МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационыые технологиии прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Курсовой проект
по курсу «Дискретный анализ»

Студент: Д. Д. Наумов
Преподаватель: Н. А. Зацепин
Группа: М8О-406Б-17
Дата:
Оценка:
Подпись:

Москва, 2020

Тема: Архиватор

Описание

Задача состоит в реализации архиватора, на основе алгоритма арифметическаго сжатия. Программа состоит из двух частей: кодера и декодера, которые соответственно сжимают и разжимают входные данные. В качестве входных данных декодеру принимаются текстовые файлы размеров меньше чем 2^{31} бит = 2,6 ГБ. На выходе получается сжатый файл, который подается на вход декодеру.

Описание алгоритма

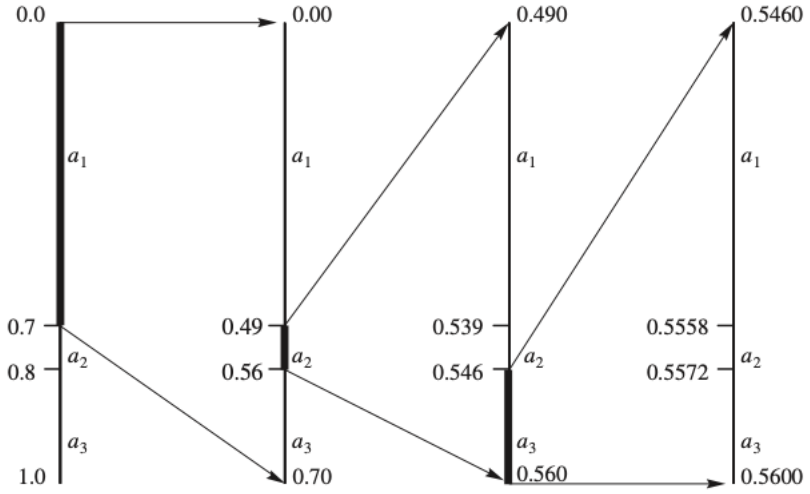
При арифметическом кодировании для кодируемой последовательности создается уникальный идентификатор или тег. Этот тег соответствует двоичной дроби, которая становится двоичным кодом последовательности. Уникальный арифметический код может быть сгенерирован для последовательности длиной m без необходимости генерировать кодовые слова для всех последовательностей длины m. Это не похоже на ситуацию с кодами Хаффмана. Чтобы сгенерировать код Хаффмана для последовательности длины m, где код не является конкатенацией кодовых слов для отдельных символов, нам необходимо получить коды Хаффмана для всех последовательностей длины m.

Чтобы отличить последовательность символов от другой последовательности символов, нам нужно пометить ее уникальным идентификатором. Один из возможных наборов тегов для представления последовательностей символов - это числа в единичном интервале $[0, 1)$. Поскольку количество чисел в единичном интервале бесконечно, должно быть возможно присвоить уникальный тег каждой отдельной последовательности символов.

Процедура создания тега работает за счет уменьшения размера интервала, в котором находится тег, по мере получения все большего количества элементов последовательности.

Пример

Рассмотрим трехбуквенный алфавит $A = \{a_1, a_2, a_3\}$ с $P(a_1) = 0{,}7$, $P(a_2) = 0{,}1$ и $P(a_3) = 0{,}2$. Используя отображение $F_X(1) = 0{,}7$, $F_X(2) = 0{,}8$ и $F_X(3) = 1$ разбиваем единичный интервал, как показано на рисунке ниже.



Для последовательности, начинающейся с $a_1, a_2, a_3, \ldots$, к тому времени, когда будет получен третий символ a_3, тег будет ограничен субынтервалом $[0,546, 0,56)$. Если бы третьим символом был a_1 вместо a_3, тег находился бы в подынтервале $[0,49, 0,539)$, который не пересекается с подынтервалом $[0,546, 0,56)$. Даже если с этого момента две последовательности идентичны (одна начинается с a_1, a_2, a_3, а другая начинается с a_1, a_2, a_1), интервал тегов для двух последовательностей всегда будет непересекающимся.

