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. 3331506/20401 Осипенко С.Ю.

Преподаватель Ананьевский М.С.

« » 2025 г.

Санкт-Петербург 2025 г.

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_bookmark0)

1. [ОПИСАНИЕ АЛГОРИТМА 4](#_bookmark1)
   1. [Построение бора 4](#_bookmark2)
   2. [Построение конечного детерминированного автомата 5](#_bookmark3)
   3. [Добавление суффиксных ссылок 5](#_bookmark4)
   4. [Краткая последовательность действий 6](#_bookmark5)
2. [РЕАЛИЗАЦИЯ АЛГОРИТМА 8](#_bookmark6)
3. [АНАЛИЗ АЛГОРИТМА 13](#_bookmark7)

[ЗАКЛЮЧЕНИЕ 14](#_bookmark8)

[СПИСОК ЛИТЕРАТУРЫ 15](#_bookmark9)

# 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](http://en.wikipedia.org/wiki/Alfred_V._Aho) 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:

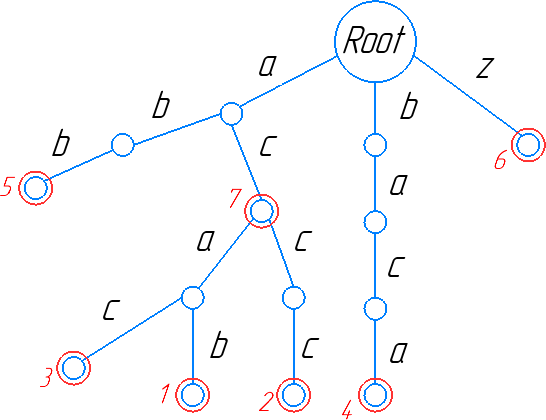
1. реализация алгоритма на языке C++;
2. анализ эффективности и сложности алгоритма.

# 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.

## Построение бора

Бор или префиксное дерево — [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), позволяющая хранить [ассоциативный массив](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), ключами которого являются строки. Представляет собой [корневое дерево](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D0%BD%D0%B5%D0%B2%D0%BE%D0%B9_%D0%B3%D1%80%D0%B0%D1%84#%D0%9A%D0%BE%D1%80%D0%BD%D0%B5%D0%B2%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), каждое ребро которого помечено некоторым символом. В отличие от [бинарных деревьев поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0), 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)

*accc*, 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).

## 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.

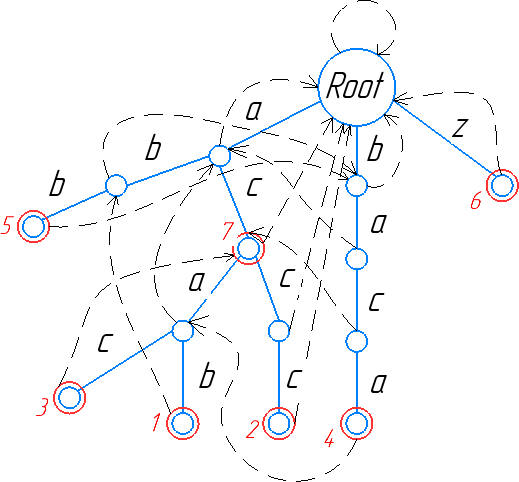
«Детерминированный» — обозначает, что автомат в каждый момент времени может находиться только в одном состоянии.

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.

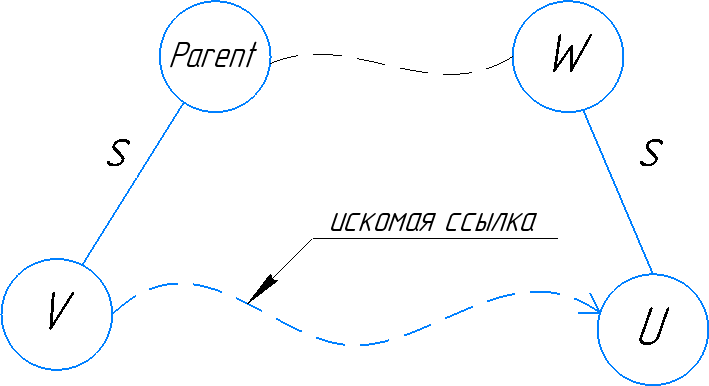
## Добавление суффиксных ссылок

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.

## 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.

# 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 рассчета 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 суф. 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("dcbcddbbbcccbbbcccbbabc");

}

Результат работы программы:



*Рисунок 4 — Результат выполнения программы*

# АНАЛИЗ АЛГОРИТМА

Данный вариант алгоритма проходит циклом по длине заданной строки *s* (𝑁 = 𝑠. 𝑙𝑒𝑛𝑔𝑡ℎ()), откуда его уже можно оценить как 𝑂(𝑁 ∙ 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.](https://ru.wikipedia.org/wiki/%D0%9A%D0%BD%D1%83%D1%82%2C_%D0%94%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%B4_%D0%AD%D1%80%D0%B2%D0%B8%D0%BD) [Искусство программирования. Т.3. Сортировка и поиск](https://books.google.ru/books?id=92rW-nktlbgC&printsec=frontcover&dq=editions%3AspjjKVwoQ3QC&hl=ru&sa=X&ved=0ahUKEwiUjLbc88rQAhVCfiwKHUJlBQoQuwUIIjAB%23v%3Donepage&q&f=false). — 2-е изд. — М.: Вильямс, 2007. — Т. 3. — 832 с.
2. [Ахо А. В.](https://ru.wikipedia.org/wiki/%D0%90%D1%85%D0%BE%2C_%D0%90%D0%BB%D1%8C%D1%84%D1%80%D0%B5%D0%B4), [Хопкрофт Дж. Э.](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%BF%D0%BA%D1%80%D0%BE%D1%84%D1%82%2C_%D0%94%D0%B6%D0%BE%D0%BD), [Ульман Дж. Д.](https://ru.wikipedia.org/wiki/%D0%A3%D0%BB%D1%8C%D0%BC%D0%B0%D0%BD%2C_%D0%94%D0%B6%D0%B5%D1%84%D1%84%D1%80%D0%B8) Структуры данных и алгоритмы — М.: Вильямс, 2003. — 384 с.
3. [Meyer B](https://en.wikipedia.org/wiki/Bertrand_Meyer)*.* [Incremental string matching](http://se.ethz.ch/~meyer/publications/string/string_matching.pdf) *//* Information Processing Letters*. —*

1985. — №21. — с. 219 - 227.