Small Parity-Check Erasure Codes - Exploration and Observations

James S. Plank

Adam L. Buchsbaum

Rebecca L. Collins

Michael G. Thomason

Technical Report UT-CS-04-537

Department of Computer Science

University of Tennessee

Knoxville, TN 37996

November, 2004

`http://www.cs.utk.edu/~plank/plank/papers/CS-04-528.html`

This paper has been submitted for publication. Please see the above web site for up-to-date information about the publication status of the paper.

Small Parity-Check Erasure Codes - Exploration and Observations

James S. Plank † Adam L. Buchsbaum ‡ Rebecca L. Collins †
Michael G. Thomason †*

Abstract

Erasure codes have profound uses in wide- and medium-area storage applications. While infinite-size codes have been developed with optimal properties, there remains a need to develop small codes with optimal properties. In this paper, we provide a framework for exploring very small codes, and we use this framework to derive optimal and near-optimal ones for discrete numbers of data bits and coding bits. These codes have heretofore been unknown and unpublished, and should be useful in practice. We also use our exploration to make observations about upper bounds for these codes, in order to gain a better understanding of them and to lead the way for future derivations of larger, optimal and near-optimal codes.

1 Introduction

Erasure codes have been gaining in popularity, as wide-area, Grid, and peer-to-peer file systems need to provide fault-tolerance and caching that works more efficiently and resiliently than by replication [FMS+04, GWGR04, PT04, RWE+01, ZL02]. In a typical erasure code setting, a file is decomposed into n equal sized *data* blocks, and from these, m additional *coding* blocks of the same size are calculated. The suite of $n + m$ blocks is distributed among the servers of a wide-area file system, and a client desiring to access the file need only grab fn of these blocks in order to recalculate the file. In this setting, f is termed the *overhead factor*, and has one as its lower bound.

Reed-Solomon codes [Pla97, PD04, Riz97] are a class of erasure codes that have ideal overhead factors (f = 1). However, their computational overhead grows quadratically with n and m, severely limiting their use. Low-Density Parity-Check (LDPC) codes [LMS+97, RU03, WK03] have arisen as important alternatives to Reed-Solomon codes. Although their overhead factors are suboptimally greater than one, their computational overheads are very low. Thus, the tradeoff between a client having to download more than n blocks of data is mitigated by the fact that recalculating the blocks of the data is extremely fast, and in particular much faster than Reed-Solomon codes.

*This material is based upon work supported by the National Science Foundation under grants CNS-0437508, ACI-0204007, ANI-0222945, and EIA-9972889. † Department of Computer Science, University of Tennessee, Knoxville, TN, 37996, [plank,rcollins,thomason]@cs.utk.edu; ‡ AT&T Labs, Shannon Laboratory, 180 Park Ave., Florham Park, NJ 07932, alb@research.att.com.

The theory for LDPC codes has been developed for asymptotics, proving that as n goes to infinity, the overhead factor of codes approaches its optimal value of one. For small values of n and m (less than 1000), there is little theory, and recent work has shown that the techniques developed for asymptotics do not fare well for small n and m [PT04].

The purpose of this paper is to start closing this hole in the theory. Rather than concentrate on large values of n and m, we concentrate on very small values, using enumeration and heuristics to derive either optimal codes for these small values, or codes that are not yet provably optimal, but represent the lowest known upper bounds. We present these codes as they should be useful to the community. Additionally, we demonstrate some properties of small codes and present observations about the codes that we have derived. We leave the proof/disproof of these observations as open questions to the community.

The significance of this work is the following:

1. To present optimal, small codes to the community. To the authors' knowledge, this is the first such presentation of codes.
2. To present upper bounds on larger codes to the community. To the authors' knowledge, this is also the first such presentation of codes.
3. To present evaluation, enumeration and pruning techniques that apply to small codes, and have not been used on LDPC codes previously.
4. To stimulate thought on small codes in hope of proving properties of codes in general that do not rely upon classical asymptotic, probabilistic arguments.

2 LDPC Basics

The material in this section is all well-known and has been presented elsewhere. See [WK03] for more detail.

Although wide-area file systems use LDPC codes to operate on blocks of data, the specification of LDPC codes is typically on bits of data. Blocks are simply composed of multiple bits. In this work, we use the following terminology:

- The number of data bits is n.
- The number of coding bits is m.
- The total number of data and coding bits is $N = n + m$.
- The *rate* $\mathcal{R}$ of a code is $\frac{n}{N}$.
- The *overhead* o of a code is the average number bits that must be present to decode all the bits of the data.
- The *overhead factor* f of a code is o/n.

LDPC codes are based on bipartite graphs known as "Tanner" graphs. These graphs have N nodes $l_1, \ldots, l_N$ on their left side, sometimes termed the "message" nodes, and m nodes $r_1, \ldots, r_m$ on their right side, termed "check" or "constraint" nodes. Edges only connect message and check nodes. An example graph is depicted in Figure 1.

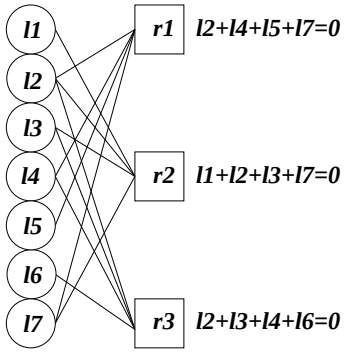


Figure 1: An example Tanner graph for $n = 4$ and $m = 3$.

The left-hand nodes hold the bits that are to be stored by the application. The edges and the right-hand nodes specify constraints that the left-hand nodes must satisfy. The most straightforward codes are "systematic" codes, where the data bits are stored in n of the left-hand nodes, and the coding bits in the remaining m left-hand nodes are calculated from the data bits and the constraints in the right-hand nodes using exclusive-or.

For example the code in Figure 1 is a systematic one, whose data bits may be stored in nodes l_1 through l_4. The coding bits are calculated as follows:

- Bit l_6 is the exclusive-or of l_2, l_3 and l_4 (from constraint r_3).
- Bit l_7 is the exclusive-or of l_1, l_2 and l_3 (from constraint r_2).
- Bit l_5 is the exclusive-or of l_2, l_4 and l_7 (from constraint r_1).

We present decoding as an act in a storage system. Suppose we store each of the N bits on a different storage server. Then we download bits from the storage server at random until we have downloaded enough bits to reconstruct the data. To decode in this manner, we start with the Tanner graph for the code, placing values of zero in each right-hand node, and leaving the left-hand nodes empty. When we download a bit, we put its value into its corresponding left-hand hand node l_i. Then, for each right-hand node r_j to which it is connected, we update the value stored in r_j to be the exclusive-or of that value and the value in l_i. We then remove the edge (l_i, r_j), from the graph. At the end of this process, if there is any right-hand node with only one incident edge, then it contains the decoded value of the left-hand node to which it is connected, and we can set the value of this left-hand node accordingly, and then remove its edges from the graph in the same manner as if it had been downloaded. Obviously, this is an iterative process.

When all nodes' values have been either downloaded or decoded, the decoding process is finished. If a code is systematic, then the data bits are held in n of the left-hand nodes. The number of exclusive-or/copy operations required equals the number of edges in the graph.

Encoding with a systematic graph is straightforward – simply decode using the n data bits.

2.1 Determining Whether A Graph Is Systematic

The following algorithm determines whether or a not a graph represents a systematic code. The algorithm iterates m times:

- Select a left-hand node that has exactly one edge to a constraint node. If there are no such left-hand nodes, then the graph does not represent a systematic code. Let the left-hand node be *code*$_i$ (for i equals 1 to m), and let the constraint node be named *const*$_i$.
- Remove *const*$_i$ and all edges incident to it.

