Теория кодирования и сжатия информации
Лабораторная работа №10

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию цифрового изображения используя алгоритм RLE. Программа архивации и разархивации должны быть представлены отдельно и работать независимо друг от друга. Определить для данного шифра характеристику 1, 2 и 3. К работе необходимо прикрепить отчет и программный проект.

2 Алгоритм

Алгоритм RLE заключается в кодировании стоящих подряд одинаковых значений в виде пар (n, byte), где n — количество повторений, byte — повторяющийся элемент.

3 Тестирование

Для проверки программы были использованы тестовые изображения 4.1.04.tiff (рис. 1, 2), 4.2.01.tiff (рис. 3, 4) и ruler.512.tiff (рис. 5, 6). Можно заметить, что после распаковки архива полученный файл совпадает с исходным.

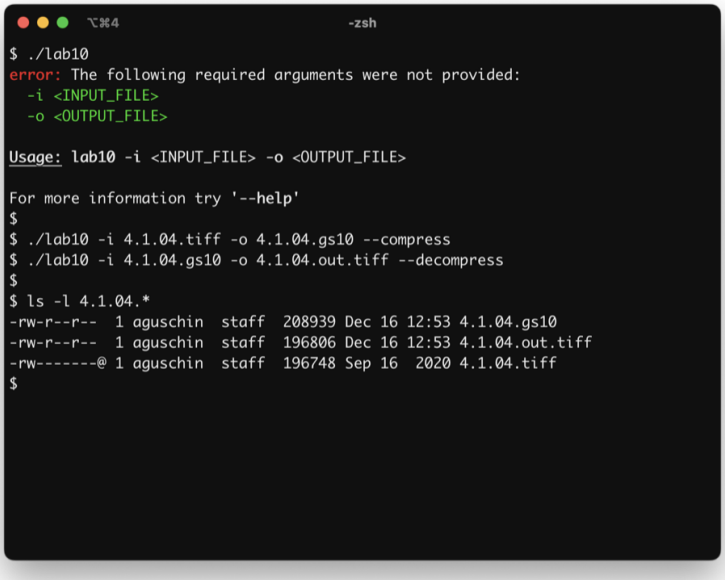

Рис. 1: Сжатие файла 4.1.04.tiff

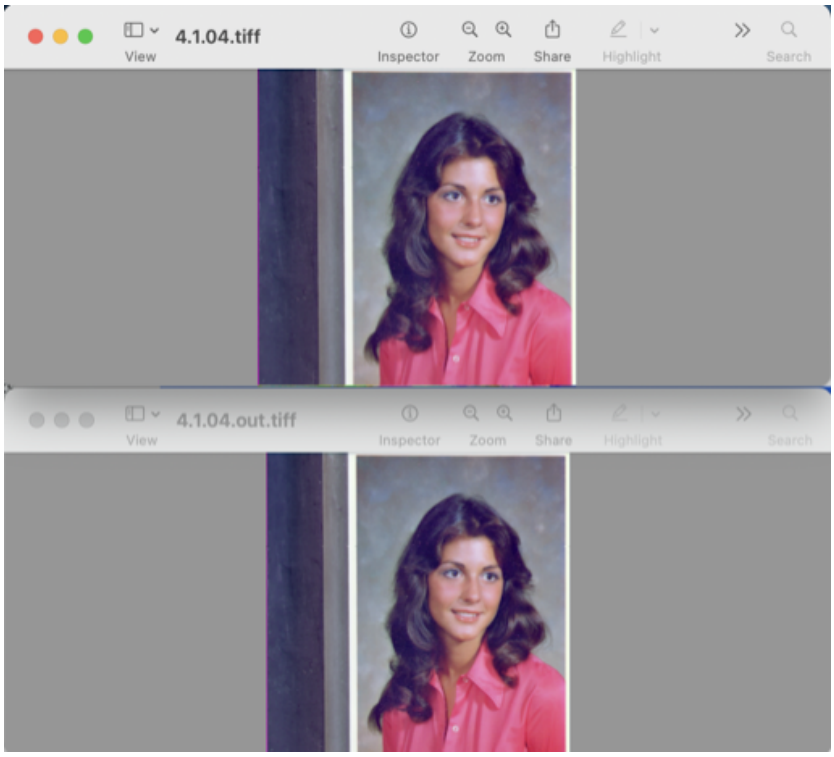

