Applied Soft Computing 47 (2016) 21–32

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

A clique-based online algorithm for constructing optical orthogonal codes

Yuan Zhang [a], Mao Peng [a], Shengxiang Yang [a,b,*]

[a] School of Mathematics and Statistics, Nanjing University of Information Science and Technology, Nanjing 210044, China
[b] Centre for Computational Intelligence (CCI), School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, United Kingdom

ARTICLE INFO

Article history:
Received 29 January 2015
Received in revised form 28 March 2016
Accepted 17 May 2016
Available online 25 May 2016

Keywords:
Optical orthogonal codes
Evolutionary algorithm
Maximum clique problem
Online algorithm

ABSTRACT

An optical orthogonal code (OOC) is a family of binary sequences with good auto- and cross-correlation properties. In the literature, various mathematical tools have been used to construct OOCs with specific parameters. But, to find a complete solution for constructing OOCs with an arbitrary setting of parameters is still difficult at the moment. In this paper, a clique-based online algorithm is proposed to construct OOCs of relatively large sizes. In the proposed algorithm, the construction of OOCs is reduced to the maximum clique problem based on specially generated graphs, where vertices represent the codewords of an OOC and edges represent the cross-correlation relationships between codeword pairs. In order to overcome the limitation of computer memory for storing large graphs, part of the graph vertices are supposed to arrive sequentially to be fed into the proposed algorithm, and a specially designed evolutionary algorithm is used to find the maximum clique of the current graph when new vertices arrive. The proposed algorithm does not use parameter-specific techniques and hence can be used for different code weight and correlation constraints. Experiments show that the proposed algorithm outperforms an offline evolutionary algorithm with guided mutation on constructing OOCs.

1. Introduction

Code division multiple access (CDMA) is one of the most attractive multiple-access schemes for optical communication networks because it allows simultaneous users to access the same optical channel asynchronously with no delay or scheduling. In optical CDMA systems, each user is assigned a unique binary signature sequence (or codeword) as its own address. A user's receiver must be able to extract its signature sequence in the presence of other users' signature sequences. Therefore, a set of signature sequences that are distinguishable from time-shifted versions of themselves and for which any two such signature sequences are easily distinguishable from (a possible time-shifted version of) each other are needed. Researchers in the CDMA domain have proposed several coding systems, such as the prime code [28] and optical orthogonal code (OOC) [31].

The mathematical model fo the OOC can be briefly described as follows. Let v, k, λ_a and λ_c be positive integers. A $(v, k, \lambda_a, \lambda_c)$-OOC $\mathcal{C}$ is a family of (0, 1) sequences with length (or order) v and weight k (i.e., the given number of one bits in the sequences) which satisfies the following two properties:

1. The auto-correlation property:

$$\sum_{t=0}^{v-1} x_t x_{t\oplus\tau} \le \lambda_a \tag{1}$$

for any $\mathbf{x} \in \mathcal{C}$ and any integer τ $(0 < \tau < v)$, where "$\oplus$" represents the modulo v addition.

2. The cross-correlation property:

$$\sum_{t=0}^{v-1} x_t y_{t\oplus\tau} \le \lambda_c \tag{2}$$

for any $\mathbf{x} \neq \mathbf{y}$ $(\mathbf{x}, \mathbf{y} \in \mathcal{C})$ and any integer τ $(0 < \tau < v)$.

The numbers λ_a and λ_c are called the auto- and cross-correlation constraints, respectively. A $(v, k, \lambda_a, \lambda_c)$-OOC with $\lambda_a = \lambda_c = \lambda$ is called "symmetric" and can be briefly denoted by (v, k, λ)-OOC. The high weight of codewords facilitates the detection of

* Corresponding author at: Centre for Computational Intelligence (CCI), School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, United Kingdom.
E-mail addresses: zhangyuanppx@163.com (Y. Zhang), mpengsjtu@gmail.com (M. Peng), syang@dmu.ac.uk (S. Yang).

http://dx.doi.org/10.1016/j.asoc.2016.05.024
1568-4946/

desired signals, and the low auto- and cross-correlations reduce the interference from unwanted signals of synchronization in a network. An example of an (11, 4, 2)-OOC is given later in Section 2.1.

Experiments have shown that the setting of OOC's auto-correlation and cross-correlation constraints guarantees a large number of asynchronous users to transmit information efficiently and reliably in a communication network. In short, the auto-correlation of each sequence in the OOC exhibits the thumbtack shape, and the cross-correlation between any two sequences remains low throughout. The lack of a network synchronization requirement enhances the flexibility of OOC-based systems [23,31]. In addition to the applications in optical multiple-access channels, OOCs also find applications in areas such as mobile radio and radar and sonar signal design. For more details, readers are referred to the survey paper by Colbourn et al. [16].

One of the central problems in the OOC research is to determine an OOC's maximum code size, which is the number of codewords in the OOC. The code size of an OOC implies the possible number of users in a communication network. In the literature, various mathematical tools have been used to construct OOCs with specific parameters (usually of small sizes) [1,2,9,11]. But, it is still a long way to go to find a complete solution for constructing OOCs with an arbitrary setting of parameters (i.e., k, λ_a, and λ_c).

In this paper, a clique-based online algorithm is proposed to construct OOCs with relatively large sizes. The proposed algorithm is based on specially generated graphs, where vertices represent the codewords of an OOC that satisfy the auto-correlation constraint and edges represent the cross-correlation relationships between codeword pairs. Based on such specially generated graphs, the construction of OOCs is reduced to the *maximum clique problem* (MCP). In order to overcome the limitation of computer memory for storing large graphs, part of the graph vertices are supposed to arrive sequentially to be fed into the proposed algorithm, and a specially designed evolutionary algorithm (EA) is used to find the maximum clique of the current graph when new vertices arrive. The proposed algorithm is tested to design $(v, 4, 2)$-OOCs. Experimental results show that the cardinalities of constructed OOCs are close to the Johnson bound (to be described in Section 2.2) when v is small. We also use the proposed algorithm to construct $(v, 4, 2, 1)$-OOCs and $(v, 5, 2)$-OOCs, indicating that the proposed algorithm can be used under different settings of parameters.

The rest of this paper is organized as follows. Section 2 describes the background and related work on OOCs, including an example of an OOC, the fundamentals of constructing OOCs and a brief review of search algorithms for constructing OOCs. Our proposed clique-based online algorithm for constructing OOCs is presented in Section 3. Experimental results are reported in Section 4. Finally, Section 5 concludes this paper with some discussions on relevant future work.

2. Background and related work

2.1. Example of an OOC

In order to better understand the concept of OOCs, we give an example here. Let $\mathcal{C}$ be an OOC of length $v = 11$ and weight $k = 4$ with three codewords C_1, C_2, C_3 as follows:

$$\begin{aligned}\mathcal{C} &= \{C_1, C_2, C_3\}\\ &= \{11000010010, 11000101000, 11101000000\}.\end{aligned} \tag{3}$$

In order to calculate the auto-correlation constraint of $\mathcal{C}$, we take $C_1 = 11000010010$ for instance. Its 11 cyclic shifts C_{1i} ($i = 0, 1, \ldots, 10$) are given as follows:

$$\begin{aligned}&C_{1,0} = 11000010010, \quad C_{1,1} = 01100001001,\\ &C_{1,2} = 10110000100, \quad C_{1,3} = 01011000010,\\ &C_{1,4} = 00101100001, \quad C_{1,5} = 10010110000,\\ &C_{1,6} = 01001011000, \quad C_{1,7} = 00100101100,\\ &C_{1,8} = 00010010110, \quad C_{1,9} = 00001001011,\\ &C_{1,10} = 10000100101.\end{aligned} \tag{4}$$

The inner product of any pair from these shifts is no more than 2, which indicates the auto-correlation constraint of C_1 is 2. Similarly, we can check that the auto-correlation constraint of C_2 and C_3 is also 2. Hence, $\mathcal{C}$ has an auto-correlation constraint of 2, i.e., $\lambda_a = 2$.

In order to work out the cross-correlation constraint of $\mathcal{C}$, we first obtain all the cyclic shifts of all codewords in $\mathcal{C}$, i.e., C_{il} ($i \in \{1, 2, 3\}$ and $l \in \{0, 1, \ldots, 10\}$), as shown in Eqs. (4)–(6), respectively.

$$\begin{aligned}&C_{2,0} = 11000101000, \quad C_{2,1} = 01100010100,\\ &C_{2,2} = 00110001010, \quad C_{2,3} = 00011000101,\\ &C_{2,4} = 10001100010, \quad C_{2,5} = 01000110001,\\ &C_{2,6} = 10100011000, \quad C_{2,7} = 01010001100,\\ &C_{2,8} = 00101000110, \quad C_{2,9} = 00010100011,\\ &C_{2,10} = 10001010001.\end{aligned} \tag{5}$$

$$\begin{aligned}&C_{3,0} = 11101000000, \quad C_{3,1} = 01110100000,\\ &C_{3,2} = 00111010000, \quad C_{3,3} = 00011101000,\\ &C_{3,4} = 00001110100, \quad C_{3,5} = 00000111010,\\ &C_{3,6} = 00000011101, \quad C_{3,7} = 10000001110,\\ &C_{3,8} = 01000000111, \quad C_{3,9} = 10100000011,\\ &C_{3,10} = 11010000001.\end{aligned} \tag{6}$$

Then, we calculate the inner product of any pair of shifts from different C_is, i.e., any pair (C_{il}, C_{jm}) ($i, j \in \{1, 2, 3\}$ and $i \neq j$, and $l, m \in \{0, 1, \ldots, 10\}$). It is not hard to check that the maximal inner product is 2. Hence, the cross-correlation constraint of $\mathcal{C}$ is 2, i.e., $\lambda_c = 2$.

