Теория кодирования и сжатия информации
Лабораторная работа №1

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию текстового файла используя алгоритм Хаффмана. Программы архивации и разархивации должны быть представлены отдельно и работать независимо друг от друга. Определить для данного шифра характеристики 1 (коэффициент сжатия) и 2 (скорость сжатия). К работе необходимо прикрепить отчет и программный проект.

2 Алгоритм

Алгоритм Хаффмана заключается в создании нового кода для каждого встречающегося в тексте символа на основе частоты встречи этого символа.

Алгоритм состоит из следующих шагов:

1. Вычислить частоты всех встретившихся символов;
2. Построить очередь с приоритетом на основе полученных частот;
3. Построить дерево с помощью очереди с приоритетом;
4. На основе дерева вычислить код для каждого символа.

3 Тестирование

Для проверки программы были использованы тестовые тексты 1 (рис. 1) и 6 (рис. 2). Можно заметить, что после распаковки архива полученный файл совпадает с исходным (проверка с помощью утилиты diff). Также можно заметить, что для файлов малого размера архив увеличивает их размер за счёт метаданных.

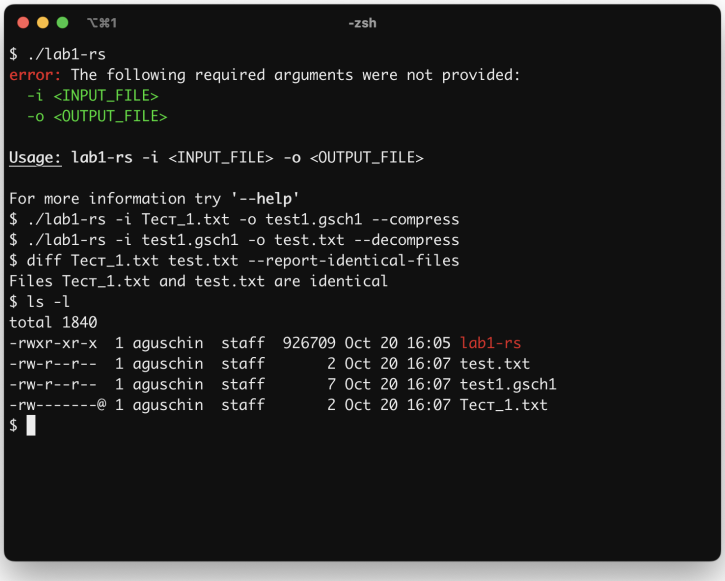

Рис. 1: Сжатие текста Тест_1.txt

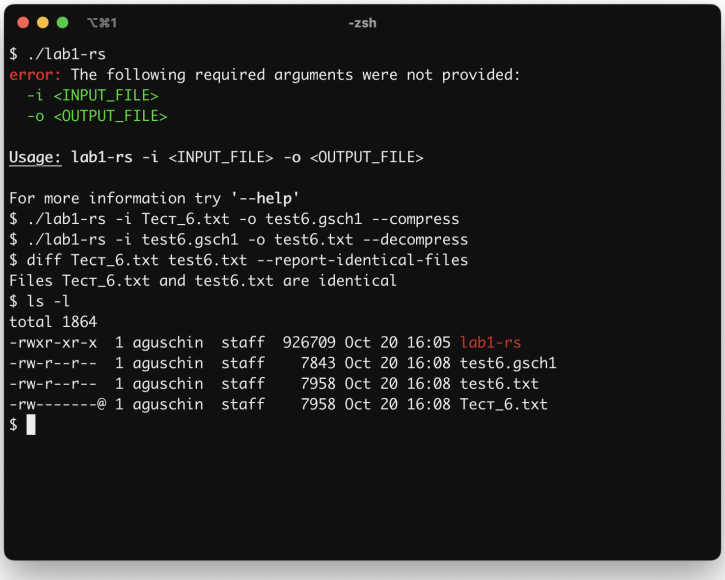

Рис. 2: Сжатие текста Тест_6.txt

4 Вычисленные характеристики

4.1 Характеристика 1 (Коэффициент сжатия)

Результаты применения программы к каждому из тестовых текстовых файлов занесены в таблицу 1. Можно заметить, что в некоторых случаях архивация с помощью алгоритма Хаффмана является неэффективной.

