Another Personal View of Generalized Huffman Coding

Michael B. Baer

November 30, 2013

Abstract

This note is a high-level introduction to my research, written for those who want an overview of this body of work, rather than just an abstract of one particular paper. As such, most mathematical and algorithmic details are omitted. The first part, intended for a general audience, explains the ideas behind lossless coding, specifically Huffman coding and variants thereof. The second part, intended for a more technical audience, summarizes the results of my dissertation, papers, correspondences, and extended abstracts. This note was written in September 2008 and revised in November 2013, its title inspired by that of Mordecai Golin's September 2001 talk, "A Personal View of Generalized Huffman Encoding."

1 Introduction to Data Compression

1.1 Concept

The concept of data is now so ubiquitous that most computer users know what a gigabyte is, if not precisely, then approximately. (A gigabyte is usually defined as eight billion bits, where a bit is a binary digit, that is, a value that can take on either a "0" or a "1". Data is stored both in computer memory and on hard drives in such a binary fashion.) Upon using too many gigabytes, a user will buy a new hard drive, or, perhaps, a new computer. Similarly, upon using too many gigabytes, a company will often have to buy a new and expensive piece of hardware for its computing infrastructure. Upon attempting to transmit too many gigabytes in a short period of time across a communication line, such as a fiber-optic line or a cellular network, communication systems often approach capacity and fail in a variety of ways: by refusing new data, by delaying all data, or by having complete system failure.

Clearly, it is a matter of some importance to household and industry just how these gigabytes are used. Like any other commodity — dollars, oil, time — they can be used wisely or wasted. For some means of communications, like mobile phones, even a millionth of a gigabyte (that is, a kilobyte) here and there can make a fair bit of difference. And, of course, a byte here and there every few

bytes makes a large difference overall. Wherever use of storage or bandwidth is restricted — as it is in most applications — the discipline of data compression enters into play.

The results of data compression are seen in action by millions daily. When copying a music CD at home, we observe that no more than about 80 minutes of music can be stored on the recordable CD. Yet when we burn the same recordable CD with data files obtained from such online sources as iTunes, it holds about 800 minutes. The compression ratio between raw production video storage space and DVD storage space is even more dramatic; both users and engineers often think of storage in terms of space, so it is natural to use this analogy here.

Part of the trick is getting rid of some of the data we don't really need. If it is acceptable to render part of a picture slightly brighter or part of a song slightly louder, we can save some of the space that stores precisely how bright or loud the reproduced media should be. Because some data is lost, this is called *lossy compression*. (This must be distinguished from "lousy compression," which, naturally, is what happens if too much data, too little data, or the wrong data is disposed of.)

Still, dropping some data here or there, however wisely, is not enough to achieve the levels of compression we see in practice, and such loss is unacceptable in transmitting, say, a program or a book. If "It was the best of times" were represented as "It was the vest of dimes" in the compressed file, this would be a problem, unlike a slight change in image brightness. *Lossless compression*, also known as *entropy coding*, is also important, and it is what the bulk of my research has concerned.

Lossless compression may seem like a bit of magic at first. How is it that every time I take a text document and form a ZIP file — ZIP files use lossless compression — I get back a smaller file from which I can recover the larger one? The answer is that not all data are equally likely. For example, let's say that I take a novel and replace every instance of "and" with "&&." The novel will clearly be shorter, "and" being a common word, and since no novel I'm aware of has "&&" in it, turning around and replacing "&&" with "and" gets back the original novel.

Clearly doing this for "and" is far more effective than, say, doing it for a less common word like "fan." Doing it for a more common word, like "the," would be even more effective. The effectiveness of a lossless compress system is thus dependent on a model of the probabilities of the data to be compressed.

1.2 Example

A simple example might, instead of compressing words, compress weather forecasts. Contrary to popular opinion, for example, Los Angeles is not always sunny, but let us say that "sunny" is a good bet 60% of the time. Another 20% will be "partly cloudy," another 10% foggy, with the remaining being 3% mostly cloudy (overcast), 3% showers, 2% rain, 1% thunderstorms, and 1% snow. (In case it's not obvious, these aren't accurate numbers, at least not for the snow.)