In summary, $\mathcal{C}$ is a (11, 4, 2)-OOC with 3 codewords.

2.2. Constructing OOCs: fundamentals

As aforementioned, one of the central problems in the OOC research is to determine the code size of an OOC, i.e., the number of codewords in an OOC. An OOC is called *maximal* if there is no other OOC of a larger size with the same set of parameters. This maximal code size is denoted by $\Phi(v, k, \lambda_a, \lambda_c)$, which can be abbreviated by $\Phi(v, k, \lambda)$ when the OOC is symmetric. It is well-known that the size of a (v, k, λ)-OOC cannot exceed the so-called Johnson bound [15], which is given as follows:

$$J(v, k, \lambda) = \left\lfloor \frac{1}{k} \left\lfloor \frac{v-1}{k-1} \cdots \left\lfloor \frac{v-\lambda}{k-\lambda} \right\rfloor \cdots \right\rfloor \right\rfloor. \tag{7}$$

where $\lfloor X \rfloor$ is the floor function, which returns the maximum integer that is less than or equal to X.

Usually, a $(v, k, \lambda_a, \lambda_c)$-OOC is said to be *optimal* when its size reaches an upper bound deducible from very general considerations which are valid for all possible pairs (v, k) (with λ_a and λ_c

fixed). Thus, an optimal OOC is maximal, but the reverse is not true generally.

In the past years, different researchers have devoted effort to the search for maximal or optimal OOCs, and many good results have been obtained by an extensive use of various mathematical tools, such as the design theory [9], difference family [1], relative difference family [11], and projective geometry [2], etc. For example, the existence problem for an optimal $(v, k, 1)$-OOC has been completely solved for $k = 3$ [9,15]. But, it is still difficult to find a complete solution for larger values of k, λ_a, and λ_c for the moment. For example, only some partial answers have been achieved for the cases of $k = 4$ or 5, $\lambda_a \leq 2$ and $\lambda_c \leq 2$ [2,3,12–14,18]. Further details can be found in the references and therein.

Since it is still a long way to find a complete solution for constructing OOCs with an arbitrary setting of parameters k, λ_a and λ_c, in this paper we investigate the construction of OOCs by using search algorithms. Note that we are not aiming to find maximal OOCs, but to present a general scheme to find OOCs with relatively large sizes.

2.3. Search algorithms for constructing OOCs

The algorithm proposed in this paper is based on graph cliques: all the codewords that satisfy the auto-correlation constraint are viewed as vertices of a graph, and edges are added to the codeword pairs if the cross-correlation constraints are satisfied. For a simple undirected graph $G = (V, E)$ with the vertex set V and edge set E, a subset S of V is a clique of G if every two distinct vertices of S are joined by an edge. Naturally, the maximal OOC construction problem can be transformed to the MCP.

The MCP is one of the well-known problems in graph theory, and is related to many real-world problems, such as in social networks, computer vision, computational biochemistry and bio-informatics. In past decades, different exact algorithms have been proposed to achieve maximum cliques, such as branch-and-bound [26] and branch-and-cut [30]. On the other hand, it has been proved that there is no polynomial-time algorithm for approximating the maximal clique within a factor of $n^{1-\varepsilon}$ for any positive ε unless $P = NP$ [21], where n is the number of vertices in the graph. As a result, it seems to be impossible to have an exact polynomial-time algorithm for solving the MCP. Hence, there has been a lot of researches on using heuristic algorithms for solving the MCP, including local search [8,22], greedy algorithms [20,27], constraint programming [29], simulated annealing [19], and EAs [7,10,33]. For more details, readers are referred to a recent survey by Wu and Hao [32].

Although there are many good algorithms for solving the MCP, clique-based algorithms have not been frequently used to design OOCs. To the best of our knowledge, it seems that the only existing clique-based algorithm is the one proposed by Chu and Colbourn [14]. In [14], a clique-based algorithm was presented to construct $(v, 4, 2)$-OOCs when v is small, and the clique solvers were a reactive local search algorithm from Battiti and Protasi [8] and an exact branch-and-bound algorithm developed by Niskanen and Ostergard [26]. Although the OOCs obtained in [14] are optimal, some parameter-specific techniques were used to accelerate graph generation. As a result, the method can not be generalized to construct OOCs with an arbitrary setting of parameters. More importantly, with the growth of the OOC order, the number of codewords satisfying the auto-correlation condition would become huge.

For example, we have used a heuristic algorithm to test the lower bounds of the graph cardinality for $(v, 4, 2)$-, $(v, 5, 2)$- and $(v, 6, 2)$-OOCs (the details will be given in Section 4.2 later). The results show that the cardinality for $(300, 4, 2)$-OOC is already over one million. In this case, even storing the graph structure in the computer memory is difficult. This may be the reason why authors in [14] chose to construct optimal $(v, 4, 2)$-OOCs for v less than 45.

Apart from clique-based algorithms, researchers have also used backtracking to search difference sets which would lead to OOCs. Recently, Baicheva and Topalova [4,5] developed a parallel backtrack search algorithm to find $(v, 4, 2, 1)$- and $(v, 5, 2, 1)$-OOCs when v is small. Such a parallel algorithm was run over IBM Blue Gene supercomputers, and returned optimal OOCs. However, it is not hard to derive from their papers that the reported backtracking is equivalent to clique search. When v is large, the corresponding graph structure still cannot be stored in supercomputers. Moreover, the backtracking uses parameter-specific techniques to speedup vertex selection. So, it cannot be generalized to construct OOCs with $\lambda_c \neq 1$.

In this paper, we propose a clique-based online algorithm to construct OOCs with relatively large sizes. In our setting, the vertices are supposed to arrive sequentially, instead of being available all at once. This way, we can build a pool to store part of the vertices, which is to overcome the limitation of computer memories. A prespecified substitution technique is used to update candidate solutions. Note that the vertices of OOC graphs are generated without using any parameter-specific techniques, which means our algorithm can be used for an arbitrary setting of code weight and correlation constraints.

3. Proposed clique-based online algorithm

In this section, we present the clique-based online algorithm to find large $(v, k, \lambda_a, \lambda_c)$-OOCs for general setting of k, λ_a and λ_c. Since the construction of maximal OOCs can be reduced to the MCP, we first need a method to generate the needed graph. Then, we will run an online algorithm to find cliques, where the so-called *Repair* and *Evolution* operations will be used frequently. After the target clique is obtained, a required OOC will follow accordingly.

3.1. Representation of OOC Codewords

As mentioned before, a $(v, k, \lambda_a, \lambda_c)$-OOC is a family of binary strings with length v and weight k. For practical reasons, v is often much bigger than k, and this would bring many zeros in each of the codeword string. In this case, a set-theoretic way can be considered to represent codewords.

A $(v, k, \lambda_a, \lambda_c)$-OOC $\mathcal{C}$ can be considered as a family of k-sets of integers modulo v in which each k-set corresponds to a codeword and the integers within each k-set specify the non-zero bits. The correlation properties in this set-theoretic framework are shown as follows:

1. The auto-correlation property:

$$|C_i \cap (C_i \oplus \tau)| \leq \lambda_a \tag{8}$$

for any $C_i \in \mathcal{C}$ and any integer τ $(0 < \tau < v)$, where "$\oplus$" represents the modulo v addition.

2. The cross-correlation property:

$$|C_i \cap (C_j \oplus \tau)| \leq \lambda_c \tag{9}$$

for any C_i, $C_j \in \mathcal{C}$ $(i \neq j)$ and any integer τ $(0 < \tau < v)$.

Throughout this paper, we will use the set vector form $(c_1, c_2, \ldots, c_k)$ to denote a codeword. Let us use the $(11, 4, 2)$-OOC in Eq. (3) as an example. In the set notation, it can be represented as follows:

$$\mathcal{C} = \{\{0, 1, 6, 9\}, \{0, 1, 5, 7\}, \{0, 1, 2, 4\}\}. \tag{10}$$

Table 1
Equivalent codewords of codeword {0, 1, 6, 9}.

