Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: А. С. Бычков
Преподаватель: Н. К. Макаров
Группа: М8О-301Б

Дата:
Оценка:
Подпись:

Москва, 2024

Курсовой проект на тему BWT, MTF, RLE

Задача: Для BWT в конец строки добавляется символ «$» для помощи в дальнейшем декодировании. Это может быть любой символ меньший чем «a». Для MTF используется следующее начальное распределение кодов:
$\$ \Longrightarrow 0$
$a \Longrightarrow 1$
$b \Longrightarrow 2$
$c \Longrightarrow 3$
...
$x \Longrightarrow 24$
$y \Longrightarrow 25$
$z \Longrightarrow 26$
Максимальная длинна текста в тестах 10^5

Формат ввода: Вам будут даны тесты двух типов. Первый тип:

compress

$< text >$

Текст состоит только из малых латинских букв. В ответ на него вам нужно вывести коды, которыми будет закодирован данный текст.

Второй тип:

decompress

$< codes >$

Вам даны коды в которые был сжат текст из малых латинских букв, вам нужно его разжать. В RLE коды записываются в порядке (количество, значение)

Формат вывода: В ответ на первый тип тестов выведите коды, которыми будет закодирован текст. Каждая пара на отдельной строке. На второй тип тестов выведите разжатый текст.

1 Описание

Опишем три класса:

1. *BWT*
2. *MTF*
3. *RLE*

Каждый из них будет иметь два публичных метода: *Encode* и *Decode*.