Название	Исходный размер, байт	Сжатый размер, байт	Коэффициент
Тест_1.txt	2	7	0.28571
Тест_2.txt	33	92	0.3587
Тест_3.txt	2739	1998	1.37087
Тест_4.txt	330	48	6.875
Тест_5.txt	59	168	0.35119
Тест_6.txt	7958	7843	1.01466
Тест_7.txt	138245	120488	1.14738
Тест_8.txt	574426	516945	1.11119
Тест_9.txt	2752	349	7.88539
Тест_10.txt	2814	368	7.64674

Таблица 1: результаты тестирования

4.2 Характеристика 2 (Скорость сжатия)

Для тестирования скорости сжатия использовался произвольный двоичный файл размера 76450438 байт (≈73 мегабайта). В результате пяти последовательных запусков, среднее время запаковки файла составило 4.2 секунды, среднее время распаковки составило 4.7 секунды.

Таким образом, средняя скорость сжатия составила 17.35924 Мбайт в секунду, а средняя скорость разжатия составила 15.51251 Мбайт в секунду.

5 Реализация

Программа реализована на языке программирования Rust с использованием библиотеки clap для чтения параметров командной строки. Сборка производится с помощью программы cargo, поставляющейся вместе с языком.

5.1 Содержимое файла priority_queue.rs

```rust
use std::cmp::Ordering;
use std::collections::BinaryHeap;

#[derive(Debug)]
pub struct Weighted<T> {
    pub weight: u32,
    pub value: T,
}

impl<T> PartialEq for Weighted<T> {
    fn eq(&self, other: &Self) -> bool {
```

```
        self.weight == other.weight
    }
}

impl<T> Eq for Weighted<T> {}

impl<T> PartialOrd for Weighted<T> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl<T> Ord for Weighted<T> {
    fn cmp(&self, other: &Self) -> Ordering {
        self.weight.cmp(&other.weight).reverse()
    }
}

#[derive(Debug)]
pub struct PriorityQueue<T> {
    heap: BinaryHeap<Weighted<T>>,
}

impl<T> PriorityQueue<T> {
    pub fn new() -> Self {
        PriorityQueue {
            heap: BinaryHeap::new(),
        }
    }

    pub fn insert(&mut self, priority: u32, value: T) {
        self.heap.push(Weighted {
            weight: priority,
            value,
        });
    }

    pub fn len(&self) -> usize {
        self.heap.len()
    }

    pub fn pop(&mut self) -> Option<Weighted<T>> {
        Some(self.heap.pop()?)
    }
}
```