Codeword	{0,1,6,9}	{1,2,7,10}	{2,3,8,0}	{3,4,9,1}
Sorted	{0,1,6,9}	{1,2,7,10}	{0,2,3,8}	{1,3,4,9}
Key	196	1660	283	1747
Codeword	{2,4,5,10}	{0,3,5,6}	{6,7,1,4}	{7,8,2,5}
Sorted	{4,5,10,2}	{5,6,0,3}	{1,4,6,7}	{2,5,7,8}
Key	3211	424	1888	3352
Codeword	{8,9,3,6}	{9,10,4,7}	{10,0,5,8}	
Sorted	{3,6,8,9}	{4,7,9,10}	{0,5,8,10}	
Key	4816	6280	703	

For the sake of convenience, a codeword can also be represented by a new variable *Key* as follows:

$$Key = \sum_{i=1}^{k} c_i v^{k-i}, \tag{11}$$

where $c_1 < c_2 < \cdots < c_k$. This way, we can easily get a new codeword out of a block vector $C \in \mathcal{C}$ with a shift $C + t$, and we call C and its shift $C + t$ equivalent. Note that equivalent codewords cannot be included in one OOC, in order to distinguish inequivalent codewords. We define the characteristic of a codeword C by the minimum *Key* of the codewords equivalent to C.

Taking the codeword {0, 1, 6, 9} mentioned above for instance, its equivalent codewords are shown in Table 1. Obviously, the *Key* value 196 of codeword {0, 1, 6, 9} is the smallest among the equivalent codewords. So, we can say the characteristic of codeword {0, 5, 8, 10} is 196 since it is equivalent to codeword {0, 1, 6, 9}.

3.2. Generation of OOC graphs

A graph is an ordered pair $G = (V, E)$ comprising a set V of vertices together with a set E of edges. For the case of an OOC, all the codewords meeting the auto-correlation condition make the vertex set, and edges are added to the codeword pairs if the cross-correlation constraints are satisfied.

In order to accelerate the search of OOC vertices, we can use the following isomorphism technique. If we have got one OOC $\mathcal{C}$, then we can get some isomorphism OOCs by mapping $\mathcal{C}$ to $t \cdot \mathcal{C}$, where v and t are relative prime numbers, and $t \cdot \mathcal{C} = \{tC(\mathrm{mod}\ \ v) | C \in \mathcal{C}\}$. We call $\mathcal{C}$ and $t \cdot \mathcal{C}$ multiplier equivalents. For example, let $\mathcal{C}$ be the (11, 4, 2)-OOC {{0, 1, 6, 9}, {0, 1, 5, 7}, {0, 1, 2, 4}}, after mapping $\mathcal{C}$ to $2 \cdot \mathcal{C}$, we can obtain a new (11, 4, 2)-OOC {{0, 1, 2, 7}, {0, 2, 3, 10}, {0, 2, 4, 8}}. Since the technique is not parameter-specific, it can be applied for any setting of parameters.

In our algorithm, this multiplier trick is used to get codewords that satisfy the auto-correlation constraint efficiently. If we get a $(v, k, \lambda_a, -)$ codeword C, we can generate a family of $(v, k, \lambda_a, -)$ codewords as follows:

$$\{t \cdot C | \quad gcd(t, v) = 1\}. \tag{12}$$

where $gcd(t, v)$ returns the greatest common divisor of t and v. Note that there might be some equivalent codewords generated. Although there might be some equivalent codewords generated, we can easily distinguish them from the representatives.

3.3. Clique repair operation

Throughout the running of our online algorithm, we will repeatedly update the current graph and current cliques, and we also need to derive a clique from any subset of vertices. Here, Marchiori's repair technique [24,25] is used. The main idea is that we first cut some vertices from the graph to get a small clique, and then expand it in a greedy manner until a maximal clique is achieved.

The pseudo-code of the repair operation is shown in Algorithm 1, where *rand*() in line 5 returns a uniform random number in the range [0, 1]. The probability parameter $\alpha \in (0, 1)$ used in the first phase must be small; otherwise, the clique returned would be very small.

Algorithm 1. *Repair*(U, α)

Require: $U \subset V$: a set of vertices,
$\alpha \in (0, 1)$: probability to leave.
Ensure: S: a clique in G.
1: $W := U, S := U$;
2: **while** W is not empty **do**
3: randomly pick $x \in W$;
4: remove x from W;
5: **if** rand()<α **then**
6: remove x from S;
7: **else**
8: remove from S and W all the vertices in S that are not connected to x;
9: **end if**
10: **end while**
11: $W := V \setminus S$;
12: **while** W is not empty **do**
13: randomly pick $x \in W$;
14: remove x from W;
15: **if** x is connected to all vertices in S **then**
16: add x to S;
17: **end if**
18: **end while**
19: **return** S

3.4. Evolution operation for the population of cliques

Our proposed algorithm is a clique-based EA. As usual, the EA maintains a population of candidate cliques (solutions). The current candidate population $pop(t)$ is supposed to have N binary strings $X^i = (x_1^i, \ldots, x_n^i)$, $i = 1, 2, \ldots, N$, where n is the number of graph vertices and each string represents a clique. Thus, $pop(t)$ is an $N \times n$ matrix with elements in {0, 1}. Inspired by the estimation of distribution algorithm (EDA), the EA maintains a probability vector $P = (p_1, \ldots, p_n)$, which is initialized as follows:

$$p_j = \frac{1}{N} \sum_{i=1}^{N} x_j^i. \tag{13}$$

At each step, we calculate the fitness[1] of each binary string in the population, sort the binary strings in the descending order of their fitness, and pick up the fittest M individuals, denoted as $X^1, X^2, \ldots, X^M$. Then, the probability vector P is updated in the same way as in the *population-based incremental learning* (PBIL) algorithm [6] as follows:

$$P = (1 - \lambda)P + \frac{\lambda}{M} \sum_{i=1}^{M} X^i. \tag{14}$$

Then, we apply the *guided mutation operator*, as shown in Algorithm 2, with the updated probability vector P to string X^1 for $N - M$ times, repair the resultant strings and use them to replace strings X^{M+1}, $X^{M+2}, \ldots, X^N$. The strings $X^2, \ldots, X^M$ do not generate offspring, but are only used to update the probability vector P.

[1] The fitness of a solution is defined as the cardinality of the clique represented by the solution.

Algorithm 2. *Mutate*(X, P, β)

Require: $P=(p_1, \ldots, p_n) \in [0, 1]^n$,
$X=(x_1, \ldots, x_n) \in \{0, 1\}^n$, $\beta \in [0, 1]$.
Ensure: $Y=(y_1, \ldots, y_n) \in \{0, 1\}^n$.

```
for j := 1 to n do
  if rand()<β then
    if rand()<p_j then
      y_j := 1;
    else
      y_j := 0;
    end if
  else
    y_j := x_j;
  end if
end for
return Y
```

The so-called guided mutation operator is a technique used to combine the global statistical information and local information to overcome the shortcoming of genetic algorithms and EDAs. For the MCP, the guided mutation operator uses the probability vector $P=(p_1, p_2, \ldots, p_n)$ to mutate an individual $X \in \{0, 1\}^n$. In detail, for each bit of a binary string that represents an individual, a coin is flipped with a head probability β. If the head turns up, the specific bit is set to 1 with the probability p_i; otherwise, it is set to 0. According to the experiments in [33], such a guided mutation operator leads to a better performance than a number of peer algorithms compared in [33].

For the probability vector P, we denote the number of positions satisfying $p_j > 0.0001$ as t_1, and the number of positions satisfying $p_j > 0.8$ as t_2. If $t_2/t_1 \geq 0.8$, then all the strings in $pop(t)$ are thought to be identical, and we return the current best candidate and terminate the evolution operation. The pseudo-code of the whole evolution operation is as shown in Algorithm 3.

Algorithm 3. *Evolution*$(pop, \lambda, \alpha, \beta)$

Require: pop : a set of N vectors X^i $(i=1, \ldots, N)$ in $\{0, 1\}^n$,
$\lambda, \alpha, \beta \in [0, 1]$.
Ensure: X: a vector in $\{0, 1\}^n$.

```
P := (1/N) Σ_{i=1}^{N} X^i;
while true do
  calculate the fitness of X^i ∈ pop (i = 1, ..., N);
  SortPop(X^i, i = 1, ..., N);
  P := (1 − λ)P + (λ/M) Σ_{i=1}^{M} X^i;
  for i := M + 1 to N do
    X := Mutate(X^1, P, β);
    X^i := Repair(X, α);
  end for
  if the stopping condition is met then
    break;
  end if
end while
X := BestVector(pop);
return X;
```

3.5. Online algorithms for the MCP

In this subsection, we present algorithms for the MCP, which are motivated by the *EA with guided mutation* (EA/G) for the MCP [33]. The experimental results reported in [33] show that EA/G performs better than two other EAs, i.e., HGA and MIMIC, which were introduced in [17,25] respectively, for solving the MCP.

Since the OOC graphs are often too big, our clique-based algorithms are given under an *online* setting: the vertices of the graph are supposed to be added one by one sequentially to be fed into the algorithm, instead of having all the vertices available from the start of running the algorithm. Hence, the algorithm is called an *online* algorithm. Since the whole input is unknown in advance, the online algorithm needs to make decisions that may later turn out not to be optimal. However, it does show efficient performance in real world.

A natural thinking is to re-calculate the clique every time a new vertex arrives, which would take a lot of time. Here, we use a different strategy which stores a pool of candidates that are potential maximum cliques, and then uses the coming vertex to update them. In the following, we develop two algorithms based on such a strategy: one is called EA with Substitution (EAS) for large graphs and another is called EA with Addition (EAA) for medium-sized graphs. For both algorithms, the number of maximal allowed candidates in the pool is set to *MaxCache*. We use a matrix to store such a pool, each row of which is denoted by clq^i $(i=1, \ldots, MaxCache)$.