Реализация этих классов ниже:

```
1  #include <cassert>
2  #include <cstddef>
3  #include <cstdint>
4  #include <iostream>
5  #include <string>
6  #include <sys/types.h>
7  #include <vector>
8  #include <algorithm>
9  #include <numeric>
10 #include <array>
11
12
13 constexpr size_t kAlphabetSize = 256;
14
15 template <typename T>
16 std::ostream& operator<<(std::ostream& os, const std::vector<T> vct) {
17     for (std::size_t i = 0; i < vct.size(); ++i) {
18         os << vct[i];
19         if (i != vct.size() - 1) {
20             os << ", ";
21         }
22     }
23     return os;
24 }
25
26 template <>
27 std::ostream& operator<<(std::ostream& os, const std::vector<uint8_t> vct) {
28     for (std::size_t i = 0; i < vct.size(); ++i) {
29         os << (int)vct[i];
30         if (i != vct.size() - 1) {
31             os << ", ";
32         }
33     }
34     return os;
35 }
36
```

```
37
38 class SuffixArray {
39 private:
40     const std::vector<int32_t> text_;
41     std::size_t n_;
42     std::vector<std::size_t> array_;
43
44     static constexpr std::size_t different_chars = 256;
45
46     inline bool is_sorted(const std::vector<std::size_t>& eq) const {
47         return (n_ * (n_ - 1) / 2) == std::reduce(eq.begin(), eq.end());
48     }
49
50     void build() {
51         std::size_t sz = 1;
52         std::vector<std::size_t> count_sort(std::max(different_chars, n_), 0);
53         std::vector<std::size_t> eq(n_, 0);
54         std::vector<std::size_t> eq_buffer(n_, 0);
55         std::vector<std::size_t> array_buffer(n_);
56
57         for (std::size_t i = 0; i < n_; ++i) {
58             ++count_sort[text_[i]];
59         }
60         for (std::size_t i = 1; i < std::max(different_chars, n_); ++i) {
61             count_sort[i] += count_sort[i - 1];
62         }
63         for (ssize_t i = n_ - 1; i >= 0; --i) {
64             array_[--count_sort[text_[i]]] = i;
65         }
66
67
68         for (std::size_t i = 1; i < n_; ++i) {
69             if (text_[array_[i]] != text_[array_[i - 1]]) {
70                 eq[array_[i]] = eq[array_[i - 1]] + 1;
71             } else {
72                 eq[array_[i]] = eq[array_[i - 1]];
73             }
74         }
75
76         while (!is_sorted(eq)) {
77             std::fill(count_sort.begin(), count_sort.end(), 0);
78             eq_buffer = eq;
79
80             for (std::size_t i = 0; i < n_; ++i) {
81                 array_buffer[i] = (array_[i] + n_ - sz) % n_;
82             }
83
84             for (std::size_t i = 0; i < n_; ++i) {
85                 ++count_sort[eq[array_buffer[i]]];
```

```
            }
            for (std::size_t i = 1; i < n_; ++i) {
                count_sort[i] += count_sort[i - 1];
            }
            for (ssize_t i = n_ - 1; i >= 0; --i) {
                array_[--count_sort[eq[array_buffer[i]]]] = array_buffer[i];
            }

            eq[0] = 0;
            for (std::size_t i = 1; i < n_; ++i) {
                if (
                    eq_buffer[array_[i - 1]] < eq_buffer[array_[i]] ||
                    eq_buffer[array_[i - 1]] == eq_buffer[array_[i]] && eq_buffer[(
                        array_[i - 1] + sz) % n_] < eq_buffer[(array_[i] + sz) % n_]
                ) {
                    eq[array_[i]] = eq[array_[i - 1]] + 1;
                } else {
                    eq[array_[i]] = eq[array_[i - 1]];
                }
            }

            sz <<= 1;
        }
    }

public:

    SuffixArray(const std::vector<int32_t>& text) : text_(text), n_(text_.size()),
        array_(n_, -1) {
        build();
    }

    const std::vector<std::size_t>& get_suffix_array() {
        return array_;
    }
};

std::vector<uint8_t> CountSort(const std::vector<uint8_t>& text) {
    std::array<size_t, kAlphabetSize> counter{};

    for (auto character : text) {
        ++counter[character];
    }

    std::vector<uint8_t> result;
    result.reserve(text.size());
    for (size_t i = 0; i < kAlphabetSize; ++i) {
        for (size_t j = 0; j < counter[i]; ++j) {
            result.push_back(static_cast<uint8_t>(i));
```

```
133         }
134     }
135     return result;
136 }
137 
138 class BWT {
139 
140     std::vector<uint8_t> CalculateEncodedText(const std::vector<uint8_t>& text, const
            std::vector<std::size_t>& suffix_array) {
141         auto n = suffix_array.size();
142         std::vector<uint8_t> result;
143         result.reserve(n);
144 
145         for (const auto& i : suffix_array) {
146             uint8_t last_byte = text[(i + (n - 1)) % n];
147             result.push_back(last_byte);
148         }
149         return result;
150     }
151 
152     size_t CalculateEncodedIndex(const std::vector<uint8_t>& encoded) {
153         return std::find(encoded.begin(), encoded.end(), 0) - encoded.begin();
154     }
155 
156     std::vector<size_t> GetDecodingPermutation(const std::vector<uint8_t>& first, const
             std::vector<uint8_t>& last) {
157         std::array<size_t, kAlphabetSize> counter;
158         std::vector<size_t> result;
159         result.reserve(first.size());
160 
161         assert(first.size() > 0);
162         assert(first.size() == last.size());
163 
164         for (size_t i = 0; i < first.size(); ++i) {
165             uint8_t byte = first[i];
166             counter[byte] = i;
167         }
168 
169         for (ssize_t i = first.size() - 1; i >= 0; --i) {
170             uint8_t byte = last[i];
171             result[counter[byte]--] = i;
172         }