If this algorithm iterates m times, then the graph represents a systematic code, with the m left-hand nodes holding the coding bits, and the n remaining left-hand nodes holding the data bits. Although the proof of correctness of this algorithm is not presented here, it should be noted that when the n data nodes are downloaded, constraint node *const*$_m$ will have one edge to it, and this edge is from node *code*$_m$. Therefore, node *code*$_m$ may be decoded. When *code*$_m$ is decoded, and all its other edges are removed from the graph, then node *const*$_{m-1}$ has only edge to it, and this edge is from node *code*$_{m-1}$. Decoding completes in this manner.

3 Three Ways of Computing Overhead

3.1 Brute-Force Enumeration

One can compute overhead in a brute-force fashion, by enumerating all $N!$ download sequences of bits, and averaging the number of bits requred to decode the data in each sequence. Obviously, this process becomes computationally intractible for rather small n and m. One may use Monte-Carlo simulation to approximate the overhead, as in [PT04]. However, there are alternative ways of computing overhead.

3.2 Recursive Overhead Calculation

In this section, we specify a technique to compute overhead recursively. Before making this specification, we give a more precise specification of the decoding process. We are given a Tanner graph G with $N = n + m$ left-hand nodes and m right-hand nodes, each of which may hold a bit. We assume that all the right-hand nodes have either zero incident edges or more than one incident edge. If a left-hand node has zero edges, then we assume that we know its value as a result of a previous decoding phase, but that we have not downloaded it.

When we start, we set the value of all right-hand nodes to zero and leave the values of all left-hand nodes blank.

To decode, we define two operations on graphs: *assigning* a value to a node, and *downloading* a node. Both operations are defined only on left-hand nodes. We start with the former. Given a left-hand node l_i, when the value of that node becomes known, it should be *assigned*. When it is assigned, for each right-hand node r_j to which l_i is connected, r_j's

value is set to the exclusive-or of its previous value and l_i's value, and then the edge (l_i, r_j) is removed from the graph. If there is any right-hand node r_j which now has only one incident edge, then the value of the left-hand node to which r_j is connected may now be assigned to be the value of r_j. Before assigning the value, however, the edge between that node and r_j should be removed, and r_j should also be removed from the graph. Note: assigning one node's value can therefore result in assigning many other nodes' values.

To *download* a node, if the node's value has already been assigned, then the node is simply removed from the graph. Otherwise, the value of the node is assigned to its downloaded value, and it is then removed from the graph.

When the values of all left-hand nodes have been assigned, the decoding process is finished.

Recursively computing the overhead $o(G)$ of graph G proceeds as follows. If all nodes have zero edges, then the overhead is zero. Otherwise, we simulate downloading each left-hand node of the graph and compute the average overhead as the average of all simulations. When we simulate downloading a node l_i, we assign its value (if unassigned), potentially decoding other nodes in the graph, and remove the node from the graph. We are then left with a *residual* graph, $R(G, l_i)$. We can recursively determine $R(G, l_i)$'s overhead. Then, the equation for determining a graph's overhead (if not zero), is:

$$o(G) = \left(\sum_{i=1}^{N} (1 + o(R(G, l_i))) \right) / N.$$

We give several examples of computing overhead in this fashion in Appendix 1.

3.3 Computing Overhead Using Residual Graphs

A third way to compute overhead is to look at a variation of the residual graph, presented above. Let $S_n(G)$ be the set of all subsets of the left-hand nodes of G that contain exactly n nodes. Let $S \in S_n(G)$. We define the residual graph R_S to be the graph that remains when all the nodes in S and their accompanying edges are removed from G. Note that unlike the residual graph in Section 3.2 above, we do not perform decoding when we remove nodes from the graph. R_S simply contains the nodes in G that are not in S.

We may calculate the overhead of R_S in either of the two manners described above. Let that overhead be $o(R_S)$. Note: the first step in doing so will be to decode right-hand nodes that are incident to only one left-hand node, and this overhead may well be zero (for example, $o(R_{\{l_1,l_2,l_3,l_4\}}) = 0$ for the graph in Figure 1).

Now, the overhead of a graph G may be defined as n plus the average overhead of the residual graphs that result when every subset of n nodes is removed from G. Formally:

$$o(G) = n + \left(\frac{\sum_{S \in S_n(G)} o(R_S)}{\binom{N}{n}} \right).$$

Note that this use of residual graphs is similar to using *stopping sets* [DPT+02] for overhead analysis.

4 Special Cases: $m = 1$ and $n = 1$

When $m = 1$, there is one coding node to n data nodes, and the optimal code is a straight parity code: $G^n_{m=1} = (\{l_1, \ldots, l_{n+1}, r_1\}, \{(l_1, r_1), \ldots, (l_{n+1}, r_1)\})$. One may easily prove using residual graphs that $o(G^n_{m=1}) = n$. When-

ever n nodes are removed from $G^n_{m=1}$, the residual graph contains one node with one edge to r_1. Clearly, the overhead of that graph is zero, and therefore the overhead of $G^n_{m=1}$ is the optimal value n.

When $n = 1$, there is one data node to m coding nodes, and the optimal code is a replication code: $G^{n=1}_m = (\{l_1, \ldots, l_{m+1}, r_1, \ldots, r_m\}, E^{n=1}_m)$, where $E^{n=1}_m = \{(l_1, r_i) | 1 \leq i \leq m\} \cup \{(l_i + 1, r_i) | 1 \leq i \leq m\}$.

It is straightforward to prove that $o(G^{n=1}_m) = 1$. Again, we use residual graphs. Suppose l_1 and all its edges are removed from the graph. Then the residual graph has exactly one edge to every right-hand node, and it may be decoded completely. Suppose instead that $l_{i \neq 1}$ and its one edge is removed from the graph. The residual graph has exactly one edge to $r_{i=1}$, which is connected to l_1. Therefore, l_1 and subsequently all other nodes may be decoded. Since the overhead of all residual graphs is zero, $o(G^{n=1}_m) = 1$.

5 Optimal and Near-Optimal Codes for $m \in \{2, 3, 4, 5\}$

In this section, we use residual graphs to derive closed-form expressions for the overhead of a graph G which has m right-hand nodes, and m is small (≤ 5). We introduce a new nomenclature for this derivation. We note that the N left-hand nodes of any graph may be partitioned into $2^m - 1$ sets, depending on the right-hand nodes to which they are connected. We label these sets $C_1, \ldots, C_{2^m-1}$, and specify that $l_i \in C_j$ if and only if:

$$j = \sum_{k=1}^{m} 2^{k-1} E(i, k),$$

where $E(i, k) = 1$ if (l_i, r_k) is an edge in G, and $E(i, k) = 0$ otherwise. Therefore, node l_1 in Figure 1 is an element of C_2, $l_2 \in C_7$, and $l_7 \in C_3$.

Let $c_i = |C_i|$. Then, one may uniquely specify a graph G by the values of each c_i, rather than by nodes and edges. For example, the graph in Figure 1 may be specified as (1, 1, 1, 1, 1, 1, 1), since there is one of each of the seven different kinds of left-hand nodes.

For later discussion, we will also define the function $e(i)$ to equal the number of one bits in the binary representation of the integer i. Therefore, if a node $l \in C_i$, then l has exactly $e(i)$ edges. Finally, we define an *Edge Class*, E_j to be the union of all C_i such that $e(i) = j$. We can then discuss the collection of nodes that have the same number of edges as the nodes in the same edge class. We will also group together all counts of nodes in edge class j as all $c_i \in E_j$.

