Project 5: Capstone Project: LT Codes

Due: 11:59 PM, Dec 08, 2016

Contents

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

[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.

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 project 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.

2 Your Task

Your task is to implement two programs. The first, an encoder, reads a file and generates an infinite stream of blocks encoded by an LT Code. The second, a decoder, reads such a stream until it is possible to reconstruct the original file.

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]

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

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 $t_1 \dots t_N$ 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*:

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

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:

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

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 $M(d-1) \leq u < M(d)$, 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. 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) %
     state = next;
     return next;
}
```

[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.

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 $M-1 = \text{MAX\_RAND}$, 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*, and to be compatible with the reference implementations. Do not make up a different approach to tackle the problem. If you do, you will fail the project. 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 name of a file to encode, the block size, and optionally a random seed. `encode` will be called as follows:

```
$ encode <file> <block-size> [<seed>]
```

Where:

1. **file** is the name of the source file to be encoded.
2. **block-size** is an integer, the size of each encoded block, in bytes.
3. **seed** is an integer, the initial seed for the random number generator.

`encode` will process the file into blocks and then continually stream blocks to `stdout`. We describe the format to stream these blocks in Sec. 2.6. `encode` must be able to handle files whose size is not a multiple of the block size.

`decode` will receive an optional drop rate (from 0 to 100), and will receive a stream of blocks on `stdin`. For any given block, it will drop the block (that is, neglect to process the block and simply move on to reading the next block) with probability given by the drop rate. Once enough blocks have been received to successfully reconstruct the original file, it will write this file to `stdout`. It will be invoked as follows:

```
$ decode [<drop-rate>]
```

You 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 10MiB file. The reference implementations can be found in `/course/cs168/pub/lt`

2.6 Data Format

The remaining aspect that we need to specify is the wire format for the blocks. The data in the blocks are to be written verbatim, and the fields in the block header are to be written in network byte order.

The block format is simple: there is a header which specifies the file size and the block size (in case this is the first block received by the encoder - remember, the encoder doesn't know ahead of time anything about the file it's receiving). It also specifies the seed for the random number generator so that the decoder can reconstruct the list of blocks that this block corresponds to. The data directly follows the header. The format is as follows:

```
Block:
  uint32_t fileSize;
  uint32_t blockSize;
  uint32_t blockSeed;
  char     data[blockSize];
```

Note that `fileSize` may not necessarily be a multiple of `blockSize`. Since all blocks are the same size, this means that the final block may extend beyond the end of the file. In this case, it doesn't matter what data is stored in this part of the block, but it is *vitally important* that it be constant - if the data changes over time, it invalidates the decoding scheme (since the data in the final block may be used to XOR with other blocks). Given that the decoder knows the file size from the block header, it is possible for the decoder to know where the actual file ends even if it does not end on a block boundary. Though it would technically be possible to decode with a scheme that allows for arbitrarily many junk blocks, it would be wasteful, and so we require that, if the file size is not a multiple of the block size, only the final block contain junk data, and it is a multiple, there is no junk data (and thus there are no extra blocks).

3 Handin

3.1 What to Hand In

Hand in your project by typing

```
$ cs168_handin lt
```

from *inside* the directory where your work is located. To reduce clutter, the handin script removes .o files and binary executable files, and runs `make clean` before handing in your assignment. You can handin more than once - the new handin will replace the older one. We should be able to rebuild your programs by running `make`.

4 Grading

4.1 Program - 90%

Most of your grade will be based on the correctness, performance, and style of your implementation. Specifically:

1 **Correctness** - Your encoder and decoder must behave correctly. Given any combination of the TA binaries and your binaries (ie, our encoder and your decoder, your encoder and our decoder, or your encoder and your decoder), you must be able to correctly send and receive a file - the output must identically reproduce the input. Additionally, given the same seed, your encoder should produce exactly the same stream of blocks as our encoder. Given the same stream of blocks, your decoder should decode in exactly the same number of blocks as our decoder.

2 **Performance** - Your programs must be reasonably performant. It must be possible to transfer a 10 MiB file between your binaries in a few seconds on a department machine (given a reasonable block size).

3 **Style** - Your program must be reasonably designed, and your code must be clean and readable.

4.2 README -10%

Please include a README file with your program. Describe your algorithms for encoding and decoding, and justify these design decisions. List any implementation details which you found difficult to get correct, and describe how you accomplished this. Describe briefly the performance of your encoder and decoder. Would it be possible to improve this performance? If so, what would be easy or difficult about this change? List any known bugs, and any ideas about potential fixes.

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.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS168 document by filling out the anonymous feedback form:

`https://piazza.com/class/isqj37mfnyz26r?cid=6`.