Московский авиационный институт
(национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Курсовой проект
по курсу «Дискретный анализ»

Архиватор

Студент: Лагуткина Мария Сергеевна
Группа: М8О-206Б-19
Преподаватель:

Дата:

Оценка:

Москва, 2021

1 Постановка задачи

Задание
Необходимо реализовать два известных метода сжатия данных для сжатия одного файла.
Формат запуска должен быть аналогичен формату запуска программы gzip. Должны быть поддерживаться следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Вариант: Арифметическое кодирование + LZW

2 LZW

Существует довольно большое семейство LZ-подобных алгоритмов, различающихся, например, методом поиска повторяющихся цепочек. Один из достаточно простых вариантов этого алгоритма, например, предполагает, что во входном потоке идет либо пара <счетчик, смещение относительно текущей позиции>, либо просто <счетчик> "пропускаемых" байт и сами значения байтов. При разархивации для пары <счетчик, смещение> копируются <счетчик> байт из выходного массива, полученного в результате разархивации, на <смещение> байт раньше, а <счетчик> (т.е. число равное счетчику) значений "пропускаемых" байт просто копируются в выходной массив из входного потока. Данный алгоритм является несимметричным по времени, поскольку требует полного перебора буфера при поиске одинаковых подстрок.

После публикации LZ78, который воспринимался читателями научного журнала как абстракция, нежели то, что можно положить в основу программного продукта, в 1984 году Терри Уэлч опубликовал Модификацию алгоритма LZ78, которая получила название LZW (Lempel-Ziv-Welch).

LZW построен вокруг таблицы фраз (словаря), которая отображает строки символов сжимаемого сообщения в коды фиксированной длины. Таблица обладает так называемым свойством предшествования, то есть для каждой фразы словаря, состоящей из некоторой фразы w и символа K, фраза w тоже содержится в словаре.
Алгоритм
Алгоритм работы кодера LZW можно описать следующим образом: Проинициализировать словарь односимвольными фразами, соответствующими символам входного алфавита (обычно это 256 ASCII символов)

```
Прочитать первый символ сообщения в текущую фразу w;
Шаг алгоритма:
Прочитать очередной символ сообщения K;
Если КОНЕЦ_СООБЩЕНИЯ
```

```
Выдать код w;
ВЫХОД;
Конец Если
Если фраза wK уже есть в словаре,
Заменить w на код фразы wK;
Повторить Шаг алгоритма;
Иначе
Выдать код w;
Добавить wK в словарь;
Повторить Шаг алгоритма;
Конец Если;
```

Описанный алгоритм не пытается оптимально выбирать фразы для добавления в словарь или оптимально разбирать сообщение. Однако в силу его простоты он может быть эффективно реализован.

Декодер LZW использует тот же словарь, что и кодер, строя его по аналогичным правилам при восстановлении сжатого сообщения. Каждый считываемый код разбивается с помощью словаря на предшествующую фразу w и символ K. Затем рекурсия продолжается для предшествующей фразы w до тех пор, пока она не окажется кодом одного символа, что и завершает декомпрессию этого кода. Обновление словаря про¬исходит для каждого декодируемого кода, кроме первого. После завершения декодирования кода его последний символ, соединенный с предыдущей фразой, добавляется в словарь. Новая фраза получает то же значение кода (позицию в словаре), что присвоил ей кодер. Так шаг за шагом декодер восстанавливает тот словарь, который построил кодер. У этого алгоритма есть недостаток: он не будет работать в исключительной ситуации. Исключительная ситуация складывается тогда, когда кодер пытается закодировать сообщение KwKwK, где фраза Kw уже присутствует в словаре. Кодер выделит Kw, выдаст код(Kw) и добавит KwK в словарь. Затем он выделит KwK и пошлет только что созданный код (KwK). Декодер при получении кода(KwK) еще не добавил этот код в словарь, потому что еще не знает символ-расширение предыдущей фразы. Тем не менее, когда декодер встречает неизвестный ему код, он может определить, какой символ выдавать первым. Это символ-расширение предыдущей фразы, который будет последним символом текущей фразы, который был последним символом предыдущей фразы, который был последним раскодированным символом.

```
КОД = Прочитать первый код сообщения();
ПредыдущийКОД = КОД;
Выдать символ K,у которого код(K) == КОД;
ПоследнийСимвол = K
Следующий код:
```

```
КОД = Прочитать очередной код сообщения(); ВходнойКОД = КОД;
Если КОНЕЦ_СООБЩЕНИЯ
ВЫХОД;
Конец Если;
Если Неизвестен(КОД) // Обработка исключительной ситуации
Выдать(ПоследнийСимвол)
КОД = ПредыдущийКОД
ВходнойКОД = код (ПредыдущийКОД,ПоследнийСимвол)
Конец Если;
Следующий символ:
Если КОД == код(wK)
В_СТЕК ( K );
КОД = код(w);
Повторить Следующий символ;
Иначе если КОД == код(K)
Выдать K;
ПоследнийСимвол = K;
Пока стек не пуст
Выдать (  ИЗ_СТЕКА()  );
Конец пока;
Добавить в словарь (Предыдущий КОД,K);
ПредыдущийКОД = ВходнойКОД;
Повторить СледующийКОД;
Конец Если;
```

3 Арифметическое кодирование

При арифметическом кодировании текст представляется вещественными числами в интервале от 0 до 1. По мере кодирования текста, отображающий его интервал уменьшается, а количество битов для его представления возрастает. Очередные символы текста сокращают величину интервала исходя из значений их вероятностей, определяемых моделью. Более вероятные символы делают это в меньшей степени, чем менее вероятные, и, следовательно, довабляют меньше битов к результату.

