

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.

```

$ ./lab5
error: The following required arguments were not provided:
  -i <INPUT_FILE>
  -o <OUTPUT_FILE>

Usage: lab5 -i <INPUT_FILE> -o <OUTPUT_FILE>

For more information try '--help'
$ ./lab5 -i test1.txt -o test1.gs5 --compress
$ ./lab5 -i test1.gs5 -o test1.out.txt --decompress
$ ls -l
total 1840
-rwxr-xr-x  1 aguschin  staff   926066 Dec 15 13:36 lab5
-rw-r--r--  1 aguschin  staff      5 Dec 15 13:36 test1.gs5
-rw-r--r--  1 aguschin  staff      2 Dec 15 13:37 test1.out.txt
-rw-----@ 1 aguschin  staff      2 Dec 15 13:36 test1.txt
$ diff --report-identical-files test1.txt test1.out.txt
Files test1.txt and test1.out.txt are identical
$
```

Рис. 1: Сжатие текста Тест_1.txt

```

$ ./lab5
error: The following required arguments were not provided:
  -i <INPUT_FILE>
  -o <OUTPUT_FILE>

Usage: lab5 -i <INPUT_FILE> -o <OUTPUT_FILE>

For more information try '--help'
$ ./lab5 -i test6.txt -o test6.gs5 --compress
$ ./lab5 -i test6.gs5 -o test6.out.txt --decompress
$ ls -l
total 1864
-rwxr-xr-x  1 aguschin  staff   926066 Dec 15 13:36 lab5
-rw-r--r--  1 aguschin  staff    6117 Dec 15 13:37 test6.gs5
-rw-r--r--  1 aguschin  staff    7958 Dec 15 13:38 test6.out.txt
-rw-----@ 1 aguschin  staff    7958 Dec 15 13:37 test6.txt
$ diff --report-identical-files test6.txt test6.out.txt
Files test6.txt and test6.out.txt are identical
$
```

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 3.45 секунд, среднее время распаковки составило 180 секунд.

Таким образом, средняя скорость сжатия составила 21.13299 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 `priority_queue.rs`

```
1 use std::cmp::Ordering;
2 use std::collections::BinaryHeap;
3
4 #[derive(Debug)]
5 pub struct Weighted<T> {
6     pub weight: u32,
7     pub value: T,
8 }
9
10 impl<T> PartialEq for Weighted<T> {
11     fn eq(&self, other: &Self) -> bool {
12         self.weight == other.weight
```

```

13     }
14 }
15
16 impl<T> Eq for Weighted<T> {}
17
18 impl<T> PartialOrd for Weighted<T> {
19     fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
20         Some(self.cmp(other))
21     }
22 }
23
24 impl<T> Ord for Weighted<T> {
25     fn cmp(&self, other: &Self) -> Ordering {
26         self.weight.cmp(&other.weight).reverse()
27     }
28 }
29
30 #[derive(Debug)]
31 pub struct PriorityQueue<T> {
32     heap: BinaryHeap<Weighted<T>>,
33 }
34
35 impl<T> PriorityQueue<T> {
36     pub fn new() -> Self {
37         PriorityQueue {
38             heap: BinaryHeap::new(),
39         }
40     }
41
42     pub fn insert(&mut self, priority: u32, value: T) {
43         self.heap.push(Weighted {
44             weight: priority,
45             value,
46         });
47     }
48
49     pub fn len(&self) -> usize {
50         self.heap.len()
51     }
52
53     pub fn pop(&mut self) -> Option<Weighted<T>> {
54         Some(self.heap.pop()?)
55     }
56 }

```

5.2 Содержимое файла huffman.rs

```

1 use crate::priority_queue::PriorityQueue;
2 use std::collections::{HashMap, HashSet};
3
4 #[derive(Clone, Copy, Debug)]
5 enum Bit {
6     Zero,
7     One,