5.1 Optimal Codes for $m = 2$

When $m = 2$, there are only three different types of left-hand nodes – those in C_1, C_2, and C_3. When n nodes are downloaded, only two remain, and there are only six possible residual graphs, which we specify by their values of c_i: (1, 1, 0), (1, 0, 1), (0, 1, 1), (2, 0, 0), (0, 2, 0), and (0, 0, 2). The first three of these contain one right-hand node with exactly one edge, which means each of these three may be completely decoded. The remaining three cannot be decoded until one of the two left-hand nodes is downloaded. Therefore, the overhead of the first three graphs is zero, and the overhead of the remaining three graphs is one.

Let $G = (c_1, c_2, c_3)$. To calculate overhead, we note that there are $\binom{c_1}{2}$ ways to download n nodes and have (2, 0, 0) as a residual graph. Likewise, there are $\binom{c_2}{2}$ ways to have (0, 2, 0) as a residual graph, and $\binom{c_3}{2}$ ways to have (0, 0, 2) as a residual graph. Therefore, the overhead of G is:

$$o(G) = n + \left(\frac{\binom{c_1}{2} + \binom{c_2}{2} + \binom{c_3}{2}}{\binom{N}{n}} \right).$$

Since $N = n + 2$, we note that $\binom{N}{n} = \binom{n+2}{2}$, and we may simplify $o(G)$ as follows:

$$\begin{aligned} o(G) &= n + \left(\frac{\frac{c_1(c_1-1)}{2} + \frac{c_2(c_2-1)}{2} + \frac{c_3(c_3-1)}{2}}{\frac{(n+2)(n+1)}{2}} \right) \\ &= n + \frac{c_1^2 + c_2^2 + c_3^2 - (c_1 + c_2 + c_3)}{(n+2)(n+1)} \\ &= n + \frac{c_1^2 + c_2^2 + c_3^2 - (n+2)}{(n+2)(n+1)}. \end{aligned}$$

Since n is a constant, in order to minimize the overhead, a graph must minimize $c_1^2 + c_2^2 + c_3^2$. It is easy to prove that this quantity is minimized when the differences among the c_i are minimized. Therefore, any graph G of the form (x,y,z), where $x, y, z \in \{\lfloor \frac{N}{3} \rfloor, \lceil \frac{N}{3} \rceil\}$ and $x + y + z = N$, is an optimal graph for that value of N.

Table 1 lists the optimal graphs for $m = 2$, and $1 \leq n \leq 10$, plus their overheads and overhead factors.

n	c_1	c_2	c_3	o	f
1	1	1	1	1.0000	1.0000
2	2	1	1	2.1667	1.0833
3	2	2	1	3.2000	1.0667
4	2	2	2	4.2000	1.0500
5	3	2	2	5.2381	1.0476
6	3	3	2	6.2500	1.0417
7	3	3	3	7.2500	1.0357
8	4	3	3	8.2667	1.0333
9	4	4	3	9.2727	1.0303
10	4	4	4	10.2727	1.0273

Table 1: Optimal codes for $m = 2$, $1 \leq n \leq 10$, plus overheads and overhead factors.

5.2 Computing Overhead for $m = 3$

When $m = 3$, there are seven potential types of left-hand nodes, denoted by $C_1, \ldots, C_7$, and a graph G may be uniquely specified by its values of c_1 through c_7. Suppose n nodes are removed from G, leaving a residual with just three left-hand nodes. There are 59 residuals that may result that cannot be decoded completely. We enumerate them, their overheads, and the number of ways that they may be generated from G's values of c_i, below:

- **Residuals with three identical left-hand nodes:** An example is (3,0,0,0,0,0,0). Clearly there are seven types of these, one for each C_i, and the overhead of decoding this type of residual is 2. Given G, the number of these types of residual is $\sum_{i=1}^{7} \binom{c_i}{3}$.

- **Residuals with exactly two identical left-hand nodes:** Examples are (2,1,0,0,0,0,0) and (0,1,0,2,0,0,0). There are 42 types of these, six for each C_i, and the overhead of decoding this type of residual is 4/3. Given G, the number of these types of residual is:

$$\sum_{i=1}^{7} \sum_{j=1, j\neq i}^{7} \binom{c_i}{2} c_j \quad = \quad \sum_{i=1}^{7} \binom{c_i}{2} (N - c_i).$$

- **(0,0,1,0,1,1,0):** This graph has exactly two edges entering each right-hand node. Its overhead is one, and the number of these graphs is $c_3 c_5 c_6$.

- **(1,0,0,0,0,1,1), (0,1,0,0,1,0,1) and (0,0,1,1,0,0,1)**: As above, these graphs have exactly two edges entering each right-hand node. Their overhead is one, and the number of these graphs is $c_1 c_6 c_7 + c_2 c_5 c_7 + c_4 c_3 c_7$.

- **(1,1,1,0,0,0,0), (1,0,0,1,1,0,0) and (0,1,0,1,0,1,0)**: These graphs have two right-hand nodes with two edges and one with zero. Their overhead is one, and the number of these graphs is $c_1 c_2 c_3 + c_1 c_4 c_5 + c_2 c_4 c_6$.

- **(0,0,1,0,1,0,1), (0,0,1,0,0,1,1) and (0,0,0,0,1,1,1)**: These graphs have two right-hand nodes with three edges and one with two. Their overhead is one, and the number of these graphs is $c_3 c_5 c_7 + c_3 c_6 c_7 + c_5 c_6 c_7$.

Therefore, the overhead of a graph with $m = 3$ is given by the following rather tedious equation:

$$o(G) \quad = \quad n + \frac{2\sum_{i=1}^{7} \binom{c_i}{3} + \frac{4}{3}\sum_{i=1}^{7} \binom{c_i}{2}(N - c_i)}{\binom{N}{3}} + \frac{c_3 c_5 c_6 + c_1 c_6 c_7 + c_2 c_5 c_7 + c_4 c_3 c_7 + c_1 c_2 c_3 + c_1 c_4 c_5 + c_2 c_4 c_6 + c_3 c_5 c_7 + c_3 c_6 c_7 + c_5 c_6 c_7}{\binom{N}{3}}.$$

Unlike for $m = 2$, minimizing this equation is not straightforward. We discuss how we enumerate graphs to determine the optimal ones below in Section 5.4.

5.3 Calculating Overhead For Arbitrary m

When $m > 3$, there are far too many residual graphs with non-zero overhead to enumerate by hand. Instead, we may enumerate them electronically and calculate their overheads. Using such an enumeration, we may calculate the overhead of any graph $G = (c_1, \ldots, c_{2^m-1})$ as follows. Given a residual graph $R = (r_1, \ldots, r_{2^m-1})$, the number of ways that R may result from downloading n bits from G is:

$$\prod_{i=1}^{2^m-1} \binom{c_i}{r_i}$$

This will be the product of at most m terms, since $\sum_{i=1}^{2^m-1} r_i = m$.

Thus, if R_m is the set of all residual graphs with non-zero overhead, then the overhead of a graph G is:

$$o(G) = n + \left(\frac{\sum_{R \in R_m} \left(o(R) \prod_{i=1}^{2^m - 1} \binom{c_i}{r_i} \right)}{\binom{N}{m}} \right).$$

Of course, the size of R_m increases exponentially, so this technique is only practical for small m. When $m = 4$, there are 2,617 residual graphs with non-zero overhead, and calculating the overhead of a graph takes roughly 2 milliseconds on a Dell Precision 330. When $m = 5$, there are 295,351 residual graphs with non-zero overhead, and calculating the overhead of a graph takes roughly 128 milliseconds. When $m = 6$, there are 105,671,841 residual graphs with non-zero overhead, and calculating the overhead of a graph is too expensive for an exhaustive exploration of the type that we are pursuing. Thus, in the data that follows, we limit m to be less than or equal to 5.

