Sprawozdanie 09: Programowanie zachłanne na przykładzie algorytmu Huffmana

Dmytro Sakharskyi - L02 - 31259 11.06.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

Wstęp

Algorytm Huffmana to klasyczny przykład algorytmu zachłannego, który pozwala na efektywną kompresję danych poprzez przypisanie krótszych kodów tym znakom, które występują częściej, oraz dłuższych kodów znakom rzadszym. W odróżnieniu od standardowego kodowania **ASCII**, w którym każdemu znakowi przypisuje się kod o tej samej długości (zwykle **8 bitów**), **kodowanie Huffmana** umożliwia zmniejszenie łącznej długości zakodowanego tekstu.

Celem niniejszych zajęć było poznanie zasady działania algorytmu **Huffmana** oraz praktyczna implementacja programu realizującego:

- budowę drzewa kodowego **Huffmana** na podstawie analizy częstotliwości znaków w tekście wejściowym,
- kodowanie oraz dekodowanie wiadomości,
- analizę skuteczności kompresji.

Dodatkowo, zadanie obejmowało rozbudowę dostarczonego kodu o nowe funkcjonalności: możliwość wczytywania tekstu z pliku, podział funkcji **buildHuffmanTree()** na logiczne kroki oraz sortowanie wyników końcowych według częstotliwości znaków.

Zadanie 1

Do przeprowadzenia analizy kodowania **Huffmana** wybrano tekst zawierający co 600+ słów. Tekst został zapisany w pliku **input.txt**, który następnie został wczytany przez program.

Wybrany tekst posłużył jako dane wejściowe do wszystkich kolejnych etapów: **analizy częstotliwości znaków**, **budowy drzewa Huffmana**, **generowania kodów binarnych** oraz **obliczania skuteczności kompresji**.

Źródło tekstu: *На Западном фронте без перемен - Ремарк Эрих Мария | читать онлайн, скачать*

Tłumacz: *Google Translate*

Przykład pracy:

```
Huffman Codes:
Char        Freq        Code
            686         111
e           348         000
t           257         1011
a           250         1010
o           224         1000
r           205         0110
s           198         0101
n           192         0100
i           166         0010
h           153         11010
l           130         11001
d           119         10010
u           92          00111
f           87          110111
c           65          110001
g           65          110000
w           62          100111
y           56          011111
,           54          011101
m           54          011110
p           44          001100
b           38          1101101
.           27          1001100
k           26          0111000
v           22          0011010
;           12          00110110
```

Zadanie 2

```cpp
unordered_map<char, int> buildFrequencyMap(const string& text) {
    unordered_map<char, int> freq;
    for (char ch : text) freq[ch]++;
    return freq;
}
```

buildFrequencyMap funkcja przelicza, ile razy każdy znak pojawia się w tekście. Dane te są później używane do budowy drzewa Huffmana.

```cpp
Node* buildHuffmanTree(const unordered_map<char, int>& freq) {
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (auto pair : freq) {
        pq.push(new Node(pair.first, pair.second));
    }

    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;
        pq.push(merged);
    }
    return pq.top();
}
```

Użycie **kolejki priorytetowej** do łączenia dwóch najmniej częstych znaków. To klasyczna realizacja **fazy redukcji** w algorytmie Huffmana.

```cpp
void generateCodes(Node* root, const string& code, unordered_map<char, string>& huffmanCode) {
    if (!root) return;
    if (!root->left && !root->right) {
        huffmanCode[root->ch] = code;
    }
    generateCodes(root->left, code + "0", huffmanCode);
    generateCodes(root->right, code + "1", huffmanCode);
}
```

Funkcja **generateCodes** rekurencyjna, która przechodzi przez drzewo od korzenia do liści i przypisuje każdemu znakowi odpowiedni kod binarny. Lewa gałąź = 0, prawa = 1.

Kod przedstawia kompletną implementację algorytmu Huffmana, który umożliwia:

- analizę częstotliwości występowania znaków w tekście,
- budowę drzewa kodowego,
- wygenerowanie binarnych kodów Huffmana dla każdego znaku,
- obliczenie długości zakodowanej wiadomości oraz współczynnika kompresji.

Zadanie 3

Funkcja **buildHuffmanTree()** została rozbita na mniejsze, logiczne funkcje, aby lepiej odzwierciedlić kroki algorytmu Huffmana. Taki podział poprawia czytelność, testowalność i zgodność z zasadami programowania modularnego.

```
priority_queue<Node*, vector<Node*>, Compare> buildPriorityQueue(const unordered_map<char, int>& freq) {
    priority_queue<Node*, vector<Node*>, Compare> pq;
    for (const auto& pair : freq) {
        pq.push(new Node(pair.first, pair.second));
    }
    return pq;
}
```

Odpowiada za krok 1 i 2 z fazy redukcji – przekształca dane wejściowe w strukturę do dalszego przetwarzania.

```
Node* buildTreeFromQueue(priority_queue<Node*, vector<Node*>, Compare>& pq) {
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;

        pq.push(merged);
    }
    return pq.top();
}
```

Odpowiada za krok 2 i 3 algorytmu Huffmana – tworzy strukturalnie poprawne drzewo kodowe poprzez łączenie dwóch najmniej prawdopodobnych elementów.

```
Node* buildHuffmanTree(const unordered_map<char, int>& freq) {
    auto pq = buildPriorityQueue(freq);
    return buildTreeFromQueue(pq);
}
```

Teraz funkcja buildHuffmanTree() pełni tylko rolę koordynatora i staje się **czytelna i łatwa do testowania**.

