15:853: Algorithms in the Real World
Fall 2019

Lecture 8

October 1 2019

Lecturer: Rashmi Vinayak *Scribe: Ksenia Korovina*

In this lecture, we will continue looking at randomized Fountain codes, in particular LT codes and Raptor codes. And next, we will get started with the next module - data compression.

8.1 Fountain and Raptor codes

8.1.1 Recap

Last lecture, we started looking at a class of randomized algorithms called Fountain codes. They are primarily used to recover from erasures. To measure the performance of randomized algorithms, we need to introduce two new metrics:

Reception overhead measures how many symbols in addition to k is needed to decode

Probability of failure to decode (can be a function of the number of coded symbols)

For Reed-Solomon codes, both of these metrics are zero. However, this doesn't mean that this is the code we dream about - this is because Reed-Solomon has high (polynomial) encoding and decoding complexity. Moreover, as with other algorithms we have encountered so far, it has to have the codeword n be fixed in advance of sending the message. However, in practice we want the flexibility of varying n for different messages.
To summarize, we are seeking the following *ideal* properties when introducing randomization and looking at the class of Fountain codes:

1. Source can generate any number of coded symbols (that's why such codes are called "Fountain" codes)
2. Receiver can decode message symbols from any subset with small reception overhead (not many more symbols than k are needed) and with high probability
3. Efficiency - linear encoding and decoding

8.1.2 LT codes

With these objectives in mind, we started looking at the class of LT codes, which are the first practical implementation of Fountain codes. Like LDPC codes, these codes are based on a graphical construction. Given some degree distribution $P_D(d)$ over positive integers,

their encoding procedure is defined as follows: until the desired number of code symbols is generated, pick a degree $d \sim P_D$, pick d distinct message symbols, and produce a new node marked with the coded symbol equal to XOR of selected message symbols, and connect that node to selected message symbols. You can refer to Figure 8.1 below for a representation of this process.

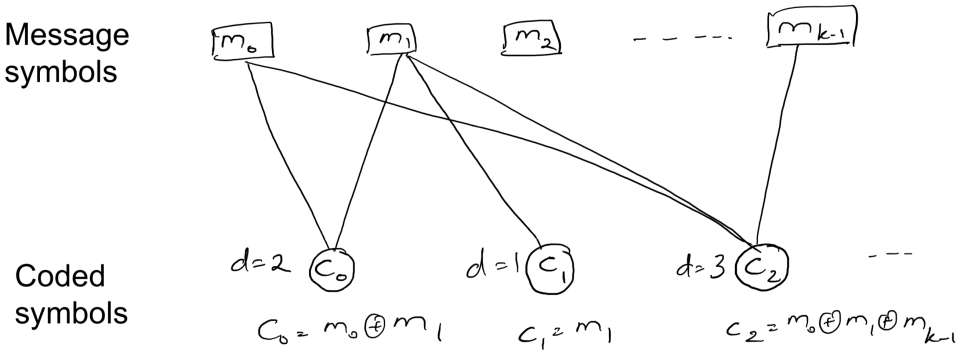

Figure 8.1. Encoding in LT codes

At decoding, we receive a graph from 8.1 with coded symbols filled, and message symbols missing. Decoding proceeds as follows: until end or failure to find a degree 1 coded symbol, select a degree 1 coded symbol c_i, decode the corresponding message symbol m_i (it's just the current value of c_i), update all coded symbols connected to m_i XORing m_i away from them, and remove m_i and c_i and corresponding edges.

As a remark, note that to we don't need to transmit the entire graph into the decoder – it suffices to simply pass the random state and coded symbols c_i, and the graph (with k empty nodes m_i to be filled at decoding) will be generated on the fly.

It is easy to see that the complexity of LT codes is linear in the size of the graph (number of edges). But that is determined by the distribution $P_D(d)$! Moreover, the decoding process outlined above can fail if there is no node of degree 1, probability of which event is also governed by the degree distribution. So, it is crucial to choose this distribution wisely. Here, we will take a look at three examples of distributions and analyze them.

