Journal of Universal Computer Science, vol. 5, no. 12 (1999), 817-827
submitted: 2/9/99, accepted: 2/12/99, appeared: 28/12/99 © Springer Pub. Co.

A Note on Bounded-Weight Error-Correcting Codes

Russell Bent, Michael Schear, and Lane A. Hemaspaandra
(Department of Computer Science
University of Rochester
Rochester, NY 14627
{rbent,mschear,lane}@cs.rochester.edu)

Gabriel Istrate
(Center for Nonlinear Studies and CIC-3 Division
Los Alamos National Laboratory
MS 258, Los Alamos, NM 87545.
gistrate@cnls.lanl.gov)

Abstract: This paper computationally obtains optimal bounded-weight, binary, error-correcting codes for a variety of distance bounds and dimensions. We compare the sizes of our codes to the sizes of optimal constant-weight, binary, error-correcting codes, and evaluate the differences.

Key Words: error-correcting codes, bounded-weight codes, constant-weight codes, experimental algorithms, heuristic algorithms, exact solutions.

Category: H.1.1, E.4.

1 Introduction

One goal of coding theory is to construct classes of codes having optimal size. Studies have investigated versions of this problem for classes of codes with various regularity properties, such as linear codes over finite fields [Brouwer], binary self-dual codes [Conway, Pless and Sloane (92)], mixed binary-ternary codes [Brouwer et al. (97)], and various classes of spherical codes [Sloane].

Two such important cases concern determining the values of $A(n,d)$ and $A(n,d,w)$, where $A(n,d)$ is the number of codewords in the largest binary code of length n having minimum distance d, and $A(n,d,w)$ is the number of codewords in the largest binary code of length n, minimum distance d, and weight w. Optimal values for $A(n,d)$ and $A(n,d,w)$ have been tabulated in [Litsyn, Rains and Sloane] and [Rains and Sloane], respectively.

It is conceivable that significant improvements in optimal code size could be obtained by relaxing the restriction on the code weight in the definition of $A(n,d,w)$ from "*equal* to w" to "*upper-bounded* by w," because there would then be a greater number of words potentially available for inclusion in the codes. We present optimal, bounded-weight, binary, error-correcting codes for a variety of distance bounds and dimensions. The method we employ to obtain the optimal codes is based on the observation that finding optimal bounded-weight codes can be transformed to finding the size of a maximum clique in a suitably defined graph. The clique-finding is accomplished primarily using the branch and bound search used in [Brouwer et al. (97)], (see also [Applegate and Johnson] and the discussion later in this paper).

2 Preliminaries

Let F be some finite set of characters—the *alphabet*. A word of length n over F is an element of F^n. A code over F of size n is a set of words of length n over F. A code over the alphabet $\{0, 1\}$ is called *binary*. Throughout this paper, we use the alphabet $F = \{0, 1\}$.

The distance, d, of a code is the smallest Hamming distance between any two codewords in the code. If we have two codewords, x and y, both of length n, we can represent these two words as $x_1x_2x_3 \cdots x_n$ and $y_1y_2y_3 \cdots y_n$, where x_j is the j^{th} bit in x. The Hamming distance between x and y is the size of the set, $\{j : 1 \leq j \leq n \wedge x_j \neq y_j\}$. The weight, w, of a binary word, x, is equal to the number of 1s in x. For a constant-weight (w) code, every word in the code has the same weight, w. In a bounded-weight (w) code, every word has at most w ones.

The standard reduction of finding optimal values of $A(n, d)$ and $A(n, d, w)$ to the problem of determining a maximum clique in a graph is as follows. The graph's vertices represent binary strings of length n (and legal weight, when appropriate). Two vertices are joined by an edge if and only if their Hamming distance is at least d.

It is easily seen that the connection between optimal code size and maximum clique in a suitably constructed graph carries over to the case of bounded-weight codes, and we indeed use exactly that in this paper.