5.2 Содержимое файла huffman.rs

```
use crate::priority_queue::PriorityQueue;
use std::collections::HashMap;

#[derive(Clone, Copy, Debug)]
enum Bit {
    Zero,
```

```
    One,
}
impl Bit {
    fn from_u8(x: u8) -> Bit {
        if x == 0 {
            Bit::Zero
        } else {
            Bit::One
        }
    }

    fn to_u8(x: &Bit) -> u8 {
        match *x {
            Bit::Zero => 0,
            Bit::One => 1,
        }
    }
}

#[derive(Debug)]
enum HuffmanTree<T>
where
    T: std::cmp::Eq + std::hash::Hash + Copy,
{
    Leaf(T),
    Node(Box<HuffmanTree<T>>, Box<HuffmanTree<T>>),
}

impl<T: std::cmp::Eq + std::hash::Hash + Copy> HuffmanTree<T> {
    fn from_queue(mut queue: PriorityQueue<T>) -> Self {
        let mut huf_queue = PriorityQueue::new();
        while let Some(p) = queue.pop() {
            huf_queue.insert(p.weight,
↪   Box::new(HuffmanTree::Leaf(p.value)));
        }
        assert!(huf_queue.len() > 0);
        while huf_queue.len() > 1 {
            let v1 = huf_queue.pop().unwrap();
            let v2 = huf_queue.pop().unwrap();
            huf_queue.insert(
                v1.weight + v2.weight,
                Box::new(HuffmanTree::Node(v1.value, v2.value)),
            );
        }
        return *huf_queue.pop().unwrap().value;
    }

    fn dfs(tree: &HuffmanTree<T>, mut acc: Vec<Bit>, code: &mut
↪   HashMap<T, Vec<Bit>>) -> Vec<Bit> {
        match tree {
            HuffmanTree::Leaf(val) => {
                code.insert(*val, acc.clone());
            }
            HuffmanTree::Node(left, right) => {
```

```
                acc.push(Bit::Zero);
                acc = HuffmanTree::dfs(left, acc, code);
                acc.pop();
                acc.push(Bit::One);
                acc = HuffmanTree::dfs(right, acc, code);
                acc.pop();
            }
        };
        return acc;
    }

    fn get_code(&self) -> HashMap<T, Vec<Bit>> {
        let mut code = HashMap::new();
        if let HuffmanTree::Leaf(val) = &self {
            code.insert(*val, vec![Bit::One]);
        } else {
            HuffmanTree::dfs(self, Vec::new(), &mut code);
        }
        return code;
    }
}

fn get_weights(data: &Vec<u8>) -> HashMap<u8, u32> {
    let mut freq = HashMap::new();
    data.iter().for_each(|byte| {
        *freq.entry(*byte).or_insert(1) += 1;
    });
    let mut freq: Vec<(u8, u32)> = freq.drain().collect();
    freq.sort();
    let mut weights = HashMap::new();
    freq.iter().enumerate().for_each(|(i, (byte, _))| {
        weights.insert(*byte, i as u32);
    });
    return weights;
}

#[derive(Debug)]
struct Metadata {
    weights: HashMap<u8, u32>,
    tree: HuffmanTree<u8>,
    code: HashMap<u8, Vec<Bit>>,
    remainder: u8,
}

impl Metadata {
    fn compute(data: &Vec<u8>) -> Self {
        let weights = get_weights(data);
        let mut queue = PriorityQueue::new();
        for (value, priority) in weights.iter() {
            queue.insert(*priority, *value);
        }
        let tree = HuffmanTree::from_queue(queue);
        let code = tree.get_code();