```

```

8 }
9 impl Bit {
10     fn from_u8(x: u8) -> Bit {
11         if x == 0 {
12             Bit::Zero
13         } else {
14             Bit::One
15         }
16     }
17
18     fn to_u8(x: &Bit) -> u8 {
19         match *x {
20             Bit::Zero => 0,
21             Bit::One  => 1,
22         }
23     }
24 }
25
26 #[derive(Debug)]
27 enum HuffmanTree<T>
28 where
29     T: Eq + Ord + Copy + std::hash::Hash,
30 {
31     Leaf(T),
32     Node(Box<HuffmanTree<T>>, Box<HuffmanTree<T>>),
33 }
34
35 impl<T: Eq + Ord + Copy + std::hash::Hash> HuffmanTree<T> {
36     fn from_queue(mut queue: PriorityQueue<T>) -> Self {
37         let mut huf_queue = PriorityQueue::new();
38         while let Some(p) = queue.pop() {
39             huf_queue.insert(p.weight,
↳ Box::new(HuffmanTree::Leaf(p.value)));
40         }
41         assert!(huf_queue.len() > 0);
42         while huf_queue.len() > 1 {
43             let v1 = huf_queue.pop().unwrap();
44             let v2 = huf_queue.pop().unwrap();
45             huf_queue.insert(
46                 v1.weight + v2.weight,
47                 Box::new(HuffmanTree::Node(v1.value, v2.value)),
48             );
49         }
50         return *huf_queue.pop().unwrap().value;
51     }
52
53     fn dfs(tree: &HuffmanTree<T>, mut acc: Vec<Bit>, code: &mut
↳ HashMap<T, Vec<Bit>>) -> Vec<Bit> {
54         match tree {
55             HuffmanTree::Leaf(val) => {
56                 code.insert(*val, acc.clone());
57             }
58             HuffmanTree::Node(left, right) => {
59                 acc.push(Bit::Zero);

```

```

60         acc = HuffmanTree::dfs(left, acc, code);
61         acc.pop();
62         acc.push(Bit::One);
63         acc = HuffmanTree::dfs(right, acc, code);
64         acc.pop();
65     }
66 };
67     return acc;
68 }
69
70 fn get_code(&self) -> HashMap<T, Vec<Bit>> {
71     let mut code = HashMap::new();
72     if let HuffmanTree::Leaf(val) = &self {
73         code.insert(*val, vec![Bit::One]);
74     } else {
75         HuffmanTree::dfs(self, Vec::new(), &mut code);
76     }
77     return code;
78 }
79 }
80
81 #[derive(Debug)]
82 struct Metadata {
83     tree: HuffmanTree<u8>,
84     alphabet: Vec<u8>,
85     code: HashMap<u8, Vec<Bit>>,
86     remainder: u8,
87 }
88
89 impl Metadata {
90     fn compute(data: &Vec<u8>) -> Self {
91         let mut alphabet = HashSet::new();
92         data.iter().for_each(|v| {
93             alphabet.insert(*v);
94         });
95         let mut alphabet = Vec::from_iter(alphabet);
96         alphabet.sort();
97
98         let mut queue = PriorityQueue::new();
99         for value in 1..=alphabet.len() as u8 {
100             queue.insert(value as u32, value);
101         }
102         let tree = HuffmanTree::from_queue(queue);
103         let code = tree.get_code();
104
105         return Self {
106             tree,
107             alphabet,
108             code,
109             remainder: 0,
110         };
111     }
112
113     fn load(data: &Vec<u8>) -> Self {

