Московский авиационный институт
(национальный исследовательский
университет)

Факультет информационных технологий и
прикладной математики

Кафедра вычислительной математики
и программирования

Курсовой проект по курсу «Дискретный анализ»

Разработка архиватора (Huffman + LZ77)

Студент: Фирфаров А.С.
Преподаватель: Журавлев А.А.
Группа: М8О-208Б
Дата:
Оценка:
Подпись:

Москва, 2019

Курсовой проект

Необходимо спроектировать и реализовать архиватор, использующий заданные методы сжатия данных для сжатия одного файла.

Формат запуска должен быть аналогичен формату запуска программы gzip, должны быть поддержаны следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Теоретическая часть

LZ77 - алгоритм сжатия без потерь, опубликованный в статье израильских математиков Авраама Лемпеля и Яакова Зива в 1977 году. Многие программы сжатия информации используют ту или иную модификацию этого алгоритма. LZ77 относится к словарным алгоритмам сжатия, в котором словарь формируется на основании уже обработанной части входного потока. Алгоритм использует скользящее окно, разделенное на две неравные части. Первая, большая по размеру, включает уже просмотренную часть сообщения. Вторая, намного меньшая, является буфером, содержащим еще незакодированные символы входного потока. Идея алгоритма заключается в поиске самого длинного совпадения между строкой буфера и всеми фразами словаря. Полученная в результате поиска фраза кодируется с помощью двух чисел и одного символа: смещения от начала буфера(offset), длины совпадения(length) и символа, следующего за совпавшей строкой буфера. Затем окно смещается на length + 1 символ вправо и алгоритм повторяется. Алгоритм имеет следующие недостатки: невозможность кодирования подстрок, стоящих друг от друга на расстоянии, большем длины словаря, малая эффективность при кодировании незначительных объёмов данных и ограничнность длины подстроки, которую можно закодировать.

Алгоритм Хаффмана — жадный алгоритм оптимального префиксного кодирования. Алгоритм был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом. Идея алгоритма заключается в замене символов на соответствующие им коды. Часто встречающимся символам будут соответствовать меньшие коды, редко встречающиеся символы получат более длинные коды. Это нужно для того, чтобы самые частотные символы занимали как можно меньше места. Ни один из кодов не является префиксом другого, что позволяет их однозначно интерпретировать. Алгоритм хорошо дополняет LZ77, позволяя улучшить степень сжатия.

Реализация

На первом этапе текст сжимается с помощью алгоритма LZ77. Ищется совпадение между словарем и буфером, и найденное самое длинное совпадение записывается во временный файл в виде упорядоченной тройки <смещение(offset), длина совпадения(length), следующий символ(nextChar)>. На каждой итерации алгоритма совпадение между словарем и буфером ищется с помощью Z-функции. Формируется вспомогательная строка вида "буфер"+ "символ разделитель"+ "словарь"+ "буфер". Z - функция для этой строки вычисляется за линейное время. Таким образом можно найти длину наибольшего совпадения и его позицию в словаре. По мере записи закодированных последовательностей во временных файл, собирается статистика "символов" для полустатического алгоритма Хаффмана. Это позволит не делать лишний проход по сжатому алгоритмом LZ77 тексту. Степень сжатия зависит от максимального размера словаря. Большой словарь позволяет найти большее совпадение, но за значительное время. Меньший словарь позволяет тратить меньше времени на поиск совпадений, но качество сжатия будет хуже.

