Bits, Signals, and Packets

An Introduction to Digital Communications & Networks

M.I.T. 6.02 Lecture Notes

Hari Balakrishnan
Christopher J. Terman
George C. Verghese

M.I.T. Department of EECS

Last update: December 2011

Contents

List of Figures **7**

List of Tables **9**

1 Introduction **1**
1.1 Themes . . . 3
1.2 Outline and Plan . . . 5

2 Information, Entropy, and the Motivation for Source Codes **7**
2.1 Information and Entropy . . . 7
2.2 Source Codes . . . 11
2.3 How Much Compression Is Possible? . . . 13
2.4 Why Compression? . . . 15

3 Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW) **19**
3.1 Properties of Good Source Codes . . . 19
3.2 Huffman Codes . . . 21
3.3 LZW: An Adaptive Variable-length Source Code . . . 25

4 Why Digital? Communication Abstractions and Digital Signaling **33**
4.1 Sources of Data . . . 33
4.2 Why Digital? . . . 34
4.3 Digital Signaling: Mapping Bits to Signals . . . 36
4.4 Clock and Data Recovery . . . 38
4.5 Line Coding with 8b/10b . . . 41
4.6 Communication Abstractions . . . 42

5 Coping with Bit Errors using Error Correction Codes **45**
5.1 Bit Errors . . . 46
5.2 The Simplest Code: Replication . . . 47
5.3 Embeddings and Hamming Distance . . . 48
5.4 Linear Block Codes and Parity Calculations . . . 51
5.5 Protecting Longer Messages with SEC Codes . . . 59

6 Convolutional Codes: Construction and Encoding **65**
6.1 Convolutional Code Construction 65
6.2 Parity Equations 67
6.3 Two Views of the Convolutional Encoder 68
6.4 The Decoding Problem 70
6.5 The Trellis 72

7 Viterbi Decoding of Convolutional Codes **75**
7.1 The Problem 75
7.2 The Viterbi Decoder 77
7.3 Soft Decision Decoding 79
7.4 Achieving Higher and Finer-Grained Rates: Puncturing 80
7.5 Performance Issues 81
7.6 Summary 83

List of Figures

1-1 Examples of shared media. 4

2-1 $H(p)$ as a function of p, maximum when $p = 1/2$. 11
2-2 Possible grades shown with probabilities, fixed- and variable-length encodings 13
2-3 Possible grades shown with probabilities and information content. 14

3-1 Variable-length code from Figure 2-2 shown in the form of a code tree. . . . 21
3-2 An example of two non-isomorphic Huffman code trees, both optimal. . . . 23
3-3 Results from encoding more than one grade at a time. 24
3-4 Pseudo-code for the LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity. 26
3-5 Pseudo-code for LZW adaptive variable-length decoder. 26
3-6 LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabc" 27
3-7 LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$ 28

4-1 Errors accumulate in analog systems. 35
4-2 If the two voltages are adequately spaced apart, we can tolerate a certain amount of noise. 36
4-3 Picking a simple threshold voltage. 37
4-4 Sampling continuous-time voltage waveforms for transmission. 38
4-5 Transmission using a clock (top) and inferring clock edges from bit transitions between 0 and 1 and vice versa at the receiver (bottom). 39
4-6 The two cases of how the adaptation should work. 40
4-7 The "big picture". 43
4-8 Expanding on the "big picture": single link view (top) and the network view (bottom). 44

5-1 Probability of a decoding error with the replication code that replaces each bit b with n copies of b. The code rate is $1/n$. 48

5-2 Codewords separated by a Hamming distance of 2 can be used to detect single bit errors. The codewords are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit codewords. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit codewords. 49

5-3 A 2×4 arrangement for an 8-bit message with row and column parity. . . . 54

5-4 Example received 8-bit messages. Which, if any, have one error? Which, if any, have two? 54

5-5 A codeword in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code. 55

5-6 Venn diagrams of Hamming codes showing which data bits are protected by each parity bit. 57

5-7 Dividing a long message into multiple SEC-protected blocks of k bits each, adding parity bits to each constituent block. The red vertical rectangles refer to bit errors. 59

5-8 Interleaving can help recover from burst errors: code each block row-wise with an SEC, but transmit them in interleaved fashion in columnar order. As long as a set of burst errors corrupts some set of k^{th} bits, the receiver can recover from *all* the errors in the burst. 60

6-1 An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$. 66

6-2 Block diagram view of convolutional coding with shift registers. 69

6-3 State-machine view of convolutional coding. 69

6-4 When the probability of bit error is less than $1/2$, maximum-likelihood decoding boils down to finding the message whose parity bit sequence, when transmitted, has the smallest Hamming distance to the received sequence. Ties may be broken arbitrarily. Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence because of the sheer number. For instance, when $N = 256$ bits (a really small packet), the number of possibilities rivals the number of atoms in the universe! 71

6-5 The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine. 72

7-1 The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine. 76

7-2 The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00. 77

7-3 The Viterbi decoder in action. This picture shows four time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown. 86

7-4 The Viterbi decoder in action (continued from Figure 7-3. The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remeber the arc that got us to this state, so that the backward pass can be done properly. 87
7-5 Branch metric for soft decision decoding. 88
7-6 The free distance of a convolutional code. 88
7-7 Error correcting performance results for different rate-1/2 codes. 89
7-8 Error correcting performance results for three different rate-1/2 convolutional codes. The parameters of the three convolutional codes are $(111, 110)$ (labeled "$K = 3$ glist=$(7, 6)$"), $(1110, 1101)$ (labeled "$K = 4$ glist=$(14, 13)$"), and $(111, 101)$ (labeled "$K = 3$ glist=$(7, 5)$"). The top three curves below the uncoded curve are for hard decision decoding; the bottom three curves are for soft decision decoding. 89

List of Tables

6-1 Examples of generator polynomials for rate 1/2 convolutional codes with different constraint lengths. 68

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: September 6, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 1
Introduction

Our mission is to expose you to a variety of different technologies and techniques in electrical engineering and computer science. We will do this by studying several salient properties of **digital communication systems**, learning both important aspects of their design, and also the basics of how to analyze their performance. Digital communication systems are well-suited for our goals because they incorporate ideas from a large subset of electrical engineering and computer science.

Equally important, the ability to disseminate and exchange information over the world's communication networks has revolutionized the way in which people work, play, and live. At the turn of the century when everyone was feeling centennial and reflective, the U.S. National Academy of Engineering produced a list of 20 technologies that made the most impact on society in the 20th century.[1] This list included life-changing innovations such as electrification, the automobile, and the airplane; joining them were four technological achievements in the area of commununication—*radio and television*, the *telephone*, the *Internet*, and *computers*—whose technological underpinnings we will be most concerned with in this book.

Somewhat surprisingly, the Internet came in only at #13, but the reason given by the committee was that it was developed toward the latter part of the century and that they believed the most dramatic and significant impacts of the Internet would occur in the 21st century. Looking at the first decade of this century, that sentiment sounds right—the ubiquitous spread of wireless networks and mobile devices, the advent of social networks, and the ability to communicate any time and from anywhere are not just changing the face of commerce and our ability to keep in touch with friends, but are instrumental in massive societal and political changes.

Communication is fundamental to our modern existence. Who among you can imagine life without the Internet and its applications and without some form of networked mobile device? Most people feel the same way—in early 2011, over 5 billion mobile phones were active worldwide, over a billion of which had "broadband" network connectivity. To put this number (5 billion) in perspective, it is larger than the number of people in the world

[1] "The Vertiginous March of Technology", obtained from nae.edu. Document at `http://bit.ly/owMoO6`

who (in 2011) have electricity, shoes, toothbrushes, or toilets![2]

What makes our communication networks work? This course is a start at understanding the answers to this question. This question is worth studying not just because of the impact that communication systems have had on the world, but also because the technical areas cover so many different fields in EECS. Before we dive in and describe the "roadmap" for the course, we want to share a bit of the philosophy behind the material.

Traditionally, in both education and in research, much of "low-level communication" has been considered an "EE" topic, covering primarily the issues governing how bits of information move across a single communication link. In a similar vein, much of "networking" has been considered a "CS" topic, covering primarily the issues of how to build communication networks composed of multiple links. In particular, many traditional courses on "digital communication" rarely concern themselves with how networks are built and how they work, while most courses on "computer networks" treat the intricacies of communication over physical links as a black box. As a result, a sizable number of people have a deep understanding of one or the other topic, but few people are expert in every aspect of the problem. As an abstraction, however, this division is one way of conquering the immense complexity of the topic, but our goal in this course is to both understand the important details, and also understand how various abstractions allow different parts of the system to be designed and modified without paying close attention (or even really understanding) what goes on elsewhere in the system.

One drawback of preserving strong boundaries between different components of a communication system is that the details of how things work in another component may remain a mystery, even to practising engineers. In the context of communication systems, this mystery usually manifests itself as things that are "above my layer" or "below my layer". And so although we will appreciate the benefits of abstraction boundaries in this course, an important goal for us is to study the most important principles and ideas that go into the complete design of a communication system. Our goal is to convey to you both the breadth of the field as well as its depth.

In short, we cover communication systems all the way from the *source*, which has some information it wishes to transmit, to *packets*, which messages are broken into for transmission over a network, to *bits*, each of which is a "0" or a "1", to *signals*, which are analog waveforms sent over physical communication links (such as wires, fiber-optic cables, radio, or acoustic waves). We describe the salient aspects of all the layers, starting from how an application might encode messages to how the network handles packets to how links manipulate bits to how bits are converted to signals for transmission. In the process, we will study networks of different sizes, ranging from the simplest *dedicated point-to-point link*, to *shared media* with a set of communicating nodes sharing a common physical communication medium, to larger *multi-hop networks* that themselves are connected to other networks to form even bigger networks.

[2]It is in fact distressing that according to a recent survey conducted by TeleNav—and we can't tell if this is a joke—40% of iPhone users say they'd rather give up their toothbrushes for a week than their iPhones! `http://www.telenav.com/about/pr-summer-travel/report-20110803.html`

1.1 Themes

Three fundamental challenges lie at the heart of all digital communication systems and networks: *reliability*, *sharing*, and *scalability*. We will spend a considerable amount of time on the first two issues in this introductory course, but much less time on the third.

1.1.1 Reliability

A large number of factors conspire to make communication unreliable, and we will study numerous techniques to improve reliability. A common theme across these different techniques is that they all use redundancy in creative and efficient ways *to provide reliability using unreliable individual components*, using the property of independent (or perhaps weakly dependent) failures of these unreliable components to achieve reliability.

The primary challenge is to overcome a wide range of faults and disturbances that one encounters in practice, including *Gaussian noise* and *interference* that distort or corrupt signals, leading to possible *bit errors* that corrupt bits on a link, to *packet losses* caused by uncorrectable bit errors, *queue overflows*, or *link and software failures* in the network. All these problems degrade communication quality.

In practice, we are interested not only in reliability, but also in speed. Most techniques to improve communication reliability involve some form of redundancy, which reduces the speed of communication. The essence of many communication systems is how reliability and speed tradeoff against one another.

Communication speeds have increased rapidly with time. In the early 1980s, people would connect to the Internet over telephone links at speeds of barely a few kilobits per second, while today 100 Megabits per second over wireless links on laptops and 1-10 Gigabits per second with wired links are commonplace.

We will develop good tools to understand why communication is unreliable and how to overcome the problems that arise. The techniques involve error-correcting codes, handling distortions caused by "inter-symbol interference" using a *linear time-invariant* channel model, *retransmission protocols* to recover from packet losses that occur for various reasons, and developing *fault-tolerant routing protocols* to find alternate paths in networks to overcome link or node failures.

1.1.2 Efficient Sharing

"An engineer can do for a dime what any fool can do for a dollar," according to folklore. A communication network in which every pair of nodes is connected with a dedicated link would be impossibly expensive to build for even moderately sized networks. *Sharing* is therefore inevitable in communication networks because the resources used to communicate aren't cheap. We will study how to share a point-to-point link, a shared medium, and an entire multi-hop network among multiple communications.

We will develop methods to share a common communication medium among nodes, a problem common to wired media such as broadcast Ethernet, wireless technologies such as wireless local-area networks (e.g., 802.11 or WiFi), cellular data networks (e.g., "3G"), and satellite networks (see Figure 1-1).

We will study modulation and demodulation, which allow us to transmit signals over different carrier frequencies. In the process, we can ensure that multiple conversations

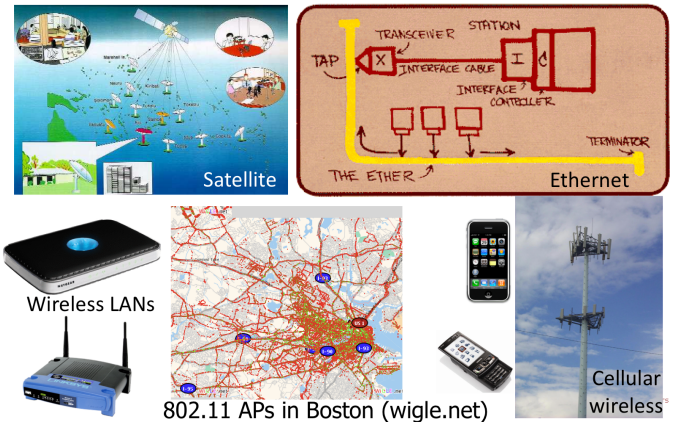


Figure 1-1: Examples of shared media.

share a communication medium by operating at different frequencies.

We will study *medium access control* (MAC) protocols, which are rules that determine how nodes must behave and react in the network—emulate either *time sharing* or *frequency sharing*. In time sharing, each node gets some duration of time to transmit data, with no other node being active. In frequency sharing, we divide the communication bandwidth (i.e., frequency range) amongst the nodes in a way that ensures a dedicated frequency sub-range for different communications, and the different communications can then occur concurrently without interference. Each scheme has its sweet spot and uses.

We will then turn to *multi-hop* networks. In these networks, multiple concurrent communications between disparate nodes occurs by sharing over the same links. That is, one might have communication between many different entities all happen over the same physical links. This sharing is orchestrated by special computers called *switches*, which implement certain operations and protocols. Multi-hop networks are generally controlled in distributed fashion, without any centralized control that determines what each node does. The questions we will address include:

1. How do multiple communications between different nodes share the network?
2. How do messages go from one place to another in the network—this task is faciliated by *routing* protocols.
3. How can we communicate information reliably across a multi-hop network (as opposed to over just a single link or shared medium)?

A word on efficiency is in order. The techniques used to share the network and achieve reliability ultimately determine the *efficiency* of the communication network. In general, one can frame the efficiency question in several ways. One approach is to minimize the capital expenditure (hardware equipment, software, link costs) and operational expenses (people, rental costs) to build and run a network capable of meeting a set of requirements (such as number of connected devices, level of performance and reliability, etc.). Another approach is to maximize the bang for the buck for a given network by maximizing the amount of "useful work" that can be done over the network. One might measure the "useful work" by calculating the aggregate throughput (in "bits per second", or at higher

speeds, the more convenient "megabits per second") achieved by the different communications, the variation of that throughput among the set of nodes, and the average delay (often called the *latency*, measured usually in milliseconds) achieved by the data transfers. Largely speaking, we will be concerned with throughput and latency in this course, and not spend much time on the broader (but no less important) questions of cost.

Of late, another aspect of efficiency that has become important in many communication systems is *energy consumption*. This issue is important both in the context of massive systems such as large data centers and for mobile computing devices such as laptops and mobile phones. Improving the energy efficiency of these systems is an important problem.

1.1.3 Scalability

In addition to reliability and efficient sharing, *scalability* (i.e., designing networks that scale to large sizes) is an important design consideration for communication networks. We will only touch on this issue, leaving most of it to later courses (6.033, 6.829).

1.2 Outline and Plan

We have divided the course into four parts: the source, and the three important abstractions (signals, bits, and packets). For pedagogic reasons, we will study them in the order given below.

1. **The source.** Ultimately, all communication is about a source wishing to send some information in the form of messages to a receiver (or to multiple receivers). Hence, it makes sense to understand the mathematical basis for *information*, to understand how to *encode* the material to be sent, and for reasons of efficiency, to understand how best to *compress* our messages so that we can send as little data as possible but yet allow the receiver to decode our messages correctly. Chapters 2 and 3 describe the key ideas behind information, *entropy* (expectation of information), and *source coding*, which enables data compression. We will study Huffman codes and the Lempel-Ziv-Welch algorithm, two widely used methods.

2. **Bits.** The main issue we will deal with here is overcoming bit errors using error-correcting codes, specifically linear block codes and convolutional codes. These codes use interesting and somewhat sophisticated algorithms that cleverly apply redundancy to reduce or eliminate bit errors. We conclude this module with a discussion of the *capacity* of the binary symmetric channel, which is a useful and key abstraction for this part of the course.

3. **Signals.** The main issues we will deal with are how to modulate bits over signals and demodulate signals to recover bits, as well as understanding how distortions of signals by communication channels can be modeled using a *linear time-invariant* (LTI) abstraction. Topics include going between time-domain and frequency-domain representations of signals, the frequency content of signals, and the frequency response of channels and filters.

4. **Packets.** The main issues we will deal with are how to share a medium using a MAC protocol, routing in multi-hop networks, and reliable data transport protocols.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: September 12, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 2
Information, Entropy, and the Motivation for Source Codes

The theory of *information* developed by Claude Shannon (SM EE '40, PhD Math '40) in the late 1940s is one of the most impactful ideas of the past century, and has changed the theory and practice of many fields of technology. The development of communication systems and networks has benefited greatly from Shannon's work. In this chapter, we will first develop the intution behind *information* and formally define it as a mathematical quantity and connect it to another property of data sources, *entropy*.

We will then show how these notions guide us to efficiently *compress* and *decompress* a data source before communicating (or storing) it without distorting the quality of information being received. A key underlying idea here is *coding*, or more precisely, *source coding*, which takes each message (or "symbol") being produced by any source of data and associate each message (symbol) with a *codeword*, while achieving several desirable properties. This mapping between input messages (symbols) and codewords is called a **code**. Our focus will be on *lossless* compression (source coding) techniques, where the recipient of any uncorrupted message can recover the original message exactly (we deal with corrupted bits later in later chapters).

2.1 Information and Entropy

One of Shannon's brilliant insights, building on earlier work by Hartley, was to realize that *regardless of the application and the semantics of the messages involved*, a general definition of information is possible. When one abstracts away the details of an application, the task of communicating something between two parties, S and R, boils down to S picking one of several (possibly infinite) messages and sending that message to R. Let's take the simplest example, when a sender wishes to send one of two messages—for concreteness, let's say that the message is to say which way the British are coming:

- "1" if by land.
- "2" if by sea.

(Had the sender been from Course VI, it would've almost certainly been "0" if by land and "1" if by sea!)

Let's say we have no prior knowledge of how the British might come, so each of these choices (messages) is equally probable. In this case, the amount of information conveyed by the sender specifying the choice is **1 bit**. Intuitively, that bit, which can take on one of two values, can be used to *encode* the particular choice. If we have to communicate a sequence of such independent events, say 1000 such events, we can encode the outcome using 1000 bits of information, each of which specifies the outcome of an associated event.

On the other hand, suppose we somehow knew that the British were far more likely to come by land than by sea (say, because there is a severe storm forecast). Then, if the message in fact says that the British are coming by sea, much more information is being conveyed than if the message said that that they were coming by land. To take another example, far more information is conveyed by my telling you that the temperature in Boston on a January day is 75°F, than if I told you that the temperature is 32°F!

The conclusion you should draw from these examples is that any quantification of "information" about an event should depend on the *probability* of the event. The greater the probability of an event, the smaller the information associated with knowing that the event has occurred.

■ 2.1.1 Information definition

Using such intuition, Hartley proposed the following definition of the information associated with an event whose probability of occurrence is p:

$$I \equiv \log(1/p) = -\log(p). \tag{2.1}$$

This definition satisfies the basic requirement that it is a decreasing function of p. But so do an infinite number of functions, so what is the intuition behind using the logarithm to define information? And what is the base of the logarithm?

The second question is easy to address: you can use any base, because $\log_a(1/p) = \log_b(1/p)/\log_a b$, for any two bases a and b. Following Shannon's convention, *we will use base 2*,[1] in which case the unit of information is called a **bit**.[2]

The answer to the first question, why the logarithmic function, is that the resulting definition has several elegant resulting properties, and it is the simplest function that provides these properties. One of these properties is **additivity**. If you have two independent events (i.e., events that have nothing to do with each other), then the probability that they both occur is equal to the *product* of the probabilities with which they each occur. What we would like is for the corresponding information to *add up*. For instance, the event that it rained in Seattle yesterday and the event that the number of students enrolled in 6.02 exceeds 150 are independent, and if I am told something about both events, the amount of information I now have should be the sum of the information in being told individually of the occurrence of the two events.

The logarithmic definition provides us with the desired additivity because, given two

[1] And we won't mention the base; if you see a log in this chapter, it will be to base 2 unless we mention otherwise.

[2] If we were to use base 10, the unit would be *Hartleys*, and if we were to use the natural log, base e, it would be *nats*, but no one uses those units in practice.

independent events A and B with probabilities p_A and p_B,

$$I_A + I_B = \log(1/p_A) + \log(1/p_B) = \log \frac{1}{p_A p_B} = \log \frac{1}{P(A \text{ and } B)}.$$

■ 2.1.2 Examples

Suppose that we're faced with N equally probable choices. What is the information received when I tell you which of the N choices occurred?

Because the probability of each choice is $1/N$, the information is $\log(1/(1/N)) = \log N$ bits.

Now suppose there are initially N equally probable choices, and I tell you something that narrows the possibilities down to one of M equally probable choices. How much information have I given you about the choice?

We can answer this question by observing that you now know that the probability of the choice narrowing done from N equi-probable possibilities to M equi-probable ones is M/N. Hence, the information you have received is $\log(1/(M/N)) = \log(N/M)$ bits. (Note that when $M = 1$, we get the expected answer of $\log N$ bits.)

We can therefore write a convenient rule:

> Suppose we have received information that narrows down a set of N equi-probable choices to one of M equi-probable choices. Then, we have received $\log(N/M)$ bits of information.

Some examples may help crystallize this concept:

One flip of a fair coin

Before the flip, there are two equally probable choices: heads or tails. After the flip, we've narrowed it down to one choice. Amount of information = $\log_2(2/1) = 1$ bit.

Roll of two dice

Each die has six faces, so in the roll of two dice there are 36 possible combinations for the outcome. Amount of information = $\log_2(36/1) = 5.2$ bits.

Learning that a randomly chosen decimal digit is even

There are ten decimal digits; five of them are even (0, 2, 4, 6, 8). Amount of information = $\log_2(10/5) = 1$ bit.

Learning that a randomly chosen decimal digit ≥ 5

Five of the ten decimal digits are greater than or equal to 5. Amount of information = $\log_2(10/5) = 1$ bit.

Learning that a randomly chosen decimal digit is a multiple of 3

Four of the ten decimal digits are multiples of 3 (0, 3, 6, 9). Amount of information = $\log_2(10/4) = 1.322$ bits.

Learning that a randomly chosen decimal digit is even, ≥ 5, and a multiple of 3

Only one of the decimal digits, 6, meets all three criteria. Amount of information =

$\log_2(10/1) = 3.322$ bits. Note that this information is same as the sum of the previous three examples: information is cumulative if the joint probability of the events revealed to us *factors* intothe product of the individual probabilities.

In this example, we can calculate the probability that they all occur together, and compare that answer with the product of the probabilities of each of them occurring individually. Let event A be "the digit is even", event B be "the digit is ≥ 5, and event C be "the digit is a multiple of 3". Then, $P(A \text{ and } B \text{ and } C) = 1/10$ because there is only one digit, 6, that satisfies all three conditions. $P(A) \cdot P(B) \cdot P(C) = 1/2 \cdot 1/2 \cdot 4/10 = 1/10$ as well. The reason information adds up is that $\log(1/P(AandBandC) = \log 1/P(A) + \log 1/P(B) + \log(1/P(C)$.

Note that *pairwise indepedence* between events is actually not necessary for information from three (or more) events to add up. In this example, $P(AandB) = P(A) \cdot P(B|A) = 1/2 \cdot 2/5 = 1/5$, while $P(A) \cdot P(B) = 1/2 \cdot 1/2 = 1/4$.

Learning that a randomly chosen decimal digit is a prime

Four of the ten decimal digits are primes—2, 3, 5, and 7. Amount of information = $\log_2(10/4) = 1.322$ bits.

Learning that a randomly chosen decimal digit is even and prime

Only one of the decimal digits, 2, meets both criteria. Amount of information = $\log_2(10/1) = 3.322$ bits. Note that this quantity is *not* the same as the sum of the information contained in knowing that the digit is even and the digit is prime. The reason is that those events are not independent: the probability that a digit is even *and* prime is $1/10$, and is *not* the product of the probabilities of the two events (i.e., not equal to $1/2 \times 4/10$).

To summarize: more information is received when learning of the occurrence of an unlikely event (small p) than learning of the occurrence of a more likely event (large p). The information learned from the occurrence of an event of probability p is defined to be $\log(1/p)$.

■ 2.1.3 Entropy

Now that we know how to measure the information contained in a given event, we can quantify the *expected information* in a set of possible outcomes. Specifically, if an event i occurs with probability $p_i, 1 \leq i \leq N$ out of a set of N events, then the average or expected information is given by

$$H(p_1, p_2, \ldots p_N) = \sum_{i=1}^{N} p_i \log(1/p_i). \tag{2.2}$$

H is also called the **entropy** (or *Shannon entropy*) of the probability distribution. Like information, it is also measured in bits. It is simply the sum of several terms, each of which is the information of a given event weighted by the probability of that event occurring. It is often useful to think of the entropy as *the average or expected* **uncertainty** *associated with this set of events*.

