Теория кодирования и сжатия информации
Лабораторная работа №5

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию текстового файла используя алгоритм «стопка книг». 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 «стопка книг» заключается в динамическом назначении более эффективного кода для чаще встречающихся символов текста.

Алгоритм состоит из следующих шагов:

1. Составить алфавит встречающихся символов в исходном тексте;
2. Создать код Хаффмана для индексов алфавита;
3. Для прочитанного символа найти индекс в алфавите;
4. Этот символ закодировать кодом индекса;
5. Переместить символ внутри алфавита на позицию с меньшим кодом.

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.

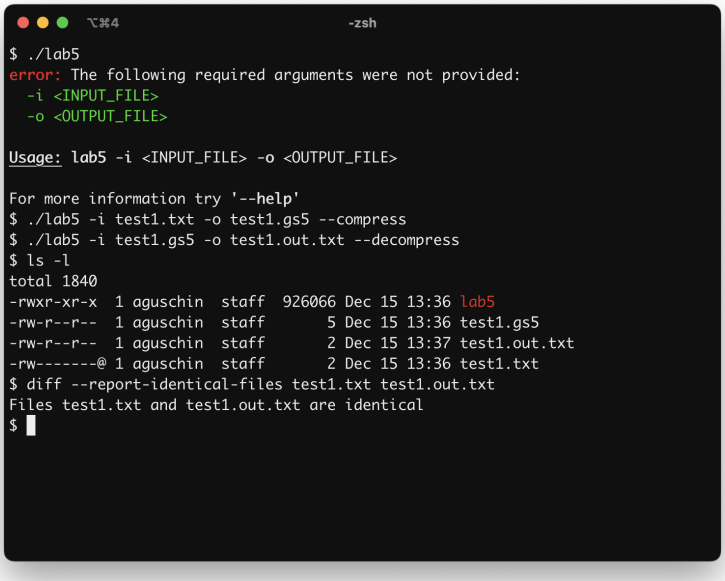

Рис. 1: Сжатие текста Тест_1.txt

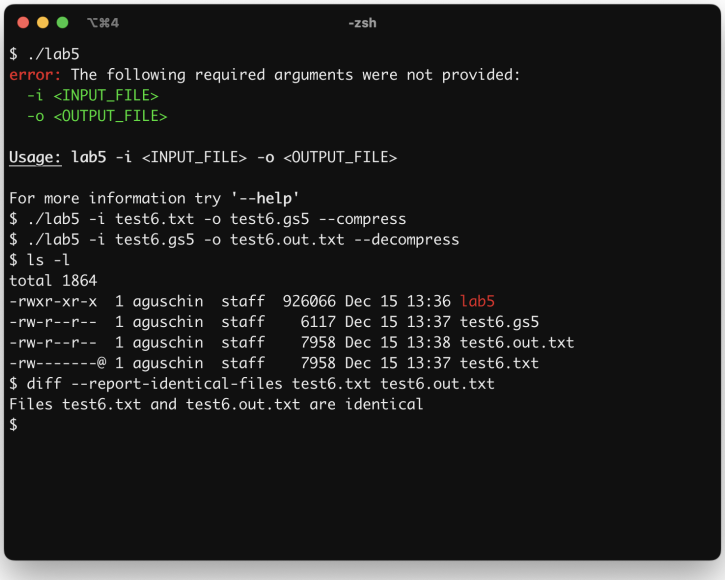

Рис. 2: Сжатие текста Тест_6.txt

4 Вычисленные характеристики

4.1 Характеристика 1 (Коэффициент сжатия)

Результаты применения программы к каждому из тестовых текстовых файлов занесены в таблицу 1.

Название	Исходный размер, байт	Сжатый размер, байт	Коэффициент
Тест_1.txt	2	5	0.4
Тест_2.txt	33	73	0.45205
Тест_3.txt	2739	3117	0.87873
Тест_4.txt	330	46	7.17391
Тест_5.txt	59	119	0.4958
Тест_6.txt	7958	6117	1.30096
Тест_7.txt	138245	104540	1.32241
Тест_8.txt	574426	404975	1.41842
Тест_9.txt	2752	348	7.90805
Тест_10.txt	2814	369	7.62602

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, HashSet};