5.4 Finding Optimal and UB$_p$ Codes for $m \in \{3, 4, 5\}$

When $m > 2$, minimizing $o(G)$ mathematically is not straightforward and remains an interesting open problem. Here we use enumeration and heuristics to find the best codes. Unfortunately, graph enumeration is also exponential in n and m; therefore, for all but the smallest values of n and m, we prune the search using a heuristic that we call *perturbation* by p elements. We take the best code for $n-1$ and generate all codes for n that can be derived from the $n-1$ code by subtracting up to p elements from the various c_i, and adding up to $p+1$ elements to the other c_i. For example, the optimal code for $m = 3$ and $n = 32$ is (6,6,5,6,4,4,4). The optimal code for $n = 33$, (6,6,5,6,5,5,3), is derived from the code for $n = 32$ by subtracting one from c_7, and adding one to c_5 and c_6 - a perturbation with $p = 1$. We use this technique to generate what we call UB$_p$ codes, which stands for *Upper Bound, perturbed by* p.

We generated optimal and UB$_p$ codes for the values of m and n listed in Table 2. The total CPU time to generate these 2910 codes is 681 days. Fortunately, however, the enumerations and perturbations are easily parallelizable, and we were able to enlist 89 machines of varying flavors and speeds to cut that time by a factor of 20.

m	Optimal Codes	UB$_p$ Codes	p	CPU time per UB$_p$ code
3	$n \leq 50$	$n \leq 1750$	6	10s
4	$n \leq 10$	$n \leq 1000$	4	3h 49m
5	$n \leq 3$	$n \leq 160$	2	78h 13m

Table 2: Range of optimal and UB$_p$ codes generated

The UB$_p$ codes are not provably optimal. We believe that for each value of m, there is a minimum value of p for which all UB$_p$ codes will be provably optimal, and that p will grow with m. For example, for $m = 3$ in our tests, the maximum value of p for which UB$_p$ $\neq$UB$_{p-1}$ is two. For $m = 4$, that value is 3, and only occurs in one case (deriving $n = 137$ from $n = 138$). Proving what this value is for a given value of m is an open question. We are confident that for $m \leq 4$ in our tests, our UB$_p$ codes are optimal. We doubt that the UB$_2$ codes for $m = 5$ are (in fact, a counterexample is given in

Section 6 below). Unfortunately, we cannot call them optimal until optimality is proved, and thus they represent the upper bounds of the best codes known.

The 2910 codes that we derived are too numerous to present in their entirety here. However, since they are important, we have published them in Technical Report form in [Pla04]. We also list the best codes for $1 \leq n \leq 10$ in Tables 4 through 6 in Appendix 2.

6 Observations

We discuss some of the collective characteristics of our UB$_p$ codes here. This discussion is in the form of questions that arise naturally when one explores these codes.

What are the overheads of the optimal and UB$_p$ codes? To answer this question, we plot the overhead factors of the best codes for $n \leq 100$ in Figure 2. For each value of m, the best overhead factor reaches its high point at $n = 2$, and then descends to approach 1 as n grows. In general, the overhead factor for each value of n is higher when m is larger. This is not strictly true, however. For example, when $n = 4$ the optimal overhead factor is 1.0955 when $m = 4$, whereas the UB$_2$ code for $m = 5$ has an overhead factor of 1.0952.

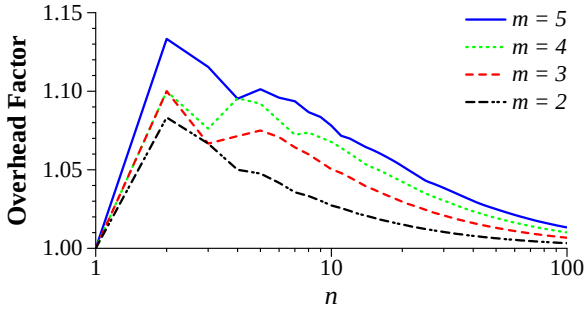


Figure 2: Overhead factors of optimal and UB$_p$ codes for $m \in \{2, 3, 4, 5\}$ and $n \leq 100$.

There are other interesting features of Figure 2. First, each curve is composed of three segments: (1) A rising from an overhead factor of 1 when $n = 1$ to a maximum value, which in each case is when $n = 2$; (2) A period where the factor follows no particular pattern; and (3) A gradual descending of the overhead factor back down to one as n grows further. It is an open problem to provide a better characterization of the optimal overhead. Note that the curve for $m = 4$ rises and falls three distinct times.

Are the best graphs regular? Regularity in LDPC's is discussed in Luby *et al*'s seminal paper on Tornado codes [LMS$^+$97] and thereafter separated into left-regularity and right-regularity [RU03, Sho99]. A graph is *left-regular* if each left-hand node has the same number of incident edges. We define a relaxed property, called *loose left-regularity* (LLR) to be when each left-hand node of a graph has either i or $i + 1$ edges for some value of i. Right-regularity and loose right-regularity (LRR) are defined similarly, except for right-hand nodes rather than left-hand nodes.

Of the 2910 best graphs for $m \in \{3, 4, 5\}$, none are left-regular, and only one is LLR. This is the code for $n = 6$,

$m = 4$, which has four nodes in C_1, and six in C_2. The remaining 2909 graphs are not LLR. Left-regularity as a property for optimality was dismissed early on in [LMS+97], so these results do not come as a surprise.

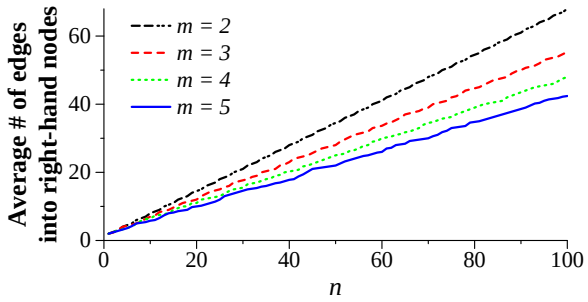


Figure 3: Average number of incoming edges for each right-hand node.

Right-regularity, however, is a different story. *Every* one of the 2910 best graphs is either right-regular or LRR. As plotted in Figure 3, the average number of edges into each right-hand node follows a linear trend for each value of m. Fitting the data to a line, we get that the average number of edges into each right hand node is $0.67n$ for $m = 2$, $0.54n$ for $m = 3$, $0.47n$ for $m = 4$, and $0.46n$ for $m = 5$.

Do the various c_i for best graphs roughly equal each other? When $m = 2$, we proved that in an optimal graph, no c_i could differ from c_j by more than one. It is logical to see if this trend extrapolates to larger m. The answer is that it does not. The first graph to exhibit this property for $m = 3$ is when $n = 18$, and the optimal graph is (4,3,3,3,3,3,2) with an overhead factor of 1.0326 as compared to (3,3,3,3,3,3,3), with an overhead factor of 1.0329. As n increases for all values of $m > 2$, this trend becomes more pronounced. For example, the best graph for $m = 3$ and $n = 1750$ has $c_1 = 289$, and $c_7 = 188$. For $m = 4$ and $n = 200$, the best graph has $c_1 = 20$, and $c_{15} = 6$. Looking at the equation for overhead in Section 5.2, it is easy to see why c_7 would have a lower value than the rest, as it is present in six of the terms in the bottom fraction, whereas the counts in E_2 are present in five terms each, and the counts in E_1 are present in only three each.