In the important special case of two *mutually exclusive* events (i.e., exactly one of the two

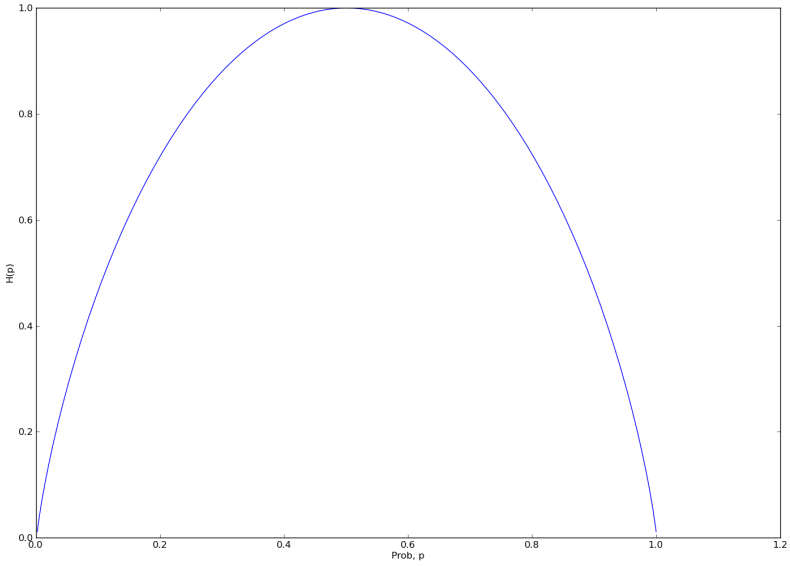


Figure 2-1: $H(p)$ **as a function of** p**, maximum when** $p = 1/2$**.**

events can occur), occuring with probabilities p and $1 - p$, respectively, the entropy

$$H(p, 1-p) = -p \log p - (1-p) \log(1-p). \tag{2.3}$$

We will be lazy and refer to this special case, $H(p, 1-p)$ as simply $H(p)$.

This entropy as a function of p is plotted in Figure 2-1. It is symmetric about $p = 1/2$, with its maximum value of 1 bit occuring when $p = 1/2$. Note that $H(0) = H(1) = 0$; although $\log(1/p) \to \infty$ as $p \to 0, \lim_{p \to 0} p \log(1/p) \to 0$.

It is easy to verify that the expression for H from Equation (2.2) is always non-negative. Moreover, $H(p_1, p_2, \ldots p_N) \leq \log N$ always.

■ 2.2 Source Codes

We now turn to the problem of *source coding*, i.e., taking a set of messages that need to be sent from a sender and *encoding* them in a way that is efficient. The notions of information and entropy will be fundamentally important in this effort.

Many messages have an obvious encoding, e.g., an ASCII text file consists of sequence of individual characters, each of which is independently encoded as a separate byte. There are other such encodings: images as a raster of color pixels (e.g., 8 bits each of red, green and blue intensity), sounds as a sequence of samples of the time-domain audio waveform, etc. What makes these encodings so popular is that they are produced and consumed by our computer's peripherals—characters typed on the keyboard, pixels received from a digital camera or sent to a display, and digitized sound samples output to the computer's audio chip.

All these encodings involve a sequence of fixed-length symbols, each of which can be

easily manipulated independently. For example, to find the 42^{nd} character in the file, one just looks at the 42^{nd} byte and interprets those 8 bits as an ASCII character. A text file containing 1000 characters takes 8000 bits to store. If the text file were HTML to be sent over the network in response to an HTTP request, it would be natural to send the 1000 bytes (8000 bits) exactly as they appear in the file.

But let's think about how we might compress the file and send fewer than 8000 bits. If the file contained English text, we'd expect that the letter *e* would occur more frequently than, say, the letter *x*. This observation suggests that if we encoded *e* for transmission using *fewer* than 8 bits—and, as a trade-off, had to encode less common characters, like *x*, using more than 8 bits—we'd expect the encoded message to be shorter *on average* than the original method. So, for example, we might choose the bit sequence 00 to represent *e* and the code 100111100 to represent *x*.

This intuition is consistent with the definition of the amount of information: commonly occurring symbols have a higher p_i and thus convey less information, so we need fewer bits to encode such symbols. Similarly, infrequently occurring symbols like *x* have a lower p_i and thus convey more information, so we'll use more bits when encoding such symbols. This intuition helps meet our goal of matching the size of the transmitted data to the information content of the message.

The mapping of information we wish to transmit or store into bit sequences is referred to as a **code**. Two examples of codes (fixed-length and variable-length) are shown in Figure 2-2, mapping different grades to bit sequences in one-to-one fashion. The fixed-length code is straightforward, but the variable-length code is not arbitrary, but has been carefully designed, as we will soon learn. Each bit sequence in the code is called a **codeword**.

When the mapping is performed at the source of the data, generally for the purpose of *compressing* the data (ideally, to match the expected number of bits to the underlying entropy), the resulting mapping is called a **source code**. Source codes are distinct from **channel codes** we will study in later chapters. Source codes *remove redundancy* and compress the data, while channel codes *add redundancy* in a controlled way to improve the error resilience of the data in the face of bit errors and erasures caused by imperfect communication channels. This chapter and the next are about source codes.

We can generalize this insight about encoding common symbols (such as the letter *e*) more succinctly than uncommon symbols into a strategy for *variable-length codes*:

> Send commonly occurring symbols using shorter codewords (fewer bits) and infrequently occurring symbols using longer codewords (more bits).

We'd expect that, on average, encoding the message with a variable-length code would take fewer bits than the original fixed-length encoding. Of course, if the message were all *x*'s the variable-length encoding would be longer, but our encoding scheme is designed to optimize the expected case, not the worst case.

Here's a simple example: suppose we had to design a system to send messages containing 1000 6.02 grades of *A*, *B*, *C* and *D* (MIT students rarely, if ever, get an F in 6.02 $\ddot\smile$). Examining past messages, we find that each of the four grades occurs with the probabilities shown in Figure 2-2.

With four possible choices for each grade, if we use the fixed-length encoding, we need 2 bits to encode a grade, for a total transmission length of 2000 bits when sending 1000 grades.

Grade	Probability	Fixed-length Code	Variable-length Code
A	$1/3$	00	10
B	$1/2$	01	0
C	$1/12$	10	110
D	$1/12$	11	111

Figure 2-2: Possible grades shown with probabilities, fixed- and variable-length encodings

Fixed-length encoding for $BCBAAB$: 01 10 01 00 00 01 (12 bits)

With a fixed-length code, the size of the transmission doesn't depend on the actual message—sending 1000 grades always takes exactly 2000 bits.

Decoding a message sent with the fixed-length code is straightforward: take each pair of received bits and look them up in the table above to determine the corresponding grade. Note that it's possible to determine, say, the 42^{nd} grade without decoding any other of the grades—just look at the 42^{nd} pair of bits.

Using the variable-length code, the number of bits needed for transmitting 1000 grades depends on the grades.

Variable-length encoding for $BCBAAB$: 0 110 0 10 10 0 (10 bits)

If the grades were all B, the transmission would take only 1000 bits; if they were all C's and D's, the transmission would take 3000 bits. But we can use the grade probabilities given in Figure 2-2 to compute the *expected length of a transmission* as

$$1000[(\frac{1}{3})(2) + (\frac{1}{2})(1) + (\frac{1}{12})(3) + (\frac{1}{12})(3)] = 1000[1\frac{2}{3}] = 1666.7 \text{ bits}$$

So, on average, using the variable-length code would shorten the transmission of 1000 grades by 333 bits, a savings of about 17%. Note that to determine, say, the 42^{nd} grade, we would need to decode the first 41 grades to determine where in the encoded message the 42^{nd} grade appears.

Using variable-length codes looks like a good approach if we want to send fewer bits on average, but preserve all the information in the original message. On the downside, we give up the ability to access an arbitrary message symbol without first decoding the message up to that point.

One obvious question to ask about a particular variable-length code: is it the best encoding possible? Might there be a different variable-length code that could do a better job, i.e., produce even shorter messages on the average? How short can the messages be on the average? We turn to this question next.

■ 2.3 How Much Compression Is Possible?

Ideally we'd like to design our compression algorithm to produce as few bits as possible: just enough bits to represent the information in the message, but no more. Ideally, we will be able to use no more bits than the amount of information, as defined in Section 2.1, contained in the message, at least on average.

Specifically, the entropy, defined by Equation (2.2), tells us the expected amount of information in a message, when the message is drawn from a set of possible messages, each occurring with some probability. The entropy is a lower bound on the amount of information that must be sent, on average, when transmitting data about a particular choice.

What happens if we violate this lower bound, i.e., we send fewer bits on the average than called for by Equation (2.2)? In this case the receiver will not have sufficient information and there will be some remaining ambiguity—exactly what ambiguity depends on the encoding, but to construct a code of fewer than the required number of bits, some of the choices must have been mapped into the same encoding. Thus, when the recipient receives one of the overloaded encodings, it will not have enough information to unambiguously determine which of the choices actually occurred.

Equation (2.2) answers our question about how much compression is possible by giving us a lower bound on the number of bits that must be sent to resolve all ambiguities at the recipient. Reprising the example from Figure 2-2, we can update the figure using Equation (2.1).

Grade	p_i	$log_2(1/p_i)$
A	$1/3$	1.58 bits
B	$1/2$	1 bit
C	$1/12$	3.58 bits
D	$1/12$	3.58 bits

Figure 2-3: Possible grades shown with probabilities and information content.

Using equation (2.2) we can compute the information content when learning of a particular grade:

$$\sum_{i=1}^{N} p_i \log_2(\frac{1}{p_i}) = (\frac{1}{3})(1.58) + (\frac{1}{2})(1) + (\frac{1}{12})(3.58) + (\frac{1}{12})(3.58) = 1.626 \text{ bits}$$

So encoding a sequence of 1000 grades requires transmitting 1626 bits on the average. The variable-length code given in Figure 2-2 encodes 1000 grades using 1667 bits on the average, and so doesn't achieve the maximum possible compression. It turns out the example code does as well as possible when encoding one grade at a time. To get closer to the lower bound, we would need to encode sequences of grades—more on this idea below.

Finding a "good" code—one where the length of the encoded message matches the information content (i.e., the entropy)—is challenging and one often has to think "outside the box". For example, consider transmitting the results of 1000 flips of an unfair coin where probability of heads is given by p_H. The information content in an unfair coin flip can be computed using equation (2.3):

$$p_H \log_2(1/p_H) + (1 - p_H) \log_2(1/(1 - p_H))$$

For $p_H = 0.999$, this entropy evaluates to .0114. Can you think of a way to encode 1000 unfair coin flips using, on average, just 11.4 bits? The recipient of the encoded message must be able to tell for each of the 1000 flips which were heads and which were tails. Hint:

with a budget of just 11 bits, one obviously can't encode each flip separately!

In fact, some effective codes leverage the context in which the encoded message is being sent. For example, if the recipient is expecting to receive a Shakespeare sonnet, then it's possible to encode the message using just 8 bits if one knows that there are only 154 Shakespeare sonnets. That is, if the sender and receiver both know the sonnets, and the sender just wishes to tell the receiver which sonnet to read or listen to, he can do that using a very small number of bits, just $\log 154$ bits if all the sonnets are equi-probable!

2.4 Why Compression?

There are several reasons for using compression:

- Shorter messages take less time to transmit and so the complete message arrives more quickly at the recipient. This is good for both the sender and recipient since it frees up their network capacity for other purposes and reduces their network charges. For high-volume senders of data (such as Google, say), the impact of sending half as many bytes is economically significant.
- Using network resources sparingly is good for *all* the users who must share the internal resources (packet queues and links) of the network. Fewer resources per message means more messages can be accommodated within the network's resource constraints.
- Over error-prone links with non-negligible bit error rates, compressing messages before they are channel-coded using error-correcting codes can help improve throughput because all the redundancy in the message can be designed in to improve error resilience, after removing any other redundancies in the original message. It is better to design in redundancy with the explicit goal of correcting bit errors, rather than rely on whatever sub-optimal redundancies happen to exist in the original message.

Compression is traditionally thought of as an *end-to-end function*, applied as part of the application-layer protocol. For instance, one might use lossless compression between a web server and browser to reduce the number of bits sent when transferring a collection of web pages. As another example, one might use a compressed image format such as JPEG to transmit images, or a format like MPEG to transmit video. However, one may also apply compression at the link layer to reduce the number of transmitted bits and eliminate redundant bits (before possibly applying an error-correcting code over the link). When applied at the link layer, compression only makes sense if the data is inherently compressible, which means it cannot already be compressed and must have enough redundancy to extract compression gains.

The next chapter describes two compression (source coding) schemes: Huffman Codes and Lempel-Ziv-Welch (LZW) compression.

Exercises

1. Several people at a party are trying to guess a 3-bit binary number. Alice is told that the number is odd; Bob is told that it is not a multiple of 3 (i.e., not 0, 3, or 6); Charlie

is told that the number contains exactly two 1's; and Deb is given all three of these clues. How much information (in bits) did each player get about the number?

2. After careful data collection, Alyssa P. Hacker observes that the probability of "HIGH" or "LOW" traffic on Storrow Drive is given by the following table:

	HIGH traffic level	LOW traffic level
If the Red Sox are playing	P(HIGH traffic) = 0.999	P(LOW traffic) = 0.001
If the Red Sox are not playing	P(HIGH traffic) = 0.25	P(LOW traffic) = 0.75

   (a) If it is known that the Red Sox are playing, then how much information in bits is conveyed by the statement that the traffic level is LOW. Give your answer as a mathematical expression.

   (b) Suppose it is known that the Red Sox are **not** playing. What is the entropy of the corresponding probability distribution of traffic? Give your answer as a mathematical expression.

3. X is an unknown 4-bit binary number picked uniformly at random from the set of all possible 4-bit numbers. You are given another 4-bit binary number, Y, and told that the Hamming distance between X (the unknown number) and Y (the number you know) is *two*. How many bits of information about X have you been given?

4. In Blackjack the dealer starts by dealing 2 cards each to himself and his opponent: one face down, one face up. After you look at your face-down card, you know a total of three cards. Assuming this was the first hand played from a new deck, how many bits of information do you have about the dealer's face down card after having seen three cards?

5. The following table shows the undergraduate and MEng enrollments for the School of Engineering.

Course (Department)	# of students	% of total
I (Civil & Env.)	121	7%
II (Mech. Eng.)	389	23%
III (Mat. Sci.)	127	7%
VI (EECS)	645	38%
X (Chem. Eng.)	237	13%
XVI (Aero & Astro)	198	12%
Total	1717	100%

   (a) When you learn a randomly chosen engineering student's department you get some number of bits of information. For which student department do you get the least amount of information?

   (b) **After studying Huffman codes in the next chapter,** design a Huffman code to encode the departments of randomly chosen groups of students. Show your Huffman tree and give the code for each course.

(c) If your code is used to send messages containing only the encodings of the departments for each student in groups of 100 randomly chosen students, what is the average length of such messages?

6. You're playing an online card game that uses a deck of 100 cards containing 3 Aces, 7 Kings, 25 Queens, 31 Jacks and 34 Tens. In each round of the game the cards are shuffled, you make a bet about what type of card will be drawn, then a single card is drawn and the winners are paid off. The drawn card is reinserted into the deck before the next round begins.

   (a) How much information do you receive when told that a Queen has been drawn during the current round?

   (b) Give a numeric expression for the information content received when learning about the outcome of a round.

   (c) **After you learn about Huffman codes in the next chapter,** construct a variable-length Huffman encoding that minimizes the length of messages that report the outcome of a sequence of rounds. The outcome of a single round is encoded as A (ace), K (king), Q (queen), J (jack) or X (ten). Specify your encoding for each of A, K, Q, J and X.

   (d) **Again, after studying Huffman codes,** use your code from part (c) to calculate the expected length of a message reporting the outcome of 1000 rounds (i.e., a message that contains 1000 symbols)?

   (e) The Nevada Gaming Commission regularly receives messages in which the outcome for each round is encoded using the symbols A, K, Q, J, and X. They discover that a large number of messages describing the outcome of 1000 rounds (i.e., messages with 1000 symbols) can be compressed by the LZW algorithm into files each containing 43 bytes in total. They decide to issue an indictment for running a crooked game. Why did the Commission issue the indictment?

7. Consider messages made up entirely of vowels (A, E, I, O, U). Here's a table of probabilities for each of the vowels:

l	p_l	$\log_2(1/p_l)$	$p_l \log_2(1/p_l)$
A	0.22	2.18	0.48
E	0.34	1.55	0.53
I	0.17	2.57	0.43
O	0.19	2.40	0.46
U	0.08	3.64	0.29
Totals	1.00	12.34	2.19

   (a) Give an expression for the number of bits of information you receive when learning that a particular vowel is either I or U.

   (b) **After studying Huffman codes in the next chapter,** use Huffman's algorithm to construct a variable-length code assuming that each vowel is encoded individually. Draw a diagram of the Huffman tree and give the encoding for each of the vowels.

(c) Using your code from part (B) above, give an expression for the expected length in bits of an encoded message transmitting 100 vowels.

(d) Ben Bitdiddle spends all night working on a more complicated encoding algorithm and sends you email claiming that using his code the expected length in bits of an encoded message transmitting 100 vowels is 197 bits. Would you pay good money for his implementation?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: September 12, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 3
Compression Algorithms: Huffman and Lempel-Ziv-Welch (LZW)

This chapter discusses **source coding**, specifically two algorithms to compress messages (i.e., a sequence of symbols). The first, Huffman coding, is efficient when one knows the probabilities of the different symbols one wishes to send. In the context of Huffman coding, a message can be thought of as a sequence of symbols, with each symbol drawn independently from some known distribution. The second, LZW (for Lempel-Ziv-Welch) is an **adaptive compression** algorithm that does not assume any a priori knowledge of the symbol probabilities. Both Huffman codes and LZW are widely used in practice, and are a part of many real-world standards such as GIF, JPEG, MPEG, MP3, and more.

■ 3.1 Properties of Good Source Codes

Suppose the source wishes to send a message, i.e., a sequence of **symbols**, drawn from some alphabet. The alphabet could be text, it could be bit sequences corresponding to a digitized picture or video obtained from a digital or analog source (we will look at an example of such a source in more detail in the next chapter), or it could be something more abstract (e.g., "ONE" if by land and "TWO" if by sea, or h for heavy traffic and ℓ for light traffic on a road).

A **code** is a mapping between symbols and **codewords**. The reason for doing the mapping is that we would like to adapt the message into a form that can be manipulated (processed), stored, and transmitted over communication channels. Codewords made of bits ("zeroes and ones") are a convenient and effective way to achieve this goal.

For example, if we want to communicate the grades of students in 6.02, we might use the following encoding:

"A" $\rightarrow$ 1
"B" $\rightarrow$ 01
"C" $\rightarrow$ 000
"D" $\rightarrow$ 001

Then, if we want to transmit a sequence of grades, we might end up sending a message such as 0010001110100001. The receiver can decode this received message as the sequence of grades "DCAAABCB" by looking up the appropriate contiguous and non-overlapping substrings of the received message in the code (i.e., the mapping) shared by it and the source.

Instantaneous codes. A useful property for a code to possess is that a symbol corresponding to a received codeword be decodable as soon as the corresponding codeword is received. Such a code is called an **instantaneous code**. The example above is an instantaneous code. The reason is that if the receiver has already decoded a sequence and now receives a "1", then it knows that the symbol *must* be "A". If it receives a "0", then it looks at the next bit; if that bit is "1", then it knows the symbol is "B"; if the next bit is instead "0", then it does not yet know what the symbol is, but the *next* bit determines uniquely whether the symbol is "C" (if "0") or "D" (if "1"). Hence, this code is instantaneous.

Code trees and prefix-free codes. A convenient way to visualize codes is using a *code tree*, as shown in Figure 3-1 for an instantaneous code with the following encoding:

"A" $\rightarrow$ 10
"B" $\rightarrow$ 0
"C" $\rightarrow$ 110
"D" $\rightarrow$ 111

In general, a code tree is a binary tree with the symbols at the nodes of the tree and the edges of the tree are labeled with "0" or "1" to signify the encoding. To find the encoding of a symbol, the receiver simply walks the path from the root (the top-most node) to that symbol, emitting the label on the edges traversed.

If, in a code tree, the symbols are all at the leaves, then the code is said to be **prefix-free**, because no codeword is a prefix of another codeword. Prefix-free codes (and code trees) are naturally instantaneous, which makes them attractive.[1]

Expected code length. Our final definition is for the expected length of a code. Given N symbols, with symbol i occurring with probability p_i, if we have a code in which symbol i has length l_i in the code tree (i.e., the codeword is ℓ_i bits long), then the expected length of the code is $\sum_{i=1}^{N} p_i \ell_i$.

In general, codes with small expected code length are interesting and useful because they allow us to **compress** messages, delivering messages without any loss of information but consuming fewer bits than without the code. Because one of our goals in designing communication systems is efficient sharing of the communication links among different users or conversations, the ability to send data in as few bits as possible is important.

We say that a code is *optimal* if its expected code length, L, is the minimum among all possible codes. The corresponding code tree gives us the optimal mapping between symbols and codewords, and is usually not unique. Shannon proved that the expected code length of any decodable code cannot be smaller than the entropy, H, of the underlying probability distribution over the symbols. He also showed the existence of codes that achieve entropy asymptotically, as the length of the coded messages approaches ∞. Thus,

[1]Somewhat unfortunately, several papers and books use the term "prefix code" to mean the same thing as a "prefix-free code". Caveat emptor.

an optimal code will have an expected code length that "matches" the entropy for long messages.

The rest of this chapter describes two optimal codes. First, Huffman codes, which are optimal instantaneous codes when the probabilities of the various symbols are given and we restrict ourselves to mapping individual symbols to codewords. It is a prefix-free, instantaneous code, satisfying the property $H \leq L \leq H + 1$. Second, the LZW algorithm, which adapts to the actual distribution of symbols in the message, not relying on any a priori knowledge of symbol probabilities.

3.2 Huffman Codes

Huffman codes give an efficient encoding for a list of symbols to be transmitted, when we know their probabilities of occurrence in the messages to be encoded. We'll use the intuition developed in the previous chapter: more likely symbols should have shorter encodings, less likely symbols should have longer encodings.

If we draw the variable-length code of Figure 2-2 as a code tree, we'll get some insight into how the encoding algorithm should work:

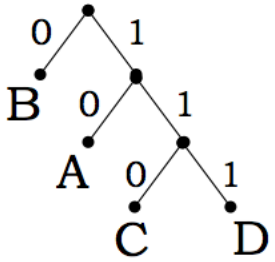


Figure 3-1: Variable-length code from Figure 2-2 shown in the form of a code tree.

To encode a symbol using the tree, start at the root and traverse the tree until you reach the symbol to be encoded—the encoding is the concatenation of the branch labels in the order the branches were visited. The destination node, which is always a "leaf" node for an instantaneous or prefix-free code, determines the path, and hence the encoding. So B is encoded as 0, C is encoded as 110, and so on. Decoding complements the process, in that now the path (codeword) determines the symbol, as described in the previous section. So 111100 is decoded as: $111 \rightarrow D$, $10 \rightarrow A$, $0 \rightarrow B$.

Looking at the tree, we see that the more probable symbols (e.g., B) are near the root of the tree and so have short encodings, while less-probable symbols (e.g., C or D) are further down and so have longer encodings. David Huffman used this observation while writing a term paper for a graduate course taught by Bob Fano here at M.I.T. in 1951 to devise an algorithm for building the decoding tree for an optimal variable-length code.

Huffman's insight was to build the decoding tree *bottom up*, starting with the least probable symbols and applying a greedy strategy. Here are the steps involved, along with a worked example based on the variable-length code in Figure 2-2. The input to the algorithm is a set of symbols and their respective probabilities of occurrence. The output is the code tree, from which one can read off the codeword corresponding to each symbol.

1. **Input:** A set S of tuples, each tuple consisting of a message symbol and its associated probability.

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.083, C), (0.083, D)\}$

2. Remove from S the two tuples with the smallest probabilities, resolving ties arbitrarily. Combine the two symbols from the removed tuples to form a new tuple (which will represent an interior node of the code tree). Compute the probability of this new tuple by adding the two probabilities from the tuples. Add this new tuple to S. (If S had N tuples to start, it now has $N - 1$, because we removed two tuples and added one.)

   Example: $S \leftarrow \{(0.333, A), (0.5, B), (0.167, C \wedge D)\}$

3. Repeat step 2 until S contains only a single tuple. (That last tuple represents the root of the code tree.)

   Example, iteration 2: $S \leftarrow \{(0.5, B), (0.5, A \wedge (C \wedge D))\}$
   Example, iteration 3: $S \leftarrow \{(1.0, B \wedge (A \wedge (C \wedge D)))\}$

Et voila! The result is a code tree representing a variable-length code for the given symbols and probabilities. As you'll see in the Exercises, the trees aren't always "tall and thin" with the left branch leading to a leaf; it's quite common for the trees to be much "bushier." As a simple example, consider input symbols A, B, C, D, E, F, G, H with equal probabilities of occurrences (1/8 for each). In the first pass, one can pick any two as the two lowest-probability symbols, so let's pick A and B without loss of generality. The combined AB symbol has probability 1/4, while the other six symbols have probability 1/8 each. In the next iteration, we can pick any two of the symbols with probability 1/8, say C and D. Continuing this process, we see that after four iterations, we would have created four sets of combined symbols, each with probability 1/4 each. Applying the algorithm, we find that the code tree is a complete binary tree where every symbol has a codeword of length 3, corresponding to all combinations of 3-bit words (000 through 111).

Huffman codes have the biggest reduction in the expected length of the encoded message when some symbols are substantially more probable than other symbols. If all symbols are equiprobable, then all codewords are roughly the same length, and there are (nearly) fixed-length encodings whose expected code lengths approach entropy and are thus close to optimal.

■ 3.2.1 Properties of Huffman Codes

We state some properties of Huffman codes here. We don't prove these properties formally, but provide intuition about why they hold.

Non-uniqueness. In a trivial way, because the 0/1 labels on any pair of branches in a code tree can be reversed, there are in general multiple different encodings that all have the same expected length. In fact, there *may* be multiple optimal codes for a given set of symbol probabilities, and depending on how ties are broken, Huffman coding can produce different *non-isomorphic* code trees, i.e., trees that look different structurally and aren't just relabelings of a single underlying tree. For example, consider six symbols with proba-

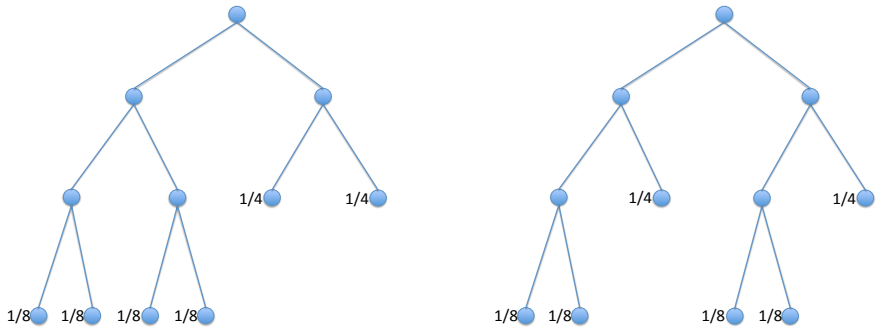


Figure 3-2: An example of two non-isomorphic Huffman code trees, both optimal.

bilities $1/4, 1/4, 1/8, 1/8, 1/8, 1/8$. The two code trees shown in Figure 3-2 are both valid Huffman (optimal) codes.

