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: И. О. Ильин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №5

Вариант №2

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 в заранее известном тексте поступающие на вход образцы с использование суффиксного массива.

Формат входных данных

Текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

Формат результата

Для каждого образца, найденного в тексте, нужно распечатать строку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

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 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 $x\alpha$ в вершину α, где x – первым символ строки, α – оставшаяся подстрока (возможно пустая). Использование суффиксных ссылок позволяет пропускать ненужные сравнения. [1].

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 $O(log(t) \cdot (p_1 + p_2 + ... + p_m))$ ищем m образцов. Итоговая сложность всех операций $O(t + log(t) \cdot (p_1 + p_2 + ... + p_m))$.

2 Исходный код

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

```
1   #pragma once
2
3  #include "suffix_array.h"
4  #include <string>
5  #include <vector>
6  #include <map>
7
8  namespace NSuffixTrees {
9
10     class TSuffixArray;
11
12     class TNode {
13     public:
14         TNode(std::string::iterator begin, std::string::iterator end);
15         ~TNode() = default;
16
17         std::string::iterator begin;
18         std::string::iterator end;
19
20         std::map< char, TNode* > to;
21         TNode* suffixLink;
22     };
23
24     class TSuffixTree {
25     public:
26         TSuffixTree(std::string str);
27         ~TSuffixTree();
28         friend TSuffixArray;
29
30     private:
31         std::string text;
32         TNode* root;
33
34         TNode* needSufLink;
35         TNode* activeNode;
36         int remainder;
37         int activeLen;
38         std::string::iterator activeEdge;
39
40         void TreeExtend(std::string::iterator toAdd);
41         void DeleteTree(TNode* node);
42
43         int GetEdgeLen(TNode* node, std::string::iterator pos) const;
```

```
44          void AddSuffixLink(TNode* node);
45          void DFS(TNode* node, std::vector<size_t>& result, size_t depth) const;
46      };
47  } // namespace NSuffixTrees
```

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

```
1   #include "suffix_tree.h"
2
3   namespace NSuffixTrees {
4       //// TNode
5       TNode::TNode(std::string::iterator begin, std::string::iterator end) :
6           begin(begin),
7           end(end),
8           suffixLink(0)
9       {}
10
11      //// TSuffixTree
12      TSuffixTree::TSuffixTree(std::string str) :
13          text(str),
14          root(new TNode(text.end(), text.end())),
15          remainder(0)
16      {
17          activeEdge = text.begin();
18          activeNode = needSufLink = root->suffixLink = root;
19          activeLen = 0;
20
21          for (std::string::iterator it = text.begin(); it != text.end(); ++it) {
22              TreeExtend(it);
23          }
24      }
25
26      TSuffixTree::~TSuffixTree() {
27          DeleteTree(root);
28      };
29
30      void TSuffixTree::DeleteTree(TNode* node) {
31          for (std::map< char, TNode* >::iterator it = node->to.begin(); it != node->to.
                end(); ++it) {
32              DeleteTree(it->second);
33          }
34          delete node;
35      }
36
37      int TSuffixTree::GetEdgeLen(TNode* node, std::string::iterator pos) const {
38          return std::min(node->end, pos + 1) - node->begin;
39      }
```

```
40
41      void TSuffixTree::TreeExtend(std::string::iterator toAdd) {
42          needSufLink = root;
43          ++remainder;
44
45          while (remainder) {
46              if (!activeLen) {
47                  activeEdge = toAdd;
48              }
49
50              TNode *next = NULL;
51              std::map< char, TNode* >::iterator it = activeNode->to.find(*activeEdge);
52              if (it != activeNode->to.end()) {
53                  next = it->second;
54              }
55
56              if (!next) {
57                  TNode* leaf = new TNode(toAdd, text.end());
58                  activeNode->to[*activeEdge] = leaf;
59                  AddSuffixLink(activeNode);
60              } else {
61                  if (activeLen >= GetEdgeLen(next, toAdd)) {
62                      activeEdge += GetEdgeLen(next, toAdd);
63                      activeLen -= GetEdgeLen(next, toAdd);
64                      activeNode = next;
65                      continue;
66                  }
67
68                  if (*(next->begin + activeLen) == *toAdd) {
69                      ++activeLen;
70                      AddSuffixLink(activeNode);
71                      break;
72                  }
73
74                  TNode* split = new TNode(next->begin, next->begin + activeLen);
75                  TNode* leaf = new TNode(toAdd, text.end());
76                  activeNode->to[*activeEdge] = split;
77
78                  split->to[*toAdd] = leaf;
79                  next->begin += activeLen;
80                  split->to[*next->begin] = next;
81                  AddSuffixLink(split);
82              }
83
84              --remainder;
85              if (activeNode == root && activeLen > 0) {
86                  --activeLen;
87                  activeEdge = toAdd - remainder + 1;
88              } else {
```

```
89                 if (activeNode->suffixLink) {
90                     activeNode = activeNode->suffixLink;
91                 } else {
92                     activeNode = root;
93                 }
94             }
95         }
96     }
97
98     void TSuffixTree::AddSuffixLink(TNode* node) {
99         if (needSufLink != root) {
100            needSufLink->suffixLink = node;
101        }
102        needSufLink = node;
103    }
104
105    void TSuffixTree::DFS(TNode* node, std::vector<size_t>& result, size_t depth) const
           {
106        if (node->to.empty()) {
107            result.push_back(text.size() - depth);
108            return;
109        }
110        for (std::map<char, TNode*>::iterator it = node->to.begin(); it != node->to.end
               (); ++it) {
111            DFS(it->second, result, depth + it->second->end - it->second->begin);
112        }
113    }
114 } // namespace NSuffixTrees
```

В *suffix_ array.h* опишем класс суффиксного массива с конструктором от суффиксного дерева.

```
1  #pragma once
2
3  #include "suffix_tree.h"
4  #include <vector>
5  #include <string>
6
7  namespace NSuffixTrees {
8      class TSuffixTree;
9
10     class TSuffixArray {
11     public:
12         TSuffixArray(const TSuffixTree& tree);
13         ~TSuffixArray() = default;
14
15         std::vector<size_t> Find(const std::string& pattern);
16     private:
17         std::string text;
18         std::vector<size_t> array;
```

```
    };
}
```

В *suffix_ array.cpp* напишем соответствующие реализации.

```
#include "suffix_array.h"
#include <algorithm>

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;
    }
} // namespace NSuffixTrees
```

3 Консоль

```
MacBook-Pro:da_lab_05 mr-ilin$ ./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
MacBook-Pro:da_lab_05 mr-ilin$ cat tests/t02.t
wowCanYouFindMeHereOrHereAndHereCanYou?
Here
You
Can
MacBook-Pro:da_lab_05 mr-ilin$ ./solution <tests/t02.t
1: 16,22,29
2: 7,36
3: 4,33
```

4 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 представляло трудности. В процессе отладки программы на тестах вскрылись многие баги и ошибки, путём исправления которых получилось доработать изначальный алгоритм.

Список литературы

[1] *Алгоритм Укконена — ИТМО Вики.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 01.06.2021).