Теория кодирования и сжатия информации
Лабораторная работа №6

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию текстового файла используя алгоритм Гилберта-Мура. 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 Гилберта-Мура 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 p_i всех встретившихся символов;
2. Отсортировать полученный список по невозрастанию;
3. Вычислить префиксные суммы q_i для списка частот;
4. Вычислить $\sigma_i = q_i + p_i/2$
5. Перевести σ_i из десятичной системы счисления в двоичную;
6. В качестве кода для символа i необходимо использовать первые $\lceil -\log_2 p_i \rceil + 1$ двоичных цифр после запятой из σ_i.

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.

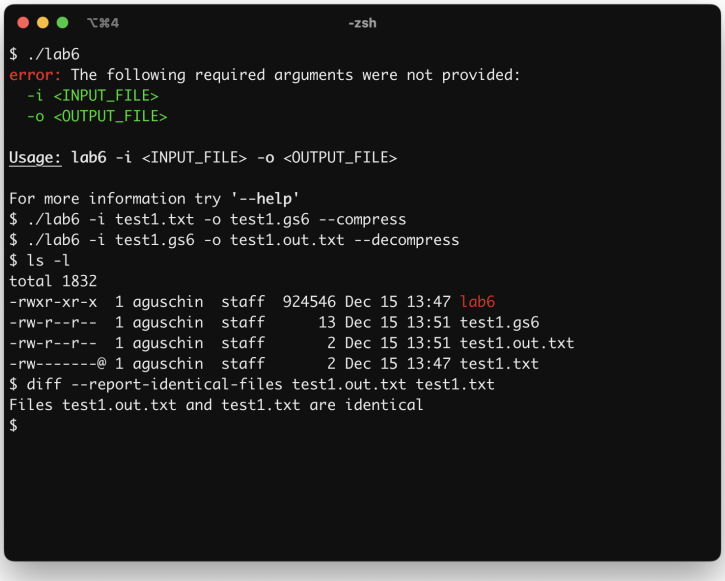

Рис. 1: Сжатие текста Тест_1.txt

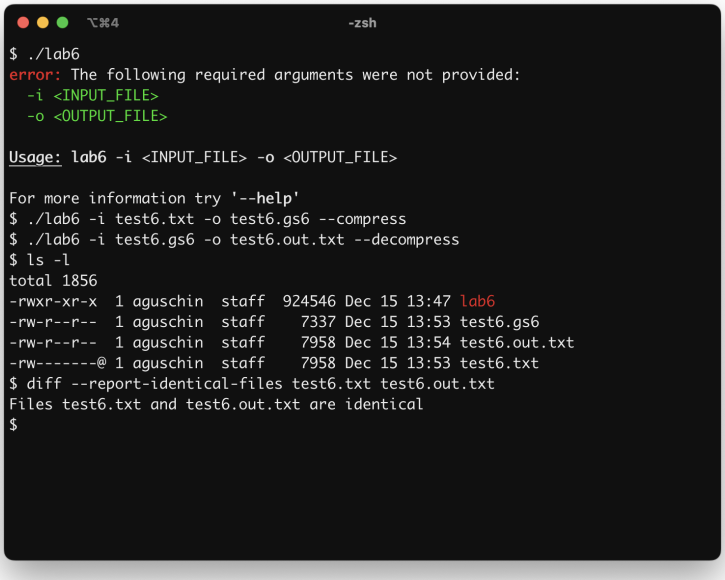

Рис. 2: Сжатие текста Тест_6.txt

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	13	0.15385
Тест_2.txt	33	196	0.16837
Тест_3.txt	2739	2564	1.06825
Тест_4.txt	330	95	3.47368
Тест_5.txt	59	349	0.16905
Тест_6.txt	7958	7337	1.08464
Тест_7.txt	138245	108673	1.27212
Тест_8.txt	574426	435482	1.31906
Тест_9.txt	2752	352	7.81818
Тест_10.txt	2814	768	3.66406

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 3.68 секунд, среднее время распаковки составило 180 секунд.

