Федеральное государственное автономное образовательное учреждение высшего образования
«Российский университет дружбы народов имени Патриса Лумумбы»
Инженерная академия
Кафедра механики и процессов управления

Курсовая работа

По информатике и программированию

Направление: Прикладная математика и информатика

Профиль: Математические методы механики космического полёта и анализ геоинформационных данных

Тема: Алгоритмы сжатия данных: реализация кодирования Хаффмана на C++

Выполнено студентом: Гаврилина Антонина Владимировна
Группа: ИПМбд-01-23
№ студенческого: 1132233511

Москва, 2025

Оглавление

Введение

Цель работы:

Разработать программную реализацию алгоритма Хаффмана для эффективного сжатия данных, изучить его теоретические основы, провести анализ эффективности.

Задачи:

1. Теоретическое исследование
2. Реализация основных этапов алгоритма:
    a. Подсчёт частот символов во входных данных.
    b. Построение дерева Хаффмана.
    c. Генерация кодовых таблиц.
    d. Кодирование/декодирование данных.
3. Тестирование программы

Теоретическая справка

Алгоритм Хаффмана — жадный алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Классический алгоритм Хаффмана

Идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством **префиксности** (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частотностей символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево)

1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение.
2. Выбираются два свободных узла дерева с наименьшими весами.
3. Создается их родитель с весом, равным их суммарному весу.
4. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков.
6. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Допустим, у нас есть следующая таблица абсолютных частотностей:

Символ	А	Б	В	Г	Д
Абсолютная частотность	15	7	6	6	5

Этот процесс можно представить как построение дерева, корень которого — символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n_0 потомков — символы из предыдущего шага и т. д.

Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до

его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке.

Построение дерева для данного примера

Символ	А	Б	В	Г	Д
Код	0	100	101	110	111

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д — наибольшим.

При этом общая длина сообщения, состоящего из приведённых в таблице символов, составит 87 бит (в среднем 2,2308 бита на символ). При использовании равномерного кодирования общая длина сообщения составила бы 117 бит (ровно 3 бита на символ).

Классический алгоритм Хаффмана имеет ряд существенных недостатков. Для восстановления содержимого сжатого сообщения декодер должен знать таблицу частотностей, которой пользовался кодер. Следовательно, длина сжатого сообщения увеличивается на длину таблицы частотностей, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию сообщения.

Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по сообщению: одного для построения модели сообщения (таблицы частотностей и Н-дерева), другого - для, собственно, кодирования.

Реализация

Включает:

1. Построение дерева Хаффмана.
2. Генерацию кодов.
3. Кодирование строки.
4. Декодирование битовой строки.
5. Пример сжатия и распаковки.

Метод основывается на создании бинарных деревьев. В нем узел может быть либо конечным, либо внутренним. Изначально все узлы считаются листьями (конечными), которые представляют сам символ и его вес (то есть частоту появления). Внутренние узлы содержат вес символа и ссылаются на два узла-наследника. По общему соглашению, бит *«0»* представляет следование по левой ветви, а *«1»* — по правой. В полном дереве *N* листьев и *N-1* внутренних узлов. Рекомендуется, чтобы при построении дерева Хаффмана отбрасывались неиспользуемые символы для получения кодов оптимальной длины.

Мы будем использовать очередь с приоритетами для построения дерева Хаффмана, где узлу с наименьшей частотой будет присвоен высший приоритет.

Основное решение и комментарии

1. Подключаем необходимые заголовки

```
#include <iostream>
#include <queue>
#include <unordered_map> // Для хранения частот и кодов
#include <vector> // Для priority_queue
#include <memory> //для умных указателей
```

2. Создаем структуру для хранения информации о каждом узле (листе) на будущем дереве Хаффмана.

```
// Узел дерева Хаффмана
struct HuffmanNode {
    char ch; // символ
    int freq; //частота
    std::shared_ptr<HuffmanNode> left, right; //умный указатель на левого и
правого потомка

    HuffmanNode(char ch, int freq) : ch(ch), freq(freq), left(nullptr),
right(nullptr) {} //хранит символ (для листовых узлов)
    HuffmanNode(int freq) : ch('\0'), freq(freq), left(nullptr), right(nullptr)
{} // для внутренних узлов
};
```

3. Создаем компаратор, чтобы в очереди доставать сначала узлы с наименьшей частотой (по умолчанию с наибольшей)

Что делает:

- Сравнивает два узла по частоте
- Возвращает true, если a должен идти после b
- **Пример:** Если a->freq=3, b->freq=5, вернёт false - порядок правильный.

```
// Компаратор для очереди с приоритетом (для того чтобы минимальный эл
доставался первым)
struct Compare {
    bool operator()(const std::shared_ptr<HuffmanNode>& a, const
std::shared_ptr<HuffmanNode>& b) {
        return a->freq > b->freq;
    }
};
```

4. Функция для генерации кодов символов (на выходе – словарь формате символ – код)

```
// Генерация кодов символов (рекурсивный обход дерева)
void generateCodes(const std::shared_ptr<HuffmanNode>& root, const std::string&
code, std::unordered_map<char, std::string>& codes) {
    if (!root) return; //если узла не сущетсвует - выходим
    if (!root->left && !root->right) {
        codes[root->ch] = code; //если у узла нет потомков это лист, хранящий
символ
    }
    generateCodes(root->left, code + "0", codes); //влево - добавляем ноль
```

```
    generateCodes(root->right, code + "1", codes); //вправо - один
}
```

5. Функция для кодирования строки

```
// Кодирование строки
std::string encode(const std::string& data, const std::unordered_map<char,
std::string>& codes) {
    std::string encoded;
    for (char ch : data) {
        encoded += codes.at(ch);
    }
    return encoded;
}
```

6. Функция декодирования строки

```
// Декодирование строки
std::string decode(const std::string& encoded, const
std::shared_ptr<HuffmanNode>& root) {
    std::string decoded;
    auto current = root; //начинаем с корня дерева
    for (char bit : encoded) {
        if (bit == '0') {
            current = current->left; //если равен 0 - идём влево
        } else {
            current = current->right;
        }
        if (!current->left && !current->right) {
            decoded += current->ch;
            current = root;
        }
    }
    return decoded;
}
```

Основное тело программы:

1. Вводим строку и считаем частоту для каждого символа в строке

```
    std::string data = "abacabad";

    // 1. Подсчет частот символов
    std::unordered_map<char, int> freq; // Пустой словарь для частот
```

```
    for (char ch : data) {
        freq[ch]++;
    }
```

2. Построение дерева Хаффмана.

- Создаем узел-лист для каждого символа и добавляем их в очередь с приоритетами.

  Тип элементов в очереди - "умные указатели" на узлы дерева Хаффмана. Они автоматически удаляются, когда становятся ненужными.

- Пока в очереди больше одного листа делаем следующее:

Удаляем два узла с наивысшим приоритетом (с самой низкой частотой) из очереди;

Создаем новый внутренний узел, где эти два узла будут наследниками, а частота появления будет равна сумме частот этих двух узлов.

Добавляем новый узел в очередь приоритетов.

- Единственный оставшийся узел будет корневым, на этом построение дерева закончится.

Важные моменты об указателях

1. **shared_ptr**: Автоматически удаляет объект, когда последний shared_ptr на него уничтожается, можно безопасно копировать
2. **Оператор ->**: ptr->field эквивалентно (*ptr).field
3. **make_shared**: безопасно создаёт объект и возвращает shared_ptr

```
 // 2. Построение дерева Хаффмана
    std::priority_queue<std::shared_ptr<HuffmanNode>,
std::vector<std::shared_ptr<HuffmanNode>>, Compare> heap;
    for (const auto& pair : freq) {
        heap.push(std::make_shared<HuffmanNode>(pair.first, pair.second));
//создание узлов
```

```
    }

    while (heap.size() > 1) {  //строим дерево, до тех пор пока не остался 1 элемент - корень дерева
        auto left = heap.top(); //берем самый верхний эл и удаляем его (т.е с самой маленькой частотой)
        heap.pop();
        auto right = heap.top(); //берем следующий
        heap.pop();
        auto merged = std::make_shared<HuffmanNode>(left->freq + right->freq); //создаем объединённый узел
        merged->left = left; //делаем их дочерними узлами
        merged->right = right;
        heap.push(merged); //возмращаем узел в кучу
    }

    auto root = heap.top(); //корень дерева
```

4. Использование описанных выше функций и проверка результата

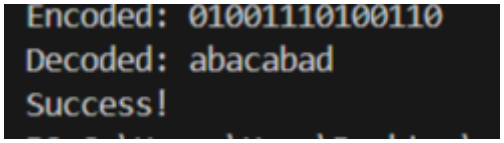
```
    // 3. Генерация кодов
    std::unordered_map<char, std::string> codes; //пустой словарь для кодов элементов
    generateCodes(root, "", codes);

    // 4. Кодирование строки
    std::string encoded = encode(data, codes);
    std::cout << "Encoded: " << encoded << std::endl;

    // 5. Декодирование
    std::string decoded = decode(encoded, root);
    std::cout << "Decoded: " << decoded << std::endl;

    // Проверка
    std::cout << (data == decoded ? "Success!" : "Error!") << std::endl;
```

