HELSINKI UNIVERSITY OF TECHNOLOGY
Networking Laboratory
S-38.138 Networking Technology, Special Assignment

Optimal Degree Distributions for LT Codes in Small Cases

Author: Tuomas Tirronen
55701P
tuomas.tirronen@hut.fi
Supervisor: Esa Hyytiä
Submitted: 25.11.2005

Contents

1 Introduction

Distribution of large files is nowadays a popular application on the Internet. Situations with one server distributing data to multiple receivers is particularly common, where the content can be operating system updates, movies or other media files, the list of possible applications is endless. Also different kinds of peer-to-peer applications, such as data replication, are becoming more popular. These concepts can be easily extended to any kind of network from small scale wireless ad-hoc networks to global communications networks.

This kind of applications can be very demanding for the network itself and to servers, i.e., the sources, distributing the content. A favorable characteristic for the distribution process is the reliability of the file transferring process to minimize the number of retransmissions and probability of errors. In 1998 John W. Byers et al. presented an interesting new idea called a digital fountain approach for reliable distribution of data [BLMR98].

The purpose of this work is to present key ideas behind digital fountain codes and their properties. The main results developed in this work are optimal probability distributions, called degree distributions, used in fountain codes.

First general discussion about coding methods and some channel models is discussed in Section 2. Section 3 presents digital fountain principle and the respective codes. After this analytical results are presented in Section 4 and simulation results in Section 5. All analytical and simulation results were obtained using Mathematica 5.0 [Wol].

2 Background for digital fountain codes

2.1 General

In general, communication over any real transmission channel is noisy, i.e., channel incurs some errors to transmitted information. Important goal naturally is to reduce this noise as much as possible. This can be achieved by improving the channel characteristics physically or by means of information and coding theory. The latter way includes adding an encoder before the channel and a decoder after it. This way the transmitted message undergoes three distinct phases during transmission: encoding, adding of possible noise and decoding. The usage of digital fountain codes fall into this latter category.

Material represented in this section can easily be found in many sources covering principles of information theory or telecommunications. The main reference that has been used in this section is [Mac03].

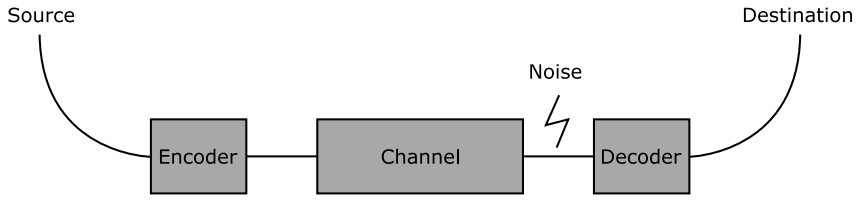

Figure 1: Channel model with encoder and decoder

2.2 Forward error correction

Forward error correction (FEC) is a way to increase reliability of a communication channel by adding redundancy to transmitted messages. In FEC, possible errors introduced by the transmission channel are corrected by the code itself without need for retransmitting the message or parts of it. Another scheme used for correcting errors is automatic repeat request (ARQ), where incorrectly transmitted parts of a message are sent again over the channel when explicitly asked for by the recipient.

ARQ scheme could cause problems situations where packet losses are frequent, i.e., when network is congested or satellite communications in bad weather. This mean that in some situations FEC is the more alluring choice, providing that efficient FEC solutions (coding methods) are available. FEC does not use any retransmissions so its properties suit particularly well to transmission of large chunks of data to many participants at the same time (it should be noted that it is possible to combine FEC and ARQ methods, i.e., the can be used to complement each other). ARQ would raise some problems when used in this

kind of application, e.g., distributing servers could choke if thousands of receivers tried to ask for a specific missing part or different parts of a message at the same time.

FEC has been traditionally used in specialized hardware components in transmission links and software based FEC systems are not widely used yet in computer communications. One reason for this is that encoding and decoding processes could add a significant delay component to the overall communication process. Nowadays even desktop PC's have plenty of computing power so the significance of encoding delay probably is not relevant anymore. Of course, the coding method used has to be efficient enough to provide a good alternative to the traditional packet based transmission method.

Traditional TCP on the Internet uses an ARQ style packet acknowledgement scheme. Some discussion on how to utilize FEC on the Internet in one-to-many data transfer over IP multicast can be found in [RFC3452]. Other request for comments covering the use of FEC on the Internet are [RFC3453] and [RFC3695].

2.3 Symmetric channel and erasure channel

Perhaps the most simple instance of a channel is a binary symmetric channel. The set of possible symbols which can be fed into the channel is called an input alphabet and denoted by A_i and correspondingly the set of possible output symbols is output alphabet, A_o. In a binary symmetric channel both sets are the same and include only two symbols: $A_i = A_o = \{0, 1\}$. When the probability of an error is p_f then a transferred symbol is the same with probability of $1 - p_f$ and changes with probability of p_f.

A binary erasure channel is a channel where input alphabet is $A_i = \{0, 1\}$ and output alphabet is $A_o = \{0, 1, ?\}$. The symbol ? in output alphabet refers to a situation when possible error has occurred and the real symbols is either 0 or 1. This means that transferred symbol is correct with probability $1 - p_f$ and output is symbol ? with probability p_f. Hence an erasure channel does not output possible erroneous bits, but instead represents them with a symbol which tells the receivers that something is wrong.

Both of these concepts can be extended to an arbitrary size of input alphabet, so that $q = |A_i|$. The channel can the be called a q-ary binary or erasure channel.

Important thing to be noticed here is that in a regular a IP-based network links can be thought as erasure channels; a packet is discarded by the network nodes if it has some errors, for example by comparing the checksum in packet header to a computed checksum. Hence the channel is q-ary erasure channel and the size of this channel is $q = 2^l$, where l is the number of bits in the packet. Thus, if the probability for error is again p_f, then the probability for receiving a packet without error is $1 - p_f$ and packet is discarded with probability p_f.

3 Digital fountain codes

3.1 Digital fountain principle

Let us consider a situation where a company has a server which is supposed to distribute a large file to hundreds of thousands of clients. Downloading a movie from a electronic movie store is one example application. Point to point connections between server and every downloader would definitely cause serious troubles for the server unless it has vast amounts of bandwidth and the processing power to handle all the connections at the same time. Multicasting is a more reasonable solution to this situation. Problem with this approach is that multicasting itself is not widely supported by ISPs or the networking equipment and thus may not be applicable to this problem.

The digital fountain principle introduced in [BLMR98] could be a way to solve the described problem. The name comes from an analogy to a fountain. We have an infinite supply of packets which we "spray" into network (a fountain sprays water drops into air), and to collect the original data we just need to collect a given amount of these packets (we hold a bucket under the fountain to get enough water for our purposes). If in above-mentioned problem we have a movie file consisting of N blocks size of ℓ then in ideal case we need just to collect N packets to decode the original file.

The digital fountain can be approximated with erasure codes. The classic Reed-Solomon codes are not practical enough for this but in [BLMR98] Tornado codes are presented which are suitable for approximating the ideal digital fountain. After this LT codes [Lub02] and Raptor codes [Sho] have been invented which also approximate the digital fountain. Raptor codes are the most efficient type of fountain codes to this day. In this work the emphasis will be on LT codes and their properties are discussed in the next section.

