Теория кодирования и сжатия информации
Лабораторная работа №11

Гущин Андрей, 431 группа, 1 подгруппа

2022 г.

1 Задача

Разработать программу осуществляющую архивацию и разархивацию цифрового изображения используя алгоритм Лемпеля — Зива — Велча (LZW). 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 LZW заключается в создании нового кода для всех встречающихся последовательностей символов. Алгоритм использует динамический словарь.

Алгоритм состоит из следующих шагов:

1. Прочитать первый символ. Так как он ни разу не встречался, добавить его в словарь.
2. Далее необходимо найти наибольшую подстроку ps, уже находящуюся в словаре.
3. Если строка не найдена, то просто добавить отдельный символ аналогично самому первому. Иначе записать найденную подстроку и добавить в словарь конкатенацию найденной записи и следующего символа.

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.

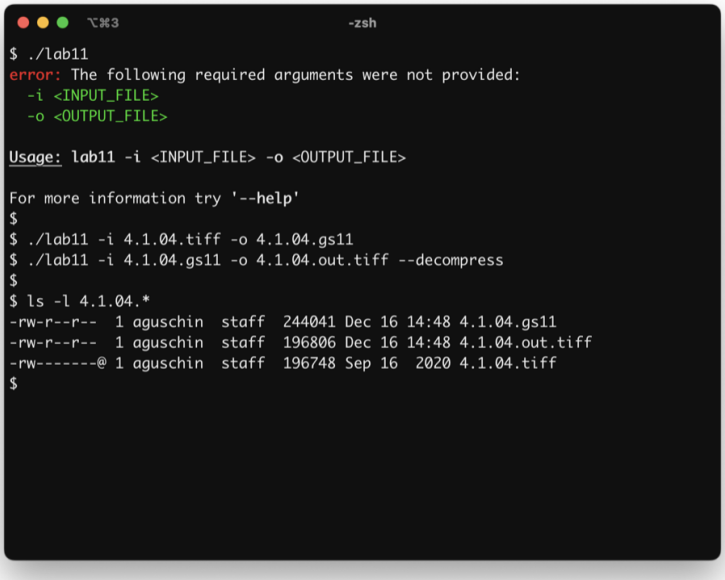

Рис. 1: Сжатие файла 4.1.04.tiff

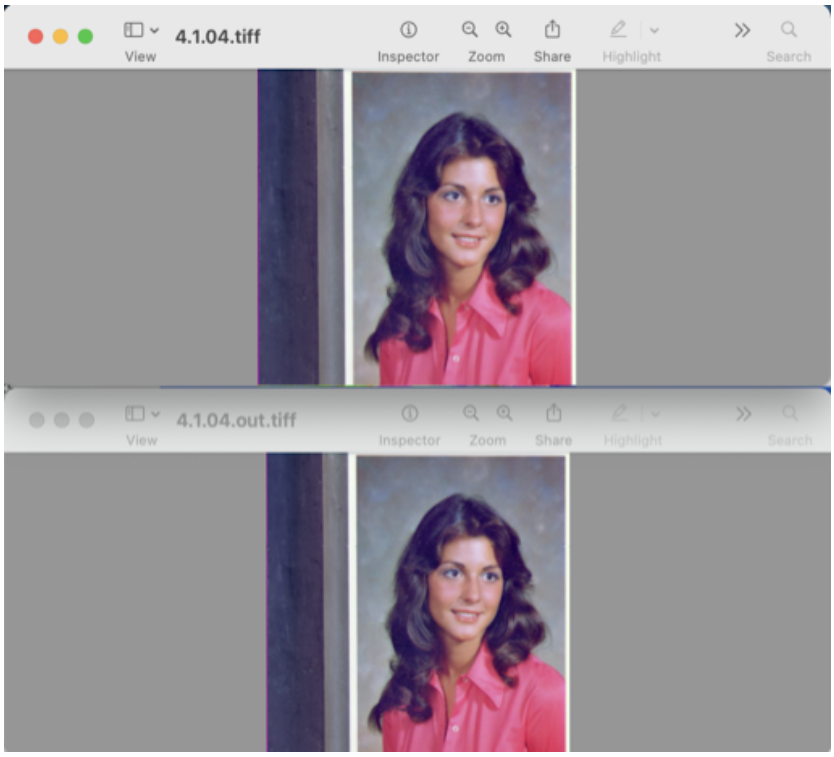

Рис. 2: Сравнение разжатого изображения с 4.1.04.tiff

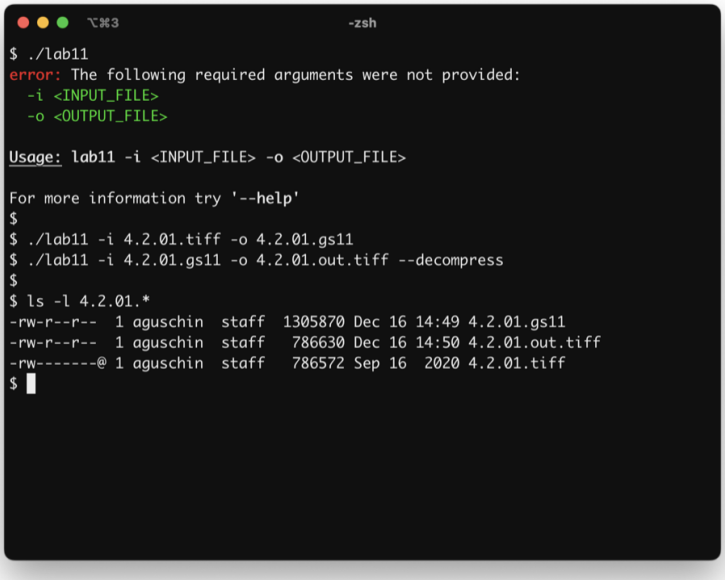

Рис. 3: Сжатие файла 4.2.01.tiff

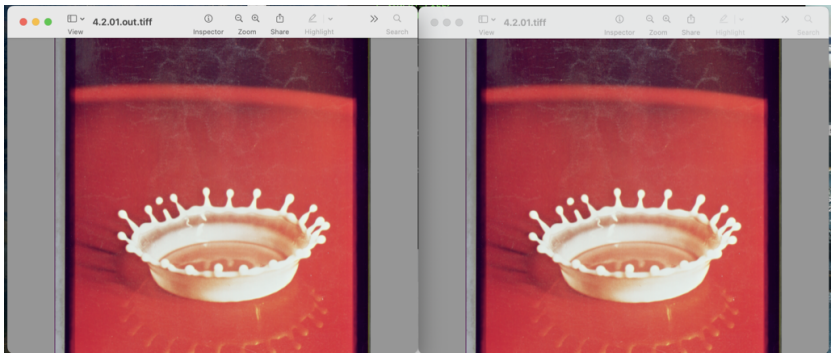

Рис. 4: Сравнение разжатого изображения с 4.2.01.tiff

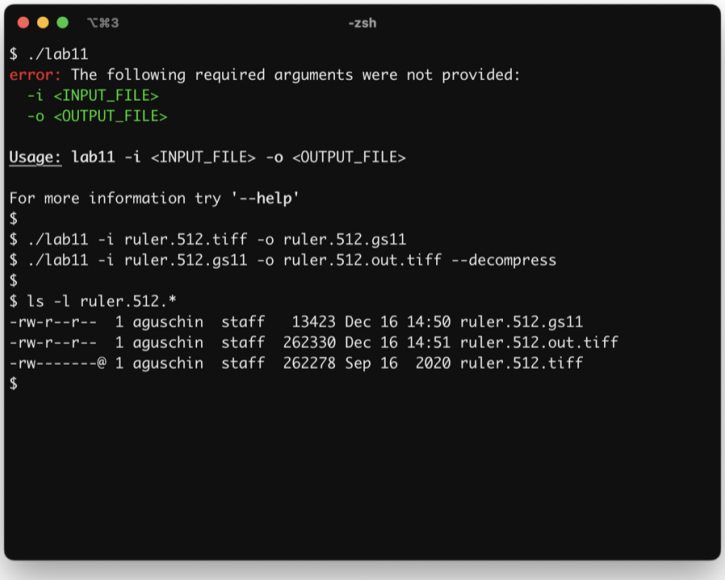

Рис. 5: Сжатие файла ruler.512.tiff

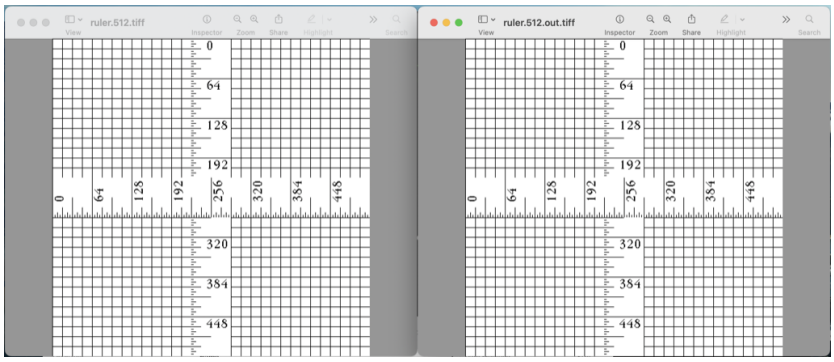

Рис. 6: Сравнение разжатого изображения с ruler.512.tiff

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	244041	0.80621
4.1.05.tiff	196748	239220	0.82246
4.1.06.tiff	196748	242346	0.81185
4.1.08.tiff	196748	260436	0.75546
4.2.01.tiff	786572	1305870	0.60234
4.2.03.tiff	786572	1165509	0.67487
4.2.05.tiff	786572	1039467	0.75671
4.2.07.tiff	786572	980445	0.80226
5.1.09.tiff	65670	80995	0.81079
5.1.11.tiff	65670	127582	0.51473
5.1.13.tiff	65670	13879	4.73161
5.1.14.tiff	65670	82720	0.79388