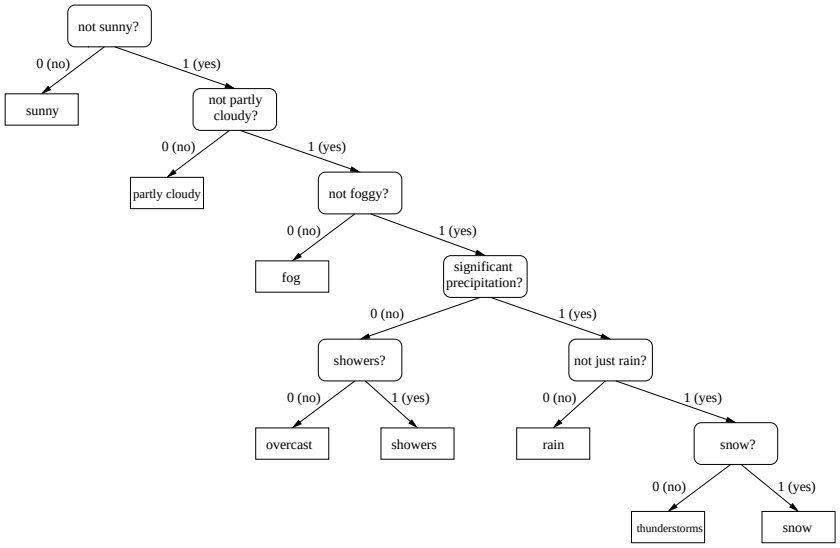

Figure 1: Flowchart / coding tree for compression of weather forecast data

Since there are eight possible forecasts, the most straightforward approach to storing or transmitting this would be to use numbers one through eight, or (all possible combinations of) three bits ("0"s and/or "1"s). However, it would be more efficient to send one bit corresponding to whether ("0") or not ("1") it is sunny; another, if necessary, for whether or not it is partly cloudy; another, if still necessary, for whether or not it is foggy; a bit distinguishing overcast or showers from worse; and another bit (for more likely events) or two (for less likely events) to distinguish the other possibilities from one another. Such a mapping from weather events to bits can be expressed by the flowchart in Figure 1 or, equivalently, described as follows:

event	index i	probability $p(i)$	codeword $c(i)$ (bit sequence)
sunny	1	0.6	0
partly cloudy	2	0.2	$1 \cdot 0$
fog	3	0.1	$1 \cdot 1 \cdot 0$
overcast	4	0.03	$1 \cdot 1 \cdot 1 \cdot 0 \cdot 0$
showers	5	0.03	$1 \cdot 1 \cdot 1 \cdot 0 \cdot 1$
rain	6	0.02	$1 \cdot 1 \cdot 1 \cdot 1 \cdot 0$
thunderstorms	7	0.01	$1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 0$
snow	8	0.01	$1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1$

Figure 1 can be thought of as a binary tree, which is a convenient way of expressing this mapping. Any general mapping from one notation to another is

called a *code* and each individual event mapping a *codeword*. Not all possible codes are feasible, however; if we were to replace the codeword for partly cloudy with a simple "1" then the bit sequence "1·1·0" could represent either two partly cloudy days followed by a sunny day or one day of fog. To avoid such ambiguity, a code should be *uniquely decodable*, which means just what it sounds like it means. The type of uniquely decodable code most commonly used in practice is one in which, looking at the sequence of bits one-by-one, a decoder can tell where a codeword ends when it ends (rather than having to wait for bits of future codewords to determine this). Such an *instantaneous* code occurs only when no codeword is fully contained within an incomplete portion of the beginning, or *prefix*, of another codeword. For this reason these codes are also called *prefix-free codes* (or sometimes, confusingly, *prefix codes*). This property is satisfied by the above code. In fact, most lossless codes used in practice are prefix codes, since there is no advantage to using other types of uniquely decodable codes, and prefix codes are the simplest type of code.

1.3 Optimality

Although simple, this coding procedure is still more complicated than assigning three bits to each of the eight codewords. What is superior about the above code? How do we measure the "goodness" of such a code? The most common way is to find a code with low *expected value*.

Expected value is the average length of a codeword, where the average is weighted by the probabilities of the events in question. Consider, for example, a coin flip, which has only two equally probable results. Clearly we should use "0" for heads and "1" for tails. Let us say, however, that we instead use a less efficient code which has "0 · 1" for heads and "1" for tails. (This is actually less silly than it sounds; such *"1"-ended codes* are an actual area of study.) If we flip the coin three times, we'll have anywhere from 3 to 6 bits representing the results. In the most likely instances, we'll have 4 or 5 bits, and the average number of bits per symbol will be about 1.33 or 1.67. There is some sense, however, in which we should determine an average according to not any particular set of outcomes, but according to a weighting over the likelihood of the events. Since we have half a chance of 1 bits and half a chance of 2 bits this "average" would be 1.5 bits, 1.5 being equal to $(0.5)(1) + (0.5)(2)$. Such a value is called the *expected value*.

We use $p(1)$ to represent the probability of the event represented by index 1 (and $p(2)$ for 2, etc.) and $l(1)$ to represent the number of bits in the corresponding codeword $c(1)$ (and $l(2)$ for $c(2)$, etc.). In the weather case, then, the expected length is

$$\begin{aligned} p(1)l(1) + p(2)l(2) + p(2)l(2) + p(3)l(3) + p(4)l(4) + \\ p(5)l(5) + p(6)l(6) + p(7)l(7) + p(8)l(8). \end{aligned} \tag{1}$$

This sum of products is 1.82 for the variable-length code whereas if $l(i) = 3$ for all of the 8 values of i, then the expected value for length would be 3. Judged on this basis, the more complicated code is superior.

