IEICE TRANS. FUNDAMENTALS, VOL.E105–A, NO.3 MARCH 2022

PAPER Special Section on Information Theory and Its Applications

A Construction of Binary Punctured Linear Codes and A Supporting Method for Best Code Search*

Takuya OHARA[†a)], ***Nonmember***, **Makoto TAKITA**[††], ***Member***, ***and*** **Masakatu MORII**[†], ***Fellow***

SUMMARY Reduction of redundancy and improvement of error-correcting capability are essential research themes in the coding theory. The best known codes constructed in various ways are recorded in a database maintained by Markus Grassl. In this paper, we propose an algorithm to construct the best code using punctured codes and a supporting method for constructing the best codes. First, we define a new evaluation function to determine deletion bits and propose an algorithm for constructing punctured linear codes. 27 new best codes were constructed in the proposed algorithm, and 112 new best codes were constructed by further modifying those best codes. Secondly, we evaluate the possibility of increasing the minimum distance based on the relationship between code length, information length, and minimum distance. We narrowed down the target (n, k) code to try the best code search based on the evaluation and found 28 new best codes. We also propose a method to rapidly derive the minimum weight of the modified cyclic codes. A cyclic code loses its cyclic structure when it is modified, so we extend the k-sparse algorithm to use it for modified cyclic codes as well. The extended k-sparse algorithm is used to verify our newly constructed best code.

key words: *best code, modified code, linear code, weight distribution, minimum distance*

1. Introduction

Error-correcting codes are indispensable technology to ensure the reliability of the information. The binary (n, k) linear code consists of k information bits and $n - k$ check bits. A major study in coding theory is to construct codes that have a high coding rate $R = k/n$ and a large error-correcting capability. The error-correcting capability of a code C is defined by the minimum Hamming distance between two codewords in C. The larger the minimum distance is, the more errors can be corrected. We refer to the code that has the largest minimum Hamming distance d found so far among the codes of the same code length n and information length k as the best code. Best known codes are maintained on the web by Markus Grassl [1]. In Oct. 2020, new best codes were found, e.g., $(207, 20)$ code, $(231, 21)$ code, and so on. However, the minimum distance of many (n, k) codes does not reach a theoretical upper bound. Therefore, various methods are still being tried to construct best codes [2]–[4].

Manuscript received February 19, 2021.
Manuscript revised June 18, 2021.
Manuscript publicized September 14, 2021.

[†]The authors are with the Graduate School of Engineering, Kobe University, Kobe-shi, 657-8501 Japan.
[††]The author is with the School of Social Information Science, University of Hyogo, Kobe-shi, 651-2197 Japan.

*The material in this paper was presented in part at International Symposium on Information Theory and its Applications 2020 [9].

a) E-mail: oohara@stu.kobe-u.ac.jp

DOI: 10.1587/transfun.2021TAP0007

One of the ways to construct the best code is the modification of codes. There are four types of modification methods: shortened codes eliminated information bits, punctured codes eliminated check bits, lengthened codes attached additional information bits, and extended codes attached additional check bits. In this paper, we call the codes before modification as original codes. And original codes are BCH codes and shortened BCH codes. We focus on the punctured codes and try to construct the best code. In general, deleting m check bits reduces the minimum distance by m, but it is possible to reduce the minimum distance to less than m by carefully selecting deletion bits. Because the number of combinations to choose m deletion bits from the $n - k$ check bits is enormous, it is necessary to reduce the amount of search. Zwanzger [7] defines an evaluation function to construct extended codes. We try to define the function to decide the check bits to delete with reference to Zwanzger's function.

Zwanzger's function requires the entire weight distribution, but it is not easy to find the distribution with a large information length. Therefore, we define a new evaluation function by using the local weight distribution [6] and propose a search algorithm to construct the best codes. 27 new best codes were constructed in the proposed algorithm, and 112 new best codes were constructed by the shortened code and extended code of those best codes.

Next, we propose a supporting method for constructing the best codes. The above 139 new best codes were constructed by determining the original code and the number of bits to be modified based on our experiences. We analyze the relationship among code length, information length, and minimum distance and propose an evaluation method narrowing down the target (n, k) code to try to construct the best code without necessary of any experiences. Based on the evaluation method, 28 new best codes were constructed by the proposed search algorithm.

We also propose a method to rapidly derive the minimum weights of modified cyclic codes. In general, a cyclic code loses its cyclic structure when it is modified. Therefore, we extend the k-sparse algorithm [6] for rapidly deriving the local weight distribution of a cyclic code and propose an algorithm that can rapidly derive the minimum weight of a modified cyclic code. We show algorithms for punctured and shortened codes of cyclic codes. The algorithm for punctured codes is used to verify our newly constructed best code, and the algorithm for shortened codes is used when shortened BCH codes are used as the original codes.

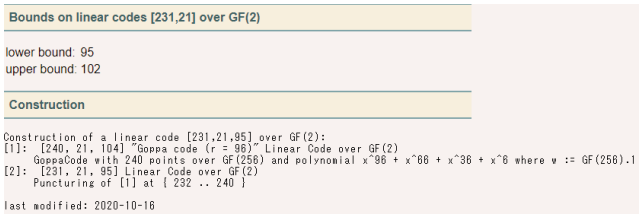


Fig. 1 (231, 21) code registered in the best known code database [1].

The remainder of the paper is organized as follows. Section 2 describes the best codes. Section 3 reviews the k-sparse algorithm. Section 4 presents the extended k-sparse algorithm for punctured and shortened codes. Section 5 proposes an algorithm for constructing best codes using punctured codes and a supporting method for best code search. Also, it shows our newly constructed best codes.

2. Best Code

The error-correcting capability of a code C is defined by the minimum value of the Hamming distance between two codewords in C, i.e., the minimum Hamming distance. Best code is the code that has the largest minimum distance d found so far among the codes of the same code length n and the information length k. Markus Grassl maintains the best known code in an online database [1]. The database contains the upper bound of minimum distance, the lower bound of minimum distance, the construction method, and the construction date of each (n, k) linear code. Denote (n, k) codes with minimum distance d as (n, k, d) codes. The upper bound is a theoretically given value, and the lower bound is the minimum distance of the constructed code. An example of the best known code recorded in the database is shown in Fig. 1. There are codes in the database that the minimum distance does not reach the upper bound. Therefore, some studies have been carried out to increase the minimum distance [2]–[4]. The best code construction method using modified codes [7] is one of them.

3. k-Sparse Algorithm

It is necessary to derive the minimum weight and the number of codewords of minimum weight when verifying whether the best code has been constructed, where the weight is the number of non-zero bits in a codeword. In general, it is necessary to generate all codewords and count their weights in order to derive the number of codewords A_w with weight w for a (n, k) linear code. However, its computational complexity is $O(n \cdot 2^k)$, making it difficult to derive for large codes. The k-sparse algorithm [6] has been proposed by Li, Mohri, and Morii as an efficient way to derive weights for cyclic codes by using their cyclic structure.

