FOUNTAIN CODES

David J.C. MacKay

Cavendish Laboratory, University of Cambridge

Keywords

Sparse-graph code, erasure channel, message-passing, LT code, raptor code.

Abstract

Fountain codes are record-breaking sparse-graph codes for channels with erasures – such as the internet, where files are transmitted in multiple small packets, each of which is either received without error or not received.

Standard file-transfer protocols simply chop a file up into K packet-sized pieces, then repeatedly transmit each packet until it is successfully received. A back-channel is required for the transmitter to find out which packets need retransmitting. In contrast, fountain codes make packets that are random functions of the whole file. The transmitter sprays packets at the receiver without any knowledge of which packets are received. Once the receiver has received any N packets, where N is just slightly greater than the original file-size K, he can recover the whole file.

In this paper I review random linear fountain codes, LT codes, and raptor codes. The computational costs of the best fountain codes are astonishingly small, scaling linearly with the file size.

1 Erasure channels

Channels with erasures are of great importance. For example, files sent over the internet are chopped into packets, and each packet is either received without error or not received. Noisy channels to which good error-correcting codes have been applied also behave like erasure channels: much of the time, the error-correcting code performs perfectly; occasionally, the decoder fails, and reports that it has failed, so the receiver knows the whole packet has been lost. A simple channel model describing this situation is a q-ary erasure channel (figure 1), which has (for all inputs in the input alphabet $\{0, 1, 2, \ldots, q-1\}$) a probability $1-f$ of transmitting the input without error, and probability f of delivering the output '?'.

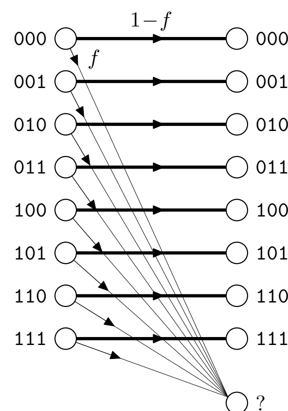

Figure 1. An erasure channel – the 8-ary erasure channel. The eight possible inputs $\{0, 1, 2, \ldots, 7\}$ are here shown by the binary packets $\{000, 001, 010, \ldots, 111\}$.

The alphabet size q is 2^l, where l is the number of bits in a packet.

Common methods for communicating over such channels employ a feedback channel from receiver to sender that is used to control the retransmission of erased packets. For example, the receiver might send back messages that identify the *missing* packets, which are then retransmitted. Alternatively, the receiver might send back messages that acknowledge each *received* packet; the sender keeps track of which packets have been acknowledged and retransmits the others until all packets have been acknowledged.

These simple retransmission protocols have the advantage that they will work regardless of the erasure probability f, but purists who have learned their Shannon theory will feel that these protocols are wasteful. If the erasure probability f is large, the number of feedback messages sent by the first protocol will be large. Under the second protocol, it's likely that the receiver will end up receiving multiple redundant copies of some packets, and heavy use is made of the feedback channel. According to Shannon, there is no need for the feedback channel: the capacity of the forward channel is $(1-f)l$ bits, whether or not we have feedback. Reliable communication should be possible at this rate, with the help of an appropriate forward error-correcting code.

The wastefulness of the simple retransmission

© 1998 IEE, WODES98 – Cagliari, Italy
Proc. of the Fourth Workshop on Discrete Event Systems

protocols is especially evident in the case of a broadcast channel with erasures – channels where one sender broadcasts to many receivers, and each receiver receives a random fraction $(1-f)$ of the packets. If every packet that is missed by one or more receivers has to be retransmitted, those retransmissions will be terribly redundant. Every receiver will have already received most of the retransmitted packets.

So, we would like to make erasure-correcting codes that require no feedback or almost no feedback. The classic block codes for erasure correction are called Reed–Solomon codes [1, 3]. An (N, K) Reed–Solomon code (over an alphabet of size $q = 2^l$) has the ideal property that if any K of the N transmitted symbols are received then the original K source symbols can be recovered. [Reed–Solomon codes exist for $N < q$.] But Reed–Solomon codes have the disadvantage that they are practical only for small K, N, and q: standard implementations of encoding and decoding have a cost of order $K(N–K) \log_2 N$ packet operations. Furthermore, with a Reed–Solomon code, as with any block code, one must estimate the erasure probability f and choose the code rate $R = K/N$ *before* transmission. If we are unlucky and f is larger than expected and the receiver receives fewer than K symbols, what are we to do? We'd like a simple way to extend the code on the fly to create a lower-rate (N', K) code. For Reed–Solomon codes, no such on-the-fly method exists.