Алгоритм

В общем виде алгоритм арифметического кодирования может быть описан следующим образом:

```
НижняяГраница = 0.0; ВерхняяГраница= 1.0;
Пока ((ОчереднойСимвол = ДайОчереднойСимвол()) != КОНЕЦ)
```

```
Интервал = ВерхняяГраница -НижняяГраница;
ВерхняяГраница = НижняяГраница + Интервал * ВерхняяГраницаИнтервалаДля
(ОчереднойСимвол);
НижняяГраница = НижняяГраница + Интервал * НижняяГраницаИнтервалаДля
(ОчереднойСимвол);
Конец Пока
Выдать (НижняяГраница)
```

Алгоритм арифметического декодирования может быть описан следующим образом:

```
Число = ПрочитатьЧисло();
Всегда
Символ =
НайтиСимволВИнтервалКоторогоПопадаетЧисло(Число)
Выдать (Символ) Интервал = ВерхняяГраницаИнтервалаДля(Символ) -
НижняяГраницаИнтервалаДля (Символ);
Число = Число -НижняяГраницаИнтервалаДля(Символ); Число = Число / Интервал;
Конец Всегда
```

В приведенном алгоритме не рассматривается вопрос остановки декодера, однако декодер должен останавливаться при обнаружении EOF.

В данном курсовом проекте используется алгоритм адаптивного арифметического кодирования.

Адаптивная модель изменяет частоты уже найденных в тексте символов. В начале все счетчики могут быть равны, что отражает отсутствие начальных данных, но по мере просмотра каждого входного символа они изменяются, приближаясь к наблюдаемым частотам. И кодировщик, и декодировщик используют одинаковые начальные значения и один и тот же алгоритм обновления, что позволит их моделям всегда оставаться на одном уровне. Кодировщик получает очередной символ, кодирует его и изменяет модель. Декодировщик определяет очередной символ на основании своей текущей модели, а затем обновляет ее. Обновление модели довольно дорого по причине необходимости поддержания накопленных сумм.

Эксперименты на различных типах данных показывают, что арифметическое кодирование всегда дает результаты не хуже, чем кодирование Хаффмана. В некоторых случаях выигрыш может быть очень существенным. Однако в силу того, что объем вычислений, необходимых при работе алгоритма арифметического кодирования, значительно выше, чем при кодировании по методу Хаффмана, он работает медленнее. Арифметическое кодирование может быть использовано в тех случаях, когда степень сжатия важнее, чем временные затраты на сжатие информации.

