Huffman

20CS6025 Data Encoding

Yizong Cheng

1-7-13

The Class

- 20CS6025 Data Encoding
- TR 11:00-12:20
- Instructor: Yizong Cheng
  - yizong.cheng@uc.edu, 513-556-1809, 536 ERC
- Textbook: Stallings: Cryptography and Network Security, 5th ed. 2011 Prentice Hall
  - We may not use this as much as
- e-book: Salomon and Motta: Handbook of Data Compression, 5th ed. 2010 Springer

e-Book

David Salomon
Giovanni Motta

With Contributions by David Bryant

Handbook of Data Compression

Fifth Edition

Data datum 1646

- Textual
- Multimedia
- Structured
  - tables, graphs, sequences
- Collections
  - documents, video, audio, sensor data
  - evolving data
- Publications
  - Algorithms, mathematical theorems and proofs, music compositions, patents, …

Data Encoding encode 1919

- Computerization
  - storage, retrieval, analysis
- Data integrity, security, authentication
- Data compression
  - lossless
  - lossy
- Data summarization, visualization
- Data monitoring and mining

Kolmogorov Complexity of Data

- Andrey Kolmogorov 1963
- The size of the program needed to generate the data is the complexity of the data.
- Jorma Risannen 1978
- Minimum Description Length (MDL)
- A machine/language must be assumed.

Variable-Length Codes

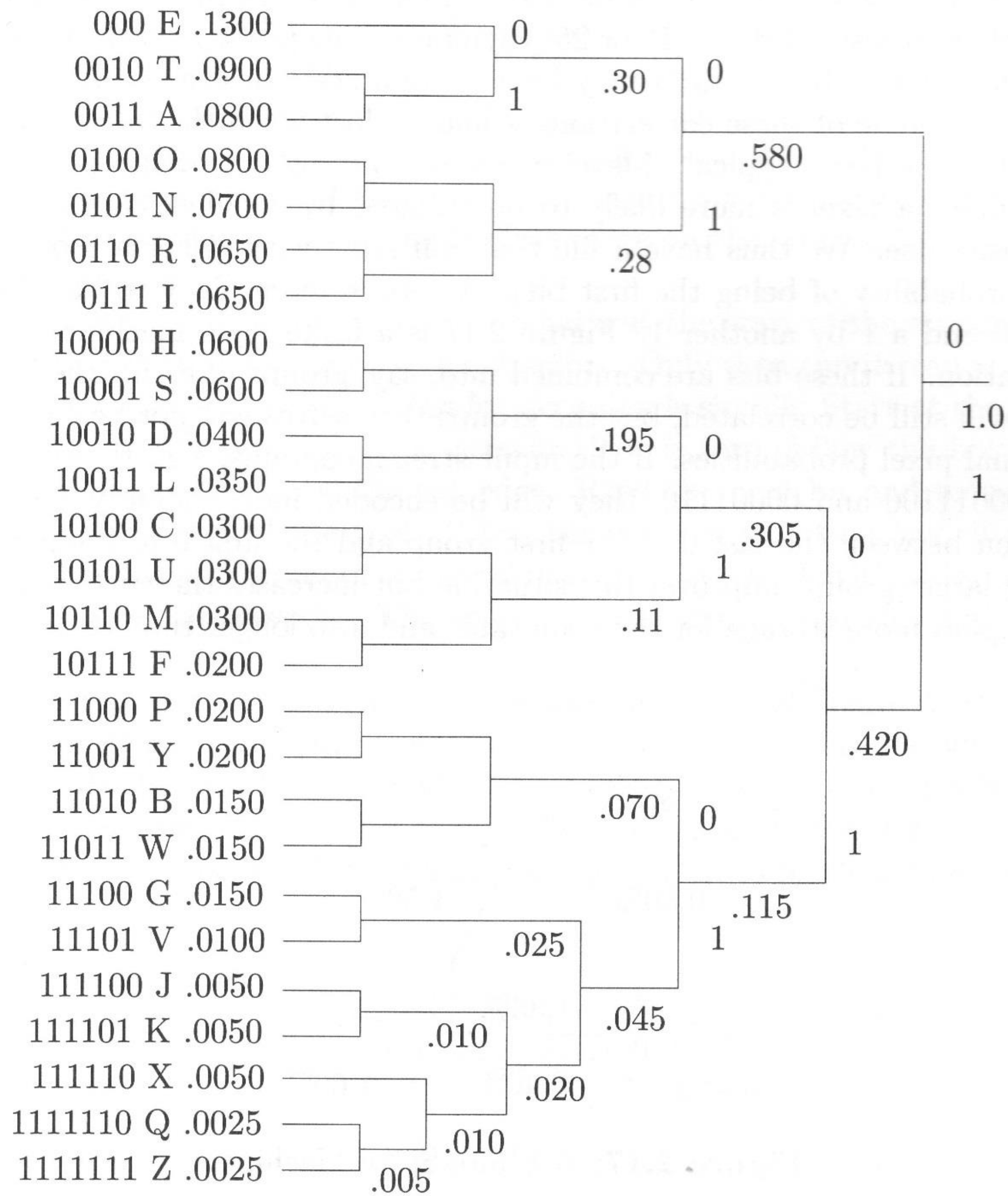
- A text is a sequence of symbols from an alphabet of size K.
- If fixed-length code is used to replace symbols we need $\log_2 K$ bits for each symbol.
- What about the idea of using longer bit strings for rare symbols and shorter ones for common symbols?