More generally, if n represents the number of events to be coded then the expected value is

$$\sum_i p(i)l(i) \qquad \text{or} \qquad p(1)l(1) + p(2)l(2) + \ldots.$$

Here $\sum_i$ represents the sum of the terms for all possible i (in this case, from 1 to n). A more informative but more complicated way of expressing this is

$$\sum_{i=1}^{n} p(i)l(i) \qquad \text{or} \qquad p(1)l(1) + p(2)l(2) + \ldots + p(n)l(n)$$

which makes the starting (index 1) and ending (index n) points explicit. For example, if $n = 8$, this expression has the same meaning as the expected value summation on the line labeled with (1). This weighted average is often denoted with the less bulky $\mathbb{E}_p[l(i)]$.

The goal of compression is to minimize the total number of bits expended, so we might reasonably ask what relationship this has with expected value. The short answer to this is that it is a close enough relationship that we generally want to find a code that minimizes expected value in order to minimize bits expended for most event combinations. Unfortunately, the exact relationship is rather technical, so you might want to skip to the paragraph after the next one if you get lost in the mathematics. Consider again a weather report, where the weather on day 1, with the probabilities above, is represented by X_1, day 2 by X_2, etc. The total number of bits expended for encoding m days of weather into bits is $\sum_{j=1}^{m} l(X_j)$ (i.e., $l(X_1)+l(X_2)+\ldots+l(X_m)$). The average number of bits per event is thus $\frac{1}{m}\sum_{j=1}^{m} l(X_j)$ (i.e., $(l(X_1) + l(X_2) + \ldots + l(X_m))/m$), which can be denoted $\mathbb{A}[l(X_1^m)]$ (although this terminology for a running average is not a widespread one like "$\mathbb{E}$" is for expected value). For example, if lengths are $l(X_1) = 1$, $l(X_2) = 4$, $l(X_3) = 4$, $l(X_4) = 3$, and $l(X_5) = 4$, then $\mathbb{A}[l(X_1^1)] = 1$, $\mathbb{A}[l(X_1^2)] = 2.5$, $\mathbb{A}[l(X_1^3)] = 3$, $\mathbb{A}[l(X_1^4)] = 3$, and $\mathbb{A}[l(X_1^5)] = 3.2$. Such a sequence of $\mathbb{A}$s may or may not converge, that is, there may or may not be a value a for which $\lim_{m\to\infty} \mathbb{A}[l(X_1^m)] = a$, where this limit (lim) expression implies that, informally, $\mathbb{A}[l(X_1^m)]$ gets closer and closer to a as m gets larger and larger.

The probabilistic relationship between the limit (if any) of this average and the expected value is given by the *strong law of large numbers*:

$$\mathbb{P}\left(\lim_{m\to\infty} \mathbb{A}[l(X_1^m)] = \mathbb{E}_p[l(i)]\right) = 1$$

where the $\mathbb{P}$ denotes probability, specifically the probability that the limit of the average length equals the expected length. What this means, in simple English, is as follows: We would like it if an arbitrary data sequence has the property that the average length of an event in the sequence will get as close as we like to the expected length; it is just a matter of how far into the sequence we have to go. The strong law of large numbers says that the probability our sequence will be such a well-behaved sequence is equal to one; that is, it is a certainty.

This guarantee may not be much of a guarantee if the required "close enough" point in the actual sequence is past the end of any practical data we'll ever see. Informally, we might speak of "having enough data for the law of large numbers to kick in." This is determined, to a first order approximation, by the central limit theorem, but it suffices to know that, in practical applications such as media files and office documents, it is safe to assume that the law of large numbers has "kicked in." It is so often assumed that expected length and average length are "close enough," that, confusingly, both are sometimes called "mean length," a term that is wise to avoid for the sake of semantics.

The code we used for weather was produced via a method that always finds the most efficient way of representing items given their probabilities, i.e., the one having the lowest expected length. Any code produced via this method is called a "Huffman code" after the person who discovered how to generate such codes for any given probability distribution over a finite number of items.

This does not necessarily mean that the best compression is achieved by using this code. As presented here, this method ignores statistical dependency among events. For example, if yesterday was rainy, the probability that today is sunny is less than it might otherwise be. These dependencies can easily be accounted for, but accounting for them does make more work for the computer (and for its programmer). Also, we've assumed that our probabilities are not only unchanging but also known and accurate, which is often not the case in practice. Finally, if we take the combined probability of two days' of weather, rather than just one, using the probabilities of the two-day possibilities to come up with a new Huffman code corresponding to two days at a time, that code will be a more efficient representation than taking each separately, resulting, in this example, in 1.8187 expected bits rather than 1.82 bits. Information theory states that, with further grouping — say of three, four, or more symbols — compression systems can get as close as possible to the fundamental coding limit, known as *entropy*, which, in this case, is about 1.7880456... bits. A method known as arithmetic coding does this grouping implicitly, however, and is thus often a less complex method of approaching entropy. Still, many compression systems choose not to use arithmetic coding, as it is more complex than most implementations of Huffman coding and thus runs more slowly and/or requires better computational devices. Other drawbacks include poor performance with certain inaccurate probability models, difficulty in programming and debugging, reduced error recovery, and patent restrictions for certain implementations. In such systems, these drawbacks are not worth putting up with for improvements in compression, which are often small. (At this point, most relevant patents on arithmetic coding have expired. However, their existence discouraged use in software, standards, and other technology still in use today. In addition, it is often difficult to determine which patents cover which forms of arithmetic coding, and thus often easier to just use the unpatented Huffman method.)