3 Results and Discussion

The constant-weight bounds, many tight, tabulated by Sloane were obtained from a variety of sources and methods [Rains and Sloane]. An elegant method for finding optimal codes of constant-weight is to use an algebraic formula. Methods of creating such formulas for certain cases are presented in [Brouwer et al. (90)]. No such algebraic formulas for instances of bounded-weight codes are available yet. In the absence of such a method we tried various other methods for obtaining good sets of codewords. Many of the algorithms used were bounded-weight variants of those suggested in the literature for calculating good constant-weight codes. These methods included simulated annealing [El Gamel et al. (87)], genetic algorithms [Vaessens, Aaarts and van Lint (93)], and a randomized greedy heuristic search. The codes generated by these methods were beaten or equaled by our final method of obtaining codes, which was creating an appropriate graph and seeking a large (in fact, usually maximum-size) clique via different clique-finding algorithms.

Since the problem of finding a maximum clique in a graph has been thoroughly investigated [Johnson and Trick (93)], it is natural to use a reduction to this problem as our basis for finding good bounded-weight codes. The reduction is accomplished by creating the graph of possible codewords acceptable under the parameters for length and weight. Each possible codeword is represented by a vertex in the graph. If two codewords have a proper Hamming distance, then an edge is placed between them. The largest clique in the graph is representative of a maximum set of codewords such that the set meets all the parameters.

We used two clique-finding algorithms suggested in [Brouwer et al. (97)]. The first algorithm is a basic branch and bound search. In the worst case, it will search all possible combinations of nodes for cliques, but in practice it keeps track of a best solution and travels only those paths that have the potential to beat the current best solution. This algorithm will always find a maximum-size clique. We used a publicly available coding from [Applegate and Johnson], (see also [Carragan and Pardalos (90)]). The second algorithm is a variant of semi-exhaustive greedy search. This algorithm may not always find the largest clique. The algorithm begins by creating two sets of nodes. The first set is nodes that are part of the clique being created and the second set is nodes that can be added to the clique set without disrupting the clique property of the set. This available node set initially contains all the nodes and the clique set is initially empty. A node is chosen from the nodes in the available set. Those nodes that are not connected to the chosen node are eliminated from the available set. This process is repeated until the number of nodes in the available set drops below a user-defined threshold, y. Once y is reached, the branch and bound algorithm is employed on the available set. The nodes are selected as follows. For a user-defined number x, x nodes are chosen at random from the available node set. The node with the most edges in the set of x nodes is chosen. We used a publicly available coding, originally by Johnson, as modified by Applegate and Johnson (see [Applegate and Johnson], also [Johnson et al. (91)]). For our purposes, good results were achieved when $x = 0.1s$, where s is the number of nodes in the original graph, and $y = 100$. We ran the algorithm a thousand times in order to increase the odds of finding the largest clique.

The branch and bound algorithm was used on parameters where the optimal constant-weight code sizes were known and the search spaces were small enough to allow results to be obtained in reasonable amounts of time. For example, it took forty-one CPU minutes to calculate $A(9,4,4)$ and this was considered reasonable. On the other hand, the calculation of $A(9,4,7)$ was terminated as it was taking an unreasonable amount of time. However, running the greedy algorithm one thousand times on $A(9,4,7)$ took just under seventy two CPU minutes.[1]

From our results, it is now clear that, with regards to changing from constant-weight to bounded-weight, there is little or no increase in number of codewords in the best code until constant-weight codes become handicapped with a decrease in search space. (As the weight of a constant-weight code increases, the search space increases initially, but then begins to decrease once $w > \lceil \frac{n}{2} \rceil$. However, in the case of bounded-weight codes, the search space continues to increase as w approaches n.) It is important to note that where there are increases in the number of words in bounded-weight codes over constant-weight codes, these new bounded-weight codes can often be obtained trivially. For example, if $w \geq d$, a bounded-weight code can be created by taking the constant-weight code at $A(n,d,w), w \geq d$ and adding the word of all 0s. This is because the word of all 0s has a Hamming distance at least d from all the words in the constant-weight code $A(n,d,w)$, when $w \geq d$. Other bounded-weight codes can be created in this manner by patching together known constant-weight codes.