Prefix Code

- If variable-length code is used, how does one know the end of each code word?
- The prefix property: no codeword is a prefix of another codeword.
- Each codeword is a leaf of a binary tree.

Huffman code

- Start with K subtrees (leaves) for K symbols
- Group the two subtrees with the smallest probabilities into one whose probability is the sum of that of the two component subtrees.
- Repeat this until there is only one tree with probability 1.


000 E .1300
0010 T .0900
0011 A .0800
0100 O .0800
0101 N .0700
0110 R .0650
0111 I .0650
10000 H .0600
10001 S .0600
10010 D .0400
10011 L .0350
10100 C .0300
10101 U .0300
10110 M .0300
10111 F .0200
11000 P .0200
11001 Y .0200
11010 B .0150
11011 W .0150
11100 G .0150
11101 V .0100
111100 J .0050
111101 K .0050
111110 X .0050
1111110 Q .0025
1111111 Z .0025
0
1
.30
0
.580
1
.28
0
1.0
1
.195
0
.305
0
1
.11
.420
.070
0
1
.115
1
.025
.045
.010
.020
.010
.005

Compression Ratio

- The compression ratio (average number of bits per symbol) is the average path length to a leaf in the binary tree, given the probabilities of visiting each leaf (the probability of each symbol).
- Theorem: The average path length is greater than or equal to the entropy $-\Sigma_t p_t \log_2 p_t$.

Proof

- $\Sigma_t p_t \log_2 p_t <= \Sigma_t p_t l_t$ is equivalent to
- $\Sigma_t p_t \log_2(2^{-lt}/p_t) <= 0$.
- $\log_2 x$ is below its tangent at $x=1$, or
- $\log_2 x <= \quad (x-1)\log_2 e$. Therefore,
- $\Sigma_t p_t \log_2(2^{-lt}/p_t) <= \Sigma_t p_t (2^{-lt}/p_t - 1)\log_2 e$
- $= (\Sigma_t 2^{-lt} - 1) \log_2 e$.
- We can show $\Sigma_t 2^{-lt} <= 1$ (Kraft’s Inequality).

Kraft's Inequality

- $\Sigma_t 2^{-lt} = 2^{-1}(\Sigma_{\text{left}} 2^{-(lt-1)} + \Sigma_{\text{right}} 2^{-(lt-1)})$
- By induction on the depth of the binary tree
- $<= \quad 2^{-1}(1+1) = 1.$

Limitations to Huffman Code

- Huffman code reaches entropy only when all symbol probabilities are powers of 1/2.
- Can be improved by arithmetic coding.
- Other problems with Huffman Code
  - definition of a “symbol”
  - assumption that symbols are independent events
- So “entropy” of data depends on many assumptions.

