Programming Comps - January 17-24, 2012

LT Erasure Codes

Out: Tuesday, Jan 17th, 2012 9AM
Due: Saturday, Jan 21st, 2012, 11:59AM
Interview: Tuesday, Jan 24th, TBD

1 Introduction

Erasure codes are a type of encoding of data, generally for transmission over a lossy medium, that survive deletions (erasures) of parts of the message.[1] They are especially useful for transmission of data across a medium or network that can drop packets of data, when it is impractical for the receiver to be in constant communication with the sender. Given a file with K blocks, the sender generates $B > K$ encoded blocks (B/K is called the *rate* of the code). The code is designed so that after receiving *any* set of blocks of size at least K', for some K' slightly larger than K, the original data can be decoded with high probability. For scenarios like broadcast, or for networks with very long one-way delays (think the Mars rover sending an image to Earth), this is much more practical than the receiver acknowledging every block, as done in the Transmission Control Protocol (TCP), which is used on the web and by many other internet applications.

Reed-Solomon codes are a type of erasure code, but are not very practical for many applications, in particular because you have to set the rate prior to encoding and transmitting. The problem is that the rate might need to change depending on the quality of the channel at the receiver(s)! There exist codes, however, that are *rateless*, in that a practically infinite number of coded blocks can be generated from a fixed set of source blocks, and the receiver can still can decode the original set of blocks with little overhead. These codes are also called *fountain* codes, in an analogy to a constant stream of water from a fountain; any set of drops from the fountain will serve the purpose of filling the receiver's bucket.

In this exam you will implement LT (Luby transform) Codes, which were the first practical rateless erasure codes, and were invented in 1998 by Michael Luby and colleagues [1]. We will base our description on chapter 50 of the book by MacKay [2], which is freely available for download. We recommend that you read that chapter before beginning your project. In particular, you will need to understand the encoding and decoding algorithms described in Sec. 50.1-50.2. We will use the distributions described in Sec. 50.3, but you do not need to understand their justification.

LT Codes form the basis of the current state of the art in rateless codes, called Raptor Codes. Raptor codes are faster than LT Codes, and generally require fewer blocks to decode, with K' very close to K. They are used in several communication standards such as in broadcast of video to mobile devices. Their implementation, however, is more involved.

[1] External methods, such as checksums, are used to detect and discard parts of the encoded data whose contents have changed (contain errors). We focus here on algorithms for encoding, and subsequently decoding, the original data from an error-free subset of the transmitted message.

2 Your Task

Your task is to implement two programs. The first, an encoder, reads a file and generates another file with blocks encoded by an LT Code. The second, a decoder, reads one such encoded file and either successfully decodes it, or says that it has insufficient information to decode.

LT Codes depend on randomness for their implementation, and we will take care to specify how you will generate the required (pseudo-)random numbers so that the encoder and decoder will work deterministically, given proper pseudo-random seeds.

2.1 Algorithm

LT Codes comprise two main algorithms, one for encoding and another for decoding. We briefly sketch them here, and refer to [2] for further detail.

Consider a source file with K fixed-length blocks s_k, $k = 1, \ldots, K$. We assume a *degree distribution* $\mu(d)$ is provided, which is a discrete probability distribution (probability mass function) on integers between 1 and K: $\mu(d) \geq 0$, $\sum_{d=1}^{K} \mu(d) = 1$. Each encoded packet t_n in the *digital fountain* is then produced as follows:

1. Randomly sample the degree d_n of the packet from $\mu(d)$.
2. Choose, uniformly at random, d_n distinct input blocks. Set t_n equal to the bitwise sum, modulo 2, of these d_n blocks.[2]

The encoded message is then these encoded packets, plus sufficient information for the decoder to determine which source blocks where combined to produce each packet.

Now suppose that N encoded packets $t_1 \ldots t_N$ have been successfully received. For each packet t_n, construct a list of the source blocks s_k which were used to encode that packet. The decoder then proceeds as follows:

1. Find a packet t_n which has *exactly one* source block s_k in its list. If no such packet exists, the decoder halts and fails. Otherwise:
   (a) Set $s_k = t_n$.
   (b) Set $t_{n'} = t_{n'} \oplus s_k$, for all packets $t_{n'}$ which include source block s_k in their encoding lists.
   (c) Delete source block s_k from *all* encoding lists.
2. Repeat step 1 until all source blocks are decoded.