Рис. 2: Сравнение разжатого изображения с 4.1.04.tiff

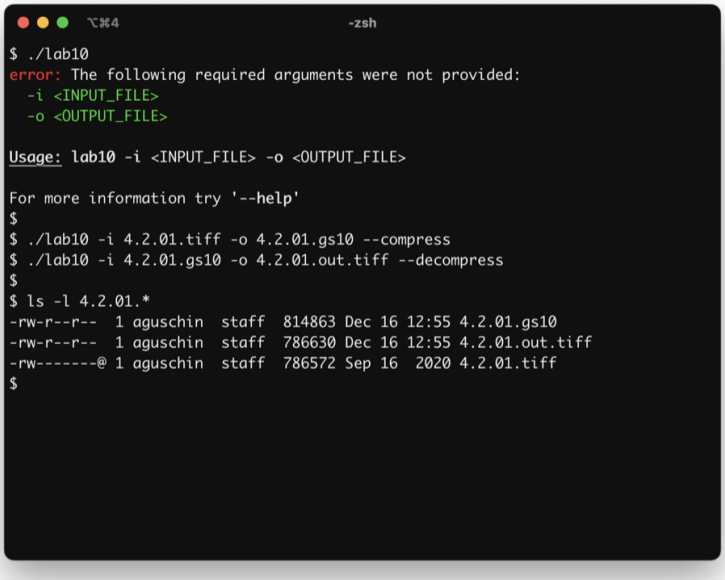

Рис. 3: Сжатие файла 4.2.01.tiff

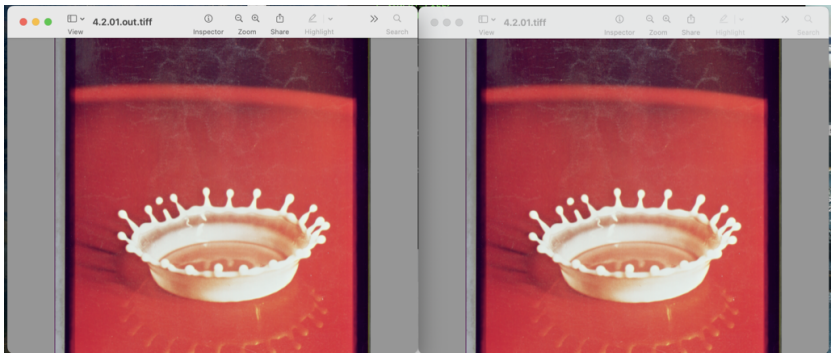

Рис. 4: Сравнение разжатого изображения с 4.2.01.tiff

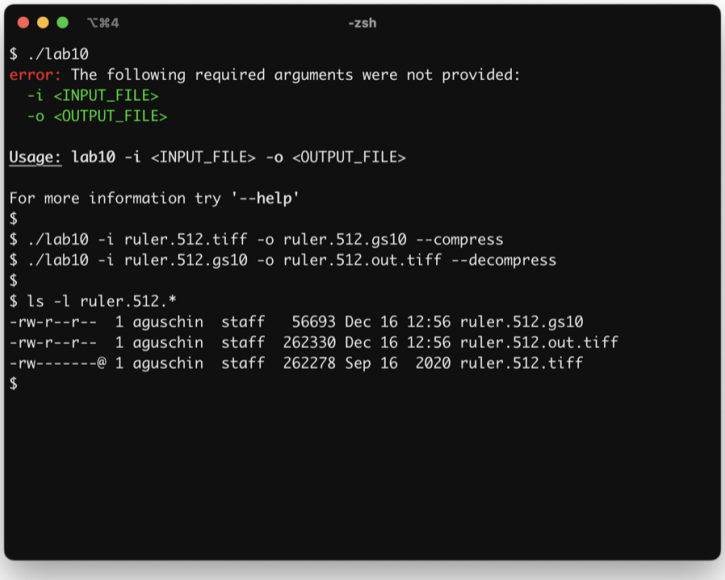

Рис. 5: Сжатие файла ruler.512.tiff

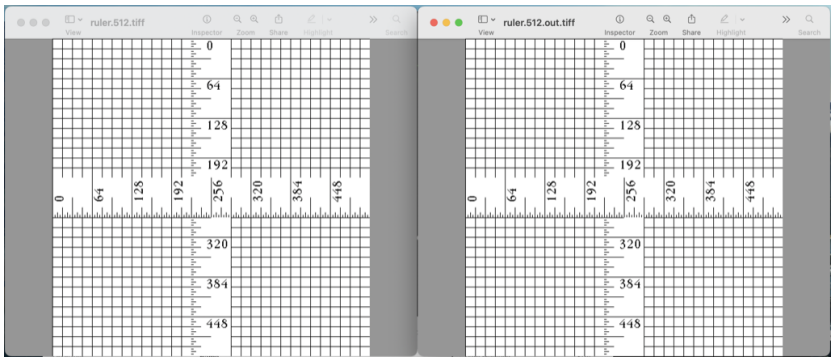

Рис. 6: Сравнение разжатого изображения с ruler.512.tiff

4 Вычисленные характеристики

4.1 Характеристика 1 (Коэффициент сжатия)

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

Название	Исходный размер, байт	Сжатый размер, байт	Коэффициент
4.1.04.tiff	196748	208939	0.94165
4.1.05.tiff	196748	206365	0.9534
4.1.06.tiff	196748	202228	0.9729
4.1.08.tiff	196748	194291	1.01265
4.2.01.tiff	786572	814863	0.96528
4.2.03.tiff	786572	807439	0.97416
4.2.05.tiff	786572	831665	0.94578
4.2.07.tiff	786572	801443	0.98144
5.1.09.tiff	65670	68122	0.96401
5.1.11.tiff	65670	69112	0.9502
5.1.13.tiff	65670	13476	4.87311
5.1.14.tiff	65670	68240	0.96234
5.2.10.tiff	262278	268112	0.97824
5.3.01.tiff	1048710	1092262	0.96013
5.3.02.tiff	1048710	1087858	0.96401
boat.512.tiff	262278	273143	0.96022
gray21.512.tiff	262278	6157	42.59834
house.tiff	786572	803252	0.97923
ruler.512.tiff	262278	56693	4.62629

Таблица 1: результаты тестирования

4.2 Характеристика 2 (Скорость сжатия)

Для тестирования скорости сжатия использовался произвольный графический файл размера 4808956 байт (≈4.6 мегабайта). В результате пяти последовательных запусков, среднее время запаковки файла составило 0.05 секунды, среднее время распаковки составило 0.05 секунд.

Таким образом, средняя скорость сжатия составила 91.72356 Мбайт в секунду, а средняя скорость разжатия составила 91.72356 Мбайт в секунду.

4.3 Характеристика 3 (Качество сжатия)

Качество изображения не изменилось после сжатия, так как этот алгоритм является алгоритмом сжатия без потерь.

