HUFFMAN CODING ALGORITHM

University of Science, VNU-HCM

Data Structures and Algorithms

Group 4

Theory Lecturer	Dr.	Nguyen Thanh Phuong	**Students**	24127104	Du Hoai Phuc - Leader
Lab Lecturer	Dr.	Nguyen Ngoc Thao		24127008	Truong Nhat Phat
	M.S.	Nguyen Thanh Tinh		24127173	Nguyen Tuan Hung

Overview of Data Compression

What is Data Compression?

The process of reducing data size without losing (or minimally losing) information.

Types of Compression

- **Lossless Compression:** Guarantees complete recovery of original data
- **Lossy Compression:** Accepts loss of some non-critical information

Historical Development

- Morse Code - assigning shorter codes to frequently used letters
- Claude Shannon (1948) - Information Theory
- David Huffman (1952) - Huffman Algorithm
- Lempel-Ziv (1970s-80s) - LZ77, LZ78, LZW

Need for Data Compression

- Save storage space
- Reduce data transmission time
- Optimize bandwidth
- Reduce storage and transmission costs

Applications

- Multimedia compression (JPEG, MP3, MPEG)
- Text compression (ZIP, GZIP, 7-Zip)
- Database compression
- Compression in embedded systems

Introduction to Huffman Algorithm

David Albert Huffman (1925-1999)

- Graduated from Ohio State University at age 18
- PhD at MIT in 1953
- Professor at University of California, Santa Cruz
- Motto: "My products are my students"

Birth of the Huffman Algorithm

In 1952, David Huffman was a student of Professor Robert M. Fano at MIT.

Final assignment: Find the most efficient way to encode a set of symbols based on frequency.

The algorithm was published in the paper "A Method for the Construction of Minimum-Redundancy Codes".

Symbolistics of Huffman Algorithm

- Lossless compression algorithm
- Uses prefix codes
- Based on symbol frequency
- Assigns shorter codes to frequent symbols
- Uses binary tree to represent codes

Prefix Code

A symbol's code is not a prefix of any other symbol's code

Valid: [00, 11, 10, 010]

Invalid: [00, 001, 10, 010]
Reason: 00 is a prefix of 001

Steps of the Huffman Algorithm

Step 1: Count symbol frequencies
Read the input data and count occurrences of each symbol

Step 2: Build a Min-Heap (priority queue)
Insert each symbol as a node into the Min-heap based on frequency
Highest priority given to symbols with lowest frequency

Step 3: Build the Huffman tree
- Take the two nodes with lowest frequencies and remove them from the heap
- Create a new node with frequency equal to the sum of the two nodes
- Make the two nodes the children of the new node
- Insert the new node into the heap
- Repeat until only one node remains in the heap (the root of the Huffman tree)

Step 4: Assign binary codes to symbols
- Traverse the Huffman tree from root to leaves
- When going through a left edge, add '0' to the code
- When going through a right edge, add '1' to the code
- When a leaf is reached, the obtained code is the Huffman code for that symbol

Step 5: Encode and decode data
Encoding: Replace each symbol in the original data with its corresponding Huffman code
Decoding: Use the Huffman tree to convert the encoded data back to the original data

Visualizing the Huffman Tree Construction Process

Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Min-Heap: H:1 u:1 m:1 a:1 n:1 f:2

Start with all symbols from "Huffman" in the min-heap, ordered by frequency.

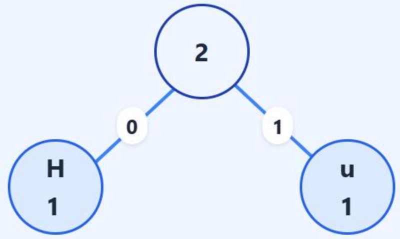
Previous Step **Step 1/6** Next Step

Visualizing the Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Huffman Tree Construction Process



Removed: ~~H:1~~ ~~u:1~~

Min-Heap: m:1 a:1 n:1 f:2 2

Take two nodes with lowest frequency: 'H' (1) and 'u' (1). Create a new node with frequency 2.

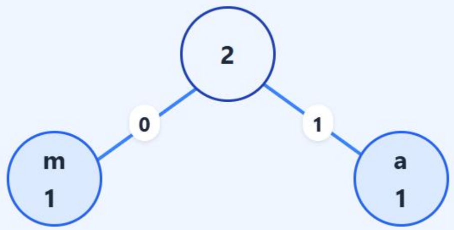
Previous Step **Step 2/6** Next Step

Visualizing the Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Huffman Tree Construction Process



Removed: ~~m:1~~ ~~a:1~~ **Min-Heap:** n:1 f:2 2 2

Take two nodes with lowest frequency: 'm' (1) and 'a' (1). Create a new node with frequency 2.