Рис. 1: Зависимость времени сжатия от размера словаря - файл 80 MB

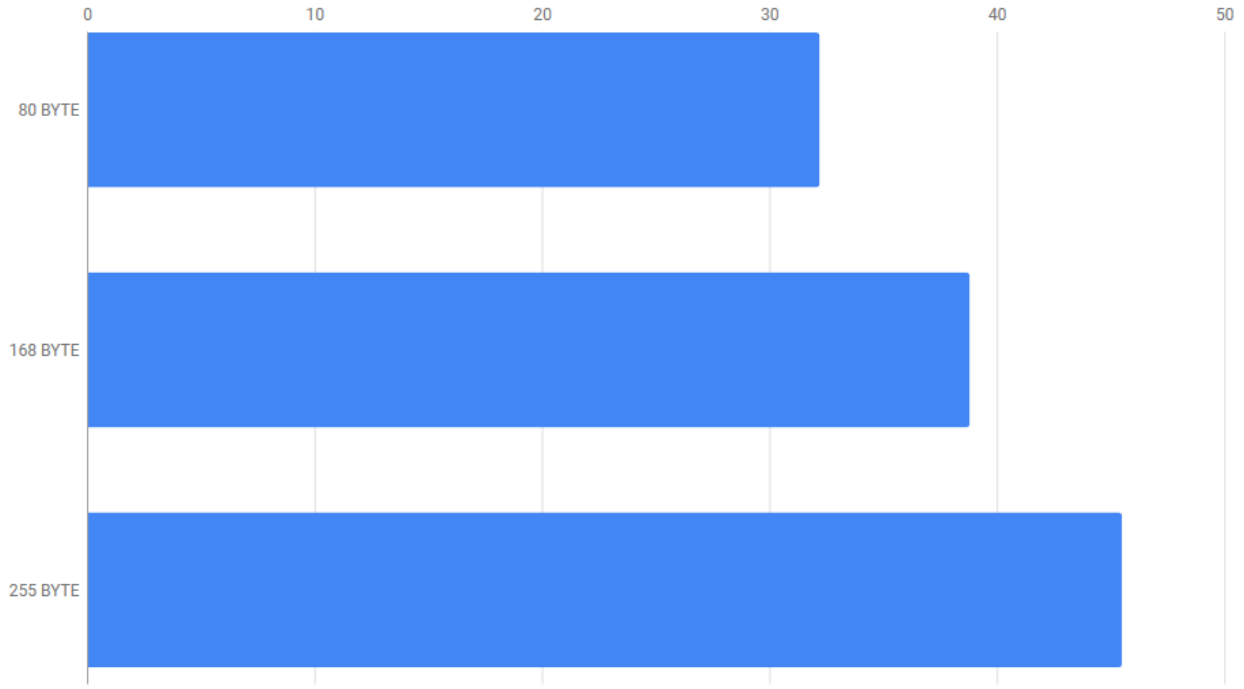

Рис. 2: Зависимость степени сжатия (%) от размера словаря - файл 80 MB

Далее полученный временный файл сжимается полустатическим алгоритмом Хаффмана. Статистика частот для каждого байта была собрана на предыдущем шаге, поэтому нет необходимости просматривать временный файл для этого. На основании этой статистики строится дерево кодов Хаффмана. При построении используется очередь с приоритетами. Время, затраченное на построение дерева, несущественно по сравнению с временем дальнейшего кодирования. Для разархивации необходимо записать в конечный файл сереализованное дерево кодов Хаффмана. Далее каждый байт заменяется на соответствующий ему код и записывается в выходной файл.

Для разархивации дерево кодов Хаффмана нужно десереализовать и заменить коды из архива на соответствующие им байты. Далее к полученному временному файлу применяется алгоритм декодирования LZ77. В результате получается исходный несжатый файл.