One-by-one degree distribution is a simplest degree distribution possible:

$$P_D(d) = \begin{cases} 1 \text{ if } d = 1, \\ 0 \text{ otherwise} \end{cases}$$

This code is efficient in terms of runtime at encoding, but what about reception overhead? To encode all message symbols, we need to be picking them with replacement equiprobably, until all k are collected. This problem is known as the *Coupon Collector's problem*, and the answer is that on average, we will need $O(k \log k)$ draws – which is a huge reception overhead to have! So we need a better degree distribution. Very important intuition comes from this: *as we decode more, finding an unseen node requires more sampling.*

Ideal soliton distribution fixes the issue of one-by-one distribution by trying to pick larger degrees in order to "collect" message nodes faster:

$$P_D(d) = \begin{cases} \frac{1}{k}, \text{if } d = 1 \\ \frac{1}{d(d-1)}, \text{ if } 1 < d \leq k \end{cases}$$

(It is a basic exercise to check that this is indeed a distribution.) Analyzing this distribution, we would see that if we start with k received symbols, the expected number of symbols we decode (learn) at each decoding stage is 1. The proof of this involves a balls and bins argument, see [Luby(2002)] for full proof. However, this result is not enough for us: expected value being good doesn't guarantee a high probability of successful decoding with low overhead – the actual value may be too "spread". How do we fix this issue?

Robust soliton distribution tries to improve upon the ideal soliton by boosing lower degree nodes, so that we don't end up having too many edges to deal with: it introduces a boosting function $\tau(d)$:

$$P_D^{\text{robust}}(d) = \frac{P_D^{\text{ideal}}(d) + \tau(d)}{\beta}, \text{ where } \beta \text{ is a normalizing constant}$$

The results about the Robust soliton distribution are stated below:

Theorem 8.1. *Under the Robust soliton degree distribution, the LT code has overhead* $O(\ln^2(k/\delta)\sqrt{k})$ *if the probability of failure to decode is bounded by* δ*:*

$$\forall \delta \in (0,1) : \#\textit{coded symbols} = k + O(\ln^2(k/\delta)\sqrt{k}) \Rightarrow P(\textit{failure to decode}) \leq \delta$$

Furthermore, average number of opeerations used for encoding each coded symbol is $O(\ln(\delta))$*, and for decoding each coded symbol is* $O(k \ln(k/\delta))$*.*

So, although the overhead is slightly better than effect i now (by a factor of $\sqrt{k}$), we still didn't achieve the goal of linear encoding/decoding complexity: we would need to have a constant average encoding complexity, but instead we are still stuck with the $\log(k/\delta)$ term, coming in due to the same reasons as $\ln(k)$ in the coupon collector problem. Overcoming this hurdle leads to **Raptor codes**.

8.1.3 Raptor Codes

The previous analysis shows that even the Robust Soliton distribution didn't give us the linear encoding and decoding complexity! Turns out that the key is to go back to the Coupon Collector problem intuition once again: the reason we need so many draws from the bin is that we need to collect **all** coupons. Is there a way to not have to collect all message nodes? Not if we want to decode all message symbols: but perhaps we don't need to!
Here is where the Raptor codes come in: the idea is to first use a **pre-code** (which is an efficient, easy to decode classical code) to encode the message with redundancy, and then apply LT code. So:

$$\text{Raptor code} = \text{pre-code} + \text{LT code}$$

Theorem 8.2 ([Shokrollahi(2004)]). *Raptor codes can generate a stream of coded symbols such that $\forall\epsilon$:*

1. *Any subset of size $k(1+\epsilon)$ is sufficient to decode with high probability:*

$$\#\textit{coded symbols} = O(k(1+\epsilon)) \Rightarrow P(\textit{failure to decode}) \leq \epsilon$$

2. *Encoding per coding symbol takes $O(\ln(1/\epsilon))$.*
3. *Decoding all message symbols takes $O(k\ln(1/\epsilon))$.*

This means we finally arrived at an linear-complexity encoding/decoding scheme with Raptor codes. Due to their efficiency, Raptor codes are widely used in practice.