[1] These CPU times were obtained using a Sun Ultra 10.

Length (n)	Weight (w)	Constant Weight	Bounded Weight
6	3	4	4
6	4	3	4
6	5	1	4
6	6	1	4
7	3	7	7
7	4	7	8
7	5	3	8
7	6	1	8
7	7	1	8
8	3	8	8
8	4	14	15
8	5	8	15
8	6	4	16
8	7	1	16
8	8	1	16
9	3	12	12
9	4	18	19
9	5	18	$19^{\star}$
9	6	12	$19^{\star}$
9	7	4	$19^{\star}$
9	8	1	$20^{\star}$
9	9	1	$20^{\star}$
10	3	13	13
10	4	30	$31^{\star}$
11	6	66	$71^{\star}$

Table 1: *Code sizes for distance 4.* Note: The values superscripted with "$\star$" were obtained through greedy search.

Clearly, a lower bound for bounded-weight codes is

$$\max_{m:0\leq m<d}\left(\sum_{j:0\leq j\leq w\ \wedge\ (j\equiv m\ (mod\ d))} A(n,d,j)\right).$$

Results from the two clique-finding algorithms seem to usually merely meet this bound, and occasionally (see discussion below) beat it. Tables 1, 2, and 3 illustrate these results. It must be noted that the performance of the semi-exhaustive search has only been tested on those parameters where the entire graph can be created and stored in memory. It remains to be seen if patched codes can be matched or beaten easily in other cases.

We now discuss more broadly our results. As noted above, in most cases the best bounded-weight codes we obtain are in fact such that codes of optimal sizes are also provided by "patching together" existing optimal constant-weight codes. However, this does not mean that that part of our paper makes no contribution. Before our paper, it remained possible that there existed bounded-weight codes for these cases having size larger than the patched-together codes. Our paper, via in many cases (namely, in all table lines other than the nine superscripted with

Length (n)	Weight (w)	Constant Weight	Bounded Weight
8	4	2	2
8	5	2	2
8	6	1	2
8	7	1	2
8	8	1	2
9	4	3	3
9	5	3	4
9	6	3	4
9	7	1	4
9	8	1	4
9	9	1	4
10	4	5	5
10	5	6	6
10	6	5	6
10	7	3	6
10	8	1	6
10	9	1	6
10	10	1	6
12	6	22	$23^{\star}$

Table 2: *Code sizes for distance 6.* Note: The value superscripted with "$\star$" was obtained through greedy search.

asterisks) establishing the maximum size achievable by *any* legal code obeying the parameters, removes this possibility. Additionally, our work shows that in some cases the obvious patching together that we mention does not achieve a maximum-sized code. For example, the size 16 code obtained for $A(8,4,6)$ is such a case (as, since $A(8,4,2)$ obviously is exactly 4, the relevant patched-together codes are of size $8+4$ and of size $14+1$, and thus both fall short of size 16).

We now turn to the question of whether, in light of our results, bounded-weight codes seem wise to use. Bounded-weight codes obviously give no fewer codewords (in a maximum-sized code) that their sister constant-weight codes. Our tables show that in many cases they give strictly more words. Of course, as w increases beyond $\lfloor n/2 \rfloor$ the size of the word-space of bounded-weight codes becomes extremely rich relative to that of constant-weight codes (which starting at weight $\lceil n/2 \rceil$ have contracting word-spaces as w increases), and even for smaller (but nonzero) values of w their word space is of course richer—which is exactly what opens up the possibility of larger-sized codes.

