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: Д. М. Чистяков
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-20
Дата:
Оценка:
Подпись:

Москва, 2023

Лабораторная работа №5

Задача:

Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

Формат ввода
Некий разрез циклической строки.

Формат вывода
Минимальный в лексикографическом смысле разрез.

1 Описание

Алгоритм Укконена -это линейно-временной онлайн-алгоритм построения деревьев суффиксов, предложенный Эско Укконеном в 1995 году. Алгоритм начинается с неявного дерева суффиксов, содержащего первый символ строки. Затем он проходит по строке, добавляя последовательные символы до тех пор, пока дерево не будет завершено.

Класс суффиксного дерева, для детерменированности описания состояния, в котором оно нахоодится в лююой момент времени, имеет следующие параметры:
активная точка представляет тройку (active node, active edge, active len)
remainder представляет собой количество новых суффиксов, которые нужно вставить

Основные правила, которые используются при добавлении в дерево нового символа:
Правило 1, после вставки из корня:
- active node остается корнем
- active edge становится первым символом нового суффикса, который нужно вставить, т.е. b
- active len уменьшается на 1

Правило 2
Если ребро разделяется и вставляется новая вершина, и если это не первая вершина, созданная на текущем шаге, ранее вставленная вершина и новая вершина соединяются через специальный указатель, суффиксную ссылку.

Правило 3
После разделения ребра из active node, которая не является корнем, переходим по суффиксной ссылке, выходящей из этой вершины, если таковая имеется active node устанавливается вершиной, на которую она указывает. Если суффиксная ссылка отсутствует, active node устанавливается корнем.active edge и active len остаются без изменений.

2 Исходный код

```
1  void SuffixTree::build() {
2    to_first_suff = new Node(0, 0, root, true);
3    to_first_suff->n_r = &n_p;
4    to_first_suff->num_of_leaf = 0;
5    last_node = to_first_suff;
6    root->childs.insert(std::make_pair(string[0], to_first_suff));
7
8    int start_in_new_phase = 1;
9    int case_number = 1;
10   std::pair<Node*, int> res_case3;
11   for (int i = 1; i <= string.size() - 1; i++) {
12     n_p++;
13     do_phase(start_in_new_phase, i, case_number, res_case3);
14   }
15 }
16
17 void SuffixTree::do_phase(int &start_in_new_phase, int end, int &case_number, std::
       pair<Node*, int> &res_case3) {
18   bool do_link = false;
19   for (int j = start_in_new_phase; j <= end; j++) {
20     std::pair<Node*, int> pos;
21     if (case_number == 3) {
22       if (res_case3.second != 0) {
23         pos = std::make_pair(res_case3.first, res_case3.second);
24       } else {
25         pos = find_path(end, end, res_case3.first);
26       }
27     } else {
28       pos = analysis_links(j, end);
29     }
30     case_number = analysis(pos, end);
31
32     if (case_number == 2) {
33       std::pair<Node*, Link> res_of_case = case2(pos, j, end, do_link);
34       last_node = res_of_case.first;
35       if (res_of_case.second == DO_LINK) {
```

```
36        do_link = true;
37      } else if (res_of_case.second == DO_LINK_TO_ROOT) {
38        last_node->suff_link = root;
39        do_link = false;
40      } else {
41        do_link = false;
42      }
43      start_in_new_phase++;
44    } else if (case_number == 3) {
45      res_case3 = case3(pos, do_link);
46      last_node = res_case3.first;
47      break;
48    }
49  }
50 }
51
52 std::pair<Node*, int> SuffixTree::analysis_links(int start, int end) {
53   if (last_node->parent == root || last_node->suff_link == root || last_node == root)
         {
54     return find_path(start, end, root);
55   } else {
56     if (last_node->suff_link != nullptr) {
57       return find_path(end, end, last_node->suff_link);
58     } else {
59       int d = last_node->len();
60       return find_path(end - d, end, last_node->parent->suff_link);
61     }
62   }
63 }
64
65 std::pair<Node*, int> SuffixTree::find_path(int start, int end, Node* it) {
66   if (it->childs.count(string[start]) == 1) {
67     int d = it->childs[string[start]]->len() - (end - start);
68     if (d >= 0) {
69       return std::make_pair(it->childs[string[start]], end - start);
70     } else {
71       return find_path(start + it->childs[string[start]]->len(), end,
72       it->childs[string[start]]);
73     }
74   } else {
75     return std::make_pair(it, -1 * (end - start + 1));
76   }
77
78 }
79
80 int SuffixTree::analysis(std::pair<Node* , int> &pos, int end) {
81   Node* node = pos.first;
82   int p = pos.second;
83
```

```
  if (p < 0) {
    return 2;
  } else {
    if (node->len() == p && node->childs.size() > 1) {
      if (node->childs.count(string[end]) > 0) {
        pos.first = node->childs[string[end]];
        pos.second = 0;
        return 3;
      }
      return 2;
    } else if (node->len() > p) {
      if (string[node->l + p] == string[end]) {
        return 3;
      } else {
        return 2;
      }
    } else {
      return 3;
    }
  }
}

std::pair<Node*, SuffixTree::Link> SuffixTree::case2(const std::pair<Node*, int> &pos,
     int start, int end, bool do_link) {
  auto node = pos.first;
  auto p = pos.second;

  if (p >= 0) {
    if (p != node->len()) {
      Node* n = new Node(node->l, node->l + p - 1, node->parent, false);
      n->childs.insert(std::make_pair(string[node->l + p], node));
      node->parent->childs.erase(string[node->l]);
      node->parent->childs.insert(std::make_pair(string[n->l], n));
      node->l += p;
      node->parent = n;
      Node* nl = new Node(end, end, n, true);
      nl->n_r = &n_p;
      nl->num_of_leaf = start;
      n->childs.insert(std::make_pair(string[end], nl));

      if (do_link) {
        last_node->suff_link = n;
      }
      if (n->len() > 1) {
        return std::make_pair(n, DO_LINK);
      } else {
        if (n->parent != root) {
          return std::make_pair(n, DO_LINK);
        }
```

```
132         return std::make_pair(n, DO_LINK_TO_ROOT);
133       }
134     } else {
135       Node* nl = new Node(end, end, node, true);
136       nl->n_r = &n_p;
137       nl->num_of_leaf = start;
138       nl->parent->childs.insert(std::make_pair(string[end], nl));
139 
140       if (do_link) {
141         last_node->suff_link = nl->parent;
142       }
143       return std::make_pair(nl->parent, DONT_LINK);
144     }
145   } else {
146     p *= -1;
147     Node* n = new Node(end - p + 1, end, node, true);
148     n->n_r = &n_p;
149     n->num_of_leaf = start;
150     n->parent->childs.insert(std::make_pair(string[end - p + 1], n));
151     if (do_link) {
152       last_node->suff_link = n->parent;
153     }
154     return std::make_pair(n->parent, DONT_LINK);
155   }
156 }
157 
158 std::pair<Node*, int> SuffixTree::case3(const std::pair<Node*, int> &pos, bool do_link
      ) {
159   if (do_link) {
160     last_node->suff_link = pos.first->parent;
161   }
162   auto node = pos.first;
163   auto p = pos.second;
164   int case3_new_j;
165   if (node->l + p == node->end()) {
166     case3_new_j = 0;
167   } else {
168     case3_new_j = p + 1;
169   }
170   return std::make_pair(node, case3_new_j);
171 }
```

