



# FILE COMPRESSION ALGORITHMS: IMPLEMENTATION AND COMPARISONS

TEAM MEMBERS:

SAGAR CHOUDHURY - 181IT140 - 7205067205

MD. ALTAF HUSSAIN - 181IT226 - 8507373593

SUMIT GUPTA - 181IT247 - 9133069128

MITHAS KUMAR - 181IT227 - 9870422853

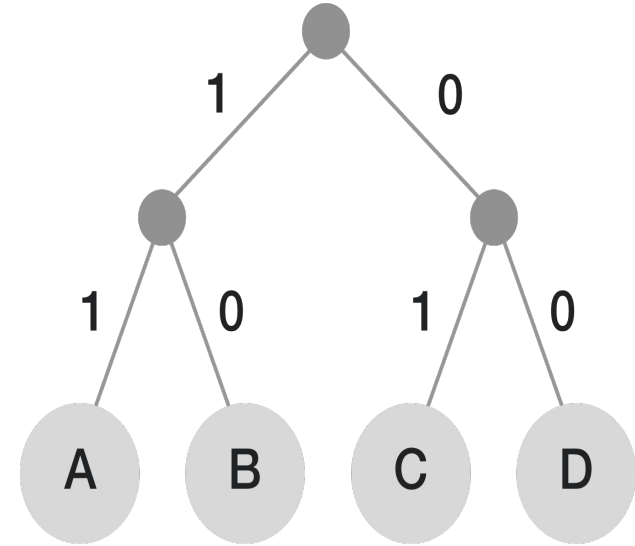


# INTRODUCTION

- Lossless compression is a class of [data compression](#) algorithms that allows the original data to be perfectly reconstructed from the compressed data.
- Lossless compression is used where it is important that the original and the decompressed data be identical, or where deviations from the original data would be unfavourable. Typical examples are executable programs, text documents, and source code.
- In this project, We have implemented 3 lossless compression algorithms which are as follows:
  - Huffman Encoding
  - Lempel-Ziv-Welch
  - Prediction by Partial Matching
- We have implemented both compression and decompression of files by all three above algorithms mentioned.

# HUFFMAN ENCODING

- In this algorithm, a variable-length code is assigned to input different characters.
- The code length is related to how frequently characters are used.
- Most frequent characters have the smallest codes and longer codes for least frequent characters.
- Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code.





# HUFFMAN CODING ALGORITHM

create a priority queue Q consisting of each unique character.

sort then in ascending order of their frequencies.

for all the unique characters:

- create a newNode
- extract minimum value from Q and assign it to leftChild of newNode
- extract minimum value from Q and assign it to rightChild of newNode
- calculate the sum of these two minimum values and assign it to the value of newNode
- insert this newNode into the tree

return rootNode

- For decoding the characters, we take the code and traverse through the tree to find the character.



# LEMPER-ZIV-WELCH

- LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes.
- LZW compression uses a code table.
- When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks.
- As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table.
- Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents.



# LZW ENCODING

- As the input data is being processed, a dictionary keeps a correspondence between the longest encountered words and a list of code values.
- The words are replaced by their corresponding codes and so the input file is compressed.
- Therefore, the efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.



# LZW ENCODING PSEUDOCODE

Initialize table with single character strings

P = first input character

WHILE not end of input stream

    C = next input character

    IF P + C is in the string table

        P = P + C

    ELSE

        output the code for P

    add P + C to the string table

    P = C

END WHILE

output code for P



# LZW DECODING

- The LZW decompressor creates the same string table during decompression. It starts with the first 256 table entries initialized to single characters.
- The string table is updated for each character in the input stream, except the first one.
- Decoding achieved by reading codes and translating them through the code table being built.





# LZW DECODING PSEUDOCODE

```
read a character k;  
output k  
w = k  
WHILE not end of input stream  
    k = next character  
    entry = dictionary entry for k  
    output entry  
    add w + entry[0] to dictionary  
    w = entry
```



# PREDICTION BY PARTIAL MATCHING

- The prediction by partial string matching algorithm is a compression method that combines predictions from different contexts using backing-off.
- It processes an input sequence symbol by symbol, accumulating counts of symbol occurrences for different contexts (up to some maximum depth  $D$ ), and using those counts to compress the input with an arithmetic coder.



# PREDICTION BY PARTIAL MATCHING

- The PPM algorithm makes use of a **trie** data structure, a search tree that maps partial strings (contexts) up to some maximum length  $D$  to a histogram of symbol occurrences.
- Each node stores one histogram, and typically also a pointer to the node of the next shorter context; these pointers are called vine pointers.
- The vine pointers help with two things: firstly, they allow the algorithm to retrieve the next shorter context quickly when computing the predictive symbol distribution; secondly, they speed up finding the context node for the next symbol in the sequence



# PREDICTION BY PARTIAL MATCHING ALGORITHM

1. Initialise the search trie with an empty root node.
2. Repeat until stop criterion is reached:
  - (a) **Fetch** - Retrieve the next input symbol  $e$ .
  - (b) **Encode** - Use the probability distribution defined by the histogram of the current trie node (and its parents) to encode  $e$ .
  - (c) **Learn** - Update the histogram of the current trie node, and also the histograms of the nodes along the vine pointer chain.