5 Реализация

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

5.1 Содержимое файла rle.rs

```
fn dump_unique(encoded: &mut Vec<i8>, unique: &mut Vec<u8>) {
    let mut ptr = 0;
    while ptr < unique.len() {
        let l = std::cmp::min(127, unique.len() - ptr);
        encoded.push(-(l as i8));
```

```
        for i in 0..l {
            encoded.push(unique[ptr + i] as i8);
        }
        ptr += l;
    }
}

fn dump_repeat(encoded: &mut Vec<i8>, byte: u8, repeat: usize) {
    let mut ptr = 0;
    while ptr < repeat {
        let l = std::cmp::min(127, repeat - ptr);
        encoded.push(l as i8);
        encoded.push(byte as i8);
        ptr += l;
    }
}

fn rle_encode(data: &Vec<u8>) -> Vec<i8> {
    let mut encoded = Vec::new();

    let mut last = data[0];
    let mut count = 1;
    let mut unique = Vec::new();

    let mut ptr = 1;
    while ptr < data.len() {
        let byte = data[ptr];
        if byte == last {
            count += 1;
        } else {
            if count == 1 {
                unique.push(last);
            } else {
                dump_unique(&mut encoded, &mut unique);
                unique.clear();
                dump_repeat(&mut encoded, last, count);
            }
            last = byte;
            count = 1;
        }
        ptr += 1;
    }
    dump_unique(&mut encoded, &mut unique);
    dump_repeat(&mut encoded, last, count);

    return encoded;
}

fn split_data(data: &Vec<u8>) -> (Vec<u8>, Vec<u8>, Vec<u8>) {
    let mut r = Vec::new();
    let mut g = Vec::new();
    let mut b = Vec::new();

    for i in 0..data.len() / 3 {
```

```
        r.push(data[i * 3 + 0]);
        g.push(data[i * 3 + 1]);
        b.push(data[i * 3 + 2]);
    }

    return (r, g, b);
}

fn dump_u32(data: &mut Vec<u8>, val: u32) {
    let val_bytes: [u8; 4] = unsafe { std::mem::transmute(val) };
    data.push(val_bytes[0]);
    data.push(val_bytes[1]);
    data.push(val_bytes[2]);
    data.push(val_bytes[3]);
}

fn read_u32(data: &Vec<u8>, offset: usize) -> u32 {
    let val_bytes = [
        data[offset + 0],
        data[offset + 1],
        data[offset + 2],
        data[offset + 3],
    ];
    return unsafe { std::mem::transmute(val_bytes) };
}

pub fn compress_rgb(data: &Vec<u8>, dim: (u32, u32)) -> Vec<u8> {
    let mut result = Vec::new();
    let (r, g, b) = split_data(data);
    let r_enc = rle_encode(&r);
    let g_enc = rle_encode(&g);
    let b_enc = rle_encode(&b);

    result.push(1);
    dump_u32(&mut result, dim.0);
    dump_u32(&mut result, dim.1);

    dump_u32(&mut result, r_enc.len() as u32);
    for elem in r_enc {
        result.push(elem as u8);
    }

    dump_u32(&mut result, g_enc.len() as u32);
    for elem in g_enc {
        result.push(elem as u8);
    }

    dump_u32(&mut result, b_enc.len() as u32);
    for elem in b_enc {
        result.push(elem as u8);
    }

    return result;
}
```

```rust

pub fn compress_gray(data: &Vec<u8>, dim: (u32, u32)) -> Vec<u8> {
    let mut result = Vec::new();
    let enc = rle_encode(data);

    result.push(0);
    dump_u32(&mut result, dim.0);
    dump_u32(&mut result, dim.1);

    dump_u32(&mut result, enc.len() as u32);
    for elem in enc {
        result.push(elem as u8);
    }

    return result;
}

fn rle_decode(data: &Vec<i8>) -> Vec<u8> {
    let mut result = Vec::new();

    let mut ptr = 0;
    while ptr < data.len() {
        let repeat = data[ptr];
        ptr += 1;
        if repeat < 0 {
            for i in 0..(repeat as i32).abs() {
                result.push(data[ptr + i as usize] as u8);
            }
            ptr += repeat.abs() as usize;
        } else {
            let byte = data[ptr];
            ptr += 1;
            for _ in 0..repeat {
                result.push(byte as u8);
            }
        }
    }

    return result;
}

pub fn decompress(data: &Vec<u8>) -> (Vec<u8>, (u32, u32)) {
    let mut result = Vec::new();
    let is_gray = data[0] == 0;
    let width = read_u32(&data, 1);
    let height = read_u32(&data, 5);

    if is_gray {
        let mut archive = Vec::new();
        for i in &data[13..] {
            archive.push(*i as i8);
        }
        let mut decoded = rle_decode(&archive);
        result.append(&mut decoded);
```

```
168     } else {
169         let mut shift = 9;
170         let rsize = read_u32(&data, shift) as usize;
171         shift += 4;
172         let mut r_archive = Vec::new();
173         for i in &data[shift..shift + rsize] {
174             r_archive.push(*i as i8);
175         }
176         shift += rsize;
177 
178         let gsize = read_u32(&data, shift) as usize;
179         shift += 4;
180         let mut g_archive = Vec::new();
181         for i in &data[shift..shift + gsize] {
182             g_archive.push(*i as i8);
183         }
184         shift += gsize;
185 
186         let bsize = read_u32(&data, shift) as usize;
187         shift += 4;
188         let mut b_archive = Vec::new();
189         for i in &data[shift..shift + bsize] {
190             b_archive.push(*i as i8);
191         }
192 
193         let r_decode = rle_decode(&r_archive);
194         let g_decode = rle_decode(&g_archive);
195         let b_decode = rle_decode(&b_archive);
196 
197         for i in 0..r_decode.len() {
198             result.push(r_decode[i]);
199             result.push(g_decode[i]);
200             result.push(b_decode[i]);
201         }
202     }
203 
204     return (result, (width, height));
205 }
```

