Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Отчёт

по курсовой работе

Дисциплина: Объектно-ориентированное программирование

Тема: Aho-Corasick algorithm

Студент гр. 3331506/90401 Вестников Р. Н.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Алгоритм Ахо – Корасик — алгоритм поиска подстроки, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, реализует поиск множества подстрок из словаря в данной строке. Широко применяется в системном программном обеспечении, например, используется в утилите поиска *grep*.

Задача алгоритма: Дан набор строк в алфавите размера k суммарной длины m. Необходимо найти для каждой строки все её вхождения в текст.

Описание алгоритма

Пусть дан набор строк $s_1, s_1, \dots, s_m$ алфавита размера k суммарной длины n, называемый словарем, и длинный текст t. Необходимо определить, есть ли в тексте хотя бы одно слово из словаря, и если есть, то на какой позиции.

Для примера будем искать в строке “*aabccbbbabcdaddcd*” следующие подстроки:

1) “*abc*”
2) “*bbb*”
3) “*add*”
4) “*cd*”
5) “*c*”
6) “*abcd*”

Реализация алгоритма осуществляется по шагам.

Первый шаг: построим по словарю бор (префиксное дерево).

Бор – структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной.

Бор для рассматриваемого словаря представлен на рис. 1.

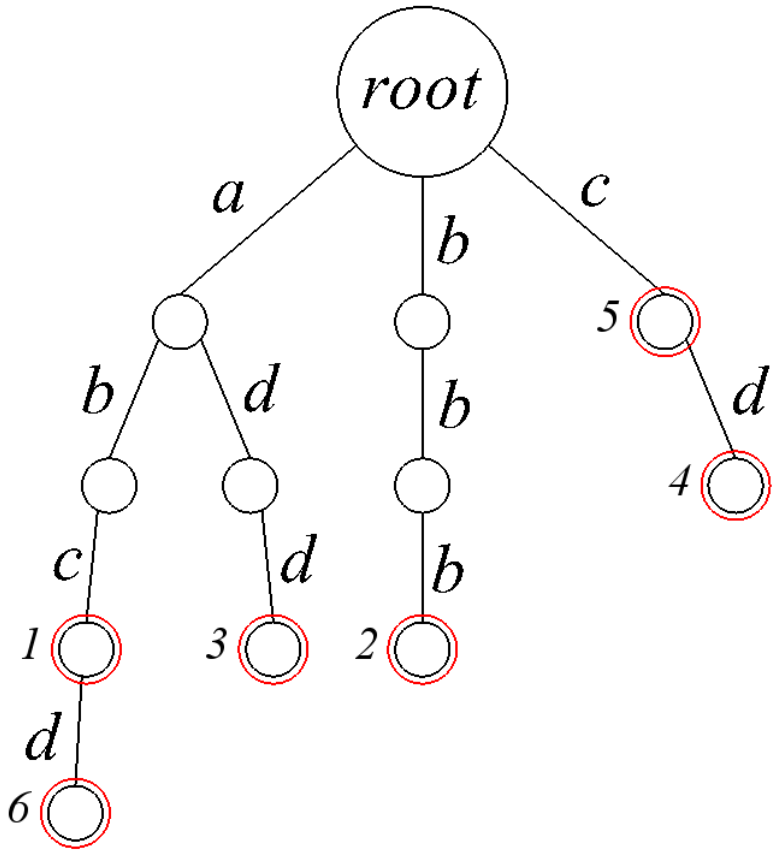

Рис. 1. Пример префиксного дерева (бора)

Второй шаг: построим конечный детерминированный автомат.

Конечный автомат – математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных. Является частным случаем абстрактного дискретного автомата, число возможных внутренних состояний которого конечно.

Если мы рассмотрим любую вершину бора, то строка, которая соответствует ей, является префиксом одной или нескольких строк из набора. Т.е. каждую вершину бора можно понимать как позицию в одной или нескольких строках из набора.

Фактически, вершины бора можно понимать как состояния конечного детерминированного автомата. Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние — т.е. в

другую позицию в наборе строк.

