МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по практической работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 7383 _____ Зуев Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Цель работы: ознакомиться с алгоритмом Ахо-Корасик на примере построения алгоритма для выполнения задачи.

Формулировка задачи:

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст T ($1 \le |T| \le 100000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p_1,\ldots,p_n\}$, $1 \le |p_i| \le 75$.

Все строки содержат символы из алфавита $\{A, C, G, T, N\}$.

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p.

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемого джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Вход:

Текст (T, $1 \le |T| \le 100000$)

Шаблон (P, $1 \le |P| \le 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Реализация задачи.

Для реализации задачи была написана структура node для хранения узлов дерева называемого бором.

Ниже представлены поля структуры node:

descendants[SIZE] — массив детей.

transition[SIZE] — массив переходов из вершины по символу (в отличие от предыдущего массива вычисляется для любого символа).

parent — указатель на родителя.

suff_link — суффикс ссылка из вершины.

compressed_suff_link — сжатая суффикс ссылка (только на терминальные вершины).

is_pat — флаг того, является ли вершина терминальной.

pat_num — номер шаблона.

pat_length — длина шаблона.

symb — символ, по которому был совершен переход из родителя в текущую вершину.

Для реализации алгоритма были написаны следующие функции:

add_string_to_trie — добавляет строку в бор. Проходит по всем символам строки. Если у текущей вершины нет сына чтобы перейти в него по текущему символу, создает вершину. Для второго задания добавляются переходы по всем символам из алфавита в массив transition при встрече джокера, причем эти переходы будут в вершину-сына, в которую переход по джокеру.

suff_func — рекурсивно вычисляет суффикс ссылку для вершины и возвращает её.

transition_func — вычисляет переход из вершины по символу с помощью суффикс ссылки из этой вершины и возвращает переход.

compressed_func — вычисляет сжатую суффикс ссылку для вершины переходя по обычным суффикс ссылкам и, если встречает терминальную или корень, то возвращает указатель на эту вершину.

search — проходит по строке параллельно идя по бору. С помощью функции построения переходов переходит на следующую вершину в боре. Для каждой вершины переходит по сжатым суффикс ссылкам. Если вершина терминальная записывает в результат пару — позиция в строке и номер шаблона для первого задания и только позиция для второго задания.

В главной функции main считывается строка, количество шаблонов и шаблоны для первого задания, строка шаблон и джокер для второго задания. Шаблоны записываются в бор и вызывается функция поиска шаблонов в строке. Так как результат хранится в контейнере map то сортировка происходит внутри контейнера. Выводит результат.

Код программы представлен в приложении Б.

Исследование сложности алгоритма.

Функция add_string_to_trie добавляет шаблон за время O(|i|), где i — это длина шаблона. Значит суммарно выполнение функции add_string_to_trie для всех шаблонов займет время O(|n|), где n — суммарная длина всех шаблонов. Функция search проходит по всем символам строки параллельно вычисляя суффикс ссылки, сжатые суффикс ссылки и переходы для вершин. Так как суффикс ссылки, сжатые суффикс ссылки и переходы вычисляются на каждом шаге и количество вершин бора не превосходит суммарного количества символов в шаблонах, то сложность увеличивается на O(|n|+|m|+|k|), где m — это длина строки, k — это количество вхождений шаблона в строку, так как префикс функция может максимум рекурсивно вызваться k раз.

В сумме получается сложность O(|n|+|m|+|k|), где n — суммарная длина всех шаблонов, m — это длина строки, k — это количество вхождений шаблона в строку.

Тестирование.

Программа была собрана в компиляторе G++ в среде разработки Qt в операционной системе Linux Ubuntu 17.10.

В ходе тестирования ошибок выявлено не было.

Корректные тестовые случаи представлены в приложении А.

Выводы.

В ходе выполнения данной работы был изучен и реализован алгоритм Ахо-Корасик поиска всех вхождений строк из множества шаблонов в другой. Оценена сложность реализованного алгоритма, она составляет O(|n|+|m|+|k|).

