Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

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

Москва, 2024

Лабораторная работа №5

Задача: Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

Примечание: Можно делать как через суффиксное дерево, так и через сортировку.

1 Описание

Суффиксный массив для строки $S[1...n]$ - массив целых чисел от 1 до n, такой, что i суффикс $S[i...n]$ исходной строки s лексикографически упорядочен. Т.е. до него стоят суффиксы, которые лексикографически меньше, а после - лексикографически больше.

Для построения такого массива воспользуемся алгоритмом, который работает за $O(n * log(n))$. Сам алгоритм таков:

1. Добавим в конец строки сентинел. Причем, сентинел должен быть лексикографически меньше любого символа алфавита, используемого в строке. В $C++$ для $std :: string$ гарантируется, что последний символ имеет код 0. Его можно использовать в качестве сентинела.

2. Теперь мысленно дополним каждый суффикс до длины, равной степени двойки. Причем, мы дополним его циклическим сдвигом нашей строки. Например, если у нас строка вместе с сентинелом выглядит так: $abcd\$$, где \$ - сентинел, то для суффикса $cd\$$ мы получим строку длины 8: $cd\$abcd\$$. Очевидно, что новые строки лексикографически останутся в том же порядке.

3. Теперь, будем строить эти суффиксы поэтапно. Сначала для каждого суффикса возьмем префикс длины 1. После длины $2, 4, ..., 2^{log(n)}$.

   (a) Для длины 1 все очевидно. Просто отсортируем все буквы строки сортировкой подсчетом. Назовем этот массив p.

   (b) Теперь построим вспомогательный массив - eq. Это классы равенства. Если $p[i] = p[i-1]$, то $eq[i] = eq[i-1]$, иначе - $eq[i] = eq[i-1] + 1$. $eq[0] = 0$ - начальное условие.

   (c) Предподсчет выше - база. Теперь будет делать переход от k к $2 * k$.

      i. На новом шаге мы должны получить массив p, в котором будут префиксы длины $2^{log(k)+1}$ всех суффиксов, а так же, массив eq для нового массива p.

      ii. Первым делом строим вспомогательный массив p', где $p'[i] = (i - 2^{log(k)})\%(n+1)$.

      iii. Массив p' содержит индексы начала всех циклических префиксов, длины $2^{log(k)+1}$, суффиксов строки s с дописанной к ней сентинелом.

      iv. Массив p' отсортирован по правой половине циклического префикса $p'[i]$. Нам осталось отсортировать по лейвой половине. Здесь можно привести такую аналогию: У нас есть набор двузначных чисел, который отсортированы по разряду единиц (десятки могут быть не упорядочены). Мы хотим отсортировать этот набор по десяткам, сохранив

сортировку по единицам, чтобы получить полностью отсортированный набор двузначных чисел.

v. Делаем сортировку подсчетом, где ключом будет являться $eq[p'[i]]$, где eq - массив классов равенства с предыдущего шага. После этого шага мы получим верно отсортированный массив p суффиксов для шага $k+1$.

vi. Осталось пересчитать массив eq для шага $k+1$. Для этого просто идем по массиву p (который для шага $k+1$) и лексикографически сравниваем пары $\{eq[p[i]], eq[(p[i]+2^{log(k)})\%(n+1)]\}$.

Когда мы получили суффиксный массив надо научиться искать в нем паттерн. Для этого можно реализовать наивный алгоритм с , описанный в [2], который будет работать за $O((log(n)+k)*|p|)$, где k - количество вхождений паттерна в текст.

Но, этот алгоритм можно значительно ускорить, используя lcp. Это ускорение описано в [1]. Для построения lcp для соседних (в суффиксном массиве) суффиксов можно использовать алгоритм Касаи, Аримуры, Арикавы, Ли, Парка [3]. Чтобы получить lcp для любой пары суффиксов, над массивом полученным в алгоритме Касаи можно построить Дерево Отрезков или любую другую структуру, которая решает RMQ. Такой поиск (в случае с деревом отрезков) будет работать со сложностью $O(log^2(n)+|p|+k)$. Если же вместо Дерева Отрезков использовать структуру, которая позволит получать RMQ за $O(1)$, то можно достичь сложности поиска $O(log(n)+|p|+k)$.