As discussed by MacKay [2], it may be helpful to visualize the decoder using a sparse bipartite graph, in which edges show which source blocks are encoded by each packet.

For those who are curious, this decoder is a special case of the celebrated *sum-product* or *loopy belief propagation* (BP) algorithm. Because there can be no errors in received packets, only complete erasures, the general BP algorithm substantially simplifies for LT codes.

We now discuss several important aspects of the implementation which you must follow.

2.2 Robust Soliton Distribution

While the encoder and decoder in Sec. 2.1 are valid algorithms for any degree distribution, the decoder only succeeds with high probability if $\mu(d)$ is chosen with care. A starting point is the *ideal soliton distribution*:

$$\rho(1) = \frac{1}{K}, \quad \rho(d) = \frac{1}{d(d-1)} \text{ for } d = 2, 3, \ldots, K. \tag{1}$$

[2] This is equivalent to the bitwise XOR operation, denoted $\oplus$, on the blocks.

This distribution optimizes the expected probability that there is one decodable source block at each iteration, but has an unacceptably high probability of failing at *some* iteration. To add robustness, we define the following non-negative function:

$$\begin{aligned}
\tau(d) &= \frac{S}{K}\frac{1}{d} && \text{for } d = 1, 2, \ldots, \lfloor K/S \rfloor - 1, \\
\tau(d) &= \frac{S}{K}\ln(S/\delta) && \text{for } d = \lfloor K/S \rfloor, \\
\tau(d) &= 0 && \text{for } d > \lfloor K/S \rfloor, \\
S &= c\ln(K/\delta)\sqrt{K}. &&
\end{aligned}$$

Here, $0 < \delta < 1$ is a (conservative) bound on the probability that the decoding fails to succeed after a certain number of packets are received. $c > 0$ is a free parameter, which can be tuned to optimize performance. The *robust soliton distribution* is

$$\mu(d) = \frac{\rho(d) + \tau(d)}{Z}, \qquad Z = \sum_{d=1}^{K} \rho(d) + \tau(d). \tag{2}$$

The inclusion of Z creates a properly normalized distribution which sums to one.

The robust soliton distribution of Eq. (2) defines the distribution $\mu(d)$ which you will use when implementing your encoder. To sample from $\mu(d)$, first compute the corresponding cumulative distribution function:

$$M(d) = \sum_{d'=1}^{d} \mu(d') \tag{3}$$

Let u denote a number uniformly distributed between 0 and 1, for example drawn from the pseudo-random generator of Sec. 2.3. We can then construct a sample d from $\mu(d)$ by finding the unique bin (degree) for which $M(d-1) \le u < M(d)$, where $M(0) = 0$.

For this assignment, we will fix the parameters for the distribution. You will use the values of $c = 0.1$ and $\delta = 0.5$.

2.3 Pseudo-Random Number Generation

As we will see in Sec. 2.4, even though the algorithms depend on randomization, we need the precise sequence of (pseudo-)random numbers used to be reproducible. To this end, you must use the pseudo-random generator we define here. We will use a very simple pseudo-random generator, a variant of a linear congruential generator, known as the Lehmer generator.[3] With the particular parameters specified below, it is called MinStd [3]. The generator is defined by the following equation:

$$next = A \cdot state \mod M \tag{4}$$

We will use $A = 16,807$ and $M = 2^{31} - 1 = 2,147,483,647$. M is a Mersenne prime, and A is a primitive root modulo M, which guarantees maximum period for the random sequence. We define three operations on a generator R:

1. R.nextInt(): returns *next*, and sets *state* = *next*.
2. R.setSeed(S): sets *state* = S.

[3] Although serving our purposes here, this pseudo-random number generator is a terrible choice for cryptography applications, as well as for use in Monte Carlo simulations.

3. R.getState(): returns *state*.

You should take care to not overflow the integer type of your language in the multiplication. Here is a snippet of C code that implements nextInt() observing the width of the data types:

```
uint32_t M = 2147483647UL;
uint32_t A = 16807;
uint32_t MAX_RAND = M - 1;

uint32_t state;

uint32_t nextInt() {
     uint32_t next = (uint32_t)(((uint64_t)state * A) % M);
     state = next;
     return next;
}
```

To produce a number uniformly distributed between 0 and 1, which you need for generating samples from $\mu(d)$, you should use double precision and divide the obtained integer by $M-1 =$ MAX_RAND, defined above.

We have provided a sequence of samples from this random number generator in Appendix A, which you can use as an indication that your generator is producing correct samples.

