On Bounded-Weight Error-Correcting Codes

Russell Bent, Michael Schear, Lane A. Hemaspaandra, and Gabriel Istrate
Department of Computer Science
University of Rochester
Rochester, NY 14627

June 1, 1999
ACM Subject Codes: H.1.1, E.4

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.

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 [Bro], binary self-dual codes [CPS92], mixed binary-ternary codes [BHOS97], and various classes of spherical codes [Slo].

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 by Litsyn, Rains, and Sloane [LRS] and Sloane and Rains [RS], 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

*Supported in part by NSF grants 9322513, 9513368/DAAD-315-PRO-fo-ab, 9701911, and 9815095/DAAD-315-PPP-gü-ab. Email: {rbent,mshear,lane,istrate}@cs.rochester.edu.
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. ([BHOS97], see also [AJ] 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, $\{j : 1 \leq j \leq n \land x_j \neq y_j\}$. 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 [RS]. 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. [BSSS90]. 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 \cite{EHSWS87}, genetic algorithms \cite{VAvL93}, 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 \cite{JT93}, 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. \cite{BHOS97}. 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 by Applegate and Johnson (\cite{AJ}, see also \cite{CP90}). 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 (\cite{AJ}, see also \cite{JAMS91}). 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.

Clearly, a lower bound for bounded-weight codes is

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

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

$^{[1]}$These CPU times were obtained using a Sun Ultra 10.
Table 1: Distance 4. Note: The values superscripted with “⋆” were obtained through greedy search.

| Length | Weight | Constant | Bounded |
|--------|--------|----------|---------|
| 6      | 3      | 4        | 4       |
| 6      | 4      | 3        | 4       |
| 6      | 5      | 1        | 4       |
| 6      | 6      | 1        | 4       |
| 7      | 4      | 7        | 7       |
| 7      | 4      | 7        | 8       |
| 7      | 5      | 3        | 8       |
| 7      | 6      | 1        | 8       |
| 7      | 7      | 1        | 8       |
| 8      | 3      | 8        | 4       |
| 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\*    |
| 9      | 6      | 12       | 19\*    |
| 9      | 7      | 4        | 19\*    |
| 9      | 8      | 1        | 20\*    |
| 9      | 9      | 1        | 20\*    |
| 10     | 3      | 13       | 13      |
| 10     | 4      | 30       | 31\*    |
| 11     | 6      | 66       | 71\*    |

Table 2: Distance 6. Note: The value superscripted with “⋆” was obtained through greedy search.

| Length | Weight | Constant | Bounded |
|--------|--------|----------|---------|
| 8      | 4      | 2        | 2       |
| 8      | 5      | 2        | 3       |
| 8      | 6      | 1        | 2       |
| 8      | 7      | 1        | 2       |
| 8      | 8      | 1        | 2       |
| 9      | 4      | 3        | 4       |
| 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       | 25\*    |
Table 3: Distance 8. Note: The value superscripted with “$*$” was obtained through greedy search.

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 asterisks) establishing the maximum size achievable by any legal code obeying 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 case $A(4, 8, 6)$ is of this sort.

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 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.

References

[AJ] D. Applegate and D. Johnson. Available from http://dimacs.rutgers.edu/pub/challenge/graph/solvers.

[BHOS97] A. Brouwer, K. Hamalainen, P. Ostergard, and N. Sloane. Bounds on mixed binary/ternary codes. IEEE Transactions on Information Theory, IT-44(1):1334–1380, January 1997.

[Bro] A. Brouwer. Bounds on the minimum distance of linear codes. http://www.win.tue.nl/math/dw/voorlincod.html.
A. Brouwer, J. Shearer, N. Sloane, and W. Smith. A new table of constant weight codes. *IEEE Transactions on Information Theory*, IT-36(6):1334–1380, November 1990.

R. Carraghan and P. Paradalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

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):183–195, 1992.

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):116–123, 1987.

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):378–406, 1991.

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.

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

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

N. Sloane. Available from http://www.research.att.com/~njas/.

R. Vaessens, E. Aarts, and J. van Lint. Genetic algorithms in coding theory: A table for $A_q(n,d)$. *Discrete Applied Mathematics*, 45(1):71–87, August 1993.
## Appendix: Codes

| A(6, 4.3) | A(8, 4.4) | A(8, 4.5) | A(9, 4.6) | A(9, 4.9) |
|-----------|-----------|-----------|-----------|-----------|
| 000111    | 00000000  | 00000111  | 00000000  | 00000000  |
| 011001    | 00111000  | 00110101  | 00110101  | 00110101  |
| 101010    | 01010101  | 10101010  | 10101010  | 01010101  |
| 110100    | 01101001  | 10010110  | 00011010  | 01100110  |
| A(6, 4.4) | 000000    | 11000011  | 11001100  | 00011001  |
| 000000    | 00110011  | 11110000  | 10100100  | 10010101  |
| 010111    | 01010101  | 00101000  | 00101000  | 01010101  |
| 101011    | 11100000  | 11001100  | 01100100  | 10010101  |
| 111000    | 01100100  | 10010000  | 11100000  | 00110001  |
| A(6, 4.5) | 000000    | 11001100  | 10000001  | 10010101  |
| 011110    | 00011111  | 10111101  | 11001110  | 01010010  |
| 111001    | 11110000  | 11101100  | 01100101  | 01001010  |
| 100111    | 11110011  | 11111111  | 11111111  | 11111111  |

| A(7, 4.3) | A(8, 4.5) | A(9, 4.3) | A(10, 4.3) |
|-----------|-----------|-----------|-------------|
| 000000    | 00011111  | 00000000  | 00000000    |
| 000000    | 11010000  | 11000011  | 11000010    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |

| A(7, 4.4) | A(8, 4.6) | A(9, 4.4) | A(10, 4.4) |
|-----------|-----------|-----------|-------------|
| 000000    | 00011111  | 00000000  | 00000111    |
| 000000    | 11010000  | 11000011  | 11000010    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |

| A(7, 4.6) | A(8, 4.7) | A(9, 4.5) | A(10, 4.5) |
|-----------|-----------|-----------|-------------|
| 000000    | 00011111  | 00000000  | 00000000    |
| 000000    | 11010000  | 11000011  | 11000010    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |

| A(7, 4.7) | A(8, 4.8) | A(9, 4.6) | A(10, 4.6) |
|-----------|-----------|-----------|-------------|
| 000000    | 00011111  | 00000000  | 00000000    |
| 000000    | 11010000  | 11000011  | 11000010    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
| 000000    | 01001010  | 00010100  | 00010100    |
| 000000    | 11001000  | 11001000  | 11001000    |