5.2.10.tiff	262278	273973	0.95731
5.3.01.tiff	1048710	1347127	0.77848
5.3.02.tiff	1048710	1243720	0.8432
boat.512.tiff	262278	366754	0.71513
gray21.512.tiff	262278	12961	20.2359
house.tiff	786572	1007154	0.78098
ruler.512.tiff	262278	13423	19.5394

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 13.25 секунды, среднее время распаковки составило 0.04 секунд.

Таким образом, средняя скорость сжатия составила 354.43367 Кбайт в секунду, а средняя скорость разжатия составила 117406.15234 Кбайт в секунду (114.65445 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 lzw.rs

```
1 const DICT_SIZE: usize = 4096;
2
3 pub fn lz_match(data: &Vec<u8>, pos1: usize, pos2: usize, length: usize)
  ↪  -> bool {
```

```
    for i in 0..length {
        if data[pos1 + i] != data[pos2 + i] {
            return false;
        }
    }
    return true;
}

fn lzw_encode(data: &Vec<u8>) -> Vec<(u32, u8)> {
    let mut encoded = Vec::new();
    let mut dict = Vec::new();
    let mut count = 1;

    let mut ptr = 0;
    while ptr < data.len() {
        let mut saved = None;
        let mut l = 1;
        while ptr + l < data.len() && l < 256 {
            let mut found = false;
            for i in (0..dict.len()).rev() {
                let (pos1, l1, val) = dict[i];
                if l != l1 {
                    continue;
                }
                if lz_match(data, pos1, ptr, l) {
                    found = true;
                    saved = Some((pos1, l, val));
                    l += 1;
                    break;
                }
            }
            if !found {
                break;
            }
        }
        if let Some((_, l, val)) = saved {
            let t = (val, data[ptr + l]);
            encoded.push(t);
            if dict.len() < DICT_SIZE {
                dict.push((ptr, l + 1, count));
                count += 1;
            }