3.2 LT codes

LT codes were presented by Michael Luby in 2002 [Lub02]. The number of encoded symbols which can be produced is virtually limitless; these kind of codes are called rateless codes. The encoding and decoding processes are rather simple and only exclusive-or (XOR, $\oplus$) and copying operations are needed. This simplicity can be deceptive as work has to be done to make these codes efficient enough to be practical.

Encoding symbols are here called packets, as they are the units which are transferred from source to receiver. Input symbols are non-overlapping continuous chunks of a file and they are called blocks. The procedures defined next follow the presentation in [Mac03, chapter 50].

3.2.1 Encoding

A digital fountain encoder needs to produce an infinite supply of packets built from blocks of the original file. The encoding procedure goes as follows:

1. Choose a degree d from a degree distribution $\rho(d)$.
2. Choose uniformly at random d blocks from the source file consisting of blocks $s_1 \ldots s_k$. Denote these as $\hat{s}_1 \ldots \hat{s}_d$.
3. The encoded packet t_n is produced by successively using bitwise XOR to each of blocks chosen in the previous step, $t_n = \hat{s}_1 \oplus \hat{s}_2 \oplus \ldots \oplus \hat{s}_d$.

The procedure is illustrated in Figure 2. Encoding generates a bipartite graph where each sent packet has as neighbors the blocks which are combined using XOR. Figure 3 shows this.

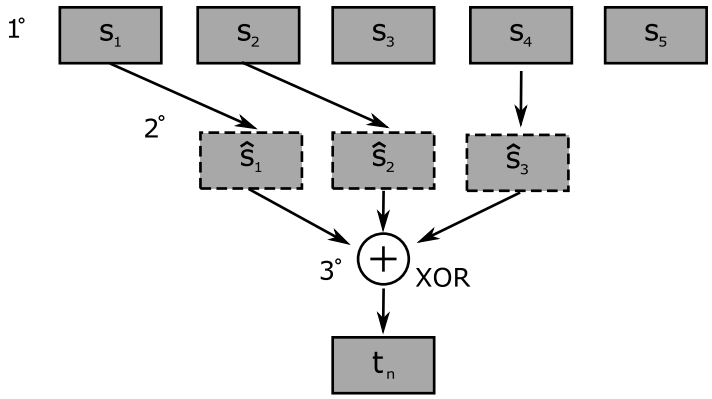

Figure 2: Encoding illustrated, d blocks from source file are chosen and combined by computing a bitwise XOR

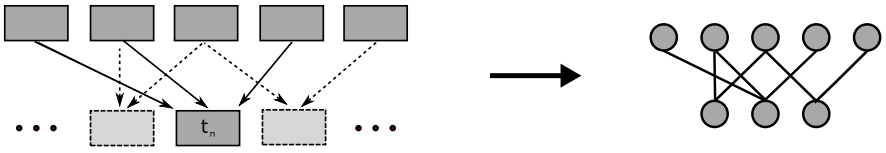

Figure 3: Encoding defines a bipartite graph

3.2.2 Decoding

The decoding procedure needs in addition to packets the information about which blocks are included in which packets. When this information is available the decoding procedure is:

1. Find a output node of degree one in the graph, i.e., find a node t_n which has only s_k as a neighbor. If no such node is available receive more packets until such a node is found.
2. Now recovered t_n is exactly the copy of its only neigbour; set $s_k = t_n$
3. For every other output node $t_i, i \neq n$ connected to the input node s_k set $t_i = s_k \oplus t_i$.
4. Remove all edges between s_k and output nodes.
5. If all s_i have been recovered, we have recovered the original file, otherwise return to step 1.

The decoding procedure is illustrated in Figure 4. The procedure defined here is referred to as simple decoding.

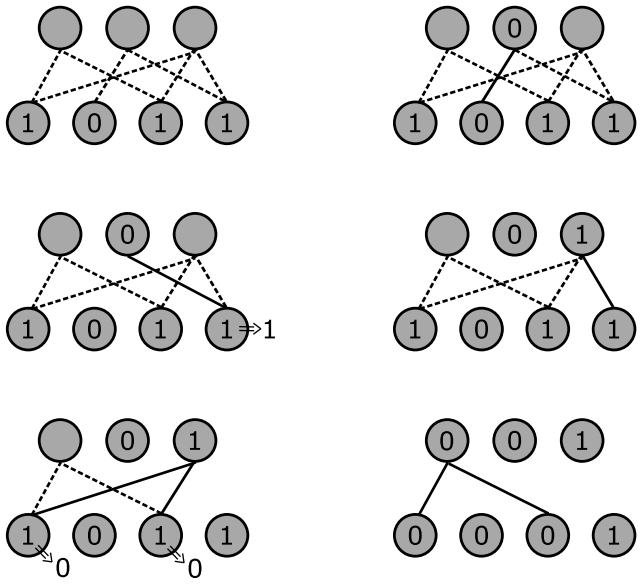

Figure 4: Decoding illustrated, process evolves from left to right and from top to down. On each step solid line represents the usable information which is used to resolve values of input nodes and to change the values of output nodes.

3.2.3 Approach using linear system of equations

The whole encoding and decoding process can also be thought as solving a linear system of equations in binary base, that is possible symbols in equations are 0 and 1. One sent packet corresponds to one linear equation, and multiple sent packets forms a linear system of equations. This system can be solved by modulo-2 matrix inversion. The disadvantage of this approach is that the inversion becomes computationally very expensive when the number of the equations grows.

Example 1 *Let us consider the situation in Figure 4. Let the output nodes correspond to vector* $\mathbf{y}$ *and the input nodes correspond to vector* $\mathbf{x}$*. Now we know that* $\mathbf{y} = (1 \quad 0 \quad 1 \quad 1)$*. Then we have the following linear system of equations:*

$$\begin{cases} x_1 + x_3 = 1 \\ x_2 = 0 \\ x_1 + x_3 = 1 \\ x_2 + x_3 = 1 \end{cases}$$

In this simple case it is easy to see that the solution is $\mathbf{x} = (0 \quad 0 \quad 1)$*. The same result can be calculated by putting above equation into matrix form and inverting the multiplicating matrix.*

This approach will be called full decoding. Name comes from the fact that here we use all the information we have from the encoding packets collected so far to calculate the original data, i.e., we solve the system of multiple equations (packets) in one step. This is in contrast to iterative decoding where only one collected packet is processed at each step.

3.3 Degree distribution

There are some design problems which should be taken into account when designing a digital fountain encoder described before. The main problem discussed is the design of a proper degree distribution. This distribution is crucial for a successful operation of a digital fountain. Many of the decoded packets should have a high degree, so that all of the input blocks are connected to some encoded packet. On the other hand, degree one packet is needed to start the decoding process and low-degree packets are needed to keep the started process going on.

The distributions presented by Luby in [Lub02] are described next. Later, in Section 5, they are used to generate simulation results. The last two of these are of more interest, only

a few arguments are given to motivate these. For more discussion about the properties of these two distributions see [Lub02] or [Mac03].

3.3.1 All-at-once distribution

Distribution which Luby calls all-at-once distribution means just that all packet degrees are one, i.e., $\rho(1) = 1$.

