

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.

```

$ ./lab2
error: The following required arguments were not provided:
  -i <INPUT_FILE>
  -o <OUTPUT_FILE>

Usage: lab2 -i <INPUT_FILE> -o <OUTPUT_FILE>

For more information try '--help'
$ ./lab2 -i test1.txt -o test1.gs2
$ ./lab2 -i test1.gs2 -o test1.decomp.txt --decompress
$ ls -l
total 1840
-rwxr-xr-x  1 aguschin  staff   925842 Dec 15 05:54 lab2
-rw-r--r--  1 aguschin  staff      2 Dec 15 05:55 test1.decomp.txt
-rw-r--r--  1 aguschin  staff      7 Dec 15 05:55 test1.gs2
-rw-----@ 1 aguschin  staff      2 Dec 15 05:54 test1.txt
$ diff --report-identical-files test1.txt test1.decomp.txt
Files test1.txt and test1.decomp.txt are identical
$
```

Рис. 1: Сжатие текста Тест_1.txt

```

$ ./lab2
error: The following required arguments were not provided:
  -i <INPUT_FILE>
  -o <OUTPUT_FILE>

Usage: lab2 -i <INPUT_FILE> -o <OUTPUT_FILE>

For more information try '--help'
$ ./lab2 -i test6.txt -o test6.gs2 --compress
$ ./lab2 -i test6.gs2 -o test6.decomp.txt --decompress
$ ls -l
total 1864
-rwxr-xr-x  1 aguschin  staff   925842 Dec 15 05:54 lab2
-rw-r--r--  1 aguschin  staff    7958 Dec 15 05:56 test6.decomp.txt
-rw-r--r--  1 aguschin  staff    8112 Dec 15 05:56 test6.gs2
-rw-----@ 1 aguschin  staff    7958 Dec 15 05:56 test6.txt
$ diff --report-identical-files test6.txt test6.decomp.txt
Files test6.txt and test6.decomp.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	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

```
1 use std::cmp::Ordering;
2
3 #[derive(Debug, Clone)]
4 pub struct Weighted<T> {
5     pub weight: u32,
6     pub value: T,
7 }
8
9 impl<T> PartialEq for Weighted<T> {
10     fn eq(&self, other: &Self) -> bool {
11         self.weight == other.weight
12     }
13 }
```

```

13 }
14
15 impl<T> Eq for Weighted<T> {}
16
17 impl<T> PartialOrd for Weighted<T> {
18     fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
19         Some(self.cmp(other))
20     }
21 }
22
23 impl<T> Ord for Weighted<T> {
24     fn cmp(&self, other: &Self) -> Ordering {
25         self.weight.cmp(&other.weight).reverse()
26     }
27 }

```

5.2 Содержимое файла fano.rs

```

1 use crate::weighted::Weighted;
2 use std::collections::HashMap;
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 FanoTree<T>
28 where
29     T: std::cmp::Eq + std::hash::Hash + Copy,
30 {
31     Leaf(T),
32     Node(Box<FanoTree<T>>, Box<FanoTree<T>>),
33 }
34
35 impl<T: std::cmp::Eq + std::hash::Hash + Copy> FanoTree<T> {
36     fn prefix_sum(weights: &Vec<Weighted<T>>) -> Vec<u32> {

```

```

37     let mut pf = Vec::new();
38     for i in 0..weights.len() {
39         if i == 0 {
40             pf.push(weights[i].weight);
41         } else {
42             pf.push(weights[i].weight + pf[i - 1]);
43         }
44     }
45     return pf;
46 }
47
48 fn from_weights(weights: &Vec<Weighted<T>>) -> Self {
49     if weights.len() == 1 {
50         return FanoTree::Leaf(weights.first().unwrap().value);
51     }
52
53     let pf = FanoTree::prefix_sum(weights);
54     let m = partition(&pf);
55
56     let l = Vec::from(&weights[..m + 1]);
57     let r = Vec::from(&weights[m + 1..]);
58
59     let l_tree = if l.len() == 1 {
60         FanoTree::Leaf(l.first().unwrap().value)
61     } else {
62         FanoTree::from_weights(&l)
63     };
64
65     let r_tree = if r.len() == 1 {
66         FanoTree::Leaf(r.first().unwrap().value)
67     } else {
68         FanoTree::from_weights(&r)
69     };
70
71     return FanoTree::Node(Box::new(l_tree), Box::new(r_tree));
72 }
73
74 fn from_hashmap(map: &HashMap<T, u32>) -> Self {
75     let mut weights = map
76         .iter()
77         .map(|(v, p)| Weighted {
78             weight: *p,
79             value: *v,
80         })
81         .collect::<Vec<Weighted<T>>>();
82     weights.sort_by_key(|p| p.weight);
83     return FanoTree::from_weights(&weights);
84 }
85
86 fn get_code_rec(tree: &FanoTree<T>, codes: &mut HashMap<T, Vec<Bit>>,
87 ↪ run: Vec<Bit>) {
88     match tree {
89         FanoTree::Node(left, right) => {
90             let mut left_run = run.clone();

```