2.4 Encoding the List of Blocks

One important aspect of the decoder is that it needs to know, for each encoded packet, the number and identity of the source blocks from which it was created. Instead of encoding the list explicitly in the packet, which could be wasteful, we will have the decoder generate this list using the same process as the one used by the encoder. Since this involves sequences of (pseudo-)random numbers, we will have to make sure the programs generate the same sequence for each block.

We will store in the encoded block the internal state of the random generator immediately before encoding the block. This state, for our generator from Sec. 2.3, is simply a 32-bit number, which we call the *seed* for the block. Given this seed, we will follow the steps below for the block. Since the state of the generator changes with each invocation, it is important to follow these steps exactly:

1. Before processing block t_n:
   (a) If encoding t_n, t_n.seed = R.getState()
   (b) If decoding t_n, R.setSeed(t_n.seed)
2. Generate r = R.nextInt() and use it to generate d from the *robust soliton distribution* (see Sec. 2.2).
3. Generate d *distinct* numbers between 0 and $K-1$, using (R.nextInt() mod K) for each one. In case of repetition, keep generating new numbers until you get d distinct source blocks. This is the list of source blocks corresponding to this encoded block.

Note that according to this, the source blocks are numbered 0 to $K-1$. Appendix B has a list of blocks generated in sequence with a fixed seed so you can compare your program.

2.5 Programs

It is your task to *implement the algorithms outlined in this document*. Do not make up a different approach to tackle the problem. If you do, you will fail the exam. The choice of algorithm is fixed, as are several of the parameters that make it possible for your encoded files to be decoded by our decoder, and our encoded files to be decoded by your decoder. That said, you are free to choose the internal data structures you will use in the encoder and the decoder, and should justify your choices in terms of practicality and efficiency.

You will write two executable console programs: `encode` and `decode`. `encode` will receive the block size, a random seed, a rate, and the name of a file to encode. Assume the file will be in the same directory as the program, to simplify handling of the name. `encode` will be called as follows:

```
$> encode <block_size> <seed> <rate> <file>
```

Where:

block_size is an integer, the size of each encoded block, in bytes.

seed is an integer, the initial seed for the random number generator.

rate is a real, > 1, controlling the number of encoded blocks generated. If there are K source blocks, your program should generate $r \cdot K$ encoded blocks.

file is the name of the source file to be encoded.

If the name of the input file is `file`, `encode` will create an output file named `file.lt`. We describe the format of the file in Sec. 2.6.

`decode` will receive the name of an encoded file, which will have all of the necessary information to decode it. Section 2.6 describes the file format in detail. It will be invoked as follows:

```
$> decode <file>
```

The output of `decode` will be one of the two options, depending on whether it successfully decodes the file:

```
Successfully decoded <file>.lt into <file>.lt.dec
```

or

```
Failed to decode <file>.lt
```

In case of successful decoding, the produced file must be *identical* to the original file. In particular, you should take care to truncate the last block of the file if the file size is not a multiple of the block size.

If you use a language that does not produce executable files directly, such as Java, you must provide wrapper shell scripts that take the arguments as described above and call the programs with the right command line. This is VERY important, as we will use automated tools to run your program.

You don't have to worry about malformed encoded files. You also don't have to worry that the files used for testing won't fit in memory, i.e., you may assume that the decoder, for example, can hold the contents of the encoded and decoded blocks in memory. With that said, you should make sure that you can decode at least a 100MB file.

You can choose to have the decoder read all encoded blocks before starting to decode, or start decoding as it reads the blocks. Bear in mind, however, that for the evaluation (see Sec. 3.3), you will need to know the minimum number of encoded blocks needed to successfully decode a file.

2.6 Data Format

The remaining aspect that we need to specify is the on-disk format for the encoded files. The file is to be written in **binary** format, not in text format. Integers in the file are to be written in **big-endian format**, also known as network byte order. This means that the integer 0x01020304 will be written to the file as the sequence of bytes 0x01, 0x02, 0x03, and 0x04. This convention is important for compatibility between your programs and ours, regardless of the language you write them in.

For Java, the methods `DataOuput.writeInt` and `DataInput.readInt` will do the right thing. For C and C++, you will need to use the conversion functions `htonl` before writing, and `ntohl` after reading. Other languages have similar mechanisms. Note that this only applies to integers, and byte sequences are written in the same order as they are in memory.