1.4 Variants of the Problem

In other cases, though, alternatives to Huffman coding are used. In fact, even when a compression method such as JPEG or MPEG uses "Huffman codes," these codes are not necessarily generated using Huffman's technique, but are rather convenient approximations of true Huffman codes. (Technically, they are fixed-to-variable-length codes, not Huffman codes.) Nevertheless, Huffman codes are often used in practice and are useful not only in the practice of compression, but in its theory as well.

It is, however, worth looking at why certain applications do not use actual Huffman codes when assigning bit sequences to events. Before describing variants of Huffman coding, I would like to point out that hundreds of papers on variants of Huffman coding are discussed in a survey by Julia Abrahams entitled, "Code and Parse Trees for Lossless Source Encoding," originally published in the *Proceedings of the Compression and Complexity of Sequences* in 1997 and updated for *Communications in Information and Systems* in 2001. (It was also excerpted as "Huffman Code Trees and Variants" in an online preprint.) This survey is comprehensive, whereas the paper you are now reading focuses on my areas of interest, which are described in the following.

Many times, an optimal code has events that are so unlikely that the corresponding Huffman codeword lengths are inconveniently large, making computation difficult due to the use of especially long codewords for such events. It is often preferable to have a suboptimal code that is length-limited, that is, has a maximum length. In fact, standards such as MPEG not only use a maximum length for even the most unlikely events, but they also limit the number of codeword lengths they use within an allowed range of codeword lengths. This is also for computational reasons, although, curiously, such problems have not previously been looked at in depth. Other codes cannot have a maximum length because they represent data such as integers, which are infinite in number. Although the Huffman algorithm cannot find an optimal code with infinite input, optimal and near-optimal codes exist for such problems.

As previously stated, the objective of minimizing expected length, with or without restrictions, rests on the assumption that the quantity of data is so great that the law of large numbers guarantees that average length will be close to expected length, and thus that is the value that is best to minimize. When dealing with fiber optic cables and gigabit Ethernet, that is a safe assumption. However, in other coding situations, resources may be low enough that this assumption no longer holds.

A low bit rate might result in needing a buffer of some sort, in which data yet to be transmitted must temporarily reside. If such a buffer is small enough, data might be lost and we would want to minimize the probability of this occurring. If the transmission rate is slow enough, second order effects might mean that a long codeword for an unlikely event unduly impacts the transmission of all codewords. (This phenomenon is called the "slow truck effect," since the long codeword is like a slow truck on a highway delaying vehicles [codeword data] that would ordinarily advance more quickly.) In such cases, longer codewords

should be shorter than they would otherwise be (meaning that, as a trade-off, some shorter codewords would have to be longer). If a connection is unreliable, on the other hand, we may want the most likely events which already have short codewords to have even shorter codewords than usual, so that a message has a higher chance of being received before the signal is lost. We could also imagine a measurement for success in which some combination of these factors is used. All these problems have different approaches and solutions.

Up until this point, the discussion has been limited to problems where the possible events coded have no particular order. This, however, is not always the case. A Huffman code can be seen as narrowing of a number of possibilities by separating them, at each step, with each bit, into two smaller subsets, continuing this subdivision until only one item remains within the realm of possibility. Often, if items are ordered, we won't be able to arbitrarily subdivide, but rather our subdivisions will be restricted to less-than/greater-than-or-equal-to divisions. This so-called *alphabetic* restriction, much like looking for a word through a dictionary, means that the solution may be less efficient but more practical than that obtained by ignoring the restriction.

Finally, there are some applications for which the goal is not to encode data into "0"s and "1"s. For many storage and communications methods, from flash memory to wireless, there are far more values to choose from to represent data. Therefore, nonbinary coding problems (i.e., those with more than two choices per output symbol) are often also useful to solve, though such problems are generally in the minority of cases considered and are often (but not always) straightforward extensions of binary cases.

2 Summary of papers

My work has concerned various instances and combinations of these situations, finding algorithms and properties relevant to the instances under consideration. My descriptions of the papers and other manuscripts will be partly chronological and partly organized by subject according to the order of exposition given in my dissertation; I will provide their titles in boldface so that this section can be easily skimmed. All papers that I have published — as well as preprints for papers yet to complete a peer review process — can be found on my web site, `http://www.mbbaer.com/`. A chart summarizing the nature of these papers is at Table 1. For papers written by others, I will use a mangled version of MLA style (*MLA Style Manual and Guide to Scholarly Publishing*, 1998), sparely using in-place references. More complete reference lists can be found within my papers. Because I have the benefit of hindsight, some of the interpretations and examples given here are not necessarily in the published papers; all fundamental results, however, are.