8.2 Data Compression

Now, we will get started with compression. It is a very important topic with numerous applications in technology:

- File compression: gzip (LZ77), bzip2 (Burrows-Wheeler), BOA (PPM) for files, ARC, PKZIP archivers, NTFS filesystem
- Communication: fax ITU-T Group 3 uses run-length+Huffman), modems (v.42bis protocol, MNP5), virtual connections
- Multimedia: images (gif, jbig), video (Blue-Ray, HDTV, DVD), audio (iTunes, iphone, PlayStation 3)
- Other structures: compression of indexes (google, lycos), meshes for computer graphics (edgebreaker), graphs, databases

Compression model. Like coding, compression involves an encoder and decoder (see Figure 8.2), with encoder receiving a message (data to be compressed), and decoder reconstructs the message. The encoder and decoder should understand common compression format. Interestingly, sometimes you can find compression being referred as coding in the literature, and vice versa.

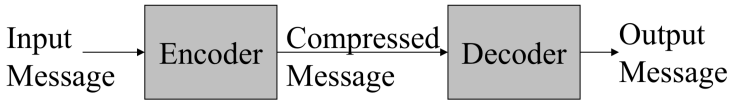

Figure 8.2. General scheme of compression process

With respect to encoder and decoder combination, compression can be categorized into **lossless**, where output message is reconstructed exactly (input = output message), and **lossy**, where the output reconstructs input approximately. Note that the definition of lossy compression doesn't necessarily imply degradation of quality. For instance, compression could lead to image denoising, dropping background noises in music, or fixing spelling correction in text.

Limits of lossless compression. In spite of these possible benefits, however, we are primarily interested in preserving input most of the time. Therefore, is it possible to always compress in a lossless way? The answer is no: in general, to reconstruct exactly the set of all input messages, we need an equally large set of compressed messages, so if one compressed string becomes shorter, another must expand! The way around this limitation lies in approaching the question statistically: if some messages are more likely than others, we will be able to compress. For example, intuitively, we cannot compress outcomes of unbiased coin tosses, but we certainly can compress when the coin lands heads almost all of the times. Therefore:

NEED BIAS IN PROBABILITY DISTRIBUTIONS!

With this statistical viewpoint, we can take a closer look of what the encoder should look like (Figure 8.3). In order to exploit data distribution bias, it will use a model that estimates the probabilities in some way.

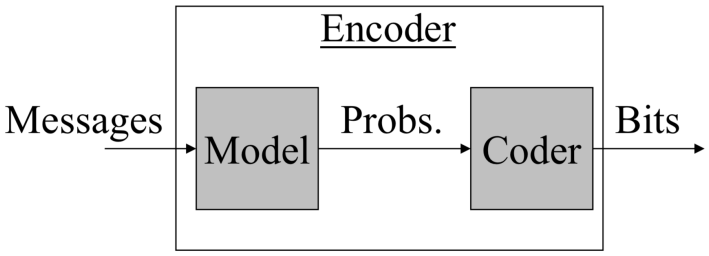

Figure 8.3. General scheme of encoder

The model can be simple (such as character counts or string repetitions), or complex (such as models of human face). Most of currently used compression algorithms employ very simple models, so building complex models, e.g. with neural network estimators, can be a promising way forward for improving compression quality.

Measuring quality of compression. In the analysis of compression algorithms, we can use a few measures of goodness of such algorithms.

1. Compression rate: for lossless compression, this will be largely determined by the quality of the model, by the argument above
2. Running time: as always, we want the algorithm to be fast and efficient

3. Generality: how broad is the class of data on which the compressor can operate
4. Loss metric (for lossy compression only): MSE or other error metrics to measure the quality of input recovery, depending on the kind of data

Benchmarks for compression. For people interested in doing projects on compression, it would be useful to know the benchmarks. There are several standard corpuses to compare algorithms:

1. Calgary corpus
2. The Archive Comparison Test and the Large Text Comprression Benchmark maintain a comparison of a broad set of compression algorithms

We will come back to the question of how much we can compress, after building up some preliminaries. Crucial instruments for analyzing compression are offered by *Information theory*.

