Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5 по курсу «Дискретный анализ»**

Студент: Е. А. Кондратьев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19
Дата:
Оценка:
Подпись:

Москва, 2021

# Лабораторная работа №5

**Задача: Вариант №5**

Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из выходных строк, необходимо воспользоваться полученным суффисным деревом для решения своего варианта задания.

Алфавит строк: строчные буквы латинского алфавита (т.е. от a до z).

**Вариант:**

Найти самую длинную общую подстроку двух строк.

**Формат входных данных**

Две строки.

**Формат результата**

На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

# 1 Описание

Требуется написать реализацию алгоритма Укконена для построения суффиксного дерева, затем с помощью него построить суффиксный массив, посредством которого будет происходить поиск образцов в тексте.

Основная идея алгоритма Укконена заключается в том, чтобы построить суффиксное дерево за линейное время.

Для этого нужно, во-первых, использовать линейное количество памяти, поэтому в рёбрах (вершинах) будем хранить два числа — позиции самого левого и самого правого символов в исходном тексте.

Во-вторых, по мере создания дерева будем создавать и использовать суффиксные ссылки, основой для которых является факт, если какая-либо строка уже была добавлена в дерево, то и все ее суффиксы тоже присутствуют в дереве. То есть суффиксной ссылкой будет являться ссылка от вершины xa в вершину a, где x – первым символ строки, a – оставшаяся подстрока (возможно пустая). Использование суффиксных ссылок позволяет пропускать ненужные сравнения.

Суффиксный массив для строки s в i-ой позиции хранит индекс начала i-го (в лексикографическом порядке) суффикса строки s. Будем строить данный массив путём обхода в глубину суффиксного дерева в лексикографическом порядке.

Сложность O(t), где t – длина текста. Поиск образцов в суффиксном массиве будем осуществлять бинарным поиском (поскольку суффиксы отсортированы лексикографически). Сложность $O(p \cdot \log(t))$, где t, p – длины текста и образца соответственно.

Таким образом, за O(t) времени и памяти мы строим суффиксное дерево с помощью алгоритма Укконена, затем за O(t) создаём суффиксный массив, в котором за суммарное время $O(\log(t) \cdot (p1 + p2 + ... + pm))$ ищем m образцов. Итоговая сложность всех операций $O(t + \log(t) \cdot (p1 + p2 + ... + pm))$.

## Код

В suffix_tree.h опишем классы суффиксного дерева и его вершин. В вершине будем хранить итераторы на начала и конец ребра, суффиксную ссылку, ука@ателя на следующие вершины будем хранитт с std::map с ключом в виде первого символа строки.

```
namespace NSuffixTrees {

  class TSuffixArray;

  class TNode {
  public:
    TNode(std::string::iterator begin, std::string::iterator end);
    ~TNode() = default;

    std::string::iterator begin;
    std::string::iterator end;

    std::map< char, TNode* > to;
    TNode* suffixLink;
  };

  class TSuffixTree {
  public:
    TSuffixTree(std::string str);
    ~TSuffixTree();
    friend TSuffixArray;

  private:
    std::string text;
    TNode* root;

    TNode* needSufLink;
    TNode* activeNode;
    int remainder;
    int activeLen;
    std::string::iterator activeEdge;

    void TreeExtend(std::string::iterator toAdd);
    void DeleteTree(TNode* node);

    int GetEdgeLen(TNode* node, std::string::iterator pos) const;
    void AddSuffixLink(TNode* node);
    void DFS(TNode* node, std::vector<size_t>& result, size_t depth) const;
  };
}
```