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

```rust
}
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: Eq + Ord + Copy + std::hash::Hash,
{
    Leaf(T),
    Node(Box<HuffmanTree<T>>, Box<HuffmanTree<T>>),
}

impl<T: Eq + Ord + Copy + std::hash::Hash> 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;
    }
}

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

impl Metadata {
    fn compute(data: &Vec<u8>) -> Self {
        let mut alphabet = HashSet::new();
        data.iter().for_each(|v| {
            alphabet.insert(*v);
        });
        let mut alphabet = Vec::from_iter(alphabet);
        alphabet.sort();

        let mut queue = PriorityQueue::new();
        for value in 1..=alphabet.len() as u8 {
            queue.insert(value as u32, value);
        }
        let tree = HuffmanTree::from_queue(queue);
        let code = tree.get_code();

        return Self {
            tree,
            alphabet,
            code,
            remainder: 0,
        };
    }

    fn load(data: &Vec<u8>) -> Self {
```

```
        let remainder = data[0];
        let dict_len = data[1] as usize + 1;
        let mut alphabet = Vec::new();
        for i in 0..dict_len {
            alphabet.push(data[2 + i]);
        }
        alphabet.sort();

        let mut queue = PriorityQueue::new();
        for value in 1..=alphabet.len() as u8 {
            queue.insert(value as u32, value);
        }
        let tree = HuffmanTree::from_queue(queue);
        let code = tree.get_code();

        return Self {
            tree,
            alphabet,
            code,
            remainder,
        };
    }

    fn dump(&self) -> Vec<u8> {
        let mut result = Vec::new();

        result.push(self.remainder);
        result.push((self.alphabet.len() - 1) as u8);
        for ch in &self.alphabet {
            result.push(*ch);
        }

        return result;
    }
}

struct BitWriter {
    buffer: Vec<Bit>,
    remainder: u8,
    result: Vec<u8>,
}

impl BitWriter {
    fn new() -> Self {
        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 alphabet = metadata.alphabet.clone();
    let mut writer = BitWriter::new();
    data.iter().for_each(|&byte| {
        let pos = alphabet.iter().position(|&x| x == byte).unwrap();
        let bits = metadata.code.get(&(pos as u8 + 1)).unwrap();
        alphabet.remove(pos);
        alphabet.push(byte);
        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;
}

struct BitReader<'a> {
    data: &'a Vec<u8>,
    metadata: &'a Metadata,
```

```
222     buffer: Vec<Bit>,
223     ptr: usize,
224 }
225 
226 impl<'a> BitReader<'a> {
227     fn new(data: &'a Vec<u8>, metadata: &'a Metadata) -> Self {
228         Self {
229             data,
230             metadata,
231             buffer: Vec::new(),
232             ptr: 0,
233         }
234     }
235 
236     fn read_byte(&mut self) {
237         if let Some(byte) = self.data.get(self.ptr) {
238             for i in 0..=7 {
239                 let bit = byte & (1 << i);
240                 self.buffer.push(Bit::from_u8(bit));
241             }
242             if self.ptr == self.data.len() - 1 {
243                 for _ in 0..self.metadata.remainder {
244                     self.buffer.pop();
245                 }
246             }
247             self.buffer.reverse();
248             self.ptr += 1;
249         }
250     }
251 
252     fn read_bit(&mut self) -> Option<Bit> {
253         if self.buffer.len() == 0 {
254             self.read_byte();
255         }
256         self.buffer.pop()
257     }
258 }
259 
260 pub fn decompress(archive: &Vec<u8>) -> Vec<u8> {
261     let mut result = Vec::new();
262     let metadata = Metadata::load(archive);
263 
264     let data = archive[2 + metadata.alphabet.len()..].to_vec();
265     let mut reader = BitReader::new(&data, &metadata);
266 
267     let mut alphabet = metadata.alphabet.clone();
268     let mut state = &metadata.tree;
269     while let Some(bit) = reader.read_bit() {
270         if let HuffmanTree::Node(left, right) = state {
271             match bit {
272                 Bit::Zero => state = left,
273                 Bit::One => state = right,
274             }
275         }
```

```
276
277             if let HuffmanTree::Leaf(pos) = state {
278                 let byte = alphabet.remove((*pos - 1) as usize);
279                 alphabet.push(byte);
280                 result.push(byte);
281                 state = &metadata.tree;
282             }
283         }
284
285         return result;
286     }
```

5.3 Содержимое файла main.rs

```
1   mod huffman;
2   mod priority_queue;
3   use clap::Parser;
4   use std::fs::File;
5   use std::io::{Error, ErrorKind, Read, Write};
6   use std::path::PathBuf;
7
8   #[derive(Parser)]
9   struct Cli {
10      #[arg(short)]
11      input_file: PathBuf,
12
13      #[arg(short)]
14      output_file: PathBuf,
15
16      #[arg(long, default_value_t = true)]
17      compress: bool,
18
19      #[arg(long, default_value_t = false)]
20      decompress: bool,
21  }
22
23  fn run_decompressor(cli: &Cli) -> Result<(), Error> {
24      let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
25      let mut archive = Vec::new();
26      input_f.read_to_end(&mut archive)?;
27      let data = huffman::decompress(&archive);
28
29      let mut output_f = File::create(cli.output_file.to_str().unwrap())?;
30      output_f.write_all(&data)?;
31
32      return Ok(());
33  }
34
35  fn run_compressor(cli: &Cli) -> Result<(), Error> {
36      let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
37      let mut data = Vec::new();
38      input_f.read_to_end(&mut data)?;
39      let archive = huffman::compress(&data);
40
```

```rust
    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"),
        };
    }

    Ok(())
}
```

5.4 Содержимое файла Cargo.toml

```toml
[package]
name = "lab5"
version = "0.1.0"
edition = "2021"

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