Описание структуры программы

Программа разделена на следующие фрагменты:

- **BitIoStream** - поток побитового ввода/вывода
- **FrequencyTable** - таблица с частотами символов
- **ArithmeticCoder** - функции для обновления границ тега
- **ArithmeticCompress** - основная программа для сжатия
- **ArithmeticDecompress** - программа для декодирования

Пример работы программы

```
(base) MacBook:src dandachok$ ./encoder ../tests/input/1mb.t ../tests/encode/1mb.az
(base) MacBook:src dandachok$ du -h ../tests/input/1mb.t
980K ../tests/input/1mb.t
(base) MacBook:src dandachok$ du -h ../tests/encode/1mb.az
568K ../tests/encode/1mb.az
(base) MacBook:src dandachok$ ./decoder ../tests/encode/1mb.az ../tests/output/1mb.t
(base) MacBook:src dandachok$ du -h ../tests/output/1mb.t
980K ../tests/output/1mb.t
(base) MacBook:src dandachok$ diff ../tests/input/1mb.t ../tests/output/1mb.t
(base) MacBook:src dandachok$
```

Результаты

Файла	Размер файла	Алгоритм	Время сжатия	Время декомпрессии	Размер сжатого файла	Коэффициент сжатия
world95.txt	3005020 Б	Арифметика	0m2.306s	0m2.725s	1920022 Б	1.56
world95.txt	3005020 Б	gzip	0m0.309s	0m4.359s	868440 Б	3.4
enwik8	100000000 Б	Арифметика	1m15.385s	1m29.597s	63502180 Б	1.57
enwik8	100000000 Б	gzip	0m8.411s	0m2.545s	36475811 Б	2.75

- **Процессор** 1,8 GHz 2-ядерный процессор Intel Core i5
- **Память** 4 ГБ 1600 MHz DDR3
- **Графика** Intel HD Graphics 4000 1536 МБ

Листинг программы

ArithmeticCoder.hpp

```
#pragma once

#include <algorithm>
#include <cstdint>
#include "BitIoStream.hpp"
#include "FrequencyTable.hpp"

class ArithmeticCoderBase {
public:
  explicit ArithmeticCoderBase(int numBits);
  virtual ~ArithmeticCoderBase() = 0;

protected:

  int numStateBits;
  std::uint64_t fullRange;
  std::uint64_t halfRange;
  std::uint64_t quarterRange;
  std::uint64_t minimumRange;
  std::uint64_t maximumTotal;
  std::uint64_t stateMask;
  std::uint64_t low;
  std::uint64_t high;

  virtual void update(const FrequencyTable &freqs, std::uint32_t symbol);
  virtual void shift() = 0;
  virtual void underflow() = 0;
};


class ArithmeticDecoder final : private ArithmeticCoderBase {
public:
  explicit ArithmeticDecoder(int numBits, BitInputStream &in);
  std::uint32_t read(const FrequencyTable &freqs);

protected:
  void shift() override;
  void underflow() override;

private:
  BitInputStream &input;
  std::uint64_t code;

  int readCodeBit();
};

```

```
47
48 class ArithmeticEncoder final : private ArithmeticCoderBase {
49
50 public:
51   explicit ArithmeticEncoder(int numBits, BitOutputStream &out);
52   void write(const FrequencyTable &freqs, std::uint32_t symbol);
53   void finish();
54
55 protected:
56   void shift() override;
57   void underflow() override;
58
59 private:
60   BitOutputStream &output;
61   unsigned long numUnderflow;
62 };
```