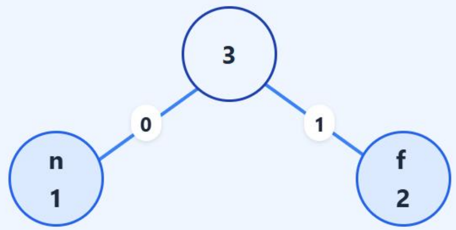
Previous Step **Step 3/6** Next Step

Visualizing the Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Huffman Tree Construction Process



Removed: ~~n:1~~ ~~f:2~~ **Min-Heap:** 2 2 3

Take two nodes with lowest frequency: 'n' (1) and 'f' (2). Create a new node with frequency 3.

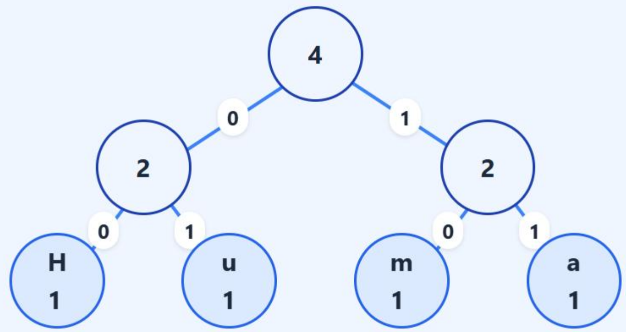
Previous Step **Step 4/6** Next Step

Visualizing the Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Huffman Tree Construction Process



Removed: ~~2~~ ~~2~~ **Min-Heap:** 3 4

Merge the first internal node (2) and second internal node (2) to create a new node with frequency 4.

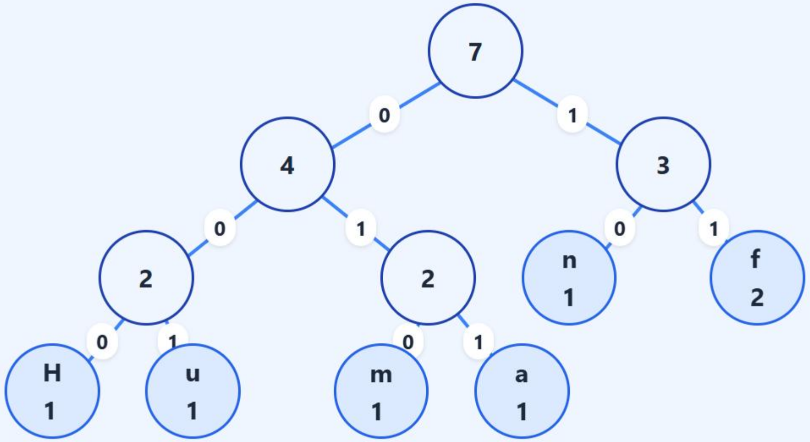
Previous Step **Step 5/6** Next Step

Visualizing the Huffman Tree Construction Process

Symbol Frequency in "Huffman"

Symbol	Frequency
H	1
u	1
f	2
m	1
a	1
n	1

Huffman Tree Construction Process



Removed:

~~3~~ ~~4~~

Min-Heap: Min-Heap is empty (Huffman tree is complete)

Complete the Huffman tree by connecting all remaining nodes based on their frequencies.

Previous Step **Step 6/6** Next Step

Huffman Codes and Encoding Process

Huffman Code Table

Symbol	Huffman Code	ASCII (for comparison)
H	000	01001000 (8 bits)
u	001	01110101 (8 bits)
f	11	01100110 (8 bits)
m	010	01101101 (8 bits)
a	011	01100001 (8 bits)
n	10	01101110 (8 bits)

Encoding Example

Original string: `Huffman`

Encoded string: `000 001 11 11 010 011 10`

Encoding details:
H (000) + u (001) + f (11) + f (11) + m (010) + a (011) + n (10)
= 000 + 001 + 11 + 11 + 010 + 011 + 10
= 000 001 11 11 010 011 10

Decoding Process

Encoded string: `000001111101001110`

Decoding steps:

1. Start from the root of the Huffman tree
2. Read each bit of the encoded string
3. Move in the tree: 0 → left, 1 → right
4. When a leaf node is reached, record the symbol and return to the root
5. Repeat until the encoded string is exhausted

Efficiency Analysis

ASCII Encoding: 7 symbols × 8 bits = 56 bits

Huffman Encoding: 18 bits

Compression Ratio: (1 − 18/56) × 100% ≈ 67.86% savings.