A different property that we define here is *Edge Class Equivalence*: If $e(i) = e(j)$, then c_i and c_j differ by at most one. In other words, the counts of distinct nodes in each edge class are roughly equal. For example, the UB_6 graph for $m = 3$ and $n = 1001$ is (166,165,133,165,133,134,108). This graph has edge class equivalence, since the counts of nodes in E_1 is equal to 165 or 166, the counts of nodes in E_2 is equal to 133 or 134, and the count of nodes in E_3 is 108. As with loose right-regularity, *every* one of the 2910 best graphs has edge class equivalence.

Since each graph has edge class equivalence, it makes sense to look at the sizes of the various E_i. Borrowing from the classical definition of Tornado Codes [LMS+97], we can define a vector Λ of graph G to be $(\Lambda_1, \Lambda_2, \ldots, \Lambda_m)$, where Λ_j is the probability that a node in G is an element of E_j. We plot the values of Λ for the 2910 best graphs below in Figure 4.

In the graphs for $m = 3$ and $m = 4$, the Λ vectors clearly converge to constants as n grows. The $m = 5$ graph may exhibit this trend as well, but without looking at higher values of n, we can only speculate. We explore this trend a little further below.

Do the values of c_i or $|E_i|$ grow monotonically with n? For $m = 2$, they do. However, for the other values of m, they

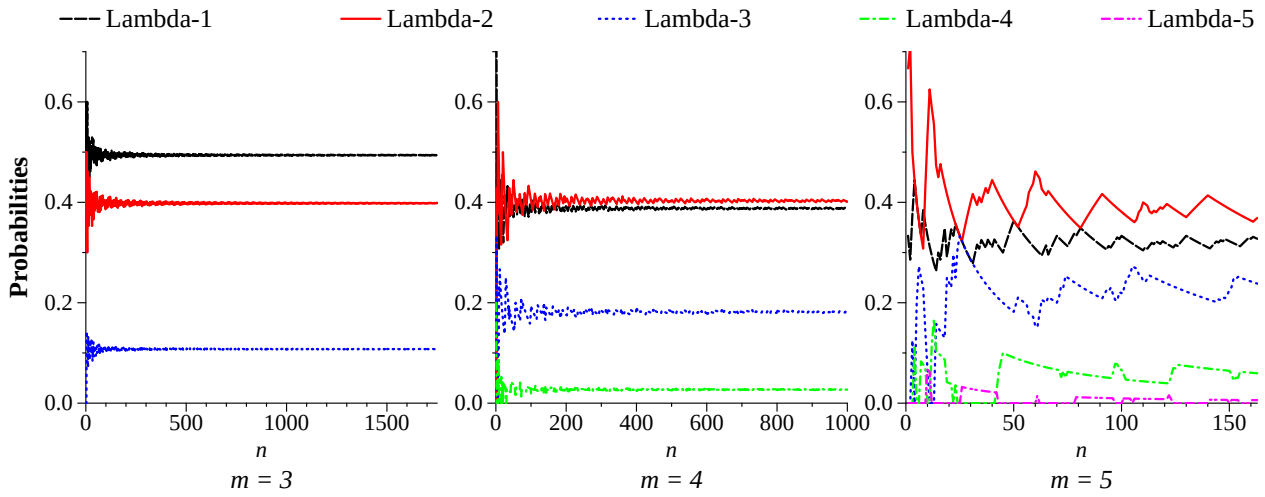


Figure 4: Values of the Λ vectors for best graphs.

do not. Otherwise, UB_0 graphs would all be optimal. As an example, consider $m = 3$ and $n = 1470$, whose best graph is (243,243,195,243,195,195,159). The best graph for $n = 1471$ is (243,243,196,242,196,196,158) – both c_4 and c_7 are less than their values for $n = 1470$. Even more striking, when $m = 4$ and $n = 141$, the best graph has $c_{15} = 5$. For $n = 142$, $c_{15} = 4$, and for $n = 143$, $c_{15} = 3$.

For a given m, is the optimal graph for n a subgraph of the optimal graph for $n+1$? For $m = 2$, the answer is yes, which may be seen easily by looking at how the various c_i grow. However, for $m > 2$, in general, the answer is no. This is because the c_i do not grow monotonically. However, in many specific instances, the answer is yes. Quantifying this, of the 1,749 optimal graphs for $m = 3$ (with $n > 1$), 1,741 of them are supersets of the optimal graphs for $n - 1$. For $m = 4$, 900 of the 999 graphs are supersets, and for $m = 5$, the number is only 114 of 159.

Can anything be learned by modeling the c_i as continuous variables? Suppose we make the assumption that the c_i are continuous variables, and that all c_i in the same edge class are equal to each other. Moreover, as implied by the graphs in Figure 4, we assume that the values of Λ converge to constants as $n \to \infty$. Then, using the last 200 values of n in each case, we average the values of Λ and display them in Table 3. For $m = 2$ and $m = 3$, we used the Maple software package to corroborate the Λ values directly from their overhead equations. With the corroboration, we were also able to prove that $o(G)$ has a local minimum value when the graphs are edge class equivalent. For $m = 4$ and $m = 5$, the equations (fourth and fifth degree polynomials with three and four variables respectively) were too complex for Maple to minimize.

We can use the Λ vectors in Table 3 as a second heuristic to compute graphs. We do this by multiplying each Λ_j by N, and rounding to the nearest integer to yield the various $|E_j|$. If $t = \sum_{j=1}^{m} |E_j| \neq N$, then we can sort $N\Lambda_j - |E_j|$, and either add one to the biggest $N - t$ counts or subtract one from the smallest $t - N$ counts. Then we enumerate all graphs that exhibit both loose right-regularity and edge class equivalence, and keep track of the best graph. This results in far fewer graphs being generated than by perturbation.

For example, when $m = 5$ and $n = 402$, the $|E_j|$ are (131,159,90,25,2). Since there are 10 values each of $c_i \in E_3$,

m	Λ_1	Λ_2	Λ_3	Λ_4	Λ_5
2	0.6667	0.3333			
3	0.4940	0.3983	0.1077		
4	0.3879	0.4030	0.1820	0.0271	
5	0.3210	0.3909	0.2215	0.0620	0.0047

Table 3: Values of the Λ vectors when the c_i are continuous variables and display edge class equivalence.

each of them will equal 90. Similarly, each of the five values of $c_i \in E_4$ will equal 5, and $c_{31} = 2$. The only values that require enumeration are the 5 combinations of $c_i \in E_1$ where four equal 26 and one equals 27, and the 10 combinations of $c_i \in E_2$ where nine equal 16, and one equals 15. That makes 50 graphs, of which only 20 are LRR. The maximum number of graphs that we enumerated in this way was 59,940, for $m = 5$ and $n = 290$. The average number of graphs generated for all $n \leq 1,000$ was 4,007. This is as compared to over 2 million graphs per value of n when generating UB_2.

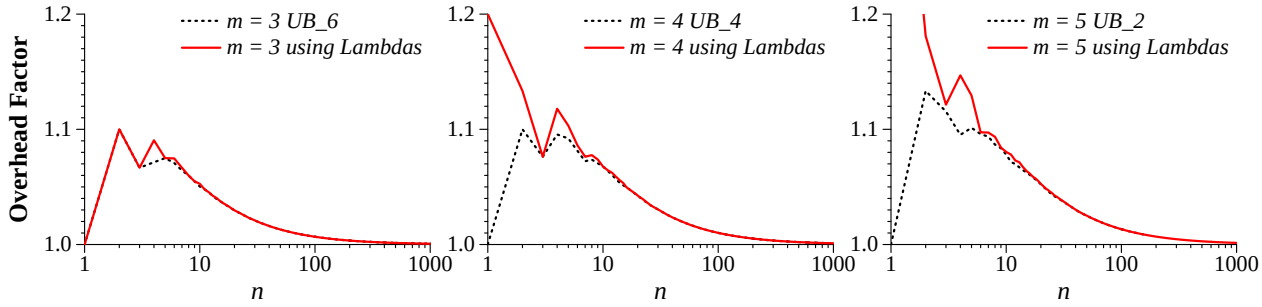


Figure 5: Overhead factors of codes created with the Λ vectors from Table 3 as compared to the UB_p codes.