Сравнение с аналогами

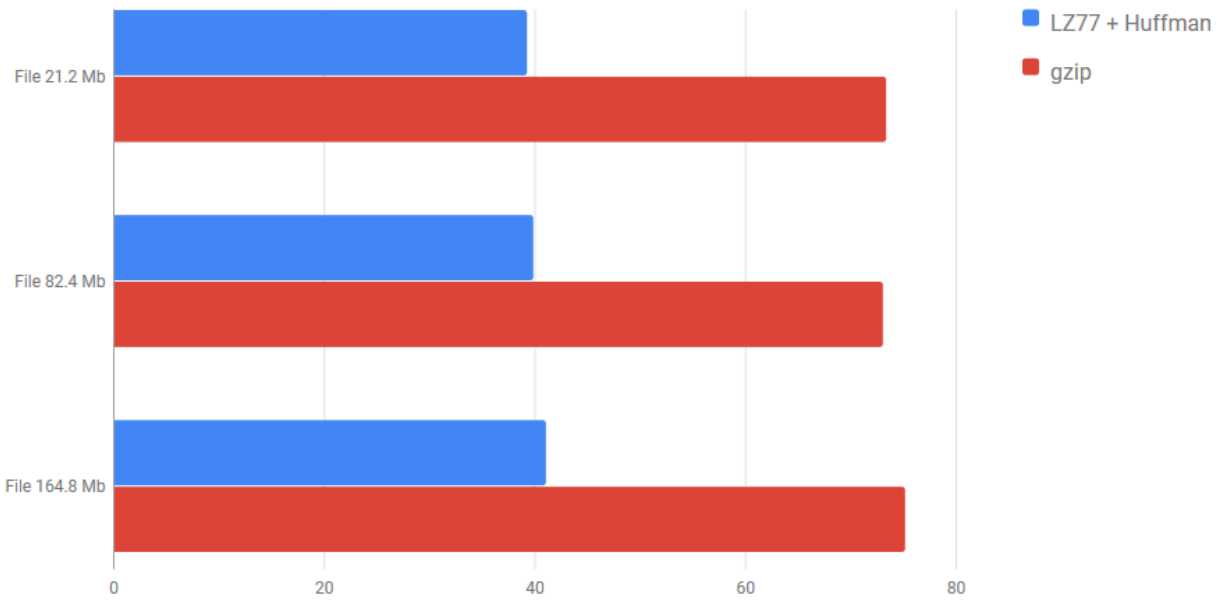

Рис. 3: Сравнение степени сжатия

Видно, что архиватор проигрывает по степени сжатия архиватору gzip. Можно добиться схожей степени сжатия, существенно увеличив размер буфера и словаря, но это также сильно увеличит время сжатия. Стоит отметить, что при сжатии маленьких файлов размер архива может быть даже больше размера оригинального файла. Это связано с необходимостью хранить дерево кодов Хаффмана и другую дополнительную информацию.

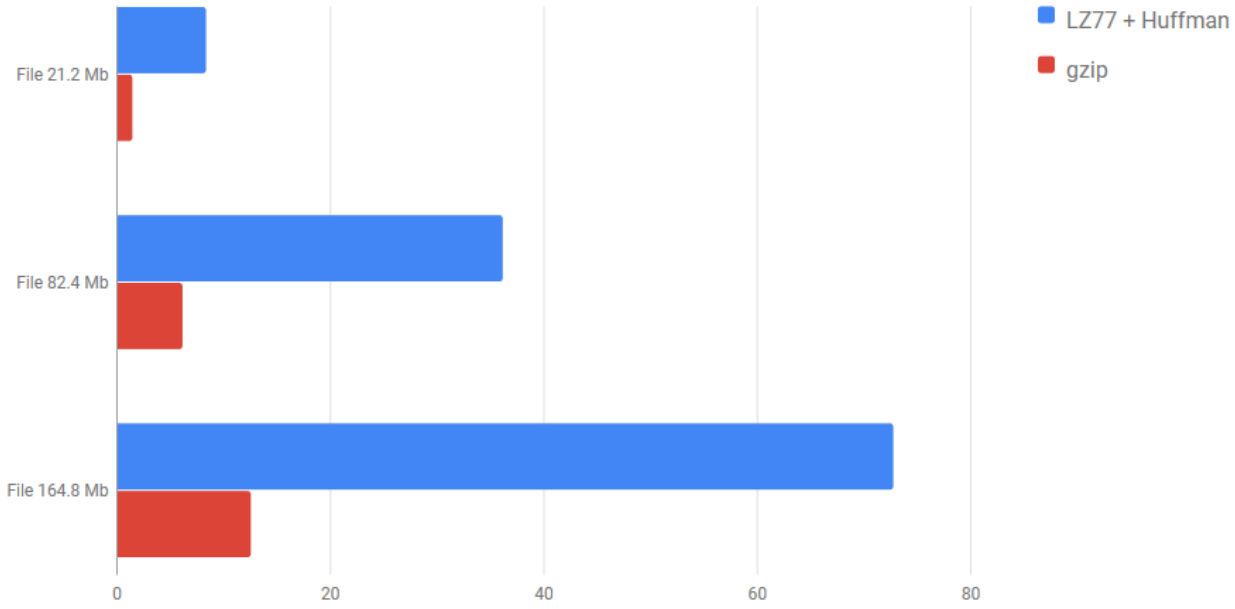

Рис. 4: Сравнение времени сжатия

Исходный код

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <vector>
5  #include <queue>
6  #include <memory>
7  #include <unordered_map>
8  #include <experimental/filesystem>
9
10 #define TEST_NUMBER 1234567890
11 #define MAX_DICT_SIZE 255
12 #define MIN_DICT_SIZE 80
13
14 uint8_t DICT_SIZE = 168;
15 const uint8_t AHEAD_SIZE = 8;
16 const uint8_t BYTE_SIZE = 8;
17
18 struct LZ77Code {
19     uint8_t offset;
20     uint8_t length;
21     char nextChar;
22 };
23
24 struct HuffTreeNode {
25     HuffTreeNode() {};
26     HuffTreeNode(const std::shared_ptr<HuffTreeNode>& _left, const std
           ::shared_ptr<HuffTreeNode>& _right) {
27         left = _left;
28         right = _right;
29         freq = left->freq + right->freq;
30     }
31     HuffTreeNode(int _freq, char _symb) : freq(_freq), symb(_symb), leaf(
           true) {};
32
```