Adaptive Huffman Coding

Problems with Static Huffman

- Requires two data scans: one to calculate frequencies, one to encode
- Needs to store the Huffman code table with the compressed data
- Not efficient with data that changes frequency over time

FGK Algorithm (Faller-Gallager-Knuth)

- Updates the Huffman tree after processing each symbol
- Nodes are numbered in order from right to left and bottom to top
- Processes data in a streaming fashion (one-pass)

Vitter Algorithm (Algorithm V)

- Similar to FGK but maintains sibling property in the tree
- Helps optimize the tree update process
- More efficient than FGK in terms of processing time

Advantages of Adaptive Huffman

- Only needs to scan data once
- No need to store code table with compressed data
- Can compress data in real-time streams
- Adapts to changing frequency distributions

Other Variants of Huffman Algorithm

n-ary Huffman

- Uses n-ary trees instead of binary trees
- Allows creating codes in base n instead of binary
- Useful in applications requiring codes in bases larger than 2

Length-Limited Huffman

- Limits the maximum code length
- Sacrifices optimality to ensure length constraints
- Useful for applications with bit length limitations

Canonical Huffman

- Rearranges codes for easier storage and decoding
- Maintains optimality of the original codes
- Useful in applications requiring fast decoding

Dynamic Huffman

- Updates the Huffman tree periodically
- Balances between compression efficiency and update cost
- Useful for data with slowly changing frequency distributions

Choosing the Right Variant

The choice of Huffman variant depends on:

- Symbolistics of the data to be compressed
- Encoding/decoding speed requirements
- Memory and computational constraints
- Application nature (real-time/offline)

Performance Analysis of Huffman Algorithm

Algorithm Complexity

Time Complexity:

- Frequency counting: O(n)
- Huffman tree construction: O(k log k)
- Data encoding: O(n)
- Total: O(n + k log k)

(n is data length, k is number of distinct symbols)

Space Complexity:

- O(k) for the Huffman tree
- O(n) for the encoded data

Experimental Results

For ASCII text files:

- File size reduction around 20%
- With data having many repeated symbols: compression ratio up to 30%
- Efficiency decreases with uniformly distributed data (high entropy)

Advantages

- Theoretically optimal for encoding individual symbols
- No data loss during decompression
- Simple and efficient
- Widely used in many popular compression algorithms

Disadvantages

- Needs to know frequencies before encoding (static version)
- Requires storing the tree/code table with the compressed data
- Not efficient with high-entropy data
- Doesn't exploit correlation between symbols

Applications of Huffman Algorithm

File Compression

- 7-Zip: Uses DEFLATE compression (combining LZ77 and Huffman)
- WinZip and WinRAR: Commercial archiving tools
- macOS Archive Utility: Built-in archiving utility in macOS
- Command-line tools: gzip, zip on Linux/macOS

Multimedia Compression

- JPEG: Uses Huffman to compress pixel data after DCT transformation
- MP3: Compresses quantized frequency data
- MPEG-4: Compresses motion vector data in video
- PNG: Image format using DEFLATE compression algorithm

Data Transmission and Networks

- HTTP Content-Encoding (GZIP): Compressing CSS, HTML, JS files
- Fax machines and telephones: Using Group 3 fax encoding
- Wireless data transmission: Saving bandwidth

Artificial Intelligence and Machine Learning

- Decision tree compression: Optimizing large decision trees
- Natural Language Processing (NLP): Dictionary-based text compression
- Language modeling: Reducing model size

Hardware and Embedded Systems

- GPUs and FPGAs: Real-time compression in high-performance computing
- Data storage systems: Reducing file size in SSDs and HDDs
- Embedded systems: Memory optimization in IoT devices
- Control boards: Reducing firmware size

Conclusion

Summary of Huffman Algorithm

- Lossless compression algorithm based on prefix codes
- Assigns shorter codes to frequent symbols, longer codes to rare symbols
- Complexity $O(n + k \log k)$ where n is data size, k is number of distinct symbols
- Compression ratio around 20-30% for normal text
- Many variants including Adaptive Huffman, Canonical Huffman, Length-Limited Huffman
- Widely applied in file compression, multimedia, networks, and embedded systems

Role in Computer Science

Despite being over 70 years old, the Huffman algorithm remains a foundation for many modern compression algorithms and is an important basic knowledge in computer science.

Huffman is a classic example of how a simple and efficient algorithm can create long-lasting and profound impacts in technology.

Thank you for your watching!

Group 4

Theory Lecturer	Dr.	Nguyen Thanh Phuong	**Students**	24127104	Du Hoai Phuc - Leader
Lab Lecturer	Dr.	Le Ngoc Thao		24127008	Truong Nhat Phat
	M.S.	Nguyen Thanh Tinh		24127173	Nguyen Tuan Hung