Optimality. Huffman codes are optimal in the sense that there are no other codes with shorter expected length, when restricted to instantaneous (prefix-free) codes and symbols occur independently in messages from a known probability distribution.

We state here some propositions that are useful in establishing the optimality of Huffman codes.

Proposition 3.1 *In any optimal code tree for a prefix-free code, each node has either zero or two children.*

To see why, suppose an optimal code tree has a node with one child. If we take that node and move it up one level to its parent, we will have reduced the expected code length, and the code will remain decodable. Hence, the original tree was not optimal, a contradiction.

Proposition 3.2 *In the code tree for a Huffman code, no node has exactly one child.*

To see why, note that we always combine the two lowest-probability nodes into a single one, which means that in the code tree, each internal node (i.e., non-leaf node) comes from two combined nodes (either internal nodes themselves, or original symbols).

Proposition 3.3 *There exists an optimal code in which the two least-probable symbols:*

- *have the longest length, and*
- *are siblings, i.e., their codewords differ in exactly the one bit (the last one).*

Proof. Let z be the least-probable symbol. If it is not at maximum depth in the optimal code tree, then some other symbol, call it s, must be at maximum depth. But because $p_z < p_s$, if we swapped z and s in the code tree, we would end up with a code with smaller expected length. Hence, z must have a codeword at least as long as every other codeword.

Now, symbol z must have a sibling in the optimal code tree, by Proposition 3.1. Call it x. Let y be the symbol with second lowest probability; i.e., $p_x \geq p_y \geq p_x$. If $p_x = p_y$, then

the proposition is proved. Let's swap x and y in the code tree, so now y is a sibling of z. The expected code length of this code tree is not larger than the pre-swap optimal code tree, because p_x is strictly greater than p_y, proving the proposition. ■

Theorem 3.1 *Huffman coding over a set of symbols with known probabilities produces a code tree whose expected length is optimal.*

Proof. Proof by induction on n, the number of symbols. Let the symbols be $x_1, x_2, \ldots, x_{n-1}, x_n$ and let their respective probabilities of occurrence be $p_1 \geq p_2 \geq \ldots \geq p_{n-1} \geq p_n$. From Proposition 3.3, there exists an optimal code tree in which x_{n-1} and x_n have the longest length and are siblings.

Inductive hypothesis: Assume that Huffman coding produces an optimal code tree on an input with $n-1$ symbols with associated probabilities of occurrence. The base case is trivial to verify.

Let H_n be the expected cost of the code tree generated by Huffman coding on the n symbols $x_1, x_2, \ldots, x_n$. Then, $H_n = H_{n-1} + p_{n-1} + p_n$, where H_{n-1} is the expected cost of the code tree generated by Huffman coding on $n-1$ input symbols $x_1, x_2, \ldots x_{n-2}, x_{n-1,n}$ with probabilities $p_1, p_2, \ldots, p_{n-2}, (p_{n-1} + p_n)$.

By the inductive hypothesis, $H_{n-1} = L_{n-1}$, the expected cost of the optimal code tree over $n-1$ symbols. Moreover, from Proposition 3.3, there exists an optimal code tree over n symbols for which $L_n = L_{n-1} + (p_{n-1} + p_n)$. Hence, there exists an optimal code tree whose expected cost, L_n, is equal to the expected cost, H_n, of the Huffman code over the n symbols. ■

Huffman coding with grouped symbols. The entropy of the distribution shown in Figure 2-2 is 1.626. The per-symbol encoding of those symbols using Huffman coding produces a code with expected length 1.667, which is noticeably larger (e.g., if we were to encode 10,000 grades, the difference would be about 410 bits). Can we apply Huffman coding to get closer to entropy?

One approach is to *group* symbols into larger "metasymbols" and encode those instead, usually with some gain in compression but at a cost of increased encoding and decoding complexity.

Consider encoding pairs of symbols, triples of symbols, quads of symbols, etc. Here's a tabulation of the results using the grades example from Figure 2-2:

Size of grouping	Number of leaves in tree	Expected length for 1000 grades
1	4	1667
2	16	1646
3	64	1637
4	256	1633

Figure 3-3: Results from encoding more than one grade at a time.

We see that we can come closer to the Shannon lower bound (i.e., entropy) of 1.626 bits by encoding grades in larger groups at a time, but at a cost of a more complex encoding

and decoding process. This approach still has two problems: first, it requires knowledge of the individual symbol probabilities, and second, it assumes that the probability of each symbol is independent and identically distributed. In practice, however, symbol probabilities change message-to-message, or even within a single message.

This last observation suggests that it would be nice to create an *adaptive* variable-length encoding that takes into account the actual content of the message. The LZW algorithm, presented in the next section, is such a method.

■ 3.3 LZW: An Adaptive Variable-length Source Code

Let's first understand the compression problem better by considering the problem of digitally representing and transmitting the text of a book written in, say, English. A simple approach is to analyze a few books and estimate the probabilities of different letters of the alphabet. Then, treat each letter as a symbol and apply Huffman coding to compress a document.

This approach is reasonable but ends up achieving relatively small gains compared to the best one can do. One big reason why is that the probability with which a letter appears in any text is not always the same. For example, a priori, "x" is one of the least frequently appearing letters, appearing only about 0.3% of the time in English text. But if in the sentence "... nothing can be said to be certain, except death and ta___", the next letter is almost certainly an "x". In this context, no other letter can be more certain!

Another reason why we might expect to do better than Huffman coding is that it is often unclear what the best symbols might be. For English text, because individual letters vary in probability by context, we might be tempted to try out words. It turns out that word occurrences also change in probability depend on context.

An approach that *adapts* to the material being compressed might avoid these shortcomings. One approach to adaptive encoding is to use a two pass process: in the first pass, count how often each symbol (or pairs of symbols, or triples—whatever level of grouping you've chosen) appears and use those counts to develop a Huffman code customized to the contents of the file. Then, in the second pass, encode the file using the customized Huffman code. This strategy is expensive but workable, yet it falls short in several ways. Whatever size symbol grouping is chosen, it won't do an optimal job on encoding recurring groups of some different size, either larger or smaller. And if the symbol probabilities change dramatically at some point in the file, a one-size-fits-all Huffman code won't be optimal; in this case one would want to change the encoding midstream.

A different approach to adaptation is taken by the popular **Lempel-Ziv-Welch (LZW)** algorithm. This method was developed originally by Ziv and Lempel, and subsequently improved by Welch. As the message to be encoded is processed, the LZW algorithm builds a *string table* that maps symbol sequences to/from an N-bit index. The string table has 2^N entries and the transmitted code can be used at the decoder as an index into the string table to retrieve the corresponding original symbol sequence. The sequences stored in the table can be arbitrarily long. The algorithm is designed so that the string table can be reconstructed by the decoder based on information in the encoded stream—the table, while central to the encoding and decoding process, is never transmitted! This property is crucial to the understanding of the LZW method.

```
initialize TABLE[0 to 255] = code for individual bytes
STRING = get input symbol
while there are still input symbols:
   SYMBOL = get input symbol
   if STRING + SYMBOL is in TABLE:
       STRING = STRING + SYMBOL
   else:
       output the code for STRING
       add STRING + SYMBOL to TABLE
       STRING = SYMBOL
output the code for STRING
```

Figure 3-4: Pseudo-code for the LZW adaptive variable-length encoder. Note that some details, like dealing with a full string table, are omitted for simplicity.

```
initialize TABLE[0 to 255] = code for individual bytes
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING

while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined: // needed because sometimes the
        ENTRY = STRING + STRING[0] // decoder may not yet have entry!
    else:
        ENTRY = TABLE[CODE]
    output ENTRY
    add STRING+ENTRY[0] to TABLE
    STRING = ENTRY
```

Figure 3-5: Pseudo-code for LZW adaptive variable-length decoder.

When encoding a byte stream,[2] the first $2^8 = 256$ entries of the string table, numbered 0 through 255, are initialized to hold all the possible one-byte sequences. The other entries will be filled in as the message byte stream is processed. The encoding strategy works as follows and is shown in pseudo-code form in Figure 3-4. First, accumulate message bytes as long as the accumulated sequences appear as some entry in the string table. At some point, appending the next byte b to the accumulated sequence S would create a sequence $S + b$ that's not in the string table, where $+$ denotes appending b to S. The encoder then executes the following steps:

1. It transmits the N-bit code for the sequence S.
2. It adds a new entry to the string table for $S + b$. If the encoder finds the table full when it goes to add an entry, it reinitializes the table before the addition is made.
3. it resets S to contain only the byte b.

[2] A byte is a contiguous string of 8 bits.

S	msg. byte	lookup	result	transmit	string table
–	a	–	–	–	–
a	b	ab	not found	index of a	table[256] = ab
b	c	bc	not found	index of b	table[257] = bc
c	a	ca	not found	index of c	table[258] = ca
a	b	ab	found	–	–
ab	c	abc	not found	256	table[259] = abc
c	a	ca	found	–	–
ca	b	cab	not found	258	table[260] = cab
b	c	bc	found	–	–
bc	a	bca	not found	257	table[261] = bca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	not found	259	table[262] = abca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	not found	262	table[263] = abcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	not found	261	table[264] = bcab
b	c	bc	found	–	–
bc	a	bca	found	–	–
bca	b	bcab	found	–	–
bcab	c	bcabc	not found	264	table[265] = bcabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	not found	260	table[266] = cabc
c	a	ca	found	–	–
ca	b	cab	found	–	–
cab	c	cabc	found	–	–
cabc	a	cabca	not found	266	table[267] = cabca
a	b	ab	found	–	–
ab	c	abc	found	–	–
abc	a	abca	found	–	–
abca	b	abcab	found	–	–
abcab	c	abcabc	not found	263	table[268] = abcabc
c	– end –	–	–	index of c	–

Figure 3-6: LZW encoding of string "abcabcabcabcabcabcabcabcabcabcabc"

received	string table	decoding
a	–	a
b	table[256] = ab	b
c	table[257] = bc	c
256	table[258] = ca	ab
258	table[259] = abc	ca
257	table[260] = cab	bc
259	table[261] = bca	abc
262	table[262] = abca	abca
261	table[263] = abcab	bca
264	table[264] = bacb	bcab
260	table[265] = bcabc	cab
266	table[266] = cabc	cabc
263	table[267] = cabca	abcab
c	table[268] = abcabc	c

Figure 3-7: LZW decoding of the sequence $a, b, c, 256, 258, 257, 259, 262, 261, 264, 260, 266, 263, c$

This process repeats until all the message bytes are consumed, at which point the encoder makes a final transmission of the N-bit code for the current sequence S.

Note that for every transmission done by the encoder, the encoder makes a new entry in the string table. With a little cleverness, the decoder, shown in pseudo-code form in Figure 3-5, can figure out what the new entry must have been as it receives each N-bit code. With a *duplicate* string table at the decoder constructed as the algorithm progresses at the decoder, it is possible to recover the original message: just use the received N-bit code as index into the decoder's string table to retrieve the original sequence of message bytes.

Figure 3-6 shows the encoder in action on a repeating sequence of *abc*. Notice that:

- The encoder algorithm is greedy—it is designed to find the longest possible match in the string table before it makes a transmission.

- The string table is filled with sequences actually found in the message stream. No encodings are wasted on sequences not actually found in the file.

- Since the encoder operates without any knowledge of what's to come in the message stream, there may be entries in the string table that don't correspond to a sequence that's repeated, i.e., some of the possible N-bit codes will never be transmitted. This property means that the encoding isn't optimal—a prescient encoder could do a better job.

- Note that in this example the amount of compression increases as the encoding progresses, i.e., more input bytes are consumed between transmissions.

- Eventually the table will fill and then be reinitialized, recycling the N-bit codes for new sequences. So the encoder will eventually adapt to changes in the probabilities of the symbols or symbol sequences.

Figure 3-7 shows the operation of the decoder on the transmit sequence produced in Figure 3-6. As each N-bit code is received, the decoder deduces the correct entry to make in the string table (i.e., the same entry as made at the encoder) and then uses the N-bit code as index into the table to retrieve the original message sequence.

There is a special case, which turns out to be important, that needs to be dealt with. There are three instances in Figure 3-7 where the decoder receives an index $(262, 264, 266)$ that it has not previously entered in *its* string table. So how does it figure out what these correspond to? A careful analysis, which you could do, shows that this situation only happens when the associated string table entry has its last symbol identical to its first symbol. To handle this issue, the decoder can simply complete the partial string that it is building up into a table entry (abc, bac, cab respectively, in the three instances in Figure 3-7) by repeating its first symbol at the end of the string (to get abca, bacb, cabc respectively, in our example), and then entering this into the string table. This step is captured in the pseudo-code in Figure 3-5 by the logic of the "if" statement there.

We conclude this chapter with some interesting observations about LZW compression:

- A common choice for the size of the string table is 4096 ($N = 12$). A larger table means the encoder has a longer memory for sequences it has seen and increases the possibility of discovering repeated sequences across longer spans of message. However, dedicating string table entries to remembering sequences that will never be seen again decreases the efficiency of the encoding.

- Early in the encoding, the encoder uses entries near the beginning of the string table, i.e., the high-order bits of the string table index will be 0 until the string table starts to fill. So the N-bit codes we transmit at the outset will be numerically small. Some variants of LZW transmit a variable-width code, where the width grows as the table fills. If $N = 12$, the initial transmissions may be only 9 bits until entry number 511 in the table is filled (i.e., 512 entries filled in all), then the code expands to 10 bits, and so on, until the maximum width N is reached.

- Some variants of LZW introduce additional special transmit codes, e.g., CLEAR to indicate when the table is reinitialized. This allows the encoder to reset the table pre-emptively if the message stream probabilities change dramatically, causing an observable drop in compression efficiency.

- There are many small details we haven't discussed. For example, when sending N-bit codes one bit at a time over a serial communication channel, we have to specify the order in the which the N bits are sent: least significant bit first, or most significant bit first. To specify N, serialization order, algorithm version, etc., most compressed file formats have a header where the encoder can communicate these details to the decoder.

■ Exercises

1. Huffman coding is used to compactly encode the species of fish tagged by a game warden. If 50% of the fish are bass and the rest are evenly divided among 15 other species, how many bits would be used to encode the species when a bass is tagged?

2. Consider a Huffman code over four symbols, A, B, C, and D. Which of these is a valid Huffman encoding? Give a brief explanation for your decisions.

   (a) $A : 0, B : 11, C : 101, D : 100$.

   (b) $A : 1, B : 01, C : 00, D : 010$.

   (c) $A : 00, B : 01, C : 110, D : 111$

3. Huffman is given four symbols, A, B, C, and D. The probability of symbol A occurring is p_A, symbol B is p_B, symbol C is p_C, and symbol D is p_D, with $p_A \geq p_B \geq p_C \geq p_D$. Write down a single condition (equation or inequality) that is both necessary and sufficient to guarantee that, when Huffman constructs the code bearing his name over these symbols, each symbol will be encoded using exactly two bits. Explain your answer.

4. Describe the contents of the string table created when encoding a very long string of all a's using the simple version of the LZW encoder shown in Figure 3-4. In this example, if the decoder has received E encoded symbols (i.e., string table indices) from the encoder, how many a's has it been able to decode?

5. Consider the pseudo-code for the LZW decoder given in Figure 3-4. Suppose that this decoder has received the following five codes from the LZW encoder (these are the first five codes from a longer compression run):

   ```
   97 -- index of 'a' in the translation table
   98 -- index of 'b' in the translation table
   257 -- index of second addition to the translation table
   256 -- index of first addition to the translation table
   258 -- index of third addition to in the translation table
   ```

   After it has finished processing the fifth code, what are the entries in the translation table and what is the cumulative output of the decoder?

   table[256]: _____

   table[257]: _____

   table[258]: _____

   table[259]: _____

   cumulative output from decoder: _____

6. Consider the LZW compression and decompression algorithms as described in this chapter. Assume that the scheme has an initial table with code words 0 through 255 corresponding to the 8-bit ASCII characters; character "a" is 97 and "b" is 98. The receiver gets the following sequence of code words, each of which is 10 bits long:

   97 97 98 98 257 256

   (a) What was the original message sent by the sender?

(b) By how many bits is the compressed message shorter than the original message (each character in the original message is 8 bits long)?

(c) What is the first string of length 3 added to the compression table? (If there's no such string, your answer should be "None".)

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: October 17, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 4
Why Digital? Communication Abstractions and Digital Signaling

This chapter describes analog and digital communication, and the differences between them. Our focus is on understanding the problems with analog communication and the motivation for the digital abstraction. We then present basic recipes for sending and receiving digital data mapped to analog signals over communication links; these recipes are needed because physical communication links are fundamentally analog in nature at the lowest level. After understanding how bits get mapped to signals and vice versa, we will present our simple *layered communication model*: messages → packets → bits → signals. The rest of this book is devoted to understanding these different layers and how the interact with each other.

4.1 Sources of Data

The purpose of communication technologies is to empower users (be they humans or applications) to send messages to each other. We have already seen in Chapters 2 and 3 how to quantify the information content in messages, and in our discussion, we tacitly decided that our messages would be represented as sequences of binary digits (bits). We now discuss why that approach makes sense.

Some sources of data are *inherently digital* in nature; i.e., their natural and native representation is in the form of bit sequences. For example, data generated by computers, either with input from people or from programs ("computer-generated data") is natively encoded using sequences of bits. In such cases, thinking of our messages as bit sequences is a no-brainer.

There are other sources of data that are in fact inherently *analog* in nature. Prominent examples are video and audio. Video scenes and images captured by a camera lens encode information about the mix of colors (the proportions and intensities) in every part of the scene being captured. Audio captured by a microphone encodes information about the loudness (intensity) and frequency (pitch), varying in time. In general, one may view the data as coming from a *continuous space of values*, and the sensors capturing the raw

data may be thought of as being capable of producing *analog data* from this continuous space. In practice, of course, there is a measurement fidelity to every sensor, so the data captured will be quantized, but the abstraction is much closer to analog than digital. Other sources of data include sensors gathering information about the environment or device (e.g., accelerometers on your mobile phone, GPS sensors on mobile devices, or climate sensors to monitor weather conditions); these data sources could be inherently analog or inherently digital depending on what they're measuring.

Regardless of the nature of a source, converting the relevant data to *digital* form is the modern way; one sees numerous advertisements for "digital" devices (e.g., cameras), with the implicit message that somehow "digital" is superior to other methods or devices. The question is, why?

4.2 Why Digital?

There are two main reasons why digital communication (and more generally, building systems using the digital abstraction) is a good idea:

1. The digital abstraction enables the *composition of modules* to build large systems.
2. The digital abstaction allows us to us sophisticated algorithms to process data to improve the quality and performance of the components of a system.

Yet, the digital abstraction is not the natural way to communicate data. Physical communication links turn out to be analog at the lowest level, so we are going to have to convert data between digital and analog, and vice versa, as it traverses different parts of the system between the sender and the receiver.

4.2.1 Why analog is natural in many applications

To understand why the digital abstraction enables modularity and composition, let us first understand how analog representations of data work. Consider first the example of a black-and-white analog television image. Here, it is natural to represent each image as a sequence of values, one per (x, y) coordinate in a picture. The values represent the *luminance*, or "shade of gray": 0 volts represents "black", 1 volt represents "white", and any value x between 0 and 1 represents the fraction of white in the image (i.e., some shade of gray). The representation of the picture itself is as a sequence of these values in some scan order, and the transmission of the picture may be done by generating voltage waveforms to represent these values.

Another example is an analog telephone, which converts sound waves to electrical signals and back. Like the analog TV system, this system does not use bits (0s and 1s) to represent data (the voice conversation) between the communicating parties.

Such analog representations are tempting for communication applications because they map well to physical link capabilities. For example, when transmitting over a wire, we can send signals at different voltage levels, and the receiver can measure the voltage to determine what the sender transmitted. Over an optical communication link, we can send signals at different intensities (and possibly also at different wavelengths), and the receiver can measure the intensity to infer what the sender might have transmitted. Over radio

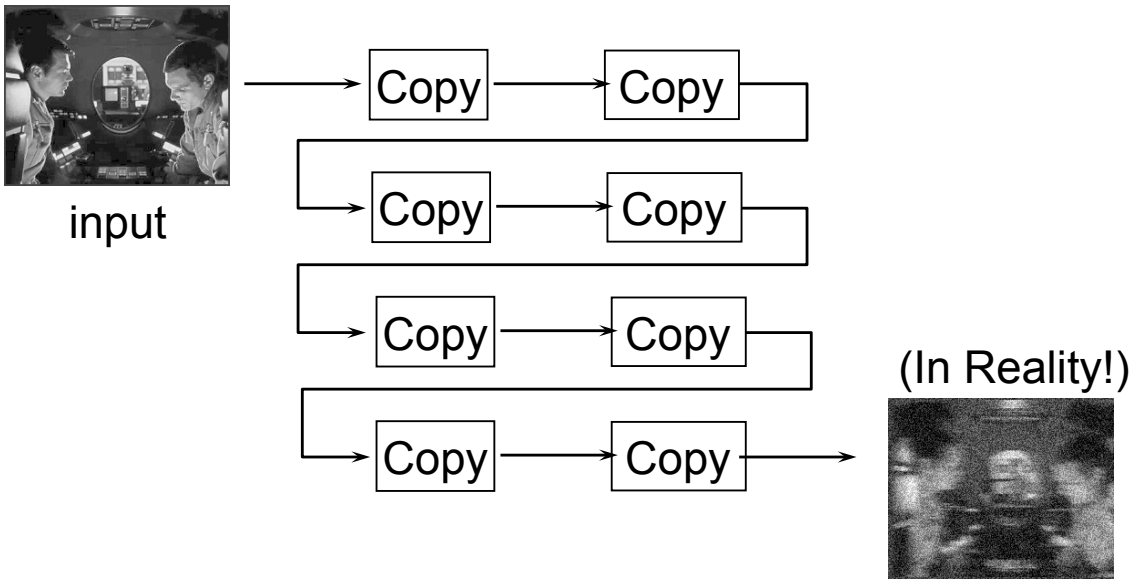


Figure 4-1: Errors accumulate in analog systems.

and acoustic media, the problem is trickier, but we can send different signals at different amplitudes "modulated" over a "carrier waveform" (as we will see in later chapters), and the receiver can measure the quantity of interest to infer what the sender might have sent.

■ 4.2.2 So why not analog?

Analog representations seem to map naturally to the inherent capabilities of communication links, so why not use them? The answer is that there is no error-free communication link. Every link suffers from perturbations, which may arise from noise (Chapter 5) or other sources of distortion. These perturbations affect the received signal; every time there is a transmission, the receiver will not get the transmitted signal exactly, but will get a perturbed version of it.

These perturbations have a cascading effect. For instance, if we have a series of COPY blocks that simply copy an incoming signal and re-send the copy, one will not get a perfect version of the signal, but a heavily perturbed version. Figure 4-1 illustrates this problem for a black-and-white analog picture sent over several COPY blocks. The problem is that when an analog input value, such as a voltage of 0.12345678 volts is put into the COPY block, the output is not the same, but something that might be 0.12?????? volts, where the "?" refers to incorrect values.

There are many reasons why the actual output differs from the input, including the manufacturing tolerance of internal components, environmental factors (temperature, power supply voltage, etc.), external influences such as interference from other transmissions, and so on. There are many sources, which we can collectively think of as "noise", for now. In later chapters, we will divide these perturbations into random components ("noise") and other perturbations that can be modeled deterministically.

These analog errors accumulate, or cascade. If the output value is $V_{in} \pm \varepsilon$ for an input value of V_{in}, then placing a number N of such units in series will make the output value

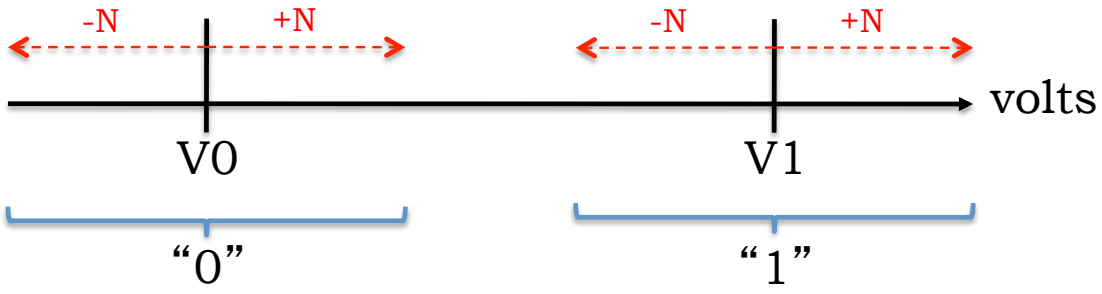


Figure 4-2: If the two voltages are adequately spaced apart, we can tolerate a certain amount of noise.

be $V_{in} \pm N\varepsilon$. If $\varepsilon = 0.01$ and $N = 100$, the output may be off by 100%!

As system engineers, we want modularity, which means we want to guarantee output values without having to worry about the details of the innards of various components. Hence, we need to figure out a way to eliminate, or at least reduce, errors at each processing stage.

The digital signaling abstraction provides us a way to achieve this goal.

■ 4.3 Digital Signaling: Mapping Bits to Signals

To ensure that we can distinguish signal from noise, we will **map bits to signals** using a fixed set of discrete values. The simplest way to do that is to use a *binary mapping* (or binary signaling) scheme. Here, we will use two voltages, V_0 volts to represent the bit “0” and V_1 volts to represent the bit “1”.

What we want is for received voltages near V_0 to be interpreted as representing a “0”, and for received voltages near V_0 to be interpreted as representing a “1”. If we would like our mapping to work reliably up to a certain amount of noise, then we need to space V_0 and V_1 far enough apart so that even noisy signals are interpreted correctly. An example is shown in Figure 4-2.

At the receiver, we can specify the behavior wih a graph that shows how incoming voltages are mapped to bits “0” and “1” respectively (Figure 4-3. This idea is intuitive: we pick the intermediate value, $V_{th} = \frac{V_0+V_1}{2}$ and declare received voltages $\leq V_{th}$ as bit “0” and all other received voltage values as bit “1”. In Chapter 5, we will see when this rule is optimal and when it isn’t, and how it can be improved when it isn’t the optimal rule. (We’ll also see what we mean by “optimal” by relating optimality to the probability of reporting the value of the bit wrongly.)

We note that it would actually be rather difficult to build a receiver that *precisely* met this specification because measuring voltages extremely accurately near V_{th} will be extremely expensive. Fortunately, we don’t need to worry too much about such values if the values V_0 and V_1 are spaced far enough apart given the noise in the system. (See the bottom picture in Figure 4-3.)

