UNIT 7B
Data Representation: Compression

Last Lecture

• Binary Trees
  – Binary search trees, max-heaps
• Graphs
Undirected and Directed Graphs

node names

some measure for the path such as its cost

Graphs in Ruby

inf = 1.0/0.0

graph =
[[ 0, 6, 7, 5 ],
 [ 6, 0, 4, inf ],
 [ 7, 4, 0, 3 ],
 [ 5, inf, 3, 0 ]]
Original Graph

A Minimal Spanning Tree
The minimum total cost to connect all vertices using edges from the original graph without using cycles. (minimum total cost = 34)

For example, what would be the minimum cost for laying cables such that all cities are connected?
Shortest Paths from Pittsburgh

The total costs of the shortest path from Pittsburgh to every other location using only edges from the original graph.

REPRESENTING NUMBERS
Unsigned Integers

• With 8 bits

<table>
<thead>
<tr>
<th></th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
</table>

• The minimum value we can represent is 0
• The maximum we value can represent is 255
• The total number of distinct values we can represent is $2^8 = 256$

Signed Integers

• Every bit represents a power of 2 except the “left-most” bit, which represents the sign of the number (0 = positive, 1 = negative)
• Example for positive integer (8 bits):

\[
\begin{array}{cccccccc}
0 & + & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & + & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[
32 + 16 + 4 = +52
\]
Negative Integers

• What about negative numbers?

• We define negative numbers as additive inverse: \(-x\) is the number \(y\) such that \(x + y = 0\).

• 00110100 is +52 but 10110100 is not negative -52 because adding these would not give 0.

Two’s complement example

Adding \(n\) to \(-n\) gives 0
For example: 011 + 101 = 000
REPRESENTING TEXT

ASCII Example

• The ASCII code for “M” is 4D hexadecimal.
• Conversion from base 16 to base 2:

<table>
<thead>
<tr>
<th>hex</th>
<th>binary</th>
<th>hex</th>
<th>binary</th>
<th>hex</th>
<th>binary</th>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
<td>8</td>
<td>1000</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
<td>9</td>
<td>1001</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
<td>A</td>
<td>1010</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
<td>B</td>
<td>1011</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

• 4D (hex) = 0100 1101 (binary) = 77 (decimal)
  (leftmost bit can be used for parity)
ASCII table

- 27 characters presented in a $2^3 \times 2^4$ table.
- Values are represented in hexadecimal (base 16).
- ASCII code for “M” is 4D (hex).

COMPRESSSION
Fixed-Width Encoding

• In a fixed-width encoding scheme, each character is given a binary code with the same number of bits.
  – Example: Standard ASCII is a fixed width encoding scheme, where each character is encoded with 7 bits. This gives us $2^7 = 128$ different codes for characters.

Fixed-Width Encoding

• Given a character set with $n$ characters, what is the minimum number of bits needed for a fixed-width encoding of these characters?
  – Since a fixed width of $k$ bits gives us $n$ unique codes to use for characters, where $n = 2^k$.
  – So given $n$ characters, the number of bits needed is given by $k = \lceil \log_2 n \rceil$. (We use the ceiling function since $\log_2 n$ may not be an integer.)
  – Example: To encode just the alphabet A-Z using a fixed-width encoding, we would need $\lceil \log_2 26 \rceil = 5$ bits: e.g. A $\Rightarrow$ 00000, B $\Rightarrow$ 00001, C $\Rightarrow$ 00010, ..., Z $\Rightarrow$ 11001.
Using Fixed-Width Encoding

- If we have a fixed-width encoding scheme using \( n \) bits for a character set and we want to transmit or store a file with \( m \) characters, we would need \( mn \) bits to store the entire file.
- Can we do better?
  - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.