```
33      bool leaf = false;
34      char symb;
35      int freq;
36
37      std::shared_ptr<HuffTreeNode> left = nullptr;
38      std::shared_ptr<HuffTreeNode> right = nullptr;
39  };
40
41  struct CompareNodes {
42      bool operator()(const std::shared_ptr<HuffTreeNode>& node1, const
            std::shared_ptr<HuffTreeNode>& node2) {
43          return node1->freq > node2->freq;
44      }
45  };
46
47  LZ77Code FindMatch(const std::string& data, int startDict, int startAhead,
        int endAhead) {
48
49      std::string s1 = data.substr(startAhead, endAhead - startAhead + 1) +
            char(245);
50      std::string s2 = data.substr(startDict, endAhead - startDict + 1);
51      std::string text = s1 + s2;
52      std::vector<int> z(text.size() - (endAhead - startAhead + 1), 0);
53
54      int left = 0;
55      int right = 0;
56
57      uint8_t offset = 0;
58      uint8_t length = 0;
59      char next = data[startAhead];
60
61      int size = z.size();
62
63      for (int i = 1; i < size; ++i) {
64          if (i <= right) {
65              z[i] = std::min(right - i + 1, z[i - left]);
```

```
66          }
67          while (i + z[i] < size && text[z[i]] == text[i + z[i]]) {
68              ++z[i];
69          }
70
71          if (i > endAhead − startAhead + 1 && z[i] > length) {
72              length = z[i];
73              offset = size − i;
74              next = startAhead + length < data.size() ? data[startAhead +
                    length] : '\0';
75              if (length == endAhead − startAhead + 1) return {offset,
                    length, next};
76          }
77
78          if (i + z[i] − 1 > right) {
79              left = i;
80              right = i + z[i] − 1;
81          }
82      }
83      return {offset, length, next};
84  }
85
86  void LZ77Compress(const std::string& fileName, const std::string& data,
        std::unordered_map<char, int>& stat) {
87
88      std::ofstream output(fileName + ".tmp", std::ios::binary);
89
90      if (data.empty()) {
91          LZ77Code last = {0, 0, '\0'};
92
93          stat[0] += 2;
94          ++stat['\0'];
95
96          output.write((char*)(&last), sizeof(LZ77Code));
97
98          output.close();
```

```
99          return;
100     }
101
102     LZ77Code firstCode = {0, 0, data[0]};
103
104     stat[0] += 2;
105     ++stat[data[0]];
106
107     output.write((char*)(&firstCode), sizeof(LZ77Code));
108
109     if (data.size() == 1) {
110         LZ77Code last = {0, 0, '\0'};
111
112         stat[0] += 2;
113         ++stat['\0'];
114
115         output.write((char*)(&last), sizeof(LZ77Code));
116         output.close();
117         return;
118     }
119
120     int startDict = 0;
121     int startAhead = 1;
122     int endAhead = data.size() - 1 > AHEAD_SIZE ? AHEAD_SIZE :
            data.size() - 1;
123
124     while (true) {
125         LZ77Code code = FindMatch(data, startDict, startAhead, endAhead
                );
126
127         ++stat[code.offset];
128         ++stat[code.length];
129         ++stat[code.nextChar];
130
131         output.write((char*)(&code), sizeof(LZ77Code));
132         if (code.nextChar == '\0') break;
```

```
133
134             startAhead += code.length + 1;
135             startDict = startAhead - DICT_SIZE >= 0 ? startAhead -
                    DICT_SIZE : 0;
136             endAhead = startAhead + AHEAD_SIZE < data.size() ?
                    startAhead + AHEAD_SIZE - 1 : data.size() - 1;
137
138             if (startAhead >= data.size()) {
139                 LZ77Code last = {0, 0, '\0'};
140
141                 stat[0] += 2;
142                 ++stat['\0'];
143
144                 output.write((char*)(&last), sizeof(LZ77Code));
145                 break;
146             }
147         }
148
149         output.close();
150         return;
151     }
152
153     void LZ77Decompress(const std::string& fileName) {
154
155         std::ofstream output(fileName, std::ios::out);
156         std::ifstream input(fileName + ".tmp", std::ios::binary | std::ios::in);
157         std::string text;
158         LZ77Code code;
159
160         while(input) {
161
162             input.read((char*)(&code), sizeof(LZ77Code));
163
164             if (code.length > 0) {
165                 uint64_t start = text.length() - code.offset;
166                 for (int i = 0; i < code.length; ++i) {
```