2 Исходный код

```cpp
#include <algorithm>
#include <cassert>
#include <cstddef>
#include <iostream>
#include <ostream>
#include <string>
#include <vector>
#include <numeric>

template <typename T>
std::ostream& operator<<(std::ostream& os, const std::vector<T> vct) {
    for (std::size_t i = 0; i < vct.size(); ++i) {
        os << vct[i];
        if (i != vct.size() - 1) {
            os << ", ";
        }
    }
    return os;
}

class SegmentTree {
private:
    std::size_t n_;
    std::vector<std::size_t> data_;

    void build(
        std::size_t l,
        std::size_t r,
        std::size_t id,
        const std::vector<std::size_t>& vct
    )
    {
        if (l == r) {
            data_[id] = vct[l];
            return;
        }
        std::size_t m = (l + r) / 2;
        std::size_t lid = id * 2 + 1;
        std::size_t rid = id * 2 + 2;

        build(l, m, lid, vct);
        build(m + 1, r, rid, vct);

        data_[id] = std::min(data_[lid], data_[rid]);
    }

    std::size_t get(
```

```
48          std::size_t ql,
49          std::size_t qr,
50          std::size_t l,
51          std::size_t r,
52          std::size_t id
53      )
54      {
55          if (ql <= l && r <= qr) {
56              return data_[id];
57          }
58
59          std::size_t m = (l + r) / 2;
60          std::size_t lid = id * 2 + 1;
61          std::size_t rid = id * 2 + 2;
62
63          if (qr <= m) {
64              return get(ql, qr, l, m, lid);
65          }
66          if (ql > m) {
67              return get(ql, qr, m + 1, r, rid);
68          }
69          return std::min(get(ql, qr, l, m, lid), get(ql, qr, m + 1, r, rid));
70      }
71
72  public:
73
74      SegmentTree() : n_(0), data_() {}
75
76      SegmentTree(const std::vector<std::size_t>& vct) : n_(vct.size()), data_(n_ * 4, 0)
            {
77          build(0, n_ - 1, 0, vct);
78      }
79
80      void build(const std::vector<std::size_t>& vct) {
81          n_ = vct.size();
82          data_.resize(n_ * 4, 0);
83          build(0, n_ - 1, 0, vct);
84      }
85
86      std::size_t get(
87          std::size_t ql,
88          std::size_t qr
89      )
90      {
91          return get(ql, qr, 0, n_ - 1, 0);
92      }
93
94  };
95
```

```
96
97  class SuffixArray {
98  private:
99      const std::string& text_;
100     std::size_t n_;
101     std::vector<std::size_t> array_;
102     std::vector<std::size_t> neighbor_lcp_;
103     SegmentTree st_;
104
105
106     static constexpr std::size_t different_chars = 256;
107
108     inline bool is_sorted(const std::vector<std::size_t>& eq) const {
109         /*      ,
110         *    eq   [0 ... n - 1]
111         */
112         return (n_ * (n_ - 1) / 2) == std::reduce(eq.begin(), eq.end());
113     }
114
115     void build() {
116         std::size_t sz = 1;
117         std::vector<std::size_t> count_sort(std::max(different_chars, n_), 0);
118         std::vector<std::size_t> eq(n_, 0);
119         std::vector<std::size_t> eq_buffer(n_, 0);
120         std::vector<std::size_t> array_buffer(n_);
121
122         /*  */
123         for (std::size_t i = 0; i < n_; ++i) {
124             ++count_sort[text_[i]];
125         }
126         for (std::size_t i = 1; i < std::max(different_chars, n_); ++i) {
127             count_sort[i] += count_sort[i - 1];
128         }
129         for (ssize_t i = n_ - 1; i >= 0; --i) {
130             array_[--count_sort[text_[i]]] = i;
131         }
132         /*  */
133
134
135         /*  eq*/
136         for (std::size_t i = 1; i < n_; ++i) {
137             if (text_[array_[i]] != text_[array_[i - 1]]) {
138                 eq[array_[i]] = eq[array_[i - 1]] + 1;
139             } else {
140                 eq[array_[i]] = eq[array_[i - 1]];
141             }
142         }
143         /*  eq*/
144
```

```
145         while (!is_sorted(eq)) {
146             std::fill(count_sort.begin(), count_sort.end(), 0);
147             eq_buffer = eq;
148 
149             /*  p'*/
150             for (std::size_t i = 0; i < n_; ++i) {
151                 array_buffer[i] = (array_[i] + n_ - sz) % n_;
152             }
153             /*  p'*/
154 
155             /*  p''*/
156             for (std::size_t i = 0; i < n_; ++i) {
157                 ++count_sort[eq[array_buffer[i]]];
158             }
159             for (std::size_t i = 1; i < n_; ++i) {
