**Министерство науки и высшего образования Российской Федерации**

**ФГАОУ ВО «Севастопольский государственный университет»**

Институт информационных технологий

Кафедра «Информационные системы»

**Отчет по лабораторной работе №6**

по дисциплине «Технологии обработки информации»

Выполнил: студент группы

ИС/б-21-2-о

Мельничук В.В.

Принял:

г. Севастополь

**Лабораторная работа №6**

Тема: Алгоритмы сжатия данных

Цель работы: изучить основные виды и алгоритмы сжатия данных и научиться решать задачи сжатия данных по методу Хаффмана и с помощью кодовых деревьев.

1. На основании приведенных в лекции сведений реализуйте алгоритмы сжатия по методу Хаффмана через префиксные коды и на основе кодовых деревьев.

2. Алфавит содержит 7 букв, которые встречаются с вероятностями 0,4; 0,2; 0,1; 0,1; 0,1; 0,05; 0,05. Осуществите кодирование по методу Хаффмана.

3. Закодируйте по алгоритму Хаффмана строку с вашим именем, отчеством, фамилией, датой и местом рождения (например, "Иванова Наталья Николаевна, 1 января 1990 года, город Тверь"). При кодировании не округляйте частоты менее, чем четыре знака после запятой – сокращение точности понижает эффективность кодирования. Подсчитайте коэффициент сжатия.

4. При кодировании по методу Фано все сообщения записываются в таблицу по степени убывания вероятности и разбиваются на две группы примерно (насколько это возможно) равной вероятности. Соответственно этой процедуре из корня кодового дерева исходят два ребра, которым в качестве весов присваиваются полученные вероятности. Двум образовавшимся вершинам приписывают кодовые символы 0 и 1. Затем каждая из групп вероятностей вновь делится на две подгруппы примерно равной вероятности. В соответствии с этим из каждой вершины 0 и 1 исходят по два ребра с весами, равными вероятностям подгрупп, а вновь образованным вершинам приписывают символы 00 и 01, 10 и 11. В результате многократного повторения процедуры разделения вероятностей и образования вершин приходим к ситуации, когда в качестве веса, приписанного ребру бинарного дерева, выступает вероятность одного из данных сообщений. В этом случае вновь образованная вершина оказывается листом дерева, т.к. процесс деления вероятностей для нее завершен. Задача кодирования считается решенной, когда на всех ветвях кодового бинарного дерева образуются листья. Закодируйте по алгоритму Фано данные текстового файла.

Ход работы:

1. Метод Хаффмана



Рисунок 1 – Результат выполнения программы

Листинг:

#include <iostream>

#include <queue>

#include <unordered\_map>

using namespace std;

// Структура узла дерева Хаффмана

struct HuffmanNode {

char data;

int frequency;

HuffmanNode\* left;

HuffmanNode\* right;

HuffmanNode(char data, int frequency) {

this->data = data;

this->frequency = frequency;

this->left = nullptr;

this->right = nullptr;

}

};

// Функция для сравнения узлов дерева Хаффмана при построении очереди с приоритетом

struct CompareNodes {

bool operator()(HuffmanNode\* lhs, HuffmanNode\* rhs) {

return lhs->frequency > rhs->frequency;

}

};

// Функция для построения дерева Хаффмана на основе префиксных кодов

HuffmanNode\* buildHuffmanTreePrefix(const unordered\_map<char, string>& prefixCodes) {

priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, CompareNodes> pq;

for (const auto& pair : prefixCodes) {

char data = pair.first;

string prefixCode = pair.second;

int frequency = prefixCode.length();

HuffmanNode\* node = new HuffmanNode(data, frequency);

pq.push(node);

}

while (pq.size() > 1) {

HuffmanNode\* left = pq.top();

pq.pop();

HuffmanNode\* right = pq.top();

pq.pop();

HuffmanNode\* parent = new HuffmanNode('\0', left->frequency + right->frequency);

parent->left = left;

parent->right = right;

pq.push(parent);

}

return pq.top();

