МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Динамическое кодировцание и декодирование по Хаффману – сравнительное исследование со “статическим” методом

Студент гр. 9382 _____ Демин В.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Демин В.В.

Группа 9382

Тема работы : Динамическое кодировцание и декодирование по Хаффману – сравнительное исследование со “статическим” методом

Исходные данные:

На вход программе подается текст для кодирования или декодирования

Содержание пояснительной записки:

«Содержание», «Введение», «Заключение», «Список использованных источников», «Ход выполнения работы», «Описание алгоритмы», «Описание интерфейса пользователя», «Программный код», «Исследование»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 14.12.2020

Дата защиты реферата: 14.12.2020

Студент _____ Демин В.В.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

Была написано программа для анализа методов кодирования по таким данным как количество символов в строке и время за которое проводится кодирование или декодирование. По этим данным программа строит графики эффективности сжатия от количества символов и времени от количества символов.

Код программы приведён в приложении А.

SUMMARY

A program was written to analyze coding methods for such data as the number of characters in a line and the time the encoding or decoding is performed. Based on this data, the program builds graphs of the compression efficiency versus the number of characters and the time versus the number of characters.

Progrgram code is given in attachment A.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной курсовой работы сравнить статический метод Хаффмана с динамическим кодированием и декодированием Хаффмана. Для того чтобы анализировать методы кодирования и декодирования необходимо собрать статистику для исследования. Для этого была написана программа на C++ с использованием библиотеки Qt, которая генерирует данные для исследования и строит по полученным характеристическим значениям(такие как, время алгоритма, длина строки до обработки, длина строки после обработки, длина алфавита).

1. Ход Выполнения работы

1.1. Постановка задачи

Для исследования двух методов кодирования необходимо собрать статистические данные, а именно время выполнения кодирования, алфавит кодируемой строки, размеры строк до кодирования и после в битах. Эти данные нам покажут насколько эффективны методы.

1.2. Описание алгоритмов

1.2.1 Статическое кодирование Хаффмана

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (H-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.

1.2.2 Статическое декодирование Хаффмана

Для декодирования кода Хаффмана необходимо знать алфавит кодированных символов, чтобы восстановить исходную строку. Зная

алфавит собирается дерево Хаффмана. И по нему происходит сопоставление строки декодирования и кодов в полученном дереве.

1.2.3 Динамическое кодирование Хаффмана

Лучше начинать моделирование с пустого дерева и добавлять в него символы только по мере их появления в сжимаемом сообщении. Но это приводит к очевидному противоречию: когда символ появляется в сообщении первый раз, он не может быть закодирован, так как его еще нет в дереве кодирования.

Чтобы разрешить это противоречие, введем специальный ESCAPE код, который будет означать, что следующий символ закодирован вне контекста модели сообщения. Например, его можно передать в поток сжатой информации как есть, не кодируя вообще. Метод "ЗакодироватьСимвол" в алгоритме адаптивного кодирования Хаффмана можно записать следующим образом.

```
ЗакодироватьСимвол(Символ)
{
 Если СимволУжеЕстьВТаблице(Символ)
 ВыдатьКодХаффманаДля(Символ);
 Иначе
 {
 ВыдатьКодХаффманаДля(ESC);
 ВыдатьСимвол(Символ);
 }
}
```

Использование специального символа ESC подразумевает определенную инициализацию дерева до начала кодирования и декодирования: в него помещаются 2 специальных символа: ESC и EOF (конец файла), с весом, равным 1.

Поскольку процесс обновления не коснется их веса, то по коду кодирования они будут перемещаться на самые удаленные ветви дерева и иметь самые длинные коды.

1.2.3 Динамическое кодирование Хаффмана

Декодер при восстановлении исходной последовательности работает подобно кодеру, т.е. он также последовательно строит дерево Хаффмана по мере декодирования последовательности, что позволяет корректно определять исходные символы.