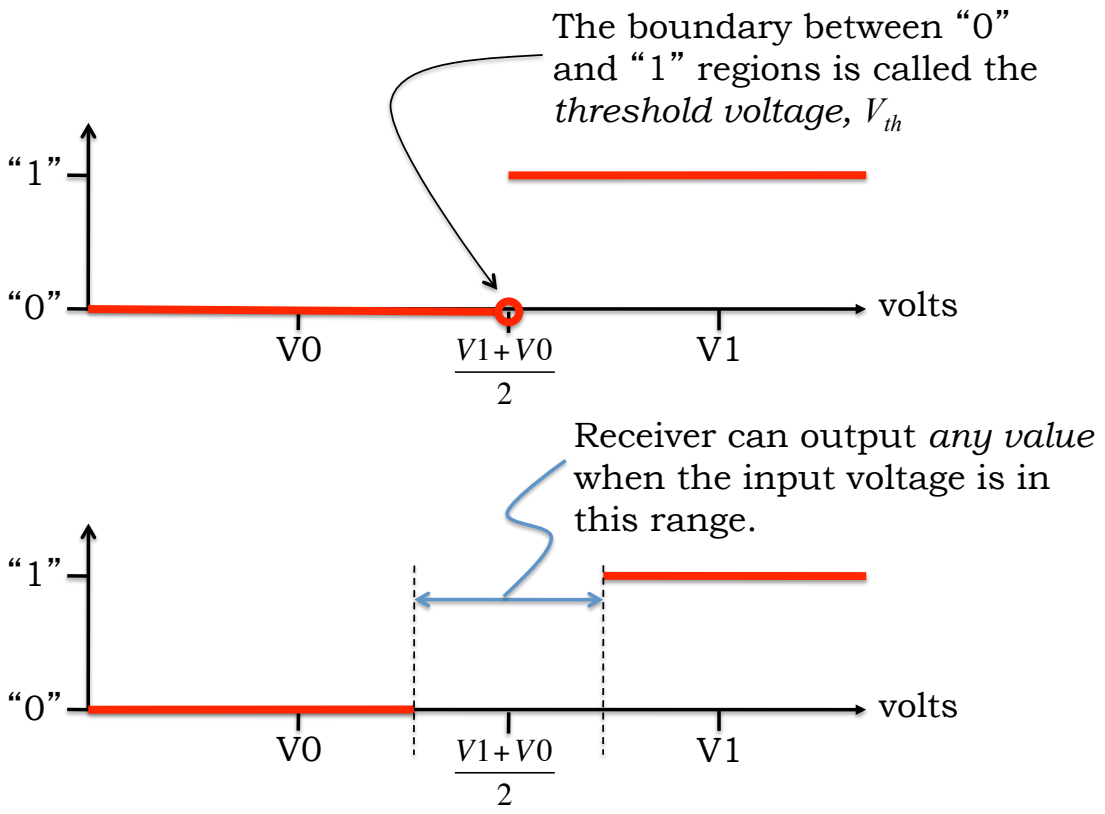


Figure 4-3: Picking a simple threshold voltage.

■ 4.3.1 Signals in this Course

Each individual transmission signal is conceptually a *fixed-voltage waveform* held for some period of time. So, to send bit "0", we will transmit a signal of fixed-voltage V_0 volts for a certain period of time; likewise, V_1 volts to send bit "1". We will represent these continuous-time signals using sequences of discrete-time *samples*. The *sample rate* is defined as the number of samples per second used in the system; the sender and receiver at either end of a communication link will agree on this sample rate in advance. (Each link could of course have a different sample rate.) The reciprocal of the sample rate is the *sample interval*, which is the time between successive samples. For example, 4 million samples per second implies a sample interval o 0.25 microseconds.

An example of the relation between continuous-time fixed-voltage waveforms (and how they relate to individual bits) and the sampling process is shown in Figure 4-4.

■ 4.3.2 Clocking Transmissions

Over a communication link, the sender and receiver need to agree on a *clock rate*. The idea is that periodic events are timed by a clock signal, as shown in Figure 4-5 (top picture). Each new bit is sent when a clock transition occurs, and each bit has many samples, sent at a regular rate. We will use the term *samples_per_bit* to refer to the number of discrete voltage samples sent for any given bit. All the samples for any given bit will of course be

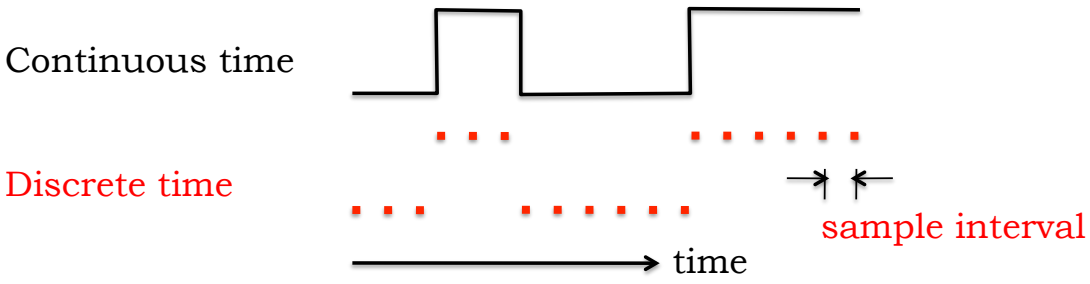


Figure 4-4: Sampling continuous-time voltage waveforms for transmission.

sent at the same voltage value.

How does the receiver recover the data that was sent? If we sent only the samples and not the clock, how can the receiver figure out what was sent?

The idea is for the receiver to infer the presence of a clock edge every time there is a transition in the received samples (Figure 4-5, bottom picture). Them using the shared knowledge of the sample rate (or sample interval), the receiver can extrapolate the remaining edges and infer the first and last sample for each bit. It can then choose the middle sample to determine the message bit, or more robustly average them all to estimate the bit.

There are two problems that need to be solved for this approach to work:

1. How to cope with differences in the sender and receiver clock frequencies?

2. How to ensure frequent transitions between 0s and 1s?

The first problem is one of clock and data recovery. The second is solved using *line coding*, of which 8b/10b coding is a common scheme. The idea is to convert groups of bits into different groups of bits that have frequent 0/1 transitions. We describe these two ideas in the next two sections. We also refer the reader to the two lab tasks in Problem Set 2, which describe these two issues and their implementation in considerable detail.

4.4 Clock and Data Recovery

In a perfect world, it would be a trivial task to find the voltage sample in the middle of each bit transmission and use that to determine the transmitted bit, or take the average. Just start the sampling index at samples_per_bit/2, then increase the index by samples_per_bit to move to the next voltage sample, and so on until you run out of voltage samples.

Alas, in the real world things are a bit more complicated. Both the transmitter and receiver use an internal clock oscillator running at the sample rate to determine when to generate or acquire the next voltage sample. And they both use counters to keep track of how many samples there are in each bit. The complication is that the frequencies of the transmitter's and receiver's clock may not be exactly matched. Say the transmitter is sending 5 voltage samples per message bit. If the receiver's clock is a little slower, the transmitter will seem to be transmitting faster, e.g., transmitting at 4.999 samples per bit.

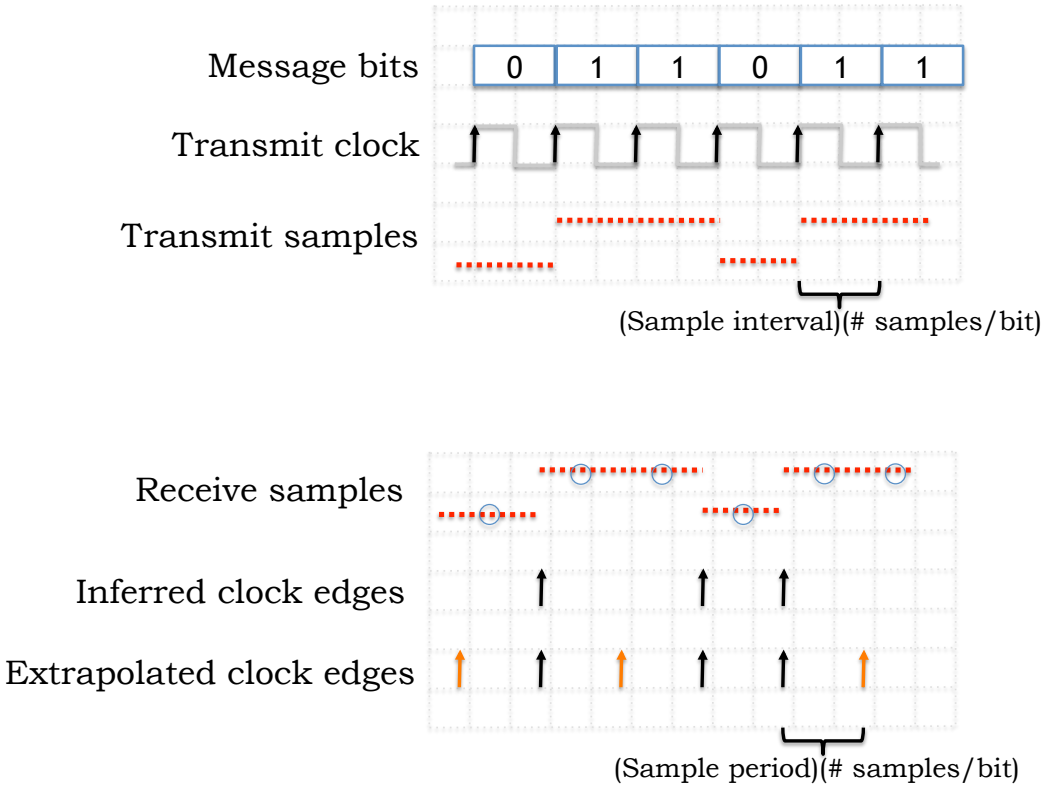


Figure 4-5: Transmission using a clock (top) and inferring clock edges from bit transitions between 0 and 1 and vice versa at the receiver (bottom).

Similarly, if the receiver's clock is a little faster, the transmitter will seem to be transmitting slower, e.g., transmitting at 5.001 samples per bit. This small difference accummulates over time, so if the receiver uses a static sampling strategy like the one outlined in the previous paragraph, it will eventually be sampling right at the transition points between two bits. And to add insult to injury, the difference in the two clock frequencies will change over time.

The fix is to have the receiver adapt the timing of it's sampling based on where it detects transitions in the voltage samples. The transition (when there is one) should happen half-way between the chosen sample points. Or to put it another way, the receiver can look at the voltage sample half-way between the two sample points and if it doesn't find a transition, it should adjust the sample index appropriately.

Figure 4-6 illustrates how the adaptation should work. The examples use a low-to-high transition, but the same strategy can obviously be used for a high-to-low transition. The two cases shown in the figure differ in value of the sample that's half-way between the current sample point and the previous sample point. Note that a transition has occurred when two consecutive sample points represent different bit values.

- Case 1: the half-way sample is the same as the current sample. In this case the half-way sample is in the same bit transmission as the current sample, i.e., we're sampling too late in the bit transmission. So when moving to the next sample, increment the

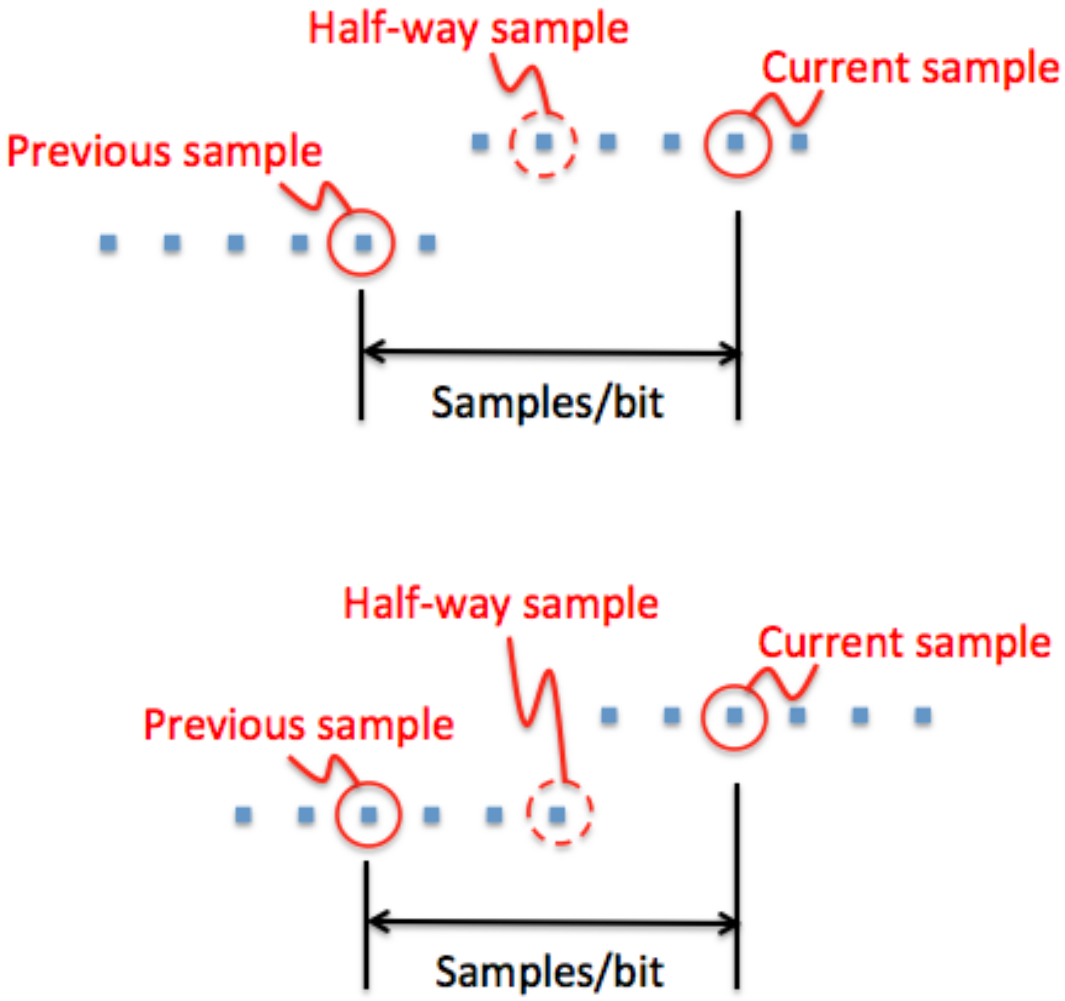


Figure 4-6: The two cases of how the adaptation should work.

index by samples_per_bit - 1 to move "back".

- Case 2: the half-way sample is different than the current sample. In this case the half-way sample is in the previous bit transmission from the current sample, i.e., we're sampling too early in the bit transmission. So when moving to the next sample, increment the index by samples_per_bit + 1 to move "forward"

If there is no transition, simply increment the sample index by samples_per_bit to move to the next sample. This keeps the sampling position approximately right until the next transition provides the information necessary to make the appropriate adjustment.

If you think about it, when there is a transition, one of the two cases above will be true and so we'll be constantly adjusting the relative position of the sampling index. That's fine – if the relative position is close to correct, we'll make the opposite adjustment next time. But if a large correction is necessary, it will take several transitions for the correction to happen. To facilitate this initial correction, in most protocols the transmission of message

begins with a training sequence of alternating 0- and 1-bits (remember each bit is actually samples_per_bit voltage samples long). This provides many transitions for the receiver's adaptation circuity to chew on.

4.5 Line Coding with 8b/10b

Line coding, using a scheme like 8b/10b, was developed to help address the following issues:

- For electrical reasons it's desirable to maintain DC balance on the wire, i.e., that on the average the number of 0's is equal to the number of 1's.
- Transitions in the received bits indicate the start of a new bit and hence are useful in synchronizing the sampling process at the receiver—the better the synchronization, the faster the maximum possible symbol rate. So ideally one would like to have frequent transitions. On the other hand each transition consumes power, so it would be nice to minimize the number of transitions consistent with the synchronization constraint and, of course, the need to send actual data! In a signaling protocol where the transitions are determined by the message content may not achieve these goals.

To address these issues we can use an encoder (called the "line coder") at the transmitter to recode the message bits into a sequence that has the properties we want, and use a decoder at the receiver to recover the original message bits. Many of today's high-speed data links (e.g., PCI-e and SATA) use an 8b/10b encoding scheme developed at IBM. The 8b/10b encoder converts 8-bit message symbols into 10 transmitted bits. There are 256 possible 8-bit words and 1024 possible 10-bit transmit symbols, so one can choose the mapping from 8-bit to 10-bit so that the the 10-bit transmit symbols have the following properties:

- The maximum run of 0's or 1's is five bits (i.e., there is at least one transition every five bits).
- At any given sample the maximum difference between the number of 1's received and the number of 0's received is six.
- Special 7-bit sequences can be inserted into the transmission that don't appear in any consecutive sequence of encoded message bits, even when considering sequences that span two transmit symbols. The receiver can do a bit-by-bit search for these unique patterns in the incoming stream and then know how the 10-bit sequences are aligned in the incoming stream.

Here's how the encoder works: collections of 8-bit words are broken into groups of words called a packet. Each packet is sent using the following wire protocol:

- A sequence of alternating 0 bits and 1 bits are sent first (recall that each bit is multiple voltage samples). This sequence is useful for making sure the receiver's clock recovery machinery has synchronized with the transmitter's clock. These bits aren't part of the message; they're there just to aid in clock recovery.

- A SYNC pattern—usually either 0011111 or 1100000 where the least-significant bit (LSB) is shown on the left—is transmitted so that the receiver can find the beginning of the packet.[1] Traditionally, the SYNC patterns are transmitted least-significant bit (LSB) first. The reason for the SYNC is that if the transmitter is sending bits continuously and the receiver starts listening at some point in the transmission, there's no easy way to locate the start of multi-bit symbols. By looking for a SYNC, the receiver can detect the start of a packet. Of course, care must be taken to ensure that a SYNC pattern showing up in the middle of the packet's contents don't confuse the receiver (usually that's handled by ensuring that the line coding scheme does not produce a SYNC pattern, but it is possible that bit errors can lead to such confusion at the receiver).
- Each byte (8 bits) in the packet data is line-coded to 10 bits and sent. Each 10-bit transmit symbol is determined by table lookup using the 8-bit word as the index. Note that all 10-bit symbols are transmitted least-significant bit (LSB) first. If the length of the packet (without SYNC) is s bytes, then the resulting size of the line-coded portion is $10s$ bits, to which the SYNC is added.

Multiple packets are sent until the complete message has been transmitted. Note that there's no particular specification of what happens between packets – the next packet may follow immediately, or the transmitter may sit idle for a while, sending, say, training sequence samples.

If the original data in a single packet is s bytes long, and the SYNC is h bits long, then the total number of bits sent is equal to $10s + h$. The "rate" of this line code, i.e., the ratio of the number of useful message bits to the total bits sent, is therefore equal to $\frac{8s}{10s+h}$. (We will properly define the concept of "code rate" in Chapter 6 more.) If the communication link is operating at R bits per second, then the rate at which useful message bits arrive is given by $\frac{8s}{10s+h} \cdot R$ bits per second with 8b/10b line coding.

■ 4.6 Communication Abstractions

Figure 4-7 shown the overall system context, tying together the concepts of the previous chapters with the rest of this book. The rest of this book is about the oval labeled "COMMUNICATION NETWORK". The simplest example of a communication network is a single physical communication link, which we start with.

At either end of the communication link are various modules, as shown in Figure 4-8. One of these is a *Mapper*, which maps bits to signals and arranges for samples to be transmitted. There is a counterpart *Demapper* at the receiving end. As shown in Figure 4-8 is a *Channel coding* module, and a counterpart *Channel decoding* module, which handle errors in transmission caused by noise.

In addition, a message, produced after source coding from the original data source, may have to be broken up into multiple packets, and sent over multiple links before reaching the receiving application or user. Over each link, three abstractions are used: *packets*, *bits*, and *signals* (Figure 4-8 bottom). Hence, it is convenient to think of the problems in data communication as being in one of these three "layers", which are one on top of the other

[1] In general any other SYNC pattern could also be sent.

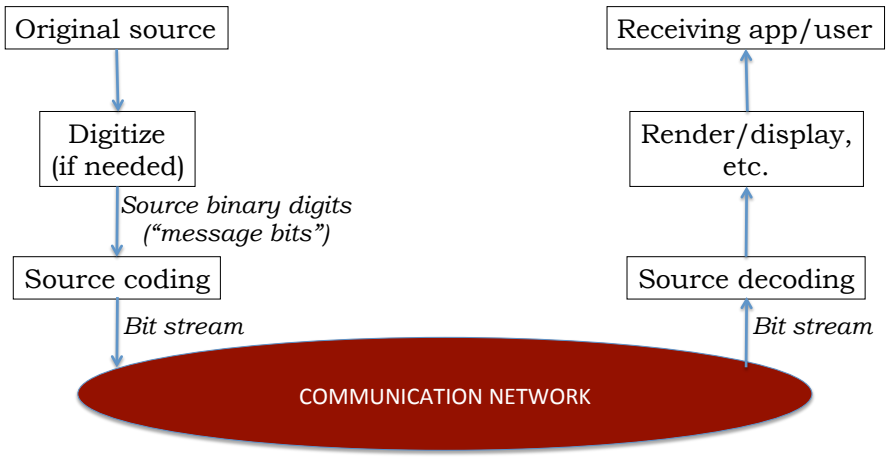


Figure 4-7: The "big picture".

(packets, bits, and signals). The rest of this book is about these three important abstractions and how they work together. We do them in the order bits, signals, and packets, for convenience and ease of exposition and understanding.

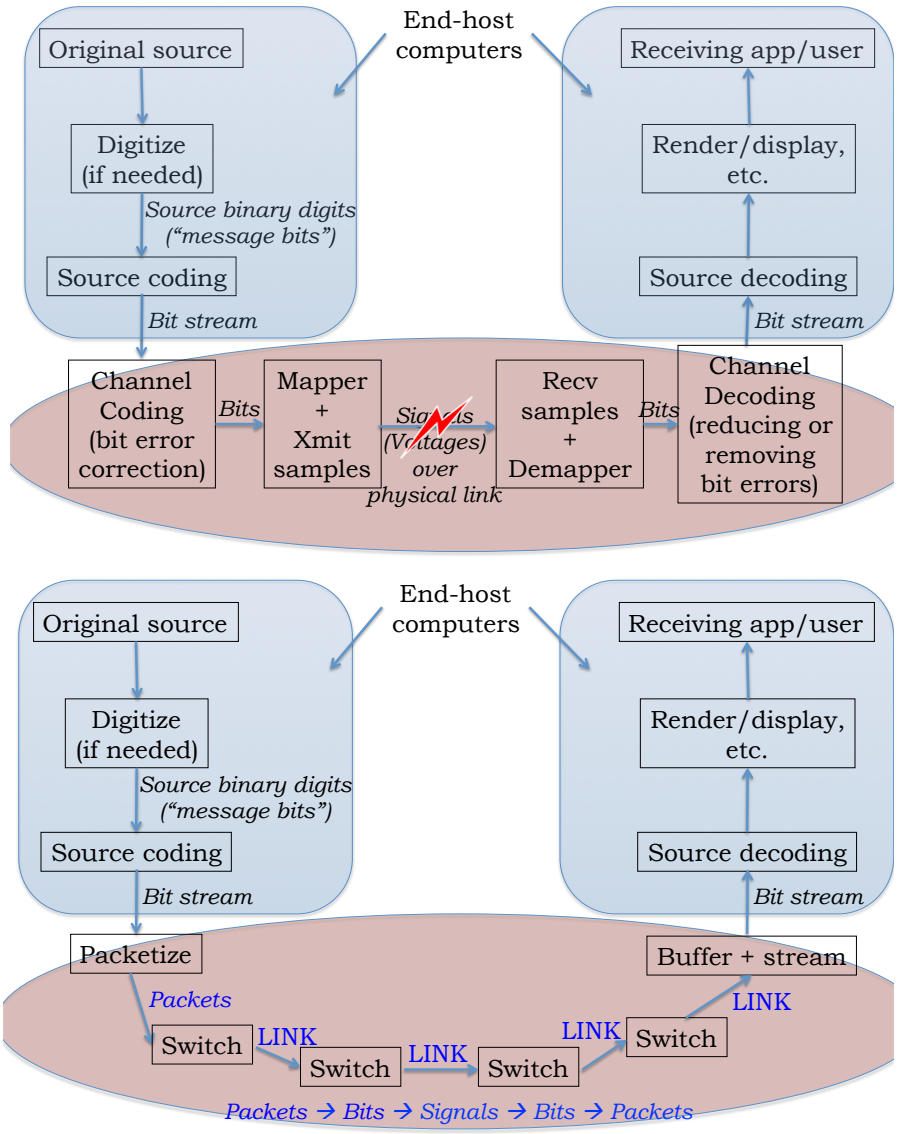


Figure 4-8: Expanding on the "big picture": single link view (top) and the network view (bottom).

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: October 14, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 5
Coping with Bit Errors using Error Correction Codes

Recall our main goal in designing digital communication networks: to send information both reliably and efficiently between nodes. Meeting that goal requires the use of techniques to combat bit errors, which are inevitable in both commmunication channels and storage media.

The key idea we will apply to achieve reliable communication is the addition of *redundancy* to the transmitted data, to improve the probability that the original message can be reconstructed from the possibly corrupted data that is received. The sender has an *encoder* whose job is to take the message and process it to produce the *coded bits* that are then sent over the channel. The receiver has a *decoder* whose job is to take the received (coded) bits and to produce its best estimate of the message. The encoder-decoder procedures together constitute **channel coding**; good channel codes provide **error correction** capabilities that reduce the bit error rate (i.e., the probability of a bit error).

With proper design, full error correction may be possible, provided only a small number of errors has occurred. Even when too many errors have occurred to permit correction, it may be possible to perform *error detection*. Error detection provides a way for the receiver to tell (with high probability) if the message was decoded correctly or not. Error detection usually works by the sender and receiver using a different code from the one used to correct errors; common examples include the *cyclic redundancy check* (CRC) or *hash functions*. These codes take n-bit messages and produce a compact "signature" of that message that is much smaller than the message (e.g., the popular CRC-32 scheme produces a 32-bit signature of an arbitrarily long message). The sender computes and transmits the signature along with the message bits, usually appending it to the end of the message. The receiver, after running the decoder to correct errors, then computes the signature over its estimate of the message bits and compares that signature to its estimate of the signature bits in the received data. If the computed and estimated signatures are not equal, then the receiver considers the message to have one or more bit errors; otherwise, it assumes that the message has been received correctly. This latter assumption is probabilistic: there is some non-zero (though very small, for good signatures) probability that the estimated and computed signatures match, but the receiver's decoded message is different from the

sender's. If the signatures don't match, the receiver and sender may use some higher-layer protocol to arrange for the message to be retransmitted; we will study such schemes later. We will not study error detection codes like CRC or hash functions in this course.