}

// Функция для построения префиксных кодов из дерева Хаффмана

void generatePrefixCodes(HuffmanNode\* root, const string& codePrefix, unordered\_map<char, string>& prefixCodes) {

if (root == nullptr) {

return;

}

if (root->data != '\0') {

prefixCodes[root->data] = codePrefix;

}

generatePrefixCodes(root->left, codePrefix + "0", prefixCodes);

generatePrefixCodes(root->right, codePrefix + "1", prefixCodes);

}

// Функция для сжатия данных с использованием префиксных кодов Хаффмана

string compressUsingPrefixCodes(const string& data, const unordered\_map<char, string>& prefixCodes) {

string compressedData;

for (char c : data) {

compressedData += prefixCodes.at(c);

}

return compressedData;

}

// Функция для декомпрессии данных с использованием префиксных кодов Хаффмана

string decompressUsingPrefixCodes(const string& compressedData, HuffmanNode\* root) {

string decompressedData;

HuffmanNode\* current = root;

for (char c : compressedData) {

if (c == '0') {

current = current->left;

}

else if (c == '1') {

current = current->right;

}

if (current->left == nullptr && current->right == nullptr) {

decompressedData += current->data;

current = root;

}

}

return decompressedData;

}

int main() {

// Задаем исходные данные

std::string data = "ABBCCCDDDDDEEEEE"; // Пример данных для сжатия

// Создаем таблицу префиксных кодов

std::unordered\_map<char, std::string> prefixCodes = {

{'A', "0"},

{'B', "10"},

{'C', "110"},

{'D', "1110"},

{'E', "1111"}

};

// Сжимаем данные

std::string compressedData = compressUsingPrefixCodes(data, prefixCodes);

std::cout << "Compressed data: " << compressedData << std::endl;

return 0;

}

1. Кодирование алфавита

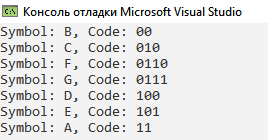


Рисунок 2 – Результат работы программы

Листинг:

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <string>

// Структура узла дерева Хаффмана

struct HuffmanNode {

char data;

int frequency;

HuffmanNode\* left;

HuffmanNode\* right;

HuffmanNode(char data, int frequency) {

this->data = data;

this->frequency = frequency;

this->left = nullptr;

this->right = nullptr;

}

};

// Функция для сравнения узлов дерева Хаффмана при построении очереди с приоритетом

struct CompareNodes {

bool operator()(HuffmanNode\* lhs, HuffmanNode\* rhs) {

return lhs->frequency > rhs->frequency;

}

};

// Функция для построения дерева Хаффмана на основе вероятностей символов

HuffmanNode\* buildHuffmanTree(const std::unordered\_map<char, double>& probabilities) {

std::priority\_queue<HuffmanNode\*, std::vector<HuffmanNode\*>, CompareNodes> pq;

// Создаем листовые узлы для каждого символа

for (const auto& pair : probabilities) {

char data = pair.first;

double probability = pair.second;

int frequency = static\_cast<int>(probability \* 1000); // Умножаем на 1000 для учета точности

HuffmanNode\* node = new HuffmanNode(data, frequency);

pq.push(node);

}

// Строим дерево Хаффмана

while (pq.size() > 1) {

HuffmanNode\* left = pq.top();

pq.pop();

HuffmanNode\* right = pq.top();

pq.pop();

HuffmanNode\* parent = new HuffmanNode('\0', left->frequency + right->frequency);

parent->left = left;

parent->right = right;

pq.push(parent);

}

return pq.top();

}

// Функция для генерации префиксных кодов из дерева Хаффмана

void generatePrefixCodes(HuffmanNode\* root, const std::string& codePrefix, std::unordered\_map<char, std::string>& prefixCodes) {

if (root == nullptr) {

return;

}

if (root->data != '\0') {

prefixCodes[root->data] = codePrefix;

}

generatePrefixCodes(root->left, codePrefix + "0", prefixCodes);

generatePrefixCodes(root->right, codePrefix + "1", prefixCodes);