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Выходные данные
CCCT 1 C	1 1 2 1 3 1
AAAAAAAAA 3 A AA AAAAA	1 1 1 2 1 3 2 1 2 2 2 3 3 1 3 2 3 3 4 1 4 2 4 3 5 1 5 2 5 3 6 1 6 2 7 1 7 2 8 1 8 2 9 1
ACCTNT 1 GT	

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <map>


#define SIZE 5

using namespace std;


int num_of_symb(char symb)
{
   switch(symb)
   {
      case 'A': return 0;
      case 'C': return 1;
      case 'G': return 2;
      case 'T': return 3;
      case 'N': return 4;
      default: return -1;
   }
}


struct node
{
   node* descendants[SIZE];
   node* transition[SIZE];
   node* parent;
   node* suff_link;
   node* compressed_suff_link;
   bool is_pat;
   int pat_num;
   int pat_length;
   char symb;

node():parent(nullptr),suff_link(nullptr),compressed_suff_link(nullptr),is_pat(0),p
at_num(-1)
   {
      for(int i = 0; i < SIZE; i++)
      {
         descendants[i] = nullptr;
         transition[i] = nullptr;
      }
   }
};

node* transition_func(node*,char);

node* suff_func(node* current)
{
```

```
    if(current->suff_link)
        return current->suff_link;
    if(current->parent == nullptr)
         current->suff_link  = current;
    else if(current->parent->parent == nullptr)
         current->suff_link = current->parent;
    else
        current->suff_link = transition_func(suff_func(current->parent), current-
>symb);
    return current->suff_link;
}

node* transition_func(node* current, char symb)
{
    if(current->transition[num_of_symb(symb)])
        return current->transition[num_of_symb(symb)];
    if(current->descendants[num_of_symb(symb)])
        current->transition[num_of_symb(symb)] = current-
>descendants[num_of_symb(symb)];
    else if(current->parent == nullptr)
        current->transition[num_of_symb(symb)] = current;
    else
        current->transition[num_of_symb(symb)] =
transition_func(suff_func(current), symb);
    return current->transition[num_of_symb(symb)];
}

node* compressed_func(node* current)
{
    if(current->compressed_suff_link)
        return current->compressed_suff_link;
    if(suff_func(current)->is_pat || suff_func(current)->parent == nullptr)
        current->compressed_suff_link = suff_func(current);
    else
        current->compressed_suff_link = compressed_func(suff_func(current));
    return current->compressed_suff_link;
}

void add_string_to_trie(const string& s, node* root, int num)
{
    node* current = root;
    for(int i = 0; i<s.length(); i++)
    {
        if(current->descendants[num_of_symb(s[i])] == nullptr)
        {
            current->descendants[num_of_symb(s[i])] = new node;
            current->descendants[num_of_symb(s[i])]->parent = current;
        }
        current = current->descendants[num_of_symb(s[i])];
        current->symb = s[i];
    }
    current->is_pat = 1;
```

```
    current->pat_num = num;
    current->pat_length = s.length();
}

void search(string s, node* root, map<int,map<int,bool>>&  result)
{
    node* current = root;
    node* compressed_suff;
    for(int i = 0; i<s.length(); i++)
    {
        current = transition_func(current, s[i]);
        if(current->is_pat)
            result[i - current->pat_length+2][current->pat_num+1] = 1;
        compressed_suff = current;
        while(compressed_suff != root)
        {
            compressed_suff = compressed_func(compressed_suff);
            if(compressed_suff->is_pat)
                result[i - compressed_suff->pat_length+2][compressed_suff-
>pat_num+1] = 1;
        }
    }
}

int main()
{
    string T;
    int n;
    cin>>T>>n;
    string P[3000];
    node* root;
    root = new node;
    root->symb = '#';
    root->suff_link = root;
    for(int i = 0; i<n;i++)
    {
        cin>>P[i];
        add_string_to_trie(P[i], root, i);
    }
    map<int, map<int, bool>> result;
    search(T, root, result);
    for(map<int, map<int,bool>>::iterator i = result.begin(); i != result.end();
i++)
        for(map<int,bool>::iterator j = i->second.begin(); j != i->second.end(); j+
+)
            cout<< i->first<< ' ' << j->first << endl;
    return 0;
}
```