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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студентка гр. 7383 _____ Иолшина В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Содержание

Цель работы

Исследовать и реализовывать задачу поиска набора образцов в строке, используя алгоритм Ахо-Корасик. Формулировка задачи: необходимо разработать программу, решающую задачу точного поиска набора образцов. Входные данные: Первая строка содержит текст (T,1≤|T|≤100000). Вторая – число n(1≤n≤3000), каждая следующая из n строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75. Все строки содержат символы из алфавита {A,C,G,T,N}. Выходные данные: 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. Также следует разработать программу для решения следующей задачи: реализовать точный поиск для одного образца с джокером. Входные данные: в первой строке указывается текст (T,1≤|T|≤100000), в котором ищем подстроки, а во второй шаблон (P,1≤|P|≤40), затем идёт символ джокера. Выходные данные: строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Реализация задачи

В данной работе используются главная функция main() и класс Aho_Korasik, а также структура element, которая используется для реализации бора. На ребрах между вершинами написана одна буква, таким образом, добираясь по ребрам из корня в какую-нибудь вершину и контангенируя буквы из ребер в порядке обхода, получаем строку, соответствующую этой вершине.

struct element

{ bool check;

int sample;

int suffix_link;

int previous_vertex;

char curr_vertex;

map edge;

map auto_move;

leaf(int previous, char vertex) { check = false; sample = 0;

if(previous == 0) suffix_link = 0;

else suffix_link = -1; previous_vertex = previous;

curr_vertex = vertex; }

};

Поля структуры element:

- check – флаг, показывающий является ли вершина конечной;
- sample – номер образца;
- suffix_link – суффиксная ссылка v, указатель на вершину u, такую что строка u – наибольший собственный суффикс строки v, или, если такой вершины нет в боре, то указатель на корень;
- previous_vertex – индекс родителя, индекс вершины, из которой пришли в текущую;
- curr_vertex – значение ребра от родителя к текущей вершине;
- edge – упорядоченный ассоциативный массив типа map, задающий соседей текущей вершины;
- auto_move – переходы автомата;
- leaf(int previous, char vertex) – конструктор, где previous – предок вершины, vertex – значение ребра.

class Aho_Korasik {

vector tree;

vector result;

int number_of_tree_nodes;

public: Aho_Korasik();

void add_pattern(string &P, int count_str);

int get_suffix_link(int index_vertex);

int get_auto_move(int index_vertex, char symbol);

void algorithm(std:: string T);

void print(); };

Параметры, хранящиеся в класса:

- tree – бор, представляемый в виде вектора структур leaf;
- result – набор строк-шаблонов, хранящихся в векторе;
- number_of_tree_nodes – количество узлов дерева, используется как индекс для заполнения бора.

Методы класса:

- Aho_Korasik – конструктор по умолчанию, используется для создания корня дерева;
- add_pattern – добавление нового образца в набор шаблонов;
- get_suffix_link – получение суффиксальной ссылки для заданной вершины;
- get_auto_move – выполнение автоматного перехода;
- algorithm – алгоритм Ахо-Корасик, выводит все вхождения образцов из P в T;
- print – функция печати информации о вершинах бора. Параметры, передаваемые в void add_pattern(string &P, int count_str):
- P – образец из набора шаблонов, которую нужно добавить;
- count_str – порядковый номер образца, начиная с единицы. Параметры, передаваемые в int get_suffix_link(int index_vertex):
- index_vertex – индекс вершины в боре, для которой нужно найти суффиксальную ссылку. Параметры, передаваемые в int get_auto_move(int

index_vertex, char symbol):

- index_vertex – индекс вершины в боре;
- symbol – символ перехода, значение ребра. Параметры, передаваемые в void algorithm(string T):
- T – текст, в котором нужно найти подстроки.

Тестирование

Программа собрана в операционной системе Ubuntu 16.04.2 LTS", с использованием компилятора g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5). В других ОС и компиляторах тестирование не проводилось.

Тестовые случаи представлены в Приложении А.

Исследование

Сложность реализованного алгоритма Ахо-Корасик с использованием структуры данных map – O((|T| + |P|) * log|A| + c), где T – длина текста, в котором осуществляется поиск, |P| – общая длина всех слов в словаре, |A| – размер алфавита и c – общая длина всех совпадений. По памяти сложность алгоритма составляет O(|T| + |P|), так как память выделяется для всего текста и для вершин шаблонов.

Выводы

В ходе написания лабораторной работы был изучен метод поиска подстрок в строке, используя алгоритм Ахо-Корасик. Был написан код на языке программирования C++, который применял этот способ для поставленной задачи. Сложность по количеству просмотренных вершин равна O((|T| + |P|) * log|A| + c), а по памяти – O(|T| + |P|).