```
167                     text += text[start + i];
168                 }
169             }
170             if (code.nextChar == '\0') break;
171             text += code.nextChar;
172         }
173         output << text;
174 
175         output.close();
176         input.close();
177 
178         std::string removeFile = fileName + ".tmp";
179         remove(removeFile.c_str());
180 }
181 
182 void CreateCodesCompres(const std::shared_ptr<HuffTreeNode>& node,
        std::string& code, std::unordered_map<char, std::string>& codes) {
183         if (!node->leaf) {
184             code.push_back('0');
185             CreateCodesCompres(node->left, code, codes);
186             code.pop_back();
187             code.push_back('1');
188             CreateCodesCompres(node->right, code, codes);
189             code.pop_back();
190         } else {
191             codes[node->symb] = code;
192         }
193 }
194 
195 void WriteEncodedString(const std::string& inputFileName, const std::
        unordered_map<char, std::string>& codes, std::ofstream& output) {
196 
197         std::ifstream input(inputFileName + ".tmp", std::ios::binary | std::ios::in)
            ;
198 
199         std::queue<int> buffer;
```

```
200     char symb = 0;
201
202     while(input.get(symb)) {
203         std::string code = codes.at(symb);
204         int i = 0;
205         while (i < code.size()) {
206             code[i] == '1' ? buffer.push(1) : buffer.push(0);
207             ++i;
208
209             if (buffer.size() == BYTE_SIZE) {
210                 uint8_t byte = 0;
211                 for (int j = 1; j <= BYTE_SIZE; ++j) {
212                     byte |= buffer.front() << (BYTE_SIZE - j);
213                     buffer.pop();
214                 }
215                 output.write((char*)(&byte), sizeof(uint8_t));
216             }
217         }
218     }
219     if (buffer.size()) {
220         while (buffer.size() < BYTE_SIZE) {
221             buffer.push(0);
222         }
223         uint8_t byte = 0;
224         for (int j = 1; j <= BYTE_SIZE; ++j) {
225             byte |= buffer.front() << (BYTE_SIZE - j);
226             buffer.pop();
227         }
228         output.write((char*)(&byte), sizeof(uint8_t));
229     }
230
231     input.close();
232 }
233
234 void Serialize(const std::shared_ptr<HuffTreeNode>& node, std::ofstream&
        output) {
```

```
235     if (node->leaf) {
236         char leafSign = '1';
237         output.write((char*)(&leafSign), sizeof(char));
238         output.write((char*)(&(node->symb)), sizeof(char));
239 
240         return;
241     }
242 
243     char leafSign = '0';
244     output.write((char*)(&leafSign), sizeof(char));
245     Serialize(node->left, output);
246     Serialize(node->right, output);
247 }
248 
249 void SerializeTree(const std::string& inputFileName, const std::shared_ptr
        <HuffTreeNode>& root, std::ofstream& output) {
250 
251     Serialize(root, output);
252 }
253 
254 
255 std::shared_ptr<HuffTreeNode> DeserializeTree(std::ifstream& input) {
256     char leafSign;
257     input.get(leafSign);
258 
259     if (leafSign == '1') {
260         char symb;
261         input.get(symb);
262 
263         std::shared_ptr<HuffTreeNode> leafNode(new HuffTreeNode(0,
                symb));
264         return leafNode;
265     }
266 
267     std::shared_ptr<HuffTreeNode> node(new HuffTreeNode());
268     node->left = DeserializeTree(input);
```

```
269     node->right = DeserializeTree(input);
270     return node;
271 }
272
273 void HuffmanCompress(const std::string& fileName, const std::
        unordered_map<char, int>& stat, uint64_t fileSize) {
274
275     std::ifstream input(fileName + ".tmp", std::ios::binary | std::ios::in);
276     std::priority_queue<std::shared_ptr<HuffTreeNode>, std::vector<std::
            shared_ptr<HuffTreeNode>>, CompareNodes> q;
277
278     char symb = 0;
279
280     for (const auto& [symb, freq] : stat) {
281         q.push(std::shared_ptr<HuffTreeNode>(new HuffTreeNode(freq,
                symb)));
282     }
283
284     while (q.size() > 1) {
285         std::shared_ptr<HuffTreeNode> left(q.top());
286         q.pop();
287
288         std::shared_ptr<HuffTreeNode> right(q.top());
289         q.pop();
290
291         std::shared_ptr<HuffTreeNode> newParent(new HuffTreeNode(left,
                right));
292         q.push(newParent);
293     }
294
295     std::string code;
296     std::unordered_map<char, std::string> codes;
297
298     CreateCodesCompres(q.top(), code, codes);
299
300     uint8_t extraZeros = 0;
```

