# 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: М. А. Инютин
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19

Дата:
Оценка:
Подпись:


**Москва, 2021**

# Лабораторная работа №5

**Задача:** 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:** линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

# 1 Описание

Требуется реализовать алгоритм Укконена построения суффиксного дерева за линейное время.

Алгоритм Укконена в самой простоей реализации имеет сложность $O(n^3)$, так как добавляет каждый суффикс каждого префикса строки в отличии от добавления всех суффиксов строки. Основная идея в том, чтобы оптимизировать его и получить сложность $O(n)$.

Заметим, что проще будет продлевать суффиксы на один символ. В некоторых случаях при продлении можно идти не по всем символам, а сразу по ребру дерева, что значительно уменьшает число шагов.

Пусть в суффиксном дереве есть строка $x\alpha$ ($x$ — первый символ строки, $\alpha$ — оставшаяся строка), тогда $\alpha$ тоже будет в суффиксном дереве, потому что $\alpha$ является суффиксом $x\alpha$. Если для строки $x\alpha$ существует некоторая вершина $u$, то существует и вершина $u$ для $\alpha$. Ссылка из $u$ в $v$ называется суффиксной ссылкой.

Суффиксные ссылки позволяют не проходить каждый раз по дереву из корня. Для построения суффиксных ссылок достаточно хранить номер последней созданной вершины при продлении. Если на этой же фазе мы создаём ещё одну новую вершину, то нужно построить суффиксную ссылку из предыдущей в текущую.

Сложность алгоритма с использованием продления суффиксов и суффиксных ссылок $O(n^2)$. Подробное доказательство изложено в [1].

Для ускорения до $O(n)$ нужно уменьшить объём потребляемой памяти. Будем в каждом ребре дерева хранить не подстроку, а только индекс начала и конца подстроки.

У исходной строки $n$ суффиксов и будет создано не более $n$ внутренних вершин, в среднем продление суффиксов работает за $O(1)$.

При использовании всех вышеописанных эвристик получим временную и пространственную сложность $O(n)$.

Для нахождения минимального лексикографического разреза строки $s$ построим суффиксное дерево от удвоенной строки и найдём лексикографически минимальный путь длины $|s|$ в дереве. Сложность $O(n)$.

## 2 Исходный код

Опишем класс дерева и вспомогательную структуру TEdge для хранения информации о ребре. Нам понадобится конструктор от строки и методы нахождения минимальной лексикографической подстроки заданной длины.

```
#ifndef SUFFIX_TREE_HPP
#define SUFFIX_TREE_HPP

#include <memory>
#include <string>
#include <vector>

class TSuffixTree {
private:
    struct TEdge {
        int Left;
        std::shared_ptr<int> Right;
        int IdTo;

        TEdge(int l, std::shared_ptr<int> r, int id) : Left(l), Right(r), IdTo(id) {}
    };

    std::string DataString;
    std::vector< std::vector< TSuffixTree::TEdge > > Data;
    std::vector<int> SuffixPtr;
    std::vector<int> PathSize;
public:
    TSuffixTree(const std::string & s);
    std::string LexMinString(const size_t n);
    std::string LexMinString(const int id, const size_t n);
};

#endif /* SUFFIX_TREE_HPP */
```

При такой реализации решении занимает несколько строк.

```
#include "suffix_tree.hpp"
#include <iostream>

const char SENTINEL = 'z' + 1;

int main() {
    std::string s;
    std::cin >> s;
    std::string t = s + s + SENTINEL;
    TSuffixTree st(t);
    std::cout << st.LexMinString(s.size()) << '\n';
    return 0;
}
```