5.2 Содержимое файла main.rs

```
1  mod rle;
2  use clap::Parser;
3  use std::fs::File;
4  use std::io::{Error, ErrorKind, Read, Write};
5  use std::path::PathBuf;
6  use tiff;
7  use tiff::decoder::DecodingResult;
8  use tiff::encoder::colortype;
9  use tiff::ColorType;
10 
11 #[derive(Parser)]
12 struct Cli {
13     #[arg(short)]
```

```
14      input_file: PathBuf,
15
16      #[arg(short)]
17      output_file: PathBuf,
18
19      #[arg(long, default_value_t = true)]
20      compress: bool,
21
22      #[arg(long, default_value_t = false)]
23      decompress: bool,
24  }
25
26  fn run_decompressor(cli: &Cli) -> Result<(), Error> {
27      let mut input_f = File::open(cli.input_file.to_str().unwrap())?;
28      let mut archive = Vec::new();
29      input_f.read_to_end(&mut archive)?;
30      let (data, dim) = rle::decompress(&archive);
31
32      let output_f = File::create(cli.output_file.to_str().unwrap())?;
33      let mut encoder = tiff::encoder::TiffEncoder::new(output_f).unwrap();
34      if archive[0] == 0 {
35          encoder
36              .write_image::<colortype::Gray8>(dim.0, dim.1, &data)
37              .unwrap();
38      } else {
39          encoder
40              .write_image::<colortype::RGB8>(dim.0, dim.1, &data)
41              .unwrap();
42      }
43
44      return Ok(());
45  }
46
47  fn run_compressor(cli: &Cli) -> Result<(), Error> {
48      let input_f = File::open(cli.input_file.to_str().unwrap())?;
49      let mut decoder = tiff::decoder::Decoder::new(&input_f).unwrap();
50      let img_coded = decoder.read_image();
51      let dim = decoder.dimensions().unwrap();
52      match img_coded {
53          Ok(DecodingResult::U8(data)) => {
54              let archive = match decoder.colortype() {
55                  Ok(ColorType::RGB(_)) => rle::compress_rgb(&data, dim),
56                  Ok(ColorType::Gray(_)) => rle::compress_gray(&data, dim),
57                  _ => panic!("unsupported colortype"),
58              };
59              let mut output_f =
    ↪  File::create(cli.output_file.to_str().unwrap())?;
60              output_f.write_all(&archive)?;
61          }
62          _ => {
63              panic!("something went wrong");
64          }
65      }
66
```

```
    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

```
[package]
name = "lab10"
version = "0.1.0"
edition = "2021"

[dependencies]
clap = { version = "4.0.17", features = ["derive"] }
tiff = "0.8.1"
```