Our plan for this chapter is as follows. To start, we will assume a binary symmetric channel (BSC), which we defined and explained in the previous chapter; here the probability of a bit "flipping" is ε. Then, we will discuss and analyze a simple redundancy scheme called a *replication code*, which will simply make n copies of any given bit. The replication code has a *code rate* of $1/n$—that is, for every useful *message* bit, we end up transmitting n total bits. The overhead of the replication code of rate c is $1 - 1/n$, which is rather high for the error correcting power of the code. We will then turn to the key ideas that allow us to build powerful codes capable of correcting errors without such a high overhead (or equivalently, capable of correcting far more errors at a given code rate compared to the replication code).

There are two big, inter-related ideas used in essentially all error correction codes. The first is the notion of **embedding**, where the messages one wishes to send are placed in a geometrically pleasing way in a larger space so that the distance between any two valid points in the embedding is large enough to enable the correction and detection of errors. The second big idea is to use **parity calculations**, which are linear functions over the bits we wish to send, to generate the redundancy in the bits that are actually sent. We will study examples of embeddings and parity calculations in the context of two classes of codes: **linear block codes**, which are an instance of the broad class of **algebraic codes**, and **convolutional codes**, which are perhaps the simplest instance of the broad class of **graphical codes**.

We start with a brief discussion of bit errors.

■ 5.1 Bit Errors

A BSC is characterized by one parameter, ε, which we can assume to be $< 1/2$, the probability of a bit error. It is a natural discretization of AWGN, which is also a single-parameter model fully characterized by the variance, σ^2. We can determine ε empirically by noting that if we send N bits over the channel, the expected number of erroneously received bits is $N \cdot \varepsilon$. By sending a long known bit pattern and counting the fraction or erroneously received bits, we can estimate ε, thanks to the *law of large numbers*. In practice, even when the BSC is a reasonable error model, the range of ε could be rather large, between 10^{-2} (or even a bit higher) all the way to 10^{-10} or even 10^{-12}. A value of ε of about 10^{-2} means that messages longer than a 100 bits will see at least one error on average; given that the typical unit of communication over a channel (a "packet") is generally between 500 bits and 12000 bits (or more, in some networks), such an error rate is too high.

But is a ε of 10^{-12} small enough that we don't need to bother about doing any error correction? The answer often depends on the data rate of the channel. If the channel has a rate of 10 Gigabits/s (available today even on commodity server-class computers), then the "low" ε of 10^{-12} means that the receiver will see one error every 10 seconds on average if the channel is continuously loaded. Unless we include some mechanisms to mitigate the situation, the applications using the channel may find errors occurring too frequently. On the other hand, a ε of 10^{-12} may be fine over a communication channel running at 10

Megabits/s, as long as there is some way to detect errors when they occur.

The BSC is perhaps the simplest error model that is realistic, but real-world channels exhibit more complex behaviors. For example, over many wireless and wired channels as well as on storage media (like CDs, DVDs, and disks), errors can occur in *bursts*. That is, the probability of any given bit being received wrongly depends on recent history: the probability is higher if the bits in the recent past were received incorrectly. Our goal is to develop techniques to mitigate the effects of both the BSC and burst errors. We'll start with techniques that work well over a BSC and then discuss how to deal with bursts.

5.2 The Simplest Code: Replication

In general, a channel code provides a way to map message words to codewords (analogous to a source code, except here the purpose is not compression but rather the addition of redundancy for error correction or detection). In a replication code, each bit b is encoded as n copies of b, and the result is delivered. If we consider bit b to be the *message word*, then the corresponding *codeword* is b^n (i.e., $bb\ldots b$, n times). In this example, there are only two possible message words (0 and 1) and two corresponding codewords. The replication code is absurdly simple, yet it's instructive and sometimes even useful in practice!

But how well does it correct errors? To answer this question, we will write out the probability of overcoming channel errors for the BSC error model with the replication code. That is, if the channel corrupts each bit with probability ε, what is the probability that the receiver decodes the received codeword correctly to produce the message word that was sent?

The answer depends on the decoding method used. A reasonable decoding method is *maximum likelihood decoding*: given a received codeword, r, which is some n-bit combination of 0's and 1's, the decoder should produce the most likely message that could have caused r to be received. Since the BSC error probability, ε, is smaller than $1/2$, the most likely option is the codeword that has the most number of bits in common with r. This decoding rule results in the minimum probability of error when all messages are equally likely.

Hence, the decoding process is as follows. First, count the number of 1's in r. If there are more than $c/2$ 1's, then decode the message as 1. If there are more than $c/2$ 0's, then decode the message as 0. When c is odd, each codeword will be decoded unambiguously. When c is even, and has an equal number of 0's and 1's, the decoder can't really tell whether the message was a 0 or 1, and the best it can do is to make an arbitrary decision. (We have tacitly assumed that the *a priori* probability of sending a message 0 is the same as that of sending a 1.)

We can write the probability of decoding error for the replication code as follows, being a bit careful with the limits of the summation:

$$P(\text{decoding error}) = \begin{cases} \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} & \text{if } n \text{ odd} \\ \sum_{i=\frac{n}{2}+1}^{n} \binom{n}{i} \varepsilon^i (1-\varepsilon)^{n-i} + \frac{1}{2}\binom{n}{n/2} \varepsilon^{n/2} (1-\varepsilon)^{n/2} & \text{if } n \text{ even} \end{cases} \tag{5.1}$$

The notation $\binom{n}{i}$ denotes the number of ways of selecting i objects (in this case, bit positions) from n objects.

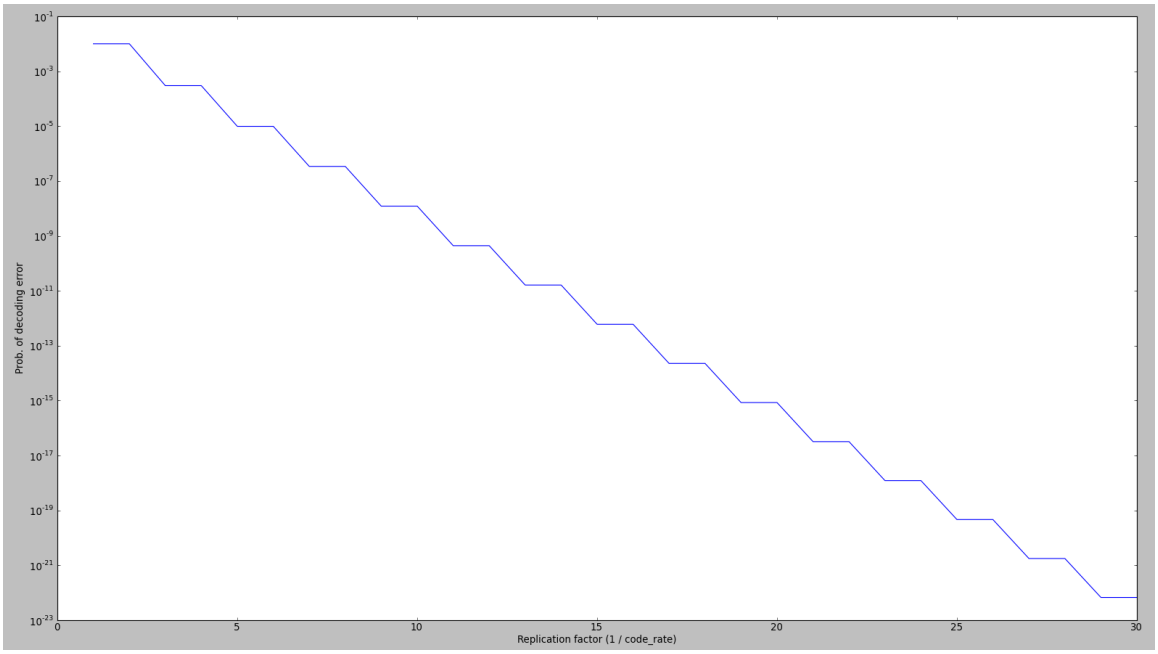


Figure 5-1: Probability of a decoding error with the replication code that replaces each bit b with n copies of b. The code rate is $1/n$.

When n is even, we add a term at the end to account for the fact that the decoder has a fifty-fifty chance of guessing correctly when it receives a codeword with an equal number of 0's and 1's.

Figure 5-1 shows the probability of decoding error as a function of the replication factor, n, for the replication code, computed using Equation (5.1). The y-axis is on a log scale, and the probability of error is more or less a straight line with negative slope (if you ignore the flat pieces), which means that the decoding error probability decreases exponentially with the code rate. It is also worth noting that the error probability is the same when $n = 2\ell$ as when $n = 2\ell - 1$. The reason, of course, is that the decoder obtains no additional information that it already didn't know from any $2\ell - 1$ of the received bits.

Despite the exponential reduction in the probability of decoding error as n increases, the replication code is extremely inefficient in terms of the overhead it incurs, for a given rate, $1/n$. As such, it is used only in situations when bandwidth is plentiful and there isn't much computation time to implement a more complex decoder.

We now turn to developing more sophisticated codes. There are two big related ideas: *embedding messages into spaces in a way that achieves structural separation* and *parity (linear) computations over the message bits*.

■ 5.3 Embeddings and Hamming Distance

Let's start our investigation into error correction by examining the situations in which error detection and correction are possible. For simplicity, we will focus on single-error correction (SEC) here. By that we mean codes that are guaranteed to produce the correct message word, given a received codeword with zero or one bit errors in it. If the received

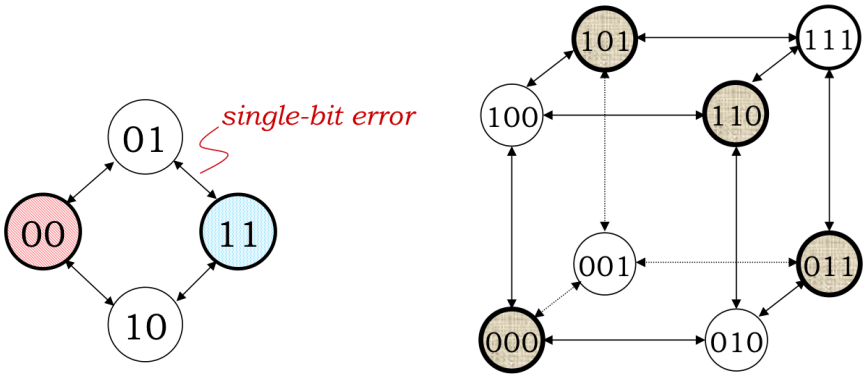


Figure 5-2: Codewords separated by a Hamming distance of 2 can be used to detect single bit errors. The codewords are shaded in each picture. The picture on the left is a (2,1) repetition code, which maps 1-bit messages to 2-bit codewords. The code on the right is a (3,2) code, which maps 2-bit messages to 3-bit codewords.

codeword has more than one bit error, then we can make no guarantees (the method might return the correct message word, but there is at least one instance where it will return the wrong answer).

There are 2^n possible n-bit strings. Define the *Hamming distance* (HD) between two n-bit words, w_1 and w_2, as the number of bit positions in which the messages differ. Thus $0 \leq \text{HD}(w_1, w_2) \leq n$.

Suppose that $\text{HD}(w_1, w_2) = 1$. Consider what happens if we transmit w_1 and there's a single bit error that inconveniently occurs at the one bit position in which w_1 and w_2 differ. From the receiver's point of view it just received w_2—the receiver can't detect the difference between receiving w_1 with a unfortunately placed bit error and receiving w_2. In this case, we cannot guarantee that all single bit errors will be corrected if we choose a code where w_1 and w_2 are both valid codewords.

What happens if we increase the Hamming distance between any two valid codewords to 2? More formally, let's restrict ourselves to only sending some subset $\mathcal{S} = \{w_1, w_2, \ldots, w_s\}$ of the 2^n possible words such that

$$\text{HD}(w_i, w_j) \geq 2 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.2}$$

Thus if the transmission of w_i is corrupted by a single error, the result is *not* an element of $\mathcal{S}$ and hence can be detected as an erroneous reception by the receiver, which knows which messages are elements of $\mathcal{S}$. A simple example is shown in Figure 5-2: 00 and 11 are valid codewords, and the receptions 01 and 10 are surely erroneous.

We define the *minimum Hamming distance of a code* as the minimum Hamming distance between any two codewords in the code. From the discussion above, it should be easy to see what happens if we use a code whose minimum Hamming distance is D. We state the property formally:

Theorem 5.1 *A code with a minimum Hamming distance of D can detect* any *error pattern of $D-1$ or fewer errors. Moreover, there is at least one error pattern with D errors that cannot be detected reliably.*

Hence, if our goal is to detect errors, we can use an embedding of the set of messages we wish to transmit into a bigger space, so that the minimum Hamming distance between any two codewords in the bigger space is at least one more than the number of errors we wish to detect. (We will discuss how to produce such embeddings in the subsequent sections.)

But what about the problem of *correcting* errors? Let's go back to Figure 5-2, with $\mathcal{S} = \{00, 11\}$. Suppose the received sequence is 01. The receiver can tell that a single error has occurred, but it can't tell whether the correct data sent was 00 or 11—both those possible patterns are equally likely under the BSC error model.

Ah, but we can extend our approach by producing an embedding with more space between valid codewords! Suppose we limit our selection of messages in $\mathcal{S}$ even further, as follows:

$$\text{HD}(w_i, w_j) \geq 3 \text{ for all } w_i, w_j \in \mathcal{S} \text{ where } i \neq j \tag{5.3}$$

How does it help to increase the minimum Hamming distance to 3? Let's define one more piece of notation: let $\mathcal{E}_{w_i}$ be the set of messages resulting from corrupting w_i with a single error. For example, $\mathcal{E}_{000} = \{001, 010, 100\}$. Note that HD($w_i$, an element of $\mathcal{E}_{w_i}$) = 1.

With a minimum Hamming distance of 3 between the valid codewords, observe that there is no intersection between $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$ when $i \neq j$. Why is that? Suppose there was a message w_k that was in both $\mathcal{E}_{w_i}$ and $\mathcal{E}_{w_j}$. We know that HD(w_i, w_k) = 1 and HD(w_j, w_k) = 1, which implies that w_i and w_j differ in at most two bits and consequently HD(w_i, w_j) ≤ 2. (This result is an application of Theorem 5.2 below, which states that the Hamming distance satisfies the triangle inequality.) That contradicts our specification that their minimum Hamming distance be 3. So the $\mathcal{E}_{w_i}$ don't intersect.

So now we can *correct* single bit errors as well: the received message is either a member of $\mathcal{S}$ (no errors), or is a member of some particular $\mathcal{E}_{w_i}$ (one error), in which case the receiver can deduce the original message was w_i. Here's another simple example: let $\mathcal{S} = \{000, 111\}$. So $\mathcal{E}_{000} = \{001, 010, 100\}$ and $\mathcal{E}_{111} = \{110, 101, 011\}$ (note that $\mathcal{E}_{000}$ doesn't intersect $\mathcal{E}_{111}$). Suppose the received sequence is 101. The receiver can tell there has been a single error because $101 \notin \mathcal{S}$. Moreover it can deduce that the original message was most likely 111 because $101 \in \mathcal{E}_{111}$.

We can formally state some properties from the above discussion, and specify the error-correcting power of a code whose minimum Hamming distance is D.

Theorem 5.2 *The Hamming distance between n-bit words satisfies the triangle inequality. That is, HD(x, y) + HD$(y, z) \geq$ HD(x, z).*

Theorem 5.3 *For a BSC error model with bit error probability $< 1/2$, the maximum likelihood decoding strategy is to map any received word to the valid codeword with smallest Hamming distance from the received one (ties may be broken arbitrarily).*

Theorem 5.4 *A code with a minimum Hamming distance of D can correct* any *error pattern of $\lfloor \frac{D-1}{2} \rfloor$ or fewer errors. Moreover, there is at least one error pattern with $\lfloor \frac{D-1}{2} \rfloor + 1$ errors that cannot be corrected reliably.*

Equation (5.3) gives us a way of determining if single-bit error correction can always be performed on a proposed set $\mathcal{S}$ of transmission messages—we could write a program to compute the Hamming distance between all pairs of messages in $\mathcal{S}$ and verify that the

minimum Hamming distance was at least 3. We can also easily generalize this idea to check if a code can always correct more errors. And we can use the observations made above to decode any received word: just find the closest valid codeword to the received one, and then use the known mapping between each distinct message and the codeword to produce the message. The message will be the correct one if the actual number of errors is no larger than the number for which error correction is guaranteed. The check for the nearest codeword may be exponential in the number of message bits we would like to send, making it a reasonable approach only if the number of bits is small.

But how do we go about finding a good embedding (i.e., good code words)? This task isn't straightforward, as the following example shows. Suppose we want to reliably send 4-bit messages so that the receiver can correct all single-bit errors in the received words. Clearly, we need to find a set of messages $\mathcal{S}$ with 2^4 elements. What should the members of $\mathcal{S}$ be?

The answer isn't obvious. Once again, we could write a program to search through possible sets of n-bit messages until it finds a set of size 16 with a minimum Hamming distance of 3. An exhaustive search shows that the minimum n is 7, and one example of $\mathcal{S}$ is:

0000000	1100001	1100110	0000111
0101010	1001011	1001100	0101101
1010010	0110011	0110100	1010101
1111000	0011001	0011110	1111111

But such exhaustive searches are impractical when we want to send even modestly longer messages. So we'd like some constructive technique for building $\mathcal{S}$. Much of the theory and practice of coding is devoted to finding such constructions and developing efficient encoding and decoding strategies.

Broadly speaking, there are two classes of code constructions, each with an enormous number of example instances. The first is the class of **algebraic block codes**. The second is the class of **graphical codes**. We will study two simple examples of **linear block codes**, which themselves are a sub-class of algebraic block codes: rectangular parity codes and Hamming codes. We also note that the replication code discussed in Section 5.2 is an example of a linear block code.

In the next two chapters, we will study **convolutional codes**, a sub-class of graphical codes.

■ 5.4 Linear Block Codes and Parity Calculations

Linear block codes are examples of algebraic block codes, which take the set of k-bit messages we wish to send (there are 2^k of them) and produce a set of 2^k codewords, each n bits long ($n \geq k$) using *algebraic operations* over the block. The word "block" refers to the fact that any long bit stream can be broken up into k-bit blocks, which are each then expanded to produce n-bit codewords that are sent.

Such codes are also called (n, k) codes, where k message bits are combined to produce n code bits (so each codeword has $n - k$ "redundancy" bits). Often, we use the notation (n, k, d), where d refers to the minimum Hamming distance of the block code. The *rate* of a

block code is defined as k/n; the larger the rate, the less the redundancy overhead incurred by the code.

A *linear* code (whether a block code or not) produces codewords from message bits by restricting the algebraic operations to *linear functions* over the message bits. By linear, we mean that any given bit in a valid codeword is computed as the weighted sum of one or more original message bits.

Linear codes, as we will see, are both powerful and efficient to implement. They are widely used in practice. In fact, *all* the codes we will study—including convolutional codes—are linear, as are most of the codes widely used in practice. We already looked at the properties of a simple linear block code: the replication code we discussed in Section 5.2 is a linear block code with parameters $(n, 1, n)$.

An important and popular class of linear codes are *binary linear codes*. The computations in the case of a binary code use arithmetic modulo 2, which has a special name: algebra in a *Galois Field* of order 2, also denoted $\mathbb{F}_2$. A field must define rules for addition and multiplication, and their inverses. Addition in $\mathbb{F}_2$ is according to the following rules: $0 + 0 = 1 + 1 = 0; 1 + 0 = 0 + 1 = 1$. Multiplication is as usual: $0 \cdot 0 = 0 \cdot 1 = 1 \cdot 0 = 0; 1 \cdot 1 = 1$. We leave you to figure out the additive and multiplicative inverses of 0 and 1. Our focus in this book will be on linear codes over $\mathbb{F}_2$, but there are natural generalizations to fields of higher order (in particular, Reed Solomon codes, which are over Galois Fields of order 2^q).

A linear code is characterized by the following theorem, which is both a necessary and a sufficient condition for a code to be linear:

Theorem 5.5 *A code is linear if, and only if, the sum of any two codewords is another codeword.*

For example, the block code defined by codewords $000, 101, 011$ is *not* a linear code, because $101 + 011 = 110$ is not a codeword. But if we add 110 to the set, we get a linear code because the sum of any two codewords is now another codeword. The code $000, 101, 011, 110$ has a minimum Hamming distance of 2 (that is, the smallest Hamming distance between any two codewords in 2), and can be used to detect all single-bit errors that occur during the transmission of a code word. You can also verify that the minimum Hamming distance of this code is equal to the smallest number of 1's in a non-zero codeword. In fact, that's a general property of all linear block codes, which we state formally below:

Theorem 5.6 *Define the weight of a codeword as the number of 1's in the word. Then, the minimum Hamming distance of a linear block code is equal to the weight of the non-zero codeword with the smallest weight.*

To see why, use the property that the sum of any two codewords must also be a codeword, and that the Hamming distance between any two codewords is equal to the weight of their sum (i.e., $\text{weight}(u + v) = \text{HD}(u, v)$). (In fact, the Hamming distance between any two bit-strings of equal length is equal to the weight of their sum.) We leave the complete proof of this theorem as a useful and instructive exercise for the reader.

The rest of this section shows how to construct linear block codes over $\mathbb{F}_2$. For simplicity, and without much loss of generality, we will focus on correcting single-bit errors. i.e., on *single-error correction* (SEC) codes.. We will show two ways of building the set $\mathcal{S}$ of

transmission messages to have single-error correction capability, and will describe how the receiver can perform error correction on the (possibly corrupted) received messages.

We will start with the *rectangular parity* code in Section 5.4.1, and then discuss the cleverer and more efficient *Hamming code* in Section 5.4.3.

■ 5.4.1 Rectangular Parity SEC Code

We define the *parity* of bits $x_1, x_2, \ldots, x_n$ as $(x_1 + x_2 + \ldots + x_n)$, where the addition is performed modulo 2 (it's the same as taking the exclusive OR of the n bits). The parity is even when the sum is 0 (i.e., the number of ones is even), and odd otherwise.

Let parity(s) denote the parity of all the bits in the bit-string s. We'll use a dot, $\cdot$, to indicate the concatenation (sequential joining) of two messages or a message and a bit. For any message M (a sequence of one or more bits), let $w = M \cdot \text{parity}(M)$. You should be able to confirm that parity(w) $= 0$. This code, which adds a parity bit to each message, is also called the *even parity* code, because the number of ones in each codeword is even. Even parity lets us detect single errors because the set of codewords, $\{w\}$, each defined as $M \cdot \text{parity}(M)$, has a Hamming distance of 2.

If we transmit w when we want to send some message M, then the receiver can take the received word, r, and compute parity(r) to determine if a single error has occurred. The receiver's parity calculation returns 1 if an odd number of the bits in the received message has been corrupted. When the receiver's parity calculation returns a 1, we say there has been a *parity error*.

This section describes a simple approach to building an SEC code by constructing multiple parity bits, each over various subsets of the message bits, and then using the resulting parity errors (or non-errors) to help pinpoint which bit was corrupted.

Rectangular code construction: Suppose we want to send a k-bit message M. Shape the k bits into a rectangular array with r rows and c columns, i.e., $k = rc$. For example, if $k = 8$, the array could be 2×4 or 4×2 (or even 8×1 or 1×8, though those are a little less interesting). Label each data bit with a subscript giving its row and column: the first bit would be d_{11}, the last bit d_{rc}. See Figure 5-3.

Define p_row(i) to be the parity of all the bits in row i of the array and let R be all the row parity bits collected into a sequence:

$$R = [\text{p\_row}(1), \text{p\_row}(2), \ldots, \text{p\_row}(r)]$$

Similarly, define p_col(j) to be the parity of all the bits in column j of the array and let C be all the column parity bits collected into a sequence:

$$C = [\text{p\_col}(1), \text{p\_col}(2), \ldots, \text{p\_col}(c)]$$

Figure 5-3 shows what we have in mind when $k = 8$.

Let $w = M \cdot R \cdot C$, i.e., the transmitted codeword consists of the original message M, followed by the row parity bits R in row order, followed by the column parity bits C in column order. The length of w is $n = rc + r + c$. This code is linear because all the parity bits are linear functions of the message bits. The rate of the code is $rc/(rc + r + c)$.

We now prove that the rectangular parity code can correct all single-bit errors.

d_{11}	d_{12}	d_{13}	d_{14}	p_row(1)
d_{21}	d_{22}	d_{23}	d_{24}	p_row(2)
p_col(1)	p_col(2)	p_col(3)	p_col(4)	

Figure 5-3: A 2×4 arrangement for an 8-bit message with row and column parity.

0	1	1	0	0
1	1	0	1	1
1	0	1	1	

(a)

1	0	0	1	1
0	0	1	0	1
1	0	1	0	

(b)

0	1	1	1	1
1	1	1	0	1
1	0	0	0	

(c)

Figure 5-4: Example received 8-bit messages. Which, if any, have one error? Which, if any, have two?

Proof of single-error correction property: This rectangular code is an SEC code for all values of r and c. We will show that it can correct all single bit errors by showing that its minimum Hamming distance is 3 (i.e., the Hamming distance between any two codewords is at least 3). Consider two different uncoded messages, M_i and M_j. There are three cases to discuss:

- If M_i and M_j differ by a single bit, then the row and column parity calculations involving that bit will result in different values. Thus, the corresponding codewords, w_i and w_j, will differ by three bits: the different data bit, the different row parity bit, and the different column parity bit. So in this case $\text{HD}(w_i, w_j) = 3$.

- If M_i and M_j differ by two bits, then either (1) the differing bits are in the same row, in which case the row parity calculation is unchanged but two column parity calculations will differ, (2) the differing bits are in the same column, in which case the column parity calculation is unchanged but two row parity calculations will differ, or (3) the differing bits are in different rows and columns, in which case there will be two row and two column parity calculations that differ. So in this case $\text{HD}(w_i, w_j) \geq 4$.

- If M_i and M_j differ by three or more bits, then $\text{HD}(w_i, w_j) \geq 3$ because w_i and w_j contain M_i and M_j respectively.

Hence we can conclude that $\text{HD}(w_i, w_j) \geq 3$ and our simple "rectangular" code will be able to correct all single-bit errors.

Decoding the rectangular code: How can the receiver's decoder correctly deduce M from the received w, which may or may not have a single bit error? (If w has more than one error, then the decoder does not have to produce a correct answer.)

