

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

РГЗ

по дисциплине: Теория информации

тема: «Совместное применение алгоритмов вероятностно-статистических и
структурных алгоритмов кодирования»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
пр. Твердохлеб Виталий Викторович

Белгород 2024 г.

РГЗ

Совместное применение алгоритмов вероятностно-статистических и структурных алгоритмов кодирования

Цель работы: разработать и оценить эффективность совместных алгоритмов кодирования, основанных на вероятности и статистике и структуре файла.

Одним из самых эффективных алгоритмов вероятно-статистического кодирования является алгоритм Хаффмана, он и был взят за основу при разработке приложения. Введём дополнительные шаги кодирования - обработка структурным алгоритмом кодирования, им может быть, например, RLE.

В результате получим консольное приложение, с помощью которого можно закодировать введённый файл и сохранить его в трёх кодировках - Хаффмана, Хаффмана + RLE и RLE + Хаффмана. Данное консольное приложение позволит исследовать плюсы и минусы совместных алгоритмов кодирования, выявить, где они эффективны и неэффективны. Спецификация:

Класс `Coder`:

1. Заголовок: `public static List<Byte> getEncodedHuffman(List<Byte> input)`
 2. Назначение: возвращает последовательность закодированных байт, полученных из сообщения `input` алгоритмом Хаффмана.
-

1. Заголовок: `private static List<Byte> getEncodedFromTable(List<Byte> input, List<TableElement> table)`
 2. Назначение: возвращает последовательность закодированных байт на основе сообщения `input` и префиксной таблицы `table`.
-

1. Заголовок: `public static List<Byte> getEncodedRLE(List<Byte> input)`
 2. Назначение: возвращает последовательность закодированных байт, полученных из сообщения `input` алгоритмом RLE.
-

1. Заголовок: `private static void writeUnrepeatingBufferBytesRLE(List<Byte> unrepeatingBuffer, List<Byte> result)`
 2. Назначение: записывает неповторяющиеся символы из буфера `unrepeatingBuffer` в массив байтов `result`.
-

1. Заголовок: `public static List<TableElement> getHuffmanTable(List<Byte> input)`
2. Назначение: возвращает префиксную таблицу для сообщения `input`.

-
1. Заголовок: `private static List<TableElement> getSegmentisedTable(List<Byte> input)`
 2. Назначение: инициализирует префиксную таблицу для сообщения `input`, разбивая его на сегменты по 8 бит.

Класс HuffmanTableElement:

1. Заголовок: `public HuffmanTableElement(HuffmanTableElement left, HuffmanTableElement right)`
2. Назначение: конструирует объект HuffmanTableElement как узел дерева с поддеревьями `left` и `right`.

-
1. Заголовок: `public HuffmanTableElement(TableElement self)`
 2. Назначение: конструирует объект HuffmanTableElement как лист со значением `self`.

-
1. Заголовок: `public List<TableElement> getTableElement()`
 2. Назначение: возвращает префиксную таблицу.

-
1. Заголовок: `private List<TableElement> getTableElement(List<Boolean> prefix)`
 2. Назначение: возвращает префиксную таблицу обходом в глубину на основе префикса `prefix`.

Класс TableElement:

1. Заголовок: `public TableElement(byte symbol)`
2. Назначение: конструирует объект TableElement на основе элемента `symbol`.

-
1. Заголовок: `int getCode()`
 2. Назначение: возвращает префиксный код элемента в виде целого числа.

Исходный код: *Main.java*

```

package rchat.info.lab3;

import rchat.info.libs.Coder;

import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Введите путь к файлу: ");
        try {
            File inputFile = new File(r.readLine());

            byte[] bytes = Files.readAllBytes(inputFile.toPath());
            Byte[] bytesConverted = new Byte[bytes.length];
            for (int i = 0; i < bytes.length; i++)
                bytesConverted[i] = bytes[i];

            {
                List<Byte> HUF = Coder.getEncodedHuffman(List.of(bytesConverted));
                System.out.println("Кодирование по алгоритму Хаффмана");
                System.out.println("Размер: " + HUF.size() + " байт");
                System.out.println("Коэффициент сжатия: " + (1.0 * bytesConverted.length / HUF.size()));
                System.out.println();
                File HUFOutput = new File(inputFile.getParent() + "/" + inputFile.getName().split("\\.")[0] + ".huf");

                byte[] HUFConverted = new byte[HUF.size()];
                for (int i = 0; i < HUF.size(); i++) {
                    HUFConverted[i] = HUF.get(i);
                }

                Files.write(HUFOutput.toPath(), HUFConverted);
                System.out.println("Результат записан в файл " + HUFOutput.getAbsolutePath());
            }

            System.out.println("=====");

            {
                List<Byte> HUF = Coder.getEncodedHuffman(List.of(bytesConverted));
                List<Byte> HUFRL = Coder.getEncodedRLE(HUF);
                System.out.println("Кодирование по алгоритмам Хаффмана и RLE");
                System.out.println("Размер: " + HUFRL.size() + " байт");
                System.out.println("Коэффициент сжатия: " + (1.0 * bytesConverted.length / HUFRL.size()));
                System.out.println();
                File HUFRLOutput = new File(inputFile.getParent() + "/" + inputFile.getName().split("\\.")[0] +
                    ↵ ".hufrl");

                byte[] HUFRLConverted = new byte[HUFRL.size()];
                for (int i = 0; i < HUFRL.size(); i++) {
                    HUFRLConverted[i] = HUFRL.get(i);
                }
            }
        }
    }
}

