**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра систем автоматизации и приборостроения**

**ОТЧЕТ**

**по лабораторной работе**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Алгоритмы сжатия данных**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1322 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Прохоров Д.А. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Пестерев Д.О. |

Санкт-Петербург

2024

## Техническое задание

1. Реализовать следующие алгоритмы:

## 1. Huffman algorithm (HA) 2. Run-length encoding (RLE) 3. BWT + RLE 4. BWT + MTF + HA 5. BWT + MTF + RLE + HA 6. LZ77 7. LZ77 + HA

## 1. Компрессоры

## HA

Классический алгоритм Хаффмана в зависимости от частоты встречаемости символов, меняет длину кодированного символа. Мне пришлось изменить это в угоду однозначности декодирования кодов. Принцип работы алгоритма следующий.

**Построение словаря кодов:**

Сначала происходит анализ входных данных для определения частоты появления каждого символа.

На основе этой статистики строится словарь, где каждому символу сопоставляется его уникальный битовый код. Длинна этого кода зависит от количества уникальных символов в строке.

def build\_dictionary(text):

    # Сбор статистики частот символов

    freq = {}

    for char in text:

        if char in freq:

            freq[char] += 1

        else:

            freq[char] = 1

    # Определение фиксированной длины кода

    n = len(freq)

    code\_length = math.ceil(math.log2(n))

    # Построение кодовой таблицы

    huffman\_codes = {}

    binary\_format = '{:0' + str(code\_length) + 'b}'

    for i, char in enumerate(freq):

        huffman\_codes[char] = binary\_format.format(i)

    return huffman\_codes

**Кодирование:**

Входные данные проходят построение кодов с использованием словаря Хаффмана. Каждый символ заменяется его соответствующим битовым кодом из словаря.

def huffman\_encode(string):

    huffman\_dict = build\_dictionary(string)

    compressed = ''.join(huffman\_dict[symbol] for symbol in string)

    return compressed, huffman\_dict

**Декодирование:**

Закодированные данные декодируются обратно в исходную последовательность символов с использованием словаря кодов Хаффмана. Коды из закодированных данных сопоставляются соответствующим символам из словаря, и они заменяются исходными символами.

def huffman\_decode(encoded\_data, codes):

    reverse\_codes = {v: k for k, v in codes.items()}

    decoded\_data = ""

    temp = ""

    for bit in encoded\_data:

        temp += bit

        if temp in reverse\_codes:

            decoded\_data += reverse\_codes[temp]

            temp = ""

    return decoded\_data

**Запись закодированных данных:**

Для хранения закодированных данных часто используются битовые последовательности. Коды символов записываются в битовом формате для компактного представления данных.

compressed, huffman\_dict = huffman\_encode(string)

    bit\_array = bitarray.bitarray()

    for code in compressed:

        bit\_array.append(int(code))

Таким образом, измененный алгоритм Хаффмана основывается на построении оптимального битового представления данных, где длина кодов зависит от количества уникальных символов, что позволяет кодировать и декодировать данные без потерь.

## RLE

Алгоритм Run-Length Encoding (RLE) предназначен для сжатия данных за счет сокращения последовательностей повторяющихся символов. **Принципы кодирования RLE (Run-Length Encoding)**

**Кодирование:**

1. **Последовательный обход данных:** Начнем с первого символа строки и будем проходить по каждому символу до конца строки.
2. **Подсчет повторений:** Для каждого символа будем подсчитывать количество его последовательных повторений.
3. **Запись кодировки:** Если символ повторяется, запишем количество повторений перед этим символом. Если символ встречается только один раз, запишем его без числа.
4. **Переход к следующему символу:** Переходим к следующему символу, который не является частью текущей последовательности повторений, и повторяем процесс до конца строки.

