particular procedure used to select a suitable permutation to be added in the current code, i.e., the content of the while loop at lines 11-12 in Algorithm 1. In this case, we adopted a permutation-based genetic algorithm (which we will refer to as EA in the following) and a simple random search (RS) as a baseline method for comparison. In particular, the RS works by drawing at random a new permutation at each iteration of the while loop, which is subsequently added only if it is at a minimum distance d from all previous permutations. On the other hand, the EA follows the Update-Pop-EA steady-state procedure described in Algorithm 2.

The update policy is the strategy by which the algorithm constructs the permutation code. The iterative approach laid out in Section 3 is based on the Incremental-EA-PA procedure (Algorithm 1), which only expands the code when a new permutation at minimum distance d from all the current ones is found. However, this update policy might easily get stuck in local optima. Intuitively, the size achievable by a permutation array constructed incrementally highly depends on the initial permutations chosen. Therefore, if the search algorithm makes a few "wrong choices" initially, it might end up with a relatively small list of permutations that cannot be further expanded.

For this reason, we also experimented with a *random reset* update policy: if a new permutation satisfying the minimum distance requirement is not found within a given number of fitness evaluations in the inner while loop of Algorithm 1, then some previous permutations – chosen at random – are removed from the current code. Also, the number of permutations to be removed is chosen randomly, but the maximum value is modeled after the cooling policy as employed in simulated annealing [13] Initially, the maximum number of codewords to remove can be as high as one-third of the current PA size ($|P|$) but is then decreased at every subsequent random reset, in order to favor the exploration of the search space at the beginning of the optimization process and its exploitation in the later stages. The actual number of permutations to be removed is a random value in $\{1, \cdots, r\}$, where r is set as $r = \frac{1}{3} \times |P| \times e^{-evals/10^6}$. The condition to invoke this reset is defined as the number of successive evaluations without increasing

the PA size, e.g., the number of unsuccessful attempts to add a new permutation to the current PA. In our experiments, this number was defined as $\max(n!, 10^5)$; the reasoning behind this is that $n!$ evaluations of the exhaustive search would be enough to find out whether any new permutation can be added to the current PA. Consequently, we use this number as the stagnation detection threshold. Note that the random reset policy can be used with any search algorithm (i.e., any type of population update) and any fitness function.

In what follows, we will denote by EA1 and EA2 the incremental EA equipped respectively with the plain update policy as in Algorithm 1 (where new permutations are only added) and with the above random reset update policy. Likewise, RS1 and RS2 will denote the analogous variants of the RS baseline algorithm concerning the update policy.

Prior to the experiments on the selected problem instances, a short tuning phase was performed on problem instance $PA(7, 5)$ to estimate the appropriate parameter values for the population size and the mutation rate p_μ of the GA. Based on those results, the population size was set to 1 000 individuals, and the mutation rate was kept at 30%. In all the experiments, the total number of evaluations (the fitness budget fb) was set to 10^7, and each experiment was executed in 30 repetitions. Concerning the variation operators, we employed the permutation-based crossovers and mutations implemented in the ECF framework[1], by choosing them uniformly at random at each evaluation.

4.2 Results

Figures 1 and 2 display the results obtained in our experiments with all search methods and across all considered problem instances, except those where $n = d$. In fact, in all those cases, each search variant managed to construct a $PA(n, n)$, or equivalently a Latin square of order n. For each problem instance (n, d), the corresponding boxplot shows the four fitness functions against the largest code size achieved by the corresponding variant of a search method. The legend for the four considered combinations of search method and update policy is reported on top of each figure.

First, one can see from the plots that all considered methods, independently from the fitness function, the search method, and the update policy, cannot scale very well concerning the problem size. Indeed, optimal solutions reaching known values for $M(n, d)$ are consistently found only in the $n = 6$ case, with the exception of a single $PA(7, 6)$ of size 42 found by EA2 with fitness function fit_3. Contrarily, for $n \geq 7$, all considered variants find PA that are significantly smaller than the best-known bounds reported in [20]. Our methods are always able to outperform the Gilbert-Varshamov bound, which is, however, quite loose as reported in Section 2.