David Huffman (1925-1999)

Being originally from Ohio, it is no wonder that Huffman went to Ohio State University for his BS (in electrical engineering). What is unusual was his age (18) when he earned it in 1944. After serving in the United States Navy, he went back to Ohio State for an MS degree (1949) and then to MIT, for a PhD (1953, electrical engineering).

That same year, Huffman joined the faculty at MIT. In 1967, he made his only career move when he went to the University of California, Santa Cruz as the founding faculty member of the Computer Science Department. During his long tenure at UCSC, Huffman played a major role in the development of the department (he served as chair from 1970 to 1973) and he is known for his motto "my products are my students." Even after his retirement, in 1994, he remained active in the department, teaching information theory and signal analysis courses.

Encoding Steps

- Count frequencies of symbols (bytes) in file.
- Build the Huffman tree for the symbols so that the expected path length to a leaf is as close to the entropy as possible.
- Generate codewords for all symbols.
- Copy the Huffman tree to the compressed file.
- Go through the original file and emit the codeword bitstream for each symbol in the file into the compressed file.

Decoding Steps

- Read the Huffman tree from the compressed file.
- Read one bit a time from the compressed file and move down the Huffman tree until a leaf is reached and emit the symbol represented by the leaf.
- Repeat this until the end of the compressed file.

Java Implementation

- H1A.java for encoding
  - Assume at most 256 symbols (bytes).
  - Count frequencies of symbols in file with count().
  - Build the Huffman tree using a priority queue that has the subtree with the smallest count at the top.
  - Codewords generated by depth-first traversal of the Huffman tree.
  - A double array version of the tree is generated and emitted to the output.
  - Each symbol in original file is replaced with its codeword as bitstream emitted to the output.

```
void count(String filename){
    byte[] buffer = new byte[blockSize];
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(filename);
    } catch (FileNotFoundException e){
      System.err.println(filename + " not found");
      System.exit(1);
    }
    int len = 0;
    for (int i = 0; i < numberOfSymbols; i++) freq[i] = 0;
   try {
    while ((len = fis.read(buffer)) >= 0){
     for (int i = 0; i < len; i++){
       int symbol = buffer[i];
       if (symbol < 0) symbol += 256;
       freq[symbol]++;
      }
    }
    fis.close();
   } catch (IOException e){
      System.err.println("IOException"); System.exit(1);
   }
```

Computing Entropy

- Entropy = $-\sum_t p_t \log_2 p_t$

```
double entropy = 0;
double log2 = Math.log(2.0);
for (int i = 0; i < numberOfSymbols; i++) if (freq[i] > 0){
 double prob = freq[i] * 1.0 / totalLen;
 entropy += prob * Math.log(prob) / log2;
}
```

Tree Building

- Priority queue Q.
- Initially, each symbol as a subtree with its frequency count is inserted into Q.
- Repeat until Q contains only one subtree
  – Take two subtrees with the smallest counts out
  – Merge the two into a subtree with count the sum of the two subtree counts
  – Insert the resulting subtree into Q.
- The remaining one subtree in Q is the root tree.

Huffman Tree Building

```
class Node implements Comparable{
  Node left, right;
  int symbol;
  int frequency;
  public Node(Node l, Node r, int s, int f){
    left = l; right = r; symbol = s; frequency = f;
  }
  public int compareTo(Object obj){
   Node n = (Node)obj;
   return frequency - n.frequency;
  }
}
```

```
Global:
 Node tree = null;
```

```
void makeTree(){  // make Huffman prefix codword tree
   PriorityQueue<Node> pq = new PriorityQueue<Node>();
   for (int i = 0; i < numberOfSymbols; i++) if (freq[i] > 0){
       actualNumberOfSymbols++;
       pq.add(new Node(null, null, i, freq[i]));
   }
   while (pq.size() > 1){
     Node a = pq.poll(); Node b = pq.poll();
     pq.add(new Node(a, b, -1, a.frequency + b.frequency));
   }
   tree = pq.poll();
 }
```