}

int main() {

// Задаем алфавит и вероятности символов

std::unordered\_map<char, double> probabilities = {

{'A', 0.4},

{'B', 0.2},

{'C', 0.1},

{'D', 0.1},

{'E', 0.1},

{'F', 0.05},

{'G', 0.05}

};

// Строим дерево Хаффмана

HuffmanNode\* root = buildHuffmanTree(probabilities);

// Генерируем префиксные коды

std::unordered\_map<char, std::string> prefixCodes;

generatePrefixCodes(root, "", prefixCodes);

// Выводим префиксные коды для каждого символа

for (const auto& pair : prefixCodes) {

std::cout << "Symbol: " << pair.first << ", Code: " << pair.second << std::endl;

}

return 0;

}

1. Кодирование своих данных



Рисунок 3 – Результат работы программы

Листинг:

#include <iostream>

#include <unordered\_map>

#include <queue>

#include <string>

#include <iomanip>

#include <locale>

// Структура узла дерева Хаффмана

struct HuffmanNode {

char data;

double frequency;

HuffmanNode\* left;

HuffmanNode\* right;

HuffmanNode(char data, double frequency) {

this->data = data;

this->frequency = frequency;

this->left = nullptr;

this->right = nullptr;

}

};

// Функция для сравнения узлов дерева Хаффмана при построении очереди с приоритетом

struct CompareNodes {

bool operator()(HuffmanNode\* lhs, HuffmanNode\* rhs) {

return lhs->frequency > rhs->frequency;

}

};

// Функция для построения дерева Хаффмана на основе вероятностей символов

HuffmanNode\* buildHuffmanTree(const std::unordered\_map<char, double>& probabilities) {

std::priority\_queue<HuffmanNode\*, std::vector<HuffmanNode\*>, CompareNodes> pq;

// Создаем листовые узлы для каждого символа

for (const auto& pair : probabilities) {

char data = pair.first;

double probability = pair.second;

HuffmanNode\* node = new HuffmanNode(data, probability);

pq.push(node);

}

// Строим дерево Хаффмана

while (pq.size() > 1) {

HuffmanNode\* left = pq.top();

pq.pop();

HuffmanNode\* right = pq.top();

pq.pop();

HuffmanNode\* parent = new HuffmanNode('\0', left->frequency + right->frequency);

parent->left = left;

parent->right = right;

pq.push(parent);

}

return pq.top();

}

// Функция для генерации префиксных кодов из дерева Хаффмана

void generatePrefixCodes(HuffmanNode\* root, const std::string& codePrefix, std::unordered\_map<char, std::string>& prefixCodes) {

if (root == nullptr) {

return;

}

if (root->data != '\0') {

prefixCodes[root->data] = codePrefix;

}

generatePrefixCodes(root->left, codePrefix + "0", prefixCodes);

generatePrefixCodes(root->right, codePrefix + "1", prefixCodes);

}

int main() {

setlocale(LC\_ALL, "Russian");

// Задаем строку для кодирования

std::string inputString = "Щукин Андрей Денисович, 31 августа 2004 года, город Севастополь";

// Подсчитываем частоты символов в строке

std::unordered\_map<char, int> charFrequencies;

for (char c : inputString) {

charFrequencies[c]++;

}

// Вычисляем вероятности символов

std::unordered\_map<char, double> probabilities;

for (const auto& pair : charFrequencies) {

char c = pair.first;

int frequency = pair.second;

double probability = static\_cast<double>(frequency) / inputString.length();

probabilities[c] = probability;

}

// Строим дерево Хаффмана

HuffmanNode\* root = buildHuffmanTree(probabilities);

// Генерируем префиксные коды

std::unordered\_map<char, std::string> prefixCodes;

generatePrefixCodes(root, "", prefixCodes);

// Кодируем строку

std::string encodedString;

for (char c : inputString) {

encodedString += prefixCodes[c];

}

// Выводим префиксные коды для каждого символа

std::cout << "Префиксные коды:" << std::endl;

for (const auto& pair : prefixCodes) {

char c = pair.first;

std::string code = pair.second;

std::cout << "'" << c << "': " << code << std::endl;