Т.е. мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние. Для этого нам нужны *суффиксные ссылки*.

Суффиксная ссылка для каждой вершины *p* – это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине *p*. Единственный особый случай – корень бора, для удобства суффиксную ссылку из него проведём в себя же. Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке.

Таким образом, мы свели задачу построения автомата к задаче нахождения суффиксных ссылок для всех вершин бора. Однако строить эти суффиксные ссылки мы будем, как ни странно, наоборот, с помощью построенных в автомате переходов.

Заметим, что если мы хотим узнать суффиксную ссылку для некоторой вершины *v*, то мы можем перейти в предка *par* текущей вершины (пусть *symb* – буква, по которой из *par* есть переход в *v*), затем перейти по его суффиксной ссылке, а затем из неё выполнить переход в автомате по букве *symb*. Иллюстрация данного действия представлена на рис. 2.

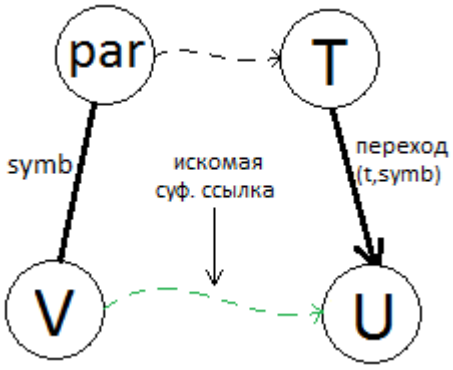

Рис. 2. Нахождение суффиксной ссылки

Таким образом, задача нахождения перехода свелась к задаче нахождения суффиксной ссылки, а задача нахождения суффиксной ссылки – к задаче нахождения суффиксной ссылки и перехода, но уже для более близких к корню вершин. Мы получили рекурсивную зависимость, но не бесконечную, и, более того, разрешить которую можно за линейное время.

Реализация бора с расставленными суффиксными ссылками представлена на рис. 3.

