3-1993

Simulated Annealing and Optimal Codes

Gary R. Greenfield

Simulated Annealing and Optimal Codes

Gary R. Greenfield
Department of Mathematics and Computer Science
University of Richmond
Richmond, Virginia 23173

March, 1993

email: grg5l@aurora.urich.edu

TR-93-02

1 Introduction

Following standard notation, an (n, m, d) code C denotes a *binary* code C which has *length* n, *size* m, and *Hamming distance* d. According to Hill [6] the "main coding theory problem" is to optimize one of these three parameters when the other two are held fixed. The usual version of this optimization problem is to find the largest code for a given length and given minimum distance. This is the problem we shall consider, thus making it clear what we mean by an "optimal code." Formally,

DEFINITION. Fix integers n and d, where $n > 0$ and $1 \leq d \leq n$. An (n, m, d) code C is *optimal* if for any (n, m', d) code C' we have $m \geq m'$.

Shadowing Hill, we let $A(n, d)$ denote the size of an optimal (n, d) binary code. It is easy to see that $A(n, 1) = 2^n$ and $A(n, n) = 2$. Since $A(n, 2r) = A(n-1, 2r-1)$ it suffices to consider the optimization problem under the assumption d is odd. The following table for $A(n, d)$ is reproduced from Hill and to the best of our knowledge is accurate circa 1986. For unknown entries, the best available bounds are given.

n	$d = 3$	$d = 5$	$d = 7$
5	4	2	—
6	8	2	—
7	16	2	2
8	20	4	2
9	40	6	2
10	72–79	12	2
11	144–158	24	4
12	256	32	4
13	512	64	8
14	1024	128	16
15	2048	256	32
16	2560–3276	256–340	36–37

Observe then that the entry for $n = 10$ and $d = 3$ is interpreted as saying $72 \leq A(10, 3) \leq 79$, indicating that a $(10, 72, 3)$ code is known to exist, and that no code with $n = 10$ and $d = 3$ can have more than 79 codewords. In other words, it would be significant if one could find even a (10,**73**,3) code! This report documents our unsuccessful attempt to search for such a code and hence shed light on the minimal case where $A(n, d)$ has not yet been

determined. This report is structured to follow, more or less, the chronology of our attempt, which is also instructive and amusing.

2 Code Construction using Neural Nets

While participating in a seminar on coding theory we stumbled across Francis and Fuller [5] which purported to find optimal codes using neural nets. Before describing the method, we should remark that the most intriguing aspect of this paper was that it stopped just short of trying to find a code which would be informative concerning $A(10, 3)$, the minimal *unsolved* case.

Given n, the 2^n binary strings of length n are the "neurons," V_i is the output signal from the i-th neuron, and $T_{i,j} = T_{j,i}$ is the strength of the connection between neurons i and j. (For convenience, one sets $T_{i,i} = 0$.) If the "threshold" for the i-th neuron is U_i, the total energy of the network is given by

$$E = -\frac{1}{2}\sum_i \sum_{j\neq i} T_{i,j}V_jV_i + \sum_i U_iV_i.$$

The network "learns" by adjusting individual outputs using the rule

$$V_i = \begin{cases} 1 & \text{if } \sum_{j\neq i} T_{i,j}V_j > U_i \\ 0 & \text{if } \sum_{j\neq i} T_{i,j}V_j \leq U_i. \end{cases}$$

The advantage to using such Hopfield nets is that after having adjusted one or more randomly selected neural outputs the change in energy

$$\Delta E = -\sum_i \sum_{j\neq i} (T_{i,j}V_j - U_i)\Delta V_i$$

must be negative, hence eventually the network will converge to a *local* minimum. In [5] U_i was set to zero, and for $j \neq i$ the connection strengths were set as follows:

$$T_{i,j} = \begin{cases} +1 & \text{if } H(i,j) \geq d \\ -J & \text{if } H(i,j) < d, \end{cases}$$

where $H(i, j)$ is the Hamming distance between neurons i and j, and J was the known optimal value $A(n, d)$. The point is that if a neuron output is one, the neuron is broadcasting that it should be included in the optimal code, hence all neurons less than Hamming distance d from it must be excluded.

We presume, since the word "iteration" was used, that the implementation swept through *all* neurons adjusting individual outputs until a sweep resulted in no adjustments being made. Evidently the local minimums that were found did not yield optimal codes because at this juncture Francis and Fuller backpedaled by introducing a probabilistic adjustment rule in place of the deterministic one given above. Worse, the probabilistic rule itself was altered after each iteration. The exposition became confused, the notion of convergence seemed to have disappeared, and so at this point we sought a more definitive explanation concerning this probabilistic approach that was marauding under the name *simulated annealing*. None the less, it was clear that processing time was a bottleneck if one wanted to consider a larger value of n, and that the methodology was justifiable only to the extent that the techniques being cobbled together were successful in obtaining an optimal (8,20,3) code.