There is a better way, pioneered by Michael Luby (2002).

2 Fountain codes

The encoder of a fountain code is a metaphorical fountain that produces an endless supply of water drops (encoded packets); let's say the original source file has a size of Kl bits, and each drop contains l encoded bits. Now, anyone who wishes to receive the encoded file holds a bucket under the fountain and collects drops until the number of drops in the bucket is a little larger than K. They can then recover the original file.

Fountain codes are *rateless* in the sense that the number of encoded packets that can be generated from the source message is potentially limitless; and the number of encoded packets generated can be determined on the fly. Fountain codes are *universal* because they are simultaneously near-optimal for every erasure channel. Regardless of the statistics of the erasure events on the channel, we can send as many encoded packets as are needed in order for the decoder to recover the source data. The source data can be decoded from any set of K' encoded packets, for K' slightly larger than K. Fountain codes can also have fantastically small encoding and decoding

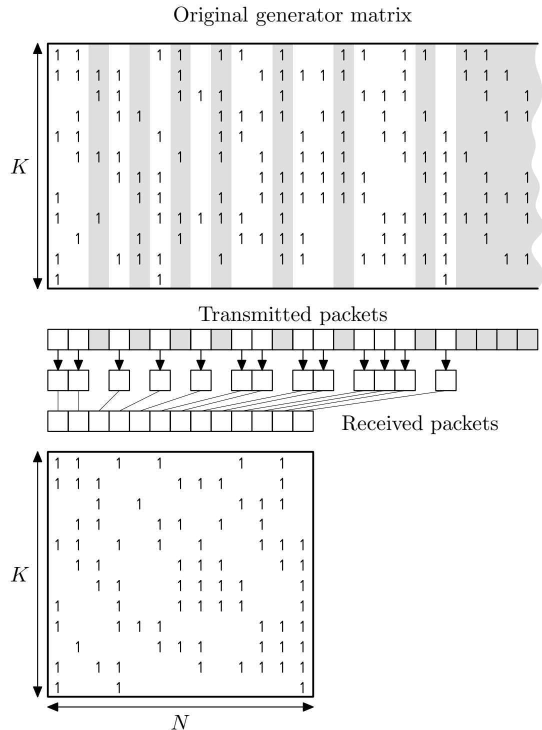

Figure 2. The generator matrix of a random linear code (top). When the packets are transmitted, some are not received, shown by the grey shading of the packets and the corresponding columns in the matrix. We can realign the columns to define the generator matrix, from the point of view of the receiver (bottom).

complexities.

To start with, we'll study the simplest fountain codes, which are random linear codes.

3 The random linear fountain

Consider the following encoder for a file of size K packets $s_1 s_2 \ldots s_K$. A 'packet' here is the elementary unit that is either transmitted intact or erased by the erasure channel. We'll assume that a packet is composed of a whole number of bits.

At each clock cycle, labelled by n, the encoder generates K random bits $\{G_{kn}\}$, and the transmitted packet t_n is set to the *bitwise sum, modulo 2*, of the source packets for which G_{nk} is 1.

$$t_n = \sum_{k=1}^{K} s_k G_{kn}. \quad (1)$$

This sum can be done by successively exclusive-oring the packets together. You can think of each set of K random bits as defining a new column in an ever growing binary generator matrix, as shown at the top of figure 2.

Now, the channel erases a bunch of the packets; a receiver, holding out his bucket, collects N packets. What is the chance that the receiver will be able

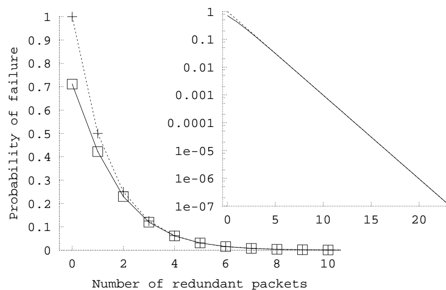

Figure 3. Performance of the random linear fountain. The solid line shows the probability that complete decoding is not possible as a function of the number of excess packets, E. The thin dashed line shows the upper bound, 2^{-E}, on the probability of error.