Definition 1 *In **all-at-once distribution** all packet degrees are* $\rho(1) = 1$.

This strategy does not produce good results, which is fairly intuitive. The expected number of packets needed in theory for decoding is fairly easy to calculate. If the receiver has already received k distinct packets, then the probability that the next packet is a new one is $\frac{N-k}{N}$. Now if we the denote by X_k the random variable for the amount of received packets until block k is received we have

$$X = X_0 + X_1 + \cdots + X_{N-1},$$

where X is the random variable for total number of packets. X_k is geometrically distributed with probability $p = \frac{N-k}{N}$, so the exceptation is $E[X_k] = \frac{1}{p} = \frac{N}{N-k}$. Consequently, the expectation of total number of packets is:

$$\begin{aligned} E[X] &= E[X_0 + X_1 + \cdots + X_{N-1}] = E[X_0] + E[X_1] + \cdots + E[X_{N-1}] \\ &= \sum_{k=0}^{N-1} \frac{N}{N-k} = N \sum_{k=0}^{N-1} \frac{1}{N-k} = N \sum_{k=1}^{N} \frac{1}{k} \approx O(N \log N), \end{aligned} \tag{1}$$

where the approximation $\sum_{k=1}^{N}(1/k) \approx O(\log N)$ is used on the last summation. This approximation can be derived by bounding the summation by integrals:

$$\int_1^{N+1} \frac{1}{x}\,dx < \sum_{k=1}^{N} \frac{1}{k} < 1 + \int_1^{N} \frac{1}{x}\,dx \iff \log(N+1) < \sum_{k=1}^{N} \frac{1}{k} < 1 + \log N \tag{2}$$

This bounding method is illustrated in Figure 5, where left part of figure shows the bounding from upper side and correspondingly right part the bounding from below. Each rectangle represents one term $1/k$ of the sum, k ranging from 1 to N, while the area under the drawn curve is the definite integral of function $1/x$. The left part of the figure shows that the area covered by N rectangles is smaller than the area of the first rectangle added to the area under the curve starting from $x = 1$ and ending when $x = N$. The right part shows that the area under N rectangles is larger than the area under the curve in the same region. This

reasoning leads to Equation 2.

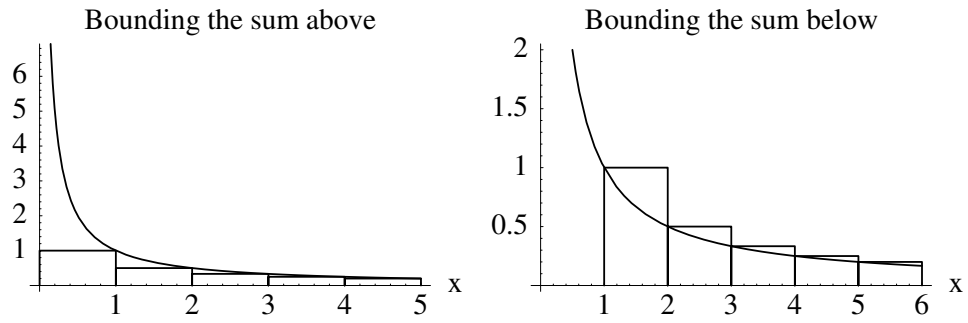

Figure 5: The method of bounding sums. Each rectangle represents one term of the bounded sum and the area under curves is used to limit the sum of terms from above and below.

3.3.2 Ideal soliton distribution

This distribution is based on the idea that on each iteration of the decoding algorithm presented in Section 3.2.2, it is ideal to have just one degree one node on each iteration. This leads to the following distribution:

Definition 2 ***Ideal soliton distribution*** $\rho(i)$ *is:*

$$\rho(i) = \begin{cases} \frac{1}{N} & \text{when } i = 1 \\ \frac{1}{i(i-1)} & \text{for } i = 2, \ldots, N, \end{cases}$$

where N *is the number of blocks in original message.*

This distribution however works poorly in practice, as we will see in Section 5. The reason for failure of the ideal soliton distribution is that it operates as desired only in expectation, i.e., one symbol of degree one is revealed in a step in expectation. Thus the variance in the process may lead to situation where there is no degree one symbols available so more packets are needed for the process to continue, this causes undesired overhead in the number of packets.

3.3.3 Robust soliton distribution

As the ideal soliton distribution works poorly in practice, Luby proposes a robust soliton distribution which is somewhat based on ideal soliton distribution presented above.

Definition 3 *For the **robust soliton distribution**, first define function:*

$$\tau(i) = \begin{cases} \frac{R}{iN} & \text{for } i = 1, \ldots, \frac{N}{R} - 1 \\ \frac{R \log R/\delta}{N} & \text{for } i = \frac{N}{R} \\ 0 & \text{for } i = \frac{N}{R} + 1, \ldots, N, \end{cases}$$

where δ is the failure probability of decoding process after N' encoded packets and $R = c \log(N/\delta)\sqrt{N}$ for some constant $c > 0$. The robust soliton distribution $\mu(i)$ is the normalized value of the sum $\rho(i) + \tau(i)$:

$$\mu(i) = \frac{\rho(i) + \tau(i)}{\sum_{i=1}^{N} \rho(i) + \tau(i)}.$$

It can be shown ([Lub02]) that decoding can be achieved using $N' = N + O(\log^2(N/\delta)\sqrt{N})$ encoding symbols by using this distribution.

This distribution ensures that number of degree one symbols during the decoding process is

$$S = c \log\left(\frac{N'}{\delta}\right)\sqrt{N'},$$

instead of one, as is in case of ideal soliton distribution. This larger value for available degree one packets in iteration causes the distribution to behave better, as variation around only one degree one packet in ideal soliton distribution is the reason for its failure.

Further discussion about the properties of the robust and ideal soliton distributions can be found in [Lub02].

3.3.4 Objectives for optimization of degree distributions

In the next sections two different and mutually exclusive approaches to optimizing the degree distribution for small cases are considered.

Definition 4 *Objective **Min.Avg.** means minimization of the mean number of packets for succesful decoding.*

Definition 5 *Objective **Max.Pr.** means maximization of the probability of succesful decoding with specified amount of packets.*

4 Analytical results using Markov chains

For simple cases, when the size N of the message is just three or four blocks, analytical results were calculated to give some insight about the behavior of the degree distribution when probability for successful decoding is maximized. These cases are modeled using Markov chains. The case $N = 3$ is simple enough to form the necessary state transition matrices explicitly by going trough all necessary steps by hand. However, when $N = 4$ the state space becomes so big that help of a computer algebra system is needed. Mathematica 5.0 [Wol] was used to generate the necessary state transition matrices for these two cases. The results in this section are exact in the sense that no simulation is used, although numerical optimization methods are utilized.

The results calculated here are later in Section 5 compared to simulation results with $N = 3$ and $N = 4$.

4.1 Modeling the fountain coding process as a Markov chain

4.1.1 Markov chain and state space reduction

The whole fountain coding process constitutes a Markov chain. From the receiver's point of view, the set of received packets and possibly decoded blocks denotes a state. State transition probabilities depend on the arrival probabilities of specific packets.