Пример:

* Исходная строка: "aaabbbbccdaa"
* Закодированная строка: "3a4b2cd2a"

def rle\_encoding(data):

    encoded\_data = ''

    i = 0

    while i < len(data):

        count = 1

        while i + 1 < len(data) and data[i] == data[i + 1]:

            i += 1

            count += 1

        if count == 1:

            encoded\_data += data[i]

        else:

            encoded\_data += str(count) + data[i]

        i += 1

    return encoded\_data

**Декодирование:**

1. **Обход закодированных данных:** Начнем с первого символа закодированной строки и будем проходить по каждому символу до конца строки.
2. **Проверка на цифры:** Если текущий символ является цифрой, то это часть числа, указывающего количество повторений следующего символа.
3. **Запись декодированных данных:** Как только встречаем символ, который не является цифрой, проверяем, есть ли перед ним число. Если есть, записываем этот символ количество раз, равное этому числу. Если числа нет, записываем символ один раз.
4. **Переход к следующему символу:** Переходим к следующему символу и повторяем процесс до конца строки.

Пример:

* Закодированная строка: "3a4b2cd2a"
* Декодированная строка: "aaabbbbccdaa"

def rle\_decoding(encoded):

    decoded = ''

    count = ''

    for char in encoded:

        if char.isdigit():

            count += char

        else:

            if count == '':

                decoded += char

            else:

                decoded += char \* int(count)

                count = ''

    return decoded

Эти функции позволяют сжать файл с использованием RLE и затем восстановить его обратно.

## BWT + RLE

Алгоритм преобразования Барроуза-Уилера (Burrows-Wheeler Transform, BWT) используется для подготовки строки к эффективному сжатию данных таким образом, что последовательности повторяющихся символов становятся более явными. Принцип его работы следующий

Принципы кодирования методом Барроуза-Уиллера (BWT):

1. **Преобразование Барроуза-Уиллера (BWT)**:

* Преобразует исходную строку в последовательность строк, каждая из которых является циклическим сдвигом исходной строки.
* Сортирует эти строки лексикографически.
* Возвращает последний символ каждой строки, формируя преобразованную строку.

Пример:

* Исходная строка: "banana"
* Преобразованная строка: "annb$aa"
* def calculate\_entropy(data):
* dict\_char = {i:0 for i in range(128)}
* for c in data:
* dict\_char[ord(c)] = dict\_char[ord(c)] + 1
* N = len(data)
* prob = np.array([dict\_char[i]/N for i in range(128)])
* entropy\_value = 0
* for i in range(128):
* if prob[i]:
* entropy\_value -= np.log2(prob[i]) \* prob[i]
* return entropy\_value
* def compute\_suffix\_array(input\_text, len\_text):
* suff = [(i, input\_text[i:]) for i in range(len\_text)]
* suff.sort(key=lambda x: x[1]) # Сортирует вращения
* suffix\_arr = [i for i, \_ in suff]
* return suffix\_arr
* def find\_last\_char(input\_text, suffix\_arr, n):
* bwt\_arr = ""
* for i in range(n):
* j = suffix\_arr[i] - 1
* if j < 0:
* j = j + n
* bwt\_arr += input\_text[j]
* return bwt\_arr
* def inverse\_burrows\_wheeler\_transform(bwt\_arr):
* len\_bwt = len(bwt\_arr)
* i = 0
* while bwt\_arr[i] != "$":
* i+=1
* x = i
* sorted\_bwt = sorted(bwt\_arr)
* l\_shift = [0] \* len\_bwt
* arr = [[] for i in range(128)]
* for i in range(len\_bwt):
* arr[ord(bwt\_arr[i])].append(i)
* for i in range(len\_bwt):
* l\_shift[i] = arr[ord(sorted\_bwt[i])].pop(0)
* decoded = [''] \* len\_bwt
* for i in range(len\_bwt):
* x = l\_shift[x]
* decoded[len\_bwt-1-i] = bwt\_arr[x]
* decoded\_str = ''.join(decoded)
* decoded\_str = decoded\_str[::-1]
* return decoded\_str[:-1]
* def burrows\_wheeler\_transform(input\_text):
* input\_text += "$"
* len\_text = len(input\_text)
* suffix\_arr = compute\_suffix\_array(input\_text, len\_text)
* bwt\_arr = find\_last\_char(input\_text, suffix\_arr, len\_text)
* return bwt\_arr

Эта функция принимает исходную строку и возвращает её преобразование методом Барроуза-Уиллера.