ArithmeticCoder.cpp

```
1  #include <limits>
2  #include <stdexcept>
3  #include "ArithmeticCoder.hpp"
4
5  using std::uint32_t;
6  using std::uint64_t;
7
8
9  ArithmeticCoderBase::ArithmeticCoderBase(int numBits) {
10   if (numBits < 1 || numBits > 63)
11     throw std::domain_error("State size out of range");
12   numStateBits = numBits;
13   fullRange = static_cast<decltype(fullRange)>(1) << numStateBits;
14   halfRange = fullRange >> 1;
15   quarterRange = halfRange >> 1;
16   minimumRange = quarterRange + 2;
17   maximumTotal = std::min(std::numeric_limits<decltype(fullRange)>::max() / fullRange,
         minimumRange);
18   stateMask = fullRange - 1;
19   low = 0;
20   high = stateMask;
21 }
22
23 ArithmeticCoderBase::~ArithmeticCoderBase() {}
24
25 void ArithmeticCoderBase::update(const FrequencyTable &freqs, uint32_t symbol) {
26   if (low >= high || (low & stateMask) != low || (high & stateMask) != high)
27     throw std::logic_error("Assertion error: Low or high out of range");
28   uint64_t range = high - low + 1;
29   if (range < minimumRange || range > fullRange)
30     throw std::logic_error("Assertion error: Range out of range");
```

```
31
32    uint32_t total = freqs.getTotal();
33    uint32_t symLow = freqs.getLow(symbol);
34    uint32_t symHigh = freqs.getHigh(symbol);
35    if (symLow == symHigh)
36      throw std::invalid_argument("Symbol has zero frequency");
37    if (total > maximumTotal)
38      throw std::invalid_argument("Cannot code symbol because total is too large");
39
40    uint64_t newLow = low + symLow * range / total;
41    uint64_t newHigh = low + symHigh * range / total - 1;
42    low = newLow;
43    high = newHigh;
44
45    while (((low ^ high) & halfRange) == 0) {
46      shift();
47      low = ((low << 1) & stateMask);
48      high = ((high << 1) & stateMask) | 1;
49    }
50
51    while ((low & ~high & quarterRange) != 0) {
52      underflow();
53      low = (low << 1) ^ halfRange;
54      high = ((high ^ halfRange) << 1) | halfRange | 1;
55    }
56  }
57
58
59  ArithmeticDecoder::ArithmeticDecoder(int numBits, BitInputStream &in) :
60      ArithmeticCoderBase(numBits),
61      input(in),
62      code(0) {
63    for (int i = 0; i < numStateBits; i++)
64      code = code << 1 | readCodeBit();
65  }
66
67  uint32_t ArithmeticDecoder::read(const FrequencyTable &freqs) {
68    uint32_t total = freqs.getTotal();
69    if (total > maximumTotal)
70      throw std::invalid_argument("Cannot decode symbol because total is too large");
71    uint64_t range = high - low + 1;
72    uint64_t offset = code - low;
73    uint64_t value = ((offset + 1) * total - 1) / range;
74    if (value * range / total > offset)
75      throw std::logic_error("Assertion error");
76    if (value >= total)
77      throw std::logic_error("Assertion error");
78
79    uint32_t start = 0;
```

```
  uint32_t end = freqs.getSymbolLimit();
  while (end - start > 1) {
    uint32_t middle = (start + end) >> 1;
    if (freqs.getLow(middle) > value)
      end = middle;
    else
      start = middle;
  }
  if (start + 1 != end)
    throw std::logic_error("Assertion error");

  uint32_t symbol = start;
  if (offset < freqs.getLow(symbol) * range / total || freqs.getHigh(symbol) * range /
      total <= offset)
    throw std::logic_error("Assertion error");
  update(freqs, symbol);
  if (code < low || code > high)
    throw std::logic_error("Assertion error: Code out of range");
  return symbol;
}

void ArithmeticDecoder::shift() {
  code = ((code << 1) & stateMask) | readCodeBit();
}

void ArithmeticDecoder::underflow() {
  code = (code & halfRange) | ((code << 1) & (stateMask >> 1)) | readCodeBit();
}

int ArithmeticDecoder::readCodeBit() {
  int temp = input.read();
  if (temp == -1)
    temp = 0;
  return temp;
}


ArithmeticEncoder::ArithmeticEncoder(int numBits, BitOutputStream &out) :
  ArithmeticCoderBase(numBits),
  output(out),
  numUnderflow(0) {}

void ArithmeticEncoder::write(const FrequencyTable &freqs, uint32_t symbol) {
  update(freqs, symbol);
}

void ArithmeticEncoder::finish() {
  output.write(1);
}
```

```

void ArithmeticEncoder::shift() {
  int bit = static_cast<int>(low >> (numStateBits - 1));
  output.write(bit);

  // Write out the saved underflow bits
  for (; numUnderflow > 0; numUnderflow--)
    output.write(bit ^ 1);
}

void ArithmeticEncoder::underflow() {
  if (numUnderflow == std::numeric_limits<decltype(numUnderflow)>::max())
    throw std::overflow_error("Maximum underflow reached");
  numUnderflow++;
}
```