For very large graphs, it is difficult to store the whole graph in the memory at a time. Let *MaxPoint* be the maximum cardinality of a graph we can deal with, and denote these *MaxPoint* points as the static part S of the graph, and the rest as the online part T of the graph. In EAS, we first find a clique of S with a standard EA, such as EA/G. Then, we choose a vertex u from S, replace it with a randomly chosen vertex v from T, and update the candidate cliques in the pool. In case that v might not be in any of the candidate cliques, we manually create a candidate clique out of v. Since the maximum allowed candidates is quite limited (no more than 100), no hash technique is adopted. The algorithm terminates after a given number of substitutions are performed. The pseudo-code of EAS is shown in Algorithm 4.

Algorithm 4. *EAS*(S, T)

Require: S: static part of graph vertices,
T: online part of graph vertices.
Ensure: x: a clique of the graph.

```
use EA/G to find a clique clq^1 of the graph induced by S;
num := 1;
x := clq^1;
while num < MaxCache do
  find the vertex u ∈ S which appears in the least number of
  current candidate cliques;
  S := S − {u};
  randomly choose a vertex v ∈ T;
  S := S ∪ {v};
  for i := 1 to num do
    mutate clq^i N times to generate a population pop;
    repair the binary strings of pop;
    clq^i := Evolution(pop, λ, α, β);
  end for
  if v is not in any clq^i and num < MaxCache then
    num := num + 1;
    clq^num := {v};
  end if
  replace v in T with u;
  denote local_clq as the best of current num cliques;
  if local_clq is better than x then
    x := local_clq;
  end if
  if the stopping condition is met then
    break;
  end if
end while
return x
```

EAA applies for graphs with a medium size, which can be viewed as a semi-online algorithm. This means that part of the vertices arrive at the same time (the static part S) and the others arrive one by one (the online part T). Since the graph cardinality is medium, we can store the whole graph in the memory. Compared with the algorithm EAS, instead of substitution, we can choose a random vertex from T and add it to S, and then update the current candidates in the pool.

3.6. Clique-based online algorithm for constructing OOCs

The problem of constructing optimal OOCs can be reduced to the MCP as follows. We first establish a graph in which vertices are

codewords that satisfy the auto-correlation property, and edges are added to the pairs of vertices (codewords) that satisfy the cross-correlation property. Then, the maximum OOC can be represented as the maximum clique in the graph.

For the problem of constructing an optimal $(v, k, \lambda_a, \lambda_c)$-OOC, the number of vertices in the corresponding graph is usually quite large. In this case, it is difficult for us even to store the graph in memory. Hence, classical MCP algorithms, e.g., EA/G, may fail to give a meaningful result. In this section, we model the OOC problem as an online version of the MCP with limited memory.

The vertices of the graph are supposed to arrive in a random order, which can be viewed as sampling from the codewords that satisfy the auto-correlation property. Since the computer memory is limited, we restrict the maximum graph cardinality to *MaxPoint*, and maintain a set of local maximal cliques with the maximum size of the set being *MaxCache*. If the current graph cardinality is exactly *MaxPoint* and a new vertex arrives, we must replace one existing vertex with the coming vertex, and then update the set of local optima cliques, like what we do in the EAS algorithm described in Section 3.5.

The whole procedure of the proposed online algorithm for constructing OOCs can be summarized as follows:

Phase 1: Generating a set of non-equivalent codewords

1. Set up an empty codeword set $\mathcal{C}$ with size *MaxPoint*;
2. Generate a binary sequence c with length v and weight k randomly;
3. Convert c into the set-theoretic form and compute its auto-correlation value;
4. If c satisfies the auto-correlation constraint, compute all its multiplier equivalents; otherwise, return to Step 2;
5. Compute the characteristics of the new codewords. If a new codeword is not equivalent to the codewords in $\mathcal{C}$, add it to $\mathcal{C}$; otherwise, discard it;
6. If the size of $\mathcal{C}$ reaches *MaxPoint* (i.e., the set is full), go to Phase 2; otherwise, return to Step 2.

Phase 2: Building the graph

7. Denote each codeword in the above set $\mathcal{C}$ as a vertex of the graph G;
8. For each pair of codewords, add an edge to G if the cross-correlation constraint is satisfied;
9. Generate the $MaxPoint \times MaxPoint$ adjacency matrix of G accordingly.

Phase 3: Finding the maximal clique

10. Denote S as the set of all the *MaxPoint* vertices of the OOC graph, and T as the set of all other vertices that satisfy the auto-correlation constraint;
11. Find the maximal clique by using the EAS algorithm;

Phase 4: Rewriting the obtained clique as an OOC

12. Expand the characteristics of clique vertices to get the OOC codewords.

In Phase 1, we aim to find all the codewords that satisfy the auto-correlation constraint and put them together to make a codeword set $\mathcal{C}$. Then, we convert $\mathcal{C}$ into a graph (Phase 2). In Phase 3, we use the EAS algorithm to get a clique: we continue to sample from unchosen graph vertices, replace an existing vertex and update local maximal cliques, until the stopping condition is satisfied (e.g., the maximal number of vertex substitutions is reached). Finally, we rewrite the obtained clique as codewords (Phase 4).

4. Experimental study

In this section, we carry out experiments to verify the performance of proposed algorithms. Two groups of experiments are conducted. The first group of experiments investigates the performance of the proposed EAS and EAA algorithms for solving the MCP, and the second group investigates the performance of the proposed online algorithm for constructing OOCs. All algorithms were implemented in C and run on an Intel (3.1 GHz) PC with 4GB RAM and Windows 7 operating system.

4.1. Experimental results on solving the MCP

Since the kernel part of online algorithm for constructing OOCs proposed in this paper is based on EAA and EAS for solving the MCP, this first group of experiments investigates the performance of EAA and EAS, in comparison with EA/G [33], for solving the MCP based on the famous set of 37 DIMACS graphs. The reason we choose EA/G is that our algorithms are built upon EA/G, just with an extra operator of Addition or Substitution under online settings. In this case, we test whether this operator could produce a significant improvement over EA/G for standard benchmarks. Some related parameters of our algorithms were set as follows: $\alpha = 0.001$ in the *Repair* operation, $\beta = 0.9$ and $\lambda = 0.7$ in the *Evolution* operation, which are the same as used in [33].

Both EAA and EAS can be divided into two phases: the first phase is to compute a maximal clique using EA/G over 80% of all the vertices (i.e., the static set S), and the second phase include the rest 20% of vertices (i.e., the online set T). The difference in the second phase between EAS and EAA is that EAA uses an addition operator, while EAS uses a substitution operator, to include the rest 20% of vertices. In both phases, M was set to be $N/2$, while $N = 10$ for EA/G in the first phase and $N = 2$ for the *Evolution* operation in the second phase.

For each graph, EAA and EAS were run independently for 30 times. The experimental results are shown in Table 2. In Table 2, the second and third columns are the average clique size and the best clique size found by EA/G cited from [33]. The fourth, fifth, sixth and eighth columns are the average maximum clique size found for the 80% of the vertices by EA/G, the average maximum clique size found by EAA, the overall maximum clique size, and the average execution time in seconds, respectively. We applied t-test to see whether there is any statistical difference between the results achieved before the Addition operation and the results achieved after the Addition operation. The p-values are placed in the seventh column.

In Table 2, the 9th to 11th and 13th columns are the average maximum clique size found for the 80% of the vertices by EA/G, the average maximum clique size found by EAS, the overall maximum clique size and the average execution time in seconds, respectively. Here, for each of the tested graphs, we bold-face the best clique size if EAA or EAS returned a better clique than EA/G. We also apply t-test to see statistical differences between the results achieved before the substitution operation and the results achieved after the substitution operation. For each run of EAS, the substitution operator was called n times, where n is the number of vertices in the graph. The p-values are placed in the 12th column.

The results show that the average size of the cliques obtained by both EAA and EAS is similar to that of the cliques obtained by EA/G. Both EAA and EAS returned one better clique than EA/G. EAA returns the same best clique size as EA/G does for 22 out of the 37 graphs, while EAS returns the same best clique size as EA/G does for 26 out of the 37 graphs. Both EAA and EAS achieve a clique with the best size only one smaller than that achieved by EA/G on 6 graphs, respectively.

It can be seen that we did not obtain better results than EA/G over most of the DIMACS graphs, especially for dense graphs. The main reason is that EA/G requires full information of the input graph and performs a more thorough search (the *Repair* operator was called 2000 times), while EAA and EAS only perform a limited

Table 2
Comparison between EA/G and online algorithms EAA and EAS for the MCP.