In Figure 5, we compare the overheads of the codes created using these Λ vectors with the UB_p codes. All three graphs are similar — for very small n (less than 10), the codes created with the Λ vectors have significantly higher overheads than their optimal counterparts. However, as n grows, the two techniques produce similarly performing codes. Interestingly, in one case ($m = 5$, $n = 57$), the Λ-generated code has a lower overhead factor (1.022258) than the UB_2 code (1.022263). This proves that as suspected, the UB_2 codes are not optimal for $m = 5$. Certainly, given the computational complexity of generating UB_2 codes for $m = 5$, for moderate to large values of n, the technique using the Λ vector is preferable. We include the graphs so generated for $n \leq 1000$ in Technical Report [Pla04].

7 Optimal Graphs for $n = 2$

We switch our focus now from fixed m to fixed n. While the graphs for $n = 2$ have greater variety than for $m = 2$, they have one practical limitation — the values of the coding bits are constrained to three distinct values — the value of the first data bit, the value of the second data bit, or the exclusive-or of the two data bits. When downloading, it is only

necessary to download two of these distinct values, after which the data bits (and therefore the rest of the coding bits) may be determined.

A graph that mirrors this line of thinking has the two data bits in l_1 and l_2. The remaining left-hand nodes are coding nodes, and each has exactly one edge from l_i to r_{i-2}. The constraint nodes are partitioned into three sets — those whose coding bits equal l_1, those whose coding bits equal l_2, and those whose coding bits equal $l_1 \oplus l_2$. Node l_1 will have an edge to every constraint node in the first and third groups, and node l_2 will have an edge to every constraint node in the second and third groups. The left-hand nodes whose values equal l_1 compose a set D_1, and consist of l_1 itself plus the coding nodes that equal l_1. There are d_1 nodes in this set. D_2 and d_2 are defined similarly, including l_2 and all coding nodes that equal l_2. Finally, D_3 is composed of the coding nodes that equal $l_1 \oplus l_2$, and there are d_3 of these nodes. Figure 6 depicts such a graph for $n = 2$ and $m = 4$ where $d_1 = d_2 = d_3 = 2$.

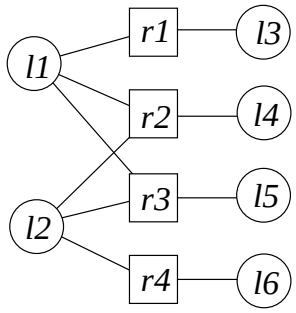


Figure 6: A graph where $n = 2$, $m = 4$, and $d_1 = d_2 = d_3 = 2$.

Suppose we download x bits, and all bits are from the same set (D_1, D_2, or D_3). Then the graph will remain undecoded, since only nodes that belong in that group will be determined. For example, in Figure 6 above, if we only download the two nodes connected to the first constraint, then those will be the only nodes that we may remove from the graph. As soon as we have downloaded nodes from two different sets, we may decode the entire graph.

Let us focus solely on downloading bits in order. Define $p_{d_1,i}$ to be the probability that the first i bits downloaded come from nodes in D_1, and that the $i+1$-st bit does not come from nodes in D_1. Each $p_{d_1,i}$ for $1 \leq i \leq d_1$ is equal to:

$$\begin{aligned} p_{d_1,i} &= \left(\frac{d_1}{m+2}\right)\left(\frac{d_1-1}{m+2-1}\right)\cdots\left(\frac{d_1-(i-1)}{m+2-(i-1)}\right)\left(\frac{m+2-d_1}{m+2-i}\right) \\ &= \frac{\binom{m+2-i}{d_1-i}(m+2-d_1)}{\binom{m+2}{d_1}(m+2-i)} \\ &= \frac{\binom{m+1-i}{d_1-i}}{\binom{m+2}{d_1}}. \end{aligned}$$

We may use $p_{d_x,i}$ for each $1 \leq i \leq d_x$ to calculate the expected value of the overhead:

$$\begin{aligned} o &= \sum_{i=1}^{d_1}(i+1)p_{d_1,i} + \sum_{i=1}^{d_2}(i+1)p_{d_2,i} + \sum_{i=1}^{d_3}(i+1)p_{d_3,i} \\ &= \frac{m+3}{m+3-d_1} + \frac{m+3}{m+3-d_2} + \frac{m+3}{m+3-d_3} - 2. \end{aligned}$$

Simple math yields that this equation is minimized when d_1, d_2, and d_3 differ by at most one. Figure 7 plots the overhead of these graphs as a function of m. Note that although the rate of these codes approaches zero as $m \to \infty$, the overhead factors appear to approach 1.25. Indeed, if we set $d_1 = d_2 = d_3 = \frac{m+2}{3}$, the equation for overhead becomes:

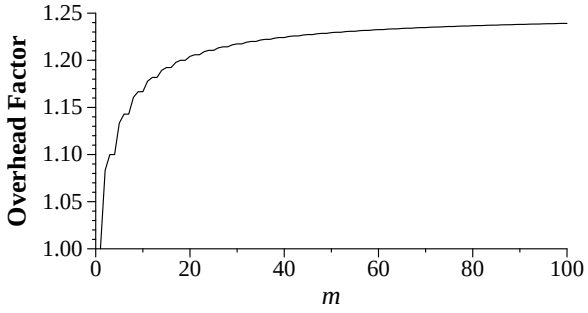


Figure 7: The overhead factors of optimal graphs when n = 2.

$$\begin{aligned} o &= \frac{3(m+3)}{m+3-\frac{m+2}{3}} - 2 \\ &= \frac{3(m+3)}{\frac{3m+9-m-2}{3}} - 2 \\ &= \frac{9m+27}{2m+7} - 2 \\ &= \frac{5m+13}{2m+7} \end{aligned}$$

The limit of this as $m \to \infty$ is $\frac{5}{2} = 2.5$, yielding an overhead factor of $\frac{2.5}{2} = 1.25$.

8 Graphs for $n = 3$: A limitation of having only m constraints

Extrapolating from the previous section, suppose $n = 3$, and our three data bits are labeled b_1, b_2 and b_3. Now, there are only seven possible values for a node: b_1, b_2, $(b_1 \oplus b_2)$, b_3, $(b_1 \oplus b_3)$, $(b_2 \oplus b_3)$, and $(b_1 \oplus b_2 \oplus b_3)$. Of the $\binom{7}{3} = 35$ combinations of three distinct values, there are seven that cannot decode the three data bits:

1. b_1, b_2 and $(b_1 \oplus b_2)$.
2. b_1, b_3 and $(b_1 \oplus b_3)$.
3. b_2, b_3 and $(b_2 \oplus b_3)$.
4. b_1, $(b_2 \oplus b_3)$ and $(b_1 \oplus b_2 \oplus b_3)$.
5. b_2, $(b_1 \oplus b_3)$ and $(b_1 \oplus b_2 \oplus b_3)$.
6. b_3, $(b_1 \oplus b_2)$ and $(b_1 \oplus b_2 \oplus b_3)$.