A quick note before beginning: As mentioned before, there are methods that are superior to the Huffman (prefix) coding framework in many settings. This is so much the case that two decades ago there was a publication whose title asked, "Is Huffman Coding Dead?" (A Bookstein and ST Klein, "Is Huffman Coding

Dead?" *Computing*, 1993.) This paper weighed the benefits and drawbacks of Huffman coding against other methods of compression, finding that there were still several reasons to prefer Huffman coding. As recently as 2007, a publication looked at which compression method was best for the ARM7TDMI processor, that used in the iPod (classic) and in most Nokia mobile phones, in the end favoring Huffman coding (Garofalo, Napoli, Petra, Strollo, "Code Compression for ARM7 Embedded Systems," *ECCTD 2007*). Finally, even if Huffman coding were strictly inferior for every application, many technologies in wide use today — such as (baseline) JPEG for digital cameras — are stuck using prefix coding (with "Huffman tables" for decoding) for the foreseeable future; no other technology has challenged JPEG as a *de facto* standard for digital photography.

2.1 Generalized Huffman Coding

My dissertation, ***Coding for General Penalties***, contains a number of results, some of them published, others not novel enough to merit publication on their own, but nonetheless worthwhile. The second chapter of the dissertation concerns prefix coding for *exponential objectives* — those of the form

$$\mathbb{E}_p[a^{l(i)}] \quad \text{or} \quad \sum_i p(i)a^{l(i)} \quad \text{or} \quad p(1)a^{l(1)} + p(2)a^{l(2)} + \ldots$$

which had been previously studied, both in theory and application, for arbitrary $a > 1$. For a between 0 and 1, this objective is one to maximize, not minimize, but still has similar properties and solutions. (The objective can be rephrased as $\log_a \mathbb{E}_p[a^{l(i)}]$ to be a minimization for both subproblems.) Properties of traditional Huffman codes are extended to these exponential expectation objectives, including a property of the corresponding coding tree (known as the sibling property) and one regarding how to break ties among optimal codes for various tie-breaking objectives. In addition, continuity properties of the objective for various values of a — where $a = 1$ is used to denote Huffman coding — are examined.

Much of the material from Chapter 3 forms my first journal publication, "**A General Framework for Codes Involving Redundancy Minimization**." This correspondence serves to unify a number of problems that Huffman-like methods solve, including one which had not previously been solved in linear time or using a Huffman-like method. The problem of finding the minimum maximum pointwise redundancy code (which is defined below) was previously solved using a variant of what is known as Shannon coding. Shannon coding was one of the first prefix coding methodologies developed, preceding the discovery of the more well-known Huffman coding algorithm. It uses what is known as *self-information*, the codeword length that would be algebraically calculated for an event if codeword length lengths were not restricted to whole numbers, equal to $-\log_2 p(i)$ for event i. This is rounded up to determine a codeword exceeding self-information by less than one bit. It is easy to construct a code with the desired codeword lengths. Being mathematical in nature, this coding

method is very different than the more algorithmic Huffman coding, although both methods guarantee codes within one bit of entropy. (This one-bit property is known as a *redundancy bound*.) The difference between average codeword length and entropy is known as expected redundancy (or sometimes mean redundancy or just redundancy). Because expectation is a linear operation, the expected redundancy is the expected value of the *pointwise redundancy*, the difference between codeword length and self-information (that is, $l(i) + \log_2 p(i)$). In any code for a finite number of events, there is a maximum pointwise redundancy, as one or more events will have a greater pointwise redundancy than others.

While the Huffman code, being optimal, never has an expected codeword length greater than that of the Shannon code, individual Shannon codeword lengths can be shorter than their counterparts in a Huffman code, which does not have the Shannon code's per-event (pointwise) guarantee about lengths and self-information. The earliest proposed method used to find a code that minimizes maximum pointwise redundancy used a generalized Shannon method, and the resulting code can have no codeword that is longer than its Shannon coding counterpart, since any such code would have longer maximum pointwise redundancy than the Shannon code. I proposed a Huffman-like method for minimizing maximum pointwise redundancy, which, in turn, can find a code with no codeword length longer than its generalized Shannon code counterpart. This might be a different code; multiple codes can have identical maximum pointwise redundancy. Codes obtained via this Huffman-like method are thus improvements on generalized Shannon codes, minimizing other desirable factors among codes optimal in minimizing maximum pointwise redundancy. Thus this modified Huffman code, unlike the Huffman codes used in previous literature, is a strict improvement on Shannon coding. In fact, this method and the traditional Huffman method are just opposite ends of a continuum of coding methods related to a modified exponential penalty, one previously known but not previously related to Shannon coding, in either its original or generalized form. This ties together not only Huffman and Shannon coding, but also provides a greater two-dimensional framework with limited symmetry encompassing families of problems previously looked upon distinctly.