8.2.1 Introduction to information theory

Information theory quantifies and investigates the concept of *information*. It aims to establish fundamental limits on representation and transmition of information, such as the minimum number of bits required to represent, communicate and secure data. The framework of information theory was proposed by Claude E. Shannon in a landmark 1948 paper "A Mathematical Theory of Communication" [Shannon(1948)], where he proposed and also solved key questions of the field.

Entropy

The key concept that we will start is that of **entropy**. And informal mnemonic definition of entropy is the measure of information content, or of how much "choice" we have in choosing a value from a specific set of events $S = \{s_1, \ldots, s_n\}$. Intuitively, this measure should not depend on the actual values of s_i, only on the relative probabilities of these events. For example, if only one of them ever happens ($P(s_i) = 1$), then the choice of this value conveys no new information to us; on the other hand, if s_i is very unlikely to happen (for example, an event of having 90 Fahrenheit temperature in October in Pittsburgh...), this conveys a lot of information.
Formally, entropy is defined as mean self-information of messages (events) $s \in S$. **Self-information** of message s is measured in bits and defined as

$$i(s) = \log_2 \frac{1}{p(s)} = -\log_2 p(s)$$

where $p(s)$ is the probability measure defined on events (messages). Then, **entropy** is a weighted (with probabilities) average of self-information and also measured in bits:

$$H(S) = \sum_{s \in S} p(s) \log_2 \frac{1}{p(s)}$$

To reiterate, any "information measure" f should only be a function of probabilities. Listing a few properties that we intuitively would want such a measure to satisfy, we can see that log function on probabilities is the only function that satisfies these properties (additionally, we choose the base 2 to operate in bits):

1. When $p(s)$ is low, its information (self-information) should be high (intuitively, improbable events are "unexpected" convey more information): $p \downarrow \Rightarrow f \uparrow$
2. Information contents of two independent messages should add up:

   $$(f(s_1, s_2), P(s_1 \wedge s_2) = P(s_1)P(s_2)) \Rightarrow f(P(s_1, s_2)) = f(P(s_1)) + f(P(s_2))$$

   Note that in particular, this implies that the information content of a certain to happen event (message) should be zero: $P(s) = 1 \Rightarrow f(P(s)) = 0$

These conditions restrict f to be a constant multiple of log of probability: one can write out the last condition as $f(P_1 P_2) = f(P_1) + f(P_2)$, differentiate both sides w.r.t. P_1, then w.r.t. P_2, and find that the solution of the resulting differential equation takes the form $f(P) = c \log(P)$. This includes logs of different bases, as all logs differ by just a constant multiple. You can find details of this proof in Shannon's paper [Shannon(1948)].

Conditional entropy

A related concept is **conditional entropy**, where in addition to the set of messages S equipped with probabilities we consider the set of possible contexts C, which are also realizations of some random variable. Conditional entropy relates to entropy in the same way as conditional probability relates to probability: it measures the amount of information conveyed by s, given the context s. Formally, we can define **conditional self-information** as

$$i(s|c) = \log_2 \frac{1}{p(s|c)}$$

and **conditional entropy** as

$$H(S|C) = \sum_{c \in C} p(c) \sum_{s \in S} p(s|c) \log_2 \frac{1}{p(s|c)}$$

Next up

Next lecture, we will look at the first examples of practical compression algorithms, starting with Huffman coding and Arithmetic coding.

Bibliography

[Luby(2002)] Michael Luby. Lt codes. *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 271–280, 2002.

[Shokrollahi(2004)] Amin Shokrollahi. Raptor codes. *International Symposium onInformation Theory, 2004. ISIT 2004. Proceedings.*, pages 36–, 2004.

[Shannon(1948)] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.