The Hawaiian Alphabet

- The Hawaiian alphabet consists of 13 characters.
  - ’ is the okina which sometimes occurs between vowels (e.g. KAMA’AINA)
- The table to the right shows each character along with its relative frequency in Hawaiian words.
The Huffman Tree

• We use a tree structure to develop the unique binary code for each letter.
• Start with each letter/frequency as its own node:

```
\begin{itemize}
  \item ‘ : 0.068
  \item A : 0.262
  \item E : 0.072
  \item H : 0.045
  \item I : 0.084
  \item K : 0.106
  \item L : 0.044
  \item M : 0.032
  \item N : 0.083
  \item O : 0.106
  \item P : 0.030
  \item U : 0.059
  \item W : 0.009
\end{itemize}
```

The Huffman Tree

• Combine lowest two frequency nodes into a tree with a new parent with the sum of their frequencies.
The Huffman Tree

• Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.
The Huffman Tree

• Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies...
• Repeat until you have one tree with all nodes linked in.
- Label all left branches with 0 and all right branches with 1

- The binary code for each character is obtained by following the path from the root to the character.
Examples:
H $\Rightarrow$ 0001
A $\Rightarrow$ 10
P $\Rightarrow$ 110011

Fixed Width vs. Huffman Coding

<table>
<thead>
<tr>
<th>Character</th>
<th>Fixed Width Code</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td>A</td>
<td>0001</td>
<td>A 10</td>
</tr>
<tr>
<td>E</td>
<td>0010</td>
<td>E 1101</td>
</tr>
<tr>
<td>H</td>
<td>0011</td>
<td>H 0001</td>
</tr>
<tr>
<td>I</td>
<td>0100</td>
<td>I 1111</td>
</tr>
<tr>
<td>K</td>
<td>0101</td>
<td>K 001</td>
</tr>
<tr>
<td>L</td>
<td>0110</td>
<td>L 0000</td>
</tr>
<tr>
<td>M</td>
<td>0111</td>
<td>M 11000</td>
</tr>
<tr>
<td>N</td>
<td>1000</td>
<td>N 1110</td>
</tr>
<tr>
<td>O</td>
<td>1001</td>
<td>O 010</td>
</tr>
<tr>
<td>P</td>
<td>1010</td>
<td>P 110011</td>
</tr>
<tr>
<td>U</td>
<td>1011</td>
<td>U 0110</td>
</tr>
<tr>
<td>W</td>
<td>1100</td>
<td>W 110010</td>
</tr>
</tbody>
</table>

ALOHA

<table>
<thead>
<tr>
<th>Character</th>
<th>Fixed Width Code</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>0000</td>
<td>0111</td>
</tr>
<tr>
<td>A</td>
<td>0001</td>
<td>A 10</td>
</tr>
<tr>
<td>E</td>
<td>0010</td>
<td>E 1101</td>
</tr>
<tr>
<td>H</td>
<td>0011</td>
<td>H 0001</td>
</tr>
<tr>
<td>I</td>
<td>0100</td>
<td>I 1111</td>
</tr>
<tr>
<td>K</td>
<td>0101</td>
<td>K 001</td>
</tr>
<tr>
<td>L</td>
<td>0110</td>
<td>L 0000</td>
</tr>
<tr>
<td>M</td>
<td>0111</td>
<td>M 11000</td>
</tr>
<tr>
<td>N</td>
<td>1000</td>
<td>N 1110</td>
</tr>
<tr>
<td>O</td>
<td>1001</td>
<td>O 010</td>
</tr>
<tr>
<td>P</td>
<td>1010</td>
<td>P 110011</td>
</tr>
<tr>
<td>U</td>
<td>1011</td>
<td>U 0110</td>
</tr>
<tr>
<td>W</td>
<td>1100</td>
<td>W 110010</td>
</tr>
</tbody>
</table>

Fixed Width:
0001 0110 1001 0011 0001
20 bits

Huffman Code:
10 0000 010 0001 10
15 bits
Variable Length Codes

• In a fixed-width code, the boundaries between letters are fixed in advance:
  0001 0110 1001 0011 0001

• With a variable-length code, the boundaries are determined by the letters themselves.
  – No letter’s code can be a prefix of another letter.
  – Example: since A is “10”, no other letter’s code can begin with “10”. All the remaining codes begin with “00”, “01”, or “11”.