The file format is simple: there is a header which has the parameters for the decoder program, followed by a sequence of encoded blocks. The following C-like listing describes the format:

```
typedef struct {
    uint32_t 0x01020304; //endianness marker
    uint32_t B;          //block size (bytes)
    uint32_t E;          //number of encoded blocks
    uint32_t F;          //size of original file (bytes)
    uint32_t K;          //number of source blocks
} header_t;

typedef struct {
    uint32_t seed;       //seed for the block
    uint8_t  data[B];    //block data
} encoded_block_t;

typedef struct {
    header_t header;
    encoded_block_t blocks[E];
} encoded_file_t;
```

Use the first integer to make sure you are writing the integers correctly. The first byte of the file will always be 0x01, followed by 0x02, 0x03, and 0x04. You can use the `hexdump` program to verify this, if you want.

3 Logistics

3.1 Material

This document is located at `http://www.cs.brown.edu/~rfonseca/comps/2012comps.pdf`.

3.2 What to Hand In

You should hand in a single file named `comps2012_<your_id>.tar.gz`, containing:

- your source files
- a file called README with *detailed* instructions on how to compile both programs. You are responsible for testing that your instructions work on a standard departmental linux machine.
- a file called Documentation.pdf, documenting and evaluating your program. Please see section 3.3 for what to include in the evaluation. This file must be a typeset PDF document.

After compiling your program, there should be two executable files (which may be wrapper scripts) named `encode` and `decode`, which we will invoke with different parameters and files. We will test your encoded files with our decoder, and your decoder with our encoded files.

Do not hand in any sample input data; we will test the program with our own input files!

To create the hand-in archive, please follow these steps to maintain the anonymity of your submission:

1. Create a directory called `comps`.
2. Copy the files to be handed in into the `comps` directory.
3. From `comps`' parent directory, issue the following command (where id is your comps id):

```
tar -c -v -z --owner <id> --group <id> -f comps2012_<id>.tar.gz comps
```

If you don't use the owner and group options, the archive will contain your real user and group names, and won't be anonymous.

You will hand in the archive file by email to `marek@cs.brown.edu` by the deadline, Saturday, January 21st, 2012, at 11:59AM. This is a HARD deadline, and won't be extended. If you don't hand in the exam by then, you will fail.

3.3 Documentation and Evaluation

As much as it is a programming exam, this is also a test of how you approach and solve problems. As in the course of the PhD program, and after you graduate, you will routinely face problems for which there is no prior solution; you will have to convince yourself that your solution is correct and efficient. Whenever you publish your results, you will create a permanent record of your work, which will be open to the scrutiny of the entire research community, forever. Thus, it is of utmost importance that you are confident of the results you publish.

Accordingly, in this exam, you must use sufficient means to know that what you are delivering is correct, and works according to the specification. This involves, but may not be limited to, having sufficient test cases. We will not give out extra test cases, and, in particular, we may test with larger files.

A key requirement of this exam is that you describe, test and evaluate your programs. Testing asserts the correctness of the programs. Evaluation asserts that the programs perform and scale as they should. All this must be in the documentation.

The documentation should allow someone to understand the important algorithms in your code. You should also justify all of your choices not covered in the specification. For example, what are the tradeoffs involved? How did you resolve them?

Your documentation should, at least:

- Describe your implementation. What are the algorithms you used for key parts? For example, how do you represent the encoded and source blocks in the decoder, and their relationships?
- Evaluate the efficiency of the encoding. For example, for a fixed K, and the given parameters for the robust soliton distribution, what is the distribution of the minimum number of blocks required to decode the file? Is this in agreement with the theory? Can you plot the distributions for different values of K? (See Figure 50.4 in MacKay's book [2]).
- Describe your testing strategy. How do you know your implementation is correct? Are there any corner cases?

- Evaluate the scalability of your programs. What parameters affect the speed and the memory required? Does it agree with what you would expect or theoretically predict? How do encoding and decoding speed vary with the number of blocks, and with the size of the file?
- Justify your choices, including implementation language, specific algorithms, and parameters. If you can, show experimental or analytical evidence to support your choices.
- Remember, graphs and tables are often the clearest summaries of experiments.

Lastly, part of the documentation is in your code. The code should be well organized, legible, and well commented. A person unfamiliar with the code should be able to read it and maintain it.

3.4 Evaluation Criteria

This exam is partly about programming, and partly about how you approach the problem, present your solution, and convince yourself and the committee that it works.