Graph	EA/G		EAA					EAS				
	Avg	Best	Avg1	Avg2	Best	*p*-Value	Time	Avg1	Avg2	Best	*p*-Value	Time
brock200_2	12.0	12	10.2	10.7	12	0.0361	0.5	10.1	11.4	12	<0.0001	0.9
brock200_4	16.5	17	15.0	15.2	16	0.0165	0.5	14.9	15.7	17	<0.0001	1.0
brock400_2	24.7	25	23.7	23.7	24	1.0000	1.1	23.8	23.9	24	0.2201	4.9
brock400_4	25.1	33	23.0	23.6	33	0.0957	1.1	23.0	24.1	33	0.0014	4.9
brock800_2	20.1	21	19.1	19.2	21	0.4469	3.3	19.4	19.7	20	0.0284	26.5
brock800_4	19.9	21	19.0	19.1	21	0.5359	3.3	19.0	19.7	21	<0.0001	26.5
C125.9	34.0	34	30.0	31.9	33	<0.0001	0.4	30.0	33.8	34	<0.0001	0.6
C250.9	44.0	44	41.0	41.9	44	<0.0001	0.8	41.0	43.1	44	<0.0001	1.8
C500.9	55.2	56	51.8	52.4	54	0.0039	1.8	51.6	53.0	55	<0.0001	8.7
C1000.9	64.4	67	61.8	62.1	64	0.3045	5.8	61.9	62.6	64	0.0110	51.8
C2000.5	14.9	16	14.6	15.0	16	0.0019	24.4	14.4	15.0	16	<0.0001	146.2
C2000.9	70.9	72	68.2	69.6	**73**	0.0004	31.0	68.1	69.4	71	0.0001	318.0
C4000.5	16.1	17	15.3	16.0	17	<0.0001	137.0	15.5	16.1	17	<0.0001	389.3
DSJC500.5	13.0	13	11.8	12.1	13	0.0098	1.3	12.0	12.6	13	<0.0001	7.1
DSJC1000.5	14.5	15	13.5	13.7	14	0.2264	4.6	13.2	14.0	15	<0.0001	33.4
gen200_p0.9_44	44.0	44	38.0	40.1	42	<0.0001	0.6	38.0	41.3	44	<0.0001	1.2
gen200_p0.9_55	55.0	55	45.0	54.0	55	<0.0001	0.7	45.0	55.0	55	<0.0001	1.2
gen400_p0.9_55	51.8	55	46.8	48.0	50	<0.0001	1.3	46.8	50.0	52	<0.0001	5.4
gen400_p0.9_65	65.0	65	48.9	49.8	53	<0.0001	1.3	48.9	53.9	64	<0.0001	5.3
gen400_p0.9_75	75.0	75	51.2	58.8	75	0.0013	1.3	50.8	72.2	75	<0.0001	5.4
hamming8-4	16.0	16	13.9	16.0	16	<0.0001	0.6	13.9	16.0	16	<0.0001	1.5
hamming10-4	39.8	40	34.3	38.5	40	<0.0001	5.9	34.4	39.4	40	<0.0001	61.8
keller4	11.0	11	11.0	11.0	11	–	0.4	11.0	11.0	11	–	0.7
keller5	26.9	27	26.7	26.8	27	0.5925	3.2	26.4	27.0	27	0.0036	29.2
keller6	53.4	56	52.9	53.3	55	0.2154	129.5	53.4	53.6	**57**	0.5183	558.5
MANN_a27	126.0	126	102.7	125.0	126	<0.0001	2.3	102.8	124.9	125	<0.0001	5.1
MANN_a45	343.7	345	275.2	337.7	342	<0.0001	16.9	275.6	338.6	340	<0.0001	79.4
MANN_a81	1097.2	1098	870.9	929.8	938	<0.0001	341.9	870.7	1087.3	1089	<0.0001	2368.0
p_hat300-1	8.0	8	7.4	7.7	8	0.0086	0.6	7.4	8.0	8	<0.0001	1.4
p_hat300-2	25.0	25	22.0	24.3	25	<0.0001	0.8	22.0	25.0	25	<0.0001	2.6
p_hat300-3	36.0	36	30.8	33.0	34	<0.0001	0.8	30.8	35.6	36	<0.0001	2.8
p_hat700-1	11.0	11	9.5	9.8	11	0.0621	1.9	9.5	10.8	11	<0.0001	8.6
p_hat700-2	44.0	44	37.0	43.2	44	<0.0001	2.7	37.0	43.9	44	<0.0001	19.9
p_hat700-3	62.0	62	55.0	61.1	62	<0.0001	3.0	55.0	61.9	62	<0.0001	26.9
p_hat1500-1	11.1	12	10.3	10.8	12	0.0001	9.1	10.6	11.1	12	0.0001	44.7
p_hat1500-2	65.0	65	59.0	64.5	65	<0.0001	16.3	59.0	64.9	65	<0.0001	103.9
p_hat1500-3	93.7	94	84.6	91.3	93	<0.0001	16.5	84.9	93.2	94	<0.0001	160.2

search (two individuals would converge soon) over each updated graph. Even so, the difference among the results from EA/G, EAA and EAS is quite small. It should also be pointed out that the *t*-test shows a promising result that both of Addition and Substitution operators improve local optimal cliques significantly.

In the following subsections, we will first show the difficulty of searching OOCs by computer algorithms in Section 4.2 and then investigate the performance of our online algorithm to construct OOCs in Section 4.3. Compared with DIMACS graphs (where the graph size is less than 5000), OOC graphs have much bigger sizes (often bigger than 10,000) and sparser structures. Hence, we just test EAS to search for OOCs.

4.2. Experimental results on generating OOC graphs

To show the difficulty of searching OOCs by computer algorithms, we present the graph cardinalities for $(v, 4, 2)$-, $(v, 5, 2)$- and $(v, 6, 2)$-OOCs. Since no mathematical formula can be found, we use computer algorithm to get a lower bound. As introduced in the previous section, an isomorphism technique can be used to accelerate this procedure. Firstly, we iteratively generate a codeword that satisfies the code length and code weight conditions at random until we get a codeword that satisfies the OOC autocorrelation constraint. Then, we use the isomorphism mapping to get more OOC codewords. The procedure continues until we have performed 100,000 random searches. The results are presented in Fig. 1 and Table 3.

It can be seen that the graph size increases quickly with the growth of the OOC code length. Taking the (300, 4, 2)-OOC for instance, the corresponding graph size is already over one million, and just storing such a graph in memory is difficult. In this case, traditional off-line algorithm may fail to obtain good results. In order to overcome the limitation of computer memory for

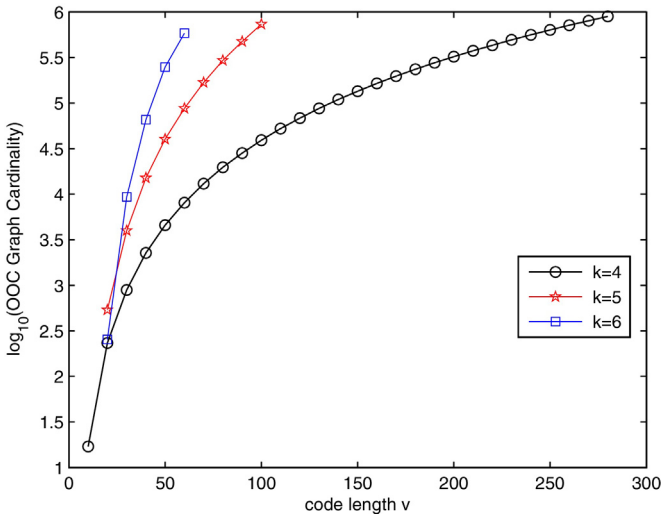

Fig. 1. Fast increasing of the graph cardinality.

Table 3
Lower bound of $(v, k, 2)$-OOC graphs.

(v, k)	Cardinality	(v, k)	Cardinality	(v, k)	Cardinality	(v, k)	Cardinality	(v, k)	Cardinality
(10, 4)	17	(100, 4)	39,151	(190, 4)	276,686	(280, 4)	894,813	(90, 5)	474,370
(20, 4)	233	(110, 4)	52,411	(200, 4)	323,251	(290, 4)	>1,000,000	(100, 5)	735,269
(30, 4)	889	(120, 4)	68,328	(210, 4)	374,705	(20, 5)	538	(110, 5)	>1,000,000
(40, 4)	2263	(130, 4)	87,297	(220, 4)	431,418	(30, 5)	3984	(20, 6)	255
(50, 4)	4577	(140, 4)	109,404	(230, 4)	493,627	(40, 5)	15,118	(30, 6)	9342
(60, 4)	8075	(150, 4)	134,917	(240, 4)	561,385	(50, 5)	40,120	(40, 6)	65,564
(70, 4)	13,057	(160, 4)	164,121	(250, 4)	635,290	(60, 5)	87,258	(50, 6)	248,510
(80, 4)	19,723	(170, 4)	197,447	(260, 4)	715,174	(70, 5)	168,177	(60, 6)	586,090
(90, 4)	28,303	(180, 4)	234,773	(270, 4)	801,532	(80, 5)	293,850	(70, 6)	>1,000,000

Table 4
Results on constructing $(v, 4, 2)$-OOCs for $10 \leq v \leq 64$.