```

```

    }

    Files.write(HUFRLEROutput.toPath(), HUFRLERConverted);
    System.out.println("Результат записан в файл " + HUFRLEROutput.getAbsolutePath());
}

System.out.println("=====");

{
    List<Byte> RLE = Coder.getEncodedRLE(List.of(bytesConverted));
    List<Byte> HUFRLER = Coder.getEncodedHuffman(RLE);
    System.out.println("Кодирование по алгоритмам RLE и Хаффмана");
    System.out.println("Размер: " + HUFRLER.size() + " байт");
    System.out.println("Коэффициент сжатия: " + (1.0 * bytesConverted.length / HUFRLER.size()));
    System.out.println();
    File HUFRLEROutput = new File(inputFile.getParent() + "/" + inputFile.getName().split("\\.")[0] +
    ↪ ".rlehuf");

    byte[] HUFRLERConverted = new byte[HUFRLER.size()];
    for (int i = 0; i < HUFRLER.size(); i++) {
        HUFRLERConverted[i] = HUFRLER.get(i);
    }

    Files.write(HUFRLEROutput.toPath(), HUFRLERConverted);
    System.out.println("Результат записан в файл " + HUFRLEROutput.getAbsolutePath());
}
} catch (IOException e) {
    System.out.println("Файл не найден");
}
}
}
}

```

Coder.java

```

package rchat.info.libs;

import java.util.*;
import java.util.stream.Collectors;

public class Coder {
    /* возвращает последовательность закодированных байт,
    * полученных из сообщения input алгоритмом RLE. */
    public static List<Byte> getEncodedRLE(List<Byte> input) {
        List<Byte> result = new ArrayList<>();

        // Создаём дек для подсчёта элементов
        Deque<AbstractMap.SimpleEntry<Byte, Integer>> queue = new ArrayDeque<>();
        // Если массив пустой, то возвращаем пустой результат
        if (input.isEmpty()) {
            return result;
        }
    }
}

```

```

// Добавляем в дек первый элемент
queue.add(new AbstractMap.SimpleEntry<>(input.get(0), 1));
for (int i = 1; i < input.size(); i++) {
    // Если байт элемента сверху равен текущему
    if (queue.peekLast().getKey().equals(input.get(i))) {
        // То увеличиваем кол-во повторяющихся байтов
        AbstractMap.SimpleEntry<Byte, Integer> element = queue.pollLast();
        queue.add(new AbstractMap.SimpleEntry<>(element.getKey(), element.getValue() + 1));
    } else {
        // Иначе кладём в дек новый элемент
        queue.add(new AbstractMap.SimpleEntry<>(input.get(i), 1));
    }
}

List<Byte> unrepeatingBuffer = new ArrayList<>();
while (!queue.isEmpty()) {
    // Пока дек не пуст
    AbstractMap.SimpleEntry<Byte, Integer> element = queue.pollFirst();

    // Если встречен неповторяющийся элемент
    if (element.getValue() == 1) {
        // То добавляем его в буффер неповторяющихся элементов
        unrepeatingBuffer.add(element.getKey());
    } else {
        // Записываем неповторяющиеся элементы
        writeUnrepeatingBufferBytesRLE(unrepeatingBuffer, result);

        // В данном случае в последовательности
        // будут повторяющиеся элементы,
        // в старшем разряде информационного бита будет 1, а
        // следующие 7 бит будут содержать количество элементов + 2.

        // Прим: 1'000_1000 - следующие 8 + 2 байт будут неповторяющимися

        // Таким образом можно сохранить [2; 129] байт, поэтому
        // получаем срез байтов размером до 129 байт
        while (element.getValue() > 0) {
            // Записываем в старший разряд инф. байта 1
            byte info = (byte) (1 << 7);

            // Записываем количество символов
            info += (byte) (element.getValue() % 130 - 2);
            result.add(info);
            result.add(element.getKey());

            element.setValue(element.getValue() - 129);
        }
    }
}

// Если остались неповторяющиеся элементы, записываем их
writeUnrepeatingBufferBytesRLE(unrepeatingBuffer, result);

