Top Ten Algorithms Class 6

John Burkardt
Department of Scientific Computing
Florida State University

....

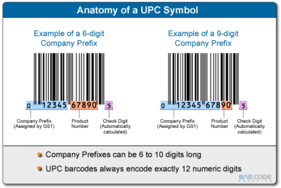
http://people.sc.fsu.edu/∼jburkardt/classes/tta_2015/class06.pdf

05 October 2015

Our Current Algorithm List

1. Bernoulli number calculation
2. Bootstrap algorithm
3. Data stream: most common item
4. Euclid's greatest common factor algorithm
5. Monte Carlo Sampling
6. PageRank algorithm for ranking web pages
7. Pancake flipping algorithm for genome relations
8. Path counting with the adjacency matrix
9. Power method for eigenvector problems
10. Probability evolution with the transition matrix
11. Prototein model of protein folding
12. Search engine indexing
13. Trees for computational biology

Feifei Xu: UPC and QR Codes


Anatomy of a UPC Symbol
Example of a 6-digit Company Prefix
Example of a 9-digit Company Prefix
0 12345 67890 5
0 12345 67890 5
Company Prefix (Assigned by GS1)
Product Number
Check Digit (Automatically calculated)
Company Prefix (Assigned by GS1)
Product Number
Check Digit (Automatically calculated)
Company Prefixes can be 6 to 10 digits long
UPC barcodes always encode exactly 12 numeric digits
BAR CODE

Nick Berry, “Wounded QR codes”, DataGenetics blog, Nov 2013.

UPC and QR codes must store and deliver information despite physical damage, low light, bad angles, viewer motion.

- How does a UPC (Uniform Price Code) work?
- Why are QR codes better than UPC’s?
- How does a QR code store information?
- What is an error correction code?
- Can codes handle bad light, bad angle, missing bits?

Error Detection

A computer has three tasks:

1. compute data;
2. store data;
3. transmit and receive data.

We need to worry about items 2 and 3!

The original ASCII code for storing characters allowed 7 bits for data, plus an 8-th parity bit.

Character	Decimal	7 bits	Parity bit
@	64	1000000	1
A	65	1000001	0
B	66	1000010	0
C	67	1000011	1
D	68	1000100	0

Computer memory also included parity bits.

Error Detection

Parity bits are a simple response to errors. In this case, if a single bit changed in the word, the sum of the bits becomes odd, and the error has been detected.

Of course, this system only catches single bit errors, and can't tell us how to correct the error.

We will see various ways of dealing with the existence of error, depending on what kinds of errors are typical, how badly we want to avoid them, and how much overhead we are willing to pay.

Repetition

Suppose we are transmitting a bank balance over a noisy line. Just like when talking in a noisy room, we can ask that the message be repeated. Even if the message is never transmitted exactly correctly, we can probably retrieve it:

```
  #1:  $ 5, 2 7 8, 5 2 3. 3 4
  #2:  $ 2, 3 6 8, 5 2 3. 7 9
  #3:  $ 5, 3 7 8, 5 8 3. 7 6
  #4:  $ 9, 9 7 6, 4 2 1. 3 1
  #5:  $ 5, 3 0 8, 5 9 4. 3 4
-----------------------------
Most:  $ 5, 3 7 8, 5 2 3. 3 4
```

Notice that the final "4" occurs most often, but is not the majority value! But the noiser the line, the more repetitions we will need. The longer the message, the more repetitions we will need.

Checksum

For our numerical example, an error that changes one digit to another is undetectable. Assuming that errors aren't all that common, it would be nice to force simple errors to stick out. A simple method is to append an extra chunk of data as an error detector, called a checksum.

Suppose we are transmitting a string of numbers:

```
4 6 7 5 6
```

Sum these numbers (28), use the remainder mod 10 (8) as a tag:

```
4 6 7 5 6 3  Rejected
4 5 7 5 6 8  Rejected
4 6 7 5 6 8  Accepted (correctly)
2 6 7 6 6 6  Accepted (incorrect, but 2 errors)
```

Checksum

Taking the remainder base 10 of the digits in a string is an example of a checksum or hash value.