to recover the entire source file without error? Let's assume that he knows the fragment of the generator matrix $\mathbf{G}$ associated with his packets – for example, maybe $\mathbf{G}$ was generated by a deterministic random-number generator, and the receiver has an identical generator that is synchronized to the encoder's. Alternatively, the sender could pick a random key, κ_n, given which the K bits $\{G_{kn}\}_{k=1}^{K}$ are determined by a pseudo-random process, and send that key in the header of the packet. As long as the packet size l is much bigger than the key size (which need only be 32 bits or so), this key introduces only a small overhead cost. In some applications, every packet will already have a header for other purposes, which the fountain code can use as its key. For brevity, let's call the K–by–N matrix fragment '$\mathbf{G}$' from now on.

Now, as we were saying, what is the chance that the receiver will be able to recover the entire source file without error?

If $N < K$, the receiver hasn't got enough information to recover the file. If $N = K$, it's conceivable that he can recover the file. If the K–by–K matrix $\mathbf{G}$ is invertible (modulo 2), the receiver can compute the inverse $\mathbf{G}^{-1}$ by Gaussian elimination, and recover

$$s_k = \sum_{n=1}^{N} t_n G_{nk}^{-1}. \tag{2}$$

So, what's the probability that a random K–by–K binary matrix is invertible? It's the product of K probabilities, each of them the probability that a new column of $\mathbf{G}$ is linearly independent of the preceding columns. The first factor, is $\left(1 - 2^{-K}\right)$, the probability that the first column of $\mathbf{G}$ is not the all-zero column. The second factor is $\left(1 - 2^{-(K-1)}\right)$, the probability that the second column of $\mathbf{G}$ is equal neither to the all-zero column nor to the first column of $\mathbf{G}$, whatever non-zero column it was. Iterating, the probability of invertibility is

$$\left(1 - 2^{-K}\right)\left(1 - 2^{-(K-1)}\right) \times \cdots \times \left(1 - \frac{1}{8}\right)\left(1 - \frac{1}{4}\right)\left(1 - \frac{1}{2}\right),$$

which is 0.289, for any K larger than 10. That's not great (we would have preferred 0.999!) but it's promisingly close to 1.

What if N is slightly greater than K? Let $N = K + E$, where E is the small number of excess packets. Our question now is, what is the probability that the random K–by–N binary matrix $\mathbf{G}$ *contains* an invertible K–by–K matrix? Let's call this probability $1 - \delta$, so that δ is the probability that the receiver will not be able to decode the file when E excess packets have been received. This failure probability δ is plotted as a function of E for the case $K = 100$ in figure 3 (it looks identical for all $K > 10$). For any K, the probability of failure is bounded above by

$$\delta(E) \leq 2^{-E}. \tag{3}$$

This bound is shown by the thin dotted line in figure 3.

In summary, the number of packets required to have probability $1 - \delta$ of success is $\simeq K + \log_2 1/\delta$. The expected encoding cost per packet is $K/2$ packet operations, since on average half of the packets must be added up. (A packet operation is the exclusive-or of two packets of size l bits.) The expected decoding cost is the sum of the cost of the matrix inversion, which is about K^3 binary operations, and the cost of applying the inverse to the received packets, which is about $K^2/2$ packet operations.

While a random code is not in the technical sense a 'perfect' code for the erasure channel (it has only a chance of 0.289 of recovering the file when K packets have arrived), it is almost perfect. An excess of E packets increases the probability of success to at least $(1 - \delta)$, where $\delta = 2^{-E}$. Thus, as the file size K increases, random linear fountain codes can get arbitrarily close to the Shannon limit. The only bad news is that their encoding and decoding costs are quadratic and cubic in the number of packets encoded. This scaling is not important if K is small (less than one thousand, say); but we'd prefer a solution with lower computational cost.

4 Intermission

Before we study better fountain codes, it will help to solve the following exercises. Imagine that we throw balls independently at random into K bins, where K is a large number such as 1000 or 10 000.

1. After $N = K$ balls have been thrown, what fraction of the bins do you expect have no balls in them?

2. If we throw three times as many balls as there are bins, is it likely that any bins will be empty?

Roughly how many balls must be thrown for it to be likely that every bin has a ball?

3. Show that in order for the probability that all K bins have at least one ball to be $1-\delta$, we require $N \simeq K \log_e(K/\delta)$ balls.