```

```

    return result;
}

// записывает неповторяющиеся символы из буфера unrepeatingBuffer в массив байтов result.
private static void writeUnrepeatingBufferBytesRLE(List<Byte> unrepeatingBuffer, List<Byte> result) {
    // Записываем в массив байтов неповторяющиеся элементы
    while (!unrepeatingBuffer.isEmpty()) {
        // В данном случае в последовательности
        // будут неповторяющиеся элементы,
        // в старшем разряде информационного бита будет 0, а
        // следующие 7 бит будут содержать количество элементов + 1.

        // Прим: 0'000_1000 - следующие 8 + 1 байт будут неповторяющимися

        // Таким образом можно сохранить [1; 128] байт, поэтому
        // получаем срез байтов размером до 128 байт
        List<Byte> slice;
        try {
            slice = unrepeatingBuffer.subList(0, 128);
            unrepeatingBuffer = unrepeatingBuffer.subList(128, unrepeatingBuffer.size());
        } catch (IndexOutOfBoundsException e) {
            slice = new ArrayList<>(unrepeatingBuffer);
            unrepeatingBuffer.clear();
        }

        // Записываем информационный байт
        result.add((byte) (slice.size() - 1));

        // Записываем в результат неповторяющиеся байты
        result.addAll(slice);
    }
}

public static class TableElement {
    // Сам байт
    public byte symbol;
    // Количество повторений байта
    public int amount;
    // Код в виде массива булеанов
    public List<Boolean> code;

    // конструирует объект TableElement на основе элемента symbol .
    public TableElement(byte symbol) {
        this.symbol = symbol;
        this.amount = 1;
        this.code = new ArrayList<>();
    }

    // возвращает префиксный код элемента в виде целого числа.
    int getCode() {
        // Преобразование кода в вид int
        int ans = 0;
        for (boolean v : code) {

```

```

        ans <<= 1;
        ans += v ? 1 : 0;
    }

    return ans;
}
}

/* возвращает последовательность закодированных байт на основе сообщения
 * input и префиксной таблицы table.*/
private static List<Byte> getEncodedFromTable(List<Byte> input, List<TableElement> table) {
    // Создаём буффер байтов
    List<Byte> result = new ArrayList<>();
    Map<Byte, TableElement> codes = table.stream()
        .collect(Collectors.toMap(tableElement -> tableElement.symbol, tableElement -> tableElement));

    int bit = 0;
    // Создаём буффер для побитовой работы
    BitSet bitSet = new BitSet();

    // Для каждого элемента в последовательности
    for (Byte in : input) {
        // Получаем элемент в таблице Шеннона Фано
        TableElement elementSchannon = codes.get(in);

        // Записываем элемент в буффер
        for (int i = elementSchannon.code.size() - 1; i >= 0; i--) {
            bitSet.set(bit, ((elementSchannon.getCode() >> i) & 1) == 1);
            bit++;
        }
    }

    // Так как при таком кодировании может получиться последовательность, длина которой не
    // делится на 8, записываем в конец дополняющий байт. Он начинается с 1 и дополняет нулями байт до коцна.
    if (bit % 8 == 0) {
        // Если длина последовательности битов делится на 8, дозаписываем лишний байт
        // 10000000
        bitSet.set(bit, true);
        bitSet.set(bit + 1, bit + 8, false);
    } else {
        // Иначе - дозаписываем по правилу
        bitSet.set(bit++, true);
        while (bit % 8 != 0) {
            bitSet.set(bit++, false);
        }
    }

    // Копируем данные из битового буффера в байтовый
    byte tmp = 0;
    for (int i = 0; i < bit; i++) {
        if (i % 8 == 0 && i != 0) {
            result.add(tmp);
            tmp = 0;
        }
    }
}