Node

Node	
Symbol	Frequency
left child	right child

Parent Node	
-1	n1 + n2
left child	right child

0

1

Left Child	
Symbol	n1
-1	-1

Right Child	
Symbol	n2
-1	-1

Codewords Generation

```
Global:
 String[] codewords = new
String[numberOfSymbols];
```

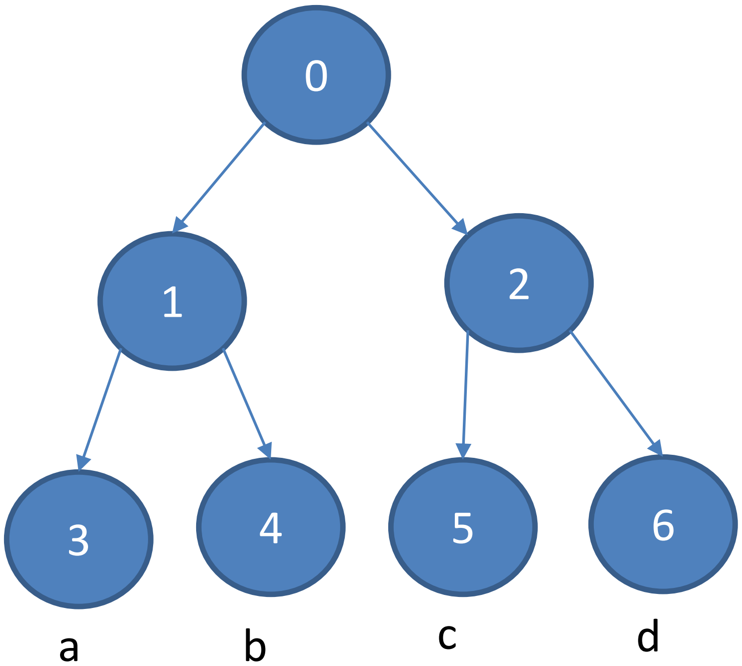
```
void dfs(Node n, String code){
// recursive depth-first search to be started as dfs(tree,””)
    if (n.symbol < 0){
      dfs(n.left, code + "0"); dfs(n.right, code + "1");
    }else codewords[n.symbol] = code;
  }

In main():
    H1A h1 = new H1A();
    h1.count(args[0]);
    h1.makeTree();
    h1.dfs(h1.tree, "");
```

Huffman Code

10 1001100
32 00
33 10111100001110
39 10111100000
40 10111100011111
41 10111100001000
44 100111
45 1011110001111001
46 0100111
58 01001100
59 101111011
63 1001000001
65 10011011
66 1011110001
67 10010000001
68 100110100
69 0100011001
70 10111101010
71 010001110
72 0100011111
73 01001101
74 010001101
75 101111000101
76 101111001
77 0100011110
78 10010000101
79 100110101
80 10010000100
81 1011110001111000
82 100100010
83 1011110100
84 100100011
85 10111100001001
86 101111000111101
87 10111101011
89 101111000110
90 100100001100
97 1000
98 010000
99 011000
100 10110
101 1111
102 110110
103 010010
104 1010
105 11100
106 0100011000
107 0100010
108 01101
109 101110
110 0101
111 0111
112 1011111
113 100100001101
114 11010
115 11101
116 1100
117 110111
118 1001001
119 100101
120 10010000000
121 011001
122 10010000111

Array Implementation of Binary Tree

left subtree index	right subtree index
1	2
3	4
5	6
0	‘a’
0	‘b’
0	‘c’
0	‘d’



