

# Generating Good Degree Distributions for Sparse Parity Check Codes using Oracles

Jeffrey Considine  
jconsidi@cs.bu.edu

October 30, 2001

## Abstract

Fast forward error correction codes are becoming an important component in bulk content delivery. They fit in naturally with multicast scenarios as a way to deal with losses and are now seeing use in peer to peer networks as a basis for distributing load. In particular, new irregular sparse parity check codes have been developed with provable average linear time performance, a significant improvement over previous codes. In this paper, we present a new heuristic for generating codes with similar performance based on observing a server with an oracle for client state. This heuristic is easy to implement and provides further intuition into the need for an irregular heavy tailed distribution.

## 1 Introduction

Content providers are continuously looking for better scalable ways of delivering bulk content. Fast forward error correction codes are becoming an important component for this in both the idealized multicast scenarios and the increasingly popular peer to peer networks. Irregular sparse parity check codes have recently been shown to be a very good choice for implementing erasure codes, codes optimized for dealing with losses instead of bit errors. Given the correct distributions for encoding, these codes are very fast and have provable probabilistic performance guarantees.

The analysis involved in generating good distributions is fairly complicated. We present a simple heuristic that can be easily implemented and tested in a single day given a pre-existing coding library. The performance of codes generated using this heuristic is comparable to the performance of codes in the literature.

## 2 Technical Background

Forward error correction (FEC) codes aim to provide reliability with little or no feedback by adding additional data providing redundancy. Specifically, we are concerned with erasure codes, codes which handle losses of whole blocks of data at a time, as opposed to more general FEC codes which deal with more arbitrary bit errors. The canonical scenario using such code has a server encoding an input file and sending data to one or more clients until the can decode its input. In such a scenario, no feedback is necessary as clients facing losses merely wait until they have received enough redundant data to decode. Such scenarios have been used as the basis for content distribution over unicast [6], multicast [2] and peer to peer networks [8].

Given a set of  $n$  input symbols (blocks),  $m > n$  output symbols are produced. Given a sufficiently large subset of the output symbols, the original  $n$  input symbols can be recovered. In the case of the original Information Dispersal Algorithm [5], any set of  $n$  output symbols sufficed to recover the  $n$  input symbols. Recent probabilistic codes such as Tornado codes [4] require about  $n(1 + \epsilon)$  output symbols for recovery where  $\epsilon$  is generally 0.05 or less but these codes run significantly faster. The term  $\epsilon$  is often referred to as the *decoding inefficiency* as it represents the overhead of output symbols received that did not contribute to decoding.

Received	$x_1$	$x_2 \oplus x_3$	$x_1 \oplus x_3$	$x_2 \oplus x_4$	$x_1 \oplus x_5 \oplus x_6$	$x_4 \oplus x_6$
Reduced	$x_1$	$x_2 \oplus x_3$	$x_3$	$x_4$	$x_5 \oplus x_6$	$x_6$
Remaining	$x_1$	$x_1$ $x_2 \oplus x_3$	$x_1$ $x_2$ $x_3$	$x_1$ $x_2$ $x_3$ $x_4$	$x_1$ $x_2$ $x_3$ $x_4$ $x_5 \oplus x_6$	$x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$

Figure 1: An Example of Decoding with Parity Check Codes

Tornado codes are probably the best known example of irregular sparse parity check erasure codes with strong probabilistic performance guarantees. Sparse parity check codes are based on output symbols which are the parity of various sets of input symbols. The number of input symbols combined to generate an output symbol (its “degree”) varies considerably. Most output symbol degrees are low but efficient distributions are generally heavy tailed - they have a few symbols of degree on the order of  $n$ . Since low degree symbols are fairly common, these codes are generally decoded with a simple recovery rule. Given an output symbol of degree one (i.e. a recovered input symbol), any other output symbols containing that input symbol may have their degree reduced by one. As output symbols are reduced to degree one, this rule is repeatedly applied until all of the input symbols are recovered. It is trivial to show that this decoding process runs in expected time  $O(n * (1 + \epsilon) * \bar{d})$  where  $\bar{d}$  is the average output symbol degree. Towards the end of the decoding process, new output symbols can trigger the recovery of many input symbols - the “whirlwind” of decoding at the end gives Tornado codes their name. Figure 1 gives an example of the decoding process.