In this paper, we give explanations using systematic cyclic codes whose $k \times n$ generator matrix is represented by $\mathbf{G} = [\mathbf{I}_k \mathbf{P}]$, where $\mathbf{I}_k$ is a $k \times k$ identity matrix and $\mathbf{P}$ is the $k \times (n-k)$ matrix of parity check symbols. Let a codeword of (n, k) cyclic codes C be $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ and a generator matrix of (n, k) cyclic codes C be $\mathrm{G}_{n,k}$. We introduce the theorems and definitions used in k-sparse algorithm as follows. We explain them with examples using the $(7, 4, 3)$ cyclic code whose generator polynomial is $1101 (= 1 + x + x^3)$ and the systematic generator matrix $\mathrm{G}_{7,4} = \begin{bmatrix} 1000110 \\ 0100011 \\ 0010111 \\ 0001101 \end{bmatrix}$.

Definition 1: [5] The set of codewords $\boldsymbol{c}_s = (c_s, c_{(s+1) \bmod n}, \ldots, c_{(s-1) \bmod n})$ $(0 \le s \le p-1)$ obtained by cyclic shifting a codeword $\boldsymbol{c}$ of length n and weight w is called the w-class generated by $\boldsymbol{c}$ where p is the least cycle such that $\boldsymbol{c}_0 = \boldsymbol{c}_p$.

For example, if a codeword $\boldsymbol{c}_0$ is $(0, 0, 0, 1, 1, 0, 1)$, $\boldsymbol{c}_1 = (0, 0, 1, 1, 0, 1, 0)$, $\boldsymbol{c}_2 = (0, 1, 1, 0, 1, 0, 0)$, $\ldots$, $\boldsymbol{c}_7 = (0, 0, 0, 1, 1, 0, 1) = \boldsymbol{c}_0$. Therefore, $p = 7$. In cyclic codes, a codeword of weight w always generates a w-class, so if we can generate at most one representative codeword from all the w-class, we can derive A_w without verifying all the codewords. At this time, the codeword that has the possibility of becoming the representative codeword is called the candidate codeword. The more the number of candidate codewords is reduced, the smaller the computational complexity becomes.

Definition 2: [5] Let $\boldsymbol{c}(i||o)$ denote the subvector$(c_j, j \in (i||o))$ of a vector $\boldsymbol{c}$, while $(i||o)$ is an ordered set of o integers $\{i, (i+1) \bmod n, \ldots, (i+o-1) \bmod n\}, (1 \le o \le n)$.

For example, $\boldsymbol{c}(4||1) = (c_4)$, $\boldsymbol{c}(4||2) = (c_4, c_5)$, and $\boldsymbol{c}(4||3) = (c_4, c_5, c_6)$.

Theorem 1: [5] Any w-class contains at least one vector $\boldsymbol{c}_s$ with

$$f_i + 1 \ge \lceil i\lambda \rceil, 1 \le i \le w, \tag{1}$$

where f_i denote the position of the i-th symbol "1" in vector $\boldsymbol{c}_s$ and $\lambda = n/w$.

For example, any 3-class of $\mathrm{G}_{7,4}$ contains a vector with $f_1 \ge 2$, $f_2 \ge 4$, and $f_3 \ge 6$.

Definition 3: [5], [6] If a vector $\boldsymbol{u}$ of length l $(1 \le l \le n)$ satisfies $W_h(\boldsymbol{u}) \le \lfloor l/\lambda \rfloor$ where $W_h(\boldsymbol{u})$ is the weight of $\boldsymbol{u}$ and $f_i + 1 \ge \lceil i\lambda \rceil, 1 \le i \le w$, then $\boldsymbol{u}$ is called l-sparse. Particularly, for $l = k$, the vector $\boldsymbol{u}$ is called k-sparse and for $l = n$, the vector $\boldsymbol{u}$ is called n-sparse.

For example, in case of 3-class of $\mathrm{G}_{7,4}$, information sequences of k-sparse are $(0, 0, 1, 0)$ and $(0, 0, 0, 1)$ because they satisfies $f_1 \ge 2$. From Definition 3, it can be seen that a subvector $\boldsymbol{c}(0||k)$ of a codeword $\boldsymbol{c}$ that is n-sparse is always k-sparse. Here $\boldsymbol{c}(0||k)$ denotes the information sequence if the cyclic code is a systematic code. For $i = w$, the position f_w of w-th symbol "1" in $\boldsymbol{c}$ satisfies

$$f_w + 1 \ge \lceil w\lambda \rceil = w \cdot (n/w) = n \, (0 \le f_w \le n-1). \tag{2}$$

That is $f_w = n - 1$, and Theorem 2 holds.

Theorem 2: [6] When the codeword $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$

Algorithm 1 k-sparse algorithm [6]

Input: w, $\mathbf{G}$ of a systematic cyclic code
Output: A_w
  $i \leftarrow 1, h[1 \ldots n][1 \ldots n] \leftarrow 0$
  while $(i \leq \lfloor k/\lambda \rfloor)$ **do**
    while Next_k-sparse(i) **do**
      $\boldsymbol{c}$ = Generate($\mathbf{G}$, w)
      if Check_condition($\boldsymbol{c}$, w) **then**
        Derive p and q of the w-class generated by $\boldsymbol{c}$
        $h[p][q] \leftarrow h[p][q] + 1$
      end if
    end while
    $i \leftarrow i + 1$
  end while
  Derive $A_w = \sum_{p=1}^{n} p \sum_{q=1}^{n} h[p][q]/q$
  return A_w

of weight w is n-sparse,

$$c_{n-1} = 1. \tag{3}$$

Based on Theorem 2, we can determine whether $\boldsymbol{c}$ is n-sparse by simply checking the c_{n-1} symbol. If the probabilities of "0" and "1" appearing in c_{n-1} are approximately equal, the number of candidate codewords will be about one-half.

Theorem 3: [6] Let the i-th $(0 \leq i \leq k-1)$ row vector of the generator matrix $\mathbf{G}$ be $\boldsymbol{g}_i$. In particular, let $\boldsymbol{g}_i$ with an odd weight be $\boldsymbol{g}_i^*$, and let the codeword generated by combining any $\boldsymbol{g}_i^*$s be $\boldsymbol{c}^*$. At this time, $W_h(\boldsymbol{c}^*)$ is odd if combining odd $\boldsymbol{g}_i^*$s and even if combining even $\boldsymbol{g}_i^*$s.

The codewords of a binary linear code are generated by multiplying the information sequence by the generator matrix. This means the XOR operation of row vector $\boldsymbol{g}_i$ of the generator matrix corresponding to the position of the symbol "1" in the information sequence. In other words, the parity (even or odd) of generated codeword is determined by the parity of the $\boldsymbol{g}_i$. w and $w+1$ often have the same value of $\lfloor k/\lambda \rfloor$. Based on Theorem 3, it is possible to separate these codewords. If k-sparse generates an equal number of odd and even weighted codewords, the number of candidate codewords will be about one-half.

For example, based on Theorem 3, we only encode $(0,0,0,1)$ of k-sparse by $\mathrm{G}_{7,4}$ when we generate a codeword of 3-class because $W_h(\boldsymbol{g}_2)$ is even.

Theorem 4: [6] The w-class of a codeword generated from k-sparse with weight i $(i \leq \lfloor k/\lambda \rfloor)$ contains at least one codeword $\boldsymbol{c}$ satisfying

$$W_h(\boldsymbol{c}(k||o)) \leq \lfloor (k+o)/\lambda \rfloor - i \tag{4}$$

for all o $(1 \leq o \leq n-k)$.