For example let us consider a file consisting of three blocks $\{a, b, c\}$. When the receiver has already received a packet with block a and another one with packets b and c, then the process is in state $\{a, bc\}$, and here bc denotes a packet with blocks b and c combined using bitwise XOR.

The number of possible distinct packets is 2^N (i.e. the number of the subsets of a set with $|N|$ elements). Using this information, total number of possible states in Markov chain described as above is 2^{2^N}. For $N = 3$ this is 256, and for $N = 4$ number of states is 65536. For $N = 4$ the state space is rather big, so the reasonable thing to do is to reduce the state space if possible.

State space reduction can be done by noticing that several states can be aggregated into one state in the Markov chain. From the point of view of the process, it does not matter which are the specific blocks included in the packets. The only relevant things are the degrees of the packets and the general composition of the packets: do the packets have common blocks together or do they not. Returning to the example above, states $\{b, ac\}$ and $\{c, ab\}$ can be aggregated together with state $\{a, bc\}$, as the general structure of all these is the same: one degree one packet and one packet where the other two blocks are combined with XOR-operation.

In general, all states which can be returned into each other by permutation of elements of the original file so that the general structure of the packet does not change, can be aggregated into one single state.

Using this state space reduction scheme, the number of states can be reduced to 12 when $N = 3$ and to 192 when $N = 4$. This is approximately the square root of the number of raw states 2^{2^N}. For $N = 5$ the number of raw states is $2^{32} \approx 4 \cdot 10^9$, square root of this is 63000, and so with $N = 5$ even the reduced state space is too big to do any useful calculations with the Mathematica modules constructed for this work.

4.1.2 State transition matrix generation algorithm

A state transition matrix for the Markov process with reduced states can be done easily after the state reductions. The following algorithm follows the Mathematica implementation of state transition matrix generation:

1. First generate reduced states in case $N = n$:
   (a) Generate all possible states, there are 2^{2^N} such raw states.
   (b) Make all possible state aggregations and provide unique representations for each of these reduced states.
2. Generate all possible arriving packets, that is, all combinations of blocks in the file.
3. With help of possible packets and states, generate matrix of possible next states.
4. Reduce the states in the matrix obtained in previous step.
5. Generate the symbolic state transition matrix in case $N = n$:
   (a) Calculate symbolic expressions for probabilities of possible arriving packets in terms of degree distribution $\rho(d)$.
   (b) Using symbolic expressions for probabilities, present the matrix obtained in step 4.
   $\rightarrow$ This is the state transition matrix $\mathbf{P}$ of reduced system.

4.1.3 Calculation of average number of steps to absorption

Now using the state transition matrix $\mathbf{P}$ we can calculate the average number of sent packets needed to recover the whole original file. This Markov chain has now the state where all blocks are decoded as an absorbing state. In the example with three blocks, using the

notation presented, this state is $\{a, b, c\}$. As this Markov chain is clearly finite, it can be written into the following canonical form:

$$\mathbf{P} = \begin{pmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{pmatrix},$$

where $\mathbf{Q}$ is the transition matrix between transient states, $\mathbf{R}$ represents transitions from transient states to absorbing ones and $\mathbf{I}$ is identity matrix corresponding to the absorbing states. When the state space is aggregated, the identity matrix $\mathbf{I}$ corresponds to just one state, i.e., $\mathbf{I} = \mathbf{1}$, an 1×1 matrix. Now the fundamental matrix $\mathbf{M} = (\mathbf{I} - \mathbf{Q})^{-\mathbf{1}}$ is well-defined with all elements positive and represents all possible transition sequences in the transient states without going to absorbing ones. A specific element m_{ij} in the fundamental matrix $\mathbf{M}$ tells the number of visiting times in state j before absorbtion when starting in state i. Using the fundamental matrix, average number of steps can then be calculated by multiplication with the initial distribution vector.

4.2 Case $N = 3$

Average number of packets needed to deliver the three block message is studied by constructing a Markov chain as described above. State transition probabilities are obtained from the degree distribution $\rho(d)$. In the following we consider both full and simple decoding for $N = 3$ and optimize the degree distribution in the sense of the two objectives presented in Section 3.3.

4.2.1 Full decoding

The state diagram of the fountain coding process when decoding is done by solving the linear system of equations is presented in Figure 6. Self transitions are not explicitly drawn. The original file consists of three blocks, a, b and c. Probabilities of different degrees are denoted by p_i, where i is the corresponding degree number. State transition matrix representing the chain is:

$$\mathbf{P_3^{full}} = \begin{pmatrix} 0 & p_1 & p_2 & p_3 & 0 & 0 & 0 & 0 \\ 0 & \frac{p_1}{3} & 0 & 0 & \frac{2}{3}(p_1+p_2) & \frac{p_2}{3}+p_3 & 0 & 0 \\ 0 & 0 & \frac{p_2}{3} & 0 & \frac{2}{3}p_1 & \frac{p_1}{3} & \frac{2}{3}p_2 & 0 \\ 0 & 0 & 0 & p_3 & 0 & p_1+p_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{(2p_1+p_2)}{3} & 0 & 0 & \frac{(p_1+2p_2+3p_3))}{3} \\ 0 & 0 & 0 & 0 & 0 & \frac{p_1+p_2+3p_3}{3} & 0 & \frac{2}{3}(p_1+p_2) \\ 0 & 0 & 0 & 0 & 0 & 0 & p_2 & p_1+p_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

where the overscript *full* refers to the fact that we are doing a full decoding ,i.e., solving the linear system of equations. Variables p_i are the probabilities from degree distribution $\rho(d)$, $p_i = \rho(i)$ and $p_1 + p_2 + p_3 = 1$.

Actual states in Markov chain are the states where a solid line starts and the absorbing state $\{a, b, c\}$ in Figure 6. Dashed lines represent state reductions, where a state is reduced to a equivalent representation. When using linear systems of equations to decode the original file, some additional state reductions can be done in addition to the reduction scheme presented in Section 4.1.1.

For example the state $\{ab, abc\}$ means that we have one packet created from blocks a and b (or b and c or a and c) and another one which is composed of all the three blocks. When solving a corresponding system of linear equations we can decode one packet fully (in case $\{ab, abc\}$, c can be solved) so the state can be reduced to $\{a, bc\}$.