173         return result;
174     }
175 
176     std::vector<int32_t> ConvertToVectorWithSentinel(const std::vector<uint8_t>& text)
            {
177         std::vector<int32_t> result(text.begin(), text.end());
178         return result;
```

```
179     }
180
181 public:
182
183     std::vector<uint8_t> Encode(const std::vector<uint8_t>& text) {
184         SuffixArray suffix_array_factory(ConvertToVectorWithSentinel(text));
185         const auto& suffix_array = suffix_array_factory.get_suffix_array();
186
187         assert(text.size() == suffix_array.size());
188
189         return CalculateEncodedText(text, suffix_array);
190     }
191
192     std::vector<uint8_t> Decode(const std::vector<uint8_t>& encoded) {
193         auto first = CountSort(encoded);
194         auto last = encoded;
195
196         auto permutation = GetDecodingPermutation(first, last);
197
198         size_t index = CalculateEncodedIndex(encoded);
199         std::vector<uint8_t> result;
200         result.reserve(first.size());
201
202         result.push_back(first[index]);
203         for (size_t i = 0; i < first.size() - 2; ++i) {
204             index = permutation[index];
205             result.push_back(first[index]);
206         }
207
208         return result;
209     }
210 };
211
212
213 class MTF {
214     std::array<uint8_t, 27> GetInitialOrder() {
215         std::array<uint8_t, 27> result;
216         for (size_t i = 0; i < 27; ++i) {
217             result[i] = i;
218         }
219         return result;
220     }
221
222 public:
223
224     std::vector<uint8_t> Encode(const std::vector<uint8_t>& text) {
225         auto arr = GetInitialOrder();
226         std::vector<uint8_t> result;
227         result.reserve(text.size());
```

```
228
229         for (auto byte : text) {
230             if ('a' <= byte && byte <= 'z') {
231                 byte -= 'a';
232                 byte += 1;
233             }
234             result.push_back(arr[byte]);
235
236             for (size_t i = 0; i < 27; ++i) {
237                 if (arr[i] < arr[byte]) {
238                     ++arr[i];
239                 }
240
241             }
242             arr[byte] = 0;
243         }
244         return result;
245     }
246
247     std::vector<uint8_t> Decode(const std::vector<uint8_t>& encoded) {
248         auto arr = GetInitialOrder();
249         std::vector<uint8_t> result;
250         result.reserve(encoded.size());
251
252         for (auto byte : encoded) {
253             size_t index_to_skip = -1;
254
255             for (size_t i = 0; i < 27; ++i) {
256                 if (arr[i] == byte) {
257                     result.push_back(i);
258                     arr[i] = 0;
259                     index_to_skip = i;
260                     break;
261                 }
262             }
263
264             for (size_t i = 0; i < 27; ++i) {
265                 if (i == index_to_skip) continue;
266                 if (arr[i] < byte) {
267                     ++arr[i];
268                 }
269             }
270         }
271
272         for (auto& elem : result) {
273             if (elem > 0) {
274                 elem += ('a' - 1);
275             }
276         }
```

```
277
278         return result;
279     }
280 };
281
282 class RLE {
283
284 public:
285     struct Encoded {
286         size_t count;
287         uint8_t byte;
288     };
289
290     std::vector<Encoded> Encode(const std::vector<uint8_t>& text) {
291         std::vector<Encoded> result;
292
293         for (size_t i = 0; i < text.size();) {
294             size_t j = i + 1;
295             while (j < text.size() && text[i] == text[j]) ++j;
296             result.push_back(Encoded{
297                 .count = j - i,
298                 .byte = text[i],
299             });
300             i = j;
301         }
302         return result;
303     }
304
305     std::vector<uint8_t> Decode(const std::vector<Encoded>& encoded) {
306         std::vector<uint8_t> result;
307
308         for (const auto& data : encoded) {
309             for (size_t i = 0; i < data.count; ++i) {
310                 result.push_back(data.byte);
311             }
312         }
313         return result;
314     }
315 };
316
317 int main() {
318     std::ios::sync_with_stdio(false);
319     std::cin.tie(NULL);
320     std::cout.tie(NULL);
321     BWT bwt;
322     MTF mtf;
323     RLE rle;
324
325     std::string type;
```

```
326     std::cin >> type;
327 
328     if (type == "compress") {
329         std::string text;
330         std::cin >> text;
331 
332         std::vector<uint8_t> bytes;
333         bytes.reserve(text.size());
334         for (int i = 0; i <= text.size(); ++i) {
335             bytes.push_back(text[i]);
336         }
337 
338         auto encoded = rle.Encode(mtf.Encode(bwt.Encode(bytes)));
339         for (const auto& data : encoded) {
340             std::cout << data.count << ' ' << (int)data.byte << '\n';
341         }
342     } else {
343         std::vector<RLE::Encoded> encoded;
344         size_t count;
345         size_t byte;
346         while (std::cin >> count && std::cin >> byte) encoded.push_back(RLE::Encoded{.
                count = count, .byte = (uint8_t)byte});
347 
348         auto decoded = bwt.Decode(mtf.Decode(rle.Decode(encoded)));
349         std::string result;
350         for (const auto& d : decoded) {
351             result.push_back(d);
352         }
353 
354         std::cout << result << '\n';
355     }
356 }
```