For example, the codeword $\boldsymbol{c} = (0,0,0,1,1,0,1)$ generated by multiplying $(0,0,0,1)$ of k-sparse by $\mathrm{G}_{7,4}$ satisfies Eq. (4) because $W_h(\boldsymbol{c}(4||1)) = 1 \leq 1$, $W_h(\boldsymbol{c}(4||2)) = 1 \leq 1$, and $W_h(\boldsymbol{c}(4||3)) = 2 \leq 2$. Some of the codewords generated from k-sparse do not satisfy n-sparse. By checking Eq. (4), we can early stop computing the codewords that are not n-sparse.

Algorithm 1 shows k-sparse algorithm. w is the weight for which we want to obtain the number of codewords in a cyclic code C, and A_w is the number of codewords of weight w. We define a function Next_k-sparse(i) as checking if there are other k-sparse of weight i and returning True if there are or False if there are not. We define a function Generate($\mathbf{G}$, w) to generate only codeword $\boldsymbol{c}$ whose weight of even or odd matches w from k-sparse based on Theorem 3. We define a function Check_condition($\boldsymbol{c}$, w) as checking whether $\boldsymbol{c}$ satisfies the following three equations: Eq. (3), $W_h(\boldsymbol{c}) = w$, and Eq. (4). It returns True if all of them are satisfied or False if any of them are not satisfied. $h[p][q]$ has the number of n-sparse where p is the least cycle and q is the number of n-sparse that satisfy Eq. (1) in every $\boldsymbol{c}_s$ $(0 \leq s < p)$.

For example, when we derive A_3 of $\mathrm{G}_{7,4}$, the codeword $\boldsymbol{c} = (0,0,0,1,1,0,1)$ is only n-sparse generated by k-sparse. By cyclic shifting $\boldsymbol{c}$ and checking Eq. (1) for $\boldsymbol{c}_s$, i,e, $f_1 \geq 2$, $f_2 \geq 4$, and $f_3 \geq 6$, we get $p = 7$ and $q = 1$. Therefore, $A_3 = 7 \cdot 1/1 = 7$.

4. Extended k-Sparse Algorithm

Cyclic codes are relatively easy to implement for coding and decoding and have a large minimum distance and excellent error-correcting capability. Therefore, we can expect to construct new best codes by making some modifications to the cyclic codes. However, if even a small modification is made to the generator matrix, the cyclic structure of the cyclic code is lost. Therefore, the k-sparse algorithm cannot be used to derive the weight distribution of the modified code, and it is common to use the exhaustive search method to generate all the codewords and verify the weights. In this section, we extend the k-sparse algorithm and propose an efficient method to derive the number of codewords of minimum weight for modified cyclic codes. We consider the properties of minimum weights for punctured codes and shortened codes and show the proposed derivation method. Let the number of bits to be modified be m.

4.1 A Method for Deriving the Number of Codewords with Minimum Weight for Punctured Cyclic Codes

We consider the properties of $(n-m, k, d')$ punctured code C_P, which is a modification of (n, k, d) cyclic code C. The punctured code C_P is constructed by deleting m bits from the check bits of the generator matrix, which is equivalent to deleting the corresponding m bits from all the codewords of the original code. Therefore, the weight distribution of the punctured code can also be derived by examining the weights of the codewords after deleting the m bits. The original codeword is $\boldsymbol{c}$, and the codeword after deleting m bits from the check symbol of $\boldsymbol{c}$ is $\boldsymbol{c'}$. In this case,

$$W_h(\boldsymbol{c'}) \geq W_h(\boldsymbol{c}) - m \tag{5}$$

is established between $\boldsymbol{c}$ and $\boldsymbol{c'}$. From Eq. (5), to derive $A_{w_{\min}}$ of a punctured code, we can generate codewords whose

Algorithm 2 Extended k-sparse algorithm for punctured codes

Input: $w = d$, $w_{\min} = \infty$, $\mathbf{G}$ of a systematic cyclic code, m, Position of the m check bits to be punctured
Output: $A_{w_{\min}}$
  $h[1 \dots n][1 \dots n][1 \dots n] \leftarrow 0$
  while $(w \le w_{\min} + m)$ **do**
    $\lambda = n/w$
    $i \leftarrow 1$
    while $(i \le \lfloor k/\lambda \rfloor)$ **do**
      while Next_k-sparse(i) **do**
        $\boldsymbol{c}$ = Generate($\mathbf{G}$, w)
        if Check_condition($\boldsymbol{c}$, w) **then**
          w' = Puncturing($\boldsymbol{c}_s$)
          if $(w' < w_{\min})$ **then**
            $w_{\min} \leftarrow w'$
          end if
          Derive p' and q of the w-class generated by $\boldsymbol{c}$
          $h[p'][q][w_{\min}] \leftarrow h[p'][q][w_{\min}] + 1$
        end if
      end while
      $i \leftarrow i + 1$
    end while
    $w \leftarrow w + 1$
  end while
  Derive $A_{w_{\min}} = \sum_{p'=1}^{n} p' \sum_{q=1}^{n} h[p'][q][w_{\min}]/q$
  return $A_{w_{\min}}$

weights are from d to $w_{\min}+m$ of C and verify the weights by deleting m check symbols where d and $w_{\min}$ are the minimum distance of C and C_P, respectively.

Algorithm 2 shows how to derive the number of the minimum weight codewords for a punctured code using the k-sparse algorithm. We define a function Puncturing($\boldsymbol{c}_s$) to generate $\boldsymbol{c}_s$ $(0 \le s < p)$ from $\boldsymbol{c}$ and calculate w', the weight of the codeword after deleting m check symbols of $\boldsymbol{c}_s$. $h[p'][q][w_{\min}]$ has the number of n-sparse where p' is the number of vectors in $\boldsymbol{c}_s$ $(0 \le s < p)$ whose weight become $w_{\min}$ after deleting the m check bits.

We show an example of puncturing the position 6 of $G_{7,4}$ and derive A_2 of the $(6, 4, 2)$ punctured code. When codewords $\boldsymbol{c}_0 = (0,0,0,1,1,0,1)$, $\boldsymbol{c}_1 = (0,0,1,1,0,1,0)$, ..., $\boldsymbol{c}_6 = (1,0,0,0,1,1,0)$ are punctured position 6, the codewords whose weight becomes 2 are $\boldsymbol{c}'_0 = (0,0,0,1,1,0)$, $\boldsymbol{c}'_4 = (1,0,1,0,0,0)$, and $\boldsymbol{c}'_5 = (0,1,0,0,0,1)$. Therefore, $p' = 3$. Then, $A_2 = 3 \cdot 1/1 = 3$.

When this algorithm is used for best code verification, the minimum distance d' of the best code is already known, so the input should be $w_{\min} = d'$ instead.

4.2 A Method for Deriving the Number of Codewords with Minimum Weight for Shortened Cyclic Codes

We consider the properties of $(n-m, k-m, d)$ shortened code C_S, which is a modification of (n, k, d) cyclic code C. Different from punctured codes, shortened codes are constructed by deleting information bits, so the number of codewords decreases but the weight of codewords does not change.