Rough calculations like these are often best solved by finding expectations instead of probabilities. Instead of finding the probability distribution of the number of empty bins, we find the expected number of empty bins. This is easier because means add, even where random variables are correlated.

The probability that one particular bin is empty after N balls have been thrown is

$$\left(1-\frac{1}{K}\right)^N \simeq e^{-N/K}. \tag{4}$$

So when $N = K$, the probability that one particular bin is empty is roughly $1/e$, and the fraction of empty bins must be roughly $1/e$ too. If we throw a total of $3K$ balls, the empty fraction drops to $1/e^3$, about 5%. We have to throw a *lot* of balls to make sure all the bins have a ball! For general N, the expected number of empty bins is

$$Ke^{-N/K}. \tag{5}$$

This expected number is a small number δ (which roughly implies that the probability that all bins have a ball is $(1-\delta)$) only if

$$N > K \log_e \frac{K}{\delta}. \tag{6}$$

5 The LT code

The LT code retains the good performance of the random linear fountain code, while drastically reducing the encoding and decoding complexities. You can think of the LT code as a sparse random linear fountain code, with a super-cheap approximate decoding algorithm.

5.1 Encoder

Each encoded packet t_n is produced from the source file $s_1 s_2 s_3 \ldots s_K$ as follows:

1. Randomly choose the degree d_n of the packet from a degree distribution $\rho(d)$; the appropriate choice of ρ depends on the source file size K, as we'll discuss later.

2. Choose, uniformly at random, d_n distinct input packets, and set t_n equal to the bitwise sum, modulo 2, of those d_n packets.

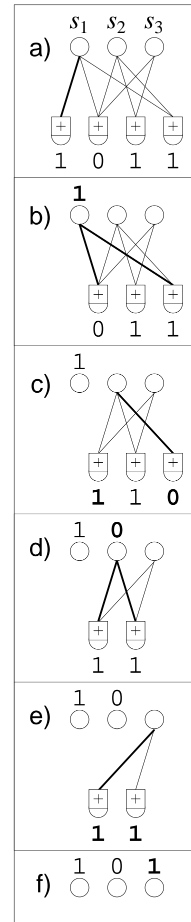

Figure 4. Example decoding for a fountain code with $K = 3$ source bits and $N = 4$ encoded bits. From [5].

This encoding operation defines a graph connecting encoded packets to source packets. If the mean degree $\bar{d}$ is significantly smaller than K then the graph is sparse. We can think of the resulting code as an irregular low-density generator-matrix code.

5.2 Decoder

Decoding a sparse-graph code is especially easy in the case of an erasure channel. The decoder's task is to recover $\mathbf{s}$ from $\mathbf{t} = \mathbf{sG}$, where $\mathbf{G}$ is the matrix associated with the graph. (Just as in the random linear fountain code, we assume the decoder somehow knows the pseudorandom matrix $\mathbf{G}$.)

The simple way to attempt to solve this problem is by message-passing. We can think of the decoding algorithm as the sum–product algorithm [5, Chs. 16, 26, and 47] if we wish, but all messages are either *completely uncertain* or *completely certain.* Uncertain messages assert that a message packet s_k could have any value, with equal probability; certain messages assert that s_k has a particular value, with probability one.

This simplicity of the messages allows a simple description of the decoding process. We'll call the

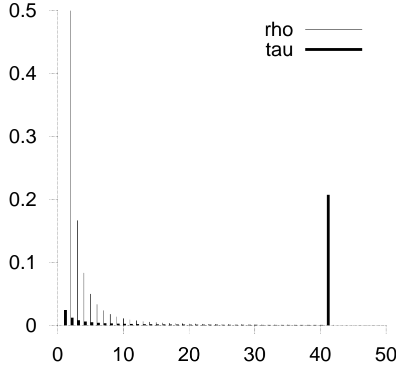

Figure 5. The distributions $\rho(d)$ and $\tau(d)$ for the case $K = 10\,000$, $c = 0.2$, $\delta = 0.05$, which gives $S = 244$, $K/S = 41$, and $Z \simeq 1.3$. The distribution τ is largest at $d = 1$ and $d = K/S$. From [5].

encoded packets $\{t_n\}$ check nodes.