The most important contribution of the work on Tornado was not their low average decoding inefficiency but the accompanying analysis proving it (as opposed to empirical results). This analysis also showed that the variation in the decoding process is small - once  $n(1 + \epsilon)$  packets are received, the probability of unrecovered input symbols remaining drops exponentially as more output symbols become available. It was also shown that for any  $\epsilon$  and sufficiently large  $n$ , there exists a code with decoding inefficiency  $\epsilon$  and  $\bar{d}$  growing logarithmically in  $1/\epsilon$ .

### 3 Assumptions and Basic Scenarios

#### 3.1 One Sided Degree Distributions

The heuristics of this paper work to optimize only the output symbol degree distribution. Later analysis of Tornado codes such as [3] also optimized relative input symbol probabilities. Instead, we assume input symbols have equal probabilities when constructing output symbols. An important side effect of this choice is that all output symbols of the same degree have equal probability.

#### 3.2 Client Server Model

In our model, the server has an input file to send to a client over a lossy channel. The input file is broken up into input symbols and encoded with one output symbol per packet sent to the client. The client either receives the entire packet or it is lost (corrupted packets are treated as losses). Upon receiving a packet, the client applies the recovery rule until it can recover no more input symbols. More expensive decoding algorithms such as gaussian elimination are not used. The only feedback from the client is to notify the server that the client has recovered the entire file.

Note that codes generated for this unicast based model are equally applicable to multicast or peer to peer models.

## 4 Oracle Based Heuristics

To generate a good degree distribution, we consider an offline simulation of a server given an oracle to take the place of feedback from the client. Using the oracle for guidance, the server can pick output symbols that are more likely to be useful in the client’s decoding process. The output symbol degree distribution of this “enlightened” server is sampled in hopes that the sample distribution is more efficient than the server’s original degree distribution. In Section 4.4, we discuss how to iterate this process to generate a good distribution starting from an arbitrary bad distribution.

An oracle providing the server with full knowledge of client state or packet loss is too powerful to give a useful sample. Such a server will know exactly which output symbols were received and can easily pick input symbols the client has not recovered, thus giving a decoding inefficiency of 0 and an average degree of 1. While clearly optimal in performance, it is equally clear that this is an unobtainable goal without feedback. This distribution is also provably bad - sending only degree one output symbols is essentially the well known Coupon Collectors Problem so it requires  $\Theta(n \lg n)$  output symbols on average. To avoid this problem, we restrict the use of information from oracles to picking output symbol degrees. In essence, the oracle is used to suggest that the probability of particular output symbol degrees should be increased.

The oracle we choose tells the server how many input symbols the client has recovered. The number of input symbols the client has recovered is not an accurate measure of how close the client is to completion, but it allows an easy calculation of the output symbol degree maximizing the probability of being reduced to degree one and possibly allowing additional input symbols to be recovered.

### 4.1 A Brief Example

As a brief example, we consider the case when more than half of the input symbols have been recovered and compare the usefulness of degree one and two output symbols. Let the number of input symbols recovered by  $xn$ , where  $0.5 < x < 1$ . The immediate utility of a degree one output symbol is  $1 - x$ . The immediate utility of a degree two output symbol is  $2 * (1 - x) * x$ . The degree two output symbols have greater immediate utility for  $0.5 < x < 1$ , i.e. exactly the situation in question. If a degree one output symbol is not immediately useful, it will never be useful since its input symbol is already recovered. On the other hand, a degree two symbol with neither input symbol recovered might be useful later. Given these probabilities, one might argue that a server knowing its client has recovered at least half of its input symbols should not send any degree one output symbols since degree two output symbols are always better in both immediate utility and overall utility.<sup>1</sup>