4 Основные файлы программы

Файлы программы: lzw.hpp, arifm.hpp, main.cpp. Сборка производится с помощью makefile.
Программа собирается в исполняемый файл gzip. Для запуска программы подаются агрументы ./gzip [-cdklrt19]... [file...].
lzw.hpp
В заголовочном файле lzw.hpp реализован класс LZW для кодирования и декодирования по алгоритму LZW. Для хранения словаря используется Trie.

```
1  #pragma once
2  #include <iostream>
3  #include <vector>
4  #include <map>
5  #include <string>
6  #include <math.h>
7  #include <bitset>
8  #include <fstream>
9  #include <algorithm>
10
11 class TrieNode {
12 private:
13     std::map<uint8_t, TrieNode*> path;
14     uint32_t code;
15
16 public:
17     friend class Trie;
18     TrieNode( uint32_t num):code(num) {}
19     void Clear();
20     virtual ~TrieNode();
21 };
22
23 class Trie {
24 private:
25     TrieNode* currentNode;
26     std::vector<TrieNode*> path;
27 public:
28     Trie() :
29         path(0xff + 1), //255
30         maxCode(1),
31         nextIncrease(2),
32         codeLen(1),
33         currentNode(nullptr) {
34         uint8_t sym = 0x0;
35         for (uint32_t i = 0; i <= 0xff; ++sym, ++i) { //conversion from char to string
36             path[sym] = new TrieNode(maxCode);
37             if (maxCode == nextIncrease) {
38                 nextIncrease *= 2;
```

```
39                 ++codeLen;
40             }
41             ++maxCode;
42         }
43     }
44 
45     uint32_t maxCode;      //next code to give, used in calculating of codeLen
46     uint32_t codeLen;      //current lenght of code in bits
47     uint32_t nextIncrease;//next value of maxCode that will increase codeLen
48 
49     void Clear();
50     uint32_t GetCode(std::string& str);
51     TrieNode* CheckWord(uint8_t sym);
52     void AddWord(std::string & str);
53     virtual ~Trie();
54 };
55 
56 void TrieNode::Clear() {
57     for (std::pair<uint8_t, TrieNode*> elem : path){
58         delete elem.second;
59     }
60         path.clear();
61 }
62 
63 TrieNode::~TrieNode() {
64     for (std::pair<uint8_t, TrieNode*> elem : path)
65         delete elem.second;
66 }
67 
68 void Trie::Clear() {
69     //root array stays in place
70     maxCode = 0xff + 2;
71     nextIncrease = 512;
72     codeLen = 9;
73     currentNode = nullptr;
74     for (TrieNode* node : path){
75         node->Clear();
76     }
77 }
78  uint32_t Trie::GetCode(std::string& str) {
79     TrieNode* currNode = path[(uint8_t)str[0]];
80     for (int i = 1; i < str.length(); ++i)  {
81         currNode = currNode->path[(uint8_t)str[i]];
82     }
83     return currNode->code;
84 }
85 
86 TrieNode* Trie::CheckWord(uint8_t sym) {
87     if (currentNode) {
```

```
88          if (currentNode->path.count(sym)) {
89              currentNode = currentNode->path[sym];
90          } else {
91              currentNode = nullptr;
92          }
93      } else {
94          currentNode = this->path[sym];
95      }
96      return currentNode;
97  }
98
99  void Trie::AddWord(std::string & str) {
100     TrieNode* currNode = path[(uint8_t)str[0]];
101     for (int i = 1; i < str.length() - 1; ++i)  {
102         currNode = currNode->path[(uint8_t)str[i]];
103     }
104     currNode->path[(uint8_t)str[str.length() - 1]] = new TrieNode(maxCode);
105     if (maxCode == nextIncrease) {
106         nextIncrease *= 2;
107         ++codeLen;
108     }
109     ++maxCode;
110 }
111
112 Trie::~Trie() {
113     for (TrieNode* node : path){
114         delete node;
115     }
116 }
117
118 class LZW{
119 private:
120     uint64_t buffer = 0;
121     uint32_t bufferCount = 0;
122     void WriteBits(std::ostream& file);
123     void AddToBuffer(uint32_t code, uint32_t len, std::ostream& file);
124
125     void ReadBits(uint32_t len,std::istream& file);
126     void Flush();
127     bool Empty(std::istream& file);
128     uint32_t GetBits(uint32_t len, std::istream& file);
129     int32_t GetCodeLen(const uint32_t maxCode);
130
131     friend Trie;
132 public:
133     void EncodeLZW(std::istream& inputData, std::ostream& outputData, uint32_t
        maxCodeLen);
134     int DecodeLZW(std::istream& encodedData, std::ostream& decodedData);
135 };
```

```
136 |
137 | void LZW::WriteBits(std::ostream& file) {
138 |     uint8_t byte = 0;
139 |     while (bufferCount >= 8) {      //8 bits from begin of buffer
140 |         bufferCount -= 8;
141 |         byte = (uint8_t) (buffer >> (bufferCount));     //delete first 8 bits stored
142 |         if (bufferCount == 0) {
143 |             buffer = 0;
144 |         }
145 |         else {
146 |             buffer = buffer << (64 - bufferCount);
147 |             buffer = buffer >> (64 - bufferCount);
148 |         }
149 |         file.write((char*)&byte, 1);
150 |     }
151 | }
152 |
153 | void LZW::AddToBuffer(uint32_t code, uint32_t len, std::ostream& file) {
154 |     buffer = (buffer << len) | (uint64_t) code;
155 |     bufferCount += len;
156 |     if (bufferCount >= 24)
157 |         WriteBits(file);
158 | }
159 |
160 | void LZW::ReadBits(uint32_t len, std::istream& file) {
161 |     uint8_t byte = 0;
162 |     while (bufferCount < len) {
163 |         byte = 0;
164 |         file.read((char*)&byte, 1);
165 |         buffer = (buffer << 8) | (uint64_t)(byte);
166 |         bufferCount += 8;
167 |         file.peek();            //to trigger eof
168 |         if (file.eof()) {
169 |             return;
170 |         }
171 |     }
172 | }
173 |
174 | void LZW::Flush() {
175 |     buffer = 0;
176 |     bufferCount = 0;
177 | }
178 |
179 | bool LZW::Empty(std::istream& file) {
180 |     file.peek();
181 |     return (bufferCount == 0) && file.eof();
182 | }
183 |
184 | uint32_t LZW::GetBits(uint32_t len, std::istream& file) {
```

```
185     uint32_t symbol = 0;
186     if (bufferCount < len){
187         ReadBits(len, file);
188     }
189     if (bufferCount < len) {
190         //input data ended
191         //return what have
192         uint32_t a = buffer;
193         file.peek();
194         buffer = 0;
195         bufferCount = 0;
196         return a;
197     }
198     //get len bits from begin of bufferCount
199     bufferCount -= len;
200     symbol = (uint32_t)(buffer >> bufferCount);
201     //delete first len bits stored
202     if (bufferCount == 0) {
203         symbol = buffer;
204         buffer = 0;
205     } else {
206         buffer = buffer << (64 - bufferCount);
207         buffer = buffer >> (64 - bufferCount);
208     }
209     return symbol;
210 }
211
212 int32_t LZW::GetCodeLen(const uint32_t maxCode) {
213     return (uint32_t)ceil(log2(maxCode));
214 }
215
216 void LZW::EncodeLZW(std::istream& inputData, std::ostream& outputData, uint32_t
        maxCodeLen) {
217     Trie trie;
218     uint32_t exitCode = 0;
219     std::string prev;
220     std::string full;
221     uint8_t curr;
222     //write in file maxCodeLen
223     AddToBuffer(maxCodeLen, 32, outputData);
224     inputData.peek();       //check for eof
225     while (!inputData.eof()) {
226         inputData.read((char*)&curr, 1);    //read symbol
227         full += curr;
228         if (trie.CheckWord(curr)) {
229             prev += curr;
230         } else {
231             AddToBuffer(trie.GetCode(prev), trie.codeLen, outputData);
232             trie.AddWord(full);
```

```
            if (trie.codeLen > maxCodeLen) {    //reset the dictionary
                trie.Clear();
            }
            prev = curr;
            full = curr;
            trie.CheckWord(curr);
        }
        inputData.peek();
    }
    if (!(prev.empty())) {
        AddToBuffer(trie.GetCode(prev), trie.codeLen, outputData);
    }
    AddToBuffer(exitCode, trie.codeLen, outputData);
    Flush();
}

int LZW::DecodeLZW(std::istream& encodedData, std::ostream& decodedData) {
    uint32_t maxCodeLen = GetBits(32, encodedData);
    if (!(maxCodeLen == 12 || maxCodeLen == 16)) {
        return 1;
    }
    std::vector<std::string> dict;
    dict.push_back("");
    uint8_t sym = 0x0;
    for (uint32_t i = 0; i <= 0xff;  ++sym, ++i) {   //conversion from char to string
        dict.push_back(std::string(1, sym));
    }

    std::string currDecode;
    uint32_t exitCode = 0;

    uint32_t code = GetBits(GetCodeLen(dict.size()), encodedData);
    std::string prev = dict[code];  //first code separatly
    std::string full = prev;
    while (!Empty(encodedData)) {
        code = GetBits(GetCodeLen(dict.size() + 1), encodedData);
        if (code == exitCode) {
            break;
        }
        if (code == dict.size()) {
            //code that is not in dictionary
            //previous decoded string + its first letter
            currDecode = prev[0];
            full += currDecode;
            dict.push_back(full);
            decodedData.write(prev.data(), prev.length());
            prev = dict.back();
            full = prev;
        } else {
```

```
            currDecode = dict[code];
            full += currDecode[0];
            dict.push_back(full);
            decodedData.write(prev.data(), prev.length());
            prev = currDecode;
            full = prev;

        }
        //now one more than currently in dictionary
        if (GetCodeLen(dict.size() + 1) > maxCodeLen) {
            decodedData.write(prev.data(), prev.length());
            dict.resize(0xff + 2);
            prev = dict[GetBits(GetCodeLen(dict.size() + 1),  encodedData)];
            full = prev;
        }
    }
    decodedData.write(prev.data(), prev.length());
    return 0;
}
```