1. Find a check node t_n that is connected to *only one* source packet s_k. (If there is no such check node, this decoding algorithm halts at this point, and fails to recover all the source packets.)
   (a) Set $s_k = t_n$.
   (b) Add s_k to all checks $t_{n'}$ that are connected to s_k:

$$t_{n'} := t_{n'} + s_k \quad \text{for all } n' \text{ such that } G_{n'k} = 1.$$

   (c) Remove all the edges connected to the source packet s_k.
2. Repeat (1) until all $\{s_k\}$ are determined.

This decoding process is illustrated in figure 4 for a toy case where each packet is just one bit. There are three source packets (shown by the upper circles) and four received packets (shown by the lower check symbols), which have the values $t_1t_2t_3t_4 = 1011$ at the start of the algorithm.

At the first iteration, the only check node that is connected to a sole source bit is the first check node (panel a). We set that source bit s_1 accordingly (panel b), discard the check node, then add the value of s_1 (1) to the checks to which it is connected (panel c), disconnecting s_1 from the graph. At the start of the second iteration (panel c), the fourth check node is connected to a sole source bit, s_2. We set s_2 to t_4 (0, in panel d), and add s_2 to the two checks it is connected to (panel e). Finally, we find that two check nodes are both connected to s_3, and they agree about the value of s_3 (as we would hope!), which is restored in panel f.

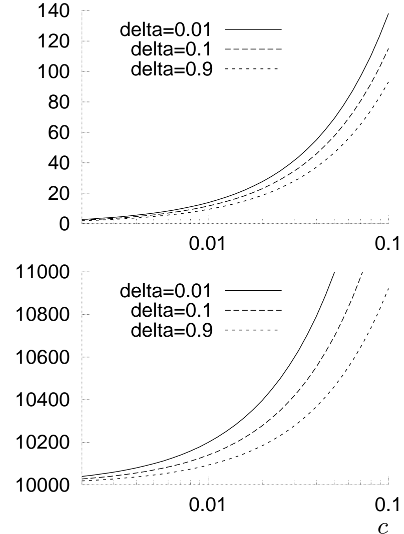

Figure 6. The number of degree-one checks S (upper figure) and the quantity K' (lower figure) as a function of the two parameters c and δ, for $K = 10\,000$. Luby's main theorem proves that there exists a value of c such that, given K' received packets, the decoding algorithm will recover the K source packets with probability $1 - \delta$. From [5].

5.3 Designing the degree distribution

The probability distribution $\rho(d)$ of the degree is a critical part of the design: occasional encoded packets must have high degree (*i.e.*, d similar to K) in order to ensure that there are not some source packets that are connected to no-one. Many packets must have low degree, so that the decoding process can get started, and keep going, and so that the total number of addition operations involved in the encoding and decoding is kept small. For a given degree distribution $\rho(d)$, the statistics of the decoding process can be predicted by an appropriate version of density evolution, a technique first developed for low-density parity-check codes [5, p. 566].

Before giving Luby's choice for $\rho(d)$, let's think about the rough properties that a satisfactory $\rho(d)$ must have. The encoding and decoding complexity are both going to scale linearly with the number of edges in the graph, so the crucial quantity is the average degree of the packets. How small can this be? The balls-in-bins exercise helps here: think of the *edges* that we create as the balls and the source packets as the bins. In order for decoding to be successful, every source packet must surely have at least one edge in it. The encoder throws edges into source packets at random, so the number of edges must be at least of order $K \log_e K$. If the number of packets received is close to Shannon's optimal K, and decoding is possible, the average degree of each packet must be at least $\log_e K$, and the encoding and decoding complexity of an LT code will definitely be at least $K \log_e K$. Luby showed that this bound on complexity can indeed be achieved by a careful choice

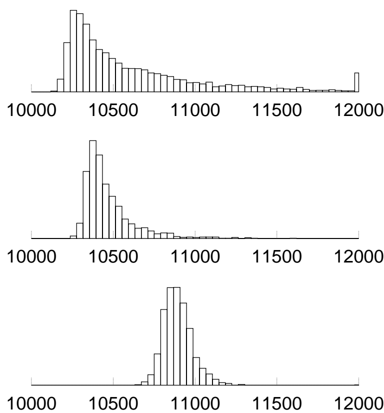

Figure 7. Histograms of the actual number of packets N required in order to recover a file of size $K = 10\,000$ packets. The parameters were as follows:
top histogram: $c = 0.01$, $\delta = 0.5$ ($S = 10$, $K/S = 1010$, and $Z \simeq 1.01$);
middle: $c = 0.03$, $\delta = 0.5$ ($S = 30$, $K/S = 337$, and $Z \simeq 1.03$);
bottom: $c = 0.1$, $\delta = 0.5$ ($S = 99$, $K/S = 101$, and $Z \simeq 1.1$). From [5].).

of degree distribution.