```



```

        tmp = (byte) (tmp * 2 + (bitSet.get(i) ? 1 : 0));
    }

    if (bit != 0)
        result.add(tmp);

    return result;
}

/* возвращает последовательность закодированных байт, полученных из
 * сообщения input алгоритмом Хаффмана.*/
public static List<Byte> getEncodedHuffman(List<Byte> input) {
    List<TableElement> table = getHuffmanTable(input);

    return getEncodedFromTable(input, table);
}

/* инициализирует префиксную таблицу для сообщения input , разбивая
 * его на сегменты по 8 бит. */
private static List<TableElement> getSegmentisedTable(List<Byte> input) {
    // Подготовим таблицу для дальнейшего использования
    List<TableElement> table = new ArrayList<>();
    for (Byte symbol : input) {
        // Ищем уникальные байты. Если байт есть - увеличиваем его кол-во в таблице,
        // иначе - добавляем новый элемент.
        Optional<TableElement> result = table.stream().filter((el) -> el.symbol == symbol).findAny();
        if (result.isPresent()) {
            result.get().amount++;
        } else {
            table.add(new TableElement(symbol));
        }
    }
}

// Сортируем таблицу по убыванию кол-ва появления символов
table.sort(Comparator.comparingInt(o -> o.amount));
Collections.reverse(table);
return table;
}

private static class HuffmanTableElement {
    TableElement self = null;
    HuffmanTableElement left = null;
    HuffmanTableElement right = null;
    int amount = 0;

    // конструирует объект HuffmanTableElement как лист со значением self .
    public HuffmanTableElement(TableElement self) {
        this.self = self;
        this.amount = self.amount;
    }

    /* конструирует объект HuffmanTableElement как узел дерева с
     * поддеревьями left и right. */

```

```

public HuffmanTableElement(HuffmanTableElement left, HuffmanTableElement right) {
    this.right = right;
    this.left = left;

    this.amount = left.amount + right.amount;
}

// возвращает префиксную таблицу.
public List<TableElement> getTableElement() {
    if (self != null) {
        return List.of(new TableElement[]{self});
    }

    return getTableElement(List.of());
}

/* возвращает префиксную таблицу обходом в глубину на основе
 * префикса prefix. */
private List<TableElement> getTableElement(List<Boolean> prefix) {
    List<TableElement> result = new ArrayList<>();

    // Проход по дереву в глубину, формируется код
    if (self == null) {
        List<Boolean> newPrefix = new ArrayList<>(prefix);
        newPrefix.add(true);
        result.addAll(left.getTableElement(newPrefix));

        newPrefix = new ArrayList<>(prefix);
        newPrefix.add(false);
        result.addAll(right.getTableElement(newPrefix));
    } else {
        self.code = prefix;
        result.add(self);
    }

    return result;
}

}

// возвращает префиксную таблицу для сообщения input.
public static List<TableElement> getHuffmanTable(List<Byte> input) {
    Queue<HuffmanTableElement> queue = new PriorityQueue<>(
        Comparator.comparingInt(o -> o.amount));
    // Создаём приоритетную очередь
    queue.addAll(getSegmentisedTable(input).stream().map(HuffmanTableElement::new).collect(Collectors.toList()));

    while (queue.size() > 1) {
        // Получаем два элемента из таблицы с наименьшими кодами, формируем новый узел дерева,
        // снова сохраняем в очередь
        HuffmanTableElement left = queue.poll();
        HuffmanTableElement right = queue.poll();
        HuffmanTableElement newElement = new HuffmanTableElement(left, right);

        queue.add(newElement);
    }
}

```

```

}

// Возвращаем итоговую таблицу
return queue.poll().getTableElement();
}
}

```

Результаты экспериментов:

ads.txt



Текст содержит случайные русские символы, цифры и т.д.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		1.896	3129
Original		1.000	5933
RLE>HUF		0.863	6873
HUF>RLE		0.267	22256

В этом случае хорошо себя показало сообщение, закодированное методом Хаффмана, так как в тексте используется не весь алфавит. Кроме того, русские символы, закодированные в UTF-8 состоят из двух байтов, первый байт достаточно часто повторяется, что позволило ещё сильнее сократить данные в кодировках. Так как в тексте мало повторяющихся подряд символов, применение RLE оказалось неоптимальным.

api-ms-win-core-datetime-l1-1-0.dll





Файл динамической библиотеки

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		1.085	21711
Original		1.000	23560
HUF>RLE		0.028	831785
RLE>HUF		0.016	1507702

Файл содержит случайные равномерно распределённые байты, поэтому кодирование по Хаффману немного уменьшило конечный результат, но не сильно. Применение RLE так же не имело смысла, так как повторяющихся байтов в файле мало, и его применение приведёт только к увеличению информации.

black.bmp

Графическое изображение, содержащее только чёрные пиксели

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF>RLE		141.183	1332
RLE>HUF		113.973	1650
HUF only		7.894	23823
Original		1.000	188056

Здесь алгоритм RLE показывает себя отлично. Алгоритмом Хаффмана можно сократить повторяющиеся чёрные пиксели, после чего снова сократить полученные байты RLE, что позволяет уменьшить размер файла в 141 раз. Использование только алгоритма Хаффмана позволяет сократить размер файла, однако только с алгоритмом RLE, убирающим повторы, можно добиться таких результатов.