Upon receiving a possibly corrupted w, the receiver checks the parity for the rows and columns by computing the sum of the appropriate data bits *and* the corresponding parity bit (all arithmetic in $\mathbb{F}_2$). This sum will be 1 if there is a parity error. Then:

- If there are no parity errors, then there has not been a single error, so the receiver can use the data bits as-is for M. This situation is shown in Figure 5-4(a).

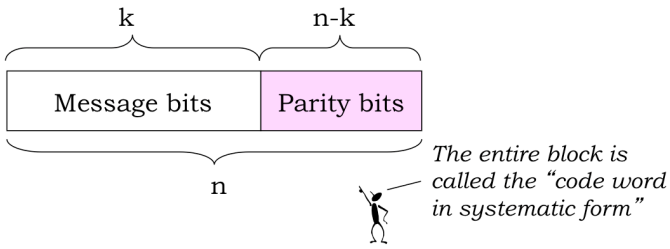


Figure 5-5: A codeword in systematic form for a block code. Any linear code can be transformed into an equivalent systematic code.

- If there is single row or column parity error, then the corresponding parity bit is in error. But the data bits are okay and can be used as-is for M. This situation is shown in Figure 5-4(c), which has a parity error only in the fourth column.
- If there is one row and one column parity error, then the data bit in that row and column has an error. The decoder repairs the error by flipping that data bit and then uses the repaired data bits for M. This situation is shown in Figure 5-4(b), where there are parity errors in the first row and fourth column indicating that d_{14} should be flipped to be a 0.
- Other combinations of row and column parity errors indicate that multiple errors have occurred. There's no "right" action the receiver can undertake because it doesn't have sufficient information to determine which bits are in error. A common approach is to use the data bits as-is for M. If they happen to be in error, that will be detected by the error detection code (mentioned near the beginning of this chapter).

This recipe will produce the most likely message, M, from the received codeword if there has been at most a single transmission error.

In the rectangular code the number of parity bits grows at least as fast as $\sqrt{k}$ (it should be easy to verify that the smallest number of parity bits occurs when the number of rows, r, and the number of columns, c, are equal). Given a fixed amount of communication "bandwidth" or resource, we're interested in devoting as much of it as possible to sending message bits, not parity bits. Are there other SEC codes that have better code rates than our simple rectangular code? A natural question to ask is: *how little redundancy can we get away with and still manage to correct errors?*

The Hamming code uses a clever construction that uses the intuition developed while answering the question mentioned above. We answer this question next.

■ 5.4.2 How many parity bits are needed in an SEC code?

Let's think about what we're trying to accomplish with an SEC code: the correction of transmissions that have a single error. For a transmitted message of length n there are $n + 1$ situations the receiver has to distinguish between: no errors and a single error in a specified position along the string of n received bits. Then, depending on the detected situation, the receiver can make, if necessary, the appropriate correction.

Our first observation, which we will state here without proof, is that any linear code can be transformed into an equivalent **systematic** code. A systematic code is one where

every n-bit codeword can be represented as the original k-bit message followed by the $n-k$ parity bits (it actually doesn't matter how the original message bits and parity bits are interspersed). Figure 5-5 shows a codeword in systematic form.

So, given a systematic code, how many parity bits do we absolutely need? We need to choose n so that single error correction is possible. Since there are $n-k$ parity bits, each combination of these bits must represent *some* error condition that we must be able to correct (or infer that there were no errors). There are 2^{n-k} possible distinct parity bit combinations, which means that we can distinguish at most that many error conditions. We therefore arrive at the constraint

$$n + 1 \leq 2^{n-k} \tag{5.4}$$

i.e., there have to be enough parity bits to distinguish all corrective actions that might need to be taken (including no action). Given k, we can determine $n-k$, the number of parity bits needed to satisfy this constraint. Taking the log (to base 2) of both sides, we can see that the number of parity bits **must** grow at least *logarithmically* with the number of message bits. Not all codes achieve this minimum (e.g., the rectangular code doesn't), but the Hamming code, which we describe next, does.

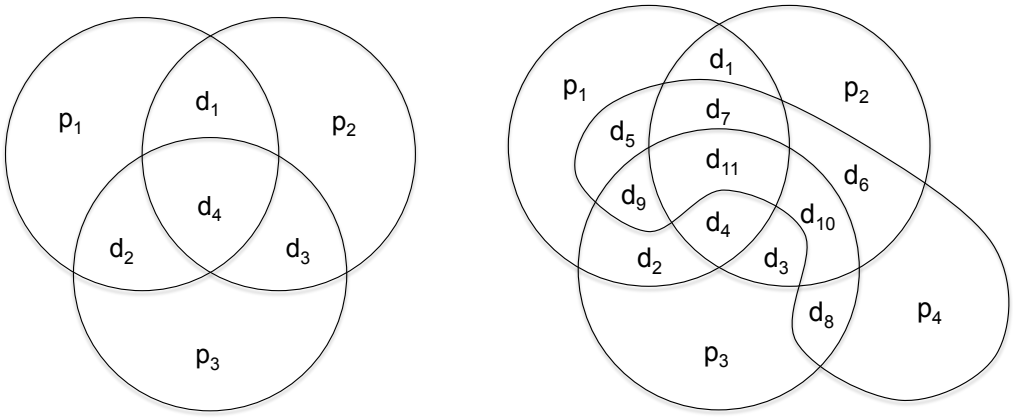
We also note that the reasoning here for an SEC code can be extended to determine a lower bound on the number of parity buts needed to correct $t > 1$ errors.

■ 5.4.3 Hamming Codes

Intuitively, it makes sense that for a code to be efficient, each parity bit should protect as many data bits as possible. By symmetry, we'd expect each parity bit to do the same amount of "work" in the sense that each parity bit would protect the same number of data bits. If some parity bit is shirking its duties, it's likely we'll need a larger number of parity bits in order to ensure that each possible single error will produce a unique combination of parity errors (it's the unique combinations that the receiver uses to deduce which bit, if any, had an error).

The class of Hamming single error correcting codes is noteworthy because they are particularly efficient in the use of parity bits: the number of parity bits used by Hamming codes grows logarithmically with the size of the codeword. Figure 5-6 shows two examples of the class: the (7,4) and (15,11) Hamming codes. The (7,4) Hamming code uses 3 parity bits to protect 4 data bits; 3 of the 4 data bits are involved in each parity computation. The (15,11) Hamming code uses 4 parity bits to protect 11 data bits, and 7 of the 11 data bits are used in each parity computation (these properties will become apparent when we discuss the logic behind the construction of the Hamming code in Section 5.4.4).

Looking at the diagrams, which show the data bits involved in each parity computation, you should convince yourself that each possible single error (don't forget errors in one of the parity bits!) results in a unique combination of parity errors. Let's work through the argument for the (7,4) Hamming code. Here are the parity-check computations performed



(a) (7,4) code (b) (15,11) code

Figure 5-6: Venn diagrams of Hamming codes showing which data bits are protected by each parity bit.

by the receiver:

$$\begin{aligned} E_1 &= (d_1 + d_2 + d_4 + p_1) \mod 2 \\ E_2 &= (d_1 + d_3 + d_4 + p_2) \mod 2 \\ E_3 &= (d_2 + d_3 + d_4 + p_3) \mod 2 \end{aligned}$$

where each E_i is called a *syndrome* bit because it helps the receiver diagnose the "illness" (errors) in the received data. For each combination of syndrome bits, we can look for the bits in each codeword that appear in *all* the E_i computations that produced 1; these bits are potential candidates for having an error since any of them could have caused the observed parity errors. Now eliminate from the candidates those bits that appear in *any* E_i computations that produced 0 since those calculations prove those bits didn't have errors. We'll be left with either no bits (no errors occurred) or one bit (the bit with the single error).

For example, if $E_1 = 1$, $E_2 = 0$ and $E_3 = 1$, we notice that bits d_2 and d_4 both appear in the computations for E_1 and E_3. However, d_4 appears in the computation for E_2 and should be eliminated, leaving d_2 as the sole candidate as the bit with the error.

Another example: suppose $E_1 = 1$, $E_2 = 0$ and $E_3 = 0$. Any of the bits appearing in the computation for E_1 could have caused the observed parity error. Eliminating those that appear in the computations for E_2 and E_3, we're left with p_1, which must be the bit with the error.

Applying this reasoning to each possible combination of parity errors, we can make a table that shows the appropriate corrective action for each combination of the syndrome bits:

$E_3E_2E_1$	Corrective Action
000	no errors
001	p_1 has an error, flip to correct
010	p_2 has an error, flip to correct
011	d_1 has an error, flip to correct
100	p_3 has an error, flip to correct
101	d_2 has an error, flip to correct
110	d_3 has an error, flip to correct
111	d_4 has an error, flip to correct

■ 5.4.4 Is There a Logic to the Hamming Code Construction?

So far so good, but the allocation of data bits to parity-bit computations may seem rather arbitrary and it's not clear how to build the corrective action table except by inspection.

The cleverness of Hamming codes is revealed if we order the data and parity bits in a certain way and assign each bit an index, starting with 1:

index	1	2	3	4	5	6	7
binary index	001	010	011	100	101	110	111
(7,4) code	p_1	p_2	d_1	p_3	d_2	d_3	d_4

This table was constructed by first allocating the parity bits to indices that are powers of two (e.g., 1, 2, 4, ...). Then the data bits are allocated to the so-far unassigned indicies, starting with the smallest index. It's easy to see how to extend this construction to any number of data bits, remembering to add additional parity bits at indices that are a power of two.

Allocating the data bits to parity computations is accomplished by looking at their respective indices in the table above. Note that we're talking about the *index* in the table, not the subscript of the bit. Specifically, d_i is included in the computation of p_j if (and only if) the logical AND of binary index(d_i) and binary index(p_j) is non-zero. Put another way, d_i is included in the computation of p_j if, and only if, index(p_j) contributes to index(d_i) when writing the latter as sums of powers of 2.

So the computation of p_1 (with an index of 1) includes all data bits with odd indices: d_1, d_2 and d_4. And the computation of p_2 (with an index of 2) includes d_1, d_3 and d_4. Finally, the computation of p_3 (with an index of 4) includes d_2, d_3 and d_4. You should verify that these calculations match the E_i equations given above.

If the parity/syndrome computations are constructed this way, it turns out that $E_3E_2E_1$, treated as a binary number, gives the index of the bit that should be corrected. For example, if $E_3E_2E_1 = 101$, then we should correct the message bit with index 5, i.e., d_2. This corrective action is exactly the one described in the earlier table we built by inspection.

The Hamming code's syndrome calculation and subsequent corrective action can be efficiently implemented using digital logic and so these codes are widely used in contexts where single error correction needs to be fast, e.g., correction of memory errors when fetching data from DRAM.

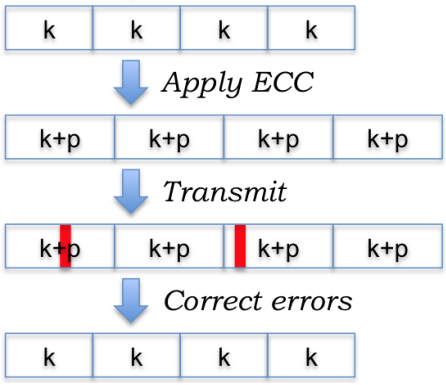


Figure 5-7: Dividing a long message into multiple SEC-protected blocks of k bits each, adding parity bits to each constituent block. The red vertical rectangles refer to bit errors.

5.5 Protecting Longer Messages with SEC Codes

SEC codes are a good building block, but they correct at most one error in a block of n coded bits. As messages get longer, the solution, of course, is to break up a longer message into smaller blocks of k bits each, and to protect each one with its own SEC code. The result might look as shown in Figure 5-7.

5.5.1 Coping with Burst Errors

Over many channels, errors occur in bursts and the BSC error model is invalid. For example, wireless channels suffer from *interference* from other transmitters and from *fading*, caused mainly by *multi-path propagation* when a given signal arrives at the receiver from multiple paths and interferes in complex ways because the different copies of the signal experience different degrees of attenuation and different delays. Another reason for fading is the presence of obstacles on the path between sender and receiver; such fading is called *shadow fading*.

The behavior of a fading channel is complicated and beyond the scope of 6.02, but the impact of fading on communication is that the random process describing the bit error probability is no longer independent and identically distributed from one bit to another. The BSC model needs to be replaced with a more complicated one in which errors may occur in *bursts*. Many such theoretical models guided by empirical data exist, but we won't go into them here. Our goal is to understand how to develop error correction mechanisms when errors occur in bursts.

But what do we mean by a "burst"? The simplest model is to model the channel as having two states, a "good" state and a "bad" state. In the "good" state, the bit error probability is p_g and in the "bad" state, it is $p_b > p_g$. Once in the good state, the channel has some probability of remaining there (generally $> 1/2$) and some probability of moving into the "bad" state, and vice versa. It should be easy to see that this simple model has the property that the probability of a bit error depends on whether the previous bit (or previous few bits) are in error or not. The reason is that the odds of being in a "good" state are high if the previous few bits have been correct.

At first sight, it might seem like the SEC schemes we studied are poorly suited for a

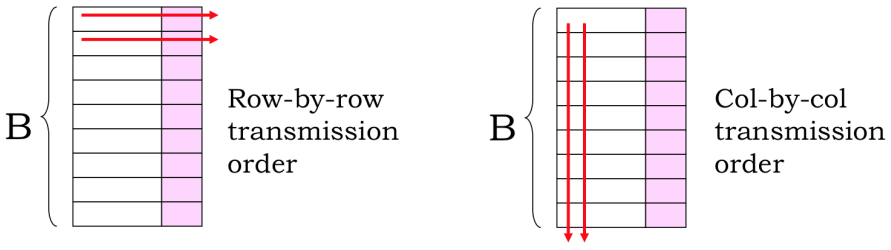


Figure 5-8: Interleaving can help recover from burst errors: code each block row-wise with an SEC, but transmit them in interleaved fashion in columnar order. As long as a set of burst errors corrupts some set of k^{th} bits, the receiver can recover from *all* the errors in the burst.

channel experiencing burst errors. The reason is shown in Figure 5-8 (left), where each block of the message is protected by its SEC parity bits. The different blocks are shown as different rows. When a burst error occurs, multiple bits in an SEC block are corrupted, and the SEC can't recover from them.

Interleaving is a commonly used technique to recover from burst errors on a channel even when the individual blocks are protected with a code that, on the face of it, is not suited for burst errors. The idea is simple: code the blocks as before, but transmit them in a "columnar" fashion, as shown in Figure 5-8 (right). That is, send the first bit of block 1, then the first bit of block 2, and so on until all the first bits of each block in a set of some predefined size are sent. Then, send the second bits of each block in sequence, then the third bits, and so on.

What happens on a burst error? Chances are that it corrupts a set of "first" bits, or a set of "second" bits, or a set of "third" bits, etc., because those are the bits sent in order on the channel. As long as only a set of k^{th} bits are corrupted, the receiver can correct *all* the errors. The reason is that each coded block will now have at most one error. Thus, SEC codes are a useful primitive to correct against burst errors, in concert with interleaving.

■ Acknowledgments

Many thanks to Katrina LaCurts for carefully reading these notes and making several useful comments.

■ Problems and Questions

1. Prove that the Hamming distance satisfies the triangle inequality. That is, show that $\text{HD}(x, y) + \text{HD}(y, z) \geq \text{HD}(x, z)$ for any three n-bit binary words.

2. Consider the following rectangular linear block code:

```
D0  D1  D2  D3  D4   | P0
D5  D6  D7  D8  D9   | P1
D10 D11 D12 D13 D14  | P2
---------------------
P3  P4  P5  P6  P7   |
```

   Here, `D0`–`D14` are data bits, `P0`–`P2` are row parity bits and `P3`–`P7` are column parity bits. What are n, k, and d for this linear code?

3. Consider a rectangular parity code as described in Section 5.4.1. Ben Bitdiddle would like use this code at a variety of different code rates and experiment with them on some channel.

   (a) Is it possible to obtain a rate lower than 1/3 with this code? Explain your answer.

   (b) Suppose he is interested in code rates like 1/2, 2/3, 3/4, etc.; i.e., in general a rate of $\frac{n-1}{n}$, for integer $n > 1$. Is it always possible to pick the parameters of the code (i.e, the block size and the number of rows and columns over which to construct the parity bits) so that any such code rate is achievable? Explain your answer.

4. Two-Bit Communications (TBC), a slightly suspect network provider, uses the following linear block code over its channels. All arithmetic is in $\mathbb{F}_2$.

$$P_0 = D_0, P_1 = (D_0 + D_1), P_2 = D_1.$$

   (a) What are n and k for this code?

   (b) Suppose we want to perform syndrome decoding over the received bits. Write out the three syndrome equations for E_0, E_1, E_2.

   (c) For the eight possible syndrome values, determine what error can be detected (none, error in a particular data or parity bit, or multiple errors). Make your choice using maximum likelihood decoding, assuming a small bit error probability (i.e., the smallest number of errors that's consistent with the given syndrome).

   (d) Suppose that the the 5-bit blocks arrive at the receiver in the following order: D_0, D_1, P_0, P_1, P_2. If 11011 arrives, what will the TBC receiver report as the received data after error correction has been performed? Explain your answer.

   (e) TBC would like to improve the code rate while still maintaining single-bit error correction. Their engineer would like to reduce the number of parity bits by 1. Give the formulas for P_0 and P_1 that will accomplish this goal, or briefly explain why no such code is possible.

5. Pairwise Communications has developed a linear block code over $\mathbb{F}_2$ with three data and three parity bits, which it calls the *pairwise code*:

$$\begin{aligned} P_1 &= D_1 + D_2 \quad \text{(Each } D_i \text{ is a data bit; each } P_i \text{ is a parity bit.)} \\ P_2 &= D_2 + D_3 \\ P_3 &= D_3 + D_1 \end{aligned}$$

   (a) Fill in the values of the following three attributes of this code:

   (i) Code rate = _____

   (ii) Number of 1s in a minimum-weight non-zero codeword = _____

   (iii) Minimum Hamming distance of the code = _____

6. Consider the same "pairwise code" as in the previous problem. The receiver computes three syndrome bits from the (possibly corrupted) received data and parity bits: $E_1 = D_1 + D_2 + P_1, E_2 = D_2 + D_3 + P_2$, and $E_3 = D_3 + D_1 + P_3$. The receiver performs maximum likelihood decoding using the syndrome bits. For the combinations of syndrome bits in the table below, state what the maximum-likelihood decoder believes has occured: no errors, a single error in a specific bit (state which one), or multiple errors.

$E_3E_2E_1$	Error pattern [No errors / Error in bit ... (specify bit) / Multiple errors]
0 0 0	
0 0 1	
0 1 0	
0 1 1	
1 0 0	
1 0 1	
1 1 0	
1 1 1	

7. Alyssa P. Hacker extends the aforementioned pairwise code by adding an *overall parity bit*. That is, she computes $P_4 = \sum_{i=1}^{3}(D_i + P_i)$, and appends P_4 to each original codeword to produce the new set of codewords. What improvement in error correction or detection capabilities, if any, does Alyssa's extended code show over Pairwise's original code? Explain your answer.

8. For each of the sets of codewords below, determine whether the code is a linear block code over $\mathbb{F}_2$ or not. Also give the rate of each code.

   (a) {000,001,010,011}.

   (b) {000, 011, 110, 101}.

   (c) {111, 100, 001, 010}.

   (d) {00000, 01111, 10100, 11011}.

   (e) {00000}.

9. For any linear block code over $\mathbb{F}_2$ with minimum Hamming distance at least $2t+1$ between codewords, show that:

$$2^{n-k} \geq 1 + \binom{n}{1} + \binom{n}{2} + \ldots \binom{n}{t}.$$

Hint: How many errors can such a code always correct?

10. For each (n, k, d) combination below, state whether a linear block code with those parameters exists or not. Please provide a brief explanation for each case: if such a code exists, give an example; if not, you may rely on a suitable necessary condition.

    (a) $(31, 26, 3)$: **Yes / No**

    (b) $(32, 27, 3)$: **Yes / No**

    (c) $(43, 42, 2)$: **Yes / No**

    (d) $(27, 18, 3)$: **Yes / No**

    (e) $(11, 5, 5)$: **Yes / No**

11. Using the Hamming code construction for the $(7, 4)$ code, construct the parity equations for the $(15, 11)$ code. How many equations does this code have? How many message bits contribute to each parity bit?

12. Prove Theorems 5.2 and 5.3. (Don't worry too much if you can't prove the latter; we will give the proof when we discuss convolutional codes in Lecture 8.)

13. The weight of a codeword in a linear block code over $\mathbb{F}_2$ is the number of 1's in the word. Show that any linear block code must either: (1) have only even weight codewords, or (2) have an equal number of even and odd weight codewords.
Hint: Proof by contradiction.

14. There are N people in a room, each wearing a hat colored red or blue, standing in a line in order of increasing height. Each person can see only the hats of the people in front, and does not know the color of his or her own hat. They play a game as a team, whose rules are simple. Each person gets to say one word: "red" or "blue". If the word they say correctly guesses the color of their hat, the team gets 1 point; if they guess wrong, 0 points. Before the game begins, they can get together to agree on a protocol (i.e., what word they will say under what conditions). Once they determine the protocol, they stop talking, form the line, and are given their hats at random.

    Can you develop a protocol that will maximize their score? What score does your protocol achieve?

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: October 2, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 6
Convolutional Codes: Construction and Encoding

This chapter introduces a widely used class of codes, called **convolutional codes**, which are used in a variety of systems including today's popular wireless standards (such as 802.11) and in satellite communications. They are also used as a building block in more powerful modern codes, such as turbo codes, which are used in wide-area cellular wireless network standards such as 3G, LTE, and 4G. Convolutional codes are beautiful because they are intuitive, one can understand them in many different ways, and there is a way to decode them so as to recover the *most likely* message from among the set of all possible transmitted messages. This chapter discusses the encoding of convolutional codes; the next one discusses how to decode convolutional codes efficiently.

Like the block codes discussed in the previous chapter, convolutional codes involve the computation of parity bits from message bits and their transmission, and they are also linear codes. Unlike block codes in systematic form, however, the sender does not send the message bits followed by (or interspersed with) the parity bits; in a convolutional code, the sender *sends only the parity bits*. These codes were invented by Peter Elias '44, an MIT EECS faculty member, in the mid-1950s. For several years, it was not known just how powerful these codes are and how best to decode them. The answers to these questions started emerging in the 1960s, with the work of people like Andrew Viterbi '57, G. David Forney (SM '65, Sc.D. '67, and MIT EECS faculty member), Jim Omura SB '63, and many others.

■ 6.1 Convolutional Code Construction

The encoder uses a *sliding window* to calculate $r > 1$ parity bits by combining various subsets of bits in the window. The combining is a simple addition in $\mathbb{F}_2$, as in the previous chapter (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, however, the windows overlap and slide by 1, as shown in Figure 6-1. The size of the window, in bits, is called the code's **constraint length**. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit. Because the parity bits are the only bits sent over the channel, a larger constraint length generally

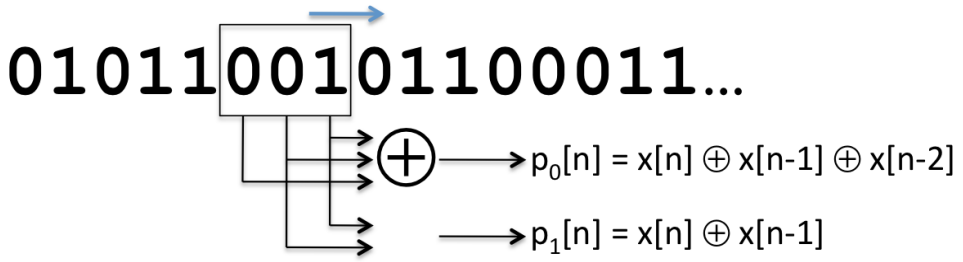


Figure 6-1: An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$.

implies a greater resilience to bit errors. The trade-off, though, is that it will take considerably longer to decode codes of long constraint length (we will see in the next chapter that the complexity of decoding is exponential in the constraint length), so one cannot increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code produces r parity bits per window and slides the window forward by one bit at a time, its rate (when calculated over long messages) is $1/r$. The greater the value of r, the higher the resilience of bit errors, but the trade-off is that a proportionally higher amount of communication bandwidth is devoted to coding overhead. In practice, we would like to pick r and the constraint length to be as small as possible while providing a low enough resulting probability of a bit error.

In 6.02, we will use K (upper case) to refer to the constraint length, a somewhat unfortunate choice because we have used k (lower case) in previous lectures to refer to the number of message bits that get encoded to produce coded bits. Although "L" might be a better way to refer to the constraint length, we'll use K because many papers and documents in the field use K (in fact, many papers use k in lower case, which is especially confusing). Because we will rarely refer to a "block" of size k while talking about convolutional codes, we hope that this notation won't cause confusion.

Armed with this notation, we can describe the encoding process succinctly. The encoder looks at K bits at a time and produces r parity bits according to carefully chosen functions that operate over various subsets of the K bits.[1] One example is shown in Figure 6-1, which shows a scheme with $K = 3$ and $r = 2$ (the rate of this code, $1/r = 1/2$). The encoder spits out r bits, which are sent sequentially, slides the window by 1 to the right, and then repeats the process. That's essentially it.

At the transmitter, the two principal remaining details that we must describe are:

1. What are good parity functions and how can we represent them conveniently?
2. How can we implement the encoder efficiently?

The rest of this lecture will discuss these issues, and also explain why these codes are called "convolutional".

[1] By convention, we will assume that each message has $K - 1$ "0" bits padded in front, so that the initial conditions work out properly.

6.2 Parity Equations

The example in Figure 6-1 shows one example of a set of *parity equations*, which govern the way in which parity bits are produced from the sequence of message bits, X. In this example, the equations are as follows (all additions are in $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{6.1}$$

The rate of this code is $1/2$.

An example of parity equations for a rate $1/3$ code is

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \\ p_2[n] &= x[n] + x[n-2] \end{aligned} \tag{6.2}$$

In general, one can view each parity equation as being produced by combining the message bits, X, and a **generator polynomial**, g. In the first example above, the generator polynomial coefficients are $(1,1,1)$ and $(1,1,0)$, while in the second, they are $(1,1,1),(1,1,0)$, and $(1,0,1)$.

We denote by g_i the K-element generator polynomial for parity bit p_i. We can then write $p_i[n]$ as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{6.3}$$

The form of the above equation is a *convolution* of g and x—hence the term "convolutional code". The number of generator polynomials is equal to the number of generated parity bits, r, in each sliding window. The rate of the code is $1/r$ if the encoder slides the window one bit at a time.

