LIÈGE université
School of Engineering

University of Liège

Faculty of applied science

Academic year 2024-2025

ELEN0060-2: Project 2
Source coding, data compression and channel coding

Pierre LORENZEN S203724

Abdelilah KHALIPHI S204896

Contents

1 Implementation

1.1 Question 1

In the implementation for this question, we first define a node class to facilitate the creation and management of the binary tree used in the Huffman coding algorithm.

Secondly, iteratively, we sort the nodes based on their probabilities at each iteration, and we merge the two lowest probabilities. This node created by the merge of the two lowest probabilities ones is then added to the tree with a probability equal to the sum of the two lowest probabilities. The children nodes are the two nodes that were merged. The process is repeated until we have a single node left, which is the root of the tree.

Tirdly, we generate the codes for each symbol by traversing the tree. We start at the root and assign a '0' code to the left child and a '1' code to the right child. We continue this process recursively until we reach a leaf node, at which point we store the generated code for that symbol.

Finally, we reorder the symbols to be consistent with the order provided at the input of the function.

To extend the Huffman code generation to any alphabet size q, we can modify the algorithm to merge the q lowest-probabilities nodes at each step and so build a q-ary tree, where each node has up to q children.

To implement this with a number of input symbols n and output alphabet size of $q \geq 2$, we can

1. Take the n symbols with their probabilities
2. Add Dummy Symbols: If $(n-1)\ mod\ (q-1) \neq 0$, add dummy symbols of probability 0 so that: $(n'-1)\ mod\ (q-1) = 0$

   where n' is the total number of symbols (real + dummy).
3. Maintain nodes sorted by probability.
4. Merge q smallest-nodes.
5. Repeat until only one node is left.
6. Label the edges with 0,1,...,q-1.

1.2 Question 2

1.3 Question 3

As explained in the theoretical course, the Lempel-Ziv algorithms can be compared on several aspects. These aspects are the dictionary construction, the parsing of the input data, the encoding, the dictionary address size, the adaptivity of the algorithm, the efficiency, and the on-line capability.
Base on this comparison, we can conclude that the advantages of the basic Lempel-Ziv algorithm are that the compression on large dictionary is more efficient than the on-line Lempel-Ziv algorithm, and that the dictionary address size is fixed. The backwards of this version is that it is not adaptive, and it is not on-line.

The advantages of the on-line Lempel-Ziv algorithm are that it is adaptive, on small and medium dictionary size the efficiency is better due to the shorter addresses, and it is on-line. The backwards of this version is that the compression on large dictionary is less efficient than the basic Lempel-Ziv algorithm, and that the dictionary address size is variable.

Aspect	Basic Lempel-Ziv	On-line Lempel-Ziv
Dictionary construction	Grows incrementally by adding new words	Same as basic
Parsing	Greedy: find the longest prefix in dictionary	Same as basic
Encoding	a tuple (address of prefix, next symbol)	Same as basic but the address is encoded dyna
Dictionary address size	Fixed-size = 2^n Where n is the number of bits	Variable-length based on current dictionary siz
Adaptivity	No adaptation done based on the dictionary	fewer bits used at early stages
Efficiency	Good compression when the dictionary is large	More efficient in early stages due to shorter ad
On-line capability	No	Yes it can transmit as the text is read

1.4 Question 4

2 Source coding and reversible (lossless) data compression

2.1 Question 5

2.2 Question 6

2.3 Question 7

2.4 Question 8

2.5 Question 9

2.6 Question 10

2.7 Question 11

2.8 Question 12

2.9 Question 13

2.10 Question 14

2.11 Question 15

2.12 Question 16

3 Channel coding

3.1 Question 17

3.2 Question 18

3.3 Question 19

3.4 Question 20

3.5 Question 21

3.6 Question 22