Fundamental matrix can now be calculated as presented in the previous section, and with the initial distribution vector $\pi^{(0)} = (\mathbf{1}\ \mathbf{0}\ \ldots\ \mathbf{0})$ we obtain the average number of steps from beginning of the process to the end:

$$E\left[\text{number of steps before absorption}\right] = \boldsymbol{\pi}^{(0)}\mathbf{M}\mathbf{e}^{\mathbf{T}} = \boldsymbol{\pi}^{(0)}(\mathbf{I}-\mathbf{Q})^{-\mathbf{1}}\mathbf{e}^{\mathbf{T}} = \boldsymbol{\pi}^{(0)}\mathbf{A}^{-\mathbf{1}}\mathbf{e}^{\mathbf{T}}$$
$$= 8 + \frac{18}{p_1-3} + \frac{1}{1-p_2} + \frac{18}{p_2-3} - \frac{9}{2\,p_1+p_2-3} + \frac{2}{p_3-1} - \frac{9}{p_1+p_2+3\,p_3-3}, \tag{3}$$

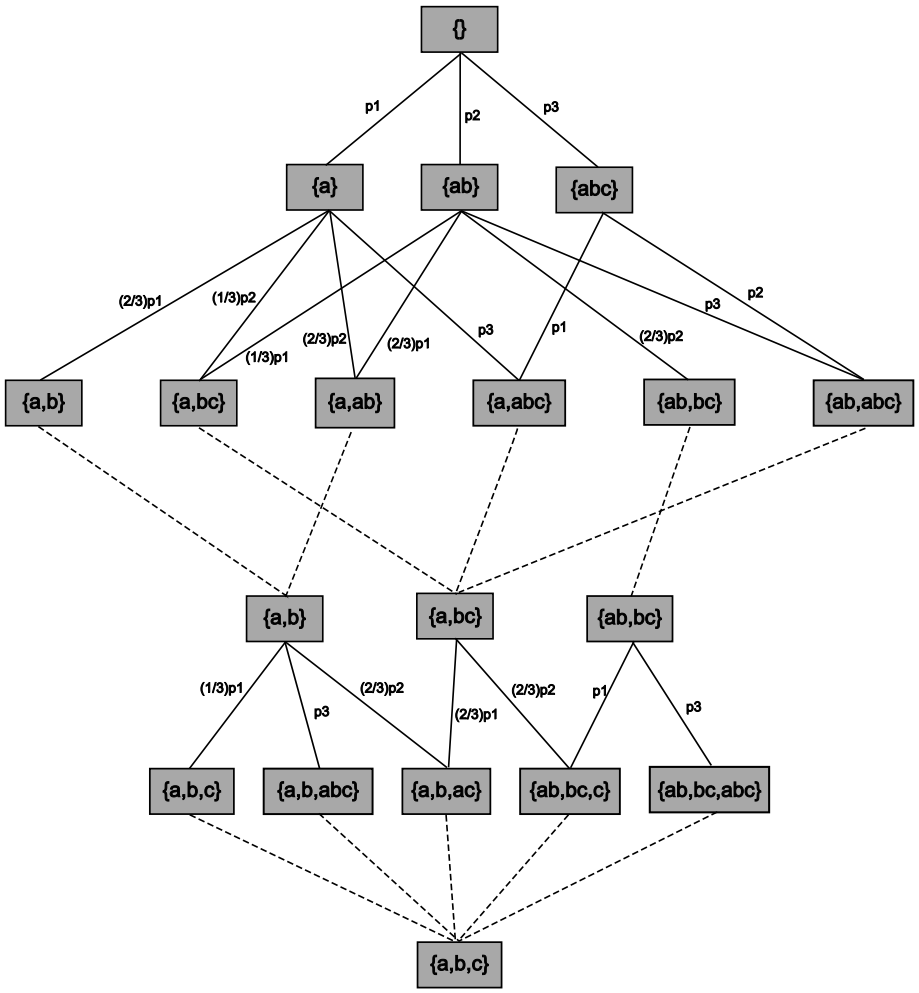

Figure 6: Decoding by solving a linear system of equations

where $\pi^{(\mathbf{0})} = (\mathbf{1}\,\mathbf{0}\,\ldots\,\mathbf{0})$, $\quad \mathbf{e^T} = (\mathbf{1}\,\ldots\,\mathbf{1})^{\mathbf{T}}$ and

$$\mathbf{A} = \begin{pmatrix} 1 & -p_1 & -p_2 & -p_3 & 0 & 0 & 0 \\ 0 & 1-\frac{p_1}{3} & 0 & 0 & -\frac{2}{3}(p_1+p_2) & -\frac{p_2}{3}-p_3 & 0 \\ 0 & 0 & 1-\frac{p_2}{3} & 0 & -\frac{2}{3}p_1 & -\frac{p_1}{3} & -\frac{2}{3}p_2 \\ 0 & 0 & 0 & 1-p_3 & 0 & -p_1-p_2 & 0 \\ 0 & 0 & 0 & 0 & 1-\frac{(2p_1+p_2)}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-\frac{p_1+p_2+3p_3}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-p_2 \end{pmatrix}.$$

From this result it is easy to calculate the optimum weights if we want to have as few steps as possible to decode the message. Numerical minimization using Mathematica yields the following results:

$$\begin{cases} p_1 = 0.429 \\ p_2 = 0.429 \\ p_3 = 0.143 \end{cases}$$

When weights are set as above, then the average number of packets needed to transfer the message succesfully is 3.917 which is seen by inserting above values into (3). With uniform distribution, $p_1 = p_2 = p_3 = \frac{1}{3}$, the average is $\frac{17}{4} = 4.25$ packets.

4.2.2 Simple decoding

Similar construction can be done when doing the decoding as presented in Section 3.2. The state transition diagram is the same as in full decoding case (Figure 6), with the exception that the state $\{ab, abc\}$ is now not reduced but is an explicit state. This is so because in the LT decoding algorithm, only received degree one packets allow the algorithm to continue operation. Now the process can continue to a different state when a degree one or a degree two packet different from already obtained one is received. The state transitions are presented in Figure 7.

The state transition matrix can be formed using Mathematica modules. The optimal weights can then be easily computed the same way as before. Optimized results for two different objectives presented in Section 3.3 are listed in Table 1. Objective *Min.Avg.* means minimization of average number of steps needed for decoding and *Max.Pr.* maximizing the probability of decoding in exactly three steps, this probability is denoted $\mathcal{P}_3$ in the table. Situation when degrees are uniformly distributed, i.e., when all degrees have the same probability, is also presented for comparison in Table 1. The differences between two objectives is not significant when looking at the excepted number of steps and probability for decoding in three steps. On the other hand, uniformly distributed packet degrees is not a good choice compared to the optimized cases. The average number of steps when all-at-once distribution presented in Section 3.3.1 is used can be calculated using (1). The result is 5.5 steps on average. This is worse than any of the results presented in Table 1.

4.3 Case $N = 4$

In the case of four blocks, the state space becomes much larger than in three block case if it is constructed similarly as before. As presented earlier, the three block-case has 12 different reduced states. With four blocks this grows to 192 reduced states.