6.2.1 An Example

Let's consider the two generator polynomials of Equations 6.1 (Figure 6-1). Here, the generator polynomials are

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 1,1,0 \end{aligned} \tag{6.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0\ \forall n < 0$), then the parity

bits from Equations 6.1 work out to be

$$\begin{aligned} p_0[0] &= (1+0+0) = 1 \\ p_1[0] &= (1+0) = 1 \\ p_0[1] &= (0+1+0) = 1 \\ p_1[1] &= (0+1) = 1 \\ p_0[2] &= (1+0+1) = 0 \\ p_1[2] &= (1+0) = 1 \\ p_0[3] &= (1+1+0) = 0 \\ p_1[3] &= (1+1) = 0. \end{aligned} \tag{6.5}$$

Therefore, the bits transmitted over the channel are $[1,1,1,1,0,0,0,0,\ldots]$.

There are several generator polynomials, but understanding how to construct good ones is outside the scope of 6.02. Some examples (found by J. Busgang) are shown in Table 6-1.

Constraint length	g_0	g_1
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	1110011001

Table 6-1: Examples of generator polynomials for rate $1/2$ convolutional codes with different constraint lengths.

■ 6.3 Two Views of the Convolutional Encoder

We now describe two views of the convolutional encoder, which we will find useful in better understanding convolutional codes and in implementing the encoding and decoding procedures. The first view is in terms of **shift registers**, where one can construct the mechanism using shift registers that are connected together. This view is useful in developing hardware encoders. The second is in terms of a **state machine**, which corresponds to a view of the encoder as a set of states with well-defined transitions between them. The state machine view will turn out to be extremely useful in figuring out how to decode a set of parity bits to reconstruct the original message bits.

■ 6.3.1 Shift-Register View

Figure 6-2 shows the same encoder as Figure 6-1 and Equations (6.1) in the form of a block diagram made up of shift registers. The $x[n-i]$ values (here there are two) are referred to

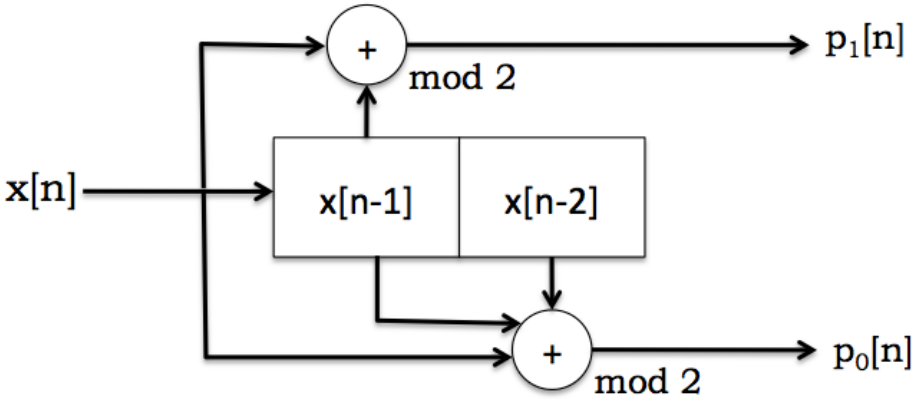


Figure 6-2: Block diagram view of convolutional coding with shift registers.

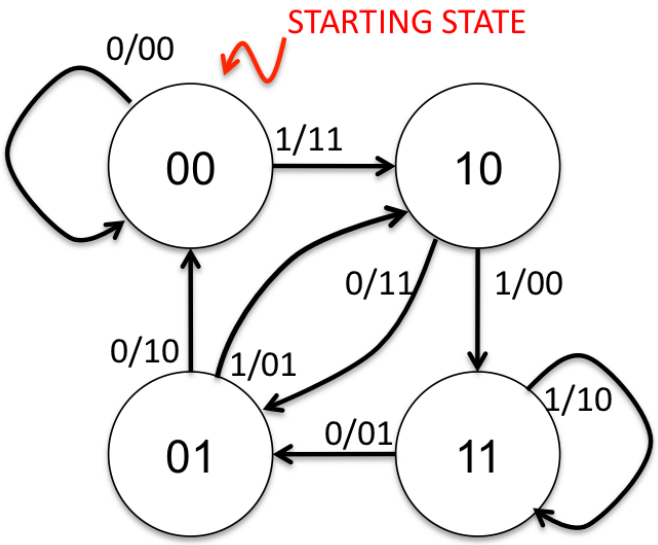


Figure 6-3: State-machine view of convolutional coding.

as the *state* of the encoder. This block diagram takes message bits in one bit at a time, and spits out parity bits (two per input bit, in this case).

Input message bits, $x[n]$, arrive from the left. (These bits arrive after being processed by the receiver's sampling and demapping procedures). The block diagram calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; two in this example). After the r parity bits are produced, the state of the encoder shifts by 1, with $x[n]$ taking the place of $x[n-1]$, $x[n-1]$ taking the place of $x[n-2]$, and so on, with $x[n-K+1]$ being discarded. This block diagram is directly amenable to a hardware implementation using shift registers.

■ 6.3.2 State-Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 6-3 for the same example that we have used throughout this chapter (Figure 6-1).

An important point to note: the state machine for a convolutional code is *identical* for all codes with a given constraint length, K, and the number of states is always 2^{K-1}. Only the p_i labels change depending on the number of generator polynomials and the values of their coefficients. Each state is labeled with $x[n-1]x[n-2]\ldots x[n-K+1]$. Each arc is labeled with $x[n]/p_0p_1\ldots$. In this example, if the message is 101100, the transmitted bits are 11 11 01 00 01 10.

This state-machine view is an elegant way to explain what the transmitter does, and also what the receiver ought to do to decode the message, as we now explain. The transmitter begins in the initial state (labeled "STARTING STATE" in Figure 6-3) and processes the message one bit at a time. For each message bit, it makes the state transition from the current state to the new one depending on the value of the input bit, and sends the parity bits that are on the corresponding arc.

The receiver, of course, does not have direct knowledge of the transmitter's state transitions. It only sees the received sequence of parity bits, with possible bit errors. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is called decoding, which we introduce next, and study in more detail in the next chapter.

■ 6.4 The Decoding Problem

As mentioned above, the receiver should determine the "best possible" sequence of transmitter states. There are many ways of defining "best", but one that is especially appealing is the *most likely* sequence of states (i.e., message bits) that must have been traversed (sent) by the transmitter. A decoder that is able to infer the most likely sequence is also called a **maximum-likelihood** decoder.

Consider the binary symmetric channel, where bits are received erroneously with probability $p_e < 1/2$. What should a maximum-likelihood decoder do when it receives r? We show now that if it decodes r as c, the nearest valid codeword with smallest Hamming distance from r, then the decoding is a maximum-likelihood one.

A maximum-likelihood decoder maximizes the quantity $\mathbb{P}(r|c)$; i.e., it finds c so that the probability that r was received given that c was sent is maximized. Consider any codeword $\tilde{c}$. If r and $\tilde{c}$ differ in d bits (i.e., their Hamming distance is d), then $\mathbb{P}(r|c) = p_e^d(1-p_e)^{N-d}$, where N is the length of the received word (and also the length of each valid codeword). It's more convenient to take the logarithm of this conditional probaility, also termed the *log-likelihood*:[2]

$$\log \mathbb{P}(r|\tilde{c}) = d\log p_e + (N-d)\log(1-p_e) = d\log\frac{p_e}{1-p_e} + N\log(1-p_e). \tag{6.6}$$

If $p_e < 1/2$, which is the practical realm of operation, then $\frac{p_e}{1-p_e} < 1$ and the log term is negative (otherwise, it's non-negative). As a result, minimizing the log likelihood boils down to minimizing d, because the second term on the RHS of Eq. (6.6) is a constant.

A simple numerical example may be useful. Suppose that bit errors are independent and identically distribute with a BER of 0.001, and that the receiver digitizes a sequence

[2]The base of the logarithm doesn't matter to us at this stage, but traditionally the log likelihood is defined as the natural logarithm (base e).

Msg	*Xmit**	*Rcvd*	*d*
0000	000000000000	111011000110	7
0001	000000111110		8
0010	000011111000		8
0011	000011010110		4
0100	001111100000		6
0101	001111011110		5
0110	001101001000		7
0111	001100100110		6
1000	111110000000		4
1001	111110111110		5
1010	111101111000		7
1011	111101000110		2
1100	110001100000		5
1101	110001011110		4
1110	110010011000		6
1111	110010100110		3

Most likely: 1011

Figure 6-4: When the probability of bit error is less than 1/2, maximum-likelihood decoding boils down to finding the message whose parity bit sequence, when transmitted, has the smallest Hamming distance to the received sequence. Ties may be broken arbitrarily. Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence because of the sheer number. For instance, when $N = 256$ bits (a really small packet), the number of possibilities rivals the number of atoms in the universe!

of analog samples into the bits 1101001. Is the sender more likely to have sent 1100111 or 1100001? The first has a Hamming distance of 3, and the probability of receiving that sequence is $(0.999)^4(0.001)^3 = 9.9 \times 10^{-10}$. The second choice has a Hamming distance of 1 and a probability of $(0.999)^6(0.001)^1 = 9.9 \times 10^{-4}$, which is *six orders of magnitude higher* and is overwhelmingly more likely.

Thus, the most likely sequence of parity bits that was transmitted must be the one with the smallest Hamming distance from the sequence of parity bits received. Given a choice of possible transmitted messages, the decoder should pick the one with the smallest such Hamming distance.

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For example, see Figure 6-4, which shows a convolutional code with $K = 3$ and rate $1/2$. If the receiver gets 111011000110, then some errors have occurred, because no valid transmitted sequence matches the received one. The last column in the example shows d, the Hamming distance to all the possible transmitted sequences, with

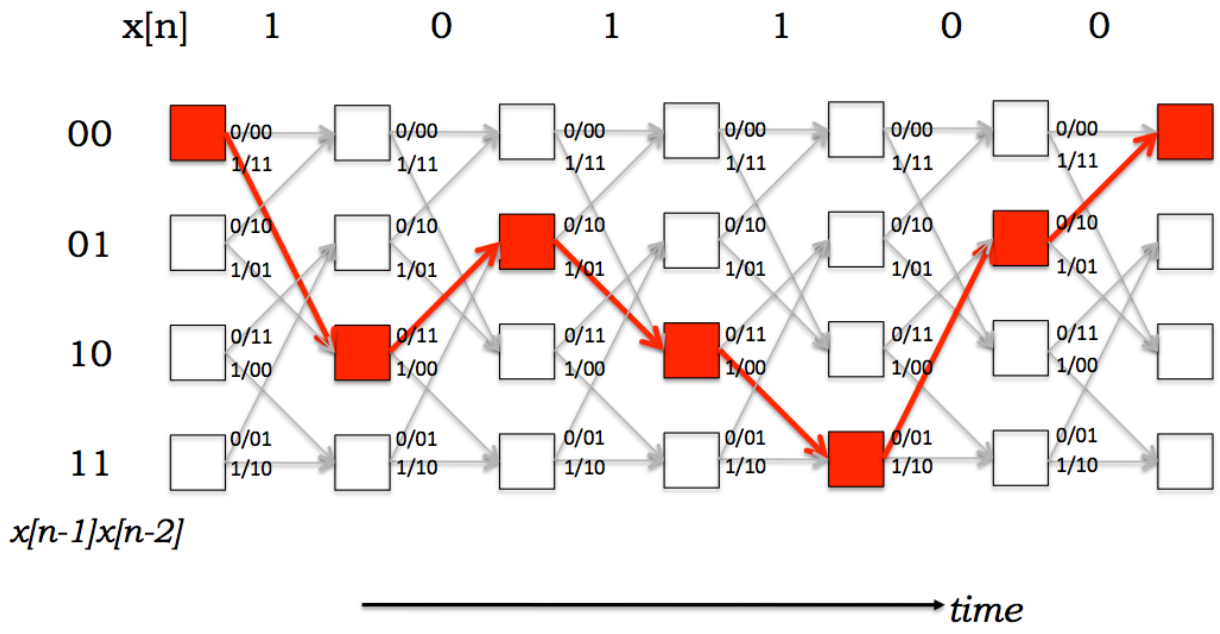


Figure 6-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

the smallest one circled. To determine the most-likely 4-bit message that led to the parity sequence received, the receiver could look for the message whose transmitted parity bits have smallest Hamming distance from the received bits. (If there are ties for the smallest, we can break them arbitrarily, because all these possibilities have the same resulting post-coded BER.)

The straightforward approach of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. The reason is that a transmit sequence of N bits has 2^N possible strings, a number that is simply too large for even small values of N, like 256 bits. We need a better plan for the receiver to navigate this unbelievable large space of possibilities and quickly determine the valid message with smallest Hamming distance. We will study a powerful and widely applicable method for solving this problem, called *Viterbi decoding*, in the next lecture. This decoding method uses a special structure called the **trellis**, which we describe next.

■ 6.5 The Trellis

The trellis is a structure derived from the state machine that will allow us to develop an efficient way to decode convolutional codes. The state machine view shows what happens at each instant when the sender has a message bit to process, but doesn't show how the system evolves in time. The *trellis* is a structure that makes the time evolution explicit. An example is shown in Figure 6-5. Each column of the trellis has the set of states; each state in a column is connected to two states in the next column—the same two states in the state diagram. The top link from each state in a column of the trellis shows what gets transmitted on a "0", while the bottom shows what gets transmitted on a "1". The picture

shows the links between states that are traversed in the trellis given the message 101100.

We can now think about what the decoder needs to do in terms of this trellis. It gets a sequence of parity bits, and needs to determine the best path through the trellis—that is, the sequence of states in the trellis that can explain the observed, and possibly corrupted, sequence of received parity bits.

The Viterbi decoder finds a **maximum-likelihood path** through the trellis. We will study it in the next chapter.

Problems and exercises on convolutional coding are at the end of the next chapter, after we discuss the decoding process.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: October 9, 2011
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 7
Viterbi Decoding of Convolutional Codes

This chapter describes an elegant and efficient method to decode convolutional codes, whose construction and encoding we described in the previous chapter. This decoding method avoids explicitly enumerating the 2^N possible combinations of N-bit parity bit sequences. This method was invented by Andrew Viterbi '57 and bears his name.

7.1 The Problem

At the receiver, we have a sequence of voltage samples corresponding to the parity bits that the transmitter has sent. For simplicity, and without loss of generality, we will assume that the receiver picks a suitable sample for the bit, or better still, averages the set of samples corresponding to a bit, digitizes that value to a "0" or "1" by comparing to the threshold voltage (the demapping step), and propagates that bit decision to the decoder.

Thus, we have a *received bit sequence*, which for a convolutionally coded stream corresponds to the stream of parity bits. If we decode this received bit sequence with no other information from the receiver's sampling and demapper, then the decoding process is termed **hard decision decoding** ("hard decoding"). If, in addition, the decoder is given the actual voltage samples and uses that information in decoding the data, we term the process **soft decision decoding** ("soft decoding").

The Viterbi decoder can be used in either case. Intuitively, because hard decision decoding makes an early decision regarding whether a bit is 0 or 1, it throws away information in the digitizing process. It might make a wrong decision, especially for voltages near the threshold, introducing a greater number of bit errors in the received bit sequence. Although it still produces the most likely transmitted sequence *given* the received bit sequence, by introducing additional errors in the early digitization, the overall reduction in the probability of bit error will be smaller than with soft decision decoding. But it is conceptually easier to understand hard decoding, so we will start with that, before going on to soft decoding.

As mentioned in the previous chapter, the trellis provides a good framework for un-

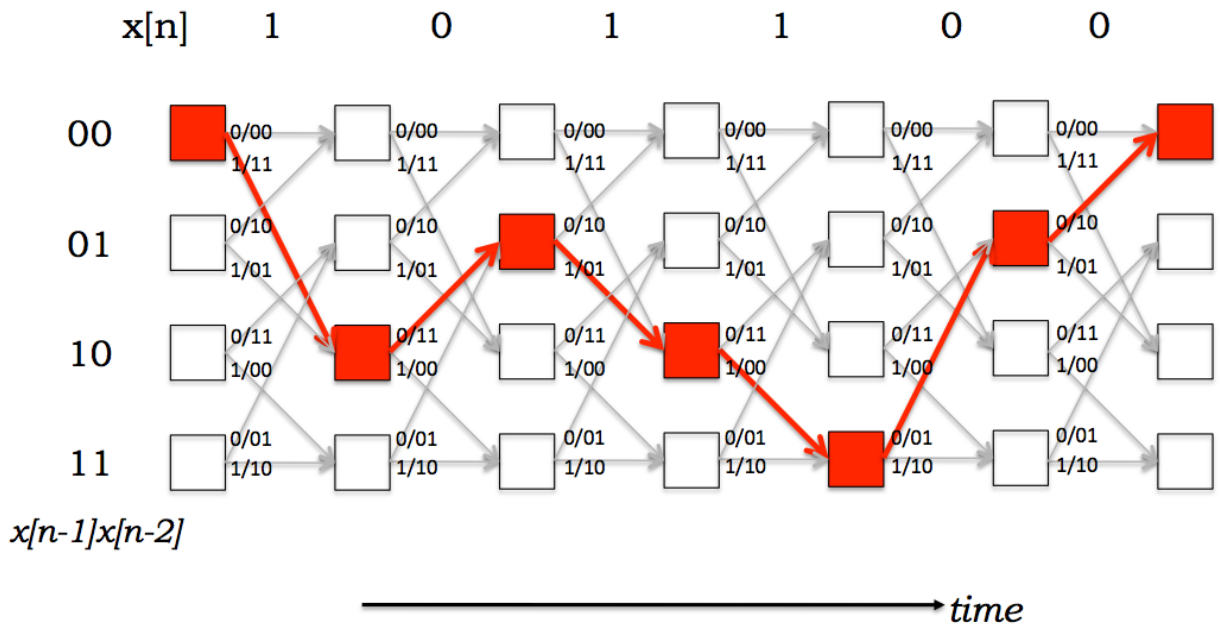


Figure 7-1: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

derstanding the decoding procedure for convolutional codes (Figure 7-1). Suppose we have the entire trellis in front of us for a code, and now receive a sequence of digitized bits (or voltage samples). If there are no errors (i.e., the signal-to-noise ratio, SNR, is high enough), then there will be some path through the states of the trellis that would exactly match the received sequence. That path (specifically, the concatenation of the encoding of each state along the path) corresponds to the transmitted parity bits. From there, getting to the original message is easy because the top arc emanating from each node in the trellis corresponds to a "0" bit and the bottom arrow corresponds to a "1" bit.

When there are bit errors, what can we do? As explained earlier, finding the *most likely* transmitted message sequence is appealing because it minimizes the BER. If we can come up with a way to capture the errors introduced by going from one state to the next, then we can accumulate those errors along a path and come up with an estimate of the total number of errors along the path. Then, the path with the smallest such accumulation of errors is the path we want, and the transmitted message sequence can be easily determined by the concatenation of states explained above.

To solve this problem, we need a way to capture any errors that occur in going through the states of the trellis, and a way to navigate the trellis without actually materializing the entire trellis (i.e., without enumerating all possible paths through it and then finding the one with smallest accumulated error). The Viterbi decoder solves these problems. It is an example of a more general approach to solving optimization problems, called *dynamic programming*. Later in the course, we will apply similar concepts in network routing, an unrelated problem, to find good paths in multi-hop networks.

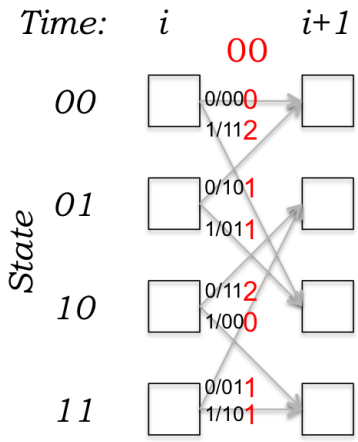


Figure 7-2: The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00.

7.2 The Viterbi Decoder

The decoding algorithm uses two metrics: the **branch metric** (BM) and the **path metric** (PM). The branch metric is a measure of the "distance" between what was transmitted and what was received, and is defined for each arc in the trellis. In hard decision decoding, where we are given a sequence of digitized parity bits, the branch metric is the *Hamming distance* between the expected parity bits and the received ones. An example is shown in Figure 7-2, where the received bits are 00. For each state transition, the number on the arc shows the branch metric for that transition. Two of the branch metrics are 0, corresponding to the only states and transitions where the corresponding Hamming distance is 0. The other non-zero branch metrics correspond to cases when there are bit errors.

The path metric is a value associated with a state in the trellis (i.e., a value associated with each node). For hard decision decoding, it corresponds to the Hamming distance over the most likely path from the initial state to the current state in the trellis. By "most likely", we mean the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states. The path with the smallest Hamming distance minimizes the total number of bit errors, and is most likely when the BER is low.

The key insight in the Viterbi algorithm is that the receiver can compute the path metric for a (state, time) pair incrementally using the path metrics of previously computed states and the branch metrics.

7.2.1 Computing the Path Metric

Suppose the receiver has computed the path metric PM$[s, i]$ for each state s at time step i (recall that there are 2^{K-1} states, where K is the constraint length of the convolutional code). In hard decision decoding, the value of PM$[s, i]$ is the total number of bit errors detected when comparing the received parity bits to the most likely transmitted message, considering all messages that could have been sent by the transmitter until time step i (starting from state "00", which we will take to be the starting state always, by convention).

Among all the possible states at time step i, the most likely state is the one with the smallest path metric. If there is more than one such state, they are all equally good possibilities.

Now, how do we determine the path metric at time step $i+1$, PM$[s, i+1]$, for each state s? To answer this question, first observe that if the transmitter is at state s at time step $i+1$, then *it must have been in only one of two possible states at time step i*. These two *predecessor states*, labeled α and β, are always the same for a given state. In fact, they depend only on the constraint length of the code and not on the parity functions. Figure 7-2 shows the predecessor states for each state (the other end of each arrow). For instance, for state 00, $\alpha = 00$ and $\beta = 01$; for state 01, $\alpha = 10$ and $\beta = 11$.

Any message sequence that leaves the transmitter in state s at time $i+1$ *must have* left the transmitter in state α or state β at time i. For example, in Figure 7-2, to arrive in state '01' at time $i+1$, one of the following two properties *must hold*:

1. The transmitter was in state '10' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent '11' as the parity bits and there were two bit errors, because we received the bits 00. Then, the path metric of the new state, PM['01', $i+1$] is equal to PM['10', i] + 2, because the new state is '01' and the corresponding path metric is larger by 2 because there are 2 errors.
2. The other (mutually exclusive) possibility is that the transmitter was in state '11' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent 01 as the parity bits and tere was one bit error, because we received 00. The path metric of the new state, PM['01', $i+1$] is equal to PM['11', i] + 1.

Formalizing the above intuition, we can see that

$$\text{PM}[s, i+1] = \min(\text{PM}[\alpha, i] + \text{BM}[\alpha \rightarrow s], \text{PM}[\beta, i] + \text{BM}[\beta \rightarrow s]), \tag{7.1}$$

where α and β are the two predecessor states.

In the decoding algorithm, it is important to remember which arc corresponds to the minimum, because we need to traverse this path from the final state to the initial one keeping track of the arcs we used, and then finally *reverse* the order of the bits to produce the most likely message.

■ 7.2.2 Finding the Most Likely Path

We can now describe how the decoder finds the maximum-likelihood path. Initially, state '00' has a cost of 0 and the other $2^{k-1} - 1$ states have a cost of ∞.

The main loop of the algorithm consists of two main steps: first, calculating the branch metric for the next set of parity bits, and second, computing the path metric for the next column. The path metric computation may be thought of as an *add-compare-select* procedure:

1. *Add* the branch metric to the path metric for the old state.
2. *Compare* the sums for paths arriving at the new state (there are only two such paths to compare at each new state because there are only two incoming arcs from the previous column).
3. *Select* the path with the smallest value, breaking ties arbitrarily. This path corresponds to the one with fewest errors.

Figure 7-3 shows the decoding algorithm in action from one time step to the next. This example shows a received bit sequence of 11 10 11 00 01 10 and how the receiver processes it. The fourth picture from the top shows all four states with the same path metric. At this stage, any of these four states and the paths leading up to them are most likely transmitted bit sequences (they all have a Hamming distance of 2). The bottom-most picture shows the same situation with only the *survivor paths* shown. A survivor path is one that has a chance of being the maximum-likelihood path; there are many other paths that can be pruned away because there is no way in which they can be most likely. The reason why the Viterbi decoder is practical is that the number of survivor paths is much, much smaller than the total number of paths in the trellis.

Another important point about the Viterbi decoder is that *future knowledge* will help it break any ties, and in fact may even cause paths that were considered "most likely" at a certain time step to change. Figure 7-4 continues the example in Figure 7-3, proceeding until all the received parity bits are decoded to produce the most likely transmitted message, which has two bit errors.

■ 7.3 Soft Decision Decoding

Hard decision decoding digitizes the received voltage signals by comparing it to a threshold, *before* passing it to the decoder. As a result, we lose information: if the voltage was 0.500001, the confidence in the digitization is surely much lower than if the voltage was 0.999999. Both are treated as "1", and the decoder now treats them the same way, even though it is overwhelmingly more likely that 0.999999 is a "1" compared to the other value.

Soft decision decoding (also sometimes known as "soft input Viterbi decoding") builds on this observation. It *does not digitize the incoming samples prior to decoding*. Rather, it uses a continuous function of the analog sample as the input to the decoder. For example, if the expected parity bit is 0 and the received voltage is 0.3 V, we might use 0.3 (or 0.3^2, or some such function) as the value of the "bit" instead of digitizing it.

For technical reasons that will become apparent later, an attractive soft decision metric is the *square* of the difference between the received voltage and the expected one. If the convolutional code produces p parity bits, and the p corresponding analog samples are $v = v_1, v_2, \ldots, v_p$, one can construct a soft decision branch metric as follows

$$\text{BM}_{\text{soft}}[u, v] = \sum_{i=1}^{p} (u_i - v_i)^2, \tag{7.2}$$

where $u = u_1, u_2, \ldots, u_p$ are the *expected* p parity bits (each a 0 or 1). Figure 7-5 shows the soft decision branch metric for $p = 2$ when u is 00.

With soft decision decoding, the decoding algorithm is identical to the one previously described for hard decision decoding, except that the branch metric is no longer an integer Hamming distance but a positive real number (if the voltages are all between 0 and 1, then the branch metric is between 0 and 1 as well).