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Ввод	Вывод	Верно?
abbaaaba 6 aba ba abba baab bab b	2 6 3 6 1 3 3 2 7 6 6 1 7 2	Да
ACT A$ $	1	Да
ACTATTCATC A$T $	1 4	Да

ПРИЛОЖЕНИЕ Б

Исходный код программы

Lr5_1.cpp

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;

struct element
{
   bool check;
   int sample;
   int suffix_link;
   int previous_vertex;
   char curr_vertex;
   map <char, int> edge;
   map <char, int> auto_move;

   element(int previous, char vertex)
   {
      check = false;
      sample = 0;
      if(previous == 0) suffix_link = 0;
      else suffix_link = -1;
      previous_vertex = previous;
      curr_vertex = vertex;
   }
};

class Aho_Korasik
{
   vector <element> tree;
   vector <string> result;
   int number_of_tree_nodes;
```

```
public:
   Aho_Korasik()
   {
      tree.push_back(element(0, 0));
      number_of_tree_nodes = 1;
   }

   void add_pattern(string &P, int count_str)
   {
      int temp = 0;
      result.push_back(P);
      for(int num = 0; num < P.length(); num++)
      {
         if(tree[temp].edge.find(P[num]) == tree[temp].edge.end())
         {
            tree.push_back(element(temp, P[num]));
            tree[temp].edge[P[num]] = number_of_tree_nodes++;
         }
         temp = tree[temp].edge[P[num]];
      }
      tree[temp].check = true;
      tree[temp].sample = count_str;
   }

   int get_suffix_link(int index_vertex)
   {
      if(tree[index_vertex].suffix_link == -1)
      {
         if(index_vertex == 0 || tree[index_vertex].previous_vertex == 0)
            tree[index_vertex].suffix_link = 0;
         else
            tree[index_vertex].suffix_link =
get_auto_move(get_suffix_link(tree[index_vertex].previous_vertex),
tree[index_vertex].curr_vertex);
      }
      return tree[index_vertex].suffix_link;
   }
```

```
    int get_auto_move(int index_vertex, char c)
    {
        if(tree[index_vertex].auto_move.find(c) == tree[index_vertex].auto_move.end())
            if(tree[index_vertex].edge.find(c) != tree[index_vertex].edge.end())
                tree[index_vertex].auto_move[c] = tree[index_vertex].edge[c];
            else if(index_vertex == 0)
                tree[index_vertex].auto_move[c] = 0;
            else
                tree[index_vertex].auto_move[c] = get_auto_move(get_suffix_link(index_vertex), c);
        return tree[index_vertex].auto_move[c];
    }

    void algorithm(string T)
    {
        int ver = 0;
        for(int i = 0; i < T.length(); i++)
        {
            ver = get_auto_move(ver, T[i]);
            for(int j = ver; j != 0; j = get_suffix_link(j))
                if(tree[j].check)
                    cout << i - result[tree[j].sample - 1].size() + 2 << " " << tree[j].sample << endl;
        }
    }

};

int main()
{
    Aho_Korasik object;
    string text, pattern;
    int count;
    cin >> text >> count;
    for(int i = 0; i < count; i++)
    {
        cin >> pattern;
        object.add_pattern(pattern, i + 1);
```

```
    }
    object.algorithm(text);
    return 0;
}
```

Lr5_2.cpp

```
#include <iostream>
#include <vector>
#include <map>
using namespace std;

struct element
{
    bool check;
    int sample;
    int suffix_link;
    int previous_vertex;
    char curr_vertex;
    map <char, int> edge;
    map <char, int> auto_move;

    element(int previous, char vertex)
    {
        check = false;
        sample = 0;
        suffix_link = -1;
        previous_vertex = previous;
        curr_vertex = vertex;
    }
};

class Aho_Korasik
{
    vector <element> tree;
    vector <string> result;
    int number_of_tree_nodes;
    char joker;
```

```
public:
   Aho_Korasik(char joker): joker(joker)
   {
      tree.push_back(element(0, 0));
      number_of_tree_nodes = 1;
   }

   void add_pattern(string &P)
   {
      int temp = 0;
      result.push_back(P);
      for(int num = 0; num < P.length(); num++)
       {
         if(tree[temp].edge.find(P[num]) == tree[temp].edge.end())
         {
            tree.push_back(element(temp, P[num]));
            tree[temp].edge[P[num]] = number_of_tree_nodes++;
         }
         temp = tree[temp].edge[P[num]];
      }
      tree[temp].check = true;
      tree[temp].sample = 1;
   }

   int get_suffix_link(int index_vertex)
   {
      if(tree[index_vertex].suffix_link == -1)
      {
         if(index_vertex == 0 || tree[index_vertex].previous_vertex == 0)
            tree[index_vertex].suffix_link = 0;
         else
            tree[index_vertex].suffix_link =
get_auto_move(get_suffix_link(tree[index_vertex].previous_vertex),
tree[index_vertex].curr_vertex);
      }
      return tree[index_vertex].suffix_link;
   }
```

```
    int get_auto_move(int index_vertex, char c)
    {
        if(tree[index_vertex].auto_move.find(c) == tree[index_vertex].auto_move.end())
            if(tree[index_vertex].edge.find(c) != tree[index_vertex].edge.end())
                tree[index_vertex].auto_move[c] = tree[index_vertex].edge[c];
            else if(tree[index_vertex].edge.find(joker) != tree[index_vertex].edge.end())
                tree[index_vertex].auto_move[c] = tree[index_vertex].edge[joker];
            else if(index_vertex == 0)
                tree[index_vertex].auto_move[c] = 0;
            else
                tree[index_vertex].auto_move[c] = get_auto_move(get_suffix_link(index_vertex), c);
        return tree[index_vertex].auto_move[c];
    }

    void algorithm(string T)
    {
        int ver = 0;
        for(int i = 0; i < T.length(); i++)
        {
            ver = get_auto_move(ver, T[i]);
            for(int j = ver; j != 0; j = get_suffix_link(j))
            {
                if(tree[j].check)
                    cout << i - result[tree[j].sample - 1].size() + 2 << endl;
            }
        }
    }
};

int main()
{
    string text, pattern;
    char joker;
    cin >> text >> pattern >> joker;
    Aho_Korasik object(joker);
    object.add_pattern(pattern);
```

```
    object.algorithm(text);
    return 0;
}
```