```
301     uint64_t bitCount = 0;
302     uint64_t byteCount = 0;
303
304     for (const auto& [symb, code] : codes) {
305         bitCount += code.length() * stat.at(symb);
306     }
307
308     extraZeros = bitCount % BYTE_SIZE == 0 ? 0 : BYTE_SIZE -
            bitCount % BYTE_SIZE;
309     byteCount = extraZeros == 0 ? bitCount / BYTE_SIZE : bitCount /
            BYTE_SIZE + 1;
310
311     input.close();
312     std::ofstream output(fileName + ".Z", std::ios::binary);
313
314     uint64_t testNumber = TEST_NUMBER;
315
316     output.write((char*)(&testNumber), sizeof(uint64_t));
317     output.write((char*)(&fileSize), sizeof(uint64_t));
318     output.write((char*)(&extraZeros), sizeof(uint8_t));
319     output.write((char*)(&byteCount), sizeof(uint64_t));
320
321     SerializeTree(fileName, q.top(), output);
322     WriteEncodedString(fileName, codes, output);
323
324     uint64_t lastPos = output.tellp();
325     output.write((char*)(&lastPos), sizeof(uint64_t));
326     output.close();
327
328     std::string removeFile = fileName + ".tmp";
329     remove(removeFile.c_str());
330 }
331
332 void CreateCodesDecompres(const std::shared_ptr<HuffTreeNode>& node,
        std::string& code, std::unordered_map<std::string, char>& codes) {
333
```

```
334     if (!node->leaf) {
335         code.push_back('0');
336         CreateCodesDecompres(node->left, code, codes);
337         code.pop_back();
338         code.push_back('1');
339         CreateCodesDecompres(node->right, code, codes);
340         code.pop_back();
341     } else {
342         codes[code] = node->symb;
343     }
344 }
345
346 void HuffmanDecompress(const std::string& fileName, std::ifstream& input)
        {
347
348     std::ofstream output(fileName + ".tmp", std::ios::binary);
349
350     uint64_t fileSize = 0;
351     uint64_t testNumber = 0;
352     uint8_t extraZeros = 0;
353     uint64_t byteCount = 0;
354
355     input.read((char*)(&testNumber), sizeof(uint64_t));
356     input.read((char*)(&fileSize), sizeof(uint64_t));
357     input.read((char*)(&extraZeros), sizeof(uint8_t));
358     input.read((char*)(&byteCount), sizeof(uint64_t));
359
360     std::shared_ptr<HuffTreeNode> root = DeserializeTree(input);
361     std::string code;
362     std::unordered_map<std::string, char> codes;
363
364     CreateCodesDecompres(root, code, codes);
365     code.clear();
366
367     for (uint64_t i = 1; i <= byteCount; ++i) {
368         uint8_t byte = 0;
```

```
369             std::string oneByte;
370             input.read((char*)(&byte), sizeof(uint8_t));
371             if (i == byteCount) {
372                 byte = byte >> extraZeros;
373                 for (int j = 0; j < BYTE_SIZE - extraZeros; ++j) {
374                     oneByte = (byte & 1 ? '1' : '0') + oneByte;
375                     byte = byte >> 1;
376                 }
377             } else {
378                 for (int j = 0; j < BYTE_SIZE; ++j) {
379                     oneByte = (byte & 1 ? '1' : '0') + oneByte;
380                     byte = byte >> 1;
381                 }
382             }
383             for (int j = 0; j < oneByte.size(); ++j) {
384                 code += oneByte[j];
385                 if (codes.count(code)) {
386                     output.write((char*)(&codes[code]), sizeof(char));
387                     code.clear();
388                 }
389             }
390         }
391
392         uint64_t lastPos = 0;
393         input.read((char*)(&lastPos), sizeof(uint64_t));
394
395         input.close();
396         output.close();
397
398         return;
399     }
400
401     void Encode(const std::string& fileName, bool keepFile, bool
        readFromStdin, bool writeToStdout) {
402
403         std::string data;
```