Algorithm 3 shows how to derive the number of the minimum weight codewords for a shortened code using the

Algorithm 3 Extended k-sparse algorithm for shortened codes

Input: w, $\mathbf{G}$ of a systematic cyclic code, Position of m information bits to be shortened
Output: A_w
  $i \leftarrow 1$, $h[1 \dots n][1 \dots n] \leftarrow 0$
  while $(i \le \lfloor k/\lambda \rfloor)$ **do**
    while Next_k-sparse(i) **do**
      $\boldsymbol{c}$ = Generate($\mathbf{G}$, w)
      if Check_condition($\boldsymbol{c}$, w) **then**
        Derive p'' and q of the w-class generated by $\boldsymbol{c}$
        $h[p''][q] \leftarrow h[p''][q] + 1$
      end if
    end while
    $i \leftarrow i + 1$
  end while
  Derive $A_w = \sum_{p''=1}^{n} p'' \sum_{q=1}^{n} h[p''][q]/q$
  return A_w

k-sparse algorithm. $h[p''][q]$ has the number of n-sparse where p'' is the number of vectors for which all m deleted symbols are "0", i.e., vectors that still exist after shortening in every $\boldsymbol{c}_s$.

We show an example of shortening the position 0 of $G_{7,4}$ and derive A_3 of the $(6, 3, 3)$ shortened code. When codewords $\boldsymbol{c}_0 = (0,0,0,1,1,0,1)$, $\boldsymbol{c}_1 = (0,0,1,1,0,1,0)$, ..., $\boldsymbol{c}_6 = (1,0,0,0,1,1,0)$ are shortened position 0, the codewords that still exist after shortening are $\boldsymbol{c}''_0 = (0,0,1,1,0,1)$, $\boldsymbol{c}''_1 = (0,1,1,0,1,0)$, $\boldsymbol{c}''_2 = (1,1,0,1,0,0)$, and $\boldsymbol{c}''_5 = (1,0,0,0,1,1)$. Therefore, $p'' = 4$. Then, $A_3 = 4 \cdot 1/1 = 4$.

This algorithm can be used when a shortened code is used as the original code.

5. Proposed Methods for Searching Best Codes

5.1 A Search Algorithm for Constructing Best Codes Using Punctured Codes

A punctured code is constructed by deleting the check bits of the original code. Because the minimum distance varies with the chosen deletion bits, the best code may be constructed if the deletion bits are chosen carefully.

Zwanzger [7] proposes an evaluation function to construct extended codes based on the weight generating function. Let the generator matrix of the q-ary (n, k) linear code be $\mathbf{G}$, and let the number of codewords of Hamming weight w be A_w, where the Hamming weight is the number of non-zero bits in a codeword. Then, the weight generating function is

$$p(\mathbf{G}) = A_0 x^0 + A_1 x^1 + \cdots + A_n x^n, \tag{6}$$

and the Zwanzger's evaluation function eval_1 is defined by

$$\mathrm{eval}_1(\mathbf{G}) = -\sum_{i=1}^{n} A_i q^{k(n-i)}. \tag{7}$$

A complete weight distribution is required for using eval_1, but it takes too much time to derive the weight distribution of code with large n or k. Therefore, we define a new

Algorithm 4 Construction method of best codes by puncturing

Input: d', m, $\mathbf{G}$ of BCH code, b
Output: $\mathbf{G}'$ of new code
  $j \leftarrow 1$
  buff[1 . . . m][1 . . . $(n-k)$], deleteList[1 . . . m], List[1 . . . m][1 . . . b]
  Flag ← True
  Add $\boldsymbol{v}(\mathbf{G}, m, d')$ to V_0
  while Flag **do**
    while $j \leq m$ **do**
      for $i = 1$ to $j-1$ **do**
        Puncturing the bit of position deleteList[i] from V'_{i-1}
      end for
      for $r = k$ to $n-j$ **do**
        buff[j][$r-k+1$] ← cal_eval(V'_{j-1}, r)
      end for
      for $i = 1$ to b **do**
        List[j][i] ← preserve(buff[j][1 . . . $(n-k)$])
      end for
      deleteList[j] ← pop(List[j][1])
      $j \leftarrow j+1$
    end while
    Derive the minimum distance $d_{\min}$ of V'_m
    if ($d_{\min} \geq d'$) **then**
      return $\mathbf{G}'$
    end if
    Flag ← False
    while $j > 1$ **do**
      $j \leftarrow j-1$
      remove deleteList[j]
      if List[j][1]$\neq \phi$ **then**
        deleteList[j] ← pop(List[j][1])
        $j \leftarrow j+1$
        Flag ← True
        Break
      end if
    end while
  end while

evaluation function based on the codewords around the minimum distance. Because the minimum distance decrease by m at most when the check bits are deleted by m bits, we only consider the codewords whose Hamming weight is less than $d'+m$ when we challenge the construction of a code with a minimum distance d'. Therefore, we define the evaluation function eval_2 as

$$\mathrm{eval}_2(V'_{j-1}, r) = \sum_{i=1}^{d'+m-(j+1)} A_i 2^{d'+m-(j+i)}, \tag{8}$$

where j ($1 \leq j \leq m$) is the j-th deletion, r is the position of the j-th deletion bit, V'_{j-1} is a set of codewords $\boldsymbol{v}'$ that are codewords after $\boldsymbol{v} \in V_0$ has $j-1$ bits removed, and V_0 is the set of codewords whose Hamming weights are less than $d'+m$. eval_2 is a weighted addition to the number of codewords whose Hamming weight is between $d-j$ and $d'+m-1-j$. If there are fewer codewords whose Hamming weight is $d-j$ after j bits deletion, eval_2 will take a smaller value. After m bits deletion, if eval_2 is 0, i.e., there are no codewords whose weight is less than d', then a code with minimum distance d' can be constructed. Because codewords whose Hamming weight is less than $d'+m$ are obtained by Algorithm 1 [6] or Algorithm 3 and the local weight distribution of the punctured code is obtained by Algorithm 2, a computation complexity for calculating eval_2 reduced compared with that of eval_1.

We propose Algorithm 4 for constructing the best code based on eval_2. Let the Hamming weight of the codeword $\boldsymbol{c}$ be $W_h(\boldsymbol{c})$ and let the codeword generated by generator matrix $\mathbf{G}$ and satisfying $W_h(\boldsymbol{v}) < d'+m$ be $\boldsymbol{v}(\mathbf{G}, m, d')$. buff[][] has the value of eval_2 of all bits that can be deleted in j-th. deleteList[] has the deletion bits and List[][] has b candidates of the j-th deletion bit. A function cal_eval(V'_{j-1}, r) means that the bit of position r is punctured from V'_{j-1} and calculate eval_2. A function preserve(buff[j][1 . . . $(n-k)$]) means that the deletion position with the lowest value out of buff[j][1], buff[j][2], . . . , buff[j][$n-k$] is extracted, and then it is removed from the buff. A function pop(List[j][1]) means that the first element of List[j] is picked up, and it is removed from List. Let the set that m bits in deleteList are deleted from V_0 be V'_m.