Самое сложное — реализация конструктора дерева и поиска лексикографически минимального пути в нём. Суффиксное дерево имеет максимум $2 * n$ вершин и $2 * n - 1$ рёбер, поэтому достаточно выгодно хранить дерево списками смежности.

```
#include "suffix_tree.hpp"

TSuffixTree::TSuffixTree(const std::string & s) : DataString(s), Data(s.size() * 2),
    SuffixPtr(s.size() * 2), PathSize(s.size() * 2) {
    bool newVertex = false;
    int l = 0;
    std::shared_ptr<int> end(new int);
    *end = 0;
    int passEq = 0;
    int curEq = 0;
    int curVertexId = 0;
    int curEdgeId = -1;
    int createdVertexId = 0;
    for (size_t i = 0; i < s.size(); ++i) {
        ++*end;
        int lastCreatedVertexId = 0;
        int nextCreatedVertexId = -1;
        while (l < *end) {
            if (curEdgeId == -1) {
                int nextEdgeId = -1;
                for (size_t j = 0; j < Data[curVertexId].size(); ++j) {
                    if (DataString[Data[curVertexId][j].Left] == DataString[l + passEq +
                         curEq]) {
                        nextEdgeId = j;
                        curEq = 1;
                        break;
                    }
                }
                if (nextEdgeId == -1) {
                    newVertex = true;
                }
                curEdgeId = nextEdgeId;
            } else {
                if (DataString[Data[curVertexId][curEdgeId].Left + curEq] == DataString[
                    l + curEq + passEq]) {
                    ++curEq;
                } else {
                    newVertex = true;
                }
            }
            if (curEq > 0 and Data[curVertexId][curEdgeId].Left + curEq == *Data[
                curVertexId][curEdgeId].Right) {
                curVertexId = Data[curVertexId][curEdgeId].IdTo;
                curEdgeId = -1;
                passEq = passEq + curEq;
                curEq = 0;
```

```
43              }
44              if (newVertex) {
45                  ++createdVertexId;
46                  if (curEq == 0 and curEdgeId == -1) {
47                      Data[curVertexId].push_back(TSuffixTree::TEdge(l + passEq, end,
                            createdVertexId));
48                  } else {
49                      nextCreatedVertexId = createdVertexId;
50                      TEdge curEdge = Data[curVertexId][curEdgeId];
51
52                      std::shared_ptr<int> newRightBorder(new int);
53                      *newRightBorder = Data[curVertexId][curEdgeId].Left + curEq;
54                      Data[curVertexId][curEdgeId].Right = newRightBorder;
55                      Data[curVertexId][curEdgeId].IdTo = createdVertexId;
56                      curEdge.Left = *newRightBorder;
57
58                      Data[createdVertexId].push_back(curEdge);
59                      ++createdVertexId;
60                      Data[createdVertexId - 1].push_back(TSuffixTree::TEdge(l + curEq +
                            passEq, end, createdVertexId));
61                      PathSize[createdVertexId - 1] = PathSize[curVertexId] + *Data[
                            curVertexId][curEdgeId].Right - Data[curVertexId][curEdgeId].
                            Left;
62
63                      if (lastCreatedVertexId > 0) {
64                          SuffixPtr[lastCreatedVertexId] = nextCreatedVertexId;
65                      }
66                      lastCreatedVertexId = nextCreatedVertexId;
67                  }
68                  int nextVertexId = SuffixPtr[curVertexId];
69                  int nextEdgeId = -1;
70                  int nextPassEq = PathSize[nextVertexId];
71                  int nextCurEq = curEq + passEq - nextPassEq - 1;
72                  while (nextCurEq > 0) {
73                      for (size_t j = 0; j < Data[nextVertexId].size(); ++j) {
74                          if (DataString[Data[nextVertexId][j].Left] == DataString[l + 1 +
                                nextPassEq]) {
75                              nextEdgeId = j;
76                              break;
77                          }
78                      }
79                      int curRight = *Data[nextVertexId][nextEdgeId].Right;
80                      int curLeft = Data[nextVertexId][nextEdgeId].Left;
81                      if (nextEdgeId != -1 and curRight - curLeft <= nextCurEq) {
82                          nextPassEq = nextPassEq + curRight - curLeft;
83                          nextCurEq = nextCurEq - curRight + curLeft;
84                          nextVertexId = Data[nextVertexId][nextEdgeId].IdTo;
85                          nextEdgeId = -1;
86                      } else {
```

```
87                      break;
88                  }
89              }
90              if (nextEdgeId != -1) {
91                  curEq = nextCurEq;
92              } else {
93                  curEq = 0;
94              }
95              passEq = nextPassEq;
96              curVertexId = nextVertexId;
97              curEdgeId = nextEdgeId;
98              ++l;
99              newVertex = false;
100         } else {
101             if (i < s.size() - 1) {
102                 break;
103             }
104         }
105     }
106 }
107 }
108
109 std::string TSuffixTree::LexMinString(const size_t n) {
110     return LexMinString(0, n);
111 }
112
113 std::string TSuffixTree::LexMinString(const int id, const size_t n) {
114     char lexMinId = 0;
115     for (size_t i = 0; i < Data[id].size(); ++i) {
116         if (DataString[Data[id][i].Left] < DataString[Data[id][lexMinId].Left]) {
117             lexMinId = i;
118         }
119     }
120     size_t curEdgeLen = *Data[id][lexMinId].Right - Data[id][lexMinId].Left;
121     if (n <= curEdgeLen) {
122         return DataString.substr(Data[id][lexMinId].Left, n);
123     } else {
124         std::string res = DataString.substr(Data[id][lexMinId].Left, curEdgeLen);
125         return res + LexMinString(Data[id][lexMinId].IdTo, n - curEdgeLen);
126     }
127 }
```