3 Code Construction using Simulated Annealing

The touted Rumelhart and McClelland reference [10] did mention simulated annealing, but useful reference material on the subject was first gleaned from [8]. In a nutshell, the philosophy behind simulated annealing is to simulate a hardening process by first "melting" a system at a very high temperature then lowering the temperature in slow stages. More importantly, at each temperature the system must be sustained for a sufficiently long period of time analogous with gently mixing and stirring. Specifically, in a system with total energy E that is to be minimized, assume a displacement to the system will result in the energy change ΔE. If $\Delta E \leq 0$ the displacement is accepted, but if $\Delta E > 0$ the displacement is accepted with probability $P(\Delta E) = \exp(-\Delta E/Q)$ where Q depends on the temperature, and the Boltzmann distribution for $P(\Delta E)$ is justified by appealing to the theory of the thermal distribution of atoms. Some bold and exciting claims were made for simulated annealing in [8] thus we were disappointed that its solutions to the Traveling Salesperson Problem were obtained by manually restarting the simulation when a less than satisfactory local minimum was reached and proffered to the viewer. In any event, certain maxims were identified.

- The energy or cost function must be representative.
- The "cooling schedule" should be divined by reading tea leaves.

- There must be an efficient *neighborhood structure* for considering feasible solutions to the optimization problem.
- Extensive processing time should be devoted to the local search performed at each temperature.

We had to learn these maxims by poring over preprints and reprints. Fortunately now they are embodied, along with an algorithm in pseudocode for simulated annealing, in the readily accessible [1]. The critical observation we want to make is that simulated annealing is intended for examining *feasible solutions* and thus a neural network that considers all binary strings seems to be an inappropriate setting for its use. In fact, the single reference where we found simulated annealing invoked for a Hopfield net in the manner most closely resembling [5] (*i.e.*, at neuron i accepting the change in output with probability $P(\Delta E_i) = (1 + \exp(-\Delta E_i/T))^{-1}$) seemed to reject it out of hand [7].

With n and d fixed our model evaluates a feasible code $X = \{x_1, \ldots, x_m\}$ using the cost function

$$f(X) = \sum_i \sum_{j \neq i} T(x_i, x_j)$$

where

$$T(x_i, x_j) = \begin{cases} -1 & \text{if } H(x_i, x_j) \geq d \\ +J & \text{if } H(x_i, x_j) < d. \end{cases}$$

We set $J = 1 + U$ where U is the *size of the code we are looking for.*

IF X HAS DESIRED DISTANCE D AND DESIRED SIZE U, THEN $T(x_i, x_j) = -1$ FOR ALL $i \neq j$ AND $f(X) = U(1 - U)$.

Since without loss of generality any code contains the string consisting of all zeros, our X will always have this codeword "hardwired" in. This anchor codeword has the added advantage of helping to prevent the code from drifting randomly through a series of equal cost feasible solutions *i.e*, thrashing. Any altered, substituted, or added codeword x will always have *weight* $W(x) = H(x, 0) = x \cdot x \geq d$. Following simulated annealing methodology, X is to be replaced by Y whenever $\Delta f = f(Y) - f(X)$ is negative, or, with probability $\exp(-\Delta f/T)$, whenever Δf is positive. T is the current temperature of the system about which we will say more later.

We first tried to build a code starting from $X = \{0\}$ using an algorithm that would try to sometimes add a codeword to X, sometimes replace a

codeword in X, or sometimes delete a codeword from X all within a simulated annealing framework. This produced hopeless and confused results. Instead, more in keeping with the spirit of simulated annealing, we decided in advance the size U we were targeting for X, and initialized X with the all zeros codeword plus $U-1$ strings of weight at least d. The perturbation Y of X was simply X with a randomly selected element x replaced by a new randomly generated string y of weight at least d. Clearly most random strings would not be of use to X, whence running the simulation for long periods at each temperature was justified. It would seem like a good idea to try exchanging y for every possible nonzero $x \in X$. We rejected this as being too close to exhaustive search as well as contrary to the philosophy that says simulated annealing works best in situations where one doesn't know how to construct X. We used an initial temperature of 100.0 and cooled to 0.01 by reducing the temperature at each stage by a factor of 0.99 and simmering at each temperature by testing sometimes as many as a 1000 feasible solutions. Later we added our own fillip by "reheating" the system each time the temperature reached 0.01, boosting the temperature back up to $100/(k+1)$ at the k-th reheat. On our "slow" SUN-3, this meant some runs took in excess of twenty four hours!

