Applications of Trees

- First application: coding and data compression
- We will define optimal variable-length binary codes and code trees
- We will study Huffman's algorithm which constructs them
- Huffman's algorithm is an example of a Greedy Algorithm, an important class of simple optimisation algorithms

Text, Codes, and Compression

- Computer systems represent data as bit strings
- Encoding: transformation of data into bit strings
- Decoding: transformation of bit strings into data
- The code defines the transformation

Text, Codes, and Compression

- For example: ASCII, the international coding standard, uses a 7-bit code
- HEX Code - Character
- 20 <space>
- 41 A
- 42 B
- 61 a

Text, Codes, and Compression

- Such encodings are called
  - fixed-length or
  - block codes
- They are attractive because the encoding and decoding is extremely simple
  - For coding, we can use a block of integers or codewords indexed by characters
  - For decoding, we can use a block of characters indexed by codewords

Text, Codes, and Compression

- For example: the sentence
  The cat sat on the mat

  is encoded in ASCII as

  1010100 110100 011001 0101 .....
- Note that the spaces are there simply to improve readability ... they don't appear in the encoded version.

Text, Codes, and Compression

- The following bit string is an ASCII encoded message:

  10001001100101110001111011111100100
  11010011101110110011101000001101001
  11100110100000110010111000011110011
  1111001

Text, Codes, and Compression

- And we can decode it by chopping it into smaller strings each of 7 bits in length and by replacing the bit strings with their corresponding characters:

  1000100(D)1100101(e)1100011(c)1101111(o)1100100(d)1101001(i)1101110(n)1100111(g)0100000()1101001(i)1110011(s)0100000()1100101(e)1100001(a)1110011(s)1111001(y)

Text, Codes, and Compression

- Every code can be thought of in terns of
- a finite alphabet of source symbols
- a finite alphabet of code symbols
- Each code maps every finite sequence or string of source symbols into a string of code symbols

Text, Codes, and Compression

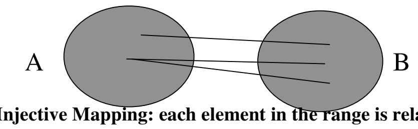
- Let A be the source alphabet
- Let B be the code alphabet
- A code f is an injective map

  f: $S_A \rightarrow S_B$

- where S_A is the set of all strings of symbols from A
- where S_B is the set of all strings of symbols from B

Text, Codes, and Compression

- Injectivity ensures that each encoded string can be decoded uniquely (we do not want two source strings that are **encoded** as the same string)



Injective Mapping: each element in the range is related to at most one element in the domain

Text, Codes, and Compression

- We are primarily interested in the code alphabet {0, 1} since we want to code source symbols strings as bit strings

Text, Codes, and Compression

- There is a problem with block codes: n symbols produce *nb* bits with a block code of length b
- For example,
  - if n = 100,000 (the number of characters in a typical 200-page book)
  - b = 7 (e.g. 7-bit ASCII code)
  - then the characters are encoded as 700,000 bits

Text, Codes, and Compression

- While we cannot encode the ASCII characters with fewer than 7 bits
- We can encode the characters with a different number of bits, depending on their frequency of occurrence.
-
-
-

Mr. T. Lysaght NUI, Maynooth

Text, Codes, and Compression

- First problem with variable length codes:
  - when scanning an encoded text from left to right (decoding it)
  - How do we know when one codeword finishes and another starts?
- We require that each codeword not be a prefix of any other codeword
- So, for the binary code alphabet, we should base the codes on binary code trees

Mr. T. Lysaght NUI, Maynooth

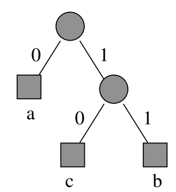
Text, Codes, and Compression

- Binary code trees:
- binary tree whose external nodes are labelled uniquely with the source alphabet symbols
-
-

Mr. T. Lysaght NUI, Maynooth

Text, Codes, and Compression

A binary code tree and its **prefix** code



Mr. T. Lysaght NUI, Maynooth

Text, Codes, and Compression

- The codeword corresponding to a symbol is the bit string given by the path from the root to the external node labeled with the symbol
- Note that, as required, no codeword is a prefix for any other codeword
  -
  -

Mr. T. Lysaght NUI, Maynooth

Text, Codes, and Compression

- Codes that satisfy the prefix property are called **prefix** codes
- Prefix codes are important because
  -
  -
  -

Mr. T. Lysaght NUI, Maynooth

Text, Codes, and Compression

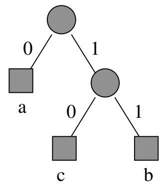
- Read the encoded message bit by bit
- Start at the root
- if the bit is a 0, move left
- if the bit is a 1, move right
- if the node is external, output the corresponding symbol and begin again at the root