            ptr += l + 1;
        } else {
            encoded.push((0, data[ptr]));
            if dict.len() < DICT_SIZE {
                dict.push((ptr, 1, count));
                count += 1;
            }
            ptr += 1;
        }
    }

    return encoded;
```

```rust
}

pub fn lzw_compress(data: &Vec<u8>) -> Vec<u8> {
    let mut result = Vec::new();
    let encoded = lzw_encode(data);

    for (value, byte) in &encoded {
        let value_bytes: [u8; 4] = unsafe { std::mem::transmute(*value) };
        result.push(value_bytes[0]);
        result.push(value_bytes[1]);
        result.push(*byte);
    }

    return result;
}

pub fn lzw_decompress(data: &Vec<u8>) -> Vec<u8> {
    let mut decoded = Vec::new();
    let mut dict = Vec::new();
    let mut count = 1;
    let mut ptr = 0;

    let mut caret = 0;
    while caret < data.len() {
        let value_bytes: [u8; 4] = [data[caret + 0], data[caret + 1], 0, 0];
        let value: u32 = unsafe { std::mem::transmute(value_bytes) };
        let byte = data[caret + 2];
        caret += 3;

        if value == 0 {
            if dict.len() < DICT_SIZE {
                dict.push((ptr, 1, count));
                count += 1;
            }
            decoded.push(byte);
            ptr += 1;
        } else {
            let (pos1, l, _) = dict[value as usize - 1];
            for i in 0..l {
                decoded.push(decoded[pos1 + i]);
            }
            decoded.push(byte);
            if dict.len() < DICT_SIZE {
                dict.push((ptr, l + 1, count));
                count += 1;
            }
            ptr += l + 1;
        }
    }

    return decoded;
}
```

```rust

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 = lzw_compress(&r);
    let g_enc = lzw_compress(&g);
    let b_enc = lzw_compress(&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);
    }