```

```
113             return Self {
114                 weights,
115                 tree,
116                 code,
117                 remainder: 0,
118             };
119         }
120 
121         fn load(data: &Vec<u8>) -> Self {
122             let remainder = data[0];
123             let dict_len = data[1] as usize + 1;
124             let mut weights = HashMap::new();
125             for i in 0..dict_len {
126                 weights.insert(data[2 + 2 * i], data[2 + 2 * i + 1] as u32);
127             }
128 
129             let mut queue = PriorityQueue::new();
130             for (value, priority) in weights.iter() {
131                 queue.insert(*priority, *value);
132             }
133             let tree = HuffmanTree::from_queue(queue);
134             let code = tree.get_code();
135 
136             return Self {
137                 weights,
138                 tree,
139                 code,
140                 remainder,
141             };
142         }
143 
144         fn dump(&self) -> Vec<u8> {
145             let mut result = Vec::new();
146 
147             result.push(self.remainder);
148             result.push((self.weights.len() - 1) as u8);
149             for (byte, weight) in &self.weights {
150                 result.push(*byte);
151                 result.push(*weight as u8);
152             }
153 
154             return result;
155         }
156     }
157 
158     struct BitWriter {
159         buffer: Vec<Bit>,
160         remainder: u8,
161         result: Vec<u8>,
162     }
163 
164     impl BitWriter {
165         fn new() -> Self {
166             Self {
```

```
            buffer: Vec::new(),
            remainder: 0,
            result: Vec::new(),
        }
    }

    fn dump_byte(&mut self) {
        let mut byte = 0_u8;
        self.buffer
            .iter()
            .map(Bit::to_u8)
            .enumerate()
            .for_each(|(i, bit)| byte |= bit << i);
        self.result.push(byte);
        self.buffer.clear();
    }

    fn write_bit(&mut self, bit: Bit) {
        self.buffer.push(bit);
        if self.buffer.len() == 8 {
            self.dump_byte();
        }
    }

    fn write_bits(&mut self, bits: &Vec<Bit>) {
        bits.iter().for_each(|bit| self.write_bit(*bit));
    }

    fn finish(&mut self) {
        let remainder = 8 - self.buffer.len() as u8 % 8;
        if remainder != 0 {
            self.dump_byte();
        }
        self.remainder = remainder;
    }
}

pub fn compress(data: &Vec<u8>) -> Vec<u8> {
    let mut result = Vec::new();
    let mut metadata = Metadata::compute(data);

    let mut writer = BitWriter::new();
    data.iter().for_each(|byte| {
        let bits = metadata.code.get(byte).unwrap();
        writer.write_bits(&bits);
    });
    writer.finish();
    metadata.remainder = writer.remainder;

    let md_dump = metadata.dump();
    md_dump.iter().for_each(|byte| result.push(*byte));
    writer.result.iter().for_each(|byte| result.push(*byte));

    return result;
```

```rust
}

struct BitReader<'a> {
    data: &'a Vec<u8>,
    metadata: &'a Metadata,
    buffer: Vec<Bit>,
    ptr: usize,
}

impl<'a> BitReader<'a> {
    fn new(data: &'a Vec<u8>, metadata: &'a Metadata) -> Self {
        Self {
            data,
            metadata,
            buffer: Vec::new(),
            ptr: 0,
        }
    }

    fn read_byte(&mut self) {
        if let Some(byte) = self.data.get(self.ptr) {
            for i in 0..=7 {
                let bit = byte & (1 << i);
                self.buffer.push(Bit::from_u8(bit));
            }
            if self.ptr == self.data.len() - 1 {
                for _ in 0..self.metadata.remainder {
                    self.buffer.pop();
                }
            }
            self.buffer.reverse();
            self.ptr += 1;
        }
    }

    fn read_bit(&mut self) -> Option<Bit> {
        if self.buffer.len() == 0 {
            self.read_byte();
        }
        self.buffer.pop()
    }
}

pub fn decompress(archive: &Vec<u8>) -> Vec<u8> {
    let mut result = Vec::new();
    let metadata = Metadata::load(archive);

    let data = archive[2 + metadata.weights.len() * 2..].to_vec();
    let mut reader = BitReader::new(&data, &metadata);

    let mut state = &metadata.tree;
    while let Some(bit) = reader.read_bit() {
        if let HuffmanTree::Node(left, right) = state {
            match bit {
```

```
                Bit::Zero => state = left,
                Bit::One => state = right,
            }
        }

        if let HuffmanTree::Leaf(byte) = state {
            result.push(*byte);
            state = &metadata.tree;
        }
    }

    return result;
}
```

5.3 Содержимое файла main.rs

```
mod huffman;
mod priority_queue;
use clap::Parser;
use std::fs::File;
use std::io::{Error, ErrorKind, Read, Write};
use std::path::PathBuf;

#[derive(Parser)]
struct Cli {
    #[arg(short)]
    input_file: PathBuf,

    #[arg(short)]
    output_file: PathBuf,

    #[arg(long, default_value_t = true)]
    compress: bool,

    #[arg(long, default_value_t = false)]
    decompress: bool,
}

fn run_decompressor(cli: &Cli) -> Result<(), Error> {
    let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
    let mut archive = Vec::new();
    input_f.read_to_end(&mut archive)?;
    let data = huffman::decompress(&archive);

    let mut output_f = File::create(cli.output_file.to_str().unwrap())?;
    output_f.write_all(&data)?;

    return Ok(());
}

fn run_compressor(cli: &Cli) -> Result<(), Error> {
    let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
    let mut data = Vec::new();
    input_f.read_to_end(&mut data)?;
```

```rust
    let archive = huffman::compress(&data);

    let mut output_f = File::create(cli.output_file.to_str().unwrap())?;
    output_f.write_all(&archive)?;

    return Ok(());
}

fn main() -> std::io::Result<()> {
    let cli = Cli::parse();

    let result = if cli.decompress {
        run_decompressor(&cli)
    } else {
        run_compressor(&cli)
    };

    if let Err(error) = result {
        match error.kind() {
            ErrorKind::NotFound => println!("Указанный файл не найден"),
            ErrorKind::AlreadyExists => println!("Указанный файл уже существует"),
            _ => println!("Произошла непредвиденная ошибка"),
        };
    }

    return Ok(());
}
```

5.4 Содержимое файла Cargo.toml

```toml
[package]
name = "lab1-rs"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = { version = "4.0.17", features = ["derive"] }
```