### 4.2 Relevant Probability

We call the expected number of input symbols recovered after receiving  $i$  output symbols  $f(i)$ . The number of additional input symbols recovered after receiving the  $i$ th packet is  $f'(i)$ . Figure 2 shows  $f$  for some distributions we generate.

$$\begin{aligned} f(0) &= 0 \\ f'(i) &= f(i) - f(i - 1) \\ f(i) &= \sum_{j=1}^i f'(j) \end{aligned}$$

Examining  $f'(i)$ , we can break it into two components, the probability that more input symbols will be recovered upon receiving the  $i$ th output symbol,  $g(i)$ , and the expected number of input symbols recovered

---

<sup>1</sup>Typical decoding processes do not recover half of the input symbols until the end (often using one of the last few output symbols) so this is not as great an optimization as it might seem.

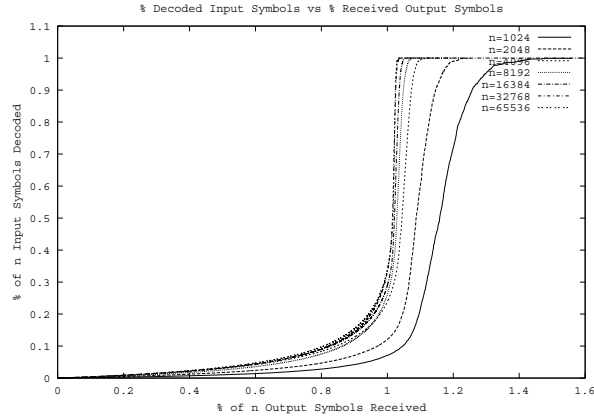


Figure 2: Decoding Progress  $f(i)$

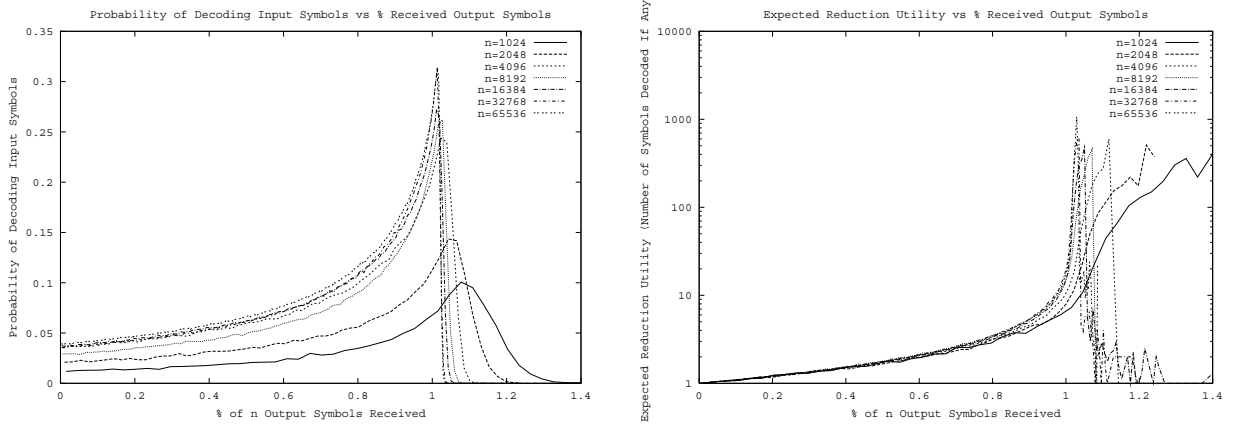


Figure 3: blah. (left) Reduction Probability -  $g(i)$ , (right) Reduction Utility -  $h(i)$

in that event,  $h(i)$ . Note that using only the recovery rule, new input symbols can be recovered if and only if a newly arrived output symbol is reduced to degree one.

$$f'(i) = g(i) * h(i)$$

$$f(i) = \sum_{j=1}^i (g(j) * h(j))$$

Simple heuristic optimizations of  $f$  may be based on optimizing  $g$  and  $h$ . Note that  $g$  and  $h$  are not independent - increasing  $g$  at a particular  $i$  tends to decrease  $h$  for later  $i$ .