}

// Выводим закодированную строку

std::cout << "Закодированная строка: " << encodedString << std::endl;

// Рассчитываем количество бит в исходной строке и закодированной строке

int originalBits = inputString.length() \* 8;

int encodedBits = encodedString.length();

// Рассчитываем коэффициент сжатия

double compressionRatio = static\_cast<double>(originalBits) / encodedBits;

// Выводим коэффициент сжатия

std::cout << "Коэффициент сжатия: " << std::fixed << std::setprecision(2) << compressionRatio << std::endl;

return 0;

}

1. Алгоритм Фано

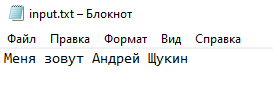


Рисунок 4 – Входные данные

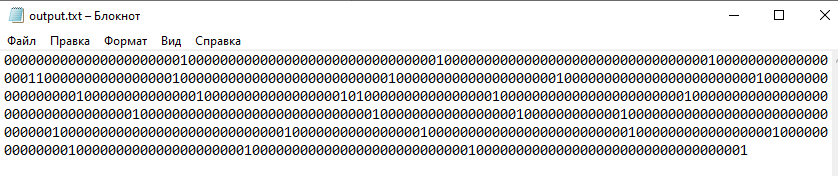


Рисунок 5 – Выходные данные

Листинг:

#include <iostream>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <fstream>

using namespace std;

// Структура узла дерева Фано

struct FanoNode {

char symbol;

double probability;

FanoNode\* left;

FanoNode\* right;

FanoNode(char sym, double prob) : symbol(sym), probability(prob), left(nullptr), right(nullptr) {}

};

// Класс для сравнения узлов при построении очереди с приоритетом

struct FanoNodeComparator {

bool operator()(const FanoNode\* a, const FanoNode\* b) const {

return a->probability < b->probability;

}

};

// Функция для построения дерева Фано

FanoNode\* buildFanoTree(const unordered\_map<char, double>& probabilities) {

priority\_queue<FanoNode\*, vector<FanoNode\*>, FanoNodeComparator> pq;

// Создание узлов для каждого символа с его вероятностью

for (const auto& entry : probabilities) {

char symbol = entry.first;

double probability = entry.second;

FanoNode\* node = new FanoNode(symbol, probability);

pq.push(node);

}

// Построение дерева Фано

while (pq.size() > 1) {

FanoNode\* left = pq.top();

pq.pop();

FanoNode\* right = pq.top();

pq.pop();

FanoNode\* parent = new FanoNode('\0', left->probability + right->probability);

parent->left = left;

parent->right = right;

pq.push(parent);

}

return pq.top();

}

// Функция для генерации кодов символов на основе дерева Фано

void generateFanoCodes(FanoNode\* root, string code, unordered\_map<char, string>& codes) {

if (!root)

return;

if (root->symbol != '\0') {

codes[root->symbol] = code;

return;

}

generateFanoCodes(root->left, code + "0", codes);

generateFanoCodes(root->right, code + "1", codes);

}

// Функция для кодирования данных файла по алгоритму Фано

void encodeFile(const string& inputFilename, const string& outputFilename) {

// Чтение данных файла и подсчет вероятностей символов

unordered\_map<char, double> probabilities;

ifstream inputFile(inputFilename);

char ch;

int totalSymbols = 0;

while (inputFile.get(ch)) {

probabilities[ch]++;

totalSymbols++;

}

inputFile.close();

// Расчет вероятностей символов

for (auto& entry : probabilities) {

entry.second /= totalSymbols;

}

// Построение дерева Фано

FanoNode\* root = buildFanoTree(probabilities);

// Генерация кодов символов

unordered\_map<char, string> codes;

generateFanoCodes(root, "", codes);

// Запись закодированных данных в файл

ofstream outputFile(outputFilename);

inputFile.open(inputFilename);

while (inputFile.get(ch)) {

outputFile << codes[ch];

}

inputFile.close();

outputFile.close();

}

int main() {

string inputFilename = "input.txt";

string outputFilename = "output.txt";

encodeFile(inputFilename, outputFilename);

return 0;

}

**Вывод:** в ходе выполнения лабораторной работы были изучены основные виды и алгоритмы сжатия данных , а так же решены задачи сжатия данных по методу Хаффмана и с помощью кодовых деревьев.