The proposed algorithm uses the backtracking method [8]. The number of punctured codes constructed by m-bit deletion is the number of combinations when selecting m bits from $n-k$ check bits. Since it is difficult to search all punctured codes, the proposed algorithm reduces the number of punctured codes to search to b^m.

5.2 A Supporting Method for Searching Best Codes

We targeted and searched for the (n,k) codes that were most likely to discover new best codes based on experience. In this section, we propose a supporting method for searching the best code. We analyze the relationship among code length n, information length k, and minimum distance d and propose an evaluation method narrowing down the target (n,k) code that is most likely to construct the best code. We consider several evaluation values e_1, e_2, e_3, and e_4 in [9]. As a result, we define

$$E(n,k) = e_1(n,k) + e_2(n,k) + e_3(n,k) + e_4(n,k) \tag{9}$$

by using them. Each evaluation value is given in the following sections. Let the minimum distance d of the (n,k) code be $d(n,k)$.

5.2.1 Evaluation Using the Minimum Distance Difference between Adjacent Parameters

In the best known code database [1], we can specify any range of n and k and know the lower bound and upper bound as shown in Fig. 2. The left value of each square shows the lower bound, and the right value shows the upper bound of the minimum distance. As shown in Fig. 2, there are some codes whose minimum distance does not increase as n or k increases or decreases. Therefore, if the difference in the minimum distance between the evaluation target and the adjacent codes is large, the minimum distance is likely to be increased. Let the minimum distance difference between

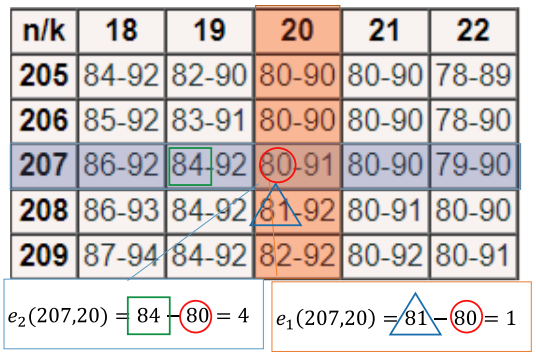


Fig. 2 Example of $e_1(207, 20)$ and $e_2(207, 20)$.

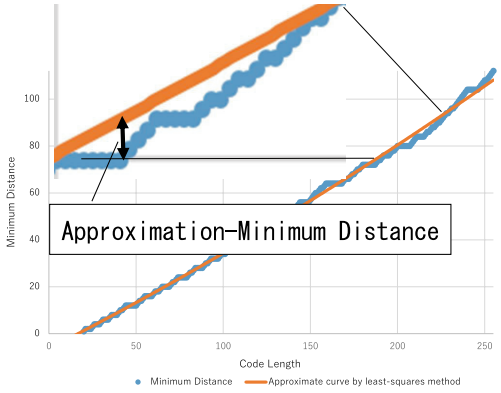


Fig. 3 Relationship between code length and minimum distance when information length is 20.

adjacent parameters of the (n, k) code be $e_1(n, k)$ and $e_2(n, k)$. Then, $e_1(n, k)$ and $e_2(n, k)$ are defined by

$$e_1(n, k) = d(n + 1, k) - d(n, k), \tag{10}$$
$$e_2(n, k) = d(n, k - 1) - d(n, k). \tag{11}$$

5.2.2 Evaluation Using the Least-Squares Method

The minimum distance of a linear code is correlated with the increase or decrease in code length and information length, as shown in Fig. 3. Focusing on this point, we approximate the relationship between the code length or the information length and the minimum distance by the least-squares method. For given measurement pairs (x_i, y_i), the least-squares method is a method to set the coefficients such that the sum of squares of the residuals is minimized so that the function is an appropriate approximation to the measurement.

First, we define a quadratic approximation curve of minimum distance as $f_k(n)$ when information length k is a constant value and code length n is a variable value. We also define the difference between the approximation $f_k(n)$ and the minimum distance be $e_3(n, k)$. For given pairs $(i, d(i, k))$, $k \le i \le 255$, $f_k(x) = a_k x^2 + b_k x + c_k$ is given by calculating a_k, b_k and c_k that minimize the sum of squares of the residuals S_k defined by

Table 1 Generator polynomials of original BCH codes.

(n, k, d)	Generator polynomial
(127, 29, 43)	94725606304a7865e875b608a
(233, 30, 88)	f64bd9710988280185c4c489af9910b5099d9395c91923131db
(255, 45, 87)	86ca2a1ec416c37c9f1c940779d83e45fcce824d23872b43a1056
(255, 47, 85)	de6355a506c50290c9a61db297cd9307fd08d86759698f02376a8

Table 2 Best codes by modifying (127, 29, 43) BCH code.

Best code	Original code	m	d^+	Method
(112, 27, 33)	(125, 27, 43)	13	1	Puncturing
	12, 13, 19, 22, 27, 39, 52, 53, 55, 78, 79, 80, 95			
(113, 27, 34)	(112, 27, 33)	1	1	Extending

Table 3 Best codes by modifying (233, 30, 88) BCH code.

Best code	Original code	m	d^+	Method
(213, 30, 73)	(233, 30, 88)	20	1	Puncturing
	1, 3, 18, 25, 35, 60, 65, 74, 93, 95, 115, 124 128, 153, 159, 161, 170, 174, 194, 198			
(214, 30, 74)	(213, 30, 73)	1	1	Extending
(216, 30, 75)	(233, 30, 88)	17	1	Puncturing
	1, 11, 18, 35, 39, 53, 62, 95, 121 129, 138, 140, 148, 149, 174, 187, 189			
(217, 30, 76)	(216, 30, 75)	1	1	Extending

$$S_k = \sum_{i=k}^{255} (d(i, k) - a_k x_i^2 - b_k x_i - c_k)^2. \tag{12}$$

Then, $e_3(n, k)$ is defined by

$$e_3(n, k) = f_k(n) - d(n, k). \tag{13}$$

Second, we define a quadratic approximation curve of minimum distance as $f_n(k)$ when code length n is a constant value and information length k is a variable value. We also define the difference between the approximation $f_n(k)$ and the minimum distance be $e_4(n, k)$. For given pairs $(i, d(n, i))$, $1 \le i \le n$, $f_n(x) = a_n x^2 + b_n x + c_n$ is given by calculating a_n, b_n and c_n that minimize the sum of squares of the residuals S_n defined by

$$S_n = \sum_{i=1}^{n} (d(n, i) - a_n x_i^2 - b_n x_i - c_n)^2. \tag{14}$$

Then, $e_4(n, k)$ is defined by

$$e_4(n, k) = f_n(k) - d(n, k). \tag{15}$$

5.3 Constructed Best Codes

In this section, we present new codes that we constructed by using Algorithm 4. To confirm that they are the best codes, we calculate the minimum distance using the extended k-sparse algorithms proposed in Sect. 4.

Firstly, we construct the 27 new best codes using Algorithm 4 from four BCH codes in Table 1 and their shortened codes. Generator polynomials of these BCH codes are shown in hexadecimal, e.g.,

Table 4 Best codes by modifying (255, 45, 87) BCH code ($33 \leq k \leq 39$).