Figure 3 shows  $g$  and  $h$  for some codes we later generate.  $g$  tends to be low for most of the process until most of the output symbols have been received.  $g$  peaks just as decoding is expected to complete and then quickly drops down to zero when most decoding processes are expected to have already finished.  $h$  is similarly low for most of the process with an even sharper spike at the end - it is not uncommon for the last reduction to recover between one third and one half of all the input symbols.

Given the number of input symbols recovered expressed as a fraction  $x$  (i.e.  $xn$  input symbols are recovered) and assume  $n$  is large relative to the output symbol degree  $d$ . The probability  $P_i(x, d)$  that an output symbol of degree  $d$  is reduced to degree  $i$  is

$$\begin{aligned} P_i(x, d) &= x^{d-i} * (1-x)^i * \binom{d}{i} \\ g(i) &= P_1(x, d) \\ &= x^{d-1} * (1-x) * d \\ P_0(x, d) &= x^d \end{aligned}$$

A simple calculation shows that  $g(i)$  is maximized when  $d = \left\lceil \frac{x}{1-x} \right\rceil$ .<sup>2</sup> Note that  $P_0(x, d) = P_1(x, d)$  when  $d = \frac{x}{1-x}$ , so maximizing  $g(i)$  also results in a significant number of output symbols that are immediately discarded because they are completely redundant. Section 4.3 elaborates on these inefficiencies.

### 4.3 Using Oracle by Itself

Since maximizing  $g(i)$  results in so many completely useless symbols, the decoding inefficiency is high (around 45%) if  $g(i)$  is always maximized. Another reason for this high inefficiency is that always maximizing  $g(i)$  results in low  $h(i)$ s. We call using the oracle to maximize  $g$  “Oracle A”. A better approach is to only maximize  $g(i)$  some of the time.

An example of how always maximizing  $g$  is inefficient comes from trying  $d = \left\lceil \frac{-1}{\ln x} \right\rceil$ .<sup>3</sup> This  $d$  is the same as  $\left\lceil \frac{x}{1-x} \right\rceil$  or one more. For  $x$  where this makes a difference,  $g$  is lower. However, the probability of all input symbols being redundant is lower and  $h$  is larger overall bringing the decoding inefficiency down to 34%, an 11% improvement. We call this use of the oracle “Oracle B”.

Figure 4 illustrates the differences between these two oracle choices. Oracle B always its degree first (top left). Before Oracle A matches degrees gain, Oracle B has a lower  $g$  function (top right). However, when plotting versus the number of output symbols received instead of the number of input symbols decoded, Oracle B progresses more quickly (bottom left). From the first time when the oracles disagree, the  $h$  function of Oracle B is always higher (bottom right). Despite a lower  $g$  function, this results in a better decoding inefficiency for Oracle B.

We use Oracle B for all later experiments.

### 4.4 Generating Distributions

Since using the oracle all the time is so inefficient, we aim to use the oracle for only a small portion of the output symbols. Instead of using the oracle to determine the whole degree distribution, we use it to give a few hints to improve an existing distribution. Essentially, we use the oracle to say “If we had a few more output symbols of this degree, more input symbols would have been recovered.”

Figure 5 shows pseudo-code for simulating a session with the oracle follows. We use these simulations as a subroutine in improving an existing distribution. Given a distribution, we run the oracle simulation several times (10 in our experiments) while sampling the output symbol degree every time an output symbol is generated. The sample distribution is used as the new distribution. If the oracle is not used excessively, the decoding inefficiency of the sample distribution will usually be better than the original distribution when used without the oracle.

---

<sup>2</sup>For fixed  $x$ ,  $P_1(x, d)$  has one local maxima, the global maximum. If  $d = \frac{x}{1-x}$ ,  $P_1(x, d) = P_1(x, d+1)$  so the global maximum is in the range  $(d, d+1)$ . It follows that  $d = \left\lceil \frac{x}{1-x} \right\rceil$  maximizes  $P_1(x, d)$  for integer  $d$ .

<sup>3</sup>The global maximum of  $P_1(x, d)$  is  $d = \frac{-1}{\ln x}$  of  $P_1(x, d)$  when  $d$  is not restricted to integers.