As expected, the random reset update policy generally achieves better results than the plain one. This effect is particularly evident in the $n = 6$ problem instances, with the combinations adopting the plain update policy obtaining

[1] Framework available at `http://ecf.zemris.fer.hr/`

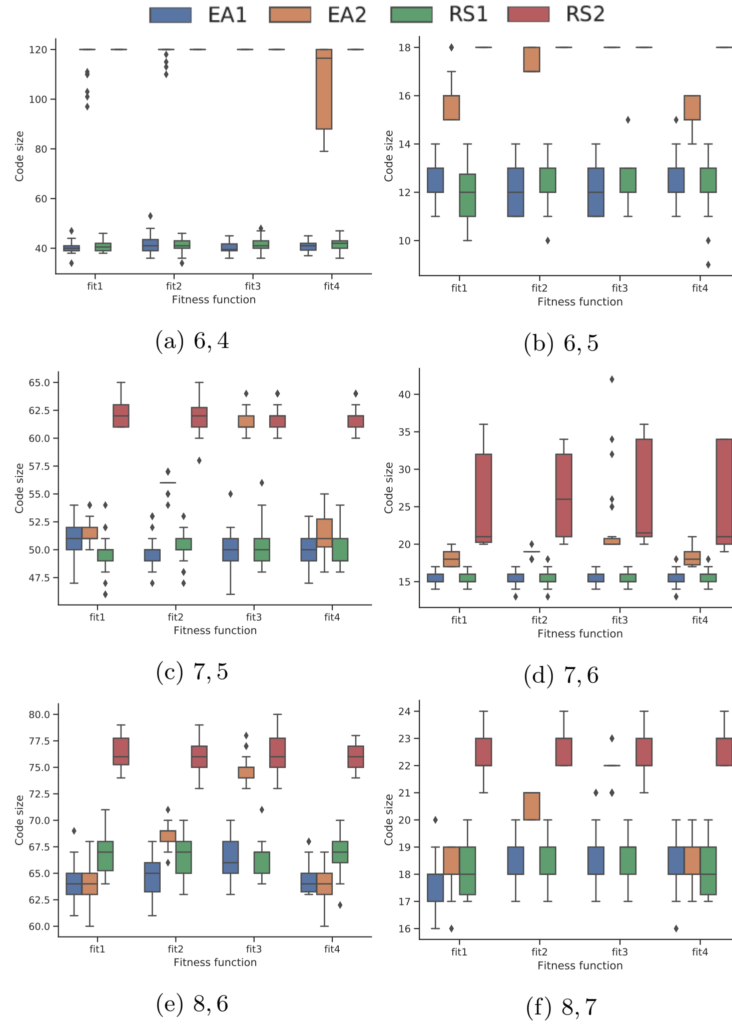
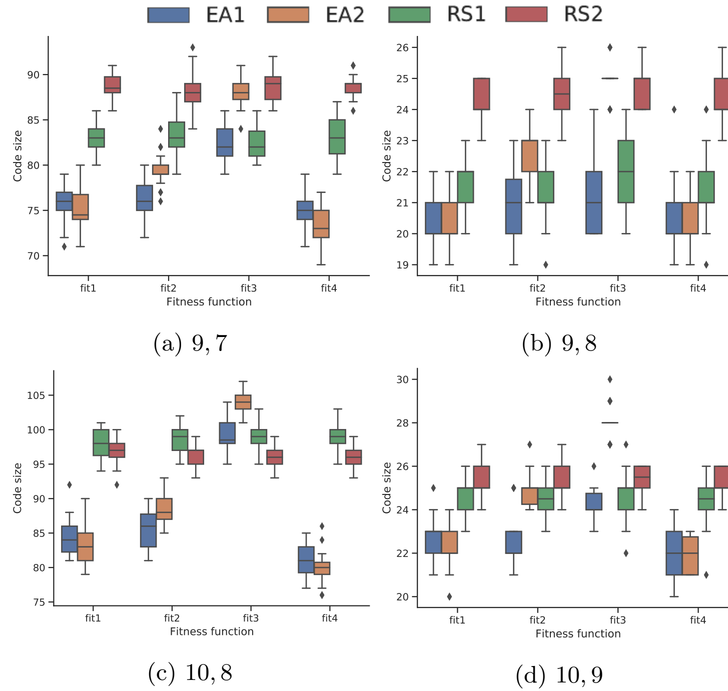


Fig. 1: Largest code size achieved by all methods across the problem instances with $n = 6, 7, 8$ and $d = n - 2, n - 1$.

considerably smaller codes than those using random resets, which instead find almost always an optimal solution. Surprisingly, by comparing the results concerning the update policies, there is no significant difference between the code sizes obtained by EA and RS. A second surprising remark concerns the fitness functions: while we expected fit_3 to be the worst-performing one in Section 3.3, it generally achieved larger code sizes than the other three.