These Huffman-like methods cannot be directly applied to probability mass functions with a (countably) infinite number of events, but, as with traditional Huffman coding, the properties inherent in the methods can be used to find optimal codes for certain classes of infinite-event probability mass functions. Methods of doing this are explored in my 2008 journal publication, "**Optimal Prefix Codes for Infinite Alphabets with Nonlinear Costs**." This paper — expanding on my conference presentation, "**Infinite-Alphabet Prefix Codes Optimal for β-Exponential Penalties**" — covers minimization of both the exponential and maximum pointwise redundancy utilities, for both geometric probability distributions (those having $p(i)$ proportional to θ^i for some θ on all positive numbers) and rapidly declining distributions such as the Poisson distribution. The paper further considers properties and applications of these codes, one of which — the "siege problem," which relates to the probability of

successful communications under adverse conditions — was also introduced as the first portion of "**Rényi to Rényi — Source Coding under Siege**." My initial queries into the topic of coding for infinite-event alphabets are contained in Chapter 5 of my dissertation. This chapter also mentions two-dimensional geometric distributions, in which geometrically distributed events are paired and the traditional objective of expected codeword length is considered. As explained in Subsection 1.1, pairing often decreases expected codeword length. In the section of the dissertation discussing this, I describe the code that is, in some sense, "approached" by the codes optimal for the two-dimensional distribution family with increasing likelihood of the most likely event. More definitive results are contained in a recent paper (F Bassino, J Clément, G Seroussi, A Viola, "Optimal Prefix Codes for Pairs of Geometrically Distributed Random Variables", *IEEE Transactions on Information Theory*, 2013), including a formal proof of this observation.

The aforementioned "**Rényi to Rényi — Source Coding under Siege**" considers aspects of exponential objectives for a finite number of items. I present optimization and approximation algorithms for the alphabetic version of this problem, which, perhaps due to its lack of amenability to common fast algorithms, was not previously considered for $a < 1$; these by themselves form the brief "**Alphabetic Coding with Exponential Costs**," a short version of my (unreleased) "**Alphabetic Coding under Siege**" (available upon request). I also present redundancy bounds that improve upon the trivial aforementioned 1-bit bounds; these are tied to the probability of the most likely codeword. I present corresponding results for minimum maximum pointwise redundancy coding in the extended abstract "**Tight Bounds on Minimum Maximum Pointwise Redundancy**," and the redundancy bound results of the aforementioned papers are combined and extended in my 2011 journal publication, "**Redundancy-Related Bounds on Generalized Huffman Codes**." These extensions apply to exponential objectives; further improvements to these bounds on exponential objectives are presented in "**On the Redundancy of Huffman Codes with Exponential Objectives**."

2.2 Quasiarithmetic Objectives

My second journal publication, "**Source Coding for Quasiarithmetic Penalties**," contains matter present in both Chapter 4 of my dissertation and my first conference publication in this field, "**Source Coding for General Penalties**." This work encompasses a broader array of coding problems, although, like the exponential objective, the broader *quasiarithmetic* family considered here was first introduced without optimal solution by Campbell (LL Campbell, "A Coding Problem and Rényi's Entropy," *Information and Control*, 1965). While the exponential objectives are of the form

$$\log_a \mathbb{E}_p[a^{l(i)}]$$

or

$$\mathbb{E}_p[a^{l(i)}]$$

quasiarithmetic objectives are those of the form

$$\varphi^{-1}\mathbb{E}_p[\varphi(l(i))]$$

or

$$\mathbb{E}_p[\varphi(l(i))] \tag{2}$$

for monotonic functions φ. This is a generalization of not only the exponential objectives, but also the traditional expected length objective, and, if φ is considered infinity for forbidden lengths, the length-limited objective (which is that of expectation over allowed lengths). It is also seen in *minimum delay coding*, in which end-to-end latency (delay) is minimized rather than codeword length (throughput) assuming an M/G/1 queueing model (LL Larmore, "Minimum Delay Coding," *SICOMP*, 1989). "Source Coding" presents an approach for optimizing coding over any finite probability mass function for any quasiarithmetic objective with convex increasing φ, which, in particular, results in an improvement in the speed of solving Larmore's minimum delay coding problems. Complementary to the algorithmic contribution are those involving properties relating to these objectives, including redundancy bounds (using a form of generalized entropy) and the existence of optimal codes (which is not a trivial matter for infinite alphabets).

One statistic of codes — which I did not use as the basis of an objective in this paper, but which nonetheless makes for an interesting digression — is that of data expansion (JF Cheng, S Dolinar, M Effros, R McEliece, "Data expansion with Huffman codes," *ISIT*, 1995). Data expansion occurs where statistics are perfectly known and uncompressed data are replaced, event-by-event, by their compressed forms. Uncompressed data take up m bits per event, where $m = \lceil \log_2 n \rceil$, n being the number of events considered and $\lceil x \rceil$ denoting the smallest integer not less than x. These data are replaced by data taking up $l(i)$ bits per event i. Thus, if all events where $l(i)$ exceeds m occur prior to the other events, the file temporarily expands by up to $\mathbb{E}_p[(l(i) - m)^+]$ bits per symbol, where x^+ denotes the maximum of x and 0. Clearly this is a penalty of the form (2), and its minimization is achieved by a simple fixed-length code. It is not a quantity one would usually want to minimize alone, but an application might trade off this measure with the more traditional measure of expected length, seeking to minimize

$$\mathbb{E}_p\left[l(i) + (l(i) - m)^+\lambda\right]$$

for some fixed $\lambda > 0$. Such a formulation linearly trades off the two quantities; Huffman coding corresponds to $\lambda = 0$ and fixed-length coding to $\lambda = \infty$. This can be used to minimize one quantity with respect to a constraint on the other using a binary search over values of λ. (Although it might first appear otherwise, such an approach is not suited to Lagrangian methods for finding optimal solutions, since the problem is an integer problem, not a real-valued problem.) One could also minimize a variety of nonlinear hybrid coding objectives using convex hull techniques discussed at the end of Chapter 4 of my dissertation.