v	Cardinality	Bound	Existing	EA/G			EAS			
				Avg	Best	Time	Avg	Best	Time	p-Value
10	17	3	Yes	3.0	3	0.2	2.6	3	0.2	0.0001
11	25	3	Yes	3.0	3	0.2	3.0	3	0.2	–
12	34	4	No	3.0	3	0.2	3.0	3	0.2	–
13	49	5	No	4.0	4	0.3	4.0	4	0.3	–
14	64	6	Yes	5.9	6	0.3	5.5	6	0.3	0.0033
15	82	7	Yes	6.8	7	0.4	6.2	7	0.3	<0.0001
16	106	8	Yes	8.0	8	0.5	7.7	8	0.4	0.0213
17	132	9	Yes	9.0	9	0.6	9.0	9	0.5	–
18	156	11	Yes	10.2	11	0.6	10.3	11	0.5	0.7558
19	195	12	Yes	11.7	12	0.8	11.2	12	0.8	0.0002
20	233	14	Yes	13.2	14	0.9	13.0	14	0.8	0.0815
21	270	15	Yes	14.1	15	1.0	14.1	15	0.9	0.6921
22	320	17	Yes	16.2	17	1.1	16.1	17	1.1	0.7212
23	374	18	Yes	17.7	18	1.5	17.5	18	1.4	0.1001
24	424	21	No	19.2	20	1.5	19.1	20	1.5	0.2320
25	495	22	Yes	21.0	21	1.9	21.0	21	2.1	0.3219
26	560	25	Yes	22.9	23	2.0	23.1	**24**	2.3	0.0250
27	630	26	Yes	24.5	25	2.4	24.5	25	2.9	1.0000
28	716	29	Yes	26.4	27	2.6	26.4	27	3.4	1.0000
29	805	30	Yes	28.4	29	3.3	28.8	**30**	4.2	0.0037
30	889	33	Yes	30.4	31	3.2	30.9	**32**	4.5	0.0001
31	1000	35	Yes	32.5	33	4.1	33.0	**34**	6.1	0.0002
32	1106	38	Yes	34.7	36	4.5	35.4	36	6.8	<0.0001
33	1215	40	Yes	37.0	38	4.9	37.6	38	8.0	<0.0001
34	1344	44	Yes	39.2	40	5.5	40.1	**41**	9.6	<0.0001
35	1480	45	Yes	41.7	43	6.7	42.7	**44**	11.5	<0.0001
36	1606	49	Yes	44.1	45	7.0	45.1	**46**	12.8	<0.0001
37	1767	51	Yes	46.3	47	8.5	48.0	**49**	15.6	<0.0001
38	1920	55	Yes	49.2	50	9.0	50.6	**51**	18.2	<0.0001
39	2079	57	Yes	51.9	53	11.2	53.3	**55**	21.3	<0.0001
40	2263	61	Yes	54.5	55	13.6	56.4	**58**	24.4	<0.0001
41	2450	63	Yes	57.4	59	15.7	59.5	**60**	28.5	<0.0001
42	2628	68	Yes	60.3	62	17.0	62.7	**63**	31.8	<0.0001
43	2849	70	Yes	63.4	64	20.9	65.8	**67**	41.3	<0.0001
44	3060	75	Yes	66.4	67	22.8	69.0	**70**	47.6	<0.0001
45	3277	77	??	69.2	70	26.4	71.9	**73**	54.9	<0.0001
46	3520	82	Yes	72.7	74	29.1	75.5	**77**	62.8	<0.0001
47	3772	84	??	75.9	77	34.1	79.1	**80**	71.2	<0.0001
48	4012	90	No	79.3	81	36.8	82.4	**84**	79.7	<0.0001
49	4300	92	Yes	82.4	84	42.6	86.0	**87**	91.3	<0.0001
50	4577	98	Yes	86.2	88	45.6	89.7	**92**	98.3	<0.0001
51	4860	100	Yes	89.9	91	51.2	93.6	**95**	111.3	<0.0001
52	5176	106	Yes	93.3	95	56.4	97.3	**98**	126.1	<0.0001
53	5499	108	??	97.1	98	64.2	101.1	**102**	138.0	<0.0001
54	5808	114	Yes	100.6	102	67.7	105.0	**106**	151.9	<0.0001
55	6175	117	??	104.5	105	76.9	109.1	**111**	168.6	<0.0001
56	6526	123	Yes	108.7	110	84.5	113.2	**114**	181.9	<0.0001
57	6885	126	Yes	112.7	114	92.7	117.8	**119**	197.2	<0.0001
58	7280	133	Yes	116.6	118	100.9	122.1	**124**	213.2	<0.0001
59	7685	135	??	121.0	122	113.7	126.3	**128**	238.1	<0.0001
60	8075	142	??	124.9	126	121.7	130.7	**133**	254.8	<0.0001
61	8525	145	Yes	129.5	130	137.2	135.2	**137**	274.7	<0.0001
62	8960	152	Yes	133.8	136	146.3	139.7	**141**	295.9	<0.0001
63	9405	155	Yes	138.2	139	163.4	144.5	**147**	331.1	<0.0001
64	9882	162	Yes	142.8	144	182.0	149.0	**150**	320.2	<0.0001

Table 5
Results on constructing $(v, 4, 2)$-OOCs for bigger v.

v	Cardinality	Bound	Existing	EA/G			EAS			
				Avg	Best	Time	Avg	Best	Time	p-Value
70	13,057	195	Yes	169.0	170	189.0	179.3	**181**	689.0	<0.0001
80	19,723	256	Yes	216.3	220	223.8	233.3	**236**	2722.5	<0.0001
90	28,303	326	Yes	269.8	274	258.1	293.1	**294**	4631.9	<0.0001
100	39,151	404	Yes	327.3	330	286.6	357.4	**359**	6295.3	<0.0001
110	52,411	490	Yes	392.4	394	328.1	428.2	**429**	7673.7	<0.0001
120	68,328	585	No	472.8	480	358.8	502.8	**504**	7967.5	<0.0001
130	87,297	688	Yes	569.3	577	391.3	588.5	**594**	8976.4	<0.0001
140	109,404	799	Yes	663.5	666	419.4	679.0	**680**	9389.4	<0.0001
150	134,917	918	Yes	757.3	759	452.3	775.5	**782**	9654.4	<0.0001
160	164,121	1046	Yes	840.8	862	479.1	865.3	**883**	9934.2	<0.0001
170	197,447	1183	Yes	961.3	973	521.9	981.0	**989**	9989.8	<0.0001
180	234,773	1327	Yes	1009.0	1033	546.5	1041.0	**1062**	10,204.1	<0.0001
190	276,686	1480	Yes	1177.5	1188	593.5	1204.5	**1218**	10,143.6	<0.0001
200	323,251	1641	Yes	1237.8	1250	616.3	1270.3	**1277**	9876.0	<0.0001
210	374,705	1811	Yes	1351.0	1361	662.1	1390.5	**1396**	10,069.0	<0.0001
220	431,418	1989	Yes	1421.3	1458	694.6	1472.8	**1506**	9846.9	<0.0001
230	493,627	2175	Yes	1630.3	1684	761.7	1673.0	**1720**	9667.3	<0.0001
240	561,385	2370	No	1538.3	1556	743.8	1622.3	**1647**	8717.4	<0.0001
250	635,290	2573	Yes	1825.3	1860	832.3	1882.3	**1904**	9776.8	<0.0001
260	715,174	2784	Yes	1845.5	1930	853.9	1929.3	**2006**	9187.4	<0.0001
270	801,532	3003	Yes	1962.8	1990	903.4	2044.0	**2061**	9303.7	<0.0001
280	894,813	3231	Yes	1925.5	2005	905.8	2042.8	**2110**	9392.4	<0.0001
290	>1,000,000	3468	Yes	2293.5	2319	1025.2	2382.0	**2404**	8656.9	<0.0001
300	>1,000,000	3712	??	2108.7	2211	1041.9	2255.7	**2347**	7937.8	<0.0001

storing large graphs, part of the graph vertices are supposed to arrive sequentially to be fed into the algorithm, and an update operation is performed to improve current candidate cliques.

4.3. Experimental results on constructing OOCs

In this group of experiments, we investigate our proposed online algorithm based on EAS to get $(v, k, \lambda_a, \lambda_c)$-OOCs with some specific parameters. Parameters α, β and λ were set to be the same as used in the first group of experiments. In the first phase of EAS, the population size N was set to be 10 for EA/G, $M = N/2 = 5$, and the *Repair* operation within EA/G was run 20,000 times, as we did for DIMACS graphs. In the second phase of EAS, the population size N was set to 2 for the *Evolution* operation, $M = N/2 = 1$, and the algorithm was terminated after 2,000 vertex substitutions. The number of candidate cliques, *MaxCache*, was set to be 100, and the parameter *MaxPoint* was set to 10,000.

Here, we give an explanation for above settings. Since our computers just have 4GB RAM, and the graph is stored as adjacency matrices, then the graph size should be less than 100,000. But, if the graph size is too small, e.g., 5000, the optimal OOC size will be small, and we cannot observe significant differences. Therefore, we set *MaxPoint* to be 10,000.

During the substitution phase of EAS, the population size N was set to 2, and the number of candidate cliques was set to 100. These parameters were set to be small for the reason of running times. Taking the (300, 4, 2)-OOC for example, even under this setting of small parameters, it still takes about three hours to perform a single run.