7. $(b_1 \oplus b_2)$, $(b_1 \oplus b_3)$, and $(b_2 \oplus b_3)$.

Any combination of four distinct values will allow one to decode the three data bits. Therefore, if we have $n = 3$ and $m = 4$, and we encode by having each of the seven bits contain a distinct value, then we can always decode with three bits when we do not receive one of the 3-bit combinations listed above. Otherwise, we will decode in four bits. The overhead of such a decoding scheme is $\frac{28*3+7*4}{35} = \frac{112}{35} = \frac{13}{5} = 3.2$.

Unfortunately, the optimal graph for $n = 3$ and $m = 4$ has an overhead of $\frac{113}{35} = 3.2286$, meaning that the optimal graph does not decode optimally! To see why this is true, consider the graph in Figure 8. This graph's overhead is 3.2286. Suppose we download nodes D, E, and G. Since $(b_1 \oplus b_2) \oplus (b_1 \oplus b_3) \oplus (b_1 \oplus b_2 \oplus b_3)$ is equal to b_1, we should be able to decode all the bits from these three values. However, when we remove nodes D, E, and G from the graph, all constraint nodes have more than one edge still incident to them, and we cannot decode. This is where the extra $\frac{1}{35}$ of overhead comes from, and there is *no* graph with seven left-hand nodes and four right-hand nodes that avoids this problem.

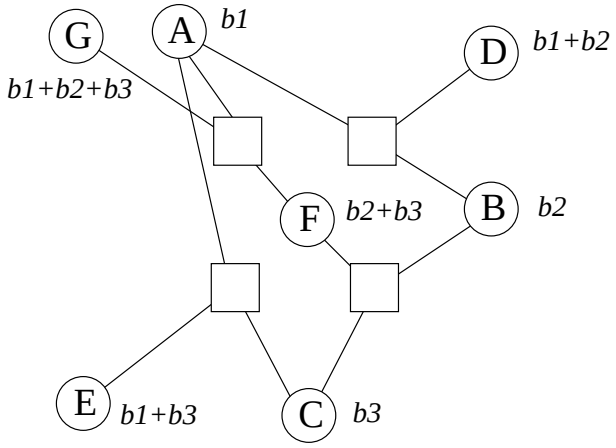


Figure 8: An optimal graph for $n = 3$ and $m = 4$.

To fix this, suppose we add a fifth constraint to the graph, which is connected to nodes D, E, and G. Now, although the graph no longer fits the standard Tanner Graph description (nor does it fit our definition of a Systematic graph), it does decode optimally. We do not explore this fact or these codes further; however, we present it here as a curiosity, and as a seed of future work on graph-based coding.

9 Related Work and Brief Discussion

Since the landmark Tornado Code paper in 1997 [LMS+97], the focus of most LDPC researchers has been achieving asymptotic optimality. There are rare exceptions, such as a paper analyzing certain classes of finite codes [DPT+02], a paper defining classes of sub-optimal codes that perform well in practice [RN04], and our previous foray into Monte-Carlo generation and analysis of finite-length codes [PT04].

The exceptions are rare, because the asymptotic is an easier case in which to succeed. To illustrate this, consider Figure 4. Whereas the simple graph construction using the Λ vectors fails to produce graphs that perform nearly optimally for small n, as n grows this graph construction method performs very well. It would not be too difficult to prove that the overhead factors of these graphs indeed approach one as $n \to \infty$, meaning that they are asymptotically optimal. Determining true optimality for finite n remains an open question, one that we will continue to address.

An important question to ask, however, is: **How important is optimality?** For example, when $m = 4$ and $n = 100$, the overhead of the UB_4 code is 101.01073, and the overhead of the code generated by the Λ vector is 101.01088. Are there any scenarios in which that extra 0.00015 is significant? Likely not. However, consider the case where $m = 4$ and $n = 4$, and a 1 GB file is broken up into 256 sets of eight blocks (4 data and 4 coding) that are distributed among faulty servers in the wide-area. When a client tries to download this file, the difference between the optimal overhead of 4.382 and the suboptimal overhead of 4.471 (generated by the Λ vector) will be significant indeed. Until true optimality is determined, suboptimal constructions such as the UB_p and Λ codes in this paper will be extremely useful. However, until optimal, finite codes are fully understood, the field of LDPC codes will continue to be an open research area.

10 Conclusion

We have performed an exploration of optimal and nearly-optimal LDPC erasure codes for small values of n and m. We have detailed three mechanisms for determining the overhead of a code exactly, and used these determinations, plus enumeration techniques to generate optimal codes for $m = 2$ and $n = 2$. For $m \in \{3, 4, 5\}$, we have generated codes with the best known upper bounds for n less than or equal to 1750, 1000, and 1000 respectively.

As part of our exploration, we have made the following observations, which should be an aid to others who need to explore these codes:

- Optimal codes are not left-regular.
- However, the best codes appear to be loosely right regular.
- They also appear to have a property that we call *edge class equivalence*. Using the above two properties can be a great aid in pruning enumerations in order to discover good codes.
- The various counts of distinct types of left-hand nodes do not have to equal each other for a graph to be optimal.
- In the best graphs with a fixed m, the Λ vector of edge count probabilities, which is the backbone of classic LDPC coding theory [LMS+97, RU03, WK03], appears to converge to a constant as $n \to \infty$. This vector may also be used to generate graphs that perform very close to optimal as n grows.
- For $n > 2$, the iterative decoding technique of LDPC's cannot decode optimally. It is an open question of how to modify the standard definition of LDPC's so that they can decode better.

The quantification of optimal parity check codes for arbitrary values of n and m remains an open question. In this paper, we have defined uppoer bounds, and we have helped to narrow the range of n and m for which we don't know optimality. We will continue work to narrow this range by trying to understand the properties and structure of optimal codes, and using them to prune the search so that it is a tractable endeavor.

References

[DPT+02] C. Di, D. Proietti, I. E. Telatar, T. J. Richardson, and R. L. Urbanke. Finite-length analysis of low-density parity-check codes on the binary erasure channel. *IEEE Transactions on Information Theory*, 48:1570–1579, June 2002.

[FMS+04] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *DSN-04: International Conference on Dependable Systems and Networks*, Florence, Italy, 2004. IEEE.

[GWGR04] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *DSN-04: International Conference on Dependable Systems and Networks*, Florence, Italy, 2004. IEEE.

[LMS+97] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing,*, pages 150–159, El Paso, TX, 1997. ACM.

[PD04] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience*, to appear, 2004.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[Pla04] J. S. Plank. Enumeration of small, optimal and near-optimal parity-check erasure codes. Technical Report UT-CS-04-535, Department of Computer Science, University of Tennessee, November 2004.

[PT04] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN-2004: The International Conference on Dependable Systems and Networks*, pages 115–124, Florence, Italy, June 2004. IEEE.

[Riz97] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[RN04] V. Roca and C. Neumann. Design, evaluation and comparison of four large FEC Codecs, LDPC, LDGM, LDGM staircase and LDGM triangle, plus a Reed-Solomon small block FEC Codec. Technical Report RR-5225, INRIA Rhone-Alpes, June 2004.

[RU03] T. Richardson and R. Urbanke. Modern coding theory. Draft from `lthcwww.epfl.ch/papers/ics.ps`, August 2003.