**Контрольные вопросы**

1. При кодировании каких данных можно использовать сжатие данных с потерями? Ответ обоснуйте.

Сжатие данных с потерями обычно применяется для данных, в которых допустима потеря части информации без существенного влияния на их восприятие или использование. Примерами таких данных могут быть аудио- и видеофайлы, изображения с высокой степенью сжатия (например, форматы JPEG), данные сенсоров и т. д. В этих случаях потеря некоторых деталей может быть приемлемой, если обеспечивается достаточно высокое качество воспроизведения или воспроизведение основной информации.

2. В чем преимущества и недостатки статических методов и словарного сжатия?

Преимущества статических методов сжатия:

Простота реализации и применения.

Быстрое сжатие и распаковка, поскольку алгоритм использует фиксированный набор правил.

Отсутствие необходимости в создании и хранении словаря или других вспомогательных структур данных.

Недостатки статических методов сжатия:

Менее эффективное сжатие по сравнению со словарными методами.

Отсутствие адаптивности к конкретным данным или структуре данных.

Преимущества словарного сжатия:

Более высокий уровень сжатия по сравнению со статическими методами.

Адаптивность к конкретным данным, что позволяет получить лучшую эффективность сжатия.

Недостатки словарного сжатия:

Больший объем памяти, необходимый для хранения словаря.

Дополнительное время, требуемое для создания и обновления словаря.

3. Каким образом кодирование по алгоритму Хаффмана через префиксный код гарантирует минимальную длину кода?

Алгоритм Хаффмана через префиксный код гарантирует минимальную длину кода благодаря использованию префиксных кодов. Префиксный код - это код, в котором нет ни одного кодового слова, являющегося префиксом другого кодового слова. В алгоритме Хаффмана каждому символу присваивается код, который является префиксным кодом. Таким образом, ни одно кодовое слово не является префиксом другого, что обеспечивает однозначность декодирования и минимальную длину кода.

4. За счет чего в методе Хаффмана поддерживается однозначность соответствия кода кодируемому символу?

В методе Хаффмана однозначность соответствия кода кодируемому символу обеспечивается путем построения дерева Хаффмана. В дереве каждый символ представлен листовым узлом, а пути от корня до листов представляют коды символов. Поскольку нет двух символов с одинаковыми кодами, соответствие кода и символа остается однозначным.

5. Почему алгоритм Хаффмана малоэффективен для файлов маленьких размеров?

Алгоритм Хаффмана имеет некоторые накладные расходы на создание и хранение дерева Хаффмана и словаря символов, которые могут быть существенными для файлов маленьких размеров. Кроме того, для небольших файлов сложно достичь значительного сжатия, так как вероятности символов могут быть близкими, и построенное дерево Хаффмана может не дать значительного выигрыша в эффективности сжатия.

1. Выполните кодирование по методу Хаффмана через префиксный код символов, которые встречаются с вероятностями 0,3; 0,2; 0,1; 0,1; 0,1; 0,05; 0,05; 0,04; 0,03; 0,03. Сравните полученный результат с данными программной реализации.

7. Докажите, что метод Хаффмана кодирует информацию без потерь.

Доказательство основывается на двух ключевых свойствах метода Хаффмана:

а) Префиксный код: Коды Хаффмана являются префиксными, то есть ни один код не является префиксом другого кода. Это обеспечивает однозначность декодирования: нет возможности двусмысленного интерпретирования закодированной последовательности.

б) Восстановление дерева Хаффмана: При раскодировании используется та же структура дерева Хаффмана, что и при кодировании. Дерево можно восстановить из декодированных данных, так как оно сохраняет информацию о структуре и кодах символов.