For decades, different researchers have devoted effort to the research of OOCs, many good results have been obtained by an extensive use of various mathematical tools. Most of these results work for code weight $k = 4$ or $k = 5$, and correlation constraint no more than 2. For bigger code weight or correlation constraint, hardly any theoretical result is known. Therefore the proposed algorithm is tested to design $(v, 4, 2)$-, $(v, 5, 2)$- and $(v, 4, 2, 1)$-OOCs, where the existence results are rich, and also indicating that the proposed algorithm can be used under different settings of code weights and correlation constraints.

4.3.1. Constructing $(v, 4, 2)$-OOCs

Since the construction on $(v, 4, 2)$-OOCs with small orders was thoroughly discussed in [14,18], we first test our algorithm EAS for $(v, 4, 2)$-OOCs with $10 \le v \le 64$ (with the graph size less than 10,000), each with 30 runs. As pointed in Section 2, Chu's branch-and-bound algorithm [14] searches for optimal solutions and cannot be generalized to an arbitrary setting of parameters, we compare EAS with its source algorithm EA/G.

The results are shown in Table 4, where the first four columns are the code length (v), correspondent graph cardinality (*Cardinality*), Johnson bound (*Bound*), and existing result cited from Feng [18] (Existing) (where "??" means that there is no existing result for the corresponding case), respectively. The following six columns are the average code size (*Avg*), best code size (*Best*), and average running time (*Time*) returned by EA/G and EAS, respectively. The last column shows the p-value of applying the t-test to compare EAS over EA/G. In the table, for each setting of v, if EAS significantly outperforms EA/G, its best code size is bold-faced.

From Table 4, it is not hard to see that EAS has a better result than EA/G for all the cases when $v \le 64$. Especially, when $v \ge 32$, the difference is significant ($p < 0.0001$). For the case of $v > 64$, the corresponding OOC graph has more than 10,000 vertices, and hence traditional off-line algorithm, such as EA/G, may fail to give a result due to the computer memory limitation in our settings.

Here, we also test EA/G for randomly generated OOC graphs when $v > 64$ (with the graph size 10,000), while EAS would do another 2000 vertex substitutions. Both EA/G and EAS were run 30 times for each code length. The results are listed in Table 5, where for each setting of v if EAS significantly outperforms EA/G, its best code size is bold-faced. It can been seen that the substitution operation improves current solutions significantly, indicating that EAS outperforms EA/G when constructing $(v, 4, 2)$-OOCs.

Table 6
Results on constructing $(v, 5, 2)$-OOCs for $10 \le v \le 100$.

v	Bound	EAS	v	Bound	EAS	v	Bound	EAS	v	Bound	EAS	v	Bound	EAS
10	0	0	29	12	9	48	35	26	67	69	49	86	119	76
11	1	0	30	13	10	49	36	27	68	73	49	87	120	78
12	1	1	31	13	11	50	39	28	69	74	51	88	121	81
13	1	1	32	15	12	51	40	29	70	75	52	89	127	81
14	2	1	33	16	12	52	40	30	71	80	54	90	129	83
15	2	2	34	16	13	53	44	31	72	81	55	91	130	84
16	3	2	35	18	14	54	45	32	73	82	56	92	136	88
17	4	4	36	19	15	55	45	34	74	87	58	93	138	88
18	4	3	37	19	15	56	49	35	75	88	60	94	139	90
19	4	4	38	22	17	57	50	36	76	90	61	95	145	92
20	5	4	39	22	18	58	51	37	77	95	64	96	147	93
21	6	5	40	23	18	59	55	38	78	96	63	97	148	94
22	6	5	41	26	19	60	56	39	79	97	65	98	155	97
23	7	6	42	26	20	61	57	40	80	102	67	99	156	100
24	8	6	43	27	21	62	61	43	81	104	69	100	158	100
25	8	7	44	30	22	63	62	43	82	105	71			
26	10	7	45	30	23	64	63	44	83	110	72			
27	10	8	46	31	24	65	67	46	84	112	73			
28	10	9	47	34	25	66	68	47	85	113	75			

4.3.2. *Constructing* $(v, 5, 2)$-*OOCs*

For the case of OOCs with a bigger weight, there is hardly any result on the existence of maximum codes [3]. However, we can still obtain some computing results in Table 6 for $v \le 100$.

From Table 6, it can be seen that EAS performs better for $(v, 4, 2)$ than for $(v, 5, 2)$. This is because, with the growth of the values of v and k, the corresponding graph may become huge and sparse, and hence, the searching for a feasible clique is more difficult in nature.

It should be pointed out that the obtained best code sizes for $(300, 4, 2)$-OOC and $(100, 5, 2)$-OOC both reach 63% of their Johnson bounds, respectively. Fig. 2(a) and (b) show the percentage of the obtained best code size to the corresponding Johnson bound and the percentage of the number of stored nodes to the number of nodes in the OOC graph (i.e., the cardinality of the OOC graph) against the code length on $(v, 4, 2)$-OOC and $(v, 5, 2)$-OOC, respectively. Note that in Fig. 2(a), the values of y for $v = 290$ and $v = 300$ are not plotted because their corresponding graph cardinalities are over one million, in which cases we just know that the percentages are less than 1%, but they cannot be calculated precisely.

From Fig. 2, it can be seen that both the percentages drop with the growing of the code length. Actually, we just stored less than 1% of the OOC graph nodes, but obtained 63% of the Johnson bound regarding the OOC code size. Moreover, the Johnson bound is a theoretical upper bound, whether it can be achieved or not is still unknown for many parameters. If we are admitted more computer memory and more running time, the result would be surely improved.

4.3.3. *Constructing* $(v, 4, 2, 1)$-*OOCs*

For the case that auto- and cross-correlation constraints are unequal, we choose to construct $(v, 4, 2, 1)$-OOCs for $10 \le v \le 100$. For $(v, 4, 2, 1)$-OOCs, Buratti [12] has proved the following bound:

$$s \le \begin{cases} \lceil \frac{v}{8} \rceil, & \text{for } v \equiv 7, 14 \pmod{56} \\ \lfloor \frac{v}{8} \rfloor, & \text{otherwise} \end{cases} \tag{15}$$

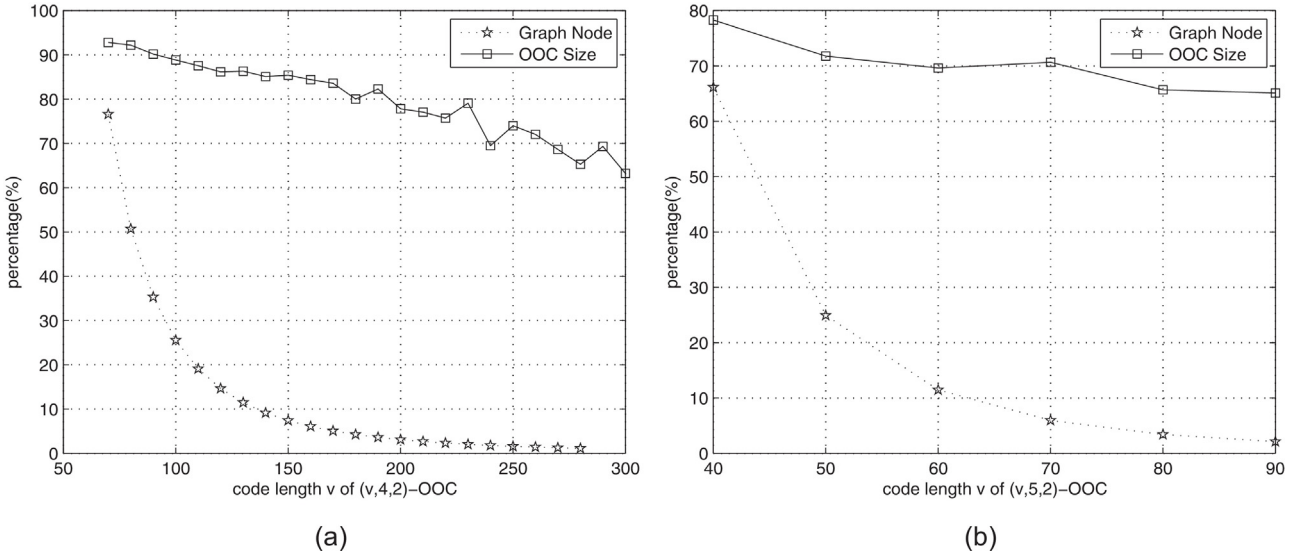

Fig. 2. The percentage of the obtained best code size to the Johnson bound and percentage of the number of stored nodes to the number of nodes in the OOC graph against the code length: (a) $(v, 4, 2)$-OOC and (b) $(v, 5, 2)$-OOC.