```
404     char symb;
405     uint64_t fileSize = 0;
406     std::unordered_map<char, int> stat;
407 
408     if (readFromStdin) {
409         std::istream& input = std::cin;
410         while(input.get(symb)) {
411             data.push_back(symb);
412         }
413     } else {
414         std::ifstream input(fileName, std::ios::in);
415 
416         if (!input) {
417             std::cout << "Файл не существует: " << fileName << std::
                    endl;
418             input.close();
419             return;
420         }
421         fileSize = std::experimental::filesystem::file_size(fileName);
422         data.reserve(fileSize + 1);
423 
424         while(input.get(symb)) {
425             data.push_back(symb);
426         }
427 
428         input.close();
429     }
430 
431     LZ77Compress(fileName, data, stat);
432     HuffmanCompress(fileName, stat, fileSize);
433 
434     if (writeToStdout) {
435         std::ostream& output = std::cout;
436         std::ifstream input(fileName + ".Z", std::ios::in);
437         char symb;
438 
```

```
439             while(input.get(symb)) {
440                 output << symb;
441             }
442             input.close();
443             keepFile = true;
444 
445             std::string removeFile = fileName + ".Z";
446             remove(removeFile.c_str());
447         }
448 
449         if (!keepFile) {
450             remove(fileName.c_str());
451         }
452 }
453 
454 void Decode(const std::string& fileName, bool keepFile, bool
        readFromStdin, bool writeToStdout) {
455 
456         std::string file = fileName;
457 
458         if (readFromStdin) {
459             std::istream& input = std::cin;
460             std::ofstream output(file + ".Z", std::ios::binary);
461             char symb;
462 
463             while(input.get(symb)) {
464                 output.write((char*)(&symb), sizeof(char));
465             }
466             output.close();
467         }
468 
469         if (fileName.size() > 2) {
470             if (fileName.substr(fileName.length() - 2) == ".Z") {
471                 file = fileName.substr(0, fileName.length() - 2);
472             }
473         }
```

```
474
475     std::ifstream input(file + ".Z", std::ios::binary | std::ios::in);
476
477     if (!input) {
478         std::cout << "Архив␣с␣таким␣именем␣не␣существует␣:␣" << file
                + ".Z" << std::endl;
479         input.close();
480         return;
481     }
482
483     HuffmanDecompress(file, input);
484     LZ77Decompress(file);
485
486     if (readFromStdin) {
487         std::string removeFile = file + ".Z";
488         remove(removeFile.c_str());
489     }
490
491     if (writeToStdout) {
492         std::ostream& output = std::cout;
493         std::ifstream input(file, std::ios::in);
494         char symb;
495
496         while(input.get(symb)) {
497             output << symb;
498         }
499         input.close();
500         keepFile = true;
501
502         std::string removeFile = file;
503         remove(removeFile.c_str());
504     }
505
506     if (!keepFile) {
507         std::string removeFile = file + ".Z";
508         remove(removeFile.c_str());
```

```
509     }
510 }
511
512 void ArchiveInformation(const std::string& fileName) {
513
514     std::string file = fileName;
515
516     if (fileName.size() > 2) {
517         if (fileName.substr(fileName.length() - 2) == ".Z") {
518             file = fileName.substr(0, fileName.length() - 2);
519         }
520     }
521
522     std::ifstream input(file + ".Z", std::ios::binary | std::ios::in);
523
524     if (!input) {
525         std::cout << "Архив с таким именем не существует : " << file
                + ".Z" << std::endl;
526         input.close();
527         return;
528     }
529
530     uint64_t uncompSize = 0;
531     uint64_t compSize = std::experimental::filesystem::file_size(file + ".Z");
532     uint64_t testNumber = 0;
533
534     input.read((char*)(&testNumber), sizeof(uint64_t));
535     input.read((char*)(&uncompSize), sizeof(uint64_t));
536     input.close();
537
538     double ratio = uncompSize == 0.0 ? 0.0 : 1.0 - (double)compSize /
            uncompSize;
539     std::cout << "compressed size: " << compSize << std::endl;
540     std::cout << "uncompressed size: " << uncompSize << std::endl;
541     std::cout << "ratio: " << ratio * 100.0 << "%" << std::endl;
542     std::cout << "uncompressed name: " << file << std::endl;
```

