Heuristic Algorithms for Constructing Binary Constant Weight Codes

Roberto Montemanni  Derek H. Smith

Abstract—Constant weight binary codes are used in a number of applications. Constructions based on mathematical structure are known for many codes. However, heuristic constructions unrelated to any mathematical structure can become of greater importance when the parameters of the code are larger.

This paper considers the problem of finding constant weight codes with the maximum number of codewords from a purely algorithmic perspective. A set of heuristic and metaheuristic methods is presented and developed into a variable neighbourhood search framework. The proposed method is applied to 383 previously studied cases with lengths between 29 and 63. For these cases it generates 153 new codes, with significantly increased numbers of codewords in comparison with existing constructions. For 10 of these new codes the number of codewords meets a known upper bound, and so these 10 codes are optimal. As well as the ability to generate new best codes, the approach has the advantage that it is a single method capable of addressing many sets of parameters in a uniform way.

Index Terms—Clique search, constant weight binary codes, heuristic algorithms, seed building, variable neighbourhood search.

I. INTRODUCTION

A constant weight binary code is a set of binary vectors of length \( n \), weight \( w \) and minimum Hamming distance \( d \). The weight of a binary vector (or codeword) is the number of 1’s in the vector. The Hamming distance \( d(x,y) \) between two vectors \( x \) and \( y \) is the number of positions in which they differ. The minimum distance of a code is the minimum Hamming distance between any pair of codewords. The maximum possible number of vectors in a constant weight code is usually referred to as \( A(n,d,w) \). The problem of determining \( A(n,d,w) \) is equivalent to the packing problem: determine the maximum number \( D(t,k,v) \) of \( k \)-subsets of a \( v \)-set such that no \( t \)-subset is covered more than once [3], [15].

Apart from their important role in the theory of error-correcting codes [14], constant weight codes have also found application in fields as diverse as the design of demultiplexers for nano-scale memories [13], the construction of frequency hopping lists for use in GSM networks [17] and the design of DNA codes for use in DNA barcoding and DNA computing [12].

Accounts of the theory of constant weight codes can be found in [14], [4]. A detailed account of upper bounds for \( A(n,d,w) \) can be found in [1]. Lower bounds for \( A(n,d,w) \) are usually obtained constructively. Code constructions for \( n \leq 28 \) can be found in [4]. In [19] a comprehensive set of constructions was described for the parameter sets appropriate to the frequency hopping application. These parameter sets were \( 29 \leq n \leq 63 \) and \( 5 \leq w \leq 8 \) with \( d = 2w - 2 \), \( d = 2w - 4 \) and \( d = 2w - 6 \). A small number of improvements to the values in [19] can be found in [9], [8], [20] and [10]. Heuristic constructions of constant weight codes (which do not attempt to identify or store any mathematical structure) are useful in applications. For longer codes they often give the best known code, particularly when no good mathematical structure has been identified. However, these mechanisms must be well designed if they are to be effective. Simply applying massive computer power without a good algorithm is unlikely to lead to a good code. Concerning metaheuristic algorithms, both simulated annealing [7] and tabu search [2] have been used to construct constant weight codes, although the number of results presented in [7], [2] is small. The methods presented here significantly improve these results.

It can be noted from the results in [19] that when no good mathematical construction is available and methods from [4] using permutation groups cease to be feasible in a reasonable run time, then lexicographic search becomes the default method. It usually performs reasonably well, and is certainly much more effective than random search [19]. There are a number of variations of lexicographic search, including the use of reverse lexicographic search and the use of seed vectors. Thus lexicographic search forms the basis of the first method presented here. In small cases clique search can also be used to find good constant weight codes. In the second method presented here clique search is used in a way which restricts the search to feasible partial problems. The two methods can then be combined in a variable neighbourhood search framework [11]. This variable neighbourhood framework diversifies the search and, together with the efficiency of the component local searches, accounts for the performance of the method.

Section II describes the development of the local search algorithms; seed building in II-A and clique search in II-B. In Section II-C the local search algorithms are developed into a variable neighbourhood search metaheuristic, and the results from the new algorithm are compared with the best of the simple local search algorithms. In Section III the 383 cases from [19] for which no optimal code is known are studied, the numbers of codewords of new best codes found are tabulated and these results compared with known upper bounds. Ten of the new codes are optimal. Finally, Section IV presents conclusions.
II. LOCAL SEARCH ALGORITHMS

In this section two families of local search algorithms for maximizing the number of codewords in a constant weight code are described. They improve the results of standard techniques by making use of novel heuristic mechanisms.