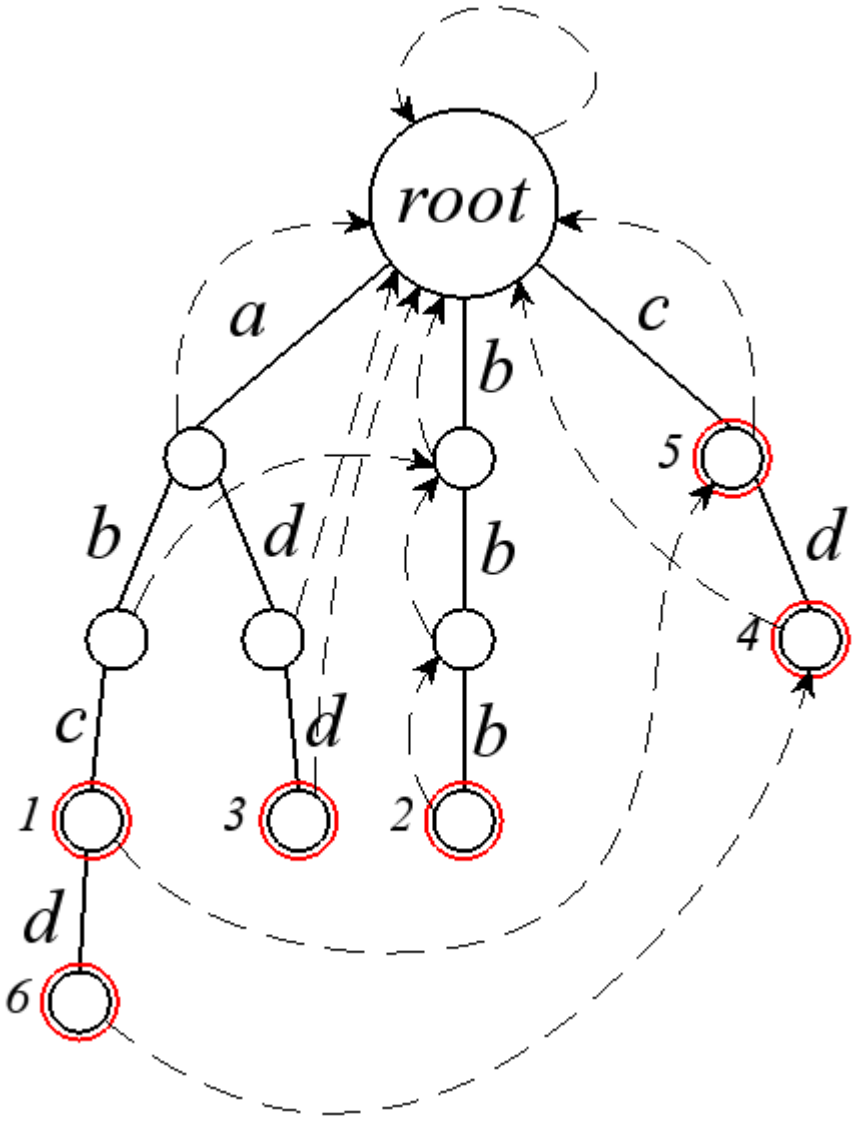

Рис. 3. Бор с расставленными суффиксными ссылками

Третий шаг: Построение сжатых суффиксных ссылок.

С автоматом несложно определить сам алгоритм: считываем строку, двигаемся из состояния в состояние по символам строки, в каждом из состояний двигаемся по суффиксным ссылкам, то есть по суффиксам строки в позиции автомата, проверяя при этом наличие их в боре.

Все бы ничего, но оказывается, что этот вариант Ахо-Корасика имеет квадратичную асимптотику относительно длины считываемой строки. Наша цель в том, чтобы двигаясь по суффиксным ссылкам попадать только в заведомо имеющиеся среди строк-образцов. Для этого введем понятие сжатых суффиксных ссылок. Сжатая суффиксная ссылка – это ближайший суффикс, имеющийся в боре, для которого $flag == true$. Число «скачков» при использование таких ссылок уменьшится и станет пропорционально количеству искомых вхождений, оканчивающихся в этой позиции. Аналогично обычным суффиксным ссылкам сжатые суффиксные ссылки могут быть найдены при помощи ленивой рекурсии.

Визуализация сжатых ссылок в сравнении с обычными суффиксными ссылками представлена на рис.4. Как видно из рисунка, переход по сжатой ссылке значительно быстрее перехода по нескольким суффиксным.