3 Консоль

```
den@vbox:~/Документы/DA/lab5$ ./a.out
xabcd
abcdx
den@vbox:~/Документы/DA/lab5$ ./a.out
abxabc
abcabx
```

4 Тест производительности

Сравним полученный алгоритм с наивным алгоритмом. Наивный алгоритм просто перебирает все возможные варианты разрезов и ищет среди них лексикографически наименьший.

```
den@vbox:~/Документы/DA/lab5$ ./a.out <test500
Time: 0.001716 ms
den@vbox:~/Документы/DA/lab5$ ./naive <test500
Time: 0.00017 ms
den@vbox:~/Документы/DA/lab5$ ./a.out <test1e4
Time: 0.0329882 ms
den@vbox:~/Документы/DA/lab5$ ./naive <test1e4
Time: 0.0042473 ms
den@vbox:~/Документы/DA/lab5$ ./a.out <test1e6
Time: 4.08355 ms
den@vbox:~/Документы/DA/lab5$ ./naive <test1e6
Time: 73.1969 ms
```

Видим, что на маленьких строках суффиксное дерево немного уступает наивному алгоритму. Это связано с тем, что построение дерева тоже требует времени. Однако на большой строке видно все преимущество суффиксных деревьев - выигрыш примерно в 18 раз. Это связано с тем, что наивный алгоритм работает за $O(n^2)$ - мы проверям n разрезов, каждый из которых сравниваем с текущим минимальным, сравнение строк происходит за $O(n)$.

5 Выводы

Выполнив пятую лабораторную работу по курсу Дискретный анализ, я познакомился с такой структурой данных как суффиксное дерево. Основное ее предназначение - поиск образцов в тексте. Поиск отличается от всеми известных алгоритмов КМП, Ахо-Корасик и др. тем, что мы предобрабатываем текст, а не образец. Однако у суффиксных деревьев есть еще много применений. Благодаря ему мы можем за линейное время найти наибольшую общую подстроку, можем рассчитать статистику совпадений, построить суффиксный массив или найти минимальный лексикографический разрез.

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.))