As for fitness functions, the most interesting remark is that the best performing one is also the simplest, namely fit_3, that measures the minimum distance of the new candidate solution from all permutations in the current code. This is a fairly



(a) $9, 7$ (b) $9, 8$

(c) $10, 8$ (d) $10, 9$

Fig. 2: Largest code size achieved by all methods across the problem instances with $n = 9, 10$ and $d = n - 2, n - 1$.

straightforward translation of the property characterizing a permutation code into an objective function to be maximized, and we hypothesized that it could underperform due to its limited range of values. The reason why fit_3 achieves the largest code sizes over all considered instances might reside in the size of the “local” search space, i.e., in the set of all permutations of size n that the EA searches at each stage of the incremental construction. Indeed, we targeted relatively small permutations, where the symmetric group is composed of at most $10! \approx 3 \cdot 10^6$ in the largest considered instance. Using finer-grained fitness functions such as fit_1, fit_2, and fit_4 might have hampered the EA search process, investing many fitness evaluations in optimizing much more information – e.g., the discounted invalid distances in Eq. (4) – than what was needed. It could be interesting to perform experiments on larger instances where the symmetric group S_n is not amenable to exhaustive search to see if this trend continues or if the additional information exploited by the other fitness functions gives an advantage over fit_3.

The relatively small size of the local search space of permutations might also be related to the substantial equivalence of the EA and RS performances. It could be the case that the results are quite similar because it does not make any difference how the local permutation is selected to incrementally expand the

code, given the size of the underlying symmetric group. Thus, in this setting, the update policy seems to be the factor that mainly influences the size of the largest codes found by our incremental approach, independently of the search method. Again, this explanation should be tested against further experiments on larger problem instances. One can see already from the plots in Figure 2 that a difference does arise for $n = 10$, with EA2 under fitness function fit_3 achieving larger codes than those obtained by RS2. It is reasonable to assume that with larger permutation sizes, the evolutionary approach takes over the random search, with an increasing gap between the two methods.

Besides the comparison with random search, the most interesting observation arising from our experiments is that this optimization problem seems to be exceptionally difficult for evolutionary algorithms. Only for the smallest instances of $PA(6,4)$ and $PA(6,5)$ could we obtain optimal solutions concerning the code size (neglecting the outlier found for $PA(7,6)$). In all other cases, the largest code found (either with EA or RS) always lies far from the best lower bounds for $M(n,d)$. This finding could be interpreted in view of the MAX-CLIQUE formulation of the problem [18]. Suppose we have a graph where the nodes are the permutations in S_n, and two nodes are connected by an edge if and only if their Hamming distance is at least d. Then, constructing the largest permutation code $PA(n,d)$ is equivalent to searching the largest clique in such a graph. With the incremental construction, one starts from a single node in the graph and then tries to expand as much as possible the clique(s) to which this node belongs. Our evolutionary algorithm, on the other hand, does not take into account the *topology* of this graph, which involves both the region where the initial permutation is located and its *neighborhood*, i.e., the set of its adjacent nodes. This problem might also be worsened because EAs are population-based methods. Hence, by starting with a set of candidate solutions generated at random, one might waste many fitness evaluations to "move" the population close to the neighborhood of the clique constructed up to that point. One strategy to cope with this issue could be to experiment with smaller population sizes or to integrate the EA with a local search step that also considers the graph representation.

5 Conclusions and Future Work

This paper addresses the optimization problem of constructing permutation codes using EA, which, as far as we know, has not been addressed before. The main question in this domain concerns finding the largest code size for a given permutation length n and a minimum distance d. We have developed an incremental construction approach, starting from a single random permutation chosen at random and then using an EA to iteratively expand the code. We evaluated our method with four fitness functions using two different update policies and in comparison to a baseline random search algorithm. Most importantly, the results of our experiments show that this optimization problem is particularly difficult for evolutionary techniques, with the largest codes found by our EA lying far from the best-known lower bounds in most of the considered problem instances.

Further interesting findings include the fact that the simplest fitness function performed the best and that the update policy seems to be crucial for finding large codes rather than the underlying search method.

In future work, we plan to improve our incremental EA approach by following the directions outlined above: experimenting with larger problem instances and including a local search optimization step. We also envision investigating a concept closely related to equidistant permutation codes, where the Hamming distance between codewords must be exactly equal to d. Equidistant permutation codes are thus a subset of permutation codes, making our iterative procedure applicable. However, since equidistant permutation codes are more rare, we believe this problem to be even harder.

