ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Huffman Encoding

The basic idea

Consider the sentence “go go gophers”

- this example is from the handout I have placed on moodle

- To store this in a text file, each character is stored with an ASCII representation in 8 bits.
  - For example, ‘g’ has ASCII code 103, which in binary representation is 1100111
  - The sentence will be stored in 13 * 8 = 104 bits in the file (remember a space is an 8 bit character too)
- We could reduce the amount of storage if we stored each character in fewer than 8 bits
  - Since there are only 7 characters used in this string, it is easy to see that we could encode the different characters using only 3 bits for each
  - Give them the codes 1 through 7, which are all represented in binary with 3 bits
  - Then the file can be stored in 13 * 3 bits, or 39 bits (actually we can only write to the file in 8-bit chunks so will need 40 bits – but the last bit should be ignored)
- As long as the de-coder of the file had the same table of characters and their bit-representations as the encoder, the file could easily be de-coded bit-wise
- However, this is only useful for files which do not use all 255 possible ASCII characters – because we need 8 bytes to store the code 255
- Huffmans idea goes even further: We could reduce the size of the file even more if characters that appear most frequently in the text are stored in a smaller no of bits than characters which appear less frequently.

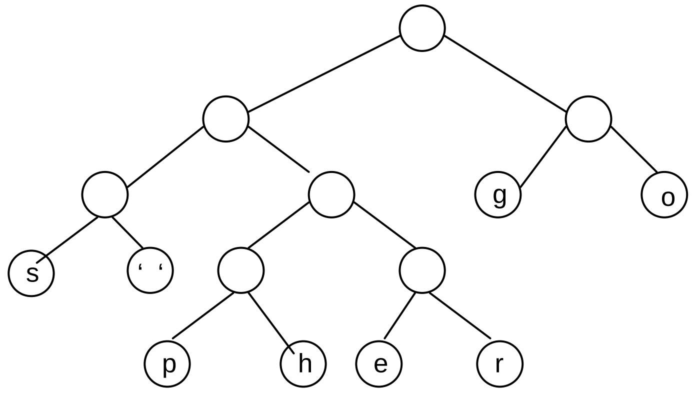
Variable length bit-encoding

Characters that appear most frequently in the text are stored in a smaller no of bits than characters which appear less frequently.

- Consider the “go go gophers” example again
- If the ‘g’ and the ‘o’, which occur most frequently, are stored in just 2 bits each, then the total no of bits needed should be reduced even more.
  - Morse code uses this idea – it uses dots and dashes to represent characters (cp 0’s and 1’s), and the frequently used letters like ‘e’ and ‘t’ have a much shorter code than ones which hardly ever get needed like ‘z’
- Say ‘g’ is 10 and ‘o’ is 11. we need to be sure that none of the other characters start with 10 or 11, so we will probably need 4 bits to represent some of them.
  - But as long as the 4-bit characters occur less frequently than the 2 bit ones, we will have saved some space.
  - Morse code doesn’t worry about this: it uses a ‘pause’ instead to separate the characters.
  - We cant use an option like that, because we can only store 0 or 1 in each bit (never a ‘pause’ signal)

Using a coding tree

- This is a special kind of binary tree called a ‘trie’
- We figure out the sequence of 0s and 1s by tracing the path through the tree to get to each character.
  - Add a 0 to the sequence each time we go left
  - Add a 1 to the sequence each time we go right
- We need to be sure that whenever we find a character in the tree, we know it’s the character we want, and not the start of some other character
  And that’s easy to do if the characters are **only at the leaf nodes of the tree,** like this.



You can see that the ‘g’ and ‘o’ will be stored in 2 bits each, the ‘s’ and the ‘ ‘ in 3 bits, and the other characters in 4 bits

Creating the map of codes for each character

1. traverse the tree in a pre-order way to find each leaf node
2. Store the sequence of right – left moves as you go (as a string of 0s and 1s)
3. When you reach a leaf node, record its character and the sequence you took to reach it.
4. Each character, and the string of 0s and 1s to reach it can be stored in a map

A (huffman) Binary Tree

Variants on the pre-order traversal may be written to supply whatever functionality is required (e.g. build up a coding map in a Huffman tree,)

```
class HuffmanTreeNode
{
  friend class HuffmanTree;
  public:
    HuffmanTreeNode(DATATYPE
   theData);
    bool isLeaf();
  private:
     HuffmanTreeNode *leftChild;
     DATATYPE data;
     HuffmanTreeNode *rightChild;
   };
```

```
class HuffmanTree
{
  public:
    HuffmanTree();
    // void preOrderTraversal();
   //example of a specialised pre-order traversal
   CodingMap getCodingMap();
  protected:
   //this will be used internally by the traversal above
    //void preOrderTraversal (BTreeNode *subTreeRoot)
     // specialised traversals may have other args,
    // depending what has to happen when a node is visited,
   e.g.
   void getCodingMap(BTreeNode *subTreeRoot, string
                              codeStrToHere, CodingMap & mapSoFar);

   HuffmanTreeNode *root;
    };
```

Building the optimal coding tree

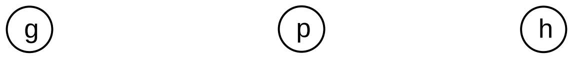
- Decide for each character what the frequency of its occurrence is in the text
- The coding tree will store 2 bits of information:
  - a pointer to its root node
  - the total frequency with which the characters in the tree occur in the text (this is the *weight* of the tree)
- Start by producing a separate tree for each character like this

Tree with weight 3 and a root node only

Tree with weight 1 and a root node only

Tree with weight 1 and a root node only

… and so on



- Then start taking 2 trees with the lowest possible weights, and combining them
- Create a new root node for the combined tree, and attach the present roots as the left and right children
- Look at the diagrams on the Huffman handout to see how to do this

Reading a character from a file

- You may need the `get()` method of the `istream` class.
- There are many overloaded versions of this method
  - In its simplest form, pass `get()` a *reference* to a character variable
  - the variable will be updated within the method
  - so on return from the call it contains the next character read from the file

```
char nextchar;
//code to open a file for reading
// lets say you have called it infile
infile.get(nextchar);
```

- now `nextchar` holds the next char read from the file.
  - It may be any character
  - including, for example, a space or a newline char or the eof char
  - Or just a ‘regular’ character