However, this does not necessarily mean that it is wise to use bounded-weight codes. As our results show, even maximum-sized bounded-weight codes give scant improvement over their sister constant-weight codes, at least in the range—$w \leq \lfloor n/2 \rfloor$—in which the bounded-weight codes don't have a prohibitively unfair advantage in search-space size. Indeed, in this range, the increase in code size we found is disappointing, and as our codes in this range are all maximum-sized, this disappointment reflects the actual, optimal state of such codes. Additionally, there is a huge cost in adopting bounded-weight codes. In particular, the deepest direct advantage of constant-weight is that their weight provides an extra type

Length (n)	Weight (w)	Constant Weight	Bounded Weight
8	5	1	2
8	6	1	2
8	7	1	2
8	8	1	2
9	5	2	2
9	6	1	2
9	7	1	2
9	8	1	2
9	9	1	2
10	5	2	2
10	6	2	2
10	7	1	2
10	8	1	2
10	9	1	2
10	10	1	2
11	5	2	2
11	6	2	2
11	7	2	2
11	8	1	2
11	9	1	2
11	10	1	2
11	11	1	2
12	5	3	3
13	5	3	3
14	7	8	$8^{\star}$

Table 3: *Code sizes for* $d = 8$. Note: The value superscripted with "$\star$" was obtained through greedy search.

of error detection. Bounded-weight codes sacrifice this extra line of protection.

However, as a final comment, we mention that maximum-sized codes may have potential future uses in alternate models of computation/communication. Though this is currently hypothetical, it is not entirely implausible. Consider for example some future alternate model of information (storage or) transmission—perhaps biological, perhaps electrical, perhaps something else—in which each (stored or) transmitted "word" has n binary "bits" (which might be represented via genetic material, or via charged particles in a given location, or so on) but such that, due to constraints of the (storage or) transmission medium, if more than w of the bits are "on" there is the possibility that the information in the word will degrade, or that the computer or transmission lines will incur physical damage. Possible reasons might include power limitations, heat dissipation, or attraction between biological components. In this admittedly extremely hypothetical setting, bounded-weight codes might play a valuable role, as their limitation would be exactly suited to the physical constraints imposed by the (storage or) transmission medium.

Acknowledgments

We thank the anonymous J.UCS referees for helpful comments. The authors were supported in part by NSF grants 9322513, 9513368/DAAD-315-PRO-fo-ab, 9701911, 9725021, and 9815095/DAAD-315-PPP-gü-ab. This work was done while Gabriel Istrate was attending the University of Rochester.

References

[Applegate and Johnson] D. Applegate and D. Johnson. Clique-finding program dfmax.c. Available from ftp://dimacs.rutgers.edu/pub/challenge/graph/solvers.

[Brouwer et al. (97)] A. Brouwer, K. Hamalainen, P. Ostergard, and N. Sloane. Bounds on mixed binary/ternary codes. *IEEE Transactions on Information Theory*, IT-44, 1 (1997), 1334–1380.

[Brouwer] A. Brouwer. Bounds on the minimum distance of linear codes. Available at http://www.win.tue.nl/math/dw/voorlincod.html.

[Brouwer et al. (90)] A. Brouwer, J. Shearer, N. Sloane, and W. Smith. A new table of constant weight codes. *IEEE Transactions on Information Theory*, IT-36, 6 (1990), 1334–1380.

[Carragan and Pardalos (90)] R. Carraghan and P. Paradalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9 (1990), 375–382.

[Conway, Pless and Sloane (92)] J. Conway, V. Pless, and N. Sloane. The binary self-dual codes of length up to 32: A revised enumeration. *J. Combinatorial Theory, Series A*, 60, 2 (1992), 183–195.

[El Gamel et al. (87)] A. El Gamel, L. Hemachandra, I. Shperling, and V. Wei. Using simulated annealing to design good codes. *IEEE Transactions on Information Theory*, IT-33, 1 (1987), 116–123.

[Johnson et al. (91)] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation—Part II, graph coloring and number partitioning. *Operations Research*, 39, 3 (1991), 378–406.