Ideally, to avoid redundancy, we'd like the received graph to have the property that just one check node has degree one at each iteration. At each iteration, when this check node is processed, the degrees in the graph are reduced in such a way that one new degree-one check node appears. *In expectation*, this ideal behaviour is achieved by the *ideal soliton distribution*,

$$\begin{array}{rcll} \rho(1) & = & 1/K & \\ \rho(d) & = & \frac{1}{d(d-1)} & \text{for } d = 2, 3, \ldots, K. \end{array} \quad (7)$$

The expected degree under this distribution is roughly $\log_e K$.

This degree distribution works poorly in practice, because fluctuations around the expected behaviour make it very likely that at some point in the decoding process there will be no degree-one check nodes; and, furthermore, a few source nodes will receive no connections at all. A small modification fixes these problems.

The *robust soliton distribution* has two extra parameters, c and δ; it is designed to ensure that the expected number of degree-one checks is about

$$S \equiv c \log_e(K/\delta)\sqrt{K}, \quad (8)$$

rather than 1, throughout the decoding process. The parameter δ is a bound on the probability that the decoding fails to run to completion after a certain

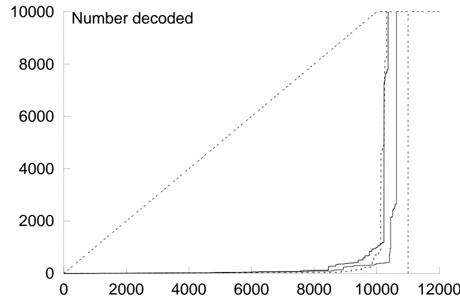

Figure 8. Practical performance of LT codes. Three experimental decodings are shown, all for codes created with the parameters $c = 0.03$, $\delta = 0.5$ ($S = 30$, $K/S = 337$, and $Z \simeq 1.03$) and a file of size $K = 10\,000$. The decoder is run greedily as packets arrive. The vertical axis shows the number of packets decoded as a function of the number of received packets. The right-hand vertical line is at a number of received packets $N = 11\,000$, *i.e.*, an overhead of 10%.

number K' of packets have been received. The parameter c is a constant of order 1, if our aim is to prove Luby's main theorem about LT codes; in practice however it can be viewed as a free parameter, with a value somewhat smaller than 1 giving good results. We define a positive function

$$\tau(d) = \begin{cases} \frac{S}{K}\frac{1}{d} & \text{for } d = 1, 2, \ldots (K/S) - !1 \\ \frac{S}{K}\log(S/\delta) & \text{for } d = K/S \\ 0 & \text{for } d > K/S \end{cases} \quad (9)$$

(see figure 5) then add the ideal soliton distribution ρ to τ and normalize to obtain the robust soliton distribution, μ:

$$\mu(d) = \frac{\rho(d) + \tau(d)}{Z}, \quad (10)$$

where $Z = \sum_d \rho(d) + \tau(d)$. The number of encoded packets required at the receiving end to ensure that the decoding can run to completion, with probability at least $1 - \delta$, is $K' = KZ$.

Luby's analysis [4] explains how the small-d end of τ has the role of ensuring that the decoding process gets started, and the spike in τ at $d = K/S$ is included to ensure that every source packet is likely to be connected to a check at least once. Luby's key result is that (for an appropriate value of the constant c) receiving $K' = K + 2\log_e(S/\delta)S$ checks ensures that all packets can be recovered with probability at least $1 - \delta$. In the illustrative figures I have set the allowable decoder failure probability δ quite large, because the actual failure probability is much smaller than is suggested by Luby's conservative analysis.