2.3 Linear Objectives

Although the aforementioned objectives are of some interest, in the vast majority of situations, bit rate is high enough that coding problems should solve the linear objective of expected throughput. This is usually equal to expected codeword length, but there are situations in which that might not be so. For example, one of the oldest well-known codes, Morse code, uses output symbols of different lengths; a "dot" is shorter than a "dash." Prefix coding with different symbol costs is thus an old problem, one for which many variants and solutions exist. The unrestricted problem actually has no known efficient solution, while adding an alphabetic restriction on the input and output results in a problem that is easily solved via dynamic programming. The alphabetically restricted problem has an analogue in binary decision trees, in which one wishes to distinguish among several outcome events based on less-than/greater-than-or-equal-to queries, and duration of the query is dependent upon the outcome. If this duration asymmetry is fixed — one duration of time for the less-than outcome and another for the greater-than-or-equal-to outcome — this is one of the already-studied problems. However, if the problem-solver can control the asymmetry of the duration — whether or not the less-than outcome takes more time than the greater-than-or-equal-to outcome — then this results in a problem that is examined in "**On Conditional Branches in Optimal Decision Trees**." The properties and solution must be modified, but are similar in flavor to those for the fixed-asymmetry case. Practically speaking, this problem arises in programming such decision trees into software, in which such asymmetries are due to branch prediction. In simpler processors, such as those for all but the most advanced mobile phones, branch prediction follows a simple model. The model is more complex for personal computers and advanced mobile devices, and the analysis must be altered accordingly. This is more fully considered, along with other variants of the problem, in the full version of the paper, "**On Conditional Branches in Optimal Search Trees**."

One sample application of such decision trees is, coincidentally, determining the codeword length of a Huffman code in the process of being decoded. The efficiency of this step is critical in determining the speed of decompression, yet in the past it has been accomplished without considering the aforementioned symmetry in branch outcome (A Moffat and A Turpin, "On the Implementation of Minimum Redundancy Prefix Codes," *IEEE Transactions on Communications*, 1997).

As speed of determining codeword length of the code being used is critical in the speed of overall decompression, Moffat and Turpin indicate that restricting the code to those with no codewords longer than a maximum length results in faster decompression. In fact, restricting the length of the minimum codeword length, as well as the maximum, helps in this. This bounded length problem and related problems are considered in "***D*-ary Bounded-Length Huffman Coding**." The full version of this paper is entitled, "**Twenty (or so) Questions: *D*-ary Length-Bounded Prefix Coding**," inspired by the guessing game device, 20Q, which asks between 20 and 25 questions in order to deter-

mine the item being guessed (which, in this case, is the outcome "event"). The answers to 20Q's questions are not restricted to "yes" and "no," and I also consider nonbinary variants of this problem; previous efficient methods of finding codes with length restrictions only apply directly to the binary case.

The aforementioned 1997 survey by Abrahams (as well as the updated 2001 version) mentions the problem of trying to find a code with restricted *fringe*, or difference between the minimum and maximum codeword length. Clearly this problem is related to the bounded-length problem, and a simple reduction to the bounded-length problem results in an efficient solution to the fringe-limited problem, unlike the solution suggested in the survey, which is not polynomial time (and thus not practical for any but the smallest of problems and those for which fringe would be naturally limited due to the nature of the input probability mass function).

The restriction of lengths to values in an interval is a special case of what I've heard of referred to as "**Length-Reserved Prefix Coding**," the title of my paper about the same. This problem restricts codeword lengths to a input set of codeword lengths. This set may be infinite, although I show in the paper that any infinite set can easily be made into a finite set with the same solution. (This problem was referred to as the "length-reserved" problem by Zhen Zhang, the idea being that forbidden lengths that were deemed "reserved.") I addressed this problem using dynamic programming in a manner that is formulated quite differently from that used for the aforementioned branch problem, one which more closely, though not completely, resembles that used for "1"-ended codes (SL Chan and MJ Golin, "A Dynamic Programming Algorithm for Constructing Optimal "1"-Ended Binary Prefix-Free Codes," *IEEE Transactions on Information Theory*, 2000). Golin, in turn, sped up this and related algorithms by an order of magnitude (M Golin, X Xu, J Yu, "A Generic Top-Down Dynamic-Programming Approach to Prefix-Free Coding," *SODA*, 2009).

This dynamic programming method can be adapted to solve other problems. For example, in Subsection 2.2, I noted that I'd found a way to find an optimal solution for a quasiarithmetic utility, one of the form (2), provided that φ was convex. However, certain previously unsolved variants of the siege problem are not convex. The dynamic programming method here can be easily adapted to solving such problems with nonconvex, monotonic φ.

