

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

**Курсовая работа**  
**Алгоритм Ахо-Карасик**  
по дисциплине: Объектно-ориентированное программирование

Студент гр. 3331506/70401

Демчева А.А.

Преподаватель

Ананьевский М.С.

« \_\_\_\_ » \_\_\_\_\_ 2020 г.

Санкт-Петербург

2020 г.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	3
1 ОПИСАНИЕ АЛГОРИТМА.....	4
1.1. Построение бора .....	4
1.2. Построение конечного детерминированного автомата.....	5
1.3. Добавление суффиксных ссылок .....	5
1.4. Краткая последовательность действий .....	6
2 РЕАЛИЗАЦИЯ АЛГОРИТМА .....	8
3 АНАЛИЗ АЛГОРИТМА.....	13
ЗАКЛЮЧЕНИЕ .....	14
СПИСОК ЛИТЕРАТУРЫ.....	15

## ВВЕДЕНИЕ

Алгоритм Ахо-Корасик является классическим решением задачи точного сопоставления множеств и реализует эффективный поиск всех вхождений заданных подстрок в основную строку.

Был разработан в 1975 году Альфредом Ахо и Маргарет Корасик. На данный момент широко используется в системном ПО, в частности — в утилите поиска *grep* (*Linux*). Этим обусловлена актуальность данной работы.

Цель курсовой работы — изучение механизма работы алгоритма Ахо-Корасик.

Поставленные задачи:

- 1) реализация алгоритма на языке C++;
- 2) анализ эффективности и сложности алгоритма.

# 1 ОПИСАНИЕ АЛГОРИТМА

Задача точного сопоставления множеств алгоритма Ахо-Корасик сформулирована следующим образом: на вход поступают набор слов и строка-образец, требуется найти все возможные вхождения слов в заданную строку.

Суть алгоритма заключается в использовании бора и построения по нему конечного детерминированного автомата.

## 1.1. Построение бора

Бор или префиксное дерево — структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Представляет собой корневое дерево, каждое ребро которого помечено некоторым символом. В отличие от бинарных деревьев поиска, идентифицирующий узел ключ не хранится в нем явно, а задаётся положением в дереве данного узла. Таким образом, получить ключ можно выписыванием подряд символов, помечающих рёбра, на пути от корня до узла. Ключ корня дерева — пустое слово.

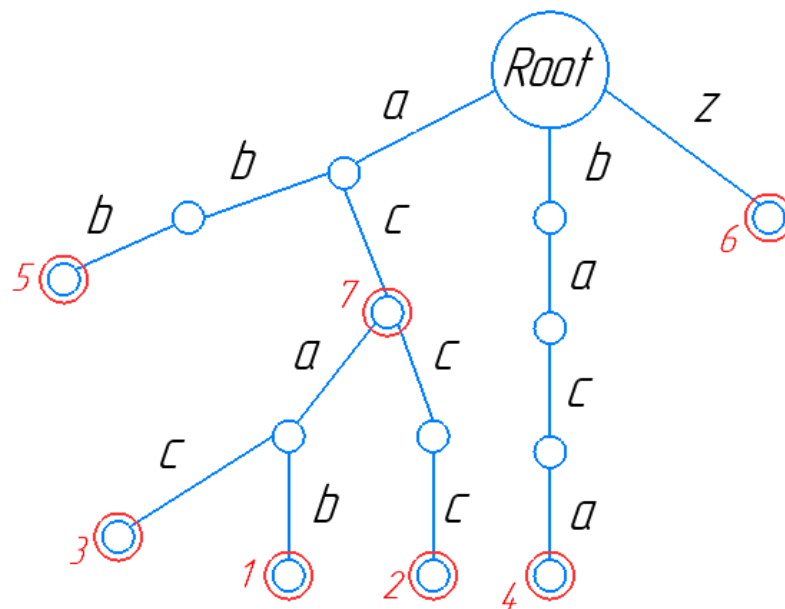


Рисунок 1 — Пример бора

На рисунке 1 показан пример бора, построенного для слов 1) *acab*, 2) *acss*, 3) *acac*, 4) *baca*, 5) *abb*, 6) *z*, 7) *ac*. Заметим, что два слова в боре имеют

общие ребра при наличии у них общего префикса. Так как одно из слов может полностью совпадать с префиксом другого слова (например, 7, 3, 1), возникает необходимость дополнительно хранить признак конца слова (обозначены красными кругами на рисунке 1).

### **1.2. Построение конечного детерминированного автомата**

Следующий шаг алгоритма — построение конечного детерминированного автомата на основе полученного бора.