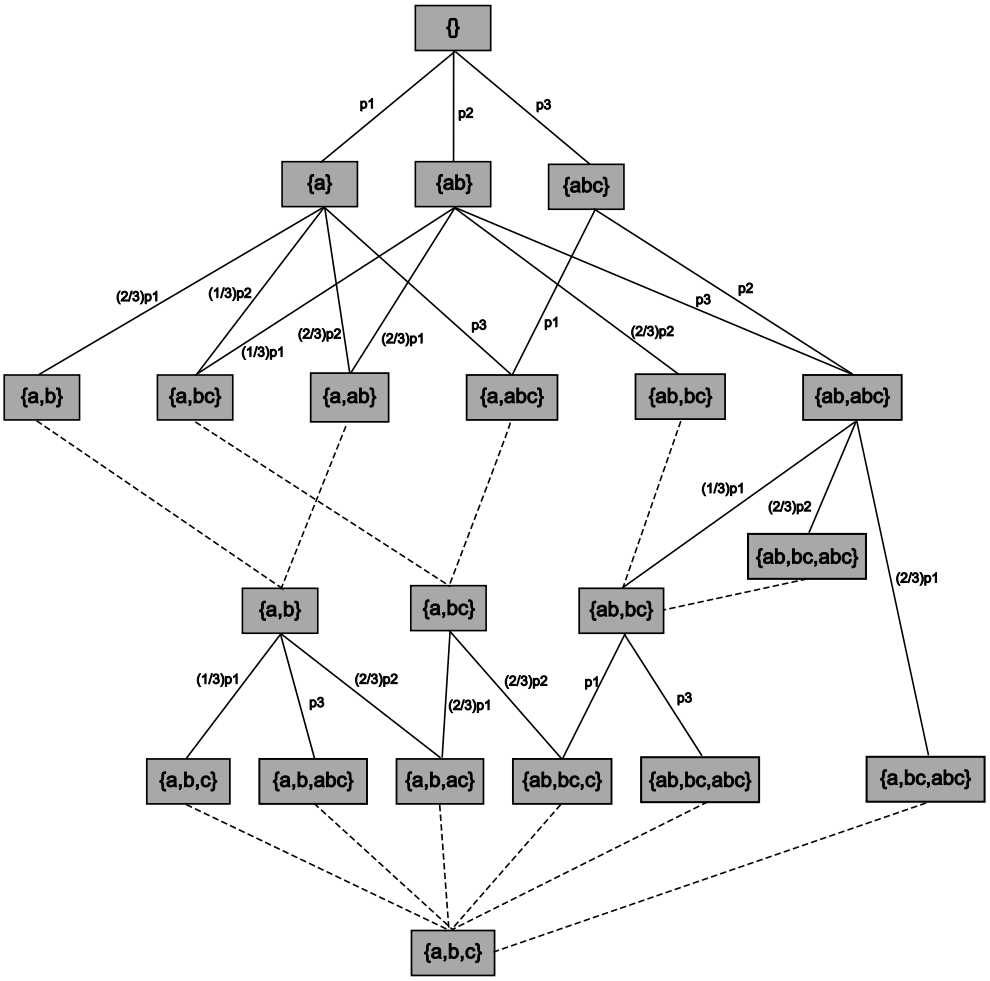

Figure 7: Decoding by fountain code approach

Table 1: Optimal weights in case $N = 3$ for simple decoding

	MinAvg	MaxPr	Uniform distr.
p_1	0.524	0.517	0.333
p_2	0.366	0.397	0.333
p_3	0.109	0.086	0.333
$E[\text{steps}]$	4.046	4.049	4.725
$\mathcal{P}_3$	0.451	0.452	0.354

With the symbolic state transition matrix, the optimal weights can be calculated similarly as in case $N = 3$. Symbolic expression for the average number of steps to absorbtion

Table 2: Optimal weights in case $N = 4$ for simple decoding

	MinAvg	MaxPr	Uniform distr.
p_1	0.422	0.429	0.25
p_2	0.385	0.430	0.25
p_3	0.112	0.100	0.25
p_4	0.061	0.041	0.25
E[steps]	5.580	5.590	7.182
$\mathcal{P}_4$	0.314	0.315	0.183

as function of different packet degree probabilities is calculated using Mathematica, and numerical optimization of this gives the needed results.

Only the simple decoding is considered here. The results for degree weights are presented in Table 2. Objective *Min.Avg.* is the same as before and the objective *Max.Pr.* is to maximize the probability of decoding in four steps. Again the results for uniform distribution are presented for comparison.

In this case the difference between the two objectives is insignificant. With good degree distribution, the message can be delivered using 5.6 packets on average. Uniform distribution is almost 1.5 packets worse when comparing the average numbers. Also the probability for a succesful decoding with exactly four steps is nearly half of the probability when using the optimal weights. All-at-once distribution for $N = 4$ gives (Equation (1)) now 8.333 steps on average, which is again a rather poor result if compared with the other presented distributions.

5 Simulation results

5.1 Simulator

The simulator used is written as a Mathematica module. This module first generates a packet as a vector of included blocks, and then runs the decoding process if degree one packets are available. Otherwise already received vectors are stored and a new one is generated. The simulator takes as a parameter the list of possible degrees, so packets from an arbitrary degree distribution can be generated.

5.2 Simulations with different degree distributions

5.2.1 Comparison between analytical results and the simulator

The correctness of simulator can be checked by calculating the average number of steps needed in optimal weight cases. The mean m and standard deviation σ were calculated from 10000 runs of the simulator with optimal degree distributions and uniform distribution. The results are presented in Table 3.

Table 3: Simulation results for simple cases

		m	σ
$N = 3$	Min.Avg.	4.056	1.342
	Max.Pr.	4.082	1.375
	Uniform distr.	4.737	2.123
$N = 4$	Min.Avg.	5.584	1.727
	Max.Pr.	5.595	1.766
	Uniform distr.	7.134	3.151

The results are near the analytical ones calculated in Section 4. With taking the standard deviations into account, we can be fairly sure that the simulator produces the same answers as presented in previous section.

5.2.2 All-at-once and uniform distribution

Equally good heading for this section would be “useless distributions”.

Results from using the all-at-once distribution defined in Section 3.3.1 are presented in Table 4 and Figure 8. The size of file considered here is 100 blocks. Statistics have been collected from 100 simulator runs and are compared to results obtained from using uniform distribution.

Table 4: All-at-once and uniform distributions compared

Distribution	m	σ
All-at-once	522	118
Uniform	2057	294

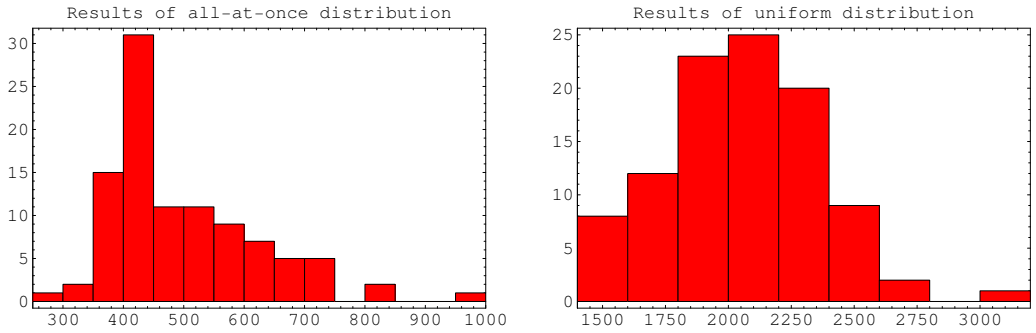

Figure 8: Histograms from simulations using all-at-once and uniform distributions. Horizontal axis shows the amount of packets needed for the decoding to complete. Number of blocks in the processed file is $N = 100$.

Results for all-at-distribution follow the expectation presented in Section 3.3.1 quite nicely, as for hundred blocks $\log N = \log 100 \approx 4.6$. The results are bad considering that with all-at-once distribution we need to send 400-700 packets and with uniform distribution even more; 1750-2500 packets. This example clearly shows that attention has to be put into finding a good degree distribution.