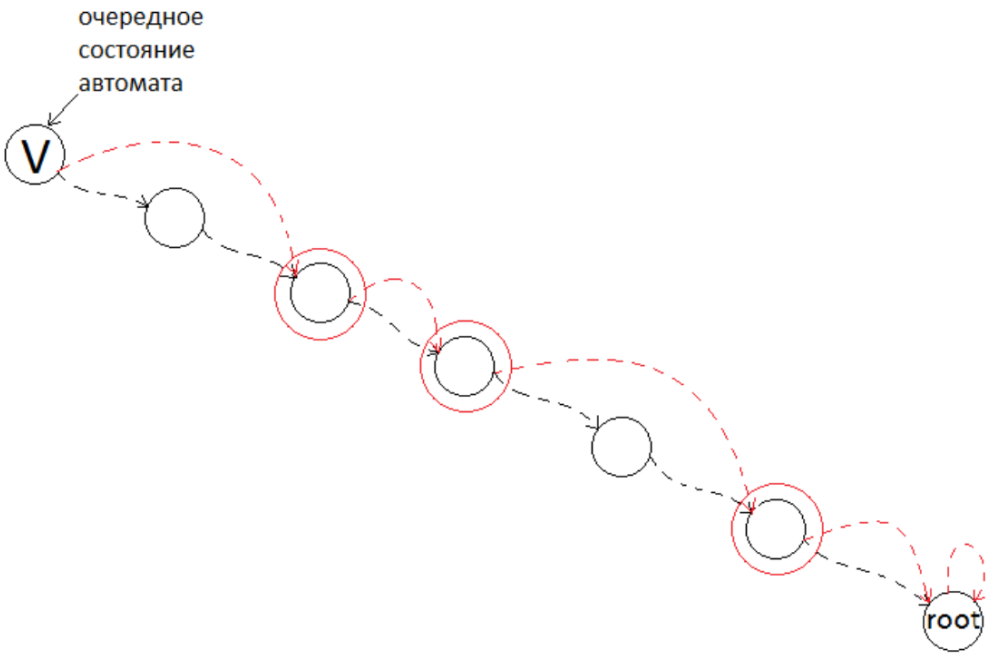

Рис. 4. Сжатые суффиксные ссылки

Результат применения алгоритма для исследуемой строки:

2 abc
4 c
5 c
6 bbb
9 abc
11 c
9 abcd
11 cd
13 add
16 c
16 cd

Как видим алгоритм успешно нашёл все возможные подстроки в исходной строке и указал их место.

Исследование алгоритма

Зависимость от размера входной строки

Произведём замеры времени выполнения алгоритма поиска подстрок в строке от размера входных данных (размера строки). Будем постепенно увеличивать входные данные при неизменном алфавите ($k = 10$) и искомых строках (бор не меняется). Зависимость скорости алгоритма от размера строки представлена в таблице 1 и на рис. 5.

Таблица 1 – Исследование скорости алгоритма от размера входной строки

Число символов исследуемой строки, млн	Время выполнения алгоритма, мс
1	62,0
2	156
3	171
4	218
5	281
6	322
7	375
8	406
9	467
10	498