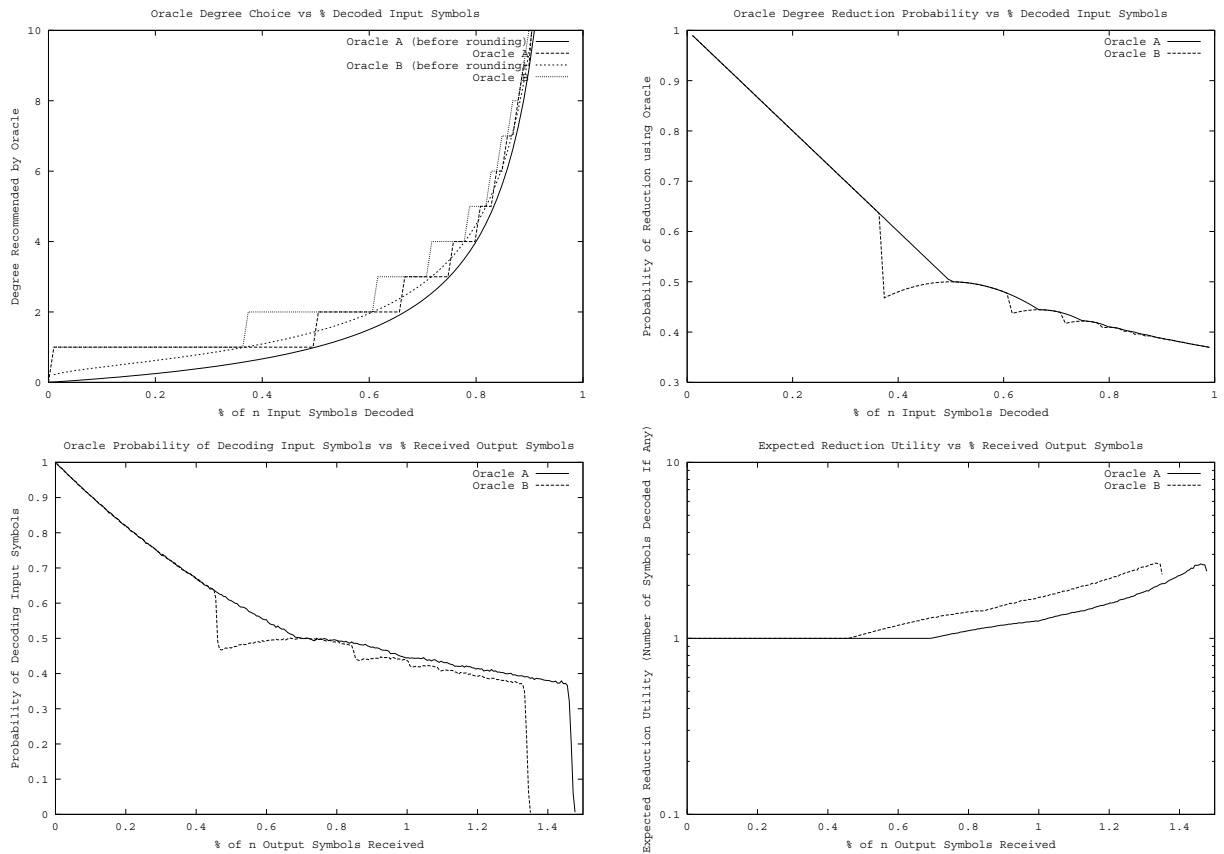


Figure 4: Reductions using Only Oracles. (top left) Oracle Choices, (top right) Oracle Choice Reduction Probability, (bottom left) Reduction Probability -  $g(i)$ , (bottom right) Reduction Utility -  $h(i)$

```

while client not decoded

    decide whether to use oracle

    if using oracle
        output symbol degree chosen by oracle
    else
        output symbol degree chosen from pre-existing distribution

    generate output symbol and pass it to client
    client decodes as much as possible