In practice, LT codes can be tuned so that a file of original size $K \simeq 10\,000$ packets is recovered with an overhead of about 5%. Figure 7 shows histograms of the actual number of packets required for a couple

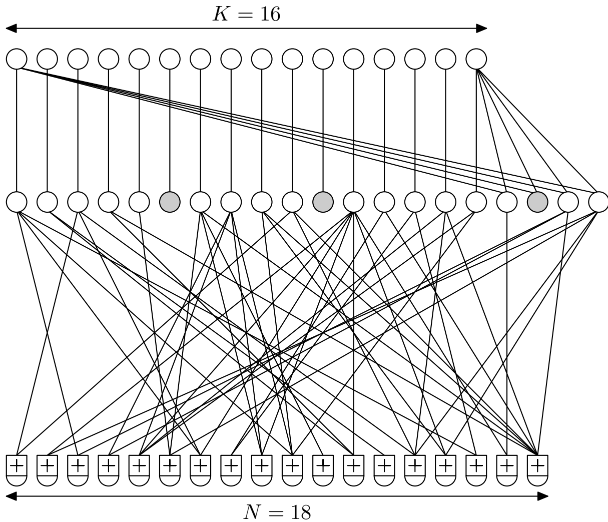

Figure 9. Schematic diagram of a raptor code. In this toy example, $K = 16$ source packets (top row) are encoded by the outer code into $\tilde{K} = 20$ pre-coded packets (centre row). The details of this outer code are not given here. These packets are encoded into $N = 18$ received packets (bottom row) with a weakened LT code. Most of the received packets have degree 2 or 3. The average degree is 3. The weakened LT code fails to connect some of the pre-coded packets to any received packet – these 3 lost packets are highlighted in grey. The LT code recovers the other 17 pre-coded packets, then the outer code is used to deduce the original 16 source packets.

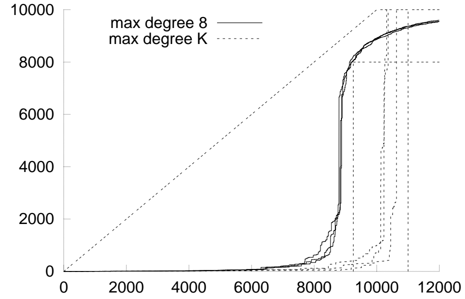

Figure 10. The idea of a weakened LT code. I truncated the LT degree distribution with parameters $c = 0.03$, $\delta = 0.5$, constraining the maximum degree to be 8. The resulting graph has mean degree 3. The decoder is run greedily as packets arrive. As in figure 8, the thick lines show the number of recovered packets as a function of the number of received packets. The thin lines are the curves for the original LT code from figure 8. Just as the original LT code usually recovers $K = 10\,000$ packets within a number of received packets $N = 11\,000$, the weakened LT code recovers 8000 packets within a received number of 9250.

of settings of the parameters, achieving mean overheads smaller than 5% and 10% respectively. Figure 8 shows the time-courses of three decoding runs. It is characteristic of a good LT code that very little decoding is possible until slightly more than K packets have been received, at which point, an avalanche of decoding takes place.

6 Raptor codes

You might think that we couldn't do any better than LT codes: their encoding and decoding costs scale as $K \log_e K$, where K is the file size. But raptor codes [8] achieve linear time encoding and decoding by concatenating a *weakened* LT code with an outer code that patches the gaps in the LT code.

LT codes had decoding and encoding complexity that scaled as $\log_e K$ per packet, because the average degree of the packets in the sparse graph was $\log_e K$. Raptor codes use an LT code with average degree $\bar{d}$ about 3. With this lower average degree, the decoder may work in the sense that it doesn't get stuck, but a fraction of the source packets will not be connected to the graph and so will not be recovered. What fraction? From the balls-in-bins exercise, the expected fraction not recovered is $\tilde{f} \equiv e^{-\bar{d}}$, which for $\bar{d} = 3$ is 5%. Moreover, if K is large, the law of large numbers assures us that the fraction of packets not recovered in any particular realisation will be very close to $\tilde{f}$.

So, here is Shokrollahi's trick: we transmit a K-packet file by first *pre-coding* the file into $\tilde{K} \simeq K/(1-\tilde{f})$ packets with *an excellent outer code* that can correct erasures if the erasure rate is exactly $\tilde{f}$; then we transmit this slightly enlarged file using a weak LT code that, once slightly more than K packets have been received, can recover $(1-\tilde{f})\tilde{K}$ of the pre-coded packets, which is roughly K packets; then we use the outer code to recover the original file (figure 9).