Best code	Original code	m	d^+	Method
(230, 33, 77)	(243, 33, 87)	13	1	Puncturing
	4, 9, 49, 69, 84, 91, 99, 103, 107, 111, 165, 174, 178			
(231, 33, 78)	(230, 33, 77)	1	1	Extending
(224, 34, 73)	(244, 34, 87)	20	1	Puncturing
	1, 8, 52, 58, 83, 99, 100, 102, 103, 104, 106, 107 110, 111, 135, 154, 187, 203, 205, 208			
(225, 34, 74)	(224, 34, 73)	1	2	Extending
(226, 34, 74)	(228, 36, 74)	2	1	Shortening
(228, 34, 75)	(244, 34, 87)	16	1	Puncturing
	6, 47, 48, 52, 67, 99, 103, 107, 109, 110 120, 135, 139, 164, 169, 209			
(229, 34, 76)	(228, 34, 75)	1	1	Extending
(226, 35, 73)	(227, 36, 73)	1	1	Shortening
(227, 35, 74)	(228, 36, 74)	1	1	Shortening
(229, 35, 75)	(245, 35, 87)	16	1	Puncturing
	11, 16, 37, 53, 60, 66, 87, 90, 105 111, 113, 163, 168, 207, 209, 210			
(230, 35, 76)	(229, 35, 75)	1	1	Extending
(227, 36, 73)	(246, 36, 87)	19	1	Puncturing
	9, 11, 15, 40, 43, 63, 103, 104, 134, 137, 145 151, 159, 162, 186, 201, 206, 208, 209			
(228, 36, 74)	(227, 36, 73)	1	1	Extending
(229, 37, 73)	(231, 39, 73)	2	1	Shortening
(230, 37, 74)	(232, 39, 74)	2	2	Shortening
(231, 37, 74)	(236, 42, 74)	5	1	Shortening
(232, 37, 75)	(247, 37, 87)	15	1	Puncturing
	19, 21, 83, 92, 103, 104, 105, 115 137, 145, 170, 176, 201, 206, 208			
(233, 37, 76)	(232, 37, 75)	1	1	Extending
(230, 38, 73)	(231, 39, 73)	1	1	Shortening
(231, 38, 74)	(232, 39, 74)	1	2	Shortening
(232, 38, 74)	(236, 42, 74)	4	1	Shortening
(224, 39, 69)	(249, 39, 87)	25	1	Puncturing
	1, 5, 10, 16, 29, 31, 33, 34, 38, 64, 68, 76, 88, 103, 114 138, 150, 153, 156, 168, 172, 207, 208, 209, 210			
(225, 39, 70)	(224, 39, 69)	1	2	Extending
(226, 39, 70)	(229, 42, 70)	3	2	Shortening
(227, 39, 70)	(232, 44, 70)	5	1	Shortening
(228, 39, 71)	(230, 41, 71)	2	1	Shortening
(229, 39, 72)	(231, 41, 72)	2	2	Shortening
(230, 39, 72)	(235, 44, 72)	5	1	Shortening
(231, 39, 73)	(249, 39, 87)	18	1	Puncturing
	15, 16, 20, 41, 53, 76, 102, 103, 114, 125 150, 169, 172, 183, 185, 195, 208, 209			
(232, 39, 74)	(231, 39, 73)	1	2	Extending
(233, 39, 74)	(236, 42, 74)	3	1	Shortening

$$100010111_{(2)} = 8b8_{(16)}. \tag{16}$$

We divide the polynomials into binary blocks of length 4, fill the last digits with zeros if necessary, and then convert to hexadecimal as in Eq. (16). Secondly, we find 120 new best codes by shortening or extending those code. When the minimum distance of constructed best code is an odd value, the minimum distance can be increased by 1-bit extended code. Some best codes are constructed by shortening because the minimum distance of a shortened code does not decrease compared with the original code. In addition, we constructed 28 new best codes by using $E(n, k)$ to choose (n, k) codes to search. As a result, we found 167 new best codes.

The parameters of constructed codes are shown in Tables 2–7. In Tables 2–7, m is the number of modifying, d^+ is the number of increased minimum distances from the best known code. The punctured positions are shown below each code in the tables, with the beginning of the check bit being 1. For example, (112, 27, 33) code is constructed by deleting the 12th, 13th, 19th, . . . , 95th check bits of (125,27,43) code from Table 2. When we use m-bit shortened BCH codes as

Table 5 Best codes by modifying (255, 45, 87) BCH code ($40 \leq k \leq 42$).

Best code	Original code	m	d^+	Method
(225, 40, 69)	(250, 40, 87)	25	1	Puncturing
	8, 14, 24, 28, 49, 58, 77, 83, 85, 88, 89, 100, 101, 113 116, 122, 136, 140, 149, 159, 166, 193, 206, 208, 210			
(226, 40, 70)	(225, 40, 69)	1	2	Extending
(227, 40, 70)	(229, 42, 70)	2	2	Shortening
(228, 40, 70)	(232, 44, 70)	4	1	Shortening
(229, 40, 71)	(230, 41, 71)	1	1	Shortening
(230, 40, 72)	(231, 41, 72)	1	2	Shortening
(231, 40, 72)	(235, 44, 72)	4	1	Shortening
(233, 40, 73)	(235, 42, 73)	2	1	Shortening
(234, 40, 74)	(236, 42, 74)	2	1	Shortening
(219, 41, 65)	(251, 41, 87)	32	1	Puncturing
	19, 20, 43, 51, 53, 57, 60, 63, 64, 75, 82, 94, 95 99, 110, 113, 119, 124, 135, 140, 147, 149, 152 162, 176, 177, 180, 189, 190, 192, 208, 209			
(220, 41, 66)	(219, 41, 65)	1	2	Extending
(221, 41, 66)	(219, 41, 65)	2	1	Extending
(223, 41, 67)	(224, 42, 67)	1	1	Shortening
(224, 41, 68)	(225, 42, 68)	1	2	Shortening
(225, 41, 68)	(227, 43, 68)	2	1	Shortening
(227, 41, 69)	(228, 42, 69)	1	1	Shortening
(228, 41, 70)	(229, 42, 70)	1	2	Shortening
(229, 41, 70)	(232, 44, 70)	3	2	Shortening
(230, 41, 71)	(251, 41, 87)	21	3	Puncturing
	10, 20, 24, 28, 35, 49, 61, 82, 84, 103, 118, 137 140, 148, 169, 171, 172, 188, 207, 208, 209			
(231, 41, 72)	(230, 41, 71)	1	3	Extending
(232, 41, 72)	(235, 44, 72)	3	2	Shortening
(233, 41, 72)	(237, 45, 72)	4	1	Shortening
(234, 41, 73)	(235, 42, 73)	1	1	Shortening
(235, 41, 74)	(236, 42, 74)	1	1	Shortening
(221, 42, 65)	(223, 44, 65)	2	1	Shortening
(222, 42, 66)	(224, 44, 66)	2	2	Shortening
(223, 42, 66)	(225, 44, 66)	2	2	Shortening
(224, 42, 67)	(252, 42, 87)	28	2	Puncturing
	8, 13, 23, 25, 39, 53, 60, 75, 77, 78, 82 83, 97, 104, 110, 115, 139, 148, 153, 154 155, 162, 185, 198, 203, 205, 207, 208			
(225, 42, 68)	(224, 42, 67)	1	2	Extending
(226, 42, 68)	(227, 43, 68)	1	2	Shortening
(227, 42, 68)	(228, 43, 68)	1	2	Shortening
(228, 42, 69)	(252, 42, 87)	24	3	Puncturing
	11, 12, 16, 32, 39, 60, 65, 72, 79, 82, 83, 109, 115, 120 123, 129, 138, 153, 195, 200, 201, 205, 207, 208			
(229, 42, 70)	(228, 42, 69)	1	3	Extending
(230, 42, 70)	(232, 44, 70)	2	2	Shortening
(231, 42, 70)	(234, 45, 70)	3	2	Shortening
(232, 42, 71)	(234, 44, 71)	2	2	Shortening
(233, 42, 72)	(235, 44, 72)	2	2	Shortening
(234, 42, 72)	(237, 45, 72)	3	1	Shortening
(235, 42, 73)	(252, 42, 87)	17	1	Puncturing
	8, 10, 24, 26, 30, 35, 49, 53, 63, 82 107, 113, 130, 131, 193, 196, 208			
(236, 42, 74)	(235, 42, 73)	1	1	Extending