1.3. Описание функций

Рассмотрим классы кодирования и декодирования

1.3.1 Статическое кодирование Хаффмана

1. vector<pair<char, string>> coding(string &str) override; - основная функция кодирования. String& str – строка которую нужно закодировать. Возвращант алфавит символов для дальнейшего декодирования, если нужно.
2. string get_code() override; - возращает закодированную строку.
3. void formAlpabet(string const &str, vector<pair<char, int>> &alphabet); - функция которая формирует алфавит символов с частотой в тексте. Пробегает по строке и считает веса символов. string const &str – исходная строка для кодирования, vector<pair<char, int>> &alphabet- алфавит в который запишется необходимая информация.
4. node * makeBinaryTree(vector<pair<char, int>> alphabet); - функция формирования бинарного дерева. vector<pair<char, int>> alphabet – алфавит для построения с весами символов. Возвращает корень построенного дерева.
5. void makeCode(node *tree, char ch, string &code, bool &flag); - рекурсивная функция, которая обходит дерево с построением кода. node *tree – узел с которого необходимо

произвести построение. char ch – символ для которого строится код. string &code – строка в которую запишется код символа. bool &flag – флаг, который покажет сформировался ли код символа.

1.3.2 Статическое декодирование Хаффмана

1. vector<pair<char, string>> decoding(string &str,vector<pair<char, string>> al) override – основная функция декодирования. string &str – строка, которую необходимо декодировать. vector<pair<char, string>> al) – алфавит символов с кодом символа.

Возвращает полученный алфавит.

2. string get_code() override; - функция которая возвращает декодированную строку

1.3.3 Динамическое кодирование Хаффмана

1. string get_code() override; - функция которая возвращает декодированную строку.
2. vector<pair<char, string>> coding(string &str) override – основная функция кодирования. string &str – строка, которую необходимо кодировать.
3. Возвращает полученный алфавит.
4. void update(char a) – функция для обновления дерева

1.3.4 Динамическое декодирование Хаффмана

1. string get_code() override; - функция которая возвращает декодированную строку.

2. vector<pair<char, string>> decoding(string &str,vector<pair<char, string>> al) override – основная функция декодирования. string &str – строка, которую необходимо декодировать. vector<pair<char, string>> al) – алфавит символов с кодом символа.

Возвращает полученный алфавит.

3. void update(char a) – функция для обновления дерева

1.4. Описание интерфейса пользователя

При запуске программы выведется меня для взаимодействия программы. Статическое и динамическое кодирование Хаффмана, Статическое и динамическое декодирование Хаффмана – кнопки для открытия окна, которое позволяет произвести определенные операции со строкой. Кнопка генератор октрывает окно для демонстрации генератора строк для анализа. Кнопка статистика открывает окно для демонстрации собранной статистики методов для последующего анализа. В этом окне команда обновить обновляет графики на экране. Можно задать количество генераций строк, чтобы собрать больше данных. И стереть все полученные данные.

2. ИССЛЕДОВАНИЕ

2.1. Генератор строк.

Генератор строк генерирует строку заданного размера, символы встречаются в строке равномерно, что позволяет анализировать методы от среднего случая к худшему. Чтобы добиться лучшего случая необходимо использовать нормальное распределение, при котором частота некоторых символов будет в разы отличаться от остальных. Количество символов ограничено только символами печатными символами таблицы ascii, так как если брать все символы, которые можно закодировать 8 битами. И при равномерном расределении будет максимально не эффективное кодирование.

2.2. Накопление статистики и анализ

В программе при каждом кодировании или декодеровании строки данные заносятся в базу для дальнейшего анализа. Чтобы получить сразу массивный объем данных для анализа. Мы используем генератор строк, который будет генерировать строки длиной от 1 до заданного нами значения. Для анализа поставленной задачи было сгенирировано 3000 строк, таким образом для каждого метода было собрано 3000 строк данных по 4 характеристики, а именно размер алфавита, размер строки до кодирования, размер строки после кодирования. Собранные данные можно увидеть на рисунке 1.

Исходя из полученных данных можно сделать следующие выводы:

1. Статическое декодирование Хаффмана благодаря тому, что алфавит декодирования задан изначально, позволяет добиться максимальной производительности. От этого вытекает проблема, хранения алфавит декодирования, но при больших файлах, эта проблема устраняется высокой эффективностью.
2. Статическое кодирование Хаффмана быстрее, чем динамическое кодирование в несколько раз. В статическом методе Хаффмана происходит обход строки перед началом кодирования, далее

строится дерево Хаффмана и по нему кодируется строка. В то же время в динамическом кодирование нет обхода строки перед началом кодирования, но при этом дерево Хаффмана строится адаптивно, то есть оно изменяется при прохождении строки, что позволит получить выйгрыш в кодировании строки, но на перестройку дерево необходимо затрачивать время.