A. Seed building

Forward lexicographic order of binary vectors is a dictionary order starting at 00...011...1. Vector \( x = x_1 \ldots x_n \) is listed before \( y = y_1 \ldots y_n \) if \( x_i < y_i \), where \( i \) is the first position in which the two vectors differ. Thus for \( w = 2 \) and \( n = 4 \) the order is 0011,0101,0110,1001,1010,1100. Reverse lexicographic order is the reverse of this order.

As observed in the introduction, algorithms that examine all possible codewords in lexicographic order or reverse lexicographic order and incrementally accept codewords that are feasible with respect to already accepted ones, can often produce fairly good codes [19], [9]. Sometimes they can produce very good codes [6]. For this reason the first method presented here is build on these orderings, combined with the concept of seed codewords [4]. These seed codewords are an initial set of codewords of weight \( w \) to which codewords are added in the given ordering if they satisfy the minimum distance criterion. Given a set of seed codewords, the set of codewords incrementally added by a given ordered search may change radically.

Seeds can be selected at random, but some computational results (see [16]) clearly show that there is a better way to proceed. The novelty of the approach proposed here resides in the dynamic evolution of the set of seed codewords, obtained using a heuristic mechanism. In this seed building algorithm a set of seeds is initially empty, and one random seed is added at a time. If this seed leads to good results it is kept, and a new random seed is designated for testing, increasing the size of the seed set. The same rationale is used to decide whether to keep subsequent seeds or not. In the same way, if after a given number of iterations the quality of the solutions provided by a set of seeds is judged to be not good enough, the most recent seed is eliminated from the set, which results therefore in a reduction in size of the seed set. In this way the set of seeds is expanded or contracted depending on the quality of the solutions provided by the set itself. What happens in practise is that the size of the seed set oscillates through a range of small values.

Pseudo-code for the seed building method (SB) is provided in Figure 1. BestSol and WorkSol represent respectively the best solution retrieved so far and the working solution. SelOrd is a parameter describing the sequence in which binary vectors are examined (i.e. forward lexicographic, reverse lexicographic or random). SeedSet is the set of seed codewords, while ItrCnt is an iteration counter and ItrSeed is a parameter indicating for how long (in terms of number of iterations) each new seed is tested. CWord is a random codeword used to extend the partial code contained in SeedSet. The algorithms works in an iterative fashion on an adaptive set of seed codewords contained in the set SeedSet, which is initially empty. At each iteration, a new codeword CWord, compatible with those in SeedSet, is generated and the partial solution WorkSol, initialized with the elements of SeedSet and CWord, is expanded by adding feasible binary vectors, examined according to the order defined by parameter SelOrd. If a new best solution is found, the set SeedSet is immediately expanded. Every ItrSeed iterations, the average size of the codes generated with the set SeedSet is judged to be not good enough, the most recently added seed. The procedure stops after a fixed computation time \( Time_{SB} \) has been reached.

B. Clique search

The idea at the basis of this local search method is that it is possible to complete, in the best possible way, a partial code by solving a maximum clique problem [18]. The construction of constant weight binary codes by solving maximale clique problems is not a new idea, but the iterative application to partial codes presented here is novel. The advantage is that the maximum clique problem can be kept to a manageable size for large problems. More precisely, given a code, a random subset of the codewords of the code is removed, leaving a partial code. It is now possible to identify all the codewords
CliqueSearch(Time\textsubscript{CS}, BestSol, CSRem, Time\textsubscript{MC})

While (computation time < Time\textsubscript{CS})

WorkSol := BestSol;

Delete \(|\{\text{WorkSol}\}| - \text{CSRem}/100\) random codewords from WorkSol;

Build the graph \(G = \{V, E\}\) where:

\(V = \{\text{all codewords of weight } w\}\)

\(E = \{(i, j) | \text{dist}(i, j) \geq d\}\); Let Clique be a solution of the maximum clique problem on \(G\) corresponding to a set of codewords. The maximum computation time is Time\textsubscript{MC} seconds;

WorkSol := WorkSol \cup Clique;

If (|WorkSol| > |BestSol|)

BestSol := WorkSol;

EndIf;

EndWhile;

Fig. 2. The Clique Search algorithm.

of weight \(w\) compatible with those already in the code, and build a graph from these codewords, where codewords are represented by vertices. Two vertices are connected if and only if the Hamming distance between the codewords is at least \(d\). It is then possible to run a maximum clique algorithm on the graph in order to complete the partial code in the best possible way. Heuristic or exact methods can be used to solve the maximum clique problem. Here the exact algorithm presented by Carraghan and Pardalos [5] is used. It is important that the search remains feasible, so if the computation time is greater than the parameter Time\textsubscript{MC}, the execution is truncated and the largest clique retrieved so far is used. The algorithm presented in [5] has the advantage that it is fast enough to frequently give an exact solution, but may still give a good heuristic solution if truncation is necessary.

Pseudo-code for the Clique Search (CS) method is provided in Figure 2. BestSol and WorkSol represent respectively the best solution retrieved so far, and the working solution. Parameter CSRem represents the target percentage of codewords that have to be deleted from the solution WorkSol before it is reconstructed by the maximum clique method described above. In practice the codewords deleted are selected randomly and CSRem is the mean percentage of codewords deleted. The Clique Search algorithm is based on an iterative mechanism and runs until a given computation time, regulated by parameter Time\textsubscript{CS}, is reached.

C. Variable neighbourhood search

Variable neighbourhood search (VNS) methods have been demonstrated to perform well and are robust (see [11]). Such algorithms work by applying different local search algorithms one after the other, aiming at differentiating the characteristics of the search-spaces visited (i.e. changing the neighbourhood).

In the current context, there are two families of local searches, with a total of four possible methods. Three of them are Seed Building procedures (as discussed in Section II), where a sample of the possible different orders for the examination of codewords are considered: forward lexicographic order, reverse lexicographic order and random order. These three orders were evaluated and were considered sufficiently representative to differentiate the variable neighbourhood search. The fourth component of the variable neighbourhood search is the Clique Search algorithm. The Seed Building type of algorithm is alternated with the Clique Search algorithm, each time starting from the best solution retrieved since the beginning. The ordering used for Seed Building is selected randomly, according to chosen probability parameters for the orders. The aim is to obtain better solutions than those retrieved by running the single local search procedures alone. The complete metaheuristic algorithm is summarized by the pseudo-code presented in Figure 3. VNS runs for Time\textsubscript{VNS} seconds. \(P_{\text{REV}}, P_{\text{FWD}}\) and \(P_{\text{RND}}\) represent the probabilities that codewords are examined in reverse lexicographic order, forward lexicographic order or random order in each run of the Seed Building procedure. The other parameters are those used by the inner local search procedures.

D. Computational experiments comparing VNS with basic local searches

In this section a set of computational experiments is described which aims to achieve an understanding of whether the metaheuristic approach described in this section is capable of improving the results of the basic local search methods described in Section II. The algorithms are run on a small, but representative subset of problems. The average, the best and the worst results obtained over ten runs are reported. The experiments discussed in this section have been carried out on an Intel Core Duo 2.0 GHz / 1 GB RAM machine. All the methods have been coded in ANSI C. All computation times reported in the remainder of this paper are expressed in seconds.

Table I is organized by blocks of rows. The first column is used to identify the problem, and there are two rows for every problem, one for each of the following methods:

- \textit{Best}_{LS}: The best results obtained by all the local search methods discussed in Section II. Notice that not all the results reported on a single line have necessarily been obtained by the same algorithm. In each case it is the best result obtained. For example, given a problem, the result
Table I: Results for metaheuristic algorithms.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Method</th>
<th>Average</th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(29,4,5)</td>
<td>VNS</td>
<td>3804.7</td>
<td>3816</td>
<td>3792</td>
</tr>
<tr>
<td>time = 1600</td>
<td></td>
<td>3806.6</td>
<td>3832</td>
<td>3787</td>
</tr>
<tr>
<td>A(29,6,5)</td>
<td>VNS</td>
<td>251.8</td>
<td>254</td>
<td>251</td>
</tr>
<tr>
<td>time = 90</td>
<td></td>
<td>253.0</td>
<td>254</td>
<td>250</td>
</tr>
<tr>
<td>A(29,6,6)</td>
<td>VNS</td>
<td>868.4</td>
<td>872</td>
<td>866</td>
</tr>
<tr>
<td>time = 1180</td>
<td></td>
<td>873.5</td>
<td>881</td>
<td>862</td>
</tr>
<tr>
<td>A(29,8,5)</td>
<td>VNS</td>
<td>31.5</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>time = 15</td>
<td></td>
<td>32.3</td>
<td>33</td>
<td>31</td>
</tr>
<tr>
<td>A(29,10,7)</td>
<td>VNS</td>
<td>36.8</td>
<td>37</td>
<td>36</td>
</tr>
<tr>
<td>time = 180</td>
<td></td>
<td>36.8</td>
<td>38</td>
<td>36</td>
</tr>
<tr>
<td>A(45,4,5)</td>
<td>VNS</td>
<td>24803.1</td>
<td>24975</td>
<td>2489</td>
</tr>
<tr>
<td>time = 3260</td>
<td></td>
<td>24914.9</td>
<td>24975</td>
<td>24587</td>
</tr>
<tr>
<td>A(45,6,5)</td>
<td>VNS</td>
<td>1025.5</td>
<td>1037</td>
<td>1015</td>
</tr>
<tr>
<td>time = 190</td>
<td></td>
<td>1035.0</td>
<td>1039</td>
<td>1031</td>
</tr>
<tr>
<td>A(45,6,6)</td>
<td>VNS</td>
<td>5935.4</td>
<td>5937</td>
<td>5934</td>
</tr>
<tr>
<td>time = 2400</td>
<td></td>
<td>5956.7</td>
<td>5937</td>
<td>5935</td>
</tr>
<tr>
<td>A(45,8,5)</td>
<td>VNS</td>
<td>73.7</td>
<td>79</td>
<td>76</td>
</tr>
<tr>
<td>time = 40</td>
<td></td>
<td>78.7</td>
<td>80</td>
<td>78</td>
</tr>
<tr>
<td>A(45,10,7)</td>
<td>VNS</td>
<td>169.1</td>
<td>170</td>
<td>169</td>
</tr>
<tr>
<td>time = 400</td>
<td></td>
<td>169.9</td>
<td>170</td>
<td>169</td>
</tr>
</tbody>
</table>

Table II: Comparison with simulated annealing and tabu search.

<table>
<thead>
<tr>
<th>Problem</th>
<th>VNS</th>
<th>SA</th>
<th>TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(22,10,9)</td>
<td>32</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>A(23,10,7)</td>
<td>19</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>A(23,10,8)</td>
<td>29</td>
<td>28</td>
<td>21</td>
</tr>
<tr>
<td>A(23,10,9)</td>
<td>41</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>A(23,11,11)</td>
<td>56</td>
<td>39</td>
<td>37</td>
</tr>
<tr>
<td>A(24,10,8)</td>
<td>35</td>
<td>33</td>
<td>25</td>
</tr>
<tr>
<td>A(24,10,9)</td>
<td>52</td>
<td>24</td>
<td>36</td>
</tr>
</tbody>
</table>

reported in the column “Best” might have been obtained by a Seed Building method, while that reported in the column “Average” might come from the Clique Search algorithm. Detailed computational results for local search methods can be found in [16].

- VNS: The Variable Neighbourhood Search method presented in Section II-C.

For each row (algorithm), the average, the best and the worst results over ten runs are reported together with the computation time allowed for the method. When the local search methods are tested independently from the VNS, each individual method is run for the same time as that allowed for the complete VNS algorithm. After some tuning experiments, the values of the remaining parameters were selected as ItrSeed = 20 (Seed Building methods) CSRem = 20, Time_MC = 30 (Clique Search). Other parameters were typically $P_{REV} = 0.55$, $P_{FWD} = 0.35$, and $P_{RND} = 0.1$ except for the case $w = 8, d = 14$ where $P_{REV} = 1.0$ was used. Many of the new best results were obtained by a programme of long runs of the VNS algorithm. For these runs the time values (in seconds) depended on $n$: TimeSB = 505 · $(n/29)^2$, TimeCS = 505 · $(n/29)^2$, TimeVNS = 16060 · $(n/29)^5$ were used. These very long run times were barely adequate to allow seed building to work effectively for $n \geq 60$ but were probably excessive for many smaller values of $n$, where the best result was found early in the run. The results have been obtained on a selection of processors with similar characteristics: Dual AMD Opteron 250 2.4GHz with 4GB RAM, Dual Intel Xeon 2.66 GHz with 4GB RAM and Intel Pentium 4 2.5GHz with 1GB RAM.

These results are summarized in Table III. Problems are identified in the first column of the table, while the previous best known lower bounds (Old LB), the new lower bounds (New LB) and the best upper bounds available (UB) are reported in the remaining columns. Bold entries denote new optimal codes. Old lower bounds are all from [19] except for A(32,6,6), A(38,8,6), A(39,8,6), that are taken from http://www.research.att.com/~njas/codes/Andw/. The upper bounds contain some improvements to the upper bounds stated in [19], which simply quoted the first Johnson bound. The value quoted in the current paper is the smallest of (i) the first Johnson bound [4], (ii) the second Johnson bound [4], (iii) equation (18) of [4], (iv) equations (25)-(27) of [1] and (v) Theorems 12 and 13 of [1]. In fact for the parameter sets studied here the first or second Johnson bound always gives the best upper bound, except in the 5 cases (n,d,w)=(53,6,5), (47,8,7), (56,10,6), (62,10,7) and (38,10,8). In these cases Theorems 12 and 13 of [1] together with non-existence of employed for the different algorithms, since VNS obtains improved solutions very quickly.

III. New Table of Constant Weight Binary Codes

The metaheuristic algorithm presented in Section II-C gives substantial improvements to the known lower bounds for many cases within the parameter ranges studied: A(n,4,5), A(n,6,5), A(n,6,6), A(n,8,5), A(n,8,6), A(n,8,7), A(n,10,6), A(n,10,7), A(n,10,8), A(n,12,7), A(n,12,8) and A(n,14,8), with $29 \leq n \leq 63$. In the table which follows the best lower bounds obtained during the development of the local search and VNS algorithms are reported. Typical settings for the VNS algorithm were ItrSeed = 20 (Seed Building), CSRem = 20, Time_MC = 30 (Clique Search). Other parameters were typically $P_{REV} = 0.55$, $P_{FWD} = 0.35$, and $P_{RND} = 0.1$ except for the case $w = 8, d = 14$ where $P_{REV} = 1.0$ was used. Many of the new best results were obtained by a programme of long runs of the VNS algorithm. For these runs the time values (in seconds) depended on $n$: TimeSB = 505 · $(n/29)^2$, TimeCS = 505 · $(n/29)^2$, TimeVNS = 16060 · $(n/29)^5$ were used. These very long run times were barely adequate to allow seed building to work effectively for $n \geq 60$ but were probably excessive for many smaller values of $n$, where the best result was found early in the run. The results have been obtained on a selection of processors with similar characteristics: Dual AMD Opteron 250 2.4GHz with 4GB RAM, Dual Intel Xeon 2.66 GHz with 4GB RAM and Intel Pentium 4 2.5GHz with 1GB RAM.

These results are summarized in Table III. Problems are identified in the first column of the table, while the previous best known lower bounds (Old LB), the new lower bounds (New LB) and the best upper bounds available (UB) are reported in the remaining columns. Bold entries denote new optimal codes. Old lower bounds are all from [19] except for A(32,6,6), A(38,8,6), A(39,8,6), that are taken from http://www.research.att.com/~njas/codes/Andw/. The upper bounds contain some improvements to the upper bounds stated in [19], which simply quoted the first Johnson bound. The value quoted in the current paper is the smallest of (i) the first Johnson bound [4], (ii) the second Johnson bound [4], (iii) equation (18) of [4], (iv) equations (25)-(27) of [1] and (v) Theorems 12 and 13 of [1]. In fact for the parameter sets studied here the first or second Johnson bound always gives the best upper bound, except in the 5 cases (n,d,w)=(53,6,5), (47,8,7), (56,10,6), (62,10,7) and (38,10,8). In these cases Theorems 12 and 13 of [1] together with non-existence of
Table III contains 153 new codes, ten of which are new optimal codes. Note that it was erroneously stated in [9] that $A(45,14,8)=14$ is optimal. In fact the second Johnson bound is 15 and the code found here has 15 codewords.

IV. CONCLUSIONS

The algorithmic generation of constant weight binary codes has been considered from a heuristic perspective, without consideration of specific mathematical constructions. Local search and metaheuristic methods have been presented and discussed. The novel methodologies produced codes which improved the best known lower bounds for 153 cases with $29 \leq n \leq 63$, i.e. for 39.95% of the parameter sets considered. In particular, 10 new optimal codes are obtained which were previously unknown.

REFERENCES


Roberto Montemanni has received the Laurea degree in Computer Science from the University of Bologna, Italy, in 1999 and the Ph.D. in Applied Mathematics from the University of Glamorgan, U.K. in 2002. He is now a Senior Researcher at Istituto Dalle Molle di Studi sull’Intelligenza Artificiale (IDSIA), Lugano, Switzerland, where he has been working since 2001. His interests include the development of exact and heuristic algorithms for optimization problems.

Derek Smith has received B.Sc. and Ph.D. degrees from the University of Southampton, U.K., and the D.Sc. degree from the University of Glamorgan, U.K. He is Professor of Mathematics at the University of Glamorgan, where he has worked since 1971. His research interests include radio frequency assignment, graph theory, coding theory, data compression, and network reliability studies.