This method also solves problems involving a code restricted not to having only certain fixed codeword lengths, but to having only a certain number of user-determined codeword lengths — e.g., four different codeword lengths, such as 4, 5, 8, and 9, or 3, 5, 7, and 10. This simple extension is practical becauce there are, relatively speaking, only a small number of candidate sets of codeword lengths. Although the method for *finding* such a code is slower than conventional Huffman coding, the resulting code results in less computation *for decompression* than conventional Huffman coding. Just as Huffman coding is often chosen because it is faster than the oft-better arithmetic coding but slower than no compression, this method can be chosen for cases in which even a conventional Huffman code is too slow to use but some compression is desired. I also presented fast suboptimal methods, as well as fast methods to find optimal

codes if only two or three codeword lengths are allowed; many of these can be solved even more quickly than Huffman coding. The problem of compression performance with two codeword lengths was recently considered on an asymptotic basis (E Figueroa and C Houdré, "On the Asymptotic Redundancy of Lossless Block Coding with Two Codeword Lengths," *IEEE Transactions on Information Theory*, 2005), though optimizing the problem was not previously considered to my knowledge.

So far I have been discussing problems for which an optimal solution is known. However, several infinite-event cases do not have a known solution. These include power laws, in which probability is inversely related to a power of the item index, i.e., $p(i) \sim ci^{-\alpha}$ for constants $c > 0$ and $\alpha > 1$, where $p(i)$ is the probability of symbol i, and $f(i) \sim g(i)$ implies that the ratio of the two functions goes to 1 with increasing i. Such power laws are well known to model several natural processes (M Mitzenmacher, "A Brief History of Generative Models for Power Law and Lognormal Distributions," *Internet Mathematics*, 2004; MEJ Newman, "Power Laws, Pareto Distributions and Zipf's Law," *Contemporary Physics*, 2005; NN Taleb, *The Black Swan: The Impact of the Highly Improbable*, 2007). (As a side note, the last of these works was considered prescient in criticizing the flawed mathematical models which justified the financial decisions leading to the recent financial crisis. These models greatly underestimated the probability of large deviations; models involving power laws generally wouldn't.) If no optimal code is known, we must settle for a nearly optimal code. In "**Prefix Codes for Power Laws with Countable Support**," the full version of "**Prefix Codes for Power Laws**," I propose a new family of prefix codes and compare their performance with that of previously proposed codes over several common probability mass functions. I find that the new codes are an improvement on the prior codes in most of the considered cases, including one relating to the representation of rational numbers in computer memory.

An area related to prefix coding is *Tunstall coding*: Although not as well known as Huffman's optimal fixed-to-variable-length coding method, the optimal variable-to-fixed-length coding technique proposed by Tunstall offers an alternative method of lossless data compression. Whereas fixed-to-variable length coding represents one event or symbol (or a fixed number of events or symbols) by a variable-length code, variable-to-fixed-length coding represents a variable number of events by a fixed-length code. Among other advantages, this means that a single error in an output symbol will not be propagated beyond the variable-length input group. Constructing a Tunstall code can be realized quite efficiently with Bernoulli input, that is, fixed-probability, independent zeros and ones — two possible event outcomes per input symbol (J Kieffer, "Fast Generation of Tunstall Codes," *ISIT*, 2007; YA Reznik, AV Anisimov, "Enumerative Encoding/Decoding of Variable-to-Fixed-Length Codes for Memoryless Sources," **Ninth International Symposium on Communication Theory and Applications**, 2007). However, prior to my work, it seems that no one examined how to more efficiently construct Tunstall codes for other types of data, which constitute the vast majority of data to be coded. This is what

	Paper	Infinite input alphabet	Finite input alphabet	Linear objectives	Exponential objectives	Quasiarithmetic objectives	Analytic	Algorithmic	Nonalphabetical output	Alphabetical output
Journal	"A General Framework..."		•		•		•	•	•	
	"Source Coding for..."	•	•		•	•	•	•	•	
	"...Infinite Alphabets..."	•			•		•	•	•	•
	"Alphabetic Coding..."		•		•		•	•		•
	"Redundancy-Related Bounds..."		•		•		•		•	
Conference	"Rényi to Rényi..."		•		•		•	•	•	•
	"...Conditional Branches..."		•	•			•	•		•
	"...D-ary..."		•	•	•	•		•	•	
	"Reserved-length..."		•	•		•		•	•	
	"...Power Laws"	•		•			•		•	•
	"Tight Bounds..."		•		•		•		•	
	"...Tunstall Code..."		•	•				•	•	
	"...Redundancy of Huffman..."		•		•		•		•	

Table 1: Nature of my published papers

I do in "**Efficient Implementation of the Generalized Tunstall Code Generation Algorithm**," where I propose an efficient realization of Tunstall's algorithm, requiring time proportional to the number of output symbols. This is especially useful for inputs that are independent, but perhaps not binary or not always the same probability. Tunstall's algorithm is not optimal for symbols and events that are not independent; however, the behavior of using Tunstall codes for such data has been shown to be well behaved (SA Savari and RG Gallager, "Generalized Tunstall Codes for Sources with Memory," *IEEE Transactions on Information Theory*, 1997).