```

Figure 5: Pseudo-code for Simulations Using an Oracle

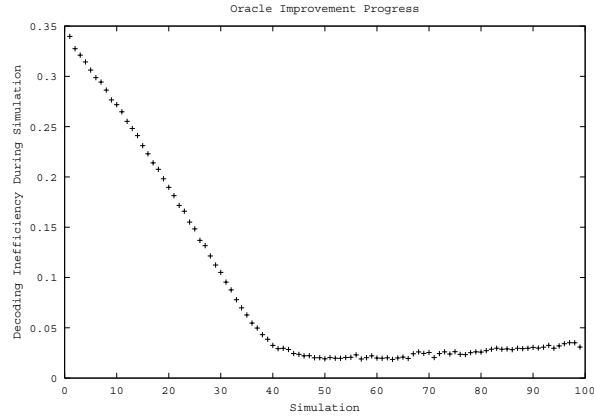


Figure 6: Decoding Inefficiency while Generating Distribution

$n$	$2^{10}$	$2^{11}$	$2^{12}$	$2^{13}$	$2^{14}$	$2^{15}$	$2^{16}$	$2^{17}$	$2^{18}$	$2^{19}$
Average Inefficiency	0.175	0.101	0.060	0.040	0.045	0.024	0.024	0.017	0.017	0.014
Standard Deviation	0.066	0.029	0.014	0.008	0.042	0.011	0.007	0.003	0.013	0.009

Table 1: Statistics for Oracle Distributions (100 trials)

To create a good distribution from scratch, we start with the distribution of all degree one output symbols and repeatedly apply the distribution improvement process of the previous paragraph. Initially, the oracle is used almost exclusively to allow large scale changes to the distribution. As the improvement process is repeated, the oracle is used less to allow finer adjustments and minimize the number of useless symbols from the oracle.<sup>4</sup>

As seen in Figure 3,  $h(i)$  is very high when  $i$  is around  $n$ . The degree distributions used for these figures were generated using the techniques described later in this paper but the shapes of  $g$  and  $h$  are typical of efficient irregular sparse erasure codes. By waiting until  $i$  is close to  $n$  (most output symbols have already been received), the expected benefits of maximizing  $g$  are much greater - the losses associated with more useless output symbols are less and  $h$  is not affected as much since most of the output symbols were not changed. We use these observations as the basis for deciding to use the oracle.

In our experiments, we run 99 simulations using the oracle less each time. For the  $i$ th simulation, the oracle is not used until  $i\%$  of all input symbols have been decoded. From that point on, the oracle is used exclusively. This restricts the oracle to tuning the end of the decoding process as the simulations progress, with the side effect that the oracle is only able to suggest high symbol degrees as the simulations proceed. Figure 6 shows the decoding inefficiencies using the oracle during these simulations for  $n = 32768$ . Table 1 shows the decoding inefficiencies of the resulting distributions for various  $n$ .

Table 1 shows the decoding inefficiencies resulting from this process for various  $n$ . Figure 7 shows the degree distributions of oracle distributions. Note the heavy tail present in these distributions.

<sup>4</sup>In a way, this is similar to the hill climbing process simulated annealing.