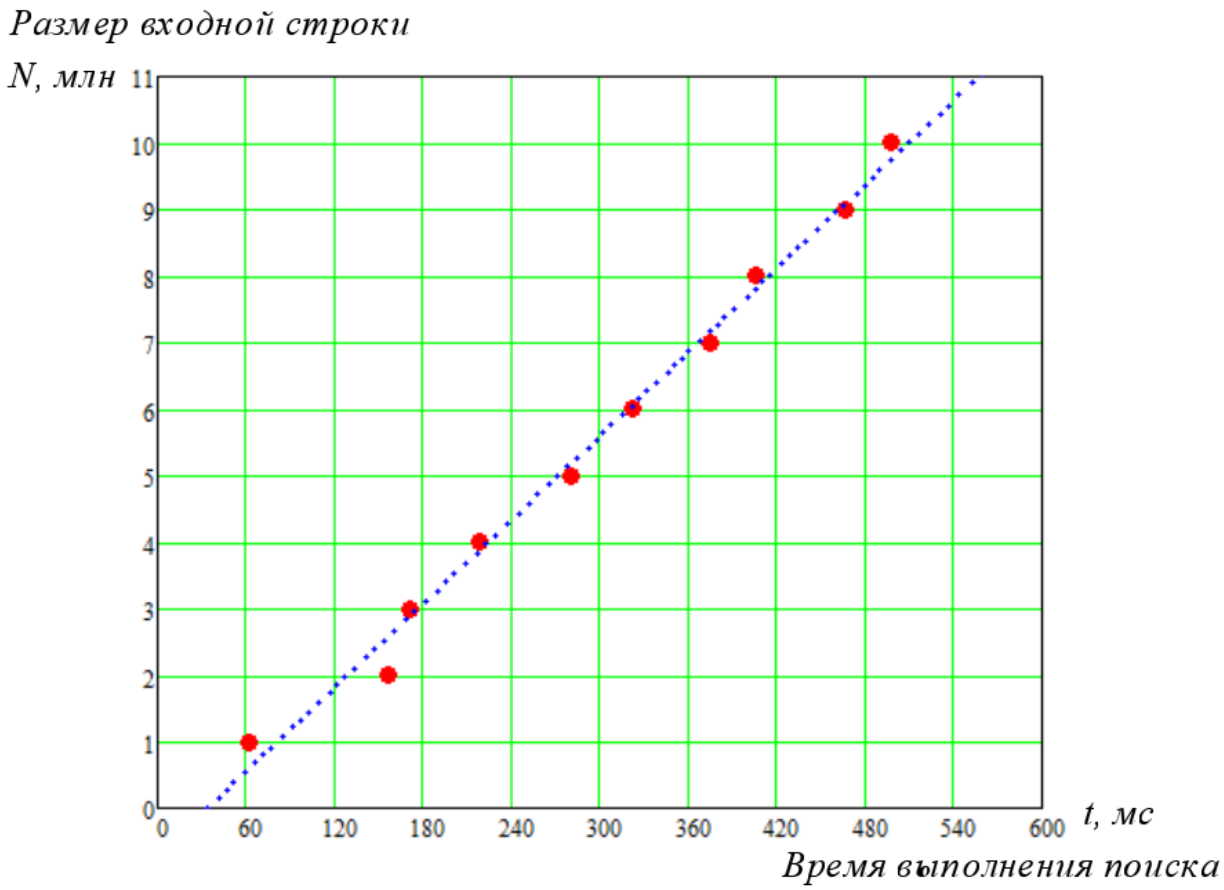

Рис. 5. График результатов исследования скорости от входной строки

Аппроксимировав все точки, можем видеть линейную зависимость времени от размера входных данных.

Зависимость от размера словаря

Далее будем увеличивать размер словаря. Входная строка будет постоянна и равна 100 тысячам символов, а размер алфавита равен 20. Зависимость скорости построения дерева от размера словаря представлена в таблице 2 и на рис. 6.

Таблица 2 – Исследование скорости алгоритма от размера словаря

Число слов в словаре, тыс	Время построения бора, мс
10	15,0
20	46,0
30	61,0
40	77,0
50	108
60	110
70	156
80	156
90	172
100	202