## PREDICTION BY PARTIAL MATCHING ALGORITHM

(d) **Advance** - Find (or create) the node in the trie corresponding to the next context, creating any missing nodes. Let  $p$  be a pointer to the current node.

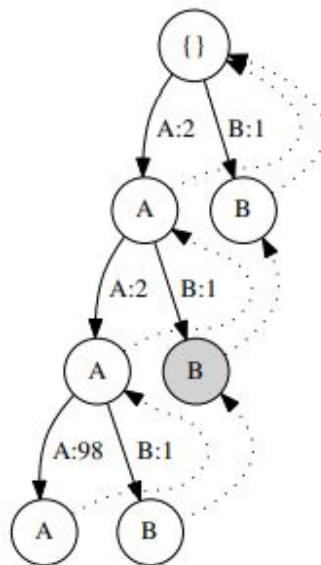
i. If the current node is at depth  $D$  in the trie, set  $p$  to the vine pointer.

ii. If the node pointed to by  $p$  does not have a child labelled  $e$ , create one.


(Make sure to set the new child's vine pointer correctly, possibly by creating additional nodes.)

iii. Point  $p$  to the child labelled  $e$ . The node identified by  $p$  is the new current node.

# Example:



# PREDICTION BY PARTIAL MATCHING PSEUDOCODE



```
begin
  while (not last character) do
    begin
      readSymbol()
      shorten context
      while (context not found and context length not -1) do
        begin
          output(escape sequence)
          shorten context
        end
      output(character)
      while (context length not -1) do
        begin
          increase count of character (create node if nonexistent)
          shorten context
        end
      end
    end
  end
end
```



## RESULTS AND ANALYSIS: Huffman Encoding

SIZE OF FILE BEFORE COMPRESSION	SIZE OF FILE AFTER COMPRESSION (HUFFMAN)	COMPRESSION RATIO
46.7 KB	29.3 KB	1.5939
104 KB	64.6 KB	1.6099
247 KB	153 KB	1.6144
990 KB	541 KB	1.8299
2070 KB	1107 KB	1.8818





## RESULTS AND ANALYSIS: LZW Encoding

SIZE OF FILE BEFORE COMPRESSION	SIZE OF FILE AFTER COMPRESSION (LZW)	COMPRESSION RATIO
46.7 KB	75.3 KB	0.6202
104 KB	155 KB	0.671
247 KB	353 KB	0.6997
990 KB	846 KB	1.1702
2070 KB	1670 KB	1.2395



## RESULTS AND ANALYSIS: PPM Encoding

SIZE OF FILE BEFORE COMPRESSION	SIZE OF FILE AFTER COMPRESSION (PPM)	COMPRESSION RATIO
46.7 KB	15.0 KB	3.1133
104 KB	29.7 KB	3.5017
247 KB	64.7 KB	3.8176
990 KB	89.6 KB	11.0491
2070 KB	178 KB	11.6292

# SIZE COMPRESSION COMPARISON





# ASYMPTOTIC RUNTIME ANALYSIS

- **For Huffman Encoding:**

Huffman encoding is based on a greedy approach to get optimal code length.

$O(n \log n)$  where  $n$  is the number of unique characters. If there are  $n$  nodes, `extractMin()` is called  $2 \cdot (n - 1)$  times. `extractMin()` takes  $O(\log n)$  time as it calls `minHeapify()`.

The overall Time Complexity:  **$O(n \log n)$** .



# ASYMPTOTIC RUNTIME ANALYSIS

- For Lempel-Ziv-Welch:

As the dictionary size is fixed and independent of the input length, LZW is in  $O(n)$  as each byte is only read once and the complexity of the operation for each character is constant.

The overall Time Complexity:  **$O(n)$** .



# ASYMPTOTIC RUNTIME ANALYSIS

- **For Prediction By Partial Matching:**

Running Time Function:  $T(n) = N(\log n + a) + S_n$

N-> Number of input symbols

n-> Current number of unique symbols

S-> Time to maintain the trie data structure

The overall Time Complexity:  **$O(N \log n)$** .



# CONCLUSION

- LZW requires no prior information about the input data stream.
- LZW can compress the input stream in one single pass.
- Huffman Coding could still compress the data significantly if certain symbols occur more frequently than others.
- In case of text it is clear intuitively and proved that probability of every next symbol is highly dependent on previous symbols. So, when we use Prediction by Partial Matching method to compress we get the best results out of the three algorithms.



**THANK YOU**