Text, Codes, and Compression

- Encoded message:

0 0 1 1 1 0 0

- Decoded message:



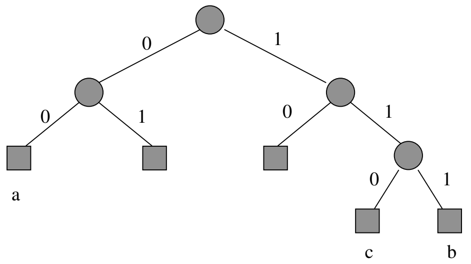
Optimal Variable-Length Codes

- What makes a good variable length code?
- Let A = $\{a_1, ..., a_n\}$, n>=1, be the alphabet of source symbols
- Let P = $\{p_1, ..., p_n\}$, n>=1, be their probability of occurrence
-

Optimal Variable-Length Codes

- Any binary tree with ***n*** external nodes labelled with the ***n*** symbols defines a prefix code
- Any prefix code for the n symbols defines a binary tree with at least n external nodes
-
-

Non-Reduced Prefix Code (Tree)



Optimal Variable-Length Codes

- Comparison of prefix codes - compare the number of bits in the encoded text
- Let A = $a_1, ..., a_n$, n >=1, be the alphabet of source symbols
- Let P = $p_1, ..., p_n$ be their probability of occurrence
- Let W = $w_1, ..., w_n$ be a prefix code for A = $a_1, ..., a_n$
- Let L = $l_1, ..., l_n$ be the lengths of W = $w_1, ..., w_n$

Optimal Variable-Length Codes

- Given a source text T with $f_1, ..., f_n$ occurrences of $a_1, ..., a_n$ respectively
- The **total number of bits** when T is encoded is
  $\sum^{n}_{i=1} f_i l_i$
- The **total number of source symbols** is
  $\sum^{n}_{i=1} f_i$
- The average length of the W-encoding is:

Optimal Variable-Length Codes

- For long enough texts, the probability p_i of a given symbol occurring is approximately
  $p_i = f_i \,/\, \sum^{n}_{i=1} f_i$
- So the expected length of the W-encoding is:
  –

Optimal Variable-Length Codes

- To compare two different codes W_1 and W_2 we can compare either
  – Alength(T, W_1) and Alength(T, W_2) or
  – Elength(W_1, P) and Elength(W_2 , P)
- We say W_1 is no worse than W_2 if
  Elength(W_1, P) <= Elength(W_2 , P)
- We say W_1 is optimal if :
  –

Optimal Variable-Length Codes

- Huffman's Algorithm
- We wish to solve the following problem:
- Given n symbols
  $A = a_1, ..., a_n$, n >= 1
  and the probability of their occurrence
  $P = p_1, ..., p_n$, respectively,
  construct an optimal prefix code for A and P

Optimal Variable-Length Codes

- This problem is an example of a global optimization problem
- Brute force (or exhaustive search) techniques are too expensive to compute:
-

Optimal Variable-Length Codes

- This algorithm takes $O(n^n)$ time, where n is the size of the alphabet
- Why? because any binary tree of size n-1 (i.e. with n external nodes) is a valid reduced prefix tree and there are n! ways of labelling the external nodes
-

Optimal Variable-Length Codes

- Huffman's Algorithm is only $O(n^2)$
- This is significant: if n = 128 (number of symbols in a 7-bit ASCII code)
- $O(n^n) = 128^{128} = 5.28 \times 10^{269}$
- $O(n^2) = 128^2 = 1.6384 \times 10^4$
- There are 31536000 seconds in a year and if we could compute 1000 000 000 steps a second then the brute force technique would still take 1.67×10^{253} years

Optimal Variable-Length Codes

- The age of the universe is estimated to be between 7 and 20 billion years, i.e.,

  7×10^9 and 20×10^9 years

- A long way off 1.67×10^{253} years!

Optimal Variable-Length Codes

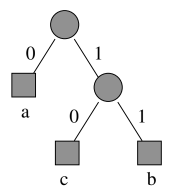
- Huffman's Algorithm uses a technique called Greediness
- It uses local optimization to achieve a globally optimum solution
  –
  –
  –

Optimal Variable-Length Codes

- Before we begin, note that we'd like a tree with the symbols which have the lowest probability to be on the longest path
- Why?
-
  –
  –

Text, Codes, and Compression

A binary code tree and its prefix code

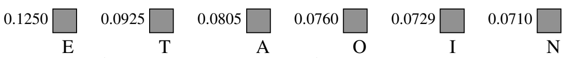


Huffman's Algorithm

- We will treat Huffman's Algorithm for just six letters, i.e, n = 6, and there are six symbols in the source alphabet.
- These are, with their probabilities,
  – E - 0.1250
  – T - 0.0925
  – A - 0.0805
  – O - 0.0760
  – I - 0.0729
  – N - 0.710