Конечный автомат — до предела упрощенная модель компьютера, имеющая конечное число состояний, которая жертвует такими особенностями, как ОЗУ, постоянная память и так далее в обмен на простоту понимания и легкость программной или аппаратной реализации. «Детерминированный» — обозначает, что автомат в каждый момент времени может находиться только в одном состоянии.

В данном случае состояния автомата соответствуют вершинам бора. Изменение состояния — переход по ребру, соответствующему следующей букве в искомом слове. Если требуемый переход выполнить невозможно (в узле нет ребра с нужным символом), используются суффиксные ссылки.

### **1.3. Добавление суффиксных ссылок**

Суффиксная ссылка вершины  $v$  — указатель на вершину  $u$ , такую, что строка  $u$  — наибольший собственный суффикс строки  $v$ , или, если такая вершина не найдена, — указатель на корень. Суффиксная ссылка из корня ведет в него же. На рисунке 2 показана расстановка суффиксных ссылок для рассматриваемого бора.

Получение ссылки для вершины  $v$  происходит следующим образом. Автомат поднимается до вершины-предка  $v$  (назовем его *parent*), переходит по его суффиксной ссылке к вершине  $w$ , а из нее запускает переход по ребру с таким же символом, как на ребре между  $v$  и *parent* (рисунок 3).

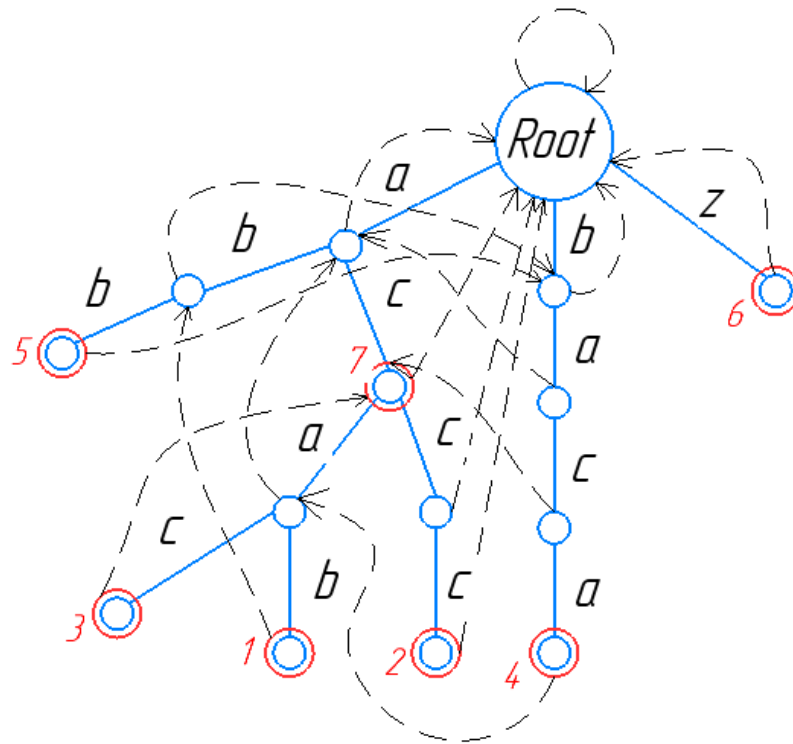


Рисунок 2 — Расстановка суффиксных ссылок

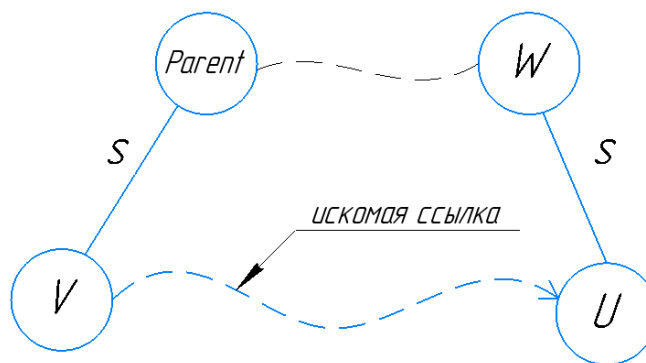


Рисунок 3 — Получение суффиксной ссылки для вершины  $v$

Таким образом, если из текущей вершины ведет ребро с искомым символом, автомат осуществляет по нему переход, иначе — переход по суффиксной ссылке к новой вершине с наиболее похожим префиксом. Процесс повторяется рекурсивно. Если автомат пришел к терминальному состоянию (попал в вершину, соответствующую концу слова), искомое слово присутствует в заданной строке.

#### 1.4. Краткая последовательность действий

Итак, последовательность действий для нахождения всех вхождений слова в заданную строку по алгоритму Ахо-Корасик:

1. Построить бор на основе заданных слов.
2. Преобразовать бор к виду конечного детерминированного автомата, ввести суффиксные ссылки.
3. Последовательно перебирая символы заданной строки, осуществлять переход по соответствующему ребру или суффиксной ссылке, пока автомат не достигнет терминального значения.

