

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-Корасика

Студент гр. 7304

Овчинников Н.В.

Преподаватель

Филатов А.Ю.

Санкт-Петербург

2019

Цель работы

Изучить алгоритм Ахо-Корасик поиска множества подстрок в строке, а также реализовать данный алгоритм на языке программирования C++.

Задание

1. Разработать программу, решающую задачу точного поиска набора образцов.
2. Используя реализацию точного множественного поиска, решить задачу точного поиска для одного образца с джокером.

Ход работы

1. Был реализован алгоритм Ахо-Корасика для поиска множества подстрок в строке на языке программирования C++.
 - a. Для корректной работы алгоритма на первом шаге была реализована структура вершины бора. В структуре `next_vertex` – массив вершин, в которые можно попасть из данной вершины; `flag` – переменная типа `bool`, которая определяет, является ли вершина финальной для какого-либо шаблона; `auto_move` – массив переходов из одного состояния в другое; `suff_link` – переменная для хранения суффиксной ссылки; `par` – номер вершины-родителя; `symbol` – символ, по которому осуществляется переход от родителя.
 - b. Далее была реализована функция, которая вставляет строку в бор. Осуществляется проход по строке, которую необходимо вставить. Если проход по уже существующему бору невозможен, то создается новая вершина и добавляется в бор.
 - c. После были реализованы две функции, которые строят конечный детерминированный автомат по данному бору. Первая функция реализует получение суффиксной ссылки для данной вершины. Вторая функция выполняет переход из одного состояния автомата в другое.
 - d. Далее была реализована функция для поиска шаблона в исходном тексте.
2. Был реализован алгоритм, который, используя реализацию точного множественного поиска, решит задачу точного поиска для одного образца с джокером.
 - a. Идея работы алгоритма:

1. C — вектор длины T , инициализированный нулями.
 2. $\mathbb{P} = \{P_1, P_2, \dots, P_k\}$ — набор максимальных подстрок P без джокеров. l_1, l_2, \dots, l_k — начальные позиции этих подстрок в P . Для $P = ab??c?ab??$ $\mathbb{P} = \{ab, c, ab\}$ и $l_1 = 1$, $l_2 = 5$ и $l_3 = 7$
 3. Алгоритмом Ахо-Корасик найти все вхождения P_i в T .
Для каждого вхождения P_i в j -й позиции текста увеличить счётчик $C[j - l_i + 1]$ на единицу.
 4. Вхождение P в T , начинающиеся в позиции p , имеется в том и только том случае, если $C(p) = k$.
 5. Время поиска $O(km)$ из-за использования массива C , если k ограничено константой, не зависящей от $|P|$, то время поиска — линейно.
- б. Для корректной работы программы структура вершины бора была модифицирована. Теперь одна вершина может хранить информацию о нескольких шаблонах, которые в ней заканчиваются.

Пример работы

1. Поиск набора образцов:

```

ACTGNACTGN
2
NA
AC
1 2
5 1
6 2

```

2. Поиск образца с джокером:

```

TANGCCNANANCAT
AN??C?
?
2
8

```

Вывод

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Ахо-Корасик для поиска множества подстрок в строке на языке программирования C++. В нём используются такие понятия, как бор, конечный детерминированный автомат, суффиксные ссылки. Бор — это дерево, в котором каждая вершина обозначает какую-то строку. Ребра между вершинами содержат букву. Таким образом, добираясь по ребрам из корня в какую-нибудь вершину, мы получим строку, соответствующую этой вершине.

Детерминированным конечным автоматом называется такой автомат, в котором

нет дуг с меткой ϵ , и из любого состояния по любому символу возможен переход не более, чем в одно состояние. Суффиксная ссылка вершины 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.

Приложение А

Исходный код программы lr5_1.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdint>

using namespace std;

const int k=5;
const string first_alphabet = "ACGTN";

struct bohr_vrtx
{
    int next_vrtx[k];           //next_vrtx[i] – номер вершины, в которую мы придем по
    символу с номером i в алфавите
    int pat_num;               //номер строки-образца, обозначаемого этой вершиной
    bool flag;                 //бит, указывающий на то, является ли наша вершина исходной
    строкой
    int suff_link;             //суффиксная ссылка
    int auto_move[k];          //запоминание перехода
    int par;                   //вершина-отец в дереве
    unsigned char symb;         //символ на ребре от par к этой вершине
    int suff_flink;            //"хорошая" ссылка
};

vector<bohr_vrtx> bohr;
vector<string> pattern;

bohr_vrtx make_bohr_vrtx(int p, unsigned char c)
{
    bohr_vrtx v;
    memset(v.next_vrtx, 255, sizeof(v.next_vrtx));
    memset(v.auto_move, 255, sizeof(v.auto_move));
    v.flag=false;
    v.suff_link=-1;
    v.par=p;
    v.symb=c;
    v.suff_flink=-1;
    return v;
}