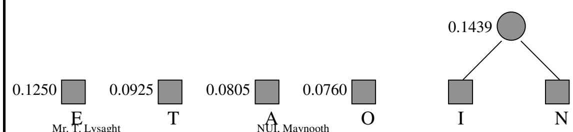
Huffman’s Algorithm

- Step 1:
- Create a forest of code trees, one for each symbol
- Each tree comprises a single external node (empty tree) labelled with its symbol and weight (probability)



Huffman’s Algorithm

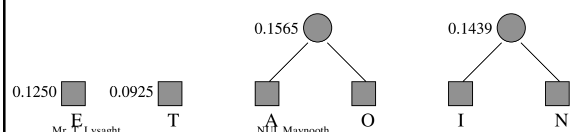
- Step 2:
  - Choose the two binary trees, B1 and B2, that have the smallest weights
  -



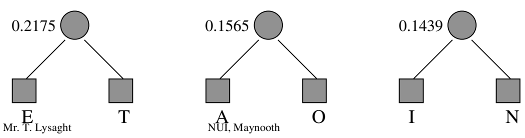
Huffman’s Algorithm

- Step 3:
  -

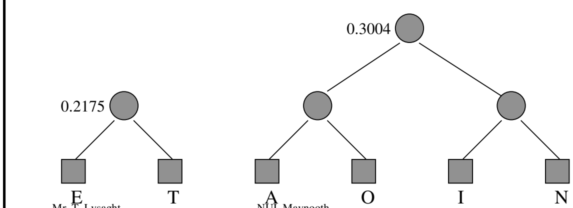
Huffman’s Algorithm



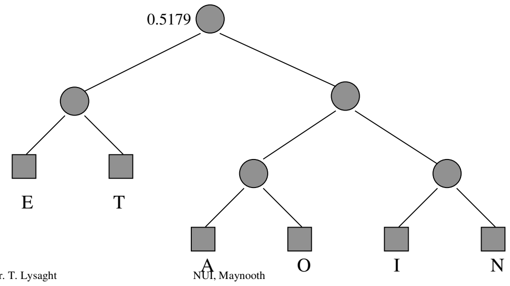
Huffman’s Algorithm



Huffman’s Algorithm



Huffman's Algorithm



Huffman's Algorithm

- The final prefix code is:
  - A 100
  - E 00
  - I 110
  - N 111
  - O 101
  - T 01

Huffman's Algorithm

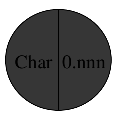
- Three phases in the algorithm
- Initialize the forest of code trees
- Construct an optimal code tree
- Compute the encoding map

Huffman's Algorithm

- Phase 1: Initialize the forest of code trees
  - How will we represent the forest of trees?
  - Better question: how will we represent our tree ... have to store both alphanumeric characters and probabilities?
  -
  -

Huffman's Algorithm

- Consequently, the initial tree is simply one internal node
- That is, it is a root (with two external nodes)

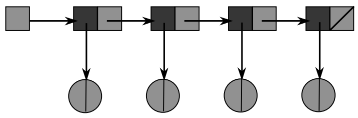


Huffman's Algorithm

- So, to create such a tree we simply invoke the following operations:
  -
  -

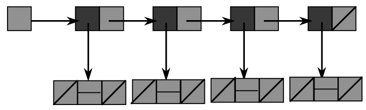
Huffman’s Algorithm

- We must also keep track of our forest
- Could represent it as a linked list of pointers to Binary trees ...



Huffman’s Algorithm

- Represented as:



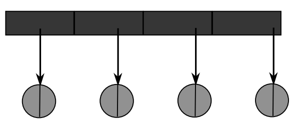
Huffman’s Algorithm

- Is there an alternative?
- Question: why do we use dynamic data structures?
- Answer:
  –
  –
- Is this the case here?
-

Huffman’s Algorithm

- So, our alternatives are? ......
- An array, indexed by number, of type ...
- *binary_tree*, i.e., each element in the array can point to a binary code tree

Huffman’s Algorithm



Huffman’s Algorithm

- What will be the dimension of this array?
-

Huffman's Algorithm

- Phase 2: construct the optimal code tree

Huffman's Algorithm

```
Pseudo-code algorithm
```


Huffman's Algorithm

let n be the number of trees initially
Repeat

Until only one tree left in the array

Huffman's Algorithm

- Phase 3: Compute the encoding map
  - We need to write out a list of source symbols together with their prefix code
  - We need to write out the contents of each external node (or each frontier internal node) together with the path to that node
  - We need to **traverse** the binary code tree in some manner

- But …. we want to print out the symbol and the prefix code:
  -
  -

- How will we represent the path?
-