You will be evaluated based on *all* of the following criteria:

- Correctness, efficiency, scalability.
- Structure, readability, and maintainability of your code
- Quality of documentation, according to Section 3.3

It is important that your work passes a critical threshold with respect to *all* criteria. A super-efficient code that is poorly documented and hardly maintainable will be regarded as a failure. So will a software engineering masterpiece that does not work as specified or that does not perform well. Definitely do not code in assembly or shell scripts!

All work must be your own. This includes code and the thinking behind it. You may not discuss the problem in any way with anyone else. Your only conduit for clarifications is by email to `rfonseca` and `sudderth`. Responses to such questions will be cc'd to the comps mailing list when appropriate.

Good luck and have fun!

References

[1] J.W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM Computer Communication Review*, volume 28, pages 56–67. ACM, 1998.

[2] D.J.C. MacKay. *Information theory, inference, and learning algorithms*. Cambridge Univ Pr, 2003.

[3] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31:1192–1201, October 1988.

A Pseudo-random sequence

Table 1 shows a list of 96 numbers generated using the random number generator from Sec. 2.3, starting with seed 2067261. Your implementation should generate the exact same sequence given this seed.

384717275	2017463455	888985702	1138961335	2001411634	1688969677	1074515293
1188541828	2077102449	366694711	1907424534	448260522	541959578	1236480519
328830814	1184067167	2033402667	343865911	475872100	753283272	1015853439
953755623	952814553	168636592	1744271351	669331060	927782434	360607371
529232563	2081904114	1611383427	604985272	1799881606	1155500400	800602979
1749219598	82656156	1927577930	2011454515	828462531	1833275016	1905310403
1423282804	293742895	2019415459	1484062225	1758739317	1166783511	1457288620
598842305	1634250293	528829321	1747066761	407146696	1031620330	1807404079
884168938	1787987373	965105540	584824989	120937804	1082141766	517654719
766608236	1630224099	1580063467	343911067	1234808992	152763936	1260514187
535763254	174078107	858017135	341298340	272379243	1590285344	344306046
1430770104	1578742469	1764217798	901816857	2043818720	1460293275	1705955009
931665166	1193174685	484635109	2004287539	632181131	1466667008	1455103190
375542294	284896725	1518207912	119683330	1473033718	1086215810	270635523

Table 1: Sequence of 98 pseudo-random numbers generated by the algorithm described in Sec. 2.3 with initial seed of 2067261. Your implementation should generate the exact same sequence given the same seed. (The sequence follows the rows in the table).

B Degree distribution and source block sequence

Table 2 shows a sample of degree and list of sources for a sequence of encoded blocks. You should be able to reproduce this list using your implementation, following Sec. 2.4.

Block seed	d	Source Blocks
166362120	1	98
634813345	2	400 62
177020911	2	49 385
1055302029	2	421 541
1364977754	12	336 109 412 410 463 231 319 564 417 305 313 461
1692838451	8	444 522 416 49 9 199 239 182
915510748	2	370 167
1536644533	2	458 555
980758720	8	236 557 326 25 418 154 230 346
1049939729	2	84 195
464738808	2	138 177
1622156932	4	109 43 446 250
667094411	33	201 291 424 197 401 108 38 85 382 401* 53 430 102 117 454 360 29 363 271 230 63 448 186 206 257 80 10 99 190 224 474 338 351 376
526649093	7	262 239 265 91 527 268 550
877036565	1	271
1891461182	2	19 566
1813567941	4	553 78 160 152
1687591223	6	240 385 542 394 465 539
886846905	9	380 345 290 31 273 79 416 108 288
1912570498	3	129 204 230
473728667	3	326 461 451
1321711281	2	439 181
706125047	2	127 144

Table 2: Given the seed on the left, the process outlined in Sec. 2.4 generates the degree d and the list of integers on the right. The parameters are $K = 571$, $c = 0.1$, $\delta = 0.5$, and the initial random seed $s = 166362120$. Note that we did *not* omit duplicate entries in the list (those marked with a *, so you can know the total number of calls to the random number generator), but you must skip these entries when creating the list on your programs.

C Errata

- (Jan 17h, 2:37pm) In page 3, the way to generate a sample from $\mu(d)$ was incorrectly stated as finding 'the unique bin (degree) for which $M(d) \leq u < M(d+1)$.' It has been corrected to read 'the unique bin for which $M(d-1) \leq u < M(d)$, where $M(0) = 0$'.