This can be regarded as a function $f(\text{text}) = \text{checksum}$ whose goal is to produce a short, easily examined value that serves as a signature or quality label or verification. For the remainder function, if any single digit is changed, the discrepancy will be visible.

However, a more sophisticated checksum would be useful, because the second most common error, made by humans, is to transpose two digits. Such an error will not be detected by a checksum of the remainder of the sum.

Checksum Examples

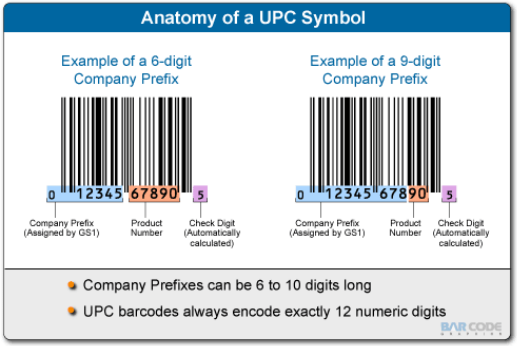
UPC: Uniform Product Code, 12 digits, digit 12 is the checksum.

$$3d_1+d_2+3d_3+d_4+3d_5+d_6+3d_7+d_8+3d_9+d_{10}+3d_{11}+d_{12} = 0 \mod (10$$

1−23456−78999−9

6−39382−00039−3

0−80047−44069−4



Anatomy of a UPC Symbol
Example of a 6-digit Company Prefix
0 12345 67890 5
Company Prefix (Assigned by GS1)
Product Number
Check Digit (Automatically calculated)
Example of a 9-digit Company Prefix
0 12345 67890 5
Company Prefix (Assigned by GS1)
Product Number
Check Digit (Automatically calculated)
Company Prefixes can be 6 to 10 digits long
UPC barcodes always encode exactly 12 numeric digits
BARCODE

Compute UPC Check Digit

```
  s = '12345678901'   <-- incomplete string
%
%  Turn characters into digits.
%
  for i = 1 : 11
    dvec(i) = str2num ( s(i) )
  end
%
%  Compute sum.
%
  dsum = sum ( dvec(1:2:11) ) * 3 + sum ( dvec(2:2:10) );
%
%  Mod.
%
  dsum = mod ( dsum, 10 );
%
%  C + DSUM = 0, mod 10
%
  c = mod ( 10 - dsum, 10 );
%
%  Append C to string
%
  s = [ s, num2str ( c ) ];
```

Listing 1: UPC Check Digit in MATLAB

Checksum Examples

ISBN: International Standard Book Number, 10 digits, last is the checksum, and can have the value 'X' meaning 10:

$$10d_1+9d_2+8d_3+7d_4+6d_5+5d_6+4d_7+3d_8+2d_9+1d_{10}=0 \mod (11)$$

```
0-691-15819-8
0-387-94677-2
0-898-71317-X
2-1234-5680-2
```

9 782123 456803

ISBN-10: 2-1234-5680-2

ISBN-13: 978-2-1234-5680-3

Checksum Examples

Luhn/IBM, any number of digits. Used on credit cards.

$$2\#d_1 + d_2 + 2\#d_3 + d_4 + 2\#d_5 + d_6 + 2\#d_7 + d_8 + \ldots = 0 \quad \text{mod } (10)$$

Here, $2\#d_1$ means compute $2 * d_1$ and sum the digits.

