

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. Т. Бахарев  
Преподаватель: А. А. Кухтичев  
Группа: М8О-206Б  
Дата:  
Оценка:  
Подпись:

Москва, 2018

## Лабораторная работа №5

**Задача:** Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из входных строк, необходимо воспользоваться полученным суффиксным деревом для решения своего варианта задания.

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

**Вариант задания : 2** Поиск с использованием суффиксного массива. Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

**Входные данные:** текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

**Выходные данные:** для каждого образца, найденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

# 1 Описание

## 2 Исходный код

suffix\_tree.h

```
1 #define MAX_LENGTH std::numeric_limits<std::size_t>::max()
2 class TNode
3 {
4 public:
5     using link = std::map<char, TNode *>;
6     link edges;
7     TNode *parent;
8     TNode *suffixLink;
9     std::size_t begin;
10    std::size_t length;
11    TNode(const std::string &, TNode * const &);
12    TNode(const std::vector<char> &, const char &);
13    TNode(const std::vector<char> &, const char &, TNode * const &);
14    TNode(const std::string &, TNode * const &, const std::size_t &);
15    virtual ~TNode();
16 };
17 class TSuffixTree;
18 class TSuffixArray;
19 class TSuffixArray
20 {
21 private:
22     std::string text;
23     std::vector<std::size_t> array;
24 public:
25     explicit TSuffixArray(const TSuffixTree &);
26     std::vector<std::size_t> Find(const std::string &pattern) const;
27     virtual ~TSuffixArray();
28 };
29 class TSuffixTree
30 {
31     std::string text;
32     TNode *root;
33     TNode *active_vertex;
34     std::size_t activeLength;
35     std::size_t activeCharIdx;
36     void DFS(TNode * const &, std::vector<std::size_t> &, const std::size_t &) const;
37     friend TSuffixArray::TSuffixArray(const TSuffixTree &);
38 public:
39     explicit TSuffixTree(const std::vector<char> &, const char &);
40     void PushBack(const char &ch);
41     virtual ~TSuffixTree();
42 };
```

suffix_tree.cpp	
void TSuffixTree::PushBack(const char& new_ch)	Функция добавления символа в суффиксное дерево в режиме реального времени
void TSuffixTree::DFS(TNode const &curr, std::vector<std::size_t> &result, const std::size_t &summary)	Преобразование суффиксного дерева в суффиксный массив при помощи алгоритма обхода в глубину.
std::vector<std::size_t> TSuffixArray::Find(const std::string &pattern)	Поиск подстроки в суффиксном массиве
file1.c	
void function_name()	Функция, «которая почти всегда работает, но неясно, что она делает».

В этом случае структуры или классы должны быть полностью приведены в листинге (без реализации методов).

```

1 | struct KV{
2 |     int key;
3 |     char value;
4 | } KV;
```

### 3 Консоль

```
alex$make
g++ suffix_tree.cpp main.cpp -std=c++11 -pedantic -Wall -Werror -Wno-sign-compare
-Wno-long-long -lm -o diskran_laba_5
alex$ cat 01.t
bbcabcacdcabbdbabacbacaddaabcaaddacbccbdaacbdcaaddbabbbdadabcbbbdadcaadacbaaada
abbacbddddccbcabccdadabaccdacaddbbccd
bbac
bbca
aabc
dcab
dadd
dbab
dbac
bbba
acbd
bbba
ccab
bdad
dadd
bbca
alex$ ./*5 <01.t
1: 17
3: 49
4: 95
5: 82
6: 1
7: 27
8: 10
10: 15,51
13: 43,84
16: 65
18: 1
alex$
```

## 4 Тест производительности

Поиск подстроки в строке при помощи суффиксного массива я сравнивал с наивным алгоритмом. Ниже приведен листинг бенчмарка и генератора тестов на python: benchmark.cpp

