Теория кодирования и сжатия информации
Лабораторная работа №2

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию текстового файла используя алгоритм Фано. 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 Фано 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. Отсортировать полученный список по невозрастанию;
3. Подразбивать список на две части так, чтобы суммы вероятностей в этих частях были примерно равны;
4. На основе попадания в левую или правую часть подразбивки элементы переносятся в левую или правую ветвь дерева;
5. 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.

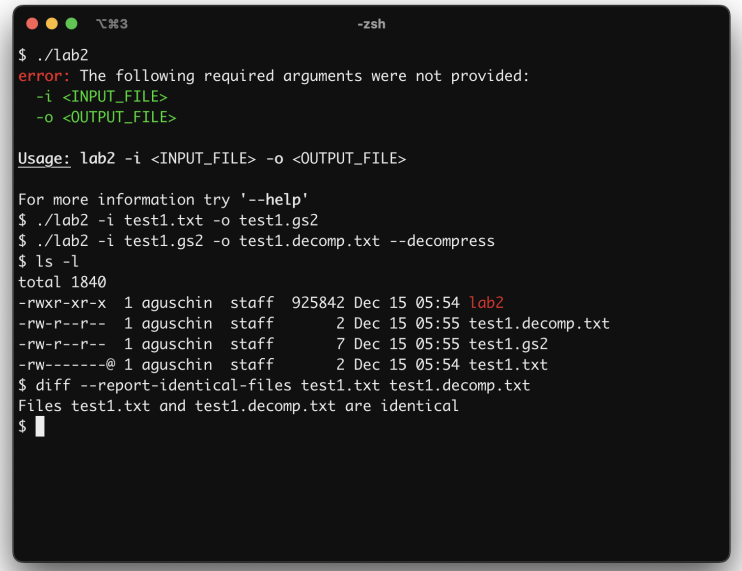

Рис. 1: Сжатие текста Тест_1.txt

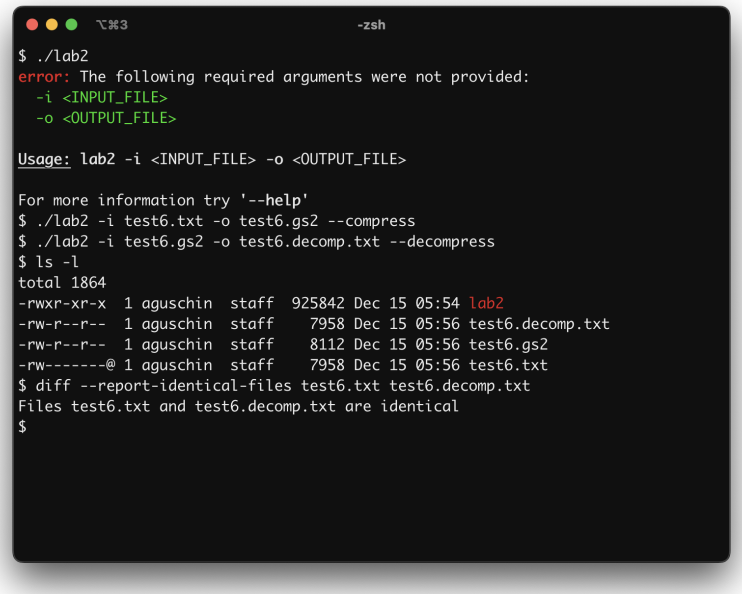

Рис. 2: Сжатие текста Тест_6.txt

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	2029	1.34993
Тест_4.txt	330	48	6.875
Тест_5.txt	59	169	0.34911
Тест_6.txt	7958	8112	0.98102
Тест_7.txt	138245	122970	1.12422
Тест_8.txt	574426	511681	1.12263
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 4.31 секунды, среднее время распаковки составило 5 секунд.

Таким образом, средняя скорость сжатия составила 16.9162 Мбайт в секунду, а средняя скорость разжатия составила 14.58176 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 weighted.rs

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

#[derive(Debug, Clone)]
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()
    }
}
```

5.2 Содержимое файла fano.rs

```
use crate::weighted::Weighted;
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 FanoTree<T>
where
    T: std::cmp::Eq + std::hash::Hash + Copy,
{
    Leaf(T),
    Node(Box<FanoTree<T>>, Box<FanoTree<T>>),
}

impl<T: std::cmp::Eq + std::hash::Hash + Copy> FanoTree<T> {
```

```rust
    fn prefix_sum(weights: &Vec<Weighted<T>>) -> Vec<u32> {
        let mut pf = Vec::new();
        for i in 0..weights.len() {
            if i == 0 {
                pf.push(weights[i].weight);
            } else {
                pf.push(weights[i].weight + pf[i - 1]);
            }
        }
        return pf;
    }

    fn from_weights(weights: &Vec<Weighted<T>>) -> Self {
        if weights.len() == 1 {
            return FanoTree::Leaf(weights.first().unwrap().value);
        }

        let pf = FanoTree::prefix_sum(weights);
        let m = partition(&pf);

        let l = Vec::from(&weights[..m + 1]);
        let r = Vec::from(&weights[m + 1..]);

        let l_tree = if l.len() == 1 {
            FanoTree::Leaf(l.first().unwrap().value)
        } else {
            FanoTree::from_weights(&l)
        };

        let r_tree = if r.len() == 1 {
            FanoTree::Leaf(r.first().unwrap().value)
        } else {
            FanoTree::from_weights(&r)
        };

        return FanoTree::Node(Box::new(l_tree), Box::new(r_tree));
    }

    fn from_hashmap(map: &HashMap<T, u32>) -> Self {
        let mut weights = map
            .iter()
            .map(|(v, p)| Weighted {
                weight: *p,
                value: *v,
            })
            .collect::<Vec<Weighted<T>>>();
        weights.sort_by_key(|p| p.weight);
        return FanoTree::from_weights(&weights);
    }

    fn get_code_rec(tree: &FanoTree<T>, codes: &mut HashMap<T, Vec<Bit>>,
    ↪ run: Vec<Bit>) {
        match tree {
            FanoTree::Node(left, right) => {
```