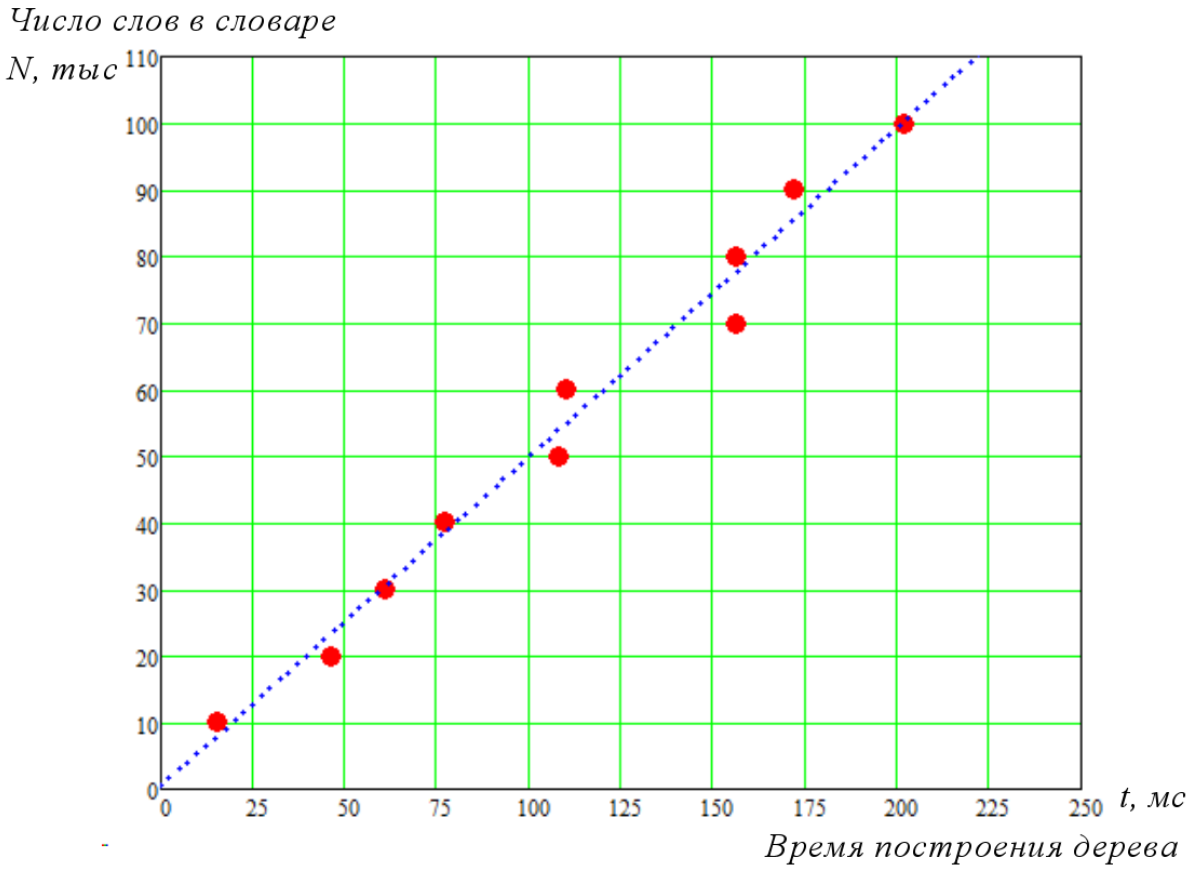

Рис. 6. График результатов исследования скорости от размера словаря

Аналогично можем видеть линейную зависимость.

Зависимость от размера алфавита

Теперь будем менять размер алфавита в диапазоне от 4 до 26. Алфавит представляет из себя латиницу, мелкого регистра. Размер входной строки равен 100 тысячам символов, размер словаря – 10 тысячам слов. Результаты представлены в таблице 3.

Таблица 3 – Исследование скорости алгоритма от размера алфавита

Число символов в алфавите	Время выполнения алгоритма, мс
4	30,0
6	30,0
8	30,0
10	30,0
12	30,0
14	31,0
18	30,0
20	30,0
24	30,0
26	31,0

Как видим, время выполнения алгоритма не меняется. Следовательно, можем сделать вывод, что размер алфавита не влияет на время поиска подстроки в строке.

Анализируя все временные исследования алгоритма, можем утверждать, что алгоритм зависит только от размеров входной строки и от размера словаря. Обе зависимости выполняются по линейному закону, что легко позволяет высчитывать время его выполнения.

Заключение

Подводя итог, можно резюмировать, что в данной работе был исследован алгоритм Ахо-Корасик. Был подробно разобран и расписан ход его выполнения. Также был написан код, реализующий этот алгоритм на языке C++. Благодаря этому были исследованы временные зависимости выполнения данного алгоритма при различных входных параметрах и сделаны соответствующие выводы. Общий вывод таков, что алгоритм позволяет максимально быстро обрабатывать огромные массивы подстрок для нахождения их в большом тексте. Благодаря этому алгоритм нашёл широкое применение в самых различных сферах.

Список литературы

1. Ахо А. В., Хопкрофт Дж. Э., Ульман Дж. Д. Структуры данных и алгоритмы — М.: Вильямс, 2003. — 384 с.

2. Алгоритм Ахо-Корасик, Хабр. URL: https://habr.com/ru/post/198682/

3. Algorithms for Competitive Programming. URL: https://cp-algorithms.com/string/aho_corasick.html

4. Алгоритм Ахо – Корасик, Википедия. URL: https://ru.wikipedia.org/wiki/Алгоритм_Ахо_—_Корасик