arifm.cpp В заголовочном файле arifm.hpp реализован класс ARIFM для арифметического кодирования и декодирования.

```
#pragma once
#include <math.h>
#include <iostream>
#include <fstream>
#include <vector>

class ARIFM {
protected:
    //how many bits will be used for encoding
    const unsigned int codeBits = 32;
    const unsigned int noOfChar = 256;
    const unsigned int EOF_sybol = noOfChar + 1;
    const unsigned int noOfSymbols = noOfChar + 1;

    // swap zone for updating tables freq and cum-freq
    std::vector<unsigned long> charToIndex = std::vector<unsigned long>(noOfChar);
    std::vector<unsigned char> indexToChar = std::vector<unsigned char>(noOfSymbols +
    1);

    // main tables of probability
    std::vector<unsigned long> frequency = std::vector<unsigned long>(noOfSymbols + 1)
    ;
    std::vector<unsigned long> cumFrequency = std::vector<unsigned long>(noOfSymbols +
     1);

    // input-output for Encode
    std::vector<unsigned char> inputEncode;
```

```
25
26      // input-output for Decode
27      std::vector<unsigned short int> inputDecode;
28      std::vector<unsigned char> outputDecode;
29
30      unsigned long bufferSize;
31
32      unsigned long l; // left bound
33      unsigned long h; // right bound
34
35      unsigned long firstQrt;
36      unsigned long half;
37      unsigned long thirdQtr;
38
39      unsigned long maxFrequency;
40
41      void CodeSymbol(unsigned long input_ch, std::ostream& outputData);
42      unsigned long long DecodeSymbol(std::istream& inputData);
43
44      void UpdateModel(unsigned long input_ch);
45
46      void BitPlusFollow(long bit, std::ostream& outputData);
47
48      unsigned short int buffer = 0; // to store bits
49      unsigned long long buffercounter = 0; // for next values to dtore in buffer
50
51      int input_bit(std::istream& inputData);
52      void output_bit(int bit, std::ostream& outputData);
53
54      unsigned long long bitsToGo;
55      unsigned long long ullbufferbits;
56      unsigned long long bitsToFollow = 0; // to form output bits
57      unsigned long long garbageBits = 0;
58      unsigned long value = 0;
59
60  public:
61      ARIFM();
62      void EncodeArifm(std::istream& inputData, std::ostream& outputData);
63      void DecodeArifm(std::istream& inputData, std::ostream& outputData);
64
65  };
66
67  ARIFM::ARIFM() {
68      bufferSize = 60000;
69      inputEncode.resize(bufferSize);
70      inputDecode.resize(bufferSize);
71      l = 0;
72      h = (1UL << codeBits) - 1; //4294967295 if 32   // 2**N - 1
73      maxFrequency = (1UL << (codeBits-2)) - 1;
```

```
74      firstQrt = (h/4 + 1);
75      half = 2*firstQrt;
76      thirdQtr = 3*firstQrt;
77      // tables of recoding symbols
78      for (unsigned int i = 0; i < noOfChar; i++) {
79          charToIndex[i] = i+1;
80          indexToChar[i+1] = i;
81      }
82      int cf = noOfSymbols;
83      for(unsigned int i = 0; i <= noOfSymbols; i++) {
84         cumFrequency[i] = cf--;
85         frequency[i] = 1;
86      }
87     frequency[0]=0;
88  }
89
90  void ARIFM::EncodeArifm(std::istream& inputData, std::ostream& outputData) {
91      bitsToGo = 16;
92      buffer = 0;
93      bitsToFollow = 0;
94      while(inputData) {
95          (inputData).read(reinterpret_cast<char*>(&inputEncode[0]), bufferSize*sizeof(
        unsigned char));
96          unsigned long long readSize = (inputData).gcount();
97          inputEncode.resize(readSize);
98          if (readSize) {
99              for(unsigned long i = 0; i < inputEncode.size(); i++) {
100                 unsigned long input_ch = charToIndex[inputEncode[i]];
101                 CodeSymbol(input_ch, outputData);
102                 UpdateModel(input_ch);
103             }
104         }
105     }
106     CodeSymbol(EOF_sybol, outputData);
107     // done encoding
108     bitsToFollow++;
109     if (l < firstQrt) { BitPlusFollow(0, outputData); }
110     else { BitPlusFollow(1, outputData); }
111     // flush buffer
112     buffer = buffer >> bitsToGo;
113     (outputData).write(reinterpret_cast<char*>(&buffer), sizeof(unsigned short int));
114 }
115
116 void ARIFM::CodeSymbol(unsigned long input_ch, std::ostream& outputData) {
117     unsigned long tmp = l;
118     l = tmp + (((h-tmp+1)*cumFrequency[input_ch])/cumFrequency[0]);
119     h = tmp + (((h-tmp+1)*cumFrequency[input_ch-1])/cumFrequency[0]) - 1;
120
121     for(;;) {
```

```
122         if (h < half) {
123             BitPlusFollow(0, outputData);
124         } else if (l >= half) {
125             BitPlusFollow(1, outputData);
126             l -= half;
127             h -= half;
128         } else if (l >= firstQrt && h < thirdQtr) {
129             bitsToFollow += 1;
130             l -= firstQrt;
131             h -= firstQrt;
132         } else
133             break;
134         l = l*2;
135         h = h*2+1;
136     }
137 }
138 
139 void ARIFM::UpdateModel(unsigned long symbol) {
140     // updating model
141     int index = 0;
142     if (cumFrequency[0] >= maxFrequency) {
143         int cum = 0;
144         for (index = noOfSymbols; index >= 0; index--) {
145             frequency[index] = (frequency[index]+1)/2;
146             cumFrequency[index] = cum;
147             cum += frequency[index];
148         }
149     }
150     // finding suitable index
151     for (index = symbol; frequency[index]==frequency[index-1]; index--);
152 
153     if ((unsigned int)index < symbol) {
154         unsigned long ch_i = indexToChar[index];
155         unsigned long ch_symbol = indexToChar[symbol];
156         indexToChar[index] = ch_symbol;
157         indexToChar[symbol] = ch_i;
158         charToIndex[ch_i] = symbol;
159         charToIndex[ch_symbol] = index;
160     }
161     // updating tables
162     frequency[index]++;
163     while(index > 0) {
164         index -= 1;
165         cumFrequency[index] += 1;
166     }
167 }
168 
169 
170 void ARIFM::DecodeArifm(std::istream& inputData, std::ostream& outputData) {
```

```
171     bitsToGo = 0;
172     garbageBits = 0;
173     value = 0;
174     while(inputData) {
175         (inputData).read(reinterpret_cast<char*>(&inputDecode[0]), bufferSize*sizeof(
        unsigned short int));
176         unsigned long long readSize = (inputData).gcount();
177         // divide by 2, unsigned short 2 bytes
178         inputDecode.resize(readSize/2);
179         buffercounter = 0;
180 
181         value = 0;
182         for (unsigned int i = 0; i < codeBits; i++){
183             value = 2*value + input_bit(inputData);
184         }
185         if (readSize) {
186             for(;;) {
187                 int input_ch = DecodeSymbol(inputData);
188                 if ((unsigned int)input_ch == EOF_sybol) break;
189                 unsigned char ch = indexToChar[input_ch];
190                 (outputData).write(reinterpret_cast<char*>(&ch), sizeof(unsigned char)
        );
191                 outputDecode.push_back(ch);
192                 UpdateModel(input_ch);
193             }
194         }
195     }
196 }
197 
198 
199 unsigned long long ARIFM::DecodeSymbol(std::istream& inputData) {
200     unsigned long long cum = (((value - l) + 1) * cumFrequency[0] - 1)/((h - l) + 1);
201     unsigned long long i = 1;
202     while (cumFrequency[i] > cum) i++;
203 
204     unsigned long long tmp = l;
205     l = tmp + ((h - tmp + 1)*cumFrequency[i])/cumFrequency[0];
206     h = tmp + ((h - tmp + 1)*cumFrequency[i-1])/cumFrequency[0] - 1;
207     for(;;) {
208         if (h < half) {
209             // do nothing
210         } else if (l >= half) {
211             value -= half;
212             l -= half;
213             h -= half;
214         } else if (l >= firstQrt && h < thirdQtr) {
215             value -= firstQrt;
216             l -= firstQrt;
217             h -= firstQrt;
```

```
218         } else break;
219         l = 2*l;
220         h = 2*h+1;
221         value = 2*value + input_bit(inputData);
222     }
223     return i;
224 }
225 
226 int ARIFM::input_bit(std::istream& inputData) {
227     int t; // to return
228     if (bitsToGo == 0) {
229         if (buffercounter == bufferSize){
230             (inputData).read(reinterpret_cast<char*>(&inputDecode[0]), bufferSize*
    sizeof(unsigned short int));
231             unsigned long long readSize = (inputData).gcount();
232             // divide by 2, unsigned short 2 bytes
233             inputDecode.resize(readSize/2);
234             buffercounter = 0;
235         }
236         buffer = inputDecode[buffercounter++];
237 
238         if (static_cast<long long>(buffer) == EOF) {
239             garbageBits += 1;
240             if (garbageBits > codeBits - 2) {
241                 return 0;
242             }
243         }
244         bitsToGo = 16;
245     }
246     t = buffer & 1;
247     buffer >>= 1;
248     bitsToGo -= 1;
249     return t;
250 }
251 
252 void ARIFM::BitPlusFollow(long bit, std::ostream& outputData) {
253     output_bit(bit, outputData);
254     while(bitsToFollow > 0) {
255         output_bit(!bit, outputData);
256         bitsToFollow--;
257     }
258 }
259 
260 void ARIFM::output_bit(int bit, std::ostream& outputData) {
261     buffer >>= 1;
262     if(bit){ buffer |= 0x8000; }
263     bitsToGo -= 1;
264     if (bitsToGo == 0) {
265         (outputData).write(reinterpret_cast<char*>(&buffer), sizeof(unsigned short int
```

```
    ));
266         bitsToGo = 16;
267     }
268 }
```