```

```
164     dump_u32(&mut result, b_enc.len() as u32);
165     for elem in b_enc {
166         result.push(elem as u8);
167     }
168 
169     return result;
170 }
171 
172 pub fn compress_gray(data: &Vec<u8>, dim: (u32, u32)) -> Vec<u8> {
173     let mut result = Vec::new();
174     let enc = lzw_compress(data);
175 
176     result.push(0);
177     dump_u32(&mut result, dim.0);
178     dump_u32(&mut result, dim.1);
179 
180     dump_u32(&mut result, enc.len() as u32);
181     for elem in enc {
182         result.push(elem as u8);
183     }
184 
185     return result;
186 }
187 
188 pub fn decompress(data: &Vec<u8>) -> (Vec<u8>, (u32, u32)) {
189     let mut result = Vec::new();
190     let is_gray = data[0] == 0;
191     let width = read_u32(&data, 1);
192     let height = read_u32(&data, 5);
193 
194     if is_gray {
195         let archive = Vec::from(&data[13..]);
196         let mut decoded = lzw_decompress(&archive);
197         result.append(&mut decoded);
198     } else {
199         let mut shift = 9;
200         let rsize = read_u32(&data, shift) as usize;
201         shift += 4;
202         let r_archive = Vec::from(&data[shift..shift + rsize]);
203         shift += rsize;
204 
205         let gsize = read_u32(&data, shift) as usize;
206         shift += 4;
207         let g_archive = Vec::from(&data[shift..shift + gsize]);
208         shift += gsize;
209 
210         let bsize = read_u32(&data, shift) as usize;
211         shift += 4;
212         let b_archive = Vec::from(&data[shift..shift + bsize]);
213 
214         let r_decode = lzw_decompress(&r_archive);
215         let g_decode = lzw_decompress(&g_archive);
216         let b_decode = lzw_decompress(&b_archive);
217 
```

```
        for i in 0..r_decode.len() {
            result.push(r_decode[i]);
            result.push(g_decode[i]);
            result.push(b_decode[i]);
        }
    }

    return (result, (width, height));
}
```

5.2 Содержимое файла main.rs

```
mod lzw;
use clap::Parser;
use std::fs::File;
use std::io::{Error, ErrorKind, Read, Write};
use std::path::PathBuf;
use tiff;
use tiff::decoder::DecodingResult;
use tiff::encoder::colortype;
use tiff::ColorType;

#[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, dim) = lzw::decompress(&archive);

    let output_f = File::create(cli.output_file.to_str().unwrap())?;
    let mut encoder = tiff::encoder::TiffEncoder::new(output_f).unwrap();
    if archive[0] == 0 {
        encoder
            .write_image::<colortype::Gray8>(dim.0, dim.1, &data)
            .unwrap();
    } else {
        encoder
            .write_image::<colortype::RGB8>(dim.0, dim.1, &data)
            .unwrap();
    }
```

```rust

    return Ok(());
}

fn run_compressor(cli: &Cli) -> Result<(), Error> {
    let input_f = File::open(cli.input_file.to_str().unwrap())?;
    let mut decoder = tiff::decoder::Decoder::new(&input_f).unwrap();
    let img_coded = decoder.read_image();
    let dim = decoder.dimensions().unwrap();
    match img_coded {
        Ok(DecodingResult::U8(data)) => {
            let archive = match decoder.colortype() {
                Ok(ColorType::RGB(_)) => lzw::compress_rgb(&data, dim),
                Ok(ColorType::Gray(_)) => lzw::compress_gray(&data, dim),
                _ => panic!("unsupported colortype"),
            };
            let mut output_f =
↪   File::create(cli.output_file.to_str().unwrap())?;
            output_f.write_all(&archive)?;
        }
        _ => {
            panic!("something went wrong");
        }
    }

    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 = "lab11"
version = "0.1.0"
```

```
edition = "2021"

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