img_random.bmp

Случайное изображение

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		1.019	46241
Original		1.000	47138
HUF>RLE		0.009	5456244
RLE>HUF		0.006	7995093

Алгоритм Хаффмана вырывается вперёд. Изображение больше не содержит повторяющихся цепочек байтов. Алгоритмы с применением RLE менее эффективны и приводят к увеличению объёма памяти.

img_random.png





То же изображение, но в нём уже используются алгоритм сжатия LZ77.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		1.002	25719
Original		1.000	25780
HUF>RLE		0.019	1353422
RLE>HUF		0.017	1511092

Повторное сжатие не дало лучших результатов и привело лишь к увеличению объёма памяти. Только алгоритм Хаффмана позволил немного уменьшить изображение.

laughter_of_terror_ru.txt

Последовательность из нескольких русских символов, пробелов и переносов строки в UTF-8.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		2.785	158
HUF>RLE		2.785	158
RLE>HUF		1.014	434
Original		1.000	440

Последовательность состоит из ограниченного алфавита, повторяющихся цепочек нет. Именно поэтому кодирование только при помощи Хаффмана снова эффективно. Последовательность содержит мало повторяющихся цепочек, поэтому применение RLE эффективно только в конце, когда символы уже преобразованы алгоритмом Хаффмана.

lorem_ipsum.txt




Последовательность из латинских символов.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		1.862	5798
Original		1.000	10797
HUF>RLE		0.295	36554
RLE>HUF		0.077	139371

В обычном тексте символы не будут равномерно распределены, поэтому алгоритм Хаффмана снова вырывается вперёд. Повторяющиеся байты в обычном тексте встречаются не так часто, следовательно применение RLE не сильно эффективно.

metal_alphabet_lyrics.txt





Последовательность из повторяющихся латинских символов, разделённых переводами строки.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
RLE>HUF		10.754	61
HUF only		1.652	397
Original		1.000	656
HUF>RLE		0.471	1394

Самым эффективным оказалось применение RLE и Хаффмана. RLE сокращает повторяющиеся байты, а Хаффмана сокращает переводы, выраженные двумя байтами.

metal_alphabet_lyrics_ru.txt

Последовательность из повторяющихся русских символов, разделённых переводами строки.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF>RLE		2.543	704
HUF only		2.180	821
RLE>HUF		2.136	838
Original		1.000	1790

В данном случае буквы будут занимать уже два байта, поэтому эффективней окажется алгоритм, который сначала применяет алгоритм Хаффмана, и затем - RLE. Одним из возможных решений для увеличения эффективности RLE является расширение сегментизации до 2 байтов.

noise.bmp




Изображение из случайных равномерно распределённых байтов

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
HUF only		3.281	57313
RLE>HUF		2.190	85878
Original		1.000	188056
HUF>RLE		0.029	6416097

Результат оказался неожиданным, алгоритм Хаффмана и Шеннона-Фано позволил уменьшить размер изображения в 3 раза. Даже совместное применение RLE и Хаффмана оказалось эффективным.

scream_of_terror_en.txt

Последовательность из латинской буквы, пробелов и переносов строки в UTF-8.

Название	Шкала	Коэффициент сжатия	Размер (в байтах)
RLE>HUF		14.120	25
HUF>RLE		9.051	39
HUF only		6.537	54
Original		1.000	353

Эффективней всего оказалось применить алгоритм RLE, который сократит повторяющиеся символы и затем алгоритм Хаффмана.

Полный исходный код эксперимента:

Main.java (https://github.com/IAmProgrammist/information_theory/blob/main/src/main/java/rchat/info/lab3/Main.java)

Coder.java (https://github.com/IAmProgrammist/information_theory/blob/main/src/main/java/rchat/info/libs/Coder.java)

Файлы эксперимента:

Ссылка (https://github.com/IAmProgrammist/information_theory/tree/main/src/assets/lab3)

Вывод: совместные алгоритмы эффективны при условии наличия особенностей, в которых оба эти алгоритма эффективны - большое количество повторяющихся строчек и малый алфавит. Их совместное применение позволит существенно уменьшить размер файла. Такие методы будут полезны, например, при кодировании очень тёмных или очень светлых изображений, где очень часто встречаются повторяющиеся подряд пиксели. Не всегда самыми эффективными, но всегда уменьшающими размер оказался алгоритмы Хаффмана без применения RLE, его можно использовать когда условие наличия множества повторяющихся цепочек не выполняется.