3. Динамическое кодирование чуть быстрее чем, динамическое декодирование. Это связано с тем, что при декодирование происходят такие же операции, как и при кодировании**. В** частности адаптивное построение дерева Хаффмана.

4. Сравнивая с теоретическими оценками, динамический метод при адаптивном перестроение дерева имеет сложность O(log(k)) для вставки в дерево и O(log(k)) для поиска элемента в дереве. При этом кодирование строки составляет сложность O(n). Таким образом общая сложность будет составлять O(log^2(k)*n).

Теоретическая оценка для статического метода O(log (n)*n) при использовании дерева.

Проанализировав график можно прийти к выводу, что наши данные совпадают с теоретическим.

5. Эффективность сжатия Динамического кодирования зависит от частотности появления новых символов. Ведь при появлении нового символа необходимо записать код «нулевого» символа и код самого символа. Таким образом при маленьких строках, но при равномерном распределении частоты символ эффективность метода уменьшается. Статическому методу не нужно записовать коды изначальных символов, записывается сразу код из дерева Хаффмана.

6. При равномерном распределении частотности символов, эффективность сжатия динамического и статического методы примерно равны. Исходя из того, что в строка с большой длиной при

равномерном распределении, не будет преобладающего элемента, который бы позволил дать больший прирост эффективности.