Table 7
Results on constructing $(v, 4, 2, 1)$-OOCs for $10 \le v \le 100$.

v	Bound	EAS	v	Bound	EAS	v	Bound	EAS	v	Bound	EAS	v	Bound	EAS
10	1	1	29	3	3	48	6	5	67	8	8	86	10	10
11	1	1	30	3	3	49	6	6	68	8	8	87	10	10
12	1	1	31	3	3	50	6	6	69	8	8	88	11	10
13	1	1	32	4	3	51	6	5	70	9	8	89	11	10
14	2	1	33	4	3	52	6	6	71	8	8	90	11	10
15	1	1	34	4	4	53	6	6	72	9	8	91	11	10
16	2	1	35	4	4	54	6	6	73	9	8	92	11	10
17	2	2	36	4	4	55	6	6	74	9	9	93	11	10
18	2	1	37	4	4	56	7	7	75	9	9	94	11	11
19	2	2	38	4	4	57	7	6	76	9	8	95	11	11
20	2	2	39	4	4	58	7	7	77	9	9	96	12	11
21	2	2	40	5	5	59	7	7	78	9	9	97	12	11
22	2	2	41	5	5	60	7	7	79	9	9	98	12	11
23	2	2	42	5	5	61	7	7	80	10	9	99	12	11
24	3	2	43	5	5	62	7	7	81	10	9	100	12	11
25	3	3	44	5	5	63	8	7	82	10	10			
26	3	3	45	5	5	64	8	7	83	10	10			
27	3	2	46	5	5	65	8	8	84	10	9			
28	3	3	47	5	5	66	8	8	85	10	9			

The experimental results are shown in Table 7, where the bound information is calculated from Eq. (15) directly. Compared with our result, Baicheva and Topalova [4] ran a parallel backtrack search algorithm on IBM supercomputers to find the optimal $(v, 4, 2, 1)$-OOCs for $v \le 181$. However, their algorithm is equivalent to clique search: with the growth of code length, the corresponding graph structure cannot be stored in even supercomputers, thus their off-line backtracking cannot be applied for general bigger v. Note that we are aiming to get OOCs of relatively large sizes for general parameters. In this sense, EAS ourperforms backtracking clearly.

5. Conclusions and future work

Due to its great importance in the field of communications, the problem of designing OOCs has attracted increasing attentions in recent years. Many good results have been obtained by an extensive use of various mathematical tools. But, it seems that we cannot expect the problem to be completely solved for arbitrary parameters in the near future. Meanwhile, the design of OOCs can be reduced to the maximum clique problem (MCP) in graph theory, for which there are many successful heuristics. As a result, we propose clique-based heuristics to design OOCs with relatively large sizes.

In this paper, we present an online algorithm for constructing OOCs based on an EA with guided mutation (EA/G) [33]. Note that the cardinality of the OOC graph is usually huge and the corresponding clique size is quite limited. Hence, the usual clique-based strategy may fail in this case. So, we suppose all the graph vertices arrive in an online manner, and present a substitution technique which leads to our EA with substitution (EAS). Experiments show that EAS has a similarly good performance to EA/G for the MCP on DIMACS benchmarks. Then, we use EAS to construct OOCs with different code weights and different correlation constraints. The experimental study shows that EAS returns OOCs of sizes close to the current best upper bound when the code length is small. Taking $(v, 4, 2)$ for instance, the ratio of the obtained code size to the Johnson bound is over 90% when $v < 100$. If more running time is admitted, the result could be improved.

It should be pointed out that our algorithm is based on EA/G. The main reason for such a choice is its balance between simplicity and efficiency. In fact, there are some existing clique algorithms which have better performance [8,10]. This would be a subject for the future work.

Acknowledgements

The work was supported by the National Natural Science Foundation of China (NSFC) under Grant 11401317 and Grant 11126291, Nanjing University of Information Science and Technology under Grant 2012x021, and the Engineering and Physical Sciences Research Council (EPSRC) of U. K. under Grant EP/K001310/1. The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] R.J.R. Abel, M. Buratti, Some progress on $(v, 4, 1)$ difference families and optical orthogonal codes, J. Combin. Theory Ser. A 106 (1) (2004) 59–75.
[2] T. Alderson, K. Mellinger, Geometric constructions of optimal optical orthogonal codes, Adv. Math. Commun 2 (4) (2008) 451–467.
[3] T. Alderson, K. Mellinger, Families of optimal OOCS with $\lambda = 2$, IEEE Trans. Inf. Theory 54 (8) (2008) 3722–3724.
[4] T. Baicheva, S. Topalova, Optimal $(v, 4, 2, 1)$ optical orthogonal codes with small parameters, J. Combin. Des. 20 (2) (2012) 142–160.
[5] T. Baicheva, S. Topalova, Optimal $(v, 5, 2, 1)$ optical orthogonal codes of small v, Appl. Algeb. Eng. Commun. Comput. 24 (2013) 165–177.
[6] S. Baluja, Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning, Tech. rep., CMU-CS-94-163, Carnegie Mellon University, 1994.
[7] R. Battiti, F. Mascia, Reactive and dynamic local search for max-clique: engineering effective building blocks, Comput. Oper. Res. 37 (3) (2010) 534–542.
[8] R. Battiti, M. Protasi, Reactive local search for the maximum clique problem, Algorithmica 29 (4) (2001) 610–637.
[9] E. Brickell, V. Wei, Optical orthogonal codes and cyclic block designs, Congr. Numer. 58 (1987) 175–192.
[10] M. Brunato, R. Battiti, R-evo: a reactive evolutionary algorithm for the maximum clique problem, IEEE Trans. Evol. Comput. 15 (6) (2011) 770–782.
[11] M. Buratti, Recursive constructions for difference matrices and relative difference families, J. Combin. Des. 6 (3) (1998) 165–182.
[12] M. Buratti, K. Momihara, A. Pasotti, New results on optimal $(v, 4, 2, 1)$ optical orthogonal codes, Des. Codes Cryptogr. 58 (1) (2011) 89–109.
[13] M. Buratti, A. Pasotti, D. Wu, On optimal $(v, 5, 2, 1)$ optical orthogonal codes, Des. Codes Cryptogr. (2011) 1–23.
[14] W. Chu, C. Colbourn, Optimal $(n, 4, 2)$-OOC of small orders, Disc. Math. 279 (1) (2004) 163–172.
[15] F. Chung, J. Salehi, V. Wei, Optical orthogonal codes: design, analysis and applications, IEEE Trans. Inf. Theory 35 (3) (1989) 595–604.
[16] C.J. Colbourn, J.H. Dinitz, D.R. Stinson, Applications of combinatorial designs to communications, cryptography, and networking, in: Surveys in Combinatorics, Cambridge University Press, 1999, pp. 37–100.
[17] J. De Bonet, C. Isbell, P. Viola, MIMIC: finding optima by estimating probability densities, Adv. Neural Inf. Process. Syst. (1997) 424–430.
[18] T. Feng, Y. Chang, L. Ji, Constructions for strictly cyclic 3-designs and applications to optimal OOCS with $\lambda = 2$, J. Combin. Theory Ser. A 115 (8) (2008) 1527–1551.
[19] X. Geng, J. Xu, J. Xiao, L. Pan, A simple simulated annealing algorithm for the maximum clique problem, Inf. Sci. 177 (22) (2007) 5064–5071.

[20] A. Grosso, M. Locatelli, F. Croce, Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem, J. Heuristics 10 (2) (2004) 135–152.
[21] J. Håstad, Clique is hard to approximate within 1-ε, Acta Math. 182 (1) (1999) 105–142.
[22] K. Katayama, A. Hamamoto, H. Narihisa, An effective local search for the maximum clique problem, Inf. Proc. Lett. 95 (5) (2005) 503–511.
[23] W.C. Kwong, P.A. Perrier, P.R. Prucnal, Performance comparison of asynchronous and synchronous code-division multiple-access techniques for fiber-optical local area networks, IEEE Trans. Commun. 39 (11) (1991) 1625–1634.
[24] E. Marchiori, A simple heuristic based genetic algorithm for the maximum clique problem, in: Proc. ACM Symp. Appl. Comput., Citeseer, 1998, pp. 366–373.
[25] E. Marchiori, Genetic, iterated and multistart local search for the maximum clique problem, Appl. Evol. Comput. (2002) 112–121.
[26] P. Ostergard, A fast algorithm for the maximum clique problem, Discrete Appl. Math. 120 (1) (2002) 197–207.
[27] J. Pardalos, M. Resende, On maximum clique problems in very large graphs, DIMACS Ser. 50 (1999) 119–130.
[28] P.R. Prucnal, M.A. Santoro, T.R. Fan, Spread spectrum fiber-optic local area network using optical processing, J. Lightwave Technol. 4 (1986) 547–554.
[29] J. Régin, Using constraint programming to solve the maximum clique problem, in: Principles and Practice of Constraint Programming-CP 2003, Springer, 2003, pp. 634–648.
[30] S. Rebennack, M. Oswald, D. Theis, H. Seitz, G. Reinelt, P.M. Pardalos, A branch and cut solver for the maximum stable set problem, J. Combin. Opt. 21 (4) (2011) 434–457.
[31] J.A. Salehi, Code division multiple access techniques in optical fiber networks-part I: Fundamental principles, IEEE Trans. Commun. 37 (8) (1989) 824–833.
[32] Q. Wu, J.-K. Hao, A review on algorithms for maximum clique problems, Eur. J. Oper. Res. 242 (3) (2015) 693–709.
[33] Q. Zhang, J. Sun, E. Tsang, An evolutionary algorithm with guided mutation for the maximum clique problem, IEEE Trans. Evol. Comput. 9 (2) (2005) 192–200.