```
543 }
544
545 void Test(const std::string& fileName) {
546
547     std::string file = fileName;
548
549     if (fileName.size() > 2) {
550         if (fileName.substr(fileName.length() - 2) == ".Z") {
551             file = fileName.substr(0, fileName.length() - 2);
552         }
553     }
554
555     std::ifstream input(file + ".Z", std::ios::binary | std::ios::in);
556
557     if (!input) {
558         std::cout << "Архив с таким именем не существует : " << file
                + ".Z" << std::endl;
559         input.close();
560         return;
561     }
562
563     uint64_t testNumber = 0;
564     uint64_t lastPos = 0;
565     uint64_t lastPosReal = 0;
566
567     input.read((char*)(&testNumber), sizeof(uint64_t));
568
569     if (testNumber != TEST_NUMBER) {
570         std::cout << "Архив был поврежден: " << file + ".Z" << std::
                endl;
571         input.close();
572         return;
573     }
574     input.seekg(-BYTE_SIZE, input.end);
575     lastPosReal = input.tellg();
576     input.read((char*)(&lastPos), sizeof(uint64_t));
```

```
577
578         if (lastPosReal != lastPos) {
579             std::cout << "Архив␣поврежден:␣" << file + ".Z" << std::endl;
580             input.close();
581             return;
582         }
583         input.close();
584     }
585
586     int main(int argc, char* argv[]) {
587
588         if (argc == 1) {
589             return 0;
590         }
591
592         std::string curProgram = argv[0];
593         curProgram = curProgram.substr(2);
594
595         bool writeToStdout = false;
596         bool readFromStdin = false;
597         bool decompress = false;
598         bool keepFile = false;
599         bool recursive = false;
600         bool info = false;
601         bool test = false;
602
603         for (int i = 1; i < argc; ++i) {
604             std::string key = argv[i];
605
606             if (key == "-c") {
607                 writeToStdout = true;
608             } else if (key == "-d") {
609                 decompress = true;
610             } else if (key == "-k") {
611                 keepFile = true;
612             } else if (key == "-r") {
```

```
613             recursive = true;
614         } else if (key == "-l") {
615             info = true;
616         } else if (key == "-t") {
617             test = true;
618         } else if (key == "-9") {
619             DICT_SIZE = MAX_DICT_SIZE;
620         } else if (key == "-1") {
621             DICT_SIZE = MIN_DICT_SIZE;
622         }
623     }
624
625     std::string fileName = !recursive ? argv[argc - 1] : "";
626     if (fileName == "-") {
627         readFromStdin = true;
628         writeToStdout = true;
629     }
630
631     std::vector<std::string> files;
632     if (recursive) {
633         for (auto& file : std::experimental::filesystem::
                 recursive_directory_iterator(std::experimental::filesystem::
                 current_path())) {
634             if (std::experimental::filesystem::is_regular_file(file.path())) {
635                 std::string name = file.path().string();
636
637                 if (name.length() >= curProgram.length() && name.substr(
                         name.length() - curProgram.length()) == curProgram) {
638                     continue;
639                 }
640                 if (decompress || info || test) {
641                     if (name.size() > 2 && name.substr(name.length() - 2)
                             == ".Z") {
642                         files.push_back(name);
643                     }
644                 } else {
```

```
645                         if (name.size() < 3 || name.substr(name.length() - 2) !=
                                ".Z") {
646                             files.push_back(name);
647                         }
648                     }
649                 }
650             }
651         } else {
652             files.push_back(fileName);
653         }
654 
655         if (info) {
656             for (const std::string& file : files) {
657                 ArchiveInformation(file);
658             }
659         }
660         if (test) {
661             for (const std::string& file : files) {
662                 Test(file);
663             }
664         }
665         if (decompress) {
666             for (const std::string& file : files) {
667                 Decode(file, keepFile, readFromStdin, writeToStdout);
668             }
669         } else {
670             if (!info && !test) {
671                 for (const std::string& file : files) {
672                     Encode(file, keepFile, readFromStdin, writeToStdout);
673                 }
674             }
675         }
676         return 0;
677 }
```

Выводы

Выполнив данный курсовой проект я узнал много нового в области сжатия данных, провел исследование в данной предметной области и применил знания, полученные в течение курса для разработки собственного архиватора. В процессе разработки архиватора я приобрел новые практическме и теоретические знания в реализации алгоритмов, предназначенных для сжатия данных без потерь. Сжатие данных является очень важной областью. Оно используется для экономии дискового пространства, пересылки больших файлов через интернет, а так же для других важных вещей. Существует большое множество архиваторов, использующих разные алгоритмы, соревнующихся между собой в степени и времени сжатия. Разработанный мной архиватор успешно проводит сжатие данных, но уступает по времени и степени сжатия известным архиваторам, таким как gzip. Процесс написание собственного архиватора был очень интересным и познавательным. Полученные навыки и знания могут пригодиться мне в будущем.