```rust
                let mut left_run = run.clone();
                left_run.push(Bit::Zero);
                FanoTree::get_code_rec(left, codes, left_run);

                let mut right_run = run.clone();
                right_run.push(Bit::One);
                FanoTree::get_code_rec(right, codes, right_run);
            }
            FanoTree::Leaf(value) => {
                if run.len() == 0 {
                    codes.insert(*value, vec![Bit::Zero]);
                } else {
                    codes.insert(*value, run.clone());
                }
            }
        }
    }

    fn get_code(&self) -> HashMap<T, Vec<Bit>> {
        let mut code = HashMap::new();
        FanoTree::get_code_rec(&self, &mut code, Vec::new());
        return code;
    }
}

fn partition(pf: &Vec<u32>) -> usize {
    fn inner(pf: &Vec<u32>, l: usize, r: usize, prev: Option<(usize,
    ↪  u32)>) -> usize {
        let m = (l + r) / 2;
        let half = (r - l) / 2;
        let val1 = pf[m];
        let val2 = pf.last().unwrap() - pf[m];
        let diff = (val1 as i32 - val2 as i32).abs() as u32;

        if let Some(prev) = prev {
            if prev.1 < diff {
                return prev.0;
            }
        }

        if half == 0 {
            return m;
        }
        if val1 > val2 {
            return inner(pf, l, r - half, Some((m, diff)));
        } else {
            return inner(pf, l + half, r, Some((m, diff)));
        }
    }
    return inner(pf, 0, pf.len() - 1, None);
}

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: FanoTree<u8>,
    code: HashMap<u8, Vec<Bit>>,
    remainder: u8,
}

impl Metadata {
    fn compute(data: &Vec<u8>) -> Self {
        let weights = get_weights(data);
        let tree = FanoTree::from_hashmap(&weights);
        let code = tree.get_code();

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

    fn load(data: &Vec<u8>) -> Self {
        let remainder = data[0];
        let dict_len = data[1] as usize + 1;
        let mut weights = HashMap::new();
        for i in 0..dict_len {
            weights.insert(data[2 + 2 * i], data[2 + 2 * i + 1] as u32);
        }
        let tree = FanoTree::from_hashmap(&weights);
        let code = tree.get_code();

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

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

```
196
197         result.push(self.remainder);
198         result.push((self.weights.len() - 1) as u8);
199         for (byte, weight) in &self.weights {
200             result.push(*byte);
201             result.push(*weight as u8);
202         }
203
204         return result;
205     }
206 }
207
208 struct BitWriter {
209     buffer: Vec<Bit>,
210     remainder: u8,
211     result: Vec<u8>,
212 }
213
214 impl BitWriter {
215     fn new() -> Self {
216         Self {
217             buffer: Vec::new(),
218             remainder: 0,
219             result: Vec::new(),
220         }
221     }
222
223     fn dump_byte(&mut self) {
224         let mut byte = 0_u8;
225         self.buffer
226             .iter()
227             .map(Bit::to_u8)
228             .enumerate()
229             .for_each(|(i, bit)| byte |= bit << i);
230         self.result.push(byte);
231         self.buffer.clear();
232     }
233
234     fn write_bit(&mut self, bit: Bit) {
235         self.buffer.push(bit);
236         if self.buffer.len() == 8 {
237             self.dump_byte();
238         }
239     }
240
241     fn write_bits(&mut self, bits: &Vec<Bit>) {
242         bits.iter().for_each(|bit| self.write_bit(*bit));
243     }
244
245     fn finish(&mut self) {
246         let remainder = 8 - self.buffer.len() as u8 % 8;
247         if remainder != 0 {
248             self.dump_byte();
249         }
```

```
            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;
    }

    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;
            }
```

```
304         }
305
306         fn read_bit(&mut self) -> Option<Bit> {
307             if self.buffer.len() == 0 {
308                 self.read_byte();
309             }
310             self.buffer.pop()
311         }
312     }
313
314     pub fn decompress(archive: &Vec<u8>) -> Vec<u8> {
315         let mut result = Vec::new();
316         let metadata = Metadata::load(archive);
317
318         let data = archive[2 + metadata.weights.len() * 2..].to_vec();
319         let mut reader = BitReader::new(&data, &metadata);
320
321         let mut state = &metadata.tree;
322         while let Some(bit) = reader.read_bit() {
323             if let FanoTree::Node(left, right) = state {
324                 match bit {
325                     Bit::Zero => state = left,
326                     Bit::One => state = right,
327                 }
328             }
329
330             if let FanoTree::Leaf(byte) = state {
331                 result.push(*byte);
332                 state = &metadata.tree;
333             }
334         }
335
336         return result;
337     }
```

5.3 Содержимое файла main.rs

```
1   mod fano;
2   mod weighted;
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,
```

```rust

    #[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 = fano::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)?;
    let archive = fano::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(())
}
```

5.4 Содержимое файла Cargo.toml

```toml
[package]
name = "lab2"
```

```
version = "0.1.0"
edition = "2021"

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