Table 6 Best codes by modifying (255, 45, 87) BCH code ($43 \leq k \leq 45$).

Best code	Original code	m	d^+	Method
(222, 43, 65)	(223, 44, 65)	1	1	Shortening
(223, 43, 66)	(224, 44, 66)	1	2	Shortening
(224, 43, 66)	(225, 44, 66)	1	2	Shortening
(226, 43, 67)	(253, 43, 87)	27	2	Puncturing
	10, 12, 19, 32, 38, 47, 56, 63, 73, 78 82, 91, 104, 118, 125, 151, 155, 156, 161 167, 180, 191, 193, 198, 206, 207, 208			
(227, 43, 68)	(226, 43, 67)	1	2	Extending
(228, 43, 68)	(226, 43, 67)	2	2	Extending
(229, 43, 68)	(226, 43, 67)	3	2	Extending
(230, 43, 69)	(231, 44, 69)	1	2	Shortening
(231, 43, 70)	(232, 44, 70)	1	2	Shortening
(232, 43, 70)	(234, 45, 70)	2	2	Shortening
(233, 43, 71)	(234, 44, 71)	1	2	Shortening
(234, 43, 72)	(235, 44, 72)	1	2	Shortening
(235, 43, 72)	(237, 45, 72)	2	1	Shortening
(237, 43, 73)	(239, 45, 73)	2	1	Shortening
(238, 43, 74)	(240, 45, 74)	2	1	Shortening
(223, 44, 65)	(254, 44, 87)	31	1	Puncturing
	3, 7, 18, 23, 26, 27, 36, 48, 49, 51, 76, 81, 87 92, 98, 109, 117, 121, 126, 135, 136, 168, 169 177, 183, 184, 189, 191, 194, 197, 206			
(224, 44, 66)	(223, 44, 65)	1	2	Extending
(225, 44, 66)	(223, 44, 65)	2	2	Extending
(231, 44, 69)	(254, 44, 87)	23	2	Puncturing
	1, 2, 9, 25, 35, 44, 69, 90, 99, 107, 110, 118, 120 130, 132, 140, 149, 172, 176, 180, 200, 203, 206			
(232, 44, 70)	(231, 44, 69)	1	2	Extending
(233, 44, 70)	(234, 45, 70)	1	2	Shortening
(234, 44, 71)	(254, 44, 87)	20	2	Puncturing
	1, 7, 12, 34, 39, 65, 67, 76, 92, 110, 111, 120 127, 138, 177, 179, 184, 194, 196, 204			
(235, 44, 72)	(234, 44, 71)	1	2	Extending
(236, 44, 72)	(237, 45, 72)	1	1	Shortening
(238, 44, 73)	(239, 45, 73)	1	1	Shortening
(239, 44, 74)	(240, 45, 74)	1	1	Shortening
(233, 45, 69)	(255, 45, 87)	22	2	Puncturing
	1, 2, 4, 7, 8, 17, 19, 20, 28, 39, 50, 86, 105, 113 124, 135, 157, 171, 183, 190, 199, 209			
(234, 45, 70)	(233, 45, 69)	1	2	Extending
(235, 45, 70)	(233, 45, 69)	2	1	Extending
(236, 45, 71)	(255, 45, 87)	19	1	Puncturing
	1, 2, 3, 14, 16, 18, 30, 36, 44, 59, 69, 86 98, 140, 141, 142, 149, 158, 206			
(237, 45, 72)	(236, 45, 71)	1	1	Extending
(239, 45, 73)	(255, 45, 87)	16	1	Puncturing
	1, 2, 6, 18, 20, 26, 36, 38, 42 69, 72, 87, 180, 192, 196, 200			
(240, 45, 74)	(239, 45, 73)	1	1	Extending
(242, 45, 75)	(255, 45, 87)	13	1	Puncturing
	1, 2, 4, 22, 42, 68, 86, 89, 166, 171, 183, 190, 198			
(243, 45, 76)	(242, 45, 76)	1	1	Extending

original codes, the first through the m-th information bits are shortened.

6. Conclusion

In this paper, we proposed an algorithm for constructing the best codes using punctured codes and a supporting method for constructing the best codes. First of all, we defined the evaluation function for puncturing to determine which check bits to delete based on local weight distribution. We pro-

Table 7 Best codes by modifying (255, 47, 85) BCH code.