Our implementation is a C program of only a few hundred lines. By maintaining an array of the $T_{i,j}$ the program can efficiently calculate $f(Y)$ by knowing $f(X)$ and maintaining the coefficients for y as a phantom last row and column of this array. In rare cases where x is exchanged for y the program is able to rapidly effect the updates by swapping.

Our results are almost anticlimactic. An optimal $(8, 20, 3)$ could be achieved in no time at all. We were able to obtain a $(10, 72, 3)$ code in a little under four hours. We were unable to construct a $(10, 74, 3)$ code though several runs achieved $(10, 74, 2)$ codes which by inspection — because our program flags those codewords that "conflict" with one another *i.e.*, flags values $T_{i,j}$ which equal J — could be trimmed to $(10, 72, 3)$ codes. We also attempted to construct a $(16, 36, 7)$ code (the other easily targeted unknown $A(n, d)$ value) to test the robustness of simulated annealing and were surprised when we could *not* construct such a code. This setback coupled with developments detailed below caused our enthusiasm for the project to wane. Since it is *de riguer* in coding theory circles, we have appended an optimal $(10, 72, 3)$ code and the output of a run that came closest to finding a $(16, 36, 7)$ code. The indexing of codewords begins at zero as a concession to the C programming language, and the cost function, for reasons that escape us at the moment, is denoted by `phiX` rather than $f(X)$.

It will be clear from the output that we calculate $f(X)$ as $\sum_i f(x_i)$ where $f(x_i) = \sum_{j \neq i} T_{i,j}$.

During the period we were actively burning up cpu cycles we located reference [4], a paper that *had* used simulated annealing to construct "good" codes. But instead of general purpose codes these were special codes — source codes, constant weight codes, and spherical codes! Realizing that the unsolved $n = 10$, $d = 3$ and $n = 16$, $d = 7$ problems we had been working on had been scrupulously avoided made us suspicious and further dampened our enthusiasm. It appeared that the authors of [4] had been opportunistic in seeking problems that had not been studied quite so intensely as ours, and a footnote in the paper revealed that what the authors had thought to be several improved spherical codes had already been bested elsewhere in the literature using other techniques. We were interested to note that on a machine comparable to ours the authors never permitted a run to last more than four hours and halted runs when no consistent improvement was observable. We did "borrow" from [4] by adopting their initial temperature and proportional cooling schedule. The straw that broke the back of simulated annealing for us, however, came with the discovery that most optimal codes could be constructed in a very naive and straightforward way, hence it did not seem "fair" that one would have to use a slow painful method like simulated annealing for some "oddball" cases. We now turn to this development.

4 Code Construction using Greedy Algorithms

We had described the optimal code problem to several non-experts. We advertised the problem as easy to understand but very hard to solve, so we didn't know what to make at first out of a communique from long time acquaintance R. Guy Lauterbach asking for the table of known results so he could compare it with the results he was obtaining using a thirty line program which was taking between five seconds and four minutes per run on a slightly more powerful machine than ours. The table he provided follows. We have highlighted entries in boldface that fall short of the best known bounds.

n	$d=3$	$d=5$	$d=7$
5	4	2	—
6	8	2	—
7	16	2	2
8	**16**	4	2
9	**32**	**4**	2
10	**64**	**8**	2
11	**128**	**16**	4
12	256	**16**	4
13	512	**32**	8
14	1024	**64**	16
15	2048	**128**	32
16	**2048**	256	**32**

The entries, all powers of two, suggested that the codes were linear, but how could they be generated so fast? We persuaded Guy, a professional programmer, to send us his code which is reproduced as Appendix B. It is a model of efficiency. (Note for example that the Hamming distance is calculated by summing the 'logical and' of the circularly shifted codeword with the hex constant one!) The algorithm is a "greedy algorithm," which considers the strings of length n in binary sequence after first initializing with the all zero string and the string ending with d ones, then adding strings to the code when they are a distance at least d from all strings already comprising the code. But why is it linear? With the help of our coding theorist, James A. Davis, who remembered hearing a talk by Vera Pless on the linearity of greedy algorithm codes, we secured the Brualdi and Pless preprint [3] which contained a proof that greedy algorithm codes were linear, and suggested ways to improve on the constructions by altering the search strategy.