5.2.3 Ideal and robust solition distribution

Simulations for ideal and robust soliton distributions were also made to show that there exists reasonable good distributions for approximating the digital fountain. The soliton and robust soliton distributions were introduced in Section 3.3. Histograms of the process with 100 runs with these distributions are presented in Figure 9. Parameters for the robust soliton distribution were $c = 0.1$ and $\delta = 0.5$

Not surprisingly the robust soliton distribution seems to perform a little better here. Nearly all cases needed less than 160 encoding packets, and the usual value is somewhere around 120 packets. The soliton distribution performs well in some cases but has some unwanted spread in the region of $200 - 400$ packets. The overhead around 20% in average obtained using the robust soliton distribution is much better than any of the other distributions presented in this paper. For N around 10000 blocks the robust soliton distribution can be tuned so that the overhead is around 5%.

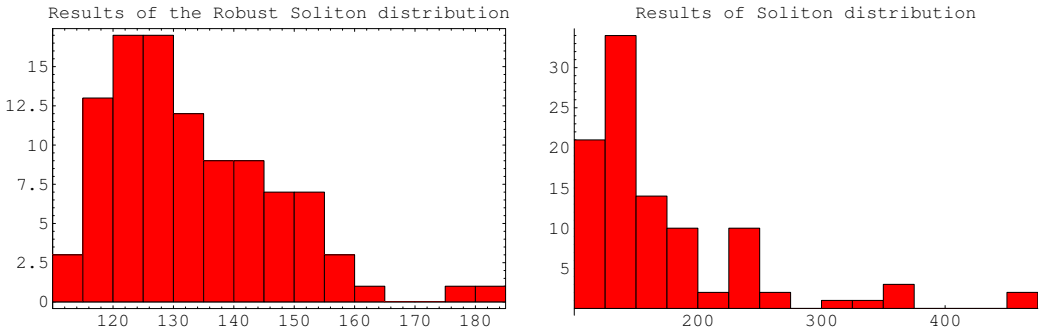

Figure 9: Histograms from simulations using the soliton and robust soliton distributions for the case $N = 100$. Note that the overhead in packets is much lower compared to Figure 8.

The arguments presented by Luby in his paper [Lub02] for the robust soliton distribution are somewhat heuristic. Thus, probably better distributions exist for large cases and, also, in general.

6 Conclusions

Digital fountain codes provide interesting alternative to traditional acknowledgement based transmission protocols. Possible applications where fountain coding could be used are numerous: large file transfer, packet transmission over one-way channel, satellite transmission, distributed storage and so on. One particular application, where digital fountain codes are useful, are peer-to-peer or P2P networks. Receiving the same file simultaneously from many peers using fountain codes is easy to implement if degree distribution and parameters are agreed upon in advance.

There are, however, some difficulties in designing a good digital fountain encoding and decoding scheme, most notably the designing of a good degree distribution.

The LT codes discussed in this paper are very sensitive to the degree distribution, and even in the cases where the file size in blocks is small, much work has to be done to derive the optimal degree distributions. The Markov chain based method described in this paper is not scalable to larger cases so alternative ways must be found.

Designing optimal degree distributions for practical cases, where the file size is several hundred or thousand blocks, provides interesting computational problems and area for further research work.

References

[BLMR98] John W. Byers, Michael Luby, Michael Mitzenmacher, and Astutosh Rege. A digital fountain approach to realiable distribution of bulk data. In *Proceedings of ACM SIGCOMM '98*, pages 56–67, 1998.

[Lub02] Michael Luby. LT Codes. In *Proceedings of The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[RFC3695] M. Luby and L. Vicisano. Compact Forward Error Correction (FEC) Schemes. RFC 3695, Internet Engineering Task Force, February 2004.

[RFC3452] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. Forward Error Correction (FEC) Building Block. RFC 3452, Internet Engineering Task Force, December 2002.

[RFC3453] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft. The Use of Forward Error Correction (FEC) in Reliable Multicast. RFC 3452, Internet Engineering Task Force, December 2002.

[Mac03] David J. C. Mackay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[Sho] Amin Shokrollahi. Raptor codes. Preprint at `http://algo.epfl.ch/pubs/raptor.pdf`.

[Wol] Wolfram Research Inc. Mathematica 5.0. `http://www.wolfram.com`.

A Mathematica source code

This appendix presents the Mathematica package used for generating the results. Package consists of functions for state transition matrix generation and step calculation to final state; another important part is the simulator which runs the LT process with specified degree distribution.