void bohr_init()
{
    bohr.push_back(make_bohr_vrtx(0, '\0'));
}

unsigned char get_number_from_alphabet(unsigned char symbol)
{
    for(size_t i = 0; i < first_alphabet.length(); i++)
    {
        if(symbol == first_alphabet[i])
        {
            return static_cast<unsigned char>(i);
        }
    }
    return 0;
}

void add_string_to_bohr(const string& s)
{

```

```

int num=0;
for (size_t i=0; i<s.length(); i++){
    unsigned char ch = get_number_from_alhabet(s[i]);
    if (bohr[num].next_vrtx[ch]==-1)
    {
        bohr.push_back(make_bohr_vrtx(num,ch));
        bohr[num].next_vrtx[ch]=bohr.size()-1;
    }
    num=bohr[num].next_vrtx[ch];
}
bohr[num].flag=true;
pattern.push_back(s);
bohr[num].pat_num=pattern.size()-1;
}

bool is_string_in_bohr(const string& s)
{
    int num=0;
    for (size_t i=0; i<s.length(); i++){
        unsigned char ch = get_number_from_alhabet(s[i]);
        if (bohr[num].next_vrtx[ch]==-1){
            return false;
        }
        num=bohr[num].next_vrtx[ch];
    }
    return true;
}

int get_auto_move(int v, unsigned char ch);

int get_suff_link(int v)
{
    if(bohr[v].suff_link==-1)
    {
        if (v==0||bohr[v].par==0)
            bohr[v].suff_link=0;
        else
            bohr[v].suff_link=get_auto_move(get_suff_link(bohr[v].par), bohr[v].symb);
    }
    return bohr[v].suff_link;
}

int get_auto_move(int v, unsigned char ch)
{
    if (bohr[v].auto_move[ch]==-1)
    {
        if (bohr[v].next_vrtx[ch]!=-1)
        {
            bohr[v].auto_move[ch]=bohr[v].next_vrtx[ch];
        }
        else
        {
            if (v==0)
                bohr[v].auto_move[ch]=0;
            else
                bohr[v].auto_move[ch]=get_auto_move(get_suff_link(v), ch);
        }
    }
    return bohr[v].auto_move[ch];
}

int get_suff_flink(int v)
{
    if (bohr[v].suff_flink==-1)
    {
        int u = get_suff_link(v);

```

```

        if(u == 0)
            bohr[v].suff_flink=0;
        else
            bohr[v].suff_flink = (bohr[u].flag) ? u : get_suff_flink(u);
    }
    return bohr[v].suff_flink;
}

void check(int v,int i)
{
    for(int u=v; u!=0; u=get_suff_flink(u))
    {
        if(bohr[u].flag)
            cout << i-pattern[bohr[u].pat_num].length() + 1 << " " << bohr[u].pat_num + 1 <<
endl;
    }
}

void find_all_pos(const string& s)
{
    int u=0;
    for(size_t i=0;i<s.length();i++)
    {
        u = get_auto_move(u, get_number_from_alhabet(s[i]));
        check(u,i+1);
    }
}

int main()
{
    size_t n;
    string text, pat;
    cin >> text >> n;

    bohr_init();
    for(size_t i = 0; i < n; i++)
    {
        cin >> pat;
        add_string_to_bohr(pat);
        pat.clear();
    }
    find_all_pos(text);