[Johnson and Trick (93)] D. Johnson and M. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, number 26 in DIMACS series in Discrete Mathematics and Theoretical Computer Science. A.M.S., 1993.

[Litsyn, Rains and Sloane] S. Litsyn, E. Rains, and N. Sloane. Table of nonlinear binary codes. Available at http://www.research.att.com/~njas/codes/And.

[Rains and Sloane] E. Rains and N. Sloane. Table of constant weight binary codes. Available at http://www.research.att.com/~njas/codes/Andw/.

[Sloane] N. Sloane. Tables for various types of spherical codes. Available from http://www.research.att.com/~njas/.

[Vaessens, Aaarts and van Lint (93)] R. Vaessens, E. Aarts, and J. van Lint. Genetic algorithms in coding theory: A table for $A_3(n, d)$. *Discrete Applied Mathematics*, 45, 1 (1993), 71–87.

Appendix: Codes

This section presents codes that give the values in Tables 1, 2, and 3.

$A(6,4,3)$
000111
011001
101010
110100

$A(6,4,4)$
000000
010111
101011
111100

$A(6,4,5)$
000000
011110
111001
100111

$A(6,4,6)$
000000
001111
110011
111100

$A(7,4,3)$
0000111
0011001
0101010
0110100
1001100
1010010
1100001

$A(7,4,4)$
0000000
0101011
0110101
1011001
1101100
1110010
1000111
0011110

$A(7,4,5)$
0000000
0101101
1010101
0110011
0011110
1001011
1100110
1111000

$A(7,4,6)$
0000000
0001111
0110011
0111100
1010101
1011010
1100110
1101001

$A(7,4,7)$
0000000
0001111
0110011
0111100
1010101
1011010
1100110
1101001

$A(8,4,3)$
00000001
00101010
00110100
01001100
01010010
10011000
10000110
11100000

$A(8,4,4)$
00000000
00111100
01011010
01101001
10010110
10100101
11000011
00110011
01010101
01100110
10011001
10101010
11001100
00001111
11110000

$A(8,4,5)$
00000000
00111100
11011000
11100100
01001101
01010110
01101010
01110001
10001110
10010101
10101001
10110010
00011011
00100111
11000011

$A(8,4,6)$
00001111
00110011
01010101
01101001
10010110
10101010
11001100
11110000
00011000
00100100
01000010
01111110
10000001
10111101
11011011
11100111

$A(8,4,7)$
00001111
00110011
01010101
01101001
10010110
10101010
11001100
11110000
00011000
00100100
01000010
01111110
10000001
10111101
11011011
11100111

$A(8,4,8)$
00001111
00110011
01010101
01101001
10010110
10101010
11001100
11110000
00011000
00100100
01000010
01111110
10000001
10111101
11011011
11100111

$A(9,4,3)$
000000111
000011001
000101010
001001100
100010100
100100001
101000010
011000001
010100100
110001000
001110000
010010010

$A(9,4,4)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101

$A(9,4,5)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101

$A(9,4,6)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101

$A(9,4,7)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101

$A(9,4,8)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101
111011111

$A(9,4,9)$
000000000
001101010
010101100
011000110
100111000
000001111
000110011
101001100
101000011
001011001
111010000
110100010
010011010
110001001
011100001
100010110
001110100
100100101
010010101
111111011

$A(10,4,3)$
0000000001
0000101010
0000110100
0001001100
1100000100
1001100000
0101000010
0010000110
0011010000
0110100000
1010001000
0100011000
1000010010

$A(10,4,4)$
0000100111
0010110001
0010101010
0000011110
0011000011
0001011001
0001101100
0001110010
0010001101
1000110100
0110010010
1000010011
0100010101
1001000101
1010011000
0100111000
1000101001
1010000110
1011100000
0110100100
0100001011
0101100001
1001001010
0111001000
0101000110
0000000000
1110000001
1101010000
0011010100
1100100010
1100001100