2 Консоль

```
>g++ bwt_mtf_rle.cpp
>./a.out
compress
abracadabra
1 1
1 18
1 5
1 3
1 2
1 5
```

```
1 4
3 0
1 5
1 0
>./a.out
decompress
1 1
1 18
1 5
1 3
1 2
1 5
1 4
3 0
1 5
1 0
abracadabra
```

3 Тест производительности

Первый тест: Файл сгенерирован следующей программой:

```
import random

def generate_random_letter():
    frequencies = [
        ("e", 0.13), ("a", 0.105), ("o", 0.081), ("n", 0.079), ("r", 0.071),
        ("i", 0.063), ("s", 0.061), ("h", 0.052), ("d", 0.038), ("l", 0.034),
        ("f", 0.029), ("c", 0.027), ("m", 0.025), ("u", 0.024), ("t", 0.24),
        ("g", 0.072), ("p", 0.065), ("w", 0.055), ("b", 0.054), ("v", 0.052),
        ("k", 0.047), ("x", 0.035), ("j", 0.029), ("q", 0.028), ("z", 0.023)
    ]
    letters, weights = zip(*frequencies)
    return random.choices(letters, weights=weights, k=1)[0]

with open('input.txt', 'w') as file:
    for i in range(10_000_000):
        file.write(generate_random_letter())
```

Символы распределены не равномерно, а в соответствии с их реальной частотой.

Процесс кодирования занял $18967ms$, а количество пар в ответе составило 9543055. Даже если взять под кодирование количества 1, то итоговый размер получится практически в два раза больше исходного.

Второй тест:

```
import random

def generate_random_letter():
    frequencies = [
        ("a", 0.25), ("g", 0.25), ("t", 0.25), ("c", 0.25)
    ]
    letters, weights = zip(*frequencies)
    return random.choices(letters, weights=weights, k=1)[0]

with open('input.txt', 'w') as file:
    for i in range(10_000_000):
        file.write(generate_random_letter())
```

В этом примере алфавит состоит всего из 4 букв (нуклеотиды цепочки ДНК).

Процесс кодирования занял $18532ms$, а количество пар итоге вышло 7500583. Опять же, сжатия практически никакого не произошло, если учитывать, что нам нужна память и под символ и под его количество.

Кроме того, те же самые тесты были проверены при длине текста в $100'000, 1'000'000$ символов. Время получилось 97 и 1441 ms соответственно. Здесь проглядывается зависимость $O(nlog_2 n)$, которая получается благодаря построению суффиксного мас-

сива.

4 Выводы

Благодаря данной курсовой работе я узнал про алгоритмы, которые могут помочь в сжатии текста. К сожалению, на проверенных мной тестах, алгоритмы сжатия никакого не делали.

Однако, чем меньше мощность алфавита, тем более эффективным становится алгоритм.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))