В suffix_tree.cpp напишем реализации методов классов TNode и TSuffixTree. TreeExtend представляет собой шаг алгоритма Укконена по добавления очередной буквы toAdd в дерево. AddSuffixLink используется для создания суффиксных ссылок. DFS используется для обхода в глубину и заполнения суффиксного массива result.

```
namespace NSuffixTrees {
  //// TNode
  TNode::TNode(std::string::iterator begin, std::string::iterator end) :
    begin(begin),
    end(end),
    suffixLink(0)
  {}
```

```
8
9   //// TSuffixTree
10  TSuffixTree::TSuffixTree(std::string str) :
11    text(str),
12    root(new TNode(text.end(), text.end())),
13    remainder(0)
14  {
15    activeEdge = text.begin();
16    activeNode = needSufLink = root->suffixLink = root;
17    activeLen = 0;
18
19    for (std::string::iterator it = text.begin(); it != text.end(); ++it) {
20      TreeExtend(it);
21    }
22  }
23
24  TSuffixTree::~TSuffixTree() {
25    DeleteTree(root);
26  };
27
28  void TSuffixTree::DeleteTree(TNode* node) {
29    for (std::map< char, TNode* >::iterator it = node->to.begin(); it != node->to.end()
          ; ++it) {
30      DeleteTree(it->second);
31    }
32    delete node;
33  }
34
35  int TSuffixTree::GetEdgeLen(TNode* node, std::string::iterator pos) const {
36    return std::min(node->end, pos + 1) - node->begin;
37  }
38
39  void TSuffixTree::TreeExtend(std::string::iterator toAdd) {
40    needSufLink = root;
41    ++remainder;
42
43    while (remainder) {
44      if (!activeLen) activeEdge = toAdd;
45
46      TNode *next = NULL;
47      std::map< char, TNode* >::iterator it = activeNode->to.find(*activeEdge);
48      if (it != activeNode->to.end()) next = it->second;
49
50      if (!next) {
51        TNode* leaf = new TNode(toAdd, text.end());
52        activeNode->to[*activeEdge] = leaf;
53        AddSuffixLink(activeNode);
54      } else {
55        if (activeLen >= GetEdgeLen(next, toAdd)) {
56          activeEdge += GetEdgeLen(next, toAdd);
57          activeLen -= GetEdgeLen(next, toAdd);
58          activeNode = next;
59          continue;
60        }
61
62        if (*(next->begin + activeLen) == *toAdd) {
63          ++activeLen;
64          AddSuffixLink(activeNode);
65          break;
66        }
67
```

```
68          TNode* split = new TNode(next->begin, next->begin + activeLen);
69          TNode* leaf = new TNode(toAdd, text.end());
70          activeNode->to[*activeEdge] = split;
71
72          split->to[*toAdd] = leaf;
73          next->begin += activeLen;
74          split->to[*next->begin] = next;
75          AddSuffixLink(split);
76        }
77
78        --remainder;
79        if (activeNode == root && activeLen > 0) {
80          --activeLen;
81          activeEdge = toAdd - remainder + 1;
82        } else {
83          if (activeNode->suffixLink) {
84            activeNode = activeNode->suffixLink;
85          } else {
86            activeNode = root;
87          }
88        }
89      }
90    }
91
92    void TSuffixTree::AddSuffixLink(TNode* node) {
93      if (needSufLink != root) needSufLink->suffixLink = node;
94      needSufLink = node;
95    }
96
97    void TSuffixTree::DFS(TNode* node, std::vector<size_t>& result, size_t depth) const
          {
98      if (node->to.empty()) {
99        result.push_back(text.size() - depth);
100       return;
101     }
102     for (std::map<char, TNode*>::iterator it = node->to.begin(); it != node->to.end();
            ++it) {
103       DFS(it->second, result, depth + it->second->end - it->second->begin);
104     }
105   }
106 }
```

suffix_array.h опишем класс суффиксного массива с конструктором от суффиксного дерева.

```
1  namespace NSuffixTrees {
2    class TSuffixTree;
3
4    class TSuffixArray {
5    public:
6      TSuffixArray(const TSuffixTree& tree);
7      ~TSuffixArray() = default;
8
9      std::vector<size_t> Find(const std::string& pattern);
10   private:
11     std::string text;
12     std::vector<size_t> array;
13   };
14 }
```

В suffix_array.cpp напишем соответствующие реализации.

```
namespace NSuffixTrees {
  TSuffixArray::TSuffixArray(const TSuffixTree& tree) :
    text(tree.text),
    array()
  {
    tree.DFS(tree.root, array, 0);
  }

  std::vector<size_t> TSuffixArray::Find(const std::string& pattern) {
    std::pair<std::vector<size_t>::iterator, std::vector<size_t>::iterator> range(array
        .begin(), array.end());
    for (size_t i = 0; i < pattern.size() && range.first != range.second; ++i) {
      range = equal_range(range.first, range.second, std::numeric_limits<size_t>::max()
          , [this, &pattern, &i] (size_t index1, size_t index2) -> bool {
        if (index1 == std::numeric_limits<size_t>::max()) {
          return pattern[i] < text[i + index2];
        } else {
          return text[i + index1] < pattern[i];
        }
      });
    }

    std::vector<size_t> result(range.first, range.second);
    std::sort(result.begin(), result.end());

    return result;
  }
}
```

# 2 Консоль

du@Du$ ./wrapper.sh [2021-06-17 15:53:13] [INFO] Compiling... g++ -std=c++17 -O3 -pedantic -Wall -Wno-unused-variable -c main.cpp -o main.o g++ -std=c++17 -O3 -pedantic -Wall -Wno-unused-variable -c suffix_tree.cpp -o suffix_tree.o g++ -std=c++17 -O3 -pedantic -Wall -Wno-unused-variable -c suffix_array.cpp -o suffix_array.o g++ -std=c++17 -O3 -pedantic -Wall -Wno-unused-variable main.o suffix_tree.o suffix_array.o -o solution [2021-06-17 15:53:15] [INFO] Executing tests/t01.t... OK [2021-06-17 15:53:15] [INFO] Executing tests/t02.t... OK [2021-06-17 15:53:15] [INFO] Executing tests/t03.t... OK [2021-06-17 15:53:15] [INFO] No failed tests,hooray du@Du$ cat tests/t02.t wowCanYouFindMeHere Here You Can du@Du$ ./solution <tests/t02.t 1: 16,22,29 2: 7,36 3: 4,33

# 3 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я познакомился с суффиксными деревьями и суффиксными массивами, а также как работать с ними. Пригодился опыт прошлых лабораторных работ с сбалансированными деревьями: перемещение по дереву и создание новых вершин не представлило трудности. В процессе отладки программы на тестах вскрылись многие баги и ошибки, путём исправления которых получилось доработать изначальный алгоритм.

# Список литературы

[1] *Поисковик - Google.*
URL: `https://www.google.com/`

[2] *Сайт с подробной документацией библиотек C++*
URL: `https://en.cppreference.com/`

[3] *Алгоритм Укконена — ИТМО Вики.*
URL: `https://neerc.ifmo.ru/`