## 2 РЕАЛИЗАЦИЯ АЛГОРИТМА

Алгоритм Ахо-Корасик был реализован на языке C++ для латинского алфавита (26 символов).

Бор и набор искомых слов были представлены с помощью векторов, позволило использовать стандартные методы этой структуры данных (такие как *size()* и *push\_back()*), а также избежать динамического выделения памяти «вручную».

Вершины бора были представлены с помощью экземпляров структуры *vertex*, каждая из которых имеет свой номер и хранится как элемент вектора *bohr*.

Добавление слова в бор реализовано в функции *add\_word\_to\_bohr*, проверка его наличия в боре — функцией *is\_string\_in\_bohr*. Функции *get\_auto\_move*, *get\_suff\_link* и *get\_g\_suff\_link* осуществляют поиск и выбор «хороших» суффиксных ссылок. Функция *check* проверяет, является ли состояние автомата терминальным. Наконец, *find\_all\_pos* выполняет поиск вхождений слова в заданную строку.

«Хорошая» суффиксная ссылка — ближайший суффикс, имеющийся в боре, для которого *flag = true*. Это понятие было введено для увеличения эффективности алгоритма и уменьшения числа «скачков» между вершинами бора.

На вход алгоритма подается набор искомых слов и строка, в которой осуществляется поиск. Результат работы программы выводится в виде номера символа в строке, с которого начинается искомое слово. Листинг программы и пример ее выполнения показаны ниже.



```

#include <iostream>
#include <vector>

using namespace std;

const int k = 26; //k - размер алфавита

struct vertex
{
    int next_vertex[k]; // номер вершины, в которую мы придем по символу
                        // с номером в алфавите
    int suff_link; // суффиксная ссылка
    int g_suff_link; //'хорошая' суффиксная ссылка
    int word_num; //номер слова, обозначаемого этой вершиной
    bool flag; //Флаг, обозначающий, что вершина является окончанием
               // слова - терминальным состоянием
    int auto_move[k]; //Запись переходов автомата - для расчета суфф
                     // ссылки
    int parent; //Номер вершины-родителя
    char symb; //Символ на ребре между parent и этой вершиной
};

vector <vertex> bohr; //Вектор для хранения бора
vector <string> pattern; //Вектор искомых слов

vertex create_vertex(int p, char c)
{
    vertex v;
    memset(v.next_vertex, 255, sizeof(v.next_vertex)); // "-1" -
                                                         // отсутствие ребра
    memset(v.auto_move, 255, sizeof(v.auto_move));
    v.flag = false;
    v.suff_link = -1; //изначально суф. ссылки нет
    v.g_suff_link = -1; //как и "хорошей" суф. ссылки
    v.parent = p;
    v.symb = c;
    return v;
}

void create_root()
{
    bohr.push_back(create_vertex(0, '$'));
    return;
}

void add_word_to_bohr(const string& word1)
{
    int vertex_num1 = 0; //Начинаем с корня
    for (size_t i = 0; i < word1.length(); i++)
    {
        char letter1 = word1[i] - 'a'; //Номер текущей буквы в алфавите
    }
}

```

```

    if (bohr[vertex_num1].next_vertex[letter1] == -1)
    {
        bohr.push_back(create_vertex(vertex_num1, letter1));
        bohr[vertex_num1].next_vertex[letter1] = bohr.size() - 1;
    }
    vertex_num1 = bohr[vertex_num1].next_vertex[letter1];
}
bohr[vertex_num1].flag = true; //Дойдя до конца слова, ставим флаг
                               терминального состояния
pattern.push_back(word1); //Добавляем ячейку в конце вектора для
                           хранения слов
bohr[vertex_num1].word_num = pattern.size() - 1; //Переходим к
                                                  следующему слову
return;
}

bool is_string_in_bohr(const string& word2)
{
    int vertex_num2 = 0; //Начинаем с корня
    for (size_t i = 0; i < word2.length(); i++) //Для каждой буквы слова
    {
        char letter2 = word2[i] - 'a'; //Номер текущей буквы в алфавите
        if (bohr[vertex_num2].next_vertex[letter2] == -1) //Если нужного
                                                         перехода нет
        {
            return false;
        }
        vertex_num2 = bohr[vertex_num2].next_vertex[letter2]; //Переходим
                                                                к следующей вершине
    }
    return true;
}

int get_auto_move(int v, char letter);

int get_suff_link(int v) //Функция определения суффиксной ссылки
{
    if (bohr[v].suff_link == -1) //если ссылка еще не была найдена
        if (v == 0 || bohr[v].parent == 0) //если текущий узел или его
                                           предок - корень
            bohr[v].suff_link = 0; //Принимаем в качестве ссылки 0
        else
            bohr[v].suff_link =
get_auto_move(get_suff_link(bohr[v].parent), bohr[v].symb); //Иначе -
        //возвращаемся к родителю и запускаем поиск ссылки от него
    return bohr[v].suff_link;
}