Figure 10 shows the properties of a crudely weakened LT code. Whereas the original LT code usually recovers $K = 10\,000$ packets within a number of received packets $N = 11\,000$, the weakened LT code usually recovers 8000 packets within a received number of 9250. Better performance can be achieved by optimizing the degree distribution.

For our excellent outer code, we require a code that can correct erasures at a known rate of 5% with low decoding complexity. Shokrollahi uses an irregular low-density parity-check code. For further information about irregular low-density parity-check codes, and fast encoding algorithms for them, see [5, pp. 567–572] and [6, 7].

7 Applications

Fountain codes are an excellent solution in a wide variety of situations. Let's mention two.

7.1 Storage

You wish to make a backup of a large file, but you are aware that your magnetic tapes and hard drives are all unreliable: catastrophic failures, in which some stored packets are permanently lost within one device, occur at a rate of something like 10^{-3} per day. How should you store your file?

A fountain code can be used to spray encoded packets all over the place, on every storage device available. To recover the file, whose size was K packets, one simply needs to find $K' \simeq K$ packets from anywhere. Corrupted packets do not matter; we simply skip over them and find more packets elsewhere.

This method of storage also has advantages in terms of *speed* of file recovery. In a hard drive, it is standard practice to store a file in successive sectors of a hard drive, to allow rapid reading of the file; but if, as occasionally happens, a packet is lost (owing to the reading head being off track for a moment, giving a burst of errors that cannot be corrected by the packet's error-correcting code), a whole revolution of the drive must be performed to bring back the packet to the head for a second read. The time taken for one revolution produces an undesirable delay in the file system. If files were instead stored using the fountain principle, with the digital drops stored in one or more consecutive sectors on the drive, then one would never need to endure the delay of re-reading a packet; packet loss would become less important, and the hard drive could consequently be operated faster, with higher noise level, and with fewer resources devoted to noisy-channel coding.

7.2 Broadcast

Imagine that ten thousand subscribers in an area wish to receive a digital movie from a broadcaster. The broadcaster can send the movie in packets over a broadcast network – for example, by a wide-bandwidth phone line, or by satellite. Imagine that $f = 0.1\%$ of the packets are lost at each house. In a standard approach in which the file is transmitted as a plain sequence of packets with no encoding, each house would have to notify the broadcaster of the fK missing packets, and request that they be retransmitted. And with ten thousand subscribers all requesting such retransmissions, there would be a retransmission request for almost every packet. Thus the broadcaster would have to repeat the entire broadcast twice in order to ensure that most subscribers have received the whole movie, and most users would have to wait roughly twice as long as the ideal time before the download was complete.

If the broadcaster uses a fountain code to encode the movie, each subscriber can recover the movie from *any* $K' \simeq K$ packets. So the broadcast needs to last for only, say, $1.1K$ packets, and every house is very likely to have successfully recovered the whole file.

Another application is broadcasting data to cars. Imagine that we want to send updates to in-car navigation databases by satellite. There are hundreds of thousands of vehicles, and they can only receive data when they are out on the open road; there are no feedback channels. A standard method for sending the data is to put it in a *carousel*, broadcasting the packets in a fixed periodic sequence. 'Yes, a car may go through a tunnel, and miss out on a few hundred packets, but it will be able to collect those missed packets an hour later when the carousel has gone through a full revolution (we hope); or maybe the following day...'

If instead the satellite uses a fountain code, each car needs to receive only an amount of data equal to the original file size (plus 5%).

References

[1] E. R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, 1968.

[2] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98, September 2–4, 1998*, 1998.

[3] S. Lin and D. J. Costello, Jr. *Error Control Coding: Fundamentals and Applications.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[4] M. Luby. LT codes. In *Proceedings of The 43rd Annual IEEE Symposium on Foundations of Computer Science, November 16–19 2002*, pages 271–282, 2002.

[5] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003. Available from `www.inference.phy.cam.ac.uk/mackay/itila/`.

[6] T. Richardson, M. A. Shokrollahi, and R. Urbanke. Design of capacity-approaching irregular low-density parity check codes. *IEEE Trans. on Info. Theory*, 47(2):619–637, 2001.

[7] T. Richardson and R. Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Trans. on Info. Theory*, 47(2):638–656, 2001.

[8] A. Shokrollahi. Raptor codes. Technical report, Laboratoire d'algorithmique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2003. Available from `algo.epfl.ch/`.