160                 count_sort[i] += count_sort[i - 1];
161             }
162             for (ssize_t i = n_ - 1; i >= 0; --i) {
163                 array_[--count_sort[eq[array_buffer[i]]]] = array_buffer[i];
164             }
165             /*  p''*/
166 
167             /*  eq*/
168             eq[0] = 0;
169             for (std::size_t i = 1; i < n_; ++i) {
170                 if (
171                     eq_buffer[array_[i - 1]] < eq_buffer[array_[i]] ||
172                     eq_buffer[array_[i - 1]] == eq_buffer[array_[i]] && eq_buffer[(
                            array_[i - 1] + sz) % n_] < eq_buffer[(array_[i] + sz) % n_]
173                 ) {
174                     eq[array_[i]] = eq[array_[i - 1]] + 1;
175                 } else {
176                     eq[array_[i]] = eq[array_[i - 1]];
177                 }
178             }
179             /*  eq*/
180 
181             sz <<= 1;
182         }
183     }
184 
185     bool is_suffix_lower(std::size_t suffix_id, const std::string& pattern) {
186         for (std::size_t i = 0; i < pattern.size() ;++i) {
187             if (i > 15) break;
188             if (text_[suffix_id + i] > pattern[i]) {
189                 return false;
190             }
191             if (text_[suffix_id + i] < pattern[i]) {
192                 return true;
```

```
            }
        }
        return false;
    }

    bool is_suffix_starts_with(std::size_t suffix_id, const std::string& start) {
        for (std::size_t i = 0; i < start.size(); ++i) {
            if (text_[suffix_id + i] != start[i]) return false;
        }
        return true;
    }

    void kasai() {
        std::vector<std::size_t> lcp(n_ - 1, 0);
        std::vector<std::size_t> array_inverse(n_, 0);

        for (std::size_t i = 0; i < n_; ++i) {
            array_inverse[array_[i]] = i;
        }

        std::size_t k = 0;

        for (std::size_t i = 0; i < n_ - 1; ++i) {


            if (k > 0) {
                --k;
            }
            std::size_t j = array_[array_inverse[i] - 1];

            assert(j >= 0);
            assert(j < n_);

            while (text_[j + k] == text_[i + k]) ++k;

            assert(array_inverse[i] - 1 >= 0);
            assert(array_inverse[i] - 1 < n_ - 1);

            lcp[array_inverse[i] - 1] = k;
        }
        neighbor_lcp_ = lcp;
    }

public:

    SuffixArray(const std::string& text) : text_(text), n_(text_.size() + 1), array_(n_
        , -1), neighbor_lcp_(n_ - 1), st_() {
        build();
        kasai();
```

```
241         st_.build(neighbor_lcp_);
242     }
243 
244 
245     const std::vector<std::size_t>& get_suffix_array() {
246         return array_;
247     }
248 
249     std::vector<std::size_t> find_all(const std::string& pattern) {
250         /* O(|si|logn_ + k|si|),  k -   */
251         std::vector<std::size_t> res;
252         if (is_suffix_lower(array_[n_ - 1], pattern)) {
253             return res;
254         }
255         std::size_t l = 0, r = n_ - 1;
256 
257         while (l < r - 1) {
258             std::size_t m = (l + r) / 2;
259             if (is_suffix_lower(array_[m], pattern)) {
260                 l = m;
261             } else {
262                 r = m;
263             }
264         }
265         for (std::size_t i = r; i < n_; ++i) {
266             if (is_suffix_starts_with(array_[i], pattern)) {
267                 res.push_back(array_[i] + 1);
268             } else {
269                 break;
270             }
271         }
272         std::sort(res.begin(), res.end());
273 
274         return res;
275     }
276 
277 
278 
279     std::vector<std::size_t> find_all_fast(const std::string& pattern) {
280         /* O(|si|logn_ + k),  k -   */
281 
282         std::vector<std::size_t> res;
283         if (is_suffix_lower(array_[n_ - 1], pattern)) {
284             return res;
285         }
286         std::size_t l = 0, r = n_ - 1;
287 
288         while (l < r - 1) {
289             std::size_t m = (l + r) / 2;
```

```
290             if (is_suffix_lower(array_[m], pattern)) {
291                 l = m;
292             } else {
293                 r = m;
294             }
295         }
296 
297         if (!is_suffix_starts_with(array_[r], pattern)) {
298             return res;
299         }
300         res.push_back(array_[r] + 1);