Codetree is a Double Array

```
void buildTree(){
 codetree = new int[actualNumberOfSymbols * 2 - 1][2];
 int treeSize = 1;
 for (int i = 0; i < actualNumberOfSymbols * 2 - 1; i++)
   codetree[i][0] = codetree[i][1] = 0;
 for (int i = 0; i < numberOfSymbols; i++)
  if (codewords[i] != null){
   int len = codewords[i].length();
   int k = 0;
   for (int j = 0; j < len; j++){
     int side = codewords[i].charAt(j) - '0';
     if (codetree[k][side] <= 0)
        codetree[k][side] = treeSize++;
     k = codetree[k][side];
   }
   codetree[k][1] = i;
 }
}
```

Output Double Array

```
void outputTree(){
   System.out.write(actualNumberOfSymbols);
   for (int i = 0; i < actualNumberOfSymbols * 2 - 1; i++){
     System.out.write(codetree[i][0]);
     System.out.write(codetree[i][1]);
   }
 }
```

```
In main():
    h1.buildTree();
    h1.outputTree();
    h1.encoding(args[0]);
```

```
void encoding(String filename){
    byte[] buffer = new byte[blockSize];
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(filename);
    } catch (FileNotFoundException e){
      System.err.println(filename + " not found");
      System.exit(1);
    }
    int len = 0;
   try {
    while ((len = fis.read(buffer)) >= 0){
      for (int i = 0; i < len; i++){
       int symbol = buffer[i];
       if (symbol < 0) symbol += 256;
       outputbits(codewords[symbol]);
      }
    }
    fis.close();
   } catch (IOException e){
      System.err.println("IOException");  System.exit(1);
   }
    if (position > 0){ System.out.write(buf); size++; }
    System.out.flush();
}
```

Output a Codeword as a Bitstring

```
void outputbits(String bitstring){
    for (int i = 0; i < bitstring.length(); i++){
     buf <<= 1;
     if (bitstring.charAt(i) == '1') buf |= 1;
     position++;
     if (position == 8){
        position = 0;
        System.out.write(buf);
        size++;
        buf = 0;
     }
    }
 }
```

```
Globals:
 int buf = 0; int position = 0;
```

Decoder A1B.java

```
void readTree(){  // read Huffman tree
 try{
  actualNumberOfSymbols = System.in.read();
  codetree = new int[actualNumberOfSymbols * 2 - 1][2];
  for (int i = 0; i < actualNumberOfSymbols * 2 - 1; i++){
    codetree[i][0] = System.in.read();
    codetree[i][1] = System.in.read();
  }
 } catch (IOException e){
    System.err.println(e);
    System.exit(1);
 }
}
```

Reading One Bit from System.in

```
int inputBit(){
   if (position == 0)
     try{
       buf = System.in.read();
       if (buf < 0){ return -1;
 }
       position = 0x80;
     }catch(IOException e){
        System.err.println(e);
        return -1;
     }
   int t = ((buf & position) == 0) ? 0 : 1;
   position >>= 1;
   return t;
 }
```

```
Globals:
 int buf = 0;
 int position = 0;
```

H1B main() and decode()

```
void decode(){
  int bit = -1;  int k = 0;
  while ((bit = inputBit()) >= 0){
    // your four lines of code?
  }
  System.out.flush();
 }

 public static void main(String[] args){
  H1B h1 = new H1B();
  h1.readTree();
  h1.decode();
 }
```

Decoding a Compressed File

- int bit; int k = 0;
- while ((bit = readBit()) >= 0) ...
- k = codeTree[k][bit];
- // Move down the code tree with index k
- // until a leaf
- if (codeTree[k][bit] < 0) // reached a leaf
- System.out.write(codeTree[k][1])
- // output symbol also need start moving down again from the root of the tree.

Homework 1: due 1-19-15

- Complete the decode() function in H1B.java.
- Run H1B to decompress a given compressed file and show the resulting file.
- Find the number of bytes in the uncompressed file and that in the compressed one and compute the compression ratio.
- Add the computation of entropy (slide 20) to H1A or implement your own H1C with it to find the entropy of the uncompressed file.
- Compare the entropy and the compression ratio.