ПРИЛОЖЕНИЕ

Код заголовочного файла программы:

```
#include <vector>
#include <cstring>

using namespace std;

#ifndef ALGORITHM_AHO_CORASICK_H
#define ALGORITHM_AHO_CORASICK_H

const int k = 26;  // alphabet size

struct pr_tree_vertex {
    int next_vertex[k];  // vertex number
    int pat_num;  // sample line number
    bool flag;  // the bit indicating whether our vertex is the original
string

    int suff_link;  // suffix link
    int suff_link_comp;  // "compressed" suffix link

    int auto_move[k];  // memorizing the transition of the automaton
    int par;  // the vertex-parent in the tree
    char symb;  // the symbol on the rim from par to this vertex
};

class Prefix_tree{
public:
    Prefix_tree();
    ~Prefix_tree() = default;
public:
    void check(int v, int i);
    void find_all_pos(const string &s);
    void add_str_to_tree(const string &s);
private:
    vector<pr_tree_vertex> tree;
    vector<string> pattern;
private:
    pr_tree_vertex make_tree_vertex(int p, char c);
    int get_suff_link_comp(int v);
    int get_auto_move(int v, char ch);
    int get_suff_link(int v);
    bool is_string_in_tree(const string &s);
};

#endif //ALGORITHM_AHO_CORASICK_H
```

Код *.cpp* файла программы:

```
#include "aho_corasick.h"

Prefix_tree::Prefix_tree() {
    tree.push_back(make_tree_vertex(0, '$'));
}

pr_tree_vertex Prefix_tree::make_tree_vertex(int p, char c) {
    pr_tree_vertex v;  //create tree
    memset(v.next_vertex, 255, sizeof(v.next_vertex));  //255 = -1
    memset(v.auto_move, 255, sizeof(v.auto_move));
    v.flag = false;
    v.suff_link = -1;  //initially, there is no suf. link
    v.suff_link_comp = -1;
    v.par = p;
    v.symb = c;
    return v;
}

void Prefix_tree::add_str_to_tree(const string &str) {
    int num = 0;  //starting from the root
    for (int i = 0; i < str.length(); i++) {
        char ch = str[i] - 'a';  //get the number in the alphabet
        if (tree[num].next_vertex[ch] == -1) {  //-1 - the sign of the
absence of an rib
            tree.push_back(make_tree_vertex(num, ch));
            tree[num].next_vertex[ch] = tree.size() - 1;
        }
        num = tree[num].next_vertex[ch];
    }
    tree[num].flag = true;
    pattern.push_back(str);
    tree[num].pat_num = pattern.size() - 1;
}

bool Prefix_tree::is_string_in_tree(const string &s) {
    int num = 0;
    for (int i = 0; i < s.length(); i++) {
        char ch = s[i] - 'a';
        if (tree[num].next_vertex[ch] == -1) {
            return false;
        }
        num = tree[num].next_vertex[ch];
    }
    return true;
}

int Prefix_tree::get_suff_link(int v) {
    if (tree[v].suff_link == -1)
        if (v == 0 || tree[v].par == 0)  //if 'v' is the root or the parent
of 'v' is the root
            tree[v].suff_link = 0;
        else
            tree[v].suff_link = get_auto_move(get_suff_link(tree[v].par),
tree[v].symb);
    return tree[v].suff_link;
}

int Prefix_tree::get_auto_move(int v, char ch) {
    if (tree[v].auto_move[ch] == -1)
        if (tree[v].next_vertex[ch] != -1)
            tree[v].auto_move[ch] = tree[v].next_vertex[ch];
```

```
        else if (v == 0)
            tree[v].auto_move[ch] = 0;
        else
            tree[v].auto_move[ch] = get_auto_move(get_suff_link(v), ch);
    return tree[v].auto_move[ch];
}

int Prefix_tree::get_suff_link_comp(int v) {
    if (tree[v].suff_link_comp == -1) {
        int u = get_suff_link(v);
        if (u == 0)  //either 'v' is the root, or the suf. link 'v' points to
the root
            tree[v].suff_link_comp = 0;
        else if (tree[u].flag) {
            tree[v].suff_link_comp = u;
        } else tree[v].suff_link_comp = get_suff_link_comp(u);
    }
    return tree[v].suff_link_comp;
}

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

void Prefix_tree::find_all_pos(const string &s) {
    int u = 0;
    for (int i = 0; i < s.length(); i++) {
        u = get_auto_move(u, s[i] - 'a');
        check(u, i + 1);
    }
}
```