Таким образом, средняя скорость сжатия составила 19.81218 Мбайт в секунду, а средняя скорость разжатия составила 0.40505 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 gilbert_moore.rs

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

#[derive(Debug, Clone)]
pub struct Weighted<T> {
    pub weight: f32,
    pub value: T,
}

#[derive(Clone, Copy, Debug, PartialEq)]
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,
        }
    }
}

fn get_probabilities(data: &Vec<u8>) -> Vec<Weighted<u8>> {
    let mut counts = HashMap::new();
    data.iter().for_each(|byte| {
        *counts.entry(*byte).or_insert(0) += 1;
    });
    let mut counts: Vec<(u8, u32)> = counts.drain().collect();
    counts.sort();

    let total = data.len() as f32;
    let mut weights = Vec::new();
    counts.iter().for_each(|(byte, count)| {
        weights.push(Weighted {
            value: *byte,
            weight: *count as f32 / total,
        });
    });
    return weights;
}

fn prefix_sum(weights: &Vec<Weighted<u8>>) -> Vec<f32> {
    let mut pf = Vec::new();
    for i in 0..weights.len() {
        if i == 0 {
            pf.push(0f32);
        } else {
            pf.push(weights[i - 1].weight + pf[i - 1]);
        }
    }
    return pf;
}

fn get_code(probabilities: &Vec<Weighted<u8>>) -> HashMap<u8, Vec<Bit>> {
    let mut code = HashMap::new();

    let mut ps = probabilities.clone();
```

```
    ps.sort_by(|a, b|
↪   a.weight.partial_cmp(&b.weight).unwrap_or(Ordering::Equal));
    ps.reverse();
    let prefix = prefix_sum(&ps);

    for i in 0..ps.len() {
        let l = (-ps[i].weight.log2()).ceil() as i32 + 1;
        if l == 1 {
            code.insert(ps[i].value, vec![Bit::Zero]);
        } else {
            let sigma = prefix[i] + ps[i].weight as f32 / 2f32;
            let bits = get_bits(sigma, l);
            code.insert(ps[i].value, bits);
        }
    }

    return code;
}

fn get_bits(x: f32, count: i32) -> Vec<Bit> {
    let mut bits = Vec::new();

    let mut tmp = x;
    for _ in 0..count {
        tmp *= 2f32;
        if tmp >= 1f32 {
            bits.push(Bit::One);
            tmp -= 1f32;
        } else {
            bits.push(Bit::Zero);
        }
    }

    return bits;
}

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

impl Metadata {
    fn compute(data: &Vec<u8>) -> Self {
        let probs = get_probabilities(&data);
        let code = get_code(&probs);

        return Self {
            probabilities: probs,
            code,
            remainder: 0,
        };
    }
```

```
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 probabilities = Vec::new();
125         for i in 0..dict_len {
126             let pstart = 2 + 5 * i + 1;
127             let prob = [
128                 data[pstart + 0],
129                 data[pstart + 1],
130                 data[pstart + 2],
131                 data[pstart + 3],
132             ];
133             let prob: f32 = unsafe { std::mem::transmute(prob) };
134             probabilities.push(Weighted {
135                 value: data[2 + 5 * i],
136                 weight: prob,
137             });
138         }
139         let code = get_code(&probabilities);
140
141         return Self {
142             probabilities,
143             code,
144             remainder,
145         };
146     }
147
148     fn dump(&self) -> Vec<u8> {
149         let mut result = Vec::new();
150
151         result.push(self.remainder);
152         result.push((self.probabilities.len() - 1) as u8);
153         for p in &self.probabilities {
154             result.push(p.value);
155             let bweight: [u8; 4] = unsafe { std::mem::transmute(p.weight)
    ↪ };
156             for b in bweight {
157                 result.push(b);
158             }
159         }
160
161         return result;
162     }
163 }
164
165 struct BitWriter {
166     buffer: Vec<Bit>,
167     remainder: u8,
168     result: Vec<u8>,
169 }
170
171 impl BitWriter {
172     fn new() -> Self {
```