main.cpp В заголовочном файле arifm.hpp реализован класс ARIFM для арифметического кодирования и декодирования.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <vector>
5  #include <experimental/filesystem>
6  #include <algorithm>
7  #include <iterator>
8
9  #include "lzw.hpp"
10 #include "arifm.hpp"
11
12 int Compress(std::string& filename, bool fromStdin, bool toStdout, bool keepFiles,
       bool fastest) {
13     std::ifstream inputData;
14     std::ofstream outputTmp;
15     std::ifstream inputTmp;
16     std::ofstream outputData;
17
18     uint32_t maxCodeLen;
19     if (fastest) {
20         maxCodeLen = 16;
21     } else {
22         maxCodeLen = 10;
23     }
24     std::string tmpname = filename + "~.tmp";   //temporary file between LZW и Arifm
25     outputTmp.open(tmpname, std::ofstream::out | std::ofstream::binary);
26     if (!(outputTmp.is_open() && outputTmp.good())) {
27         std::cerr << "Error creating temporary file" << std::endl;
28         outputTmp.close();
29         return 1;
30     }
31
32     if (fromStdin) {     //read in stdin
33         std::istream& inputStd = std::cin;
34         LZW lzw;
35         lzw.EncodeLZW(inputStd, outputTmp, maxCodeLen);
36         outputTmp.close();
37     } else {            //read in file
38         inputData.open(filename, std::ifstream::in | std::ifstream::binary);
39         if (!(inputData.is_open())) {
40             std::cerr << "Error opening file " << filename << std::endl;
41             inputData.close();
```