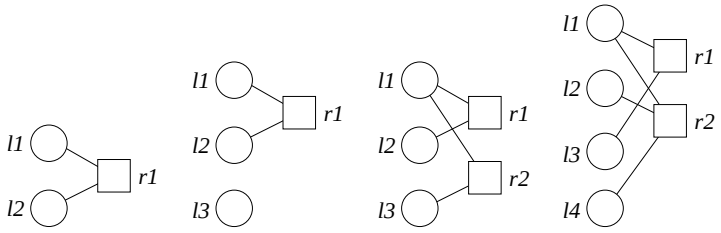
[RWE+01] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[Sho99] M. A. Shokrollahi. New sequences of linear time erasure codes approaching the channel capacity. In *Proceedings of AAECC-13, Lecture Notes in CS 1719*, pages 65–76, New York, 1999. Springer-Verlag.

[WK03] S. B. Wicker and S. Kim. *Fundamentals of Codes, Graphs, and Iterative Decoding*. Kluwer Academic Publishers, Norwell, MA, 2003.

[ZL02] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 330–339, October 2002.

11 Appendix 1 – Example of recursive overhead calculation



(a): G_1 (b): G_2 (c): G_3 (d): G_4

Figure 9: Example graphs for determining overhead.

11.0.1 Example 1: G_1

Let G_1 be the graph with $n = 1$, $m = 1$, and edges $\{(l_1, r_1), (l_2, r_1)\}$, depicted in Figure 9(a).

This is a systematic code, where either node may be the data or coding bit. To calculate G_1's overhead, we first simulate downloading bit l_1 by removing l_1 and edge (l_1, r_1) from the graph. This leaves node r_1 with only one incident edge, (l_2, r_1) which means that we can remove r_2 and its edge from the graph and assign node l_2's value. Thus, $R(G_1, l_1)$ is simply l_2, whose overhead is zero.

Similarly, downloading l_2 leaves only l_1 as a residual graph. Therefore, the overhead of G_1 is:

$$
\begin{aligned}
o(G_1) &= \left(\sum_{i=1}^{2}(1+o(R(G_1,l_i)))\right)/2 \\
&= ((1+o(R(G_1,l_1)))+(1+o(R(G_1,l_2))))/2 \\
&= ((1+0)+(1+0))/2 \\
&= 1.
\end{aligned}
$$

11.0.2 Example 2: G_2

Let G_2 be the graph with $n = 2$, $m = 1$ and edges $\{(l_1, r_1), (l_2, r_1)\}$, depicted in Figure 9(b). Note, this is a graph that represents a residual graph of a download, where we have already determined l_3's value, but it has not been downloaded yet. To decode, if we download either nodes l_1 or l_2, then we may assign the value of all nodes, and decoding is complete. If we download node l_3, then we simply remove that node from the graph, and are left with graph G_1 as a residual. Therefore, the overhead of decoding G_2 is:

$$
\begin{aligned}
o(G_2) &= ((1+o(R(G_2,l_1)))+(1+o(R(G_2,l_2)))+(1+o(R(G_2,l_3))))/3 \\
&= ((1+0)+(1+0)+(1+o(G_1))/3 \\
&= (1+1+2)/3 = 4/3.
\end{aligned}
$$

11.0.3 Example 3: G_3

Let G_3 be the graph with $n = 1$, $m = 2$ and edges $\{(l_1, r_1), (l_1, r_2), (l_2, r_1), (l_3, r_2)\}$, depicted in Figure 9(c). s is a simple systematic replication code, where l_1 is the data bit, and l_2 and l_3 are the coding bits. To decode, when any of the three nodes is downloaded, the values of the other two may be assigned. Therefore, the overhead of G_3 is:

$$
\begin{aligned}
o(G_3) &= ((1+o(R(G_3,l_1)))+(1+o(R(G_3,l_2)))+(1+o(R(G_3,l_3))))/3 \\
&= ((1+0)+(1+0)+(1+0))/3 \\
&= 1.
\end{aligned}
$$

11.0.4 Example 4: G_4

Finally, let G_4 be the graph with $n = 2$, $m = 2$ and edges $\{(l_1, r_1), (l_1, r_2), (l_2, r_2), (l_3, r_1), (l_4, r_2)\}$, depicted in Figure 9(d). This is systematic code with data bits in l_1 and l_2, and coding bits in l_3 and l_4. To decode, we look at the residual graphs of downloading each l_i. Downloading l_1 leaves us with a graph equivalent to G_2. Downloading l_2 leaves us with a

graph equivalent to G_3, and since l_4 is equivalent to l_2, downloading it also leaves us with G_3 as a residual graph. Finally, downloading l_3 leaves us with a graph equivalent to G_2 as a residual. Therefore, the overhead of G_4 is:

$$\begin{aligned} o(G_4) &= \left(\sum_{i=1}^{4}(1+o(R(G_4,l_i)))\right)/4 \\ &= ((1+o(G_2))+(1+o(G_3))+(1+o(G_2))+(1+o(G_3)))/4 \\ &= ((1+4/3)+(1+1)+(1+4/3)+(1+1))/4 \\ &= (26/3)/4 = 13/6 = 2.16667. \end{aligned}$$

Thus, the overhead factor of G_4 is 13/12 = 1.0833.

Appendix 2 – Optimal Codes for $m \in \{3,4,5\}$

n	c_1	c_2	c_3	c_4	c_5	c_6	c_7	o	f
1	1	0	1	1	0	1	0	1.0000	1.0000
2	1	1	1	1	1	0	0	2.2000	1.1000
3	1	1	1	1	1	1	0	3.2000	1.0667
4	1	1	1	1	1	1	1	4.2857	1.0714
5	2	1	1	1	1	1	1	5.3750	1.0750
6	2	2	1	1	1	1	1	6.4246	1.0708
7	2	2	1	2	1	1	1	7.4500	1.0643
8	2	2	2	2	1	1	1	8.4788	1.0598
9	2	2	2	2	2	1	1	9.4939	1.0549
10	2	2	2	2	2	2	1	10.5035	1.0503

Table 4: Optimal codes for $m = 3$, $1 \leq n \leq 10$, plus overheads and overhead factors.

n	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	o	f
1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1	1.0000	1.0000
2	1	1	0	1	0	0	1	1	0	0	1	0	0	0	0	2.2000	1.1000
3	1	1	0	1	0	0	1	0	1	1	0	1	0	0	0	3.2286	1.0762
4	1	1	0	1	0	0	1	1	1	1	0	1	0	0	0	4.3821	1.0955
5	1	1	1	1	0	0	1	1	1	1	0	1	0	0	0	5.4603	1.0921
6	1	1	1	1	1	1	0	1	1	1	0	1	0	0	0	6.4881	1.0813
7	1	1	1	1	1	1	0	1	1	1	0	1	0	0	1	7.5061	1.0723
8	1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	8.5894	1.0737
9	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	9.6350	1.0706
10	1	2	1	1	1	1	1	1	1	1	1	1	1	0	0	10.6771	1.0677

Table 5: Optimal codes for $m = 4$, $1 \leq n \leq 10$, plus overheads and overhead factors.

n	c_i, for i from 1 to 31	o	f
1	10100100000100010000000010000000	1.0000	1.0000
2	10101000010000010001000010000000	2.2667	1.1333
3	00110101100000011000000000001000	3.3464	1.1155
4	01111001100000011000000000000010	4.3810	1.0952
5	01111001100001011000010000000000	5.5063	1.1013
6	01111001100001011000010001000000	6.5753	1.0959
7	01111001100001011000010001001000	7.6553	1.0936
8	11111001100000011100001000101000	8.6938	1.0867
9	11111001100000011100101010100000	9.7522	1.0836
10	11111001110100011001010100000001	10.7807	1.0781

Table 6: Optimal codes for $m = 5$, $1 \leq n \leq 3$, and UB_2 codes for $m = 5$, $4 \leq n \leq 10$, plus overheads and overhead factors.