301 
302         for (std::size_t i = r + 1; i < n_; ++i) {
303 
304             /*  ...   lcp  .*/
305             if (neighbor_lcp_[i - 1] >= pattern.size()) {
306                 res.push_back(array_[i] + 1);
307             } else {
308                 break;
309             }
310         }
311         std::sort(res.begin(), res.end());
312 
313         return res;
314     }
315 
316     std::size_t lcp_with_suffix_naive(std::size_t suffix_ind, const std::string&
        pattern) {
317         for (std::size_t i = 0; i < pattern.size(); ++i) {
318             if (text_[array_[suffix_ind] + i] != pattern[i]) return i;
319         }
320         return pattern.size();
321     }
322 
323     std::size_t common_part_from(std::size_t start, std::size_t suffix_ind, const std::
        string& pattern) {
324         std::size_t k = 0;
325         while (start + k < pattern.size() && text_[array_[suffix_ind] + start + k] ==
            pattern[start + k]) ++k;
326         return k;
327     }
328 
329     std::vector<std::size_t> find_all_super_fast(const std::string& pattern) {
330         /* O(|si| + log^2n_ + k),  k -   */
331 
332         std::vector<std::size_t> res;
333         if (is_suffix_lower(array_[n_ - 1], pattern)) {
334             return res;
335         }
```

```
336         std::size_t L = 0, R = n_ - 1;
337         std::size_t l = lcp_with_suffix_naive(L, pattern);
338         std::size_t r = lcp_with_suffix_naive(R, pattern);
339 
340 
341         while (L < R - 1) {
342             std::size_t M = (L + R) / 2;
343             std::size_t ml = st_.get(L, M - 1);
344             std::size_t mr = st_.get(M, R - 1);
345 
346             if (l >= r) {
347                 if (ml > l) {
348                     L = M;
349                 } else if (ml < l) {
350                     R = M;
351                     r = ml;
352                 } else {
353                     std::size_t k = common_part_from(l, M, pattern);
354                     if (l + k == pattern.size()) {
355                         R = M;
356                         r = pattern.size();
357                     } else if (text_[array_[M] + l + k] < pattern[l + k]) {
358                         L = M;
359                         l = l + k;
360                     } else if (text_[array_[M] + l + k] > pattern[l + k]) {
361                         R = M;
362                         r = l + k;
363                     }
364                 }
365             } else {
366                 if (mr > r) {
367                     R = M;
368                 } else if (mr < r) {
369                     L = M;
370                     l = mr;
371                 } else {
372                     std::size_t k = common_part_from(r, M, pattern);
373                     if (r + k == pattern.size()) {
374                         R = M; //   , ..      ,    >=
375                         r = pattern.size();
376                     } else if (text_[array_[M] + r + k] < pattern[r + k]) {
377                         L = M;
378                         l = r + k;
379                     } else if (text_[array_[M] + r + k] > pattern[r + k]) {
380                         R = M;
381                         r = r + k;
382                     }
383                 }
384             }
```

```
385         }
386 
387         if (!is_suffix_starts_with(array_[R], pattern)) {
388             return res;
389         }
390         res.push_back(array_[R] + 1);
391 
392         for (std::size_t i = R + 1; i < n_; ++i) {
393 
394             /*  ...   lcp  .*/
395             if (neighbor_lcp_[i - 1] >= pattern.size()) {
396                 res.push_back(array_[i] + 1);
397             } else {
398                 break;
399             }
400         }
401         std::sort(res.begin(), res.end());
402 
403         return res;
404     }
405 };
406 
407 int main() {
408     std::ios::sync_with_stdio(false);
409     std::cin.tie(NULL);
410 
411     std::string text, pattern;
412     std::cin >> text;
413     SuffixArray sa(text);
414 
415 
416     std::size_t i = 0;
417     while(std::cin >> pattern) {
418         ++i;
419         const auto& vct = sa.find_all_super_fast(pattern);
420 
421         if (vct.empty()) continue;
422         std::cout << i << ": " << vct << '\n';
423     }
424 
425     return 0;
426 }
```