```

```

114         let remainder = data[0];
115         let dict_len = data[1] as usize + 1;
116         let mut alphabet = Vec::new();
117         for i in 0..dict_len {
118             alphabet.push(data[2 + i]);
119         }
120         alphabet.sort();
121
122         let mut queue = PriorityQueue::new();
123         for value in 1..=alphabet.len() as u8 {
124             queue.insert(value as u32, value);
125         }
126         let tree = HuffmanTree::from_queue(queue);
127         let code = tree.get_code();
128
129         return Self {
130             tree,
131             alphabet,
132             code,
133             remainder,
134         };
135     }
136
137     fn dump(&self) -> Vec<u8> {
138         let mut result = Vec::new();
139
140         result.push(self.remainder);
141         result.push((self.alphabet.len() - 1) as u8);
142         for ch in &self.alphabet {
143             result.push(*ch);
144         }
145
146         return result;
147     }
148 }
149
150 struct BitWriter {
151     buffer: Vec<Bit>,
152     remainder: u8,
153     result: Vec<u8>,
154 }
155
156 impl BitWriter {
157     fn new() -> Self {
158         Self {
159             buffer: Vec::new(),
160             remainder: 0,
161             result: Vec::new(),
162         }
163     }
164
165     fn dump_byte(&mut self) {
166         let mut byte = 0_u8;
167         self.buffer

```

```

168         .iter()
169         .map(Bit::to_u8)
170         .enumerate()
171         .for_each(|(i, bit)| byte |= bit << i);
172     self.result.push(byte);
173     self.buffer.clear();
174 }
175
176 fn write_bit(&mut self, bit: Bit) {
177     self.buffer.push(bit);
178     if self.buffer.len() == 8 {
179         self.dump_byte();
180     }
181 }
182
183 fn write_bits(&mut self, bits: &Vec<Bit>) {
184     bits.iter().for_each(|bit| self.write_bit(*bit));
185 }
186
187 fn finish(&mut self) {
188     let remainder = 8 - self.buffer.len() as u8 % 8;
189     if remainder != 0 {
190         self.dump_byte();
191     }
192     self.remainder = remainder;
193 }
194 }
195
196 pub fn compress(data: &Vec<u8>) -> Vec<u8> {
197     let mut result = Vec::new();
198     let mut metadata = Metadata::compute(data);
199
200     let mut alphabet = metadata.alphabet.clone();
201     let mut writer = BitWriter::new();
202     data.iter().for_each(|&byte| {
203         let pos = alphabet.iter().position(|&x| x == byte).unwrap();
204         let bits = metadata.code.get((pos as u8 + 1)).unwrap();
205         alphabet.remove(pos);
206         alphabet.push(byte);
207         writer.write_bits(&bits);
208     });
209     writer.finish();
210     metadata.remainder = writer.remainder;
211
212     let md_dump = metadata.dump();
213     md_dump.iter().for_each(|byte| result.push(*byte));
214     writer.result.iter().for_each(|byte| result.push(*byte));
215
216     return result;
217 }
218
219 struct BitReader<'a> {
220     data: &'a Vec<u8>,
221     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         if let HuffmanTree::Leaf(pos) = state {
277             let byte = alphabet.remove((*pos - 1) as usize);
278             alphabet.push(byte);
279             result.push(byte);
280             state = &metadata.tree;
281         }
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

```

```

41     let mut output_f = File::create(cli.output_file.to_str().unwrap())?;
42     output_f.write_all(&archive)?;
43
44     return Ok(());
45 }
46
47 fn main() -> std::io::Result<()> {
48     let cli = Cli::parse();
49
50     let result = if cli.decompress {
51         run_decompressor(&cli)
52     } else {
53         run_compressor(&cli)
54     };
55
56     if let Err(error) = result {
57         match error.kind() {
58             ErrorKind::NotFound => println!("Указанный файл не найден"),
59             ErrorKind::AlreadyExists => println!("Указанный файл уже
↪ существует"),
60             _ => println!("Произошла непредвиденная ошибка"),
61         };
62     }
63
64     Ok(())
65 }

```

5.4 Содержимое файла Cargo.toml

```

1  [package]
2  name = "lab5"
3  version = "0.1.0"
4  edition = "2021"
5
6  [dependencies]
7  clap = { version = "4.0.17", features = ["derive"] }

```