    return 0;
}

```

Приложение Б

Исходный код программы lr5_2.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdint>
#include <cstring>

using namespace std;

const int k = 5;
const string first_alphabet = "ACGTN";

struct bohr_vrtx
{
    int next_vrtx[k];           //next_vrtx[i] – номер вершины, в которую мы придем по
    символу с номером i в алфавите
    int pat_num;               //номер строки-образца, обозначаемого этой вершиной
    bool flag;                 //бит, указывающий на то, является ли наша вершина
    исходной строкой
    int suff_link;             //суффиксная ссылка
    int auto_move[k];          //запоминание перехода
    int par;                   //вершина-отец в дереве
    unsigned char symb;         //символ на ребре от par к этой вершине
    int suff_flink;            //"хорошая" ссылка
    int pattern_num[40];
};

struct WildCardPattern
{
    size_t index;
    int pattern_num;

    explicit WildCardPattern(long long int o_index, int o_pat_num) : index(o_index),
    pattern_num(o_pat_num)
    {}
};

vector<bohr_vrtx> bohr;
vector<string> pattern;
vector<WildCardPattern> wc_patterns;

bohr_vrtx make_bohr_vrtx(int p, unsigned char c)
{
    bohr_vrtx v;
    memset(v.next_vrtx, 255, sizeof(v.next_vrtx));
    memset(v.auto_move, 255, sizeof(v.auto_move));
    v.flag = false;
    v.suff_link = -1;
    v.par = p;
    v.symb = c;
    v.suff_flink = -1;
    memset(v.pattern_num, 255, sizeof(v.pattern_num));
    return v;
}

void bohr_init()
{
    bohr.push_back(make_bohr_vrtx(0, '\0'));
}
```



```

unsigned char get_number_from_alphabet(unsigned char symbol)
{
    for (size_t i = 0; i < first_alphabet.length(); i++)
    {
        if (symbol == first_alphabet[i])
        {
            return static_cast<unsigned char>(i);
        }
    }
    return 0;
}

void add_string_to_bohr(const string& s)
{
    int num = 0;
    for (size_t i = 0; i < s.length(); i++)
    {
        unsigned char ch = get_number_from_alphabet(s[i]);
        if (bohr[num].next_vrtx[ch] == -1)
        {
            bohr.push_back(make_bohr_vrtx(num, ch));
            bohr[num].next_vrtx[ch] = bohr.size() - 1;
        }
        num = bohr[num].next_vrtx[ch];
    }
    bohr[num].flag = true;
    pattern.push_back(s);
    //bohr[num].pat_num = pattern.size() - 1;
    for (size_t i = 0; i < 40; i++)
    {
        if (bohr[num].pattern_num[i] == -1) {
            bohr[num].pattern_num[i] = pattern.size() - 1;
            break;
        }
    }
}

int get_auto_move(int v, unsigned char ch);

int get_suff_link(int v)
{
    if (bohr[v].suff_link == -1)
    {
        if (v == 0 || bohr[v].par == 0)
            bohr[v].suff_link = 0;
        else
            bohr[v].suff_link = get_auto_move(get_suff_link(bohr[v].par),
bohr[v].symb);
    }
    return bohr[v].suff_link;
}

int get_auto_move(int v, unsigned char ch)
{
    if (bohr[v].auto_move[ch] == -1)
    {
        if (bohr[v].next_vrtx[ch] != -1)
        {
            bohr[v].auto_move[ch] = bohr[v].next_vrtx[ch];
        }
        else
        {
            if (v == 0)
                bohr[v].auto_move[ch] = 0;
            else
                bohr[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
        }
    }
}

```

```

    }
    }
    return bohr[v].auto_move[ch];
}

int get_suff_flink(int v)
{
    if (bohr[v].suff_flink == -1)
    {
        int u = get_suff_link(v);
        if (u == 0)
            bohr[v].suff_flink = 0;
        else
            bohr[v].suff_flink = (bohr[u].flag) ? u : get_suff_flink(u);
    }
    return bohr[v].suff_flink;
}

void check(int v, int i)
{
    for (int u = v; u != 0; u = get_suff_flink(u))
    {
        if (bohr[u].flag)
        {
            for (size_t j = 0; j < 40; j++)
            {
                if (bohr[u].pattern_num[j] != -1)
                {
                    wc_patterns.push_back(WildCardPattern(i -
pattern[bohr[u].pattern_num[j]].length(), bohr[u].pattern_num[j]));
                }
                else
                    break;
            }
        }
    }
}

void find_all_pos(const string& s)
{
    int u = 0;
    for (size_t i = 0; i < s.length(); i++)
    {
        u = get_auto_move(u, get_number_from_alphabet(s[i]));
        check(u, i + 1);
    }
}

int divide_P(const string& P, char wild_card, vector<int>& patterns_pos)
{
    int counter_P = 0;
    string tmp;
    for (size_t i = 0; i < P.length(); i++)
    {
        if (P[i] == wild_card)
        {
            if (tmp.empty())
                continue;

            patterns_pos.push_back(i - tmp.length());
            counter_P++;
            add_string_to_bohr(tmp);
            tmp.clear();
        }
        else
        {

```

```

        tmp = tmp + P[i];
    }
}

if (!tmp.empty())
{
    patterns_pos.push_back(P.length() - tmp.length());
    counter_P++;
    add_string_to_bohr(tmp);
}

return counter_P;
}

int main()
{
    vector<int> patterns_pos;
    char wild_card;
    string T, P;
    cin >> T >> P >> wild_card;

    bohr_init();
    int counter = divide_P(P, wild_card, patterns_pos);
    find_all_pos(T);

    int T_size = static_cast<int>(T.length());
    int P_size = static_cast<int>(P.length());

    vector<int> C(T_size, 0);

    for (size_t i = 0; i < wc_patterns.size(); i++)
    {
        if (wc_patterns[i].index < patterns_pos[wc_patterns[i].pattern_num])
            continue;
        C[wc_patterns[i].index - patterns_pos[wc_patterns[i].pattern_num]]++;
        if (C[wc_patterns[i].index - patterns_pos[wc_patterns[i].pattern_num]] ==
counter && wc_patterns[i].index - patterns_pos[wc_patterns[i].pattern_num] <= T_size -
P_size)
            cout << wc_patterns[i].index - patterns_pos[wc_patterns[i].pattern_num]
+ 1 << endl;
    }

    return 0;
}

```