5 Conclusion

Our efforts devoted to the $A(10,3)$ problem provide heuristic evidence that its value should be 72. They convince us that the space of codes with $n=10$ and $m=72$ must be very *flat*. Thus local minimums are easy to find, and if these are not global minimums for the $n=10$ and $d=3$ problem, then the geometry must be very strange and exotic. Regarding simulated annealing, we are led to doubt its role in solving *global* optimization problems, but

we are impressed by its ability to be *tuned* to provide constructions of near optimal nonlinear solutions. We also found it instructive to compare the cost functions used in [4] for source, constant weight, and spherical codes with our naive general purpose but easily computed cost function. Perhaps simulated annealing suffers less from the vagaries of cooling schedules and probability distributions then from the sensitivity of its cost or energy functions. Finally, the near optimal size of greedy codes, their ease of construction, and their linearity serve to reinforce their importance as *practical* codes. But in the grand scheme — *arbitrary* values of n and d — one must conclude that much remains to be discovered about the theory and construction of optimal codes.

Acknowledgement

I would like to thank first and foremost James A. Davis for serving as coding theory expert and willing consultant on this project. I would like to thank Kathy Hoke for providing mainstream combinatorial optimization examples using simulated annealing thereby enlightening me about how neighborhood structures might be successfully designed and implemented. And, of course, I want to thank my longtime best friend Guy Lauterbach for contributing a zest and enthusiasm that ensured this project would have a beginning, middle, and end.

References

[1] E. H. L. Aarts, On jargon: simulated annealing, *The UMAP Journal*, Volume 13, Number 1, 1992, 79–89.

[2] E. H. L. Aarts & J. H. M. Korst, *Simulated Annealing and Boltzmann Machines*, Wiley, New York NY, 1989.

[3] R. A. Brualdi & V. S. Pless, Greedy codes, preprint.

[4] A. A. El Gamal, L. A. Hemachandra, I. Shperling, & V. K. Wei, Using simulated annealing to design good codes, *IEEE Transactions on Information Theory*, Volume IT-33, Number 1, January 1987, 116–123.

[5] G. Francis & K. Fuller, Using neural networks to solve coding theory problems, *J. Undergraduate Research in Math*, 1992, 55–58.

[6] R. Hill, *A First Course in Coding Theory*, Oxford University Press, New York NY, 1986.

[7] T. Khanna, *Foundations of Neural Networks*, Addison-Wesley, Reading MA, 1990.

[8] S. Kirkpatrick, C. D. Gelatt, Jr., & M. P. Vecchi, Optimization by simulated annealing, *Science*, Volume 220, Number 4598, 13 May 1983, 671–680.

[9] P. J. M. van Laarhoven & E. H. L. Aarts, *Simulated Annealing: Theory and Practice*, Reidel, Dordrecht, The Netherlands, 1987.

[10] D. E. Rumelhart & J. L. McClelland, *Parallel Distribute Processing*, MIT Press, Cambridge MA, 1986.

i = 0	0000000000	i = 36	0100001011
i = 1	1010011101	i = 37	0011100011
i = 2	1101110100	i = 38	0001110000
i = 3	0100100001	i = 39	0101110111
i = 4	1001000110	i = 40	1111100101
i = 5	0111101110	i = 41	1110101001
i = 6	0110001101	i = 42	0010111111
i = 7	0100111010	i = 43	1100010000
i = 8	1111110010	i = 44	1001100001
i = 9	0111111101	i = 45	1110010111
i = 10	1011001111	i = 46	0000000111
i = 11	1010011010	i = 47	1001111110
i = 12	0010001110	i = 48	1100111101
i = 13	0011010110	i = 49	0101011110
i = 14	0111000111	i = 50	1100001110
i = 15	1000110011	i = 51	0110110011
i = 16	0001010011	i = 52	0101000100
i = 17	1010110000	i = 53	1001011000
i = 18	0110011000	i = 54	1110111110
i = 19	1101000011	i = 55	0000111001
i = 20	1011101100	i = 56	1001010101
i = 21	0010100101	i = 57	0101100010
i = 22	0011001001	i = 58	1100100111
i = 23	0001001010	i = 59	1011000000
i = 24	1101111011	i = 60	1111011100
i = 25	0100101100	i = 61	1010100110
i = 26	0000110110	i = 62	0001101111
i = 27	1011110111	i = 63	0010101000
i = 28	0110110100	i = 64	1111010001
i = 29	0100010101	i = 65	1000001001
i = 30	1010000011	i = 66	1101101000
i = 31	1101001101	i = 67	1011111001
i = 32	0110000010	i = 68	0000011100
i = 33	0010010001	i = 69	1111001010
i = 34	1110000100	i = 70	0011111010
i = 35	0101011001	i = 71	1000101010