It turns out that this soft decision metric is closely related to the *probability of the decoding being correct* when the channel experiences additive Gaussian noise. First, let's look at the simple case of 1 parity bit (the more general case is a straightforward extension). Suppose

the receiver gets the i^{th} parity bit as v_i volts. (In hard decision decoding, it would decode $-$ as 0 or 1 depending on whether v_i was smaller or larger than 0.5.) What is the probability that v_i would have been received given that bit u_i (either 0 or 1) was sent? With zero-mean additive Gaussian noise, the PDF of this event is given by

$$f(v_i|u_i) = \frac{e^{-d_i^2/2\sigma^2}}{\sqrt{2\pi\sigma^2}}, \tag{7.3}$$

where $d_i = v_i^2$ if $u_i = 0$ and $d_i = (v_i - 1)^2$ if $u_i = 1$.

The log likelihood of this PDF is proportional to $-d_i^2$. Moreover, along a path, the PDF of the sequence $V = v_1, v_2, \ldots, v_p$ being received given that a code word $U = u_i, u_2, \ldots, u_p$ was sent, is given by the product of a number of terms each resembling Eq. (7.3). The logarithm of this PDF for the path is equal to the sum of the individual log likelihoods, and is proportional to $-\sum_i d_i^2$. But that's precisely the negative of the branch metric we defined in Eq. (7.2), which the Viterbi decoder minimizes along the different possible paths! Minimizing this path metric is identical to maximizing the log likelihood along the different paths, implying that the soft decision decoder produces the most likely path that is consistent with the received voltage sequence.

This direct relationship with the logarithm of the probability is the reason why we chose the sum of squares as the branch metric in Eq. (7.2). A different noise distribution (other than Gaussian) may entail a different soft decoding branch metric to obtain an analogous connection to the PDF of a correct decoding.

■ 7.4 Achieving Higher and Finer-Grained Rates: Puncturing

As described thus far, a convolutional code achieves a maximum rate of $1/r$, where r is the number of parity bit streams produced by the code. But what if we want a rate greater than $1/2$, or a rate between $1/r$ and $1/(r+1)$ for some r?

A general technique called **puncturing** gives us a way to do that. The idea is straightforward: the encoder does not send every parity bit produced on each stream, but "punctures" the stream sending only a subset of the bits that are agreed-upon between the encoder and decoder. For example, one might use a rate-$1/2$ code along with the puncturing schedule specified as a vector; for example, we might use the vector (101) on the first parity stream and (110) on the second. This notation means that the encoder sends the first and third bits but not the second bit on the first stream, and sends the first and second bits but not the third bit on the second stream. Thus, whereas the encoder would have sent two parity bits for every message bit without puncturing, it would now send four parity bits (instead of six) for every three message bits, giving a rate of $3/4$.

In this example, suppose the sender in the rate-$1/2$ code, without puncturing, emitted bits $p_0[0]p_1[0]p_0[1]p_1[1]p_0[2]p_1[2]\ldots$. Then, with the puncturing schedule given, the bits emitted would be $p_0[0]p_1[0] - p_1[1]p_0[2] - \ldots$, where each $-$ refers to an omitted bit.

At the decoder, when using a punctured code, missing parity bits don't participate in the calculation of branch metrics. Otherwise, the procedure is the same as before. We can think of each missing parity bit as a blank ('$-$') and run the decoder by just skipping over the blanks.

7.5 Performance Issues

There are three important performance metrics for convolutional coding and decoding:

1. How much state and space does the encoder need?
2. How much time does the decoder take?
3. What is the reduction in the bit error rate, and how does that compare with other codes?

7.5.1 Encoder and Decoder Complexity

The first question is the easiest: the amount of space is linear in K, the constraint length, and the encoder is much easier to implement than the Viterbi decoder. The decoding time depends mainly on K; as described, we need to process $O(2^K)$ transitions each bit time, so the time complexity is exponential in K. Moreover, as described, we can decode the first bits of the message only at the very end. A little thought will show that although a little future knowledge is useful, it is unlikely that what happens at bit time 1000 will change our decoding decision for bit 1, if the constraint length is, say, 6. In fact, in practice the decoder starts to decode bits once it has reached a time step that is a small multiple of the constraint length; experimental data suggests that $5 \cdot K$ message bit times (or thereabouts) is a reasonable decoding window, regardless of how long the parity bit stream corresponding to the message it.

7.5.2 Free Distance

The reduction in error probability and comparisons with other codes is a more involved and detailed issue. The answer depends on the constraint length (generally speaking, larger K has better error correction), the number of generators (larger this number, the lower the rate, and the better the error correction), and the amount of noise.

In fact, these factors are indirect proxies for the **free distance**, which largely determines how well a convolutional code corrects errors. Because convolutional codes are linear, everything we learned about linear codes applies here. In particular, the Hamming distance of a linear code, i.e., the minimum Hamming distance between any two valid codewords, is equal to the number of ones in the smallest non-zero codeword with minimum weight, where the weight of a codeword is the number of ones it contains.

In the context of convolutional codes, the smallest Hamming distance between any two valid codewords is called the *free distance*. Specifically, the free distance of a convolutional code is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. Figure 7-6 illustrates this notion with an example. In this example, the free distance is 4, and it takes 8 output bits to get back to the correct state, so one would expect this code to be able to correct up to $\lfloor (4-1)/2 \rfloor = 1$ bit error in blocks of 8 bits, if the block starts at the first parity bit. In fact, this error correction power is essentially the same as an $(8, 4, 3)$ rectangular parity code. Note that the free distance in this example is 4, not 5: the smallest non-zero path metric between the initial 00 state and a future 00 state goes like this: $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics increase as $0 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

Why do we define a "free distance", rather than just call it the Hamming distance, if it is defined the same way? The reason is that any code with Hamming distance D (whether linear or not) can correct all patterns of up to $\lfloor \frac{D-1}{2} \rfloor$ errors. If we just applied the same notion to convolutional codes, we will conclude that we can correct all single-bit errors in the example given, or in general, we can correct some fixed number of errors.

Now, convolutional coding produces an unbounded bit stream; these codes are markedly distinct from block codes in this regard. As a result, the $\lfloor \frac{D-1}{2} \rfloor$ formula is not too instructive because it doesn't capture the true error correction properties of the code. A convolutional code (with Viterbi decoding) can correct $t = \lfloor \frac{D-1}{2} \rfloor$ errors as long as these errors are "far enough apart". So the notion we use is the free distance because, in a sense, errors can keep occurring and as long as no more than t of them occur in a closely spaced burst, the decoder can correct them all.

■ 7.5.3 Comparing Codes: Simulation Results

In addition to the performance different hard decision convolutional codes, an important question is how much better soft decision decoding is compared to hard decision decoding. We address these questions by describing some simulation results here.

Figure 7-7 shows some representative performance results for a set of codes all of the same code rate $(1/2)$.[1] The top-most curve shows the uncoded probability of bit error, which may be modeled using the erfc function. The x axis plots the SNR on the dB scale, as defined in Chapter 5 (lower noise is toward the right). The y axis shows the probability of a decoding error on a log scale.

This figure shows the performance of three codes:

1. The $(8, 4, 3)$ rectangular parity code.

2. A convolutional code with generators $(111, 100)$ and constraint length $K = 3$, shown in the picture as "$K = 3$".

3. A convolutional code with generators $(1110, 1101)$ and constraint length $K = 4$, shown in the picture as "$K = 4$".

Some observations:

1. The probability of error is roughly the same for the rectangular parity code and hard decision decoding with $K = 3$. The free distance of the $K = 3$ convolutional code is 4, which means it can correct one bit error over blocks that are similar in length to the rectangular parity code we are comparing with. Intuitively, both schemes essentially produce parity bits that are built from similar amounts of history. In the rectangular parity case, the row parity bit comes from two successive message bits, while the column parity comes from two message bits with one skipped in between. But we also send the message bits, so we're mimicking a similar constraint length (amount of memory) to the $K = 3$ convolutional code.

[1] You will produce similar pictures in one of your lab tasks using your implementations of the Viterbi and rectangular parity code decoders.

2. The probability of error for a given amount of noise is noticeably lower for the $K = 4$ code compared to $K = 3$ code; the reason is that the free distance of this $K = 4$ code is 6, and it takes 7 trellis edges to achieve that ($000 \rightarrow 100 \rightarrow 010 \rightarrow 001 \rightarrow 000$), meaning that the code can correct up to 2 bit errors in sliding windows of length $2 \cdot 4 = 8$ bits.

3. The probability of error for a given amount of noise is dramatically lower with soft decision decoding than hard decision decoding. In fact, $K = 3$ and soft decoding beats $K = 4$ and hard decoding in these graphs. For a given error probability (and signal), the degree of noise that can be tolerated with soft decoding is much higher (about 2.5–3 dB, which is a good rule-of-thumb to apply in practice for the gain from soft decoding, all other things being equal).

Figure 7-8 shows a comparison of three different convolutional codes together with the uncoded case. Two of the codes are the same as in Figure 7-7, i.e., $(111, 110)$ and $(1110, 1101)$; these were picked because they were recommended by Bussgang's paper. The third code is $(111, 101)$, with parity equations

$$p_0[n] = x[n] + x[n-1] + x[n-2]$$
$$p_1[n] = x[n] + x[n-2].$$

The results of this comparison are shown in Figure 7-8. These graphs show the probability of decoding error (BER after decoding) for experiments that transmit messages of length 500,000 bits each. (Because the BER of the best codes in this set are on the order of 10^{-6}, we actually need to run the experiment over even longer messages when the SNR is higher than 3 dB; that's why we don't see results for one of the codes, where the experiment encountered no errors.)

Interestingly, these results show that the code $(111, 101)$ is stronger than the other two codes, even though its constraint length, 3, is smaller than that of $(1110, 1101)$. To understand why, we can calculate the free distance of this code, which turns out to be 5. This free distance is smaller than that of $(1110, 1101)$, whose free distance is 6, *but* the number of trellis edges to go from state 00 back to state 00 in the $(111, 101)$ case is only 3, corresponding to a 6-bit block. The relevant state transitions are $00 \rightarrow 10 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics are $0 \rightarrow 2 \rightarrow 3 \rightarrow 5$. Hence, its error correcting power is marginally stronger than the $(1110, 1101)$ code.

■ 7.6 Summary

From its relatively modest, though hugely impactful, beginnings as a method to decode convolutional codes, Viterbi decoding has become one of the most widely used algorithms in a wide range of fields and engineering systems. Modern disk drives with "PRML" technology to speed-up accesses, speech recognition systems, natural language systems, and a variety of communication networks use this scheme or its variants.

In fact, a more modern view of the soft decision decoding technique described in this lecture is to think of the procedure as finding the most likely set of traversed states in a *Hidden Markov Model* (HMM). Some underlying phenomenon is modeled as a Markov state machine with probabilistic transitions between its states; we see noisy observations

from each state, and would like to piece together the observations to determine the most likely sequence of states traversed. It turns out that the Viterbi decoder is an excellent starting point to solve this class of problems (and sometimes the complete solution).

On the other hand, despite its undeniable success, Viterbi decoding isn't the only way to decode convolutional codes. For one thing, its computational complexity is exponential in the constraint length, K, because it does require each of these states to be enumerated. When K is large, one may use other decoding methods such as BCJR or Fano's sequential decoding scheme, for instance.

Convolutional codes themselves are very popular over both wired and wireless links. They are sometimes used as the "inner code" with an outer block error correcting code, but they may also be used with just an outer error detection code. They are also used as a component in more powerful codes line turbo codes, which are currently one of the highest-performing codes used in practice.

■ Problems and Exercises

1. Please check out and solve the online problems on error correction codes at `http://web.mit.edu/6.02/www/f2011/handouts/tutprobs/ecc.html`

2. Consider a convolutional code whose parity equations are

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-3] \\ p_1[n] &= x[n] + x[n-1] + x[n-2] \\ p_2[n] &= x[n] + x[n-2] + x[n-3] \end{aligned}$$

   (a) What is the rate of this code? How many states are in the state machine representation of this code?

   (b) Suppose the decoder reaches the state "110" during the forward pass of the Viterbi algorithm with this convolutional code.

      i. How many predecessor states (i.e., immediately preceding states) does state "110" have?

      ii. What are the bit-sequence representations of the predecessor states of state "110"?

      iii. What are the expected parity bits for the transitions from each of these predecessor states to state "110"? Specify each predecessor state and the expected parity bits associated with the corresponding transition below.

   (c) To increase the rate of the given code, Lem E. Tweakit punctures the p_0 parity stream using the vector (1 0 1 1 0), which means that every second and fifth bit produced on the stream are *not sent*. In addition, she punctures the p_1 parity stream using the vector (1 1 0 1 1). She sends the p_2 parity stream unchanged. What is the rate of the punctured code?

3. Let `conv_encode(x)` be the resulting bit-stream after encoding bit-string x with a convolutional code, C. Similarly, let `conv_decode(y)` be the result of decoding y to produce the maximum-likelihood estimate of the encoded message. Suppose we

send a message M using code C over some channel. Let P = `conv_encode(M)` and let R be the result of sending P over the channel and digitizing the received samples at the receiver (i.e., R is another bit-stream). Suppose we use Viterbi decoding on R, knowing C, and find that the maximum-likelihood estimate of M is $\hat{M}$. During the decoding, we find that the minimum path metric among all the states in the final stage of the trellis is D_{min}.

D_{min} is the Hamming distance between _____ and _____. Fill in the blanks, explaining your answer.

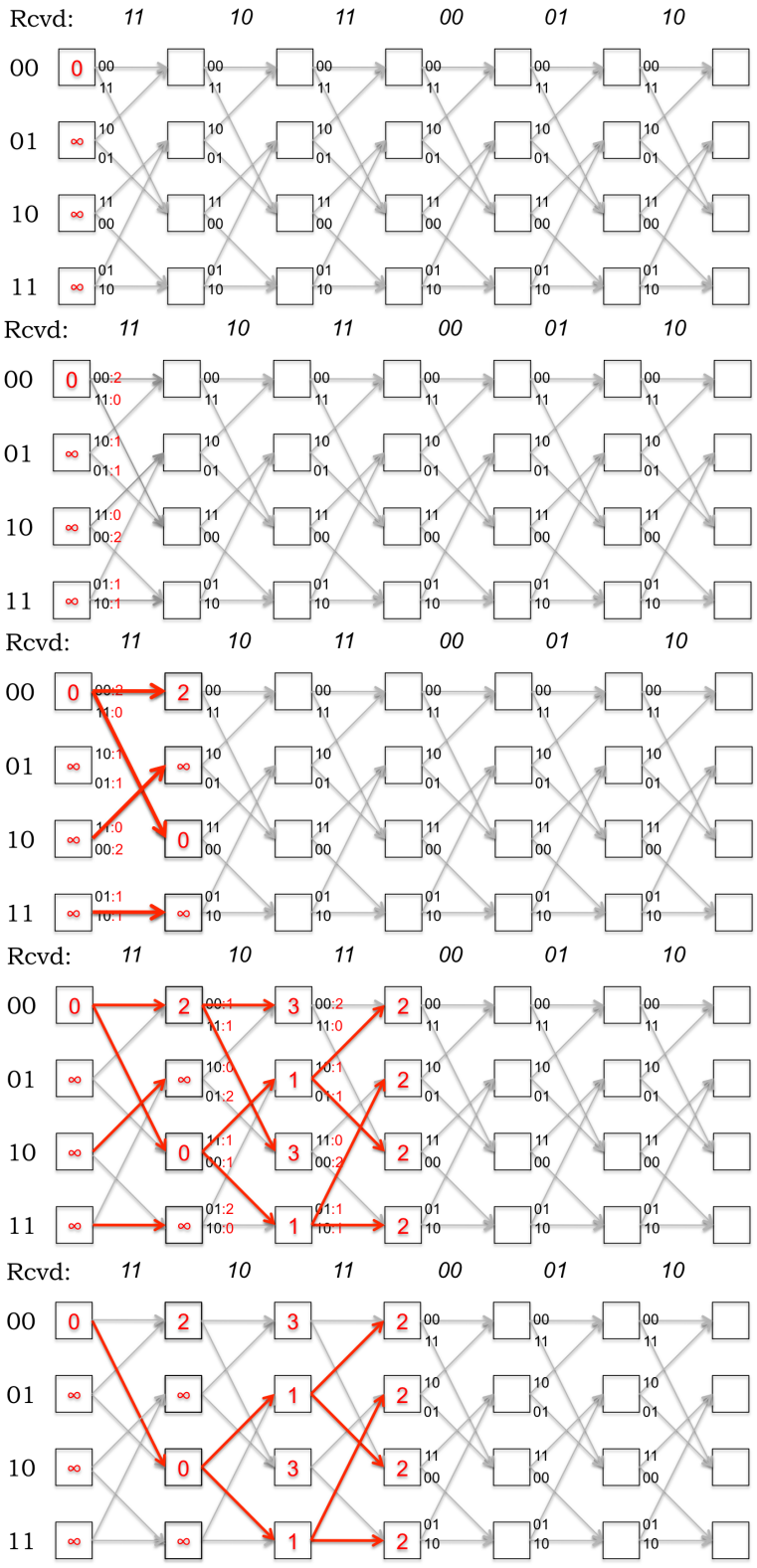


Figure 7-3: The Viterbi decoder in action. This picture shows four time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown.

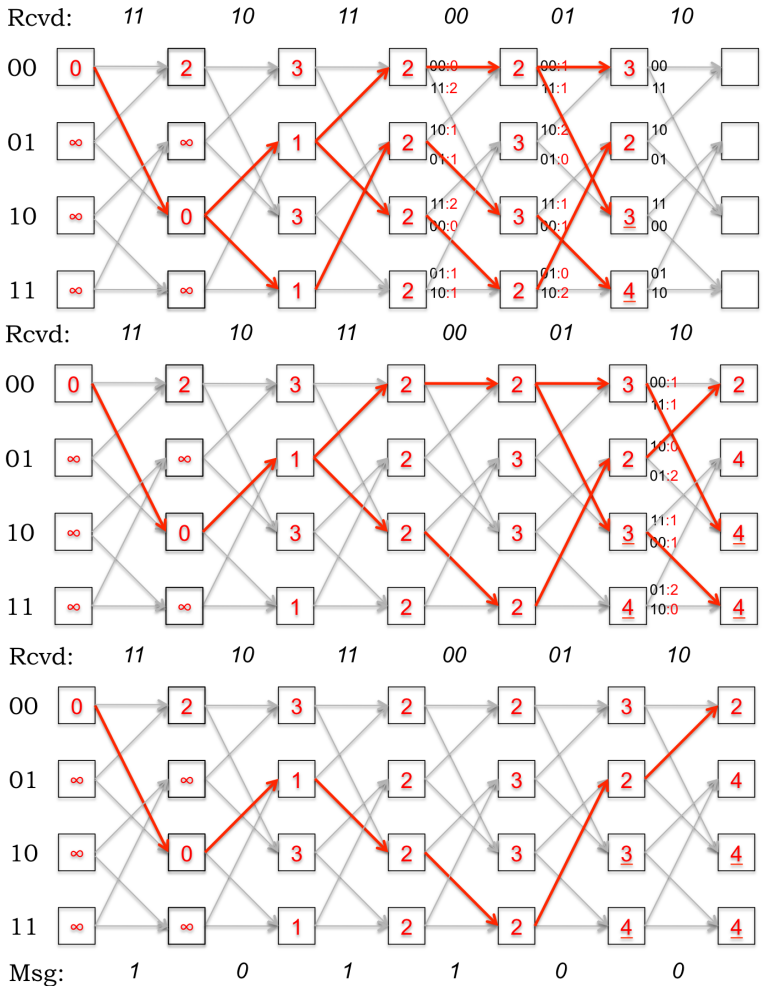


Figure 7-4: The Viterbi decoder in action (continued from Figure 7-3. The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remeber the arc that got us to this state, so that the backward pass can be done properly.

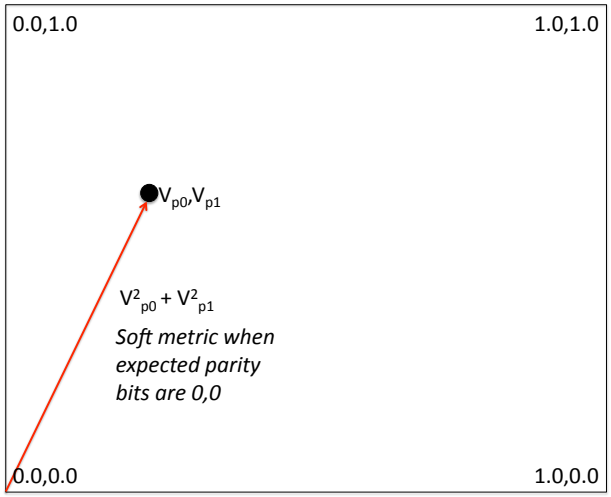
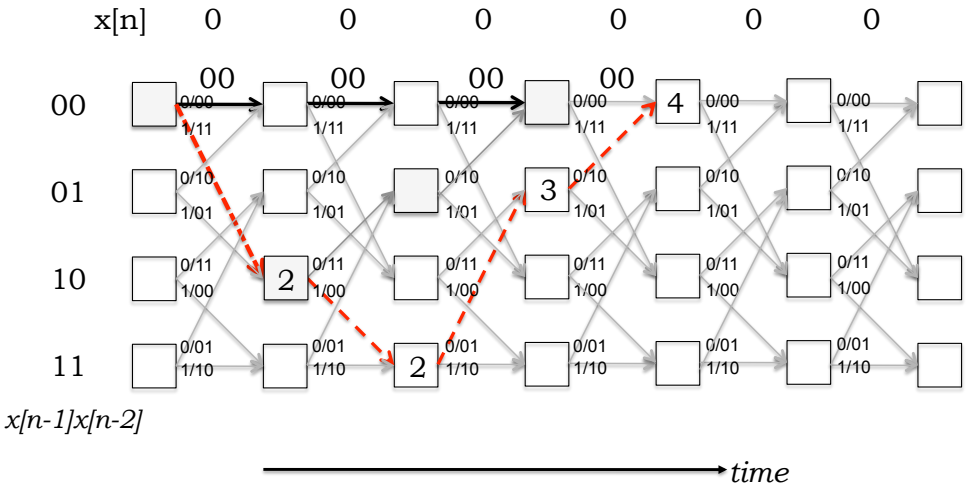


Figure 7-5: Branch metric for soft decision decoding.



The free distance is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. It is 4 in this example. The path 00 → 10 →01 → 00 has a shorter length, but a higher path metric (of 5), so it is not the free distance.

Figure 7-6: The free distance of a convolutional code.

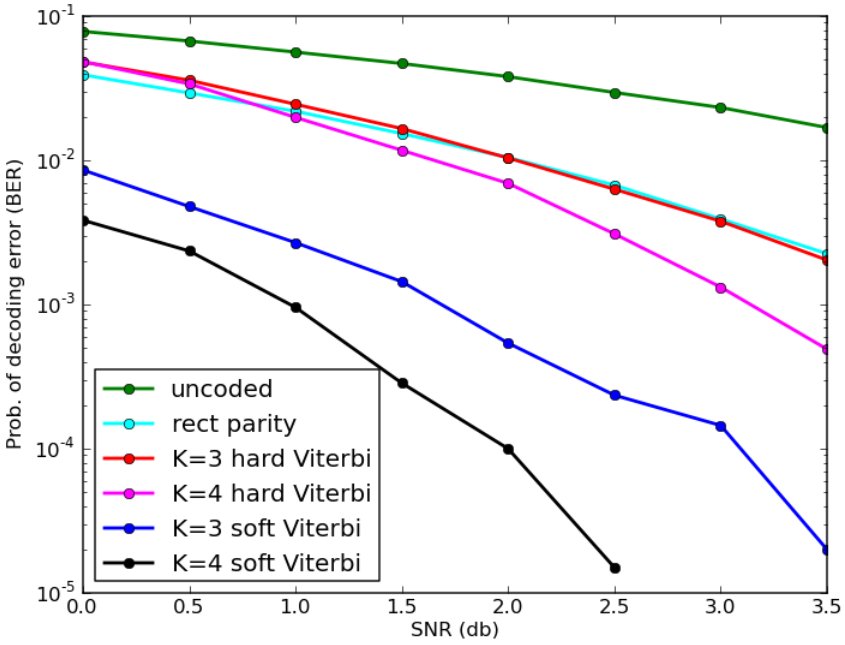


Figure 7-7: Error correcting performance results for different rate-1/2 codes.

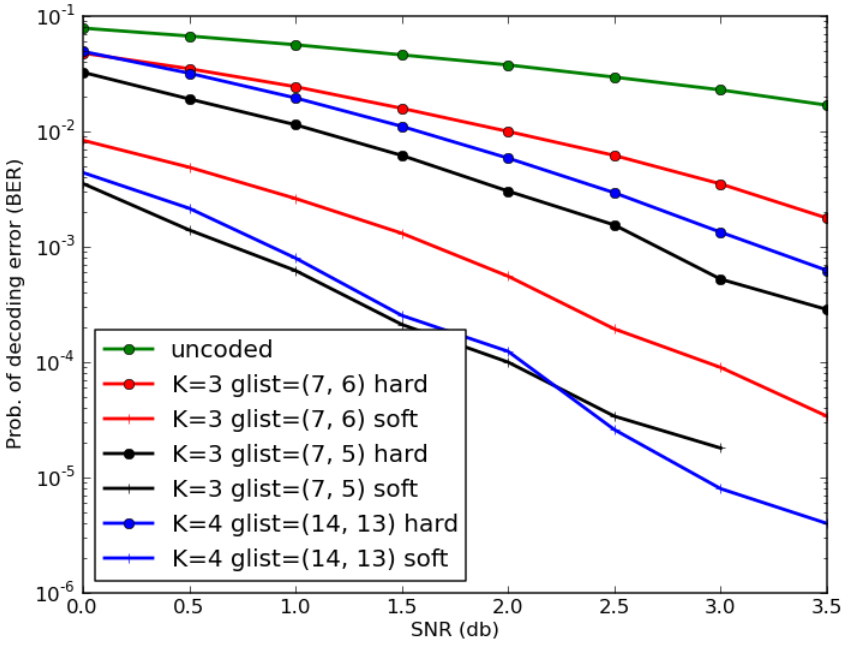


Figure 7-8: Error correcting performance results for three different rate-1/2 convolutional codes. The parameters of the three convolutional codes are $(111, 110)$ **(labeled "**$K = 3$ **glist=**$(7, 6)$**"),** $(1110, 1101)$ **(labeled "**$K = 4$ **glist=**$(14, 13)$**"), and** $(111, 101)$ **(labeled "**$K = 3$ **glist=**$(7, 5)$**"). The top three curves below the uncoded curve are for hard decision decoding; the bottom three curves are for soft decision decoding.**