A.1 Package fountain.m

```
(*************************************************************

  Digital Fountain codes - simple case

  File :      fountain.m
  Version :   0.5
  Authors:    Tuomas Tirronen

  Related Packages:

    <none>

  Changes:

    14-02-2005 Package created
    21-02-2005 First working(?) version
    24-02-2005 Changed almost everything, this should work
    25-02-2005 Some changes, e.g, parameter to simulator is function
               which returns random symbol list instead of degree list
    07-03-2005 Added functions to calculate number of different states
               of decoding process
    26-04-2005 Cleaning up
    04-11-2005 Final version for special assignment

*************************************************************)

BeginPackage["fountain`"];

Options[FountainSim]={Debug->False};

ListDeg::usage="listdeg [ p ]: Returns an integer from probability distribution
specified by p (list of probabilities )";
```

```
DegVect::usage="degvect[ deg, n ]: Returns a list consisting of 'deg' unique
random symbols from the range 1,...,' n'. ";

FountainSim:: usage="FountainSim[ deg, n ]:\ n
\n
Paramter 'deg' is the degree distribution of packets, and\n
parameter 'n' is length of message in packets.\ n
\n
If optional argument Debug == True, then running simulate prints debug messages\n
\n
See also: DegVect and ListDeg.\ n
";

(******* State calculations *******)
<< DiscreteMath`Combinatorica`

Replicas :: usage="Replicas[x, elements]: Shows states identical to x, elements
is the list of possible elements in message";

NofReplicas:: usage="NofReplicas[x, elements]: Shows number of states
identical to x, elements is the list of possible elements in message";

UniqueRepr::usage="UniqueRepr[x,elements]: Shows the state used for
unique representation on state set where x belongs to, elements is the
list of possible elements in message";

StateSubsets :: usage="StateSubsets [n]: Returns list of possible states
when number of possible elements is n, all states with at least one
element of length one are excluded.\ n
See also: States . \n"

StateTable :: usage="StateTable [n]: Displays table of all possible states
when number of elements is n.\ n See also: States .\ n";

States :: usage="States [n]: Displays all reduced states of n elements";

(****** Creation of state transition matrix ******)
AppStates:: usage="AppStates[n]: Intermediate matrix used when calculating
state transition matrix. Result is combination matrix of all possible
states appended with every possible packet.\ n
```

```
   See also: UrStates, TransMatrix.\n";

76 ArrPackets:: usage="ArrPackets[n]: Shows proper packets received in the
   decoding process when the total number of elements is n\n
   See also: TransProbs, TransMatrix.\n";

   UrStates:: usage="UrStates[n]: Intermediate matrix used when calculating
81 state transition matrix. Resulting matrix shows reduced states of AppStates
   matrix.\n
   See also: AppStates, TransMatrix.\n";

   TransProbs:: usage="TransProbs[n]: Shows (symbolic) probabilities of
86 arriving packets, here p[1] stands for degree 1 packets, p[2] for
   degree 2 and so on.\n
   See also: TransMatrix.\n";

   TransMatrix:: usage="TransMatrix[n]: Shows state transition matrix when
91 there is n possible elements in the decoding process";

   (******** case n=3 *********)

   StepsToLastState :: usage="StepsToLastState[P]: Returns the mean no.
96 of steps from state i to last state n.";

   MFull3::usage="MFull3[p1,p2,p3]: Returns reduced state transition
   matrix of fountain code decoding
   in case of n=3 using a perfect decoder.";
101
   MSimple3::usage="MSimple3[p1,p2,p3]: Returns reduced state transition
   matrix of fountain code decoding
   in case of n=3 using a 'simple' decoder.";


106
   Begin["`Private`"];

   (***************** function implementations **********************)

111 (**** A random number based on the list of probabilities ****)
   ListDeg[ p_ ] := Module[
       {
           sum=0, tr =Random[]
```

```
    },
    First [ First [ Position [ Map[ sum += #&, p] , _?(# >= tr &) ]]]
];


(**** Generation of new vector ****)
DegVect[ deg_, n_ ]:= Module[{ nv={}, x, ddeg=If[ 2*deg<n, deg, n-deg ] },
  Do[ While[ MemberQ[ nv, x = Random[ Integer, {1,n} ]]]; nv=Append[nv,x], {ddeg} ];
  If [ ddeg==deg, nv, Complement[ Range[1,n], nv ] ]
];

(**** Simulator itself ****)
FountainSim[ newvect_, n_, opts___ ] := Module[
    {
        rounds, nv, worklist , decoded, newsyms, newsym, tmp,
        debug = Debug /. {opts} /. Options[ FountainSim ]
    },
    rounds   = 0;
    decoded  = {};   (* List of decoded symbols *)
    worklist = {};   (* Working list : keeps track of contents of received packets *)

    While[ Length[ decoded ] < n,

        rounds++;
        nv = newvect[];

        If [ debug,
          Print["**** Round ", rounds , " **** "];
          Print["New vector = ", nv ];
        ];

        (**** Processing of new vector ****)
        nv = Complement[ nv, decoded ];
        Which[
          Length[nv] == 1,    (* new one packet element *)
            newsyms = nv;
            While[ newsyms != {},
              If [ debug,
                Print["New resolved symbols = ", newsyms ];
                Print["Worklist before purge = ", worklist ];
              ];
```

```
156                 Scan[ worklist =DeleteCases[ worklist , #, {2} ];&, newsyms ];
                    (*worklist  = Map[ Complement[#,newsyms]&, worklist ];
                      "equally fast as above" *)
                    worklist  = Union[ DeleteCases[ worklist , {}, {1} ] ];
                    decoded  = Join[ decoded, newsyms ];
161                 newsyms = Flatten[ Select [ worklist , Length[#] == 1& ] ];
                  ];
                , (* else if *)
                  Length[nv] > 1,       (* new composite element *)
                    worklist  = Append[ worklist, nv ];
166             ];

                If[ debug,
                  Print["Worklist after purge = ", worklist ];
                  Print["Decoded symbols      = ", decoded ];
171               Pause[0.1]
                ];
        ];
        rounds
    ];
176
    (********** State calculations **********)
    (*\label{states}*)
    Replicas[ x_, elements_ ] := Union[ Sort[ Sort / @ #] & /@
        (x /. Thread[ Rule[ elements, # ] ] & /@ Permutations[ elements ] ) ]
181
    NofReplicas[ x_, elements_ ] := Length[ Replicas[ x ] ]

    UniqueRepr[ x_, elements_ ] := Union[ Sort[ Sort / @ #] & /@
        (x /. Thread[ Rule[ elements, # ] ] & /@ Permutations[ elements ] ) ] // First
186
    StateSubsets [n_] := Subsets[DeleteCases[Subsets[n] // Sort // Rest, {_}]]

    StateTable [n_] := (Table[
                         (Table[
                          {n + 1 - j}, {j, 1, i}]~Join~# &) / @
191                       StateSubsets [n - i]  , {i, 0, n - 2}]
                          // Flatten[#, 1] &)~Join~{Table[
                              {i}, {i, n - 1}]}~Join~{Table[{i}, {i, n}]}

196 States [n_] := Module[{elements = Range[n], rawstates  = StateTable [n]},
```

```
        UniqueRepr[#,elements]& /@ rawstates // Union
]

(********** Generation of state transition matrix **********)

(************** Case n=3 **************************)

StepsToLastState [P_] := Module[
  {
    A, b,
    n = Length[P]
  },
  A = Append[Drop[P - IdentityMatrix[n], -1], Append[Table[0, {n - 1}], 1]];
  b = Append[Table[-1, {n - 1}], 0];
  LinearSolve[A, b]
];

(* ******* Alternate way to compute nof steps to last state ********
   StepsToLastState [P_] := Module[
  {
    A, b,
    n = Length[P]
  },
  d = Position [P,1] // Flatten ;
  A = Append[Drop[IdentityMatrix[n]-P, {d[[1]]}], Append[Table[0, {n - 1}], 1]];
  b = Append[Table[1, {n - 1}], 0];
  LinearSolve[A, b]
]; *)

(* ******* Alternate way to compute nof steps to last state ********S
 StepsToLastState [m_, n_] := Module[
    {
      l = Length[m]
      },
    pos = Position [m, Plus @@ Array[p, n]] // Flatten ; (*
      searches place of last state eg. position of p[1] + p[2] + p[3] *)

    Inverse[Drop[m, {pos[[1]]}, {pos[[2]]}] -
          IdentityMatrix[l - 1]. Table[-1, {l - 1}]
]; *)
```

```
(* Matrix for full decoding *)
MFull3[p1_,p2_,p3_]:=
    {
      {0,p1,p2,p3,0,0,0,0},
      {0,p1/3,0,0,(2/3)(  p1+p2),p2/3+p3,0,0},
      {0,0, p2/3,0,(2/3)(  p1), p1/3+p3,(2/3) p2,0},
      {0,0,0,  p3,0, p1+p2,0,0},
      {0,0,0,0,(2   p1+p2)/3,0,0,(1/3)(  p1+2 p2+3 p3)},
      {0,0,0,0,0,(1/3)(   p1+p2)+p3,0,(2/3)( p1+p2)},
      {0,0,0,0,0,0,   p2,p1+p3},
      {0,0,0,0,0,0,0,0,1}
      };

(* Matrix for simple decoding *)
MSimple3[p1_,p2_,p3_]:=
    {
      {0,p1,p2,p3,0,0,0,0,0},
      {0,p1/3,0,0,0,(2/3)(   p1+p2),p2/3+p3,0,0},
      {0,0, p2/3,0, p3,(2/3)( p1), p1/3,(2/3)  p2,0},
      {0,0,0,  p3,p2,0, p1,0,0},
      {0,0,0,0,  p2/3+p3,0, p1/3,(2/3)  p2,(2/3) p1},
      {0,0,0,0,0,(2   p1+p2)/3,0,0,(1/3)(  p1+2 p2+3 p3)},
      {0,0,0,0,0,0,(1/3)(   p1+p2)+p3,0,(2/3)( p1+p2)},
      {0,0,0,0,0,0,0,   p2+p3,p1},
      {0,0,0,0,0,0,0,0,1}
      };

End[];
EndPackage[];
```