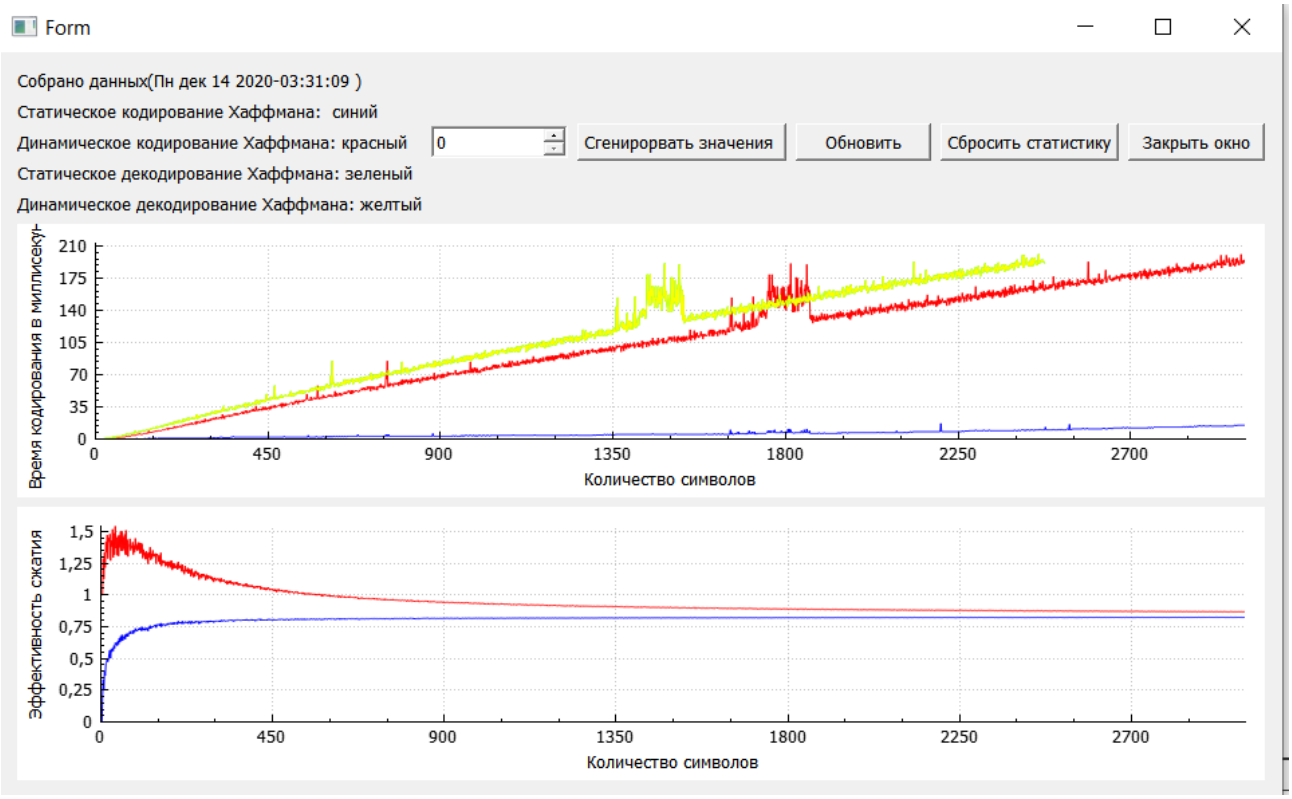

Рис. 1

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы была создана программа, позволяющая анализировать методы кодирования и декодирования, при генерации строк с равномерным распределением встречаймости символов в строке. Интерпретировав полученную статистику, можем сделать вывод, что динамический метод проигрывает в эффективности применения к строке при равномерном распределении частотности символов. Если необходима скорость сжатия, то лучше использовать статический метод, если необходимо эффективность сжатия, то для этой цели можно применить масштабирование символов для динамического метода, что при не равномерном распределении даст огромную эффективность в сжатии.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Адаптивные коды Хаффмана URL:https://scask.ru/a_lect_cod.php?id=32
2. Код Хаффмана URL: https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%A5%D0%B0%D1%84%D1%84%D0%BC%D0%B0%D0%BD%D0%B0
3. Алгоритм Хаффмана на пальцах URL: https://habr.com/ru/post/144200/
4. Huffman Encoding URL:https://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html#:~:text=Operation%20of%20the%20Huffman%20algorithm,iterations%2C%20one%20for%20each%20item.
5. Adaptive Huffman coding – FGK URL: Adaptive Huffman coding – FGK

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД

Название файла: coderstatichaffman,h

```
#ifndef CODERSTATICHAFFMAN_H
#define CODERSTATICHAFFMAN_H
#include "ICoder.h"
#include "BinaryTree.h"


class CoderStaticHaffman:public ICoder
{
    string full_code;
    void formAlpabet(string const &str, vector<pair<char, int>> &alphabet);
    static bool compair_alphabet(pair<char, int> a1, pair<char, int> a2);
    node * makeBinaryTree(vector<pair<char, int>> alphabet);
    void makeCode(node *tree, char ch, string &code, bool &flag);
    FileLog* file;

public:
    CoderStaticHaffman();
    vector<pair<char, string>> coding(string &str) override;
    string get_code() override;
    void notify(string message) override;

};

#endif // CODERSTATICHAFFMAN_H
```

Название файла: coderdynamichuffman,h

```
#ifndef CODERDYNAMICHUFFMAN_H
#define CODERDYNAMICHUFFMAN_H
#include "ICoder.h"
#include <BinaryTree.h>


class CoderDynamicHuffman:public ICoder {
```

```
    Node *root;
    string full_code;
    unordered_map<char, Node *> getNode;
    vector<Node *> levelOrder();
    void update(char a);
    void new_node();
    FileLog* file;
public:

    string get_code() override;

    CoderDynamicHuffman();

    vector<pair<char,string>> coding(string& str) override;
    void notify(string message) override;
};

#endif // CODERDYNAMICHUFFMAN_H
```

Название файла: decoderdynamichuffman,h

```
#ifndef DECODERDYNAMICHUFFMAN_H
#define DECODERDYNAMICHUFFMAN_H
#include "IDecoder.h"
#include <cmath>
#include <BinaryTree.h>



class DecoderDynamicHuffman: public IDecoder
{
    string full_code;
    Node *root;
    unordered_map<char, Node *> getNode;
    FileLog* file;
    vector<Node *> levelOrder();

    void update(char a);
    void new_node();
```

```
    public:
        DecoderDynamicHuffman();
        vector<pair<char, string>> decoding(string
&str,vector<pair<char, string>> al) override;
        string get_code() override;
        bool need_alphabet() override;
        void notify(string message) override;
    };

    #endif // DECODERDYNAMICHUFFMAN_H
```

Название файла: decoderstatichuffman,h

```
    #ifndef DECODERSTATICHUFFMAN_H
    #define DECODERSTATICHUFFMAN_H
    #include "IDecoder.h"
    #include <QDebug>

    class DecoderStaticHuffman: public IDecoder
    {
        string full_code;
        FileLog* file;
    public:
        DecoderStaticHuffman();
         vector<pair<char, string>> decoding(string
&str,vector<pair<char, string>> al) override;
         string get_code() override;
         bool need_alphabet() override;
         void notify(string message) override;
    };

    #endif // DECODERSTATICHUFFMAN_H
```