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 std::vector<int> Find(const std::string& pattern, const std::string& text)
6 {
7     std::vector<int> res;
8     for(int i = 0; i < text.size(); ++i)
9     {
10         int k = i;
11         for(int j = 0; j < pattern.size(); ++j)
12         {
13             if(text[k] == pattern[j])
14                 ++k;
15             else
16                 break;
17         }
18         if(i - k == pattern.size() - 1)
19             res.push_back(i);
20     }
21     return res;
22 }
23 int main()
24 {
25     std::string text;
26     std::string pattern;
27     std::cin>>text;
28     size_t n = 1;
29     std::string buf;
30     while(std::cin>>pattern)
31     {
32         std::vector<int> res = Find(pattern, text);
33         if(!res.empty())
34         {
35             std::cout<<n<<" ": ";
36             for(auto i : res)
37                 std::cout<<res[i]<<" ";
38         }
39         ++n;
40     }
41     return 0;
42 }
```

## Generator.py

```
1 import random
2 import sys
3 def GenText(l, alphabet):
4     text = str()
5     for i in range(l):
6         text += random.choice(alphabet)
7     return text
8
9 inputFile = open("01.t", "w")
10 outputFile = open("01.a", "w")
11 textLen = 1000000
12 alphabet = ('a', 'b', 'c', 'd')
13 text = str(GenText(textLen, alphabet)) # Generate text
14 inputFile.write(text + '\n')
15 pCount = random.randint(1000, 2000) # Generate count of patterns
16 for i in range(1, pCount):
17     pattern = str()
18     for j in range(random.randint(20, 50)):
19         pattern += random.choice(alphabet)
20     inputFile.write(pattern + "\n")
21     buf = str()
22     res = text.find(pattern, 0, len(text))
23     isMatch = False
24     if res != -1:
25         buf += str(i) + ": " #write num of pattern if we have at least one match
26     while res != -1:
27         if isMatch:
28             buf += ", "
29             isMatch = True
30             buf += str(res)
31             res = text.find(pattern, res + 1, len(text))
32     if len(buf) != 0:
33         outputFile.write(buf + "\n")
```



```

alex$python3 Gen*
alex$wc 01.t
1120      1120 1004472 01.t
alex
alex$make
g++ suffix_tree.cpp main.cpp -std=c++11 -pedantic -Wall -Werror -Wno-sign-compare
-Wno-long-long -lm -o diskran_laba_5
alex$g++ -std=c++11 bench*
alex$time ./a.out <01.t >tmp

real      0m30.215s
user      0m30.204s
sys       0m0.000s
alex$time ./a.out <01.t >tmp
real      0m2.946s
user      0m2.556s
sys       0m0.308s

```

Как видно из теста производительности, суффиксное дерево быстрее чем наивный алгоритм. Разница колоссальная. Текст состоит из большого количества символов, шаблоны много меньше по длине. Но если длина паттерна будет, к примеру, 4-5 символов, то разницы во времени работы наших алгоритмов не будет. Это связано с тем, что наивный алгоритм поиска хорошо показывает себя, когда нам надо найти в большом тексте маленький фрагмент. Время работы будет линейно зависеть от длины исходного текста(длину паттерна в этом случае можно принять за константу). Однако при работе с большими образцами, время работы этого алгоритма возрастет до квадрата от длины текста(в худшем случае), в то время как суффиксное дерево будет работать линейно от длины шаблона! Так что, по моему мнению, нет абсолютно универсального и удобного метода поиска подстроки в строке. Нужно смотреть по ситуации, что в данном случае проще, удобнее и самое главное быстрее.

## 5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я познакомился с новым алгоритмом поиска подстрок в тексте – суффиксное дерево. Он крайне эффективен, если нам заранее известен текст. Мы подготавливаем структуру дерева для эффективного поиска любого количества образцов. Время построения суффиксного дерева – линейное. Время работы при поиске подстрок – тоже линейное, зависящее от их длин. Существует несколько реализаций построений суффиксных деревьев. Однако самым простым и эффективным является алгоритм Укконена. Его большой плюс в том, что нам не нужно хранить строку с текстом целиком. Мы считываем символ за символом и строим наше дерево. Это так называемый алгоритм реального времени(online algorithm). Его можно использовать в большом спектре задач, где нужна интерактивность и минимальное время отклика.

## Список литературы

- [1] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*
- [2] *Суффиксное дерево – Википедия.*  
URL: [https://ru.wikipedia.org/wiki/Суффиксное\\_дерево](https://ru.wikipedia.org/wiki/Суффиксное_дерево) (дата обращения: 01.10.2018).
- [3] *Суффиксное дерево. Алгоритм Укконена*  
URL: <http://e-maxx.ru/algo/ukkonen> (дата обращения: 01.10.2018).