Best code	Original code	m	d^+	Method
(225, 43, 66)	(229, 47, 66)	4	2	Shortening
(226, 44, 66)	(229, 47, 66)	3	2	Shortening
(227, 44, 66)	(230, 47, 66)	3	1	Shortening
(228, 44, 67)	(231, 47, 67)	3	1	Shortening
(229, 44, 68)	(232, 47, 68)	3	2	Shortening
(230, 44, 68)	(233, 47, 68)	3	2	Shortening
(222, 45, 63)	(224, 47, 63)	2	1	Shortening
(223, 45, 64)	(225, 47, 64)	2	2	Shortening
(224, 45, 64)	(226, 47, 64)	2	2	Shortening
(225, 45, 64)	(227, 47, 64)	2	1	Shortening
(226, 45, 65)	(228, 47, 65)	2	1	Shortening
(227, 45, 66)	(229, 47, 66)	2	2	Shortening
(228, 45, 66)	(230, 47, 66)	2	2	Shortening
(229, 45, 67)	(231, 47, 67)	2	3	Shortening
(230, 45, 68)	(232, 47, 68)	2	3	Shortening
(231, 45, 68)	(233, 47, 68)	2	2	Shortening
(232, 45, 68)	(234, 47, 68)	2	2	Shortening
(223, 46, 63)	(224, 47, 63)	1	1	Shortening
(224, 46, 64)	(225, 47, 64)	1	2	Shortening
(225, 46, 64)	(226, 47, 64)	1	2	Shortening
(226, 46, 64)	(227, 47, 64)	1	2	Shortening
(227, 46, 65)	(228, 47, 65)	1	3	Shortening
(228, 46, 66)	(229, 47, 66)	1	3	Shortening
(229, 46, 66)	(230, 47, 66)	1	2	Shortening
(230, 46, 67)	(231, 47, 67)	1	3	Shortening
(231, 46, 68)	(232, 47, 68)	1	3	Shortening
(232, 46, 68)	(233, 47, 68)	1	2	Shortening
(233, 46, 68)	(234, 47, 68)	1	2	Shortening
(234, 46, 69)	(235, 47, 69)	1	2	Shortening
(235, 46, 70)	(236, 47, 70)	1	2	Shortening
(236, 46, 70)	(237, 47, 70)	1	1	Shortening
(237, 46, 71)	(238, 47, 71)	1	1	Shortening
(238, 46, 72)	(239, 47, 72)	1	1	Shortening
(240, 46, 73)	(241, 47, 73)	1	1	Shortening
(241, 46, 74)	(242, 47, 74)	1	2	Shortening
(242, 46, 74)	(243, 47, 74)	1	1	Shortening
(224, 47, 63)	(255, 47, 85)	31	1	Puncturing
	1, 6, 12, 18, 21, 35, 39, 42, 48, 52, 69, 77, 85, 86 103, 120, 122, 137, 144, 152, 154, 156, 158, 171 177, 183, 188, 196, 198, 205, 207			
(225, 47, 64)	(224, 47, 63)	1	2	Extending
(226, 47, 64)	(224, 47, 63)	2	2	Extending
(227, 47, 64)	(224, 47, 63)	3	2	Extending
(228, 47, 65)	(255, 47, 85)	27	3	Puncturing
	1, 3, 8, 18, 35, 52, 54, 69, 80, 86, 90, 91, 98, 103, 120, 121 136, 137, 142, 154, 157, 166, 171, 178, 188, 202, 205			
(229, 47, 66)	(228, 47, 65)	1	4	Extending
(230, 47, 66)	(228, 47, 65)	2	3	Extending
(231, 47, 67)	(255, 47, 85)	24	3	Puncturing
	1, 5, 18, 26, 35, 42, 52, 60, 61, 69, 74, 86, 103, 120 122, 129, 137, 153, 154, 160, 171, 188, 191, 205			
(232, 47, 68)	(231, 47, 67)	1	3	Extending
(233, 47, 68)	(231, 47, 67)	2	2	Extending
(234, 47, 68)	(231, 47, 67)	3	2	Extending
(235, 47, 69)	(255, 47, 85)	20	2	Puncturing
	1, 2, 10, 18, 49, 52, 69, 72, 83, 86, 103, 108, 120 134, 137, 154, 165, 171, 205, 208			
(236, 47, 70)	(235, 47, 69)	1	2	Extending
(237, 47, 70)	(235, 47, 69)	2	1	Extending
(238, 47, 71)	(255, 47, 85)	17	1	Puncturing
	1, 4, 13, 35, 52, 69, 85, 86, 103, 120 136, 137, 154, 171, 188, 205, 207			
(239, 47, 72)	(238, 47, 71)	1	1	Extending
(241, 47, 73)	(255, 47, 85)	14	1	Puncturing
	1, 2, 18, 35, 52, 69, 86, 103, 120, 137, 154, 171, 188, 205			
(242, 47, 74)	(241, 47, 73)	1	2	Extending
(243, 47, 74)	(241, 47, 73)	2	1	Extending

posed an algorithm for constructing the punctured code with a large minimum Hamming distance using the backtracking method based on this evaluation function. It is especially effective when cyclic codes and the shortened codes are used as the original code of the punctured code because their local weight distribution can be derived by the k-sparse algorithm [6] or proposed extended k-sparse algorithm.

Secondly, we proposed the method to evaluate the possibility of constructing the new best code by analyzing the information registered in the database. We can easily choose target (n, k) codes to search by using this method. By searching for the (n, k) codes with high evaluation values, we were able to construct 167 new best codes.

Our newly constructed best codes are guaranteed to be the best codes by deriving the minimum weights using the proposed extended k-sparse algorithm.

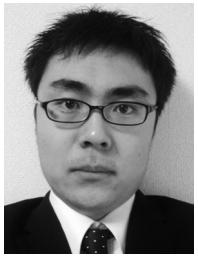
Acknowledgments

This work was supported by JSPS KAKENHI Grant Number JP20K11810.

References

[1] M. Grassl, "Code tables: Bounds on the parameters of various types of codes," http://codetables.de, accessed June 2021.
[2] C. Ding, C. Fan, and Z. Zhou, "The dimension and minimum distance of two classes of primitive BCH codes," Finite Fields Th. App., vol.45, pp.237–263, May 2016.
[3] Y. Kageyama and T. Maruta, "On the geometric constructions of optimal linear codes," Des. Codes Cryptogr., vol.81, no.3, pp.469–480, 2016.
[4] N. Aydin and D. Foret, "New linear codes over $GF(3)$, $GF(11)$, and $GF(13)$," Journal of Algebra Combinatorics Discrete Structures and Applications, vol.6, no.1, pp.13–20, 2019.
[5] A.M. Berg and I.I. Dumer, "On computing the weight spectrum of cyclic codes," IEEE Trans. Inf. Theory, vol.38, no.4, pp.1382–1386, 1992.
[6] Z. Li, M. Mohri, and M. Morii, "An efficient method for computing the minimum weight of high rate binary cyclic codes," ISITA2006, pp.741–746, Nov. 2006.
[7] J. Zwanzger,"A heuristic algorithm for the construction of good linear codes," IEEE Trans. Inf. Theory, vol.54, no.5, pp.2388–2392, 2008.
[8] H.A. Priestley and M.P. Ward, "A multipurpose backtracking algorithm," J. Symb. Comput., vol.18, no.1, pp.1–40, July 1994.
[9] T. Ohara, M. Takita, and M. Morii, "A construction of binary punctured linear codes and a supporting method for best code search," ISITA2020, pp.170–174, Oct. 2020.

Takuya Ohara received his B.E. degree from Kobe University, Japan in 2020. He is currently pursuing Master's degree at Kobe University. His research interests include error correcting codes and their applications.



Makoto Takita received his B.E., M.E., and D.E. degrees from Kobe University, Japan, in 2014, 2015, and 2018, respectively. In 2018, he was a Researcher at the Graduate School of Engineering, Kobe University, Japan. Since 2019, he has been an Assistant Professor at the School of Social Information Science, University of Hyogo, Japan. His research interests include coding theory, information networks, and information security.

Masakatu Morii received his B.E. degree in electrical engineering and his M.E. degree in electronics engineering from Saga University, Saga, Japan, and his D.E. degree in communication engineering from Osaka University, Osaka, Japan in 1983, 1985, and 1989, respectively. From 1989 to 1990, he was an Instructor in the Department of Electronics and Information Science, Kyoto Institute of Technology, Japan. From 1990 to 1995, he was an Associate Professor in the Department of Computer Science, Faculty of Engineering at Ehime University, Japan. From 1995 to 2005, he was a Professor in the Department of Intelligent Systems and Information Science, Faculty of Engineering, at the University of Tokushima, Japan. Since 2005, he has been a Professor in the Department of Electrical and Electronics Engineering, Faculty of Engineering, at Kobe University, Japan. His research interests include error correcting codes, cryptography, discrete mathematics, computer networks, and information security. He is a member of the IEEE.