```
42              return 1;
43          }
44          LZW lzw;
45          lzw.EncodeLZW(inputData, outputTmp, maxCodeLen);     //encode by LZW in a
    temporary file ~.tmp
46          outputTmp.close();
47          inputData.close();
48      }
49
50      inputTmp.open(tmpname, std::ifstream::in | std::ifstream::binary); //open
    temporary file ~.tmp
51      if (!(inputTmp.is_open() )) {
52          std::cerr << "Error while opening temporary file" << std::endl;
53          inputTmp.close();
54          return 1;
55      }
56      if (toStdout) {     //if was key -c, using stdout
57          std::ostream& outputStd = std::cout;
58          uint8_t byte = 0xee;
59          outputStd.write((char*)&byte, 1);
60
61          ARIFM arifm;
62          arifm.EncodeArifm(inputTmp, outputStd);
63
64          inputTmp.close();
65      } else {            //write in file
66          outputData.open(filename + ".Z", std::ofstream::out | std::ofstream::binary);
67          if (!(outputData.is_open() && outputData.good())) {
68              std::cerr << "Error creating archive file" << std::endl;
69              outputData.close();
70              return 1;
71          }
72          uint8_t specialByte = 0xee;  //first byte writing 238
73          outputData.write((char*)&specialByte, 1);
74
75          ARIFM arifm;
76          arifm.EncodeArifm(inputTmp, outputData);
77
78          //write down uncompressed size
79          outputData.write((char*)&specialByte, 1);
80          uint64_t uncompressedSize = std::experimental::filesystem::file_size(filename)
    ;
81          outputData.write((char*)&uncompressedSize, 8);
82
83          inputTmp.close();
84          outputData.close();
85      }
86      if (remove(tmpname.data())) {   //delete temporary file name~.tmp
87          std::cerr << "Error while removing temporary file" << std::endl;
```