References

1. Banzhaf, W.: The "molecular" traveling salesman. Biological Cybernetics **64**(1), 7–14 (1990)
2. Chu, W., Colbourn, C.J., Dukes, P.: Constructions for permutation codes in powerline communications. Des. Codes Cryptogr. **32**(1-3), 51–64 (2004)
3. Colbourn, C.J., Dinitz, J.H.: Combinatorial designs. In: Rosen, K.H., Michaels, J.G., Gross, J.L., Grossman, J.W., Shier, D.R. (eds.) Handbook of Discrete and Combinatorial Mathematics. CRC Press (1999)
4. Colbourn, C.J., Kløve, T., Ling, A.C.H.: Permutation arrays for powerline communication and mutually orthogonal latin squares. IEEE Trans. Inf. Theory **50**(6), 1289–1291 (2004)
5. Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups, Grundlehren der mathematischen Wissenschaften, vol. 290. Springer (1988)
6. Daemen, J., Rijmen, V.: The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition. Springer (2020)
7. Ferreira, H.C., Vinck, A.H.: Interference cancellation with permutation trellis codes. In: IEEE 52nd Vehicular Technology Conference Fall 2000. vol. 5, pp. 2401–2407. IEEE (2000)
8. Fogel, D.B.: Applying evolutionary programming to selected traveling salesman problems. Cybernetics and systems **24**(1), 27–36 (1993)
9. Goldberg, D.E., Jr., R.L.: Alleles, loci and the traveling salesman problem. In: Grefenstette, J.J. (ed.) Proceedings of the 1st International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 1985. pp. 154–159. Lawrence Erlbaum Associates (1985)
10. Han Vinck, A.: Coded modulation for powerline communications. AEU Int. J. Eletron. Commun. **54**(1), 45–49 (2000)
11. Jiang, A., Mateescu, R., Schwartz, M., Bruck, J.: Rank modulation for flash memories. In: Kschischang, F.R., Yang, E. (eds.) 2008 IEEE International Symposium on Information Theory, ISIT 2008, Toronto, ON, Canada, July 6-11, 2008. pp. 1731–1735. IEEE (2008)
12. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA. pp. 85–103. The IBM Research Symposia Series, Plenum Press, New York (1972)

13. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
14. Knezevic, K., Picek, S., Mariot, L., Jakobovic, D., Leporati, A.: The design of (almost) disjunct matrices by evolutionary algorithms. In: Fagan, D., Martín-Vide, C., O'Neill, M., Vega-Rodríguez, M.A. (eds.) Theory and Practice of Natural Computing - 7th International Conference, TPNC 2018, Dublin, Ireland, December 12-14, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11324, pp. 152–163. Springer (2018)
15. Liu, M., Sim, S.M.: Lightweight MDS generalized circulant matrices. In: Peyrin, T. (ed.) Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9783, pp. 101–120. Springer (2016)
16. Mariot, L., Picek, S., Jakobovic, D., Leporati, A.: Evolutionary algorithms for the design of orthogonal latin squares based on cellular automata. In: Bosman, P.A.N. (ed.) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017. pp. 306–313. ACM (2017)
17. Mariot, L., Picek, S., Jakobovic, D., Leporati, A.: Evolutionary search of binary orthogonal arrays. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, L.D. (eds.) Parallel Problem Solving from Nature - PPSN XV - 15th International Conference, Coimbra, Portugal, September 8-12, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11101, pp. 121–133. Springer (2018)
18. Montemanni, R., Barta, J., Smith, D.H.: Graph colouring and branch and bound approaches for permutation code algorithms. In: Rocha, Á., Correia, A.M.R., Adeli, H., Reis, L.P., Teixeira, M.M. (eds.) New Advances in Information Systems and Technologies - Volume 1 [WorldCIST'16, Recife, Pernambuco, Brazil, March 22-24, 2016]. Advances in Intelligent Systems and Computing, vol. 444, pp. 223–232. Springer (2016)
19. Oliver, I.M., Smith, D.J., Holland, J.R.C.: A study of permutation crossover operators on the traveling salesman problem. In: Grefenstette, J.J. (ed.) Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, USA, July 1987. pp. 224–230. Lawrence Erlbaum Associates (1987)
20. Smith, D.H., Montemanni, R.: A new table of permutation codes. Des. Codes Cryptogr. **63**(2), 241–253 (2012)
21. Smith, D.H., Montemanni, R.: Permutation codes with specified packing radius. Des. Codes Cryptogr. **69**(1), 95–106 (2013)
22. Stinson, D.R.: Combinatorial designs - constructions and analysis. Springer (2004)
23. Syswerda, G., Palmucci, J.: The application of genetic algorithms to resource scheduling. In: Belew, R.K., Booker, L.B. (eds.) Proceedings of the 4th International Conference on Genetic Algorithms, San Diego, CA, USA, July 1991. pp. 502–508. Morgan Kaufmann (1991)
24. De la Torre, D., Colbourn, C., Ling, A.: An application of permutation arrays to block ciphers. Congressus Numerantium pp. 5–8 (2000)
25. Vaudenay, S.: On the need for multipermutations: Cryptanalysis of MD4 and SAFER. In: Preneel, B. (ed.) Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings. Lecture Notes in Computer Science, vol. 1008, pp. 286–297. Springer (1994)