RLE работает точно так же, как и в предыдущем компрессоре.

## BWT+MTF+HA

Алгоритм Move-To-Front (MTF) также используется для преобразования данных с целью повышения эффективности их последующего сжатия. Принцип его работы следующий

**MTF-кодирование:**

* Создается алфавит, содержащий все возможные символы, упорядоченные по порядку их первого появления в тексте.
* Для каждого символа входного текста находится его индекс в алфавите.
* Символ перемещается в начало алфавита, а его индекс записывается в закодированный поток.
* После перемещения символа в начало алфавита порядок символов в нем переупорядочивается так, чтобы символ, который был использован, оказался в самом начале.
* def mtf(text):
* alphabet = [chr(i) for i in range(256)]  # Создаем алфавит символов
* result = []
* for char in text:
* index = alphabet.index(char)  # Находим индекс символа в алфавите
* result.append(index)  # Добавляем индекс в результат
* alphabet.pop(index)  # Удаляем символ из алфавита
* alphabet.insert(0, char)  # Перемещаем символ в начало алфавита
* return result

**MTF-декодирование:**

* По аналогии с кодированием, алфавит формируется в начале процесса, включающего все возможные символы в порядке их первоначального появления.
* Индексы, полученные в результате MTF-кодирования, используются для выбора символов из алфавита.
* Выбранный символ перемещается в начало алфавита, и из него формируется восстановленная последовательность символов.
* def inverse\_mtf(indices):
* alphabet = [chr(i) for i in range(256)]
* result = []
* for index in indices:
* char = alphabet[index]  # Получаем символ по индексу из алфавита
* result.append(char)  # Добавляем символ в результат
* alphabet.pop(index)  # Удаляем символ из алфавита
* alphabet.insert(0, char)  # Перемещаем символ в начало алфавита
* return ''.join(result)

BWT работает точно так же, как и в предыдущем компрессоре.

HA изменен для корректного декодирования в список

## BWT+MTF+RLE+HA

## Использует алгоритмы из предыдущих компрессоров, но RLE переписан, для работы с массивами

def encode\_rle(index):

    if not index:

        return []

    if len(index) == 1:

        return [index[0]]

    encoded = []

    i = 0

    while i < len(index) - 1:

        count = 1

        while i < len(index) - 1 and index[i] == index[i+1]:

            i += 1

            count += 1

        if count == 1:

            encoded.append(index[i])

        else:

            encoded.append((index[i], count))

        i += 1

    if i < len(index):

        encoded.append((index[i], 1))

    return encoded

def decode\_rle(encoded):

    decoded = []

    i = 0

    while i < len(encoded):

        if isinstance(encoded[i], int):

            decoded.append(encoded[i])

            i += 1

        elif isinstance(encoded[i], tuple):

            pixel = encoded[i][0]

            count = encoded[i][1]

            decoded.extend([pixel] \* count)

            i += 1

    return decoded

## LZ77

Алгоритм LZ77 сжимает данные, основываясь на поиске и кодировании повторяющихся последовательностей символов. Принцип его работы следующий:

Кодирование:

* Для каждого символа в исходной строке ищем самое длинное совпадение в пределах окна поиска.
* Если совпадение найдено, создаем тройку (смещение, длина совпадения, следующий символ). Если совпадения нет, создаем тройку с нулевыми значениями для смещения и длины, а в качестве следующего символа указываем текущий символ.
* Продвигаемся вперед на длину найденного совпадения плюс один символ, либо на один символ, если совпадения не было.
* Записываем тройки в выходной массив.
* def compress(self, input\_data):
* i = 0
* output = []
* while i < len(input\_data):
* match = self.\_find\_longest\_match(input\_data, i)
* if match:
* (best\_match\_distance, best\_match\_length) = match
* next\_symbol = input\_data[i + best\_match\_length]
* output.append((best\_match\_distance, best\_match\_length, next\_symbol))
* i += best\_match\_length + 1
* else:
* output.append((0, 0, input\_data[i]))
* i += 1
* return output
* def \_find\_longest\_match(self, data, current\_position):
* # Поиск самого длинного совпадения в окне
* end\_of\_buffer = min(current\_position + self.window\_size, len(data))
* best\_match\_distance = -1
* best\_match\_length = -1
* for j in range(current\_position + 1, end\_of\_buffer):
* start\_index = max(0, current\_position - self.window\_size)
* substring = data[current\_position:j]
* for i in range(start\_index, current\_position):
* match\_length = 0
* while match\_length < len(substring) and data[i + match\_length] == substring[match\_length]:
* match\_length += 1
* if match\_length > best\_match\_length:
* # Если текущее совпадение длиннее предыдущего, обновляем значения
* best\_match\_distance = current\_position - i
* best\_match\_length = match\_length
* if best\_match\_distance > 0 and best\_match\_length > 0:
* return best\_match\_distance, best\_match\_length

Декодирование:

* Считываем тройки (смещение, длина, следующий символ) из сжатого файла.
* Для каждой тройки добавляем в выходную строку символы, соответствующие найденному смещению и длине совпадения, затем добавляем следующий символ.
* Записываем восстановленную строку в выходной файл.
* def decompress(self, compressed\_data):
* decompressed\_data = []
* for item in compressed\_data:
* # Для каждого элемента сжатых данных добавляем декодированные символы в выходную строку
* (distance, length, symbol) = item
* if distance > 0:
* # Если дистанция больше нуля, восстанавливаем символы из буфера
* start = len(decompressed\_data) - distance
* for j in range(length):
* decompressed\_data.append(decompressed\_data[start + j])
* decompressed\_data.append(symbol)
* return ''.join(decompressed\_data)

## LZ77+HA

Компрессор использует алгоритмы описанные выше

## Реализация:

Ссылка на репозиторий Github с исходным кодом:

<https://github.com/mistakins/CompressPy>

## Результаты исследования эффективности различных компрессоров

Энтропия тестового текста 4.68045

Размер 100 Кб

Текст для теста представляет собой строку строчных букв:   
jujuonysopnagrntypqcgjrkvmumojziilgzyzj… и так далее

|  |  |
| --- | --- |
| Компрессор | Коэффициент сжатия |
| Huffman algorithm (HA) | 1.58 |
| Run-length encoding (RLE) | 1.01 |
| BWT + RLE | 1.11 |
| BWT + MTF + HA | 1.33 |
| BWT + MTF + RLE + HA | 1.63 |

## Выводы об эффективности различных алгоритмов сжатия:

**HA:**

Коэффициент сжатия: 1.58

Эффективный алгоритм, его главное преимущество универсальность, но чем больше в тексте будет уникальных символов, тем менее эффективным он будет.

**RLE:**

Коэффициент сжатия: 1.01

Эффективен для данных с длинными последовательностями повторяющихся символов, но в нашем случае коэффициент сжатия мизерный, и не идет в сравнение с другими методами.

**BWT + RLE:**

Коэффициент сжатия: 1.11

BWT значительно увеличивает коэффициент сжатия, демонстрируя высокую эффективность.

**BWT + MTF + HA:**

Коэффициент сжатия: 1.33

Сочетание этих алгоритмов вопреки ожиданиям не улучшает коэффициент сжатия, и уступает обычному HA

**BWT + MTF + RLE + HA:**

Коэффициент сжатия: 1.63

Наивысший коэффициент сжатия среди всех представленных методов.

## Вывод

В ходе проделанной работы были реализованы и проверены различные алгоритмы сжатия. Наиболее эффективным методом сжатия данных из представленных является BWT + MTF + RLE + HA, который достигает наивысшего коэффициента сжатия 1.63.