Zadanie 4

Zgodnie z poleceniem, do programu dodano możliwość wczytywania tekstu wejściowego z pliku o rozszerzeniu **.txt**. Dzięki temu użytkownik nie musi wpisywać danych ręcznie — program automatycznie analizuje zawartość pliku i przetwarza go przy użyciu algorytmu Huffmana.

```
// Step 1: Read input text from .txt
ifstream file("input.txt");
if (!file) {
    cerr << "Error: input.txt not found!" << endl;
    return 1;
}

string text((istreambuf_iterator<char>(file)), istreambuf_iterator<char>());
file.close();
```

- **ifstream** otwiera plik input.txt.
- Zawartość pliku jest wczytywana w całości do zmiennej text.
- Jeśli plik nie zostanie odnaleziony, program zwraca komunikat o błędzie i kończy działanie.
- Dalej tekst ten wykorzystywany jest do budowy mapy częstotliwości i tworzenia drzewa Huffmana.

Zadanie 5

a. Liczba bitów w kodowaniu ASCII

W kodowaniu ASCII każdy znak reprezentowany jest przez 8 bitów. Dlatego łączna liczba bitów potrzebna do zakodowania całego tekstu wejściowego obliczana jest według wzoru:

```
int originalBits = static_cast<int>(text.size()) * 8;
```

Liczba bitów = liczba znaków × 8

b. Liczba bitów w kodowaniu Huffmana
Dla każdego znaku w tekście sprawdzana jest długość jego kodu Huffmana i sumowana zgodnie z częstością jego występowania:

```
int huffmanBits = calculateEncodedSize(text, huffmanCode);
```

Funkcja calculateEncodedSize() oblicza łączną liczbę bitów, jakie będą potrzebne do zapisania całego tekstu w kodzie Huffmana.

c. Współczynnik kompresji
Stosunek liczby bitów w kodowaniu Huffmana do liczby bitów w ASCII:

```
double compression = (double)huffmanBits / originalBits;
```

liczba z zakresu 0–1, gdzie niższa wartość oznacza lepszą kompresję.

Zadanie 6

W celu zwiększenia przejrzystości i użyteczności wyświetlanych wyników, program został rozszerzony o możliwość prezentowania dodatkowej informacji — liczby wystąpień danego znaku w analizowanym tekście.

```
void displayHuffmanCodes(const unordered_map<char, string>& huffmanCode,
    const unordered_map<char, int>& freq) {
    vector<pair<char, int>> sorted;
    for (auto& p : freq) sorted.push_back(p);
    sort(sorted.begin(), sorted.end(), [](const auto& a, const auto& b) {
        return a.second > b.second;
        });

    cout << left << setw(10) << "Char" << setw(10) << "Freq" << "Code" << endl;
    for (auto& p : sorted) {
        cout << left << setw(10) << p.first << setw(10) << p.second << huffmanCode.at(p.first) << endl;
    }
}
```

Dzięki temu użytkownik otrzymuje pełny obraz:

- jaki znak wystąpił,
- ile razy się pojawił,
- jaki ma przypisany kod binarny Huffmana.

Przykład:

```
Huffman Codes:
Char        Freq        Code
            686         111
e           348         000
t           257         1011
a           250         1010
o           224         1000
r           205         0110
```

Zadanie 7

W ramach tego zadania ulepszono sposób prezentacji kodów Huffmana poprzez ich **posortowanie od najczęściej do najrzadziej występującego znaku**. Dzięki temu użytkownik może łatwo ocenić, które znaki dominują w tekście i jakie kody zostały im przypisane.

Do sortowania wykorzystano standardową funkcję **sort()** z biblioteką **<algorithm>**, która porównuje częstotliwości znaków:

```
sort(sorted.begin(), sorted.end(), [](const auto& a, const auto& b) {
    return a.second > b.second;
    });
```

To sprawia, że znaki w wynikowym zestawieniu są posortowane malejąco według liczby wystąpień.

Wnioski

W trakcie realizacji ćwiczenia zapoznano się z zasadą działania algorytmu **Huffmana**, który umożliwia **bezstratną kompresję danych tekstowych**. Zastosowany algorytm należy do klasy algorytmów **zachłannych** i wykorzystuje częstotliwość występowania znaków do przypisania im kodów binarnych o zmiennej długości.

W ramach zadania:

- wybrano i wczytano tekst wejściowy z pliku .txt,
- zbudowano mapę częstotliwości wystąpień znaków,
- utworzono drzewo Huffmana,
- wygenerowano kody Huffmana dla każdego znaku,
- obliczono liczbę bitów potrzebnych do zakodowania tekstu w formacie ASCII oraz przy użyciu algorytmu Huffmana,
- wyznaczono współczynnik kompresji,
- wyświetlono listę kodów wraz z częstotliwościami i posortowano ją malejąco według liczby wystąpień.

Zastosowanie algorytmu **Huffmana** pozwoliło uzyskać realną **redukcję** rozmiaru danych, co **potwierdziło skuteczność** tej metody w **kompresji tekstu**.