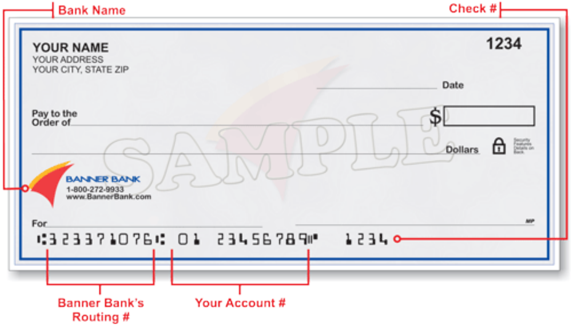
```
799-273-9871-3
499-273-9871-6
123-456-781-234-567-0
```

Checksum Examples

US Bank Routing Numbers: 9 digits, last digit is checksum.

$$3d_1 + 7d_2 + d_3 + 3d_4 + 7d_5 + d_6 + 3d_7 + 7d_8 + d_9 = 0 \quad \text{mod } (10)$$

323–371–076



Redundancy

One of the problems with our data is that any change in the data creates new data that looks just as legitimate as the original. But suppose our message literally spelled out the bank balance?

```
Sent:    fice threm s.ven ei#7t give two ttree thxxe foux
Assumed:five three seven eight five two three three four
```

It takes more than a single error to turn a "five" into a "four", so the original message is still recognizable, even after most of the words in it have been hit with one or two errors.

Redundancy, which is cheaper than repetition, means adding something extra to a message that makes it more resilient.

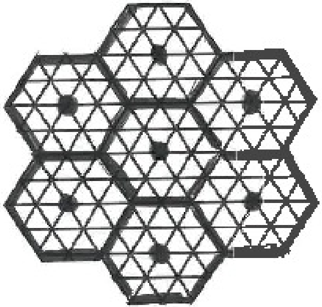
In fact, we were not only able to detect errors in this single copy of the message, but also to correct them.

Redundancy Example

If 12345 is our message, the “distance” to 22345, or 12845 is just 1 - a single error gives a valid, but erroneous, message.

What if we space our valid numbers, so if one digit is altered, we can correct it because there’s only one valid answer nearby?

If hexagon centers are our data, and an error moves us one node away, this arrangement corrects 1 bit errors and detects 2 bit errors.



Hamming Codes

This is the idea behind Hamming codes.

Suppose we want to transmit words of 4 bits (binary 0000 to 1111). The Hamming (7,4) code transforms each word w into a 7 bit word x, which is then transmitted. The receiver gets a 7 bit value y (which might be x, or maybe x with one bit damaged.) recovers a value z, which we hope is equal to w!

```
w -----> x ---------> y ------> z
  code     transmit     decode
```

The transformation from data word to code word is accomplished by a 7x4 matrix G, so that

$$x = G * w \pmod 2$$

Hamming Codes

The matrix G has the form:

$$\begin{matrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

and a typical transformation $x = G * w$ would be:

$$\begin{matrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} * \begin{matrix} 0 \\ 1 \\ 0 \\ 1 \end{matrix} = \begin{matrix} 2 \\ 1 \\ 0 \\ 2 \\ 1 \\ 0 \\ 1 \end{matrix} = \begin{matrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{matrix}$$

Hamming Codes - Error Detection and Correction

The matrix H has the form:

```
1  0  1  0  1  0  1
0  1  1  0  0  1  1
0  0  0  1  1  1  1
```

Our word $w = (0,1,0,1)$ became codeword $x = (0,1,0,0,1,0,1)$ and was received as y. Compute $H * y$ to detect errors:

```
H * (0,1,0,0,1,0,1) (mod 2 ) = (0,0,0)
H * (1,1,0,0,1,0,1) (mod 2 ) = (1,0,0) (bit #1 is wrong!)
H * (0,0,0,0,1,0,1) (mod 2 ) = (0,1,0) (bit #2 is wrong!)
H * (0,1,1,0,1,0,1) (mod 2 ) = (1,1,0) (bit #3 is wrong!)
...
H * (0,1,0,0,1,0,0) (mod 2 ) = (1,1,1) (bit #7 is wrong!)
```

Hamming Codes - Convert Corrected Codeword to Data

The matrix R has the form:

```
0 0 1 0 0 0 0
0 0 0 0 1 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 1
```

Our codeword y has been corrected, but it contains 7 bits. We can recover the original 4 bit codeword by $z = R * y$:

```
R * (0,1,0,0,1,0,1) (mod 2 ) = (0,1,0,1)
```

...or you can just notice that, for this code, the data is in columns 3, 5, 6 and 7.