Programming the Huffman Tree

• Let’s write Ruby code to produce a Huffman encoding of an alphabet.
• At each step we need to find the two nodes with the lowest frequency scores.
• This will be easy if nodes are kept in a list that is sorted by score value.
• Solution: use a priority queue.
Priority Queues

NOTE: For this unit, you will need RubyLabs set up and you will need to include BitLab (see p. 167)

- A priority queue (PQ) is like an array that is sorted.
  
  ```ruby
  pq = PriorityQueue.new
  => []
  ```

- To add element into the priority queue in its correct position, we use the `<<` operator:
  
  ```ruby
  pq << "peach"
pq << "apple"
pq << "banana"
=> ["apple", "banana", "peach"]
  ```

Priority Queues (cont’d)

- To remove the first element from the priority queue, we will use the `shift` method:
  
  ```ruby
  fruit1 = pq.shift
  => "apple"
pq
  => ["banana", "peach"]
  ```
Tree Nodes

- We can store all of the node data into a 2-dimensional array:

  ```
  table = [ ["'", 0.068], ["A", 0.262],
           ["E", 0.072], ["H", 0.045], ["I", 0.084],
           ["K", 0.106], ["L", 0.044], ["M", 0.032],
           ["N", 0.083], ["O", 0.106], ["P", 0.030],
           ["U", 0.059], ["W", 0.009] ]
  ```

- A tree node consists of two values, the character and its frequency. Making one of the tree nodes:

  ```python
  char = table[2].first  # "E"
  freq = table[2].last   # 0.072
  node = Node.new(char, freq)
  ```

Building a PQ of Single Nodes

```python
def make_pq(table):
    pq = PriorityQueue.new
    for item in table:
        char = item.first  # "E"
        freq = item.last  # 0.072
        node = Node.new(char, freq)
        pq << node
    return pq
```

Remember: each item in the table is a 2-element array with a character and a frequency.
Building our Priority Queue

pq = make_pq(table)
=> [(W: 0.009 ), (P: 0.030 ),
    (M: 0.032 ), (L: 0.044 ),
    (H: 0.045 ), (U: 0.059 ),
    ('.: 0.068 ), (E: 0.072 ),
    (N: 0.083 ), (I: 0.084 ),
    (K: 0.106 ), (O: 0.106 ),
    (A: 0.262 )]

This is our priority queue showing the 13 nodes in sorted order based on frequency.

Building a Huffman Tree

(Slightly different than book version fig 7.9)

def build_tree(pq)
    while pq.length > 1
        node1 = pq.shift
        node2 = pq.shift
        pq << Node.combine(node1, node2)
    end
    return pq.first
end

Creates a new node with node1 as its left child and node2 as its right child
Building our Huffman Tree

tree = build_tree(pq)

=> ( 1.000 ( 0.428 ( 0.195 ( 0.089
( L: 0.044 ) ( H: 0.045 ) ) ( K: 0.106 ) )
( 0.233 ( O: 0.106 ) ( 0.127 ( U: 0.059 )
( ': 0.068 ) ) ) ) ( 0.572 ( A: 0.262 )
( 0.310 ( 0.143 ( 0.071 ( M: 0.032 )
( 0.039 ( W: 0.009 ) ( P: 0.030 ) ) )
( E: 0.072 ) ) ( 0.167 ( N: 0.083 )
( I: 0.084 ) ) ) ) )

This is just our Huffman tree expressed using recursively nested parenthetical components:
( root ( left ) ( right ) )

Examples:
H => 0001
A => 10
P => 110011
Assigning Codes, Encoding & Decoding

ht = assign_codes(tree)

ht["W"] => 110010
ht["A"] => 10

msg = encode("ALOHA", tree)
=> 100000010000110
decode(msg, tree)
=> "ALOHA"

from BitLab takes a Huffman tree and returns a hash table that maps each letter to its binary code

Note the [] syntax. This returns the code associated with the character from the hash table.

from BitLab encode and decode functions

Next Lecture

• Representing images and sound