3 Консоль

```
>g++ suffix-array.cpp
>./a.out
abaabb
ab
aabb
cd
1: 1,4
2: 3
```

4 Тест производительности

Тест производительности представляем из себя следующее:

1. Есть скрипт $data - gen.py$, который принимает 4 параметра:
   (a) Длину текста
   (b) Длину паттернов
   (c) Количество паттернов
   (d) Алфавит
2. Далее, запускается алгоритм с использование суффиксного массива
3. А после - наивный алгоритм.

```
>python3 data-gen.py
100000 4 10000 abcdef
>g++ benchmark.cpp
>./a.out <output.txt
SuffixArray: 334 ms
Naive: 11927 ms

>python3 data-gen.py
100000 4 100000 abcdef
>g++ benchmark.cpp
>./a.out <output.txt
SuffixArray: 2934 ms
Naive: 120362 ms
```

Как видно, при длине текста в $100'000$ символов, и $10'000$ паттернов, каждый из которых длины 4, суффиксный массив выигрывает в 35 раз! Причем, большую часть времени занимает именно построение массива. В этом можно убедиться, увеличив количество паттернов до $100'000$ и увидев, что разница уже в 41 раз.

5 Выводы

Благодаря этой лабораторной работе я узнал о еще одном методе поиска подстроки в строке. Суффиксный Массив - очень мощная структура в случае, когда надо в одном тексте искать множество паттернов, которые могут приходить *online*. Еще одно преимущество Суффиксного Массива - его простота, если сравнивать с Суффиксным Деревом.

Эта структура данных может помочь мне в спортивном программировании, т.к. задачи, которые она решает часто встречаются, да и написать эту структуру не составит особого труда.

Список литературы

[1] Гастфилд Дэн *Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология* — СПб.: «Невский Диалект», 2003. Перевод с английского: И. В. Романовский. — 654 с. (ISBN 5-7940-0103-8 (рус.))

[2] *Алгоритм поиска подстроки в строке с помощью суффиксного массива - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_поиска_подстроки_в_строке`
(дата обращения: 20.09.2024).

[3] *Алгоритм Касаи и др. - ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Касаи_и_др.`
(дата обращения: 20.09.2024).