```

90         left_run.push(Bit::Zero);
91         FanoTree::get_code_rec(left, codes, left_run);
92
93         let mut right_run = run.clone();
94         right_run.push(Bit::One);
95         FanoTree::get_code_rec(right, codes, right_run);
96     }
97     FanoTree::Leaf(value) => {
98         if run.len() == 0 {
99             codes.insert(*value, vec![Bit::Zero]);
100         } else {
101             codes.insert(*value, run.clone());
102         }
103     }
104 }
105 }
106
107 fn get_code(&self) -> HashMap<T, Vec<Bit>> {
108     let mut code = HashMap::new();
109     FanoTree::get_code_rec(&self, &mut code, Vec::new());
110     return code;
111 }
112 }
113
114 fn partition(pf: &Vec<u32>) -> usize {
115     fn inner(pf: &Vec<u32>, l: usize, r: usize, prev: Option<(usize,
116 ↪ u32)>) -> usize {
117         let m = (l + r) / 2;
118         let half = (r - l) / 2;
119         let val1 = pf[m];
120         let val2 = pf.last().unwrap() - pf[m];
121         let diff = (val1 as i32 - val2 as i32).abs() as u32;
122
123         if let Some(prev) = prev {
124             if prev.1 < diff {
125                 return prev.0;
126             }
127         }
128
129         if half == 0 {
130             return m;
131         }
132         if val1 > val2 {
133             return inner(pf, l, r - half, Some((m, diff)));
134         } else {
135             return inner(pf, l + half, r, Some((m, diff)));
136         }
137     }
138     return inner(pf, 0, pf.len() - 1, None);
139 }
140
141 fn get_weights(data: &Vec<u8>) -> HashMap<u8, u32> {
142     let mut freq = HashMap::new();
143     data.iter().for_each(|byte| {

```

```

143         *freq.entry(*byte).or_insert(1) += 1;
144     });
145     let mut freq: Vec<u8, u32> = freq.drain().collect();
146     freq.sort();
147     let mut weights = HashMap::new();
148     freq.iter().enumerate().for_each(|(i, (byte, _))| {
149         weights.insert(*byte, i as u32);
150     });
151     return weights;
152 }
153
154 #[derive(Debug)]
155 struct Metadata {
156     weights: HashMap<u8, u32>,
157     tree: FanoTree<u8>,
158     code: HashMap<u8, Vec<Bit>>,
159     remainder: u8,
160 }
161
162 impl Metadata {
163     fn compute(data: &Vec<u8>) -> Self {
164         let weights = get_weights(data);
165         let tree = FanoTree::from_hashmap(&weights);
166         let code = tree.get_code();
167
168         return Self {
169             weights,
170             tree,
171             code,
172             remainder: 0,
173         };
174     }
175
176     fn load(data: &Vec<u8>) -> Self {
177         let remainder = data[0];
178         let dict_len = data[1] as usize + 1;
179         let mut weights = HashMap::new();
180         for i in 0..dict_len {
181             weights.insert(data[2 + 2 * i], data[2 + 2 * i + 1] as u32);
182         }
183         let tree = FanoTree::from_hashmap(&weights);
184         let code = tree.get_code();
185
186         return Self {
187             weights,
188             tree,
189             code,
190             remainder,
191         };
192     }
193
194     fn dump(&self) -> Vec<u8> {
195         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         }
250         self.remainder = remainder;

```



```

251     }
252 }
253
254 pub fn compress(data: &Vec<u8>) -> Vec<u8> {
255     let mut result = Vec::new();
256     let mut metadata = Metadata::compute(data);
257
258     let mut writer = BitWriter::new();
259     data.iter().for_each(|byte| {
260         let bits = metadata.code.get(byte).unwrap();
261         writer.write_bits(&bits);
262     });
263     writer.finish();
264     metadata.remainder = writer.remainder;
265
266     let md_dump = metadata.dump();
267     md_dump.iter().for_each(|byte| result.push(*byte));
268     writer.result.iter().for_each(|byte| result.push(*byte));
269
270     return result;
271 }
272
273 struct BitReader<'a> {
274     data: &'a Vec<u8>,
275     metadata: &'a Metadata,
276     buffer: Vec<Bit>,
277     ptr: usize,
278 }
279
280 impl<'a> BitReader<'a> {
281     fn new(data: &'a Vec<u8>, metadata: &'a Metadata) -> Self {
282         Self {
283             data,
284             metadata,
285             buffer: Vec::new(),
286             ptr: 0,
287         }
288     }
289
290     fn read_byte(&mut self) {
291         if let Some(byte) = self.data.get(self.ptr) {
292             for i in 0..=7 {
293                 let bit = byte & (1 << i);
294                 self.buffer.push(Bit::from_u8(bit));
295             }
296             if self.ptr == self.data.len() - 1 {
297                 for _ in 0..self.metadata.remainder {
298                     self.buffer.pop();
299                 }
300             }
301             self.buffer.reverse();
302             self.ptr += 1;
303         }
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,
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 = fano::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 = fano::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 = "lab2"
3 version = "0.1.0"

```

```
4 edition = "2021"  
5  
6 [dependencies]  
7 clap = { version = "4.0.17", features = ["derive"] }
```