```
88          return 1;
89      }
90      if (!keepFiles && !fromStdin) {     //delete sorce, if this need
91          if (remove(filename.data())) {
92              std::cerr << "Error while removing file " << filename << std::endl;
93              return 1;
94          }
95      }
96
97      return 0;
98  }
99  int Decompress(std::string& filename, bool fromStdin, bool toStdout, bool keepFiles) {
100     std::ifstream inputData;
101     std::ofstream outputTmp;
102     std::ifstream inputTmp;
103     std::ofstream outputData;
104     uint32_t ret = 0;       //error control
105
106     std::string tmpname = filename + "~.tmp";   //temporary file between LZW и Arifm
107     std::string unpackedname = "";
108     outputTmp.open(tmpname, std::ofstream::out | std::ofstream::binary);
109     if (!(outputTmp.is_open() && outputTmp.good())) {
110         std::cerr << "Error while creating temporary file" << std::endl;
111         outputTmp.close();
112         return 1;
113     }
114
115     if (fromStdin) { //read from stdin
116         std::istream& inputStd = std::cin;
117         uint8_t specialByte = 0xee;
118         uint8_t byte;
119         inputData.read((char*)&byte, 1);
120         if (byte == specialByte) {      //if reading byte 238, data is good
121             ARIFM arifm;
122             arifm.DecodeArifm(inputStd, outputTmp);
123         }
124         else {
125             ret = 1;        //else error
126         }
127         outputTmp.close();
128
129     } else {    //read from file
130         unpackedname = filename.substr(0, filename.length() - 2);
131         if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z") {
132             std::cerr << "File does not have extension .Z:" << filename << std::endl;
133             return 1;
134         }
135         inputData.open(filename, std::ifstream::in | std::ifstream::binary);
136         if (!(inputData.is_open())) {
```

```
137             std::cerr << "Error while opening archive file " << filename << std::endl;
138             inputData.close();
139             return 1;
140         }
141         uint8_t specialByte = 0xee;
142         uint8_t byte;
143         inputData.clear();
144         inputData.read((char*)&byte, 1);
145         if (byte == specialByte) {      //if reading byte 238, data is good
146             ARIFM arifm;
147             arifm.DecodeArifm(inputData, outputTmp);
148         }
149         else {
150             ret = 1;
151         }
152         inputData.close();
153         outputTmp.close();
154     }
155     if (ret) {
156         return ret;
157     }
158     //декодирование LZW
159     inputTmp.open(tmpname, std::ifstream::in | std::ifstream::binary);
160     if (!(inputTmp.is_open() )) {
161         std::cerr << "Error while opening temporary file" << std::endl;
162         inputTmp.close();
163         return 1;
164     }
165     if (toStdout) {
166         std::ostream& outputStd = std::cout;
167         LZW lzw;
168         ret = lzw.DecodeLZW(inputTmp, outputStd);
169         inputTmp.close();
170     } else {
171         outputData.open(unpackedname , std::ofstream::out | std::ofstream::binary);
172         if (!(outputData.is_open() && outputData.good())) {
173             std::cerr << "Error creating file: " << unpackedname << std::endl;
174             outputData.close();
175             return 1;
176         }
177         LZW lzw;
178         ret = lzw.DecodeLZW(inputTmp, outputData);
179         outputData.close();
180         inputTmp.close();
181     }
182 
183     if (remove(tmpname.data())) {
184         std::cerr << "Error removing temporary file" << std::endl;
185         return 1;
```

```
186     }
187
188     if (!keepFiles && !fromStdin) {       //delete sorce, if this need
189         if (remove(filename.data())) {
190             std::cerr << "Error removing file " << filename << std::endl;
191             return 1;
192         }
193     }
194
195     if (ret) {
196         std::cerr << "Wrong archive structure:" << std::endl;
197     }
198     return ret;
199 }
200
201 void checkIntegrity(std::string& filename) {
202     //there is special byte 0xee in the beggining
203     std::ifstream inputData;
204     if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z") {
205         std::cerr << filename << " is not a .Z archive" << std::endl;
206         return;
207     }
208     inputData.open(filename, std::ifstream::in | std::ifstream::binary);
209     uint8_t byte;
210     bool corruption = false;
211     inputData.read((char*)&byte, 1);
212     if (byte != 0xee)
213         corruption = true;
214     else {
215         inputData.seekg(-9, inputData.end);
216         inputData.read((char*)&byte, 1);
217         if (byte != 0xee)
218             corruption = true;
219     }
220
221     inputData.close();
222     if (corruption)
223         std::cout << "Archive file " << filename << " corrupted" << std::endl;
224     else
225         std::cout << "Archive file " << filename << " is consistent" << std::endl;
226 }
227
228 void getInfo(std::string& filename) {
229
230     uint64_t uncompressedSize = 0, compressedSize;
231     std::string unpackedname = filename.substr(0, filename.length() - 2);
232     if (filename.substr(filename.length() - 2, filename.length() - 1) != ".Z") {
233         std::cerr << filename << " is not a .Z archive" << std::endl;
234     }
```