Appendix A.1 An n = 10, m = 72, d = 3 code.

```
N =  16
D =   7
U =  36
J =  37

seed     = 1186
U*(1-U)  =  -1260
NREHEATS =     10
TSIMMER  =    500
TSTART   = 100.00
TSTOP    =   0.00
TFACTOR  =   0.99

loop =  0  phiXsum =  16448  starting temperature = 100.00
loop =  1  phiXsum =    184  starting temperature =  50.00
loop =  2  phiXsum =    108  starting temperature =  33.33
loop =  3  phiXsum =   -500  starting temperature =  25.00
loop =  4  phiXsum =    488  starting temperature =  20.00
loop =  5  phiXsum =    336  starting temperature =  16.67
loop =  6  phiXsum =     32  starting temperature =  14.29
loop =  7  phiXsum =     32  starting temperature =  12.50
loop =  8  phiXsum =   -196  starting temperature =  11.11
loop =  9  phiXsum =   -728  starting temperature =  10.00

i =  0  phiX(i) = -35 0000000000000000
i =  1  phiX(i) = -35 1101001001110010
i =  2  phiX(i) = -35 1011011000100111
i =  3  phiX(i) = -35 1110101001001110
i =  4  phiX(i) =   3 0111111010010010    19
i =  5  phiX(i) = -35 1111000111100100
i =  6  phiX(i) = -35 1101011100001100
i =  7  phiX(i) = -35 0001101011000111
i =  8  phiX(i) = -35 1111110101000011
i =  9  phiX(i) = -35 1010100011110011
i = 10  phiX(i) = -35 1000011111000010
i = 11  phiX(i) = -35 1110010100111010
i = 12  phiX(i) = -35 1110001110010111
i = 13  phiX(i) = -35 1100101100100001
i = 14  phiX(i) =   3 1010111101110100    21
i = 15  phiX(i) =   3 1110011011101001    22
i = 16  phiX(i) = -35 0000001010111110
i = 17  phiX(i) = -35 0001010110110101
i = 18  phiX(i) = -35 1011010011011110
i = 19  phiX(i) =   3 1111111010010000     4
i = 20  phiX(i) = -35 1100110010100110
i = 21  phiX(i) =   3 0010111101110110    14
i = 22  phiX(i) =   3 0110011011101011    15
i = 23  phiX(i) =   3 0001110001101001    27
i = 24  phiX(i) = -35 0111101000111101
i = 25  phiX(i) =   3 0010111110001101    28
i = 26  phiX(i) = -35 1011101110101010
i = 27  phiX(i) =   3 0011110001101000    23
i = 28  phiX(i) =   3 0110110110001101    25
i = 29  phiX(i) = -35 0100011001010101
i = 30  phiX(i) = -35 1101111111111111
i = 31  phiX(i) = -35 0100100111111000
i = 32  phiX(i) = -35 1011001101011001
i = 33  phiX(i) = -35 1101000010001011
i = 34  phiX(i) = -35 1000111000011011
i = 35  phiX(i) = -35 1000000101101111

start time         = Tue May 12 12:56:45 1992
finish time        = Wed May 13 17:37:47 1992
finish distance    =     2
finish phiXsum     =  -880
desired phiXsum    = -1260
```

Appendix A.2 An attempt to construct an n = 16, m = 36, d = 7 code.

```
main(argc, argv)
int argc;
char *argv[];
{
    register int n, d;
    register unsigned long guess, limit;
    unsigned long val[5000];
    int vx, i;
    n = getarg(argv[1]);
    d = getarg(argv[2]);
    limit = 1 << n;
    guess = 1 << d;
    printf ("n=%d, d=%d, limit=%8.8x\n", n,d,limit);
    vx = 1;
    val[0] = 0;
    val[1] = guess-1;
    printf("vx=%3d, v=%8.8x\n", 0, 0);
    printf("vx=%3d, v=%8.8x\n", 1, val[1]);

    while (guess < limit) {
        i = vx;
        do {
            if (dist(guess ^ val[i]) < d) break;
        } while (i--);
        if (i < 0) {
            val[++vx] = guess;
            printf("vx=%3d, v=%8.8x\n", vx, guess);
        }
        guess++;
    }
}


int dist(x)
register unsigned long x;
{
    register int t = 0;
    while (x) {
        if (x & 0x1)
            t++;
        x >>= 1;
    }
    return (t);
}


int getarg(sp)
char *sp;
{
    int n;
    n = *sp++ -'0';
    if (*sp >= '0' && *sp <= '9')
        n = n*10 + *sp - '0';
    return(n);
}
```

Appendix B A C program for finding greedy (n,d) codes. Invoked as (prog) (n) (d).