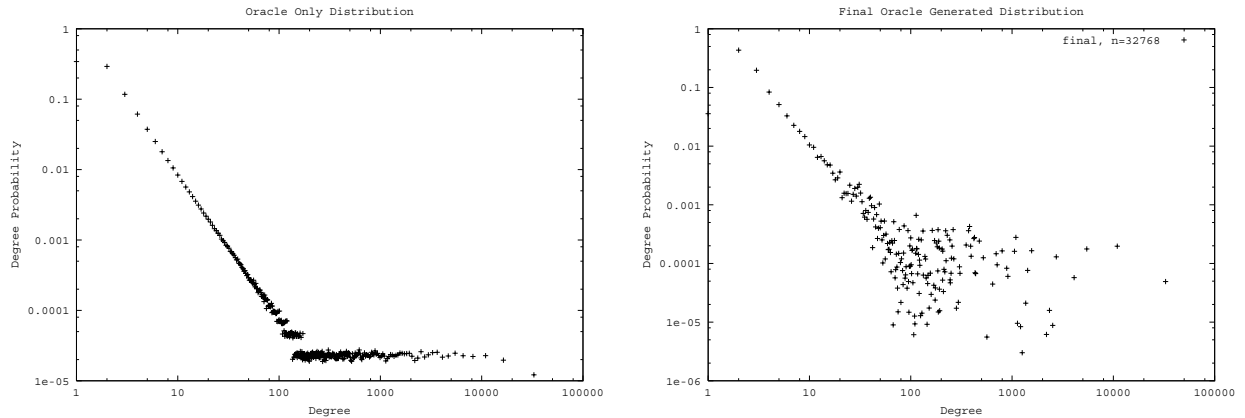


Figure 7: Degree Distributions Using Oracles with  $n=32768$ . (left) Oracle B, (right) Final Oracle Generated Distribution

## 4.5 Truncating Distributions

Our procedure for generating degree distributions runs in time approximately linear in  $n$ .<sup>5</sup> For large  $n$ , say  $2^{19} = 524288$ , this still takes a long time. However, we can take advantage of generating a large distribution once and use it for smaller  $n$ . Essentially, we generate a distribution for a large  $n_0$  and can truncate it to accommodate smaller  $n$ . This truncation is equivalent to generating a degree from the distribution for  $n_0$  and retrying if it is larger than  $n$ .

We have used this to accommodate a large range of file sizes after generating just one large distribution. Curiously, these truncated distributions are some times more efficient than directly generating a distribution. This is probably a side effect of spending more time at each stage of the decoding process when a larger  $n$  is used generating the distribution.

## 5 Conclusions and Future Work

We have presented a simple heuristic for generating good degree distributions. The performance of this heuristic is very good when considering only decoding inefficiency. This heuristic could be further improved by using a better formula for the probability - for large  $d$ , the formula we used is increasingly inaccurate particularly when  $d$  is close to  $n$ .<sup>6</sup> Additionally, other choices of when to invoke the oracle, such as picking the last  $i\%$  of *output symbols* or a random  $i\%$  of all output symbols would probably yield comparable results with lower average degrees.

When considering other measures such as average degree or higher moments, our heuristic does not compare as well. The average degrees are significantly higher than codes such as Tornado codes (often 30% – 50% higher). Additionally, the lack of strong guarantees allows the presence of outliers where a few input symbols take a long time to decode (see Figure 3). For cases such as these, more powerful techniques such as gaussian elimination or those of [1] are probably applicable.

In conclusion, our heuristics work well for anyone looking for codes with a low decoding inefficiency that can be implemented very quickly and there are still many ways to explore to improve this heuristic.

<sup>5</sup>Our distribution generation procedure is randomized and the average degree tends to increase a little with  $n$

<sup>6</sup>For example, when all but one input symbol is decoded, a symbol of degree  $n$  has a 100% chance of triggering the recovery of the last input symbol which is not reflected by the approximation.

## 6 Acknowledgements

This work was done within the context of the ShapeShifter project [7] under the direction of John Byers. John Byers also made the comments on some handcrafted distributions that directly lead to the oracle and its own hints.

## References

- [1] D. Blumstein. Optimization of tornado encoding and decoding for small file sizes. Senior Work for Distinction Report, Boston University, May 2000.
- [2] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98*, Vancouver, September 1998.
- [3] M. Luby, M. Mitzenmacher, and A. Shokrollahi. Analysis of random processes via and-or tree evaluation. In *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998.
- [4] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proceedings of the 29<sup>th</sup> ACM Symposium on Theory of Computing*, April 1997.
- [5] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 38:335–348, 1989.
- [6] S. Rost, J. Byers, and A. Bestavros. The cyclone server architecture: Streamlining delivery of popular content. In *Proceedings of the 6th International Web Caching and Content Delivery Workshop (WCW)*, Boston, MA, June 2001.
- [7] The Shapeshifter project. <http://www.cs.bu.edu/groups/shapeshifter>.
- [8] Swarmcast. <http://www.swarmcast.com>.