```
235     else {
236         std::cout << filename << ":" << std::endl;
237         compressedSize = std::experimental::filesystem::file_size(filename);
238         std::cout << "\tcompressed " << compressedSize << std::endl;
239 
240         std::ifstream inputData;
241         inputData.open(filename, std::ifstream::in | std::ifstream::binary);
242         if (!(inputData.is_open())) {
243             std::cerr << "Error while opening file " << filename << std::endl;
244             inputData.close();
245         }
246 
247         inputData.seekg(-8, inputData.end);
248         inputData.read((char*)&uncompressedSize, 8);
249         inputData.close();
250         std::cout << "\tuncompressed " << uncompressedSize << std::endl;
251         std::cout << "\tuncompressed_name " << unpackedname << std::endl;
252     }
253 }
254 int main(int argc, char const * argv[]) {
255     bool flagWriteToStdout = false;
256     bool flagDecompress = false;
257     bool flagKeepFiles = false;
258     bool flagListProperties = false;
259     bool flagRecursive = false;
260     bool flagTestIntegrity = false;
261     bool flagFastest = true; //default -9
262 
263     bool flagCompress = false;
264     bool flagReadFromStdin = false;
265     std::string filename;
266     for (int i = 1; i <= argc - 1; ++i)  {
267         std::string arg = argv[i];
268         if (arg == "-c") {
269             flagWriteToStdout = true;
270         } else if (arg == "-d") {
271             flagDecompress = true;
272         } else if (arg == "-k") {
273             flagKeepFiles = true;
274         } else if (arg == "-l") {
275             flagListProperties = true;
276         } else if (arg == "-r") {
277             flagRecursive = true;
278         } else if (arg == "-t") {
279             flagTestIntegrity = true;
280         } else if (arg == "-1") {
281             flagFastest = true;
282         } else if (arg == "-9") {
283             flagFastest = false;
```

```
284         }
285         else{
286             filename = arg;
287         }
288     }
289     if ( filename == "-" ) {
290         flagWriteToStdout = true;
291         flagReadFromStdin = true;
292     }
293 
294     if (!flagTestIntegrity && !flagDecompress && !flagListProperties)
295         flagCompress = true;
296 
297     if (flagRecursive) {
298         std::experimental::filesystem::recursive_directory_iterator dir(filename), end
    ;
299         std::vector<std::string> files;
300         while (dir != end) {
301             if (std::experimental::filesystem::is_regular_file(dir->path())) {
302                 files.push_back(dir->path().c_str());
303             }
304             ++dir;
305         }
306         if (flagCompress) {
307             for (std::string& file : files)
308                 if (Compress(file, false, false, flagKeepFiles, flagFastest))
309                     return 1;
310         }
311         else if (flagDecompress) {
312             for (std::string& file : files)
313                 if (Decompress(file, false, false, flagKeepFiles))
314                     return 1;
315         } else if (flagListProperties && !flagReadFromStdin) {
316             getInfo(filename);
317         } else if (flagTestIntegrity) {
318             checkIntegrity(filename);
319         }
320         return 0;
321     }
322     if (flagCompress) {
323         return Compress(filename, flagReadFromStdin, flagWriteToStdout, flagKeepFiles
    , flagFastest);
324     } else if (flagDecompress) {
325         return Decompress(filename, flagReadFromStdin, flagWriteToStdout,
    flagKeepFiles);
326 
327     } else if (flagListProperties && !flagReadFromStdin) {
328         getInfo(filename);
329     } else if (flagTestIntegrity) {
```

```
330         checkIntegrity(filename);
331     }
332     return 0;
333 }
```

5 Вывод

При работе над курсовой я подробно изучила особенности использованных алгоритмов сжатия. Для меня оказалось сложным реализация и отладка кодирования в бинарном представлении, при этом написание базового варианта курсовой не составило сложностей. Однако мне помогло то, что эти алгоритмы уже описаны в известных источниках. Также немалую часть курсовой занимает обработка ключей архиватора. Их изучение и реализация были достаточно интересными, для этого также пришлось посмотреть принципы работы архиватора GZIP.

Список литературы

[1] Witten Ian H., Neal Radford M., Cleary John G. *Arithmetic coding for data compression* — Communications of the ACM.- June 1987.- Vol. 30.-№6.

[2] Сайт по методам сжатия данных, изображений и видео *https://www.compression.ru/compression.ru/* (дата обращения 25.05.21)

[3] Мастрюков Д. *Алгоритмы сжатия информации. Часть 3. Алгоритмы группы LZ* — Монитор, N2, 1994. С10-13.