```
173             Self {
174                 buffer: Vec::new(),
175                 remainder: 0,
176                 result: Vec::new(),
177             }
178         }
179
180         fn dump_byte(&mut self) {
181             let mut byte = 0_u8;
182             self.buffer
183                 .iter()
184                 .map(Bit::to_u8)
185                 .enumerate()
186                 .for_each(|(i, bit)| byte |= bit << i);
187             self.result.push(byte);
188             self.buffer.clear();
189         }
190
191         fn write_bit(&mut self, bit: Bit) {
192             self.buffer.push(bit);
193             if self.buffer.len() == 8 {
194                 self.dump_byte();
195             }
196         }
197
198         fn write_bits(&mut self, bits: &Vec<Bit>) {
199             bits.iter().for_each(|bit| self.write_bit(*bit));
200         }
201
202         fn finish(&mut self) {
203             let remainder = 8 - self.buffer.len() as u8 % 8;
204             if remainder != 0 {
205                 self.dump_byte();
206             }
207             self.remainder = remainder;
208         }
209     }
210
211     pub fn compress(data: &Vec<u8>) -> Vec<u8> {
212         let mut result = Vec::new();
213         let mut metadata = Metadata::compute(data);
214
215         let mut writer = BitWriter::new();
216         data.iter().for_each(|byte| {
217             let bits = metadata.code.get(byte).unwrap();
218             writer.write_bits(&bits);
219         });
220         writer.finish();
221         metadata.remainder = writer.remainder;
222
223         let md_dump = metadata.dump();
224         md_dump.iter().for_each(|byte| result.push(*byte));
225         writer.result.iter().for_each(|byte| result.push(*byte));
226
```

```
227     return result;
228 }
229 
230 struct BitReader<'a> {
231     data: &'a Vec<u8>,
232     metadata: &'a Metadata,
233     buffer: Vec<Bit>,
234     ptr: usize,
235 }
236 
237 impl<'a> BitReader<'a> {
238     fn new(data: &'a Vec<u8>, metadata: &'a Metadata) -> Self {
239         Self {
240             data,
241             metadata,
242             buffer: Vec::new(),
243             ptr: 0,
244         }
245     }
246 
247     fn read_byte(&mut self) {
248         if let Some(byte) = self.data.get(self.ptr) {
249             for i in 0..=7 {
250                 let bit = byte & (1 << i);
251                 self.buffer.push(Bit::from_u8(bit));
252             }
253             if self.ptr == self.data.len() - 1 {
254                 for _ in 0..self.metadata.remainder {
255                     self.buffer.pop();
256                 }
257             }
258             self.buffer.reverse();
259             self.ptr += 1;
260         }
261     }
262 
263     fn read_bit(&mut self) -> Option<Bit> {
264         if self.buffer.len() == 0 {
265             self.read_byte();
266         }
267         self.buffer.pop()
268     }
269 }
270 
271 pub fn decompress(archive: &Vec<u8>) -> Vec<u8> {
272     let mut result = Vec::new();
273     let metadata = Metadata::load(archive);
274 
275     let data = archive[2 + metadata.probabilities.len() * 5..].to_vec();
276     let mut reader = BitReader::new(&data, &metadata);
277 
278     let mut run = Vec::new();
279     while let Some(bit) = reader.read_bit() {
280         run.push(bit);
```

```
        'checker: for p in &metadata.code {
            if p.1.len() != run.len() {
                continue;
            }
            for i in 0..run.len() {
                if p.1[i] != run[i] {
                    continue 'checker;
                }
            }
            result.push(*p.0);
            run.clear();
            break;
        }
    }

    return result;
}
```

5.2 Содержимое файла main.rs

```
mod gilbert_moore;
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 = gilbert_moore::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> {
```

```rust
    let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
    let mut data = Vec::new();
    input_f.read_to_end(&mut data)?;
    let archive = gilbert_moore::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"),
        };
    }

    Ok(())
}
```

5.3 Содержимое файла Cargo.toml

```toml
[package]
name = "lab6"
version = "0.1.0"
edition = "2021"

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