Вывод программы



Весь код:

```
#include <iostream>
#include <queue>
#include <unordered_map> // Для хранения частот и кодов
#include <vector> // Для внутреннего использования priority_queue
#include <memory>

// Узел дерева Хаффмана
struct HuffmanNode {
    char ch; // символ
    int freq; //частота
    std::shared_ptr<HuffmanNode> left, right; //умный указатель на левого и
правого потомка

    HuffmanNode(char ch, int freq) : ch(ch), freq(freq), left(nullptr),
right(nullptr) {} //хранит символ (для листовых узлов)
    HuffmanNode(int freq) : ch('\0'), freq(freq), left(nullptr), right(nullptr)
{} // для внутренних узлов
};

// Компаратор для очереди с приоритетом (для того чтобы минимальный эл доставался
первым)
struct Compare {
    bool operator()(const std::shared_ptr<HuffmanNode>& a, const
std::shared_ptr<HuffmanNode>& b) {
        return a->freq > b->freq;
    }
};

// Генерация кодов символов (рекурсивный обход дерева)
void generateCodes(const std::shared_ptr<HuffmanNode>& root, const std::string&
code, std::unordered_map<char, std::string>& codes) {
    if (!root) return; //если узла не сущетсвует - выходим
    if (!root->left && !root->right) {
        codes[root->ch] = code; //если у узла нет потомков это лист, хранящий
символ
    }
    generateCodes(root->left, code + "0", codes); //влево - добавляем ноль
    generateCodes(root->right, code + "1", codes); //вправо - один
}

// Кодирование строки
```

```
std::string encode(const std::string& data, const std::unordered_map<char,
std::string>& codes) {
    std::string encoded;
    for (char ch : data) {
        encoded += codes.at(ch);
    }
    return encoded;
}

// Декодирование битовой строки
std::string decode(const std::string& encoded, const
std::shared_ptr<HuffmanNode>& root) {
    std::string decoded;
    auto current = root; //начинаем с корня дерева
    for (char bit : encoded) {
        if (bit == '0') {
            current = current->left; //если равен 0 - идём влево
        } else {
            current = current->right;
        }
        if (!current->left && !current->right) {
            decoded += current->ch;
            current = root;
        }
    }
    return decoded;
}

int main() {
    std::string data = "abacabad";

    // 1. Подсчет частот символов
    std::unordered_map<char, int> freq; // Пустой словарь для частот
    for (char ch : data) {
        freq[ch]++;
    }

    // 2. Построение дерева Хаффмана
    std::priority_queue<std::shared_ptr<HuffmanNode>,
std::vector<std::shared_ptr<HuffmanNode>>, Compare> heap;
    for (const auto& pair : freq) {
        heap.push(std::make_shared<HuffmanNode>(pair.first, pair.second));
//создание узлов
    }

    while (heap.size() > 1) {  //строим дерево, до тех пор пока не остался 1
элемент - корень дерева
        auto left = heap.top(); //берем самый верхний эл и удаляем его (т.е с
самой маленькой частотой)
```

```
        heap.pop();
        auto right = heap.top(); //берем следующий
        heap.pop();
        auto merged = std::make_shared<HuffmanNode>(left->freq + right->freq);
//создаем объединённый узел
        merged->left = left; //делаем их дочерними узлами
        merged->right = right;
        heap.push(merged); //возмращаем узел в кучу
    }

    auto root = heap.top(); //корень дерева

    // 3. Генерация кодов
    std::unordered_map<char, std::string> codes; //пустой словарь для кодов
элементов
    generateCodes(root, "", codes);

    // 4. Кодирование строки
    std::string encoded = encode(data, codes);
    std::cout << "Encoded: " << encoded << std::endl;

    // 5. Декодирование
    std::string decoded = decode(encoded, root);
    std::cout << "Decoded: " << decoded << std::endl;

    // Проверка
    std::cout << (data == decoded ? "Success!" : "Error!") << std::endl;

    return 0;
}
```

Источники

https://habr.com/ru/companies/otus/articles/497566/

https://ru.wikipedia.org/wiki/Код_Хаффмана

Ссылка на гитхаб:

https://github.com/antonina-g/Gavrilina_Antonina_Vladimirovna_2_course