```

```

int get_auto_move(int v, char letter)
{
    if (bohr[v].auto_move[letter] == -1) //Если состояние уже было
                                         найдено
        if (bohr[v].next_vertex[letter] != -1) //Если определен переход к
                                                следующей вершине
            bohr[v].auto_move[letter] = bohr[v].next_vertex[letter];
            //Принимаем его в качестве суфф. ссылки
        else
            if (v == 0) //Если рассматриваемая вершина - корень
                bohr[v].auto_move[letter] = 0; //Принимаем в качестве ссылки 0
            else
                bohr[v].auto_move[letter] = get_auto_move(get_suff_link(v),
letter); //Иначе - продолжаем поиск
    return bohr[v].auto_move[letter];
}

int get_g_suff_link(int v)
{
    if (bohr[v].g_suff_link != -1) //если "хорошая" ссылка еще не найдена
        return bohr[v].g_suff_link;
    int u = get_suff_link(v);
    if (u == 0) //если v - корень, или его суфф. ссылка указывает на
                                                         корень
        bohr[v].g_suff_link = 0; //Тогда "хорошая" ссылка = 0
    else
    {
        if (bohr[u].flag)
            bohr[v].g_suff_link = u;
        else
            bohr[v].g_suff_link = get_g_suff_link(u);
    }
    return bohr[v].g_suff_link;
}

void check(int v, int i) // i - последняя рассмотренная буква в искомом
                        слове
{
    for (int u = v; u != 0; u = get_g_suff_link(u))
    {
        if (bohr[u].flag) //Если автомат пришел в терминальное состояние
            cout << i - pattern[bohr[u].word_num].length() + 1 << " " <<
pattern[bohr[u].word_num] << endl;
    }
}

void find_all_pos(const string& s)
{
    int u = 0;
    for (size_t i = 0; i < s.length(); i++)
    {

```

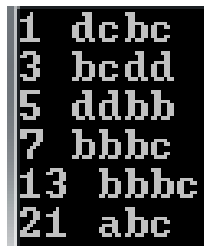
```

        u = get_auto_move(u, s[i] - 'a');
        check(u, i + 1);
    }
}

int main()
{
    create_root();
    add_word_to_bohr("abc");
    add_word_to_bohr("dcbc");
    add_word_to_bohr("ddbb");
    add_word_to_bohr("bcdd");
    add_word_to_bohr("bbbc");
    find_all_pos("dcbcddbbccbbccbbabc");
}

```

Результат работы программы:



```

1  dcbe
3  bcdd
5  ddbb
7  bbbc
13 bbbc
21 abc

```

Рисунок 4 — Результат выполнения программы

### 3 АНАЛИЗ АЛГОРИТМА

Данный вариант алгоритма проходит циклом по длине заданной строки  $s$  ( $N = s.length()$ ), откуда его уже можно оценить как  $O(N \cdot O(check))$ . Но так как  $check$  переходит только по заранее помеченным вершинам («хорошие» суффиксные ссылки), то общую асимптотику можно оценить как  $O(N + t)$ , где  $t$  — количество всех возможных вхождений всех строк-образцов в  $s$ .

В общем случае вычислительную сложность алгоритма Ахо-Корасик можно оценить следующим выражением:

$$O(wk + s + n)$$

где  $w$  — общая длина заданных искомых слов;  $k$  — размер алфавита;  $s$  — длина строки, в которой производится поиск;  $n$  — общая длина всех совпадений.

Как видим, задача поиска вхождений набора строк в заданный текст выполняется за линейное время и зависит от нескольких параметров.

Полученный результат может быть улучшен, если для хранения бора будет использована более совершенная структура данных — например, красно-черное дерево.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсовой работы был реализован алгоритм Ахо-Корасик, определены области его применения, разобраны механизмы построения и использования префиксного дерева и конечного детерминированного автомата, была произведена оценка вычислительной сложности алгоритма .

## СПИСОК ЛИТЕРАТУРЫ

1. Кнут Д. Э. Искусство программирования. Т.3. Сортировка и поиск. — 2-е изд. — М.: Вильямс, 2007. — Т. 3. — 832 с.
2. Ахо А. В., Хопкрофт Дж. Э., Ульман Дж. Д. Структуры данных и алгоритмы — М.: Вильямс, 2003. — 384 с.
3. Meyer B. Incremental string matching // Information Processing Letters. — 1985. — №21. — с. 219 - 227.