ArithmeticCompress.cpp

```
#include <cstdint>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <vector>
#include "ArithmeticCoder.hpp"
#include "BitIoStream.hpp"
#include "FrequencyTable.hpp"

using std::uint32_t;


int main(int argc, char *argv[]) {
  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " InputFile OutputFile" << std::endl;
    return EXIT_FAILURE;
  }
  const char *inputFile = argv[1];
  const char *outputFile = argv[2];

  std::ifstream in(inputFile, std::ios::binary);
  SimpleFrequencyTable freqs(std::vector<uint32_t>(257, 0));
  freqs.increment(256);
  while (true) {
    int b = in.get();
    if (b == EOF)
      break;
    if (b < 0 || b > 255)
      throw std::logic_error("Assertion error");
    freqs.increment(static_cast<uint32_t>(b));
  }
```

```

  in.clear();
  in.seekg(0);
  std::ofstream out(outputFile, std::ios::binary);
  BitOutputStream bout(out);
  try {

    for (uint32_t i = 0; i < 256; i++) {
      uint32_t freq = freqs.get(i);
      for (int j = 31; j >= 0; j--)
        bout.write(static_cast<int>((freq >> j) & 1)); // Big endian
    }

    ArithmeticEncoder enc(32, bout);
    while (true) {
      int symbol = in.get();
      if (symbol == EOF)
        break;
      if (symbol < 0 || symbol > 255)
        throw std::logic_error("Assertion error");
      enc.write(freqs, static_cast<uint32_t>(symbol));
    }

    enc.write(freqs, 256);
    enc.finish();
    bout.finish();
    return EXIT_SUCCESS;

  } catch (const char *msg) {
    std::cerr << msg << std::endl;
    return EXIT_FAILURE;
  }
}
```

ArithmeticDecompress.cpp

```
#include <cstdint>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <limits>
#include <vector>
#include "ArithmeticCoder.hpp"
#include "BitIoStream.hpp"
#include "FrequencyTable.hpp"

using std::uint32_t;


int main(int argc, char *argv[]) {
```

```
  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " InputFile OutputFile" << std::endl;
    return EXIT_FAILURE;
  }
  const char *inputFile = argv[1];
  const char *outputFile = argv[2];

  std::ifstream in(inputFile, std::ios::binary);
  std::ofstream out(outputFile, std::ios::binary);
  BitInputStream bin(in);
  try {

    SimpleFrequencyTable freqs(std::vector<uint32_t>(257, 0));
    for (uint32_t i = 0; i < 256; i++) {
      uint32_t freq = 0;
      for (int j = 0; j < 32; j++)
        freq = (freq << 1) | bin.readNoEof(); // Big endian
      freqs.set(i, freq);
    }
    freqs.increment(256);

    ArithmeticDecoder dec(32, bin);
    while (true) {
      uint32_t symbol = dec.read(freqs);
      if (symbol == 256)
        break;
      int b = static_cast<int>(symbol);
      if (std::numeric_limits<char>::is_signed)
        b -= (b >> 7) << 8;
      out.put(static_cast<char>(b));
    }
    return EXIT_SUCCESS;

  } catch (const char *msg) {
    std::cerr << msg << std::endl;
    return EXIT_FAILURE;
  }
}
```

Выводы

Идея арифметического сжатия довольно простая, однако при реализации возникает много нюансов. Из-за ограничений точности типов с плавающей запятой, невозможно просто пересчитывать границы интервала. Необходимо переходить от вещественных значений к целым. Так же при реализации возникают ограничения на максимальный размер входного файла.