Hamming Codes

```
 0: (0 0 0 0) -->  (0 0 0 0 0 0 0)
 1: (0 0 0 1) -->  (1 1 0 1 0 0 1)
 2: (0 0 1 0) -->  (0 1 0 1 0 1 0)
 3: (0 0 1 1) -->  (1 0 0 0 0 1 1)
 4: (0 1 0 0) -->  (1 0 0 1 1 0 0)
 5: (0 1 0 1) -->  (0 1 0 0 1 0 1)
 6: (0 1 1 0) -->  (1 1 0 0 1 1 0)
 7: (0 1 1 1) -->  (0 0 0 1 1 1 1)
 8: (1 0 0 0) -->  (1 1 1 0 0 0 0)
 9: (1 0 0 1) -->  (0 0 1 1 0 0 1)
10: (1 0 1 0) -->  (1 0 1 1 0 1 0)
11: (1 0 1 1) -->  (0 1 1 0 0 1 1)
12: (1 1 0 0) -->  (0 1 1 1 1 0 0)
13: (1 1 0 1) -->  (1 0 1 0 1 0 1)
14: (1 1 1 0) -->  (0 0 1 0 1 1 0)
15: (1 1 1 1) -->  (1 1 1 1 1 1 1)
```

Hamming Codes

Given two strings of n digits, the Hamming distance is the number of digits that differ. The Hamming distance measures how easily one data word can be corrupted into another.

Before, 0000 had a Hamming distance of 1 to the values 0001, 0010, 0100, 1000. Now, using the Hamming code, it is not possible to alter (0000000) to another value without making at least 3 changes. This increased distance is true for any pair of 7 bit codewords.

Hamming codes can be created for data of any size, and as the data size increases, the relative amount of overhead from parity bits decreases. For instance, H(72,64) can handle 64 bits of data with 8 parity bits.

Reed-Solomon Codes

Instead of Hamming codes, Reed-Solomon codes are used for error correction in CDs, DVDs, QR codes and data transmission protocols such as DSL and WiMAX.

The n bits of data $(d_1, d_2, ..., d_n)$ are used as polynomial coefficients:

$$p(x) = \sum_{i=1}^{n} d_i * x^{i-1}$$

and the codeword is the sequence of polynomial values at $n + q$ points (with arithmetic carried out over a specific field).

We can afford to lose up q bits of the message...because the polynomial can be reconstructed from its value at any n points.

We can also detect and correct up to $\frac{q}{2}$ bit errors.

Student Volunteer?

Several telecom companies are fighting for a contract to maintain cellphone towers. One difficulty is that they must assign a frequency to each tower, in such a way that nearby towers don't use the same one.

I have discovered such a scheme, and I want to offer it to the highest bidder. Until I have all the bids, I don't want to reveal my scheme, otherwise any of the bidders can steal it. But the bidders don't want to bid unless they believe I really have a scheme that works.

Is there a way to convince each bidder that I've got a scheme that works, without revealing the actual scheme?

Reference: Matthew Green, *Zero Knowledge Proofs*, http://blog.cryptographyengineering.com/2014/11/zero-knowledge-proofs-illustrated-primer.html

Student Volunteer?

I am watching a radar screen that is updated every minute.

The radar screen is a 500x500 array of pixels which are OFF or ON.

The radar signal is noisy, so that each pixel has a chance of turning on even though there's nothing there.

I know there are two intruders somewhere on my screen. One is not moving, and one is moving at a constant speed and direction. After observing the screen for 30 minutes, can I detect the location of either intruder?

Reference: Nick Berry, *Detecting Targets in Noisy Radar Signals*, http://datagenetics.com/blog/may22014/index.html