ECE F344

Information Theory and Coding (ITC)

Sem-2

2024-25

Assignment 1

C++ code for Huffman coding algorithm

Dated: 16 Feb 2025

Submitted by

Lakshya Jain

2022AAPS0247P

Introduction

In the code prepared by me for this assignment, the code first asks the user how many symbols he/she wants to use (giving provision for any number n ranging from 1 to 10 - 10 was chosen arbitrarily in order to not overly complicate the time complexity of the code).

After that the user can enter the probability for every symbol one by one. In each entry, the remaining probability (1 minus sum of probabilities entered till that point) is shown. After the user enters the probabilities for all the symbols, the code checks whether the cumulative probability adds up to 1, if not, then the user is prompted to restart the code.

Then the user is asked to enter the base for the Huffman code. After this, the code generates the Huffman codes for all the symbols and displays it, also calculating the entropy, average code length and code efficiency using the appropriate formulas.

After this, the user is prompted to enter the message to be encoded, following with the encoded sequence is generated. After this, the code asks for an encoded sequence to decode. If the same encoded sequence from the previous step (or any other random sequence) is entered, the decoded symbols are generated by the code, using the Huffman codes previously generated.

Code in C++

```
#include <iostream>
#include <queue>
#include <map>
#include <cmath>
#include <vector>
#include <algorithm>

using namespace std;

struct Node {
    char symbol;
    double probability;
    vector<Node*> children;
    string code;
};

struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->probability > b->probability;
    }
};

void generateCodes(Node* root, string code) {
    if (!root) return;
    if (root->symbol != '\0') {
        root->code = code;
        return;
    }
    for (size_t i = 0; i < root->children.size(); i++) {
        generateCodes(root->children[i], code + to_string(i));
    }
}

Node* buildHuffmanTree(vector<pair<char, double>> &symbols, int base) {
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (auto &s : symbols) {
        Node* newNode = new Node{s.first, s.second, {}};
```

```
            pq.push(newNode);
        }

        while (pq.size() > 1) {
            vector<Node*> group;
            for (int i = 0; i < base && !pq.empty(); i++) {
                group.push_back(pq.top());
                pq.pop();
            }

            Node* parent = new Node{'\0', 0, group};
            for (auto child : group) {
                parent->probability += child->probability;
            }
            pq.push(parent);
        }
        return pq.top();
}

void collectCodes(Node* root, map<char, string>& codes) {
    if (!root) return;
    if (root->symbol != '\0') {
        codes[root->symbol] = root->code;
    }
    for (auto child : root->children) {
        collectCodes(child, codes);
    }
}

double calculateEntropy(vector<pair<char, double>>& symbols, int base) {
    double entropy = 0;
    for (auto &s : symbols) {
        entropy -= s.second * (log(s.second) / log(base));
    }
    return entropy;
}

double averageCodeLength(map<char, string>& codes, vector<pair<char,
double>>& symbols) {
    double avgLength = 0;
```

```
    for (auto &s : symbols) {
        avgLength += s.second * codes[s.first].length();
    }
    return avgLength;
}

string decode(Node* root, string encoded, map<char, string>& codes) {
    string decoded = "";
    Node* current = root;
    for (char c : encoded) {
        if (!current->children.empty()) {
            current = current->children[c - '0'];
        }
        if (current->symbol != '\0') {
            decoded += current->symbol;
            current = root;
        }
    }

    cout << "Decoded Huffman Code: ";
    for (char c : decoded) {
        cout << codes[c] << " ";
    }
    cout << endl;

    return decoded;
}

string encode(string message, map<char, string>& codes) {
    string encoded = "";
    for (char c : message) {
        encoded += codes[c];
    }
    return encoded;
}

int main() {
    int numSymbols;
    cout << "Enter number of symbols (max 10): ";
    cin >> numSymbols;
```

```
    if (numSymbols < 1 || numSymbols > 10) {
        cout << "Invalid number! Defaulting to 7 symbols." << endl;
        numSymbols = 7;
    }

    vector<pair<char, double>> symbols;
    double cumulativeProbability = 0.0;

    for (int i = 0; i < numSymbols; i++) {
        char symbol = 'A' + i;
        double probability;
        cout << "Enter probability for " << symbol << ": ";
        cin >> probability;
        cumulativeProbability += probability;
        if (cumulativeProbability > 1.0) {
            cout << "Total probability exceeds 1. Please re-enter
probability for " << symbol << "!" << endl;
            cumulativeProbability -= probability;
            i--;
            continue;
        }
        symbols.push_back({symbol, probability});
        cout << "Probability left to be assigned for remaining variables
is: " << 1 - cumulativeProbability << endl;
    }

    if (cumulativeProbability != 1.0) {
        cout << "Error: Sum of all probabilities must be exactly 1. Please
restart and enter valid probabilities." << endl;
        return 1;
    }

    sort(symbols.begin(), symbols.end(), [](const pair<char, double>& a,
const pair<char, double>& b) {
        return a.second > b.second;
    });

    int base;
    cout << "Enter base for Huffman coding: ";
    cin >> base;
```

```
    if (base < 2) {
        cout << "Invalid base! Defaulting to base-2." << endl;
        base = 2;
    }

    Node* root = buildHuffmanTree(symbols, base);
    generateCodes(root, "");

    map<char, string> codes;
    collectCodes(root, codes);

    cout << "\nGenerated Huffman Codes:" << endl;
    for (auto &entry : codes) {
        cout << entry.first << " : " << entry.second << endl;
    }

    double entropy = calculateEntropy(symbols, base);
    double avgLength = averageCodeLength(codes, symbols);
    double efficiency = (entropy / avgLength) * 100;

    cout << "\nEntropy: " << entropy << " bits/symbol" << endl;
    cout << "Average Code Length: " << avgLength << " bits" << endl;
    cout << "Coding Efficiency: " << (efficiency > 100 ? 100 : efficiency)
<< " %" << endl;

    string message;
    cout << "\nEnter a message to encode: ";
    cin >> message;
    string encodedMessage = encode(message, codes);
    cout << "Encoded Sequence: " << encodedMessage << endl;

    string encoded;
    cout << "\nEnter encoded sequence to decode: ";
    cin >> encoded;

    string decoded = decode(root, encoded, codes);
    cout << "Decoded Data: " << decoded << endl;

    return 0;}
```

Output

```
Enter number of symbols (max 10): 7
Enter probability for A: 0.3
Probability left to be assigned for remaining variables is: 0.7
Enter probability for B: 0.08
Probability left to be assigned for remaining variables is: 0.62
Enter probability for C: 0.05
Probability left to be assigned for remaining variables is: 0.57
Enter probability for D: 0.2
Probability left to be assigned for remaining variables is: 0.37
Enter probability for E: 0.1
Probability left to be assigned for remaining variables is: 0.27
Enter probability for F: 0.15
Probability left to be assigned for remaining variables is: 0.12
Enter probability for G: 0.12
Probability left to be assigned for remaining variables is: 0
```

```
Enter base for Huffman coding: 2

Generated Huffman Codes:
A : 11
B : 1001
C : 1000
D : 00
E : 010
F : 101
G : 011

Entropy: 2.60289 bits/symbol
Average Code Length: 2.63 bits
Coding Efficiency: 98.969 %

Enter a message to encode: ABCFE
Encoded Sequence: 1110011000101010

Enter encoded sequence to decode: 1110011000101010
Decoded Huffman Code: 11 1001 1000 101 010
Decoded Data: ABCFE
```