# 3 Консоль

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ make
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror -c suffix_tree.cpp -o
suffix_tree.o
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror -c main.cpp -o main.o
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror suffix_tree.o main.o
-o solution
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ cat tests/1.in
xabcd
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ ./solution <tests/1.in
abcdx
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ cat tests/2.in
abracadabra
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ ./solution <tests/2.in
aabracadabr
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ cat tests/3.in
abaacaababaab
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ ./solution <tests/3.in
aababaababaac
```

# 4 Тест производительности

Сравним наиный алгоритм, который сортирует все циклические разрезы строки, с алгоритмом, использующим суффиксное дерево.

Тесты состоят из строк длины 100, 1000, 10000 и 50000.

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab5$ make bench

./solution <tests/1.in
Suffix tree time 0.178 ms
./benchmark <tests/1.in
Naive time 0.116 ms

./solution <tests/2.in
Suffix tree time 1.680 ms
./benchmark <tests/2.in
Naive time 3.180 ms

./solution <tests/3.in
Suffix tree time 17.795 ms
./benchmark <tests/3.in
Naive time 81.510 ms

./solution <tests/4.in
Suffix tree time 18.455 ms
./benchmark <tests/4.in
Naive time 1334.365 ms
```

Видно, что на маленькой строке константа в сложности алгоритма Укконена даёт о себе знать, а на больших строках квадратичная сложность не может соревноваться с линейной.

# 5 Выводы

Во время выполнения лабораторной работы я вспомнил префиксное дерево, способы хранения графов, реализовал алгоритм Укконена и ознакомился с приложениями суффиксного дерева.

Сложнее всего было понять, как алгоритм достигает линейной сложности и как работают суффиксные ссылки. Помогли рисунки в тетради и визуализатор [2].

Суффиксное дерево позволяет быстро искать множество шаблонов в тексте, чего не могут другие алгоритмы. Но в повседневных задачах чаще требуется найти один шаблон в тексте, где лучше использовать более простые алгоритмы: алгоритм Кнута-Морриса-Пратта, Бойера-Мура, Рабина-Карпа.

# Список литературы

[1] *Алгоритм Укконена — Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 19.03.2021).

[2] *Visualization of Ukkonen's Algorithm*
URL: `http://brenden.github.io/ukkonen-animation/` (дата обращения: 18.03.2021).