$A(11,4,6)$
00110111010
00001111110
00101110011
10101111000
10111100010
10011011010
10000111011
00011101011
01011110010
01001111001
10011110001
00010111101
10010110110
00011010111
11001101010
10101001011
10001100111
01010011011
11010111000
11001010011
00111011001
10001011101
10110101001
11011001001
10110010011
10101010110
11010100011
10011101100
01101011010
10010001111
11000011110
10100101110
00100011111
10110011100
00111001110
11111010000
11011000110
11110001010
11100011001
11100110010
01011011100
10111000101
10100110101
01111000011
11010010101
01100101011
01001001111
01000110111
00110100111
11000101101
01010101110
11101100001
01100000000
00001100000
00000000011
10000001000
00010010000
11100000111
01100111100
01101010101
01111101000
01011100101
11001110100
00101101101
01110110001
01110001101
00111110100
01101100110
01110010110
11110100100
11101001100

$A(8,6,4)$
00000011
11110000

$A(8,6,5)$
00000001
01111100

$A(8,6,6)$
00000000
11111100

$A(8,6,7)$
00000000
01111110

$A(8,6,8)$
00000000
00111111

$A(9,6,4)$
000000011
110010100
001111000

$A(9,6,5)$
000000111
101110100
110011001
011101010

$A(9,6,6)$
000000000
111111000
001110111
110001111

$A(9,6,7)$
000000000
111110001
011101110
100011111

$A(9,6,8)$
000000000
001111110
111001101
110110011

$A(9,6,9)$
000000000
000111111
111000111
111111000

$A(10,6,4)$
0000001111
0001110001
0110010010
1010100100
1101001000

$A(10,6,5)$
1111000001
0001011101
1000110011
0110011010
1100101100
0011100110

$A(10,6,6)$
0000000000
1111001100
0011010111
1100100111
1001111010
0110111001

$A(10,6,7)$
0000000000
0111101100
1010100111
1100111010
1011011001
0101010111

$A(10,6,8)$
0000000000
1101100011
1001111100
1110011010
0110101101
0011010111

$A(10,6,9)$
0000000000
0001111110
1110001110
0111010101
1011101001
1100110011

$A(10,6,10)$
0000000000
0000111111
0111000111
1110110001
1101101010
1011011100

$A(12,6,6)$
010010111100
011100101001
001000011111
011001110010
011110000110
001011100101
110000100111
111010010001
000110110011
100011010110
000000000000
101100110100
000101101110
010011001011
100001111001
010101010101
110111100000
100110001101
110100011010
111001001100
001111011000
101101000011
101010101010

$A(8,8,5)$
00000111
11111000

$A(8,8,6)$
00000011
11111100

$A(8,8,7)$
00000101
11111010

$A(8,8,8)$
00100001
11011110

$A(9,8,5)$
000000111
111110000

$A(9,8,6)$
000000111
111110000

$A(9,8,7)$
000000001
111111100

$A(9,8,8)$
000000000
111111110

$A(9,8,9)$
000000000
011111111

$A(10,8,5)$
1111100000
0000000111

$A(10,8,6)$
0011111110
1100010000

$A(10,8,7)$
1111111000
0000010110

$A(10,8,8)$
0000000000
1111111100

$A(10,8,9)$
0000000000
0111111110

$A(10,8,10)$
1111100000
0000011111

$A(11,8,5)$
11111000000
00100101110

$A(11,8,6)$
00000000011
00011111100

$A(11,8,7)$
11111000000
00000100110

$A(11,8,8)$
00001111111
11110110001

$A(11,8,9)$
00001111111
11110110001

$A(11,8,10)$
00000000000
00111111110

$A(11,8,11)$
00000000000
00011111111

$A(12,8,5)$
000000000111
011100011000
100011110000

$A(13,8,5)$
0000000000111
0111000101000
1000111100000

$A(14,8,7)$